CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLGY
TAUGHT BY HÅKAN NILSSON

Project work:

Projection of a mesh on a .stl surface

Developed for OpenFOAM-1.6.ext

Author:
Christoffer Järpner

Peer reviewed by:
Martin de Maré
Qingming Liu
Ehsan Yasari

October 16, 2011

Contents

1 Introduction

SnappyHexMesh is a utility in the opensource software OpenFOAM, the utility or tool works in such a way that provided a mesh, and a surface. *SnappyHexMesh* will cut through the mesh where the mesh and the surface intersect and then redo the mesh so that the mesh is following the surface. As a tool it is very quick and effective, probably one of the faster ways to generate a complex mesh provided by any tool or software. However since *snappyHexMesh* cuts through the surface and generates its own mesh, the mesh isn’t always very nice, especially in the boundary layer regions, where for correct solutions a mesh needs to be very fine. This is especially true for a very complex geometry like the geometrical topology of the real world ground. See the master thesis [1] for an analysis of how *snappyHexMesh* compares to commercial mesh generation software. As shown in the report the main problem of *snappyHexMesh* is the boundary layer generation, and the fact that it might work well on simple surfaces, but on more complex surfaces the boundary layer might not even be created in certain points or similarly bad properties.

How is a fine mesh generated for a complex topology? *snappyHexMesh* has an amazing potential in its way to snap the mesh points to a surface. This tutorial is going to describe how to modify *snappyHexMesh* to snap an already generated mesh to a surface, without cutting in the mesh.

First of all important, already existing codes will be explained, with a more in-depth explanation of the parts that’s important for the modifications done in the later part.

2 Theoretical background: the Basics of snappyHexMesh

Since *snappyHexMesh* is the base of *mysnappyHexMesh*, this chapter will go through the basics of how *snappyHexMesh* works.

As previously mentioned *snappyHexMesh* is a utility in OpenFOAM, the code can be found in `$WM_PROJECT_DIR/applications/utilities/mesh/generation/snappyHexMesh`. *snappyHexMesh* uses several different libraries, but there are three main libraries or "programs" called:

- *autoRefineDriver*
- *autoSnapDriver*
- *autoLayerDriver*

These three "programs" can be found in the folder *`$WM_PROJECT_DIR/src/autoMesh/autoHexMesh/autoHexMeshDriver`*/ and are responsible for the cutting and refining of the mesh (*autoRefineDriver*), the snapping of the mesh on to the surface (*autoSnapDriver*) and the creation of boundary layers (*autoLayerDriver*).

For the purpose of creating a projection of a surface on to the grid, the most important part of *snappyHexMesh* is the *autoSnapDriver* which will be discussed further in chapter 2.3. To understand what *snappyHexMesh* does, some basic understanding of how *autoRefineDriver* operates is also good; therefore *autoRefineDriver* will be discussed to some extent in chapter 2.2. Finally the "user interface" of the *snappyHexMesh* utility, its dictionary where the user can control what shall happen will briefly be mentioned in chapter 2.1.

2.1 snappyHexMeshDict

A part of *snappyHexMeshDict* can be seen below; the important part of this code is the three lines at the bottom, *castellatedMesh*, *snap* and *addLayers*. These lines activates the three utilities, if *castellatedMesh* is set as true it activates the *autoRefineDriver,* and if *snap* is set to true it activates the *autoSnapDriver,* and if *addLayers* is set to true it activates the *autoLayerDriver.*

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      snappyHexMeshDict;
}
castellatedMesh true;
snap            false;
addLayers       false;
```

snappyHexMeshDict has a lot of so called sub-dictionaries, the sub-dictionaries can be seen below.

```
geometry
{
}

castellatedMeshControls
{
}

snapControls
{
}

addLayersControls
{
}

meshQualityControls
{
}
```

In each of these sub-dictionaries the different utilities of *snappyHexMesh* can be controlled, the sub-dictionaries *geometry* and *meshQualityControls* are general settings that will affect the outcome of all the utilities in *snappyHexMesh.* In the *geometry* sub-dictionary the user needs to specify which surface that *snappyHexMesh* should cut, and snap to, this usually looks something like the code below:

```
geometry
{
    AcrossRiver.stl
    {
        type triSurfaceMesh;
        name AcrossRiver;
    }

    refinementBox
    {
        type searchableBox;
        min (659531  4.7513e+06   1028);
        max (662381  4.75454e+06  1200);
    }
};
```

Where *AcrossRiver.stl* is the surface file that's going to be used, and *triSurfaceMesh* defines what kind of surface it is, in this case a surface geometry that's built of a multitude of triangular surfaces, which together creates an advanced geometry.

In the *meshQualityControls* the user have a multitude of options that's going to effect the end-result of how the final mesh is going to look, for example there is a option called *maxNonOrtho* which describes how much a hexagon cell are allowed to be non-orthogonal, i.e. how skewed it is allowed to become. So if the mesh quality of the generated mesh is not satisfactory, the *meshQualityControls* is a good place to start messing around with.

The other sub-dictionaries are there to control the outcome of a specific utility, for example the options in *snapControls* determines how the snapping of the mesh to the surface is going to be handled, these options will be explained further in the chapters 2.3.1 – 2.3.4.

2.2 autoRefineDriver

AutoRefineDriver is called in the *snappyHexMeshDict* dictionary, located in the system folder of the case, by selecting the option "*castellatedMesh*" as true (*snappyHexMeshDict* is described in the previous chapter, 2.1). The program will go through the mesh cells and see if the mesh intersects with the surface specified. Every time it does, it will split the cell and put it in a specific boundary. The boundary will be named specifically after the surface that the user specified in their *snappyHexMeshDict*. For example if the user wishes to snap to the surface "*AcrossRiver.stl*" the file needs to be placed in the folder $case_folder/constant/trisurface and then called on in snappyHexMeshDict according to what was mentioned in chapter 2.1. With this done the boundary will be called AcrossRiver_patch0, where *AcrossRiver* comes from the surface file name, and *patch0* comes from the first line in the surface file, seen in the code below.

```
solid patch0
  facet normal -0.0331497 -0.099449 0.99449
    outer loop
```

This is important because when projecting something the *autoRefineDriver* won't be used but the *autoSnapDriver* must still recognize which boundary that should be projected.

autoRefineDriver will then continue to refine the boundary just created to a certain point specified in the *snappyHexMeshDict*. When *autoRefineDriver* is done, a mesh similar to figure 1 is usually the result.

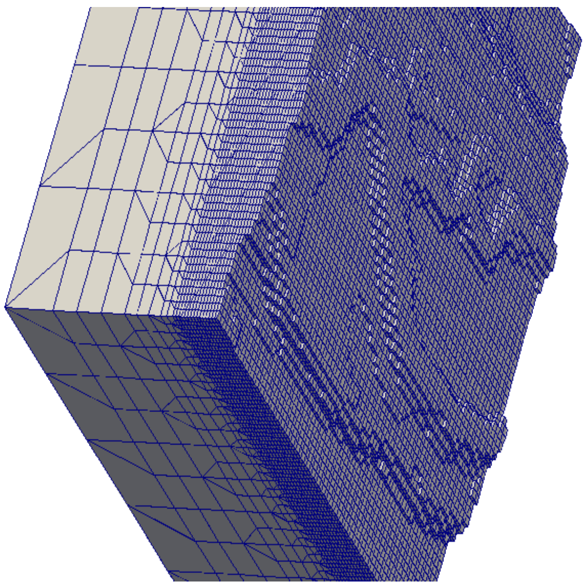
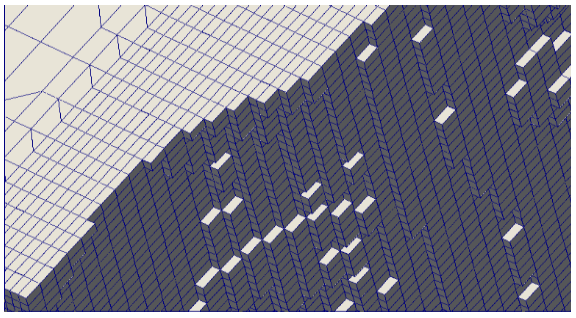

Figure 1, these figures above show how autoRefineDriver works on a mesh.

2.3 autoSnapDriver

Similarly to *autoRefineDriver*, *autoSnapDriver* is called by selecting the "*snap*" option as true in *snappyHexMeshDict*. *AutoSnapDriver*, as the name implies, snaps the mesh boundary cut by *autoRefineDriver* to the surface. *AutoSnapDriver* is very complicated, but very simplified it starts off by:

1. Calculating the mesh-points location, and the cell-centers of the surface.
2. After that the driver goes through every mesh-point that's going to be moved and checks the distance to the closest surface cell-center.
3. The mesh will then be moved that distance so the mesh and the surface connects.
4. The program will then check so that no couple of mesh-points connects to the same surface cell.

In point 2 the word closest is underlined, this is because it's a very important thing to keep in mind with *snappyHexMesh*. It tries to snap to the closest surface point, and as such there is a rather big risk that one point will have several cells connected to it. This is why point 4 is so important, it moves the cells which share surface point, and spreads it out. When *autoSnapDriver* is done, a mesh similar to figure 2 is usually the result.

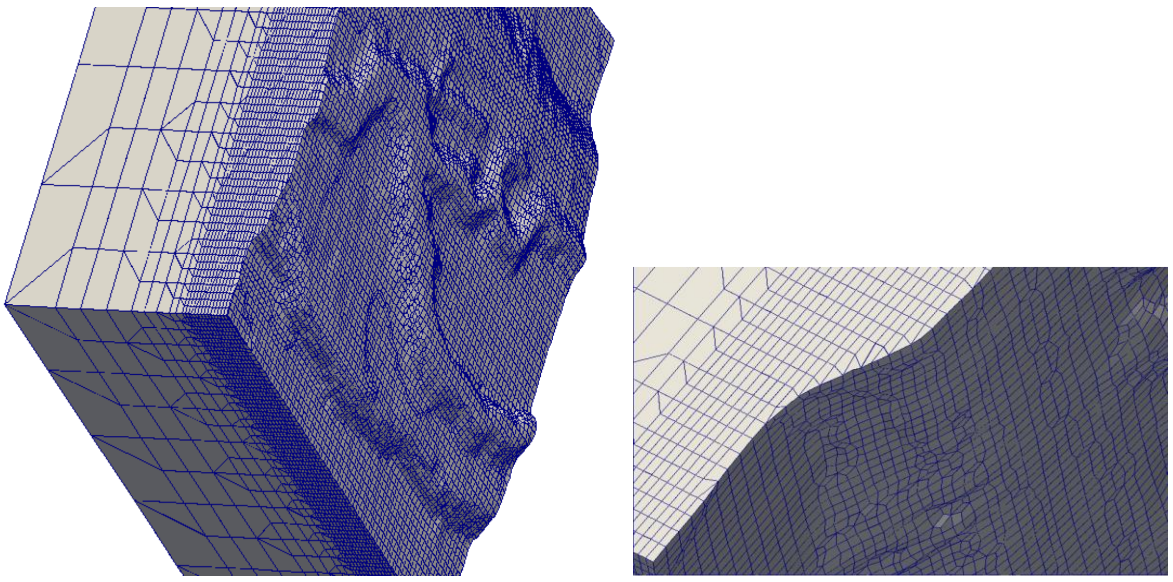

Figure 2, these figures above show how autoSnapDriver works on a mesh.

In *snappyHexMeshDict* there are several options in controlling how this driver is going to work, how many iterations it's going to do (to get the best possible mesh) and so on, the options can be found in "*snapControls*" and are;

1. nSmoothPatch ,
2. tolerance,

3. nSolveIter,

4. nRelaxIter.

What do these options do?
Digging through the code of *snappyHexMesh.C* and *autoSnapDriver.C* all of these data's can be found and by changing the values, the way *snappyHexMesh* operates can be changed.

2.3.1 nSmoothPatch

nSmoothPatch can be found in the *autoSnapDriver*.C. Below the code where *nSmoothPatch* is used can be seen. The code shows that *nSmoothPatch* is the upper limit for the for-loop, i.e. the higher the value the more times this loops is going to be run.

```
    for(label smoothIter = 0; smoothIter <
snapParams.nSmoothPatch(); smoothIter++)
    {
        Info<< "Smoothing iteration " << smoothIter << endl;
        checkFaces.setSize(mesh.nFaces());
        forAll(checkFaces, faceI)
        {
            checkFaces[faceI] = faceI;
        }

        pointField patchDisp(smoothPatchDisplacement(meshMover,
baffles));

        // The current mesh is the starting mesh to smooth from.
        meshMover.setDisplacement(patchDisp);
        meshMover.correct();

        scalar oldErrorReduction = -1;

        for (label snapIter = 0; snapIter < 2*snapParams.nSnap();
snapIter++)
        {
            Info<< nl << "Scaling iteration " << snapIter << endl;

            if (snapIter == snapParams.nSnap())
            {
                Info<< "Displacement scaling for error reduction set to
0."
                    << endl;
                oldErrorReduction = meshMover.setErrorReduction(0.0);
            }

            // Try to adapt mesh to obtain displacement by smoothly
            // decreasing displacement at error locations.
            if (meshMover.scaleMesh(checkFaces, baffles, true,
nInitErrors))
            {
                Info<< "Successfully moved mesh" << endl;
                break;
            }
        }

        if (oldErrorReduction >= 0)
```

```
            {
                meshMover.setErrorReduction(oldErrorReduction);
            }
            Info<< endl;
        }
```

In the loop, the program is trying to smooth the external mesh i.e. the higher the value the *nSmoothPatch*, the smoother the projected surface will become, this can, if one looks closely, be seen in figure 3 and 4, where figure 3 shows how a mesh could look if *nsmoothPatch* is 1, and figure 4 shows how it could look with *nSmoothPatch*=6.

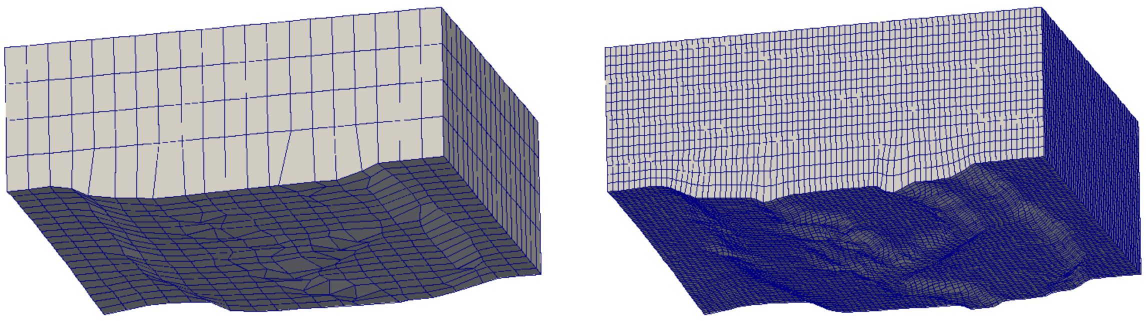

Figure 3, nSmootPatch=1, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

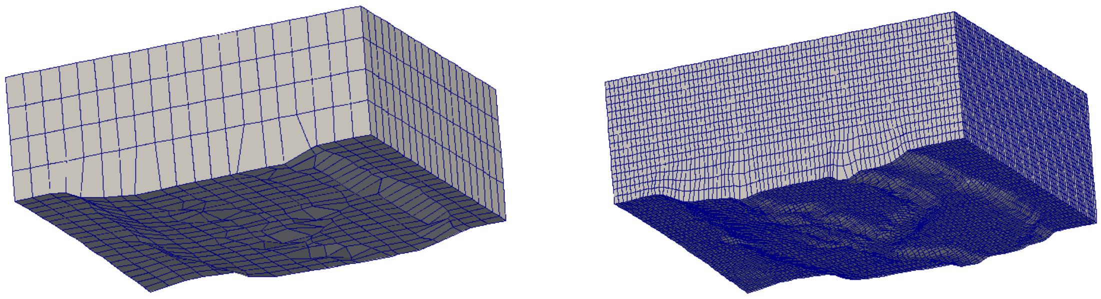

Figure 4, nSmootPatch=6, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

Looking at the figures one can see that figure 3 has a bit sharper edges when it comes to the projected surface.

2.3.2 Tolerance

In the location
`/opt/openfoam170/src/autoMesh/autoHexMesh/autoHexMeshDriver/snapParameters/`
a file called *snapParameters.C* can be found, this piece of code is there to read the *snapControls* subdictionary that is specified in *snappyHexMeshDict*. The *snapParameters* file reads the dictionary and saves the four options, the option *tolerance*, is in *snapParameters* saved as *snapTol*. *snapTol* can be found in the *autoSnapDriver*.C. Below you can see the code where *snapTol* is used. And as can be seen *snapTol is* used at the end of the code, multiplied by the longest edge of a cell.

```
Foam::scalarField Foam::myautoSnapDriver::calcSnapDistance
(
    const snapParameters& snapParams,
    const indirectPrimitivePatch& pp
) const
{
    const edgeList& edges = pp.edges();
    const labelListList& pointEdges = pp.pointEdges();
    const pointField& localPoints = pp.localPoints();
    const fvMesh& mesh = meshRefiner_.mesh();

    scalarField maxEdgeLen(localPoints.size(), -GREAT);

    forAll(pointEdges, pointI)
    {
        const labelList& pEdges = pointEdges[pointI];

        forAll(pEdges, pEdgeI)
        {
            const edge& e = edges[pEdges[pEdgeI]];

            scalar len = e.mag(localPoints);

            maxEdgeLen[pointI] = max(maxEdgeLen[pointI], len);
        }
    }

    syncTools::syncPointList
    (
        mesh,
        pp.meshPoints(),
        maxEdgeLen,
        maxEqOp<scalar>(),  // combine op
        -GREAT,             // null value
        false               // no separation
    );

    return snapParams.snapTol()*maxEdgeLen;
}
```

This option changes how long distance the program should look for a point to snap, the distance will be the number put in "tolerance"*"length of the longest edge in the cell". This option will be very important in *mysnappyHexMesh,* it determines how fine the initial mesh can be. In figure 5 and 6 a mesh can be seen where in figure 5 the value of tolerance is set to 1, and figure 6 shows how it could look with tolerance=10.

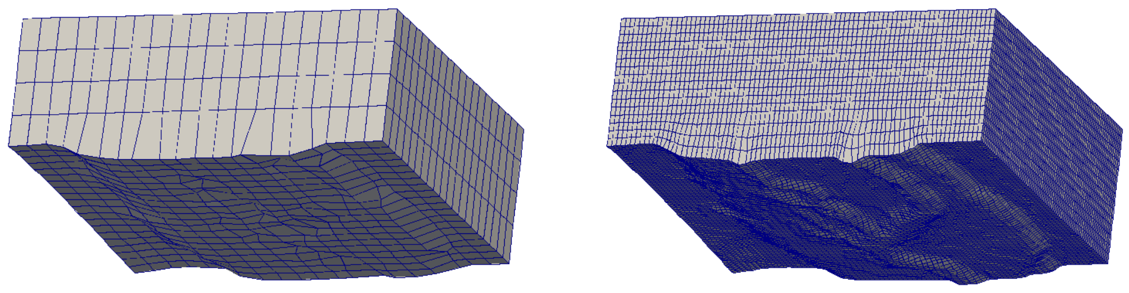

Figure 5, tolerance=1, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

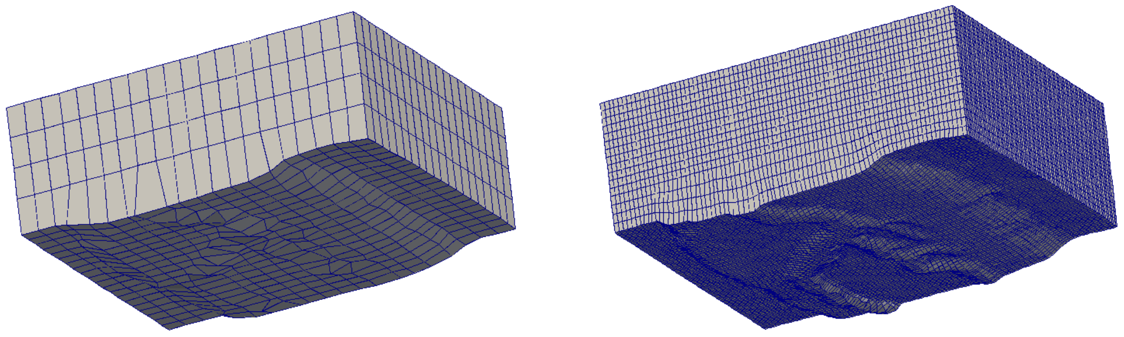

Figure 6, tolerance=10, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

Looking at the figures one can see at the coarse mesh, when the tolerance is set to 10, follows the surface better than when the tolerance is set to 1.

2.3.3 nSolveIter

In the location `/opt/openfoam170/src/autoMesh/autoHexMesh/autoHexMeshDriver/snapParameters`/ a file called *snapParameters.C* can be found, this piece of code is there to read the *snapControls* subdictionary that is specified in *snappyHexMeshDict*. The *snapParameters* file reads the dictionary and saves the four options, the option *nSolveIter,* is in *snapParameters* saved as *nSmoothDispl*. *nSmoothDispl* can be found in the *autoSnapDriver*.C. Below you can see the code where *nSmoothDispl* is used. As can be seen *nSmoothDispl* is used as the upper limit for a for-loop.

from autoSnapDriver
```
for (label iter = 0; iter < snapParams.nSmoothDispl(); iter++)
    {
        if ((iter % 10) == 0)
        {
            Info<< "Iteration " << iter << endl;
        }
```

```
            pointVectorField oldDisp(disp);

            meshMover.smooth(oldDisp, edgeGamma, false, disp);
        }
```

This option changes how many times the "*snapping*" part of *snappyHexMesh* should be run, i.e. the higher this number is the better mesh quality will be gained, i.e. the more equidistant mesh will be created when it comes to the boundary, but also the longer the snapping will take. For mysnappyHexMesh this value will mostly be important when the surface have a big difference in height. In figure 7 and 8 a mesh can be seen where in figure 7 the value of nSolveIter is set to 1, and figure 8 shows how it could look with nSolverIter=100.

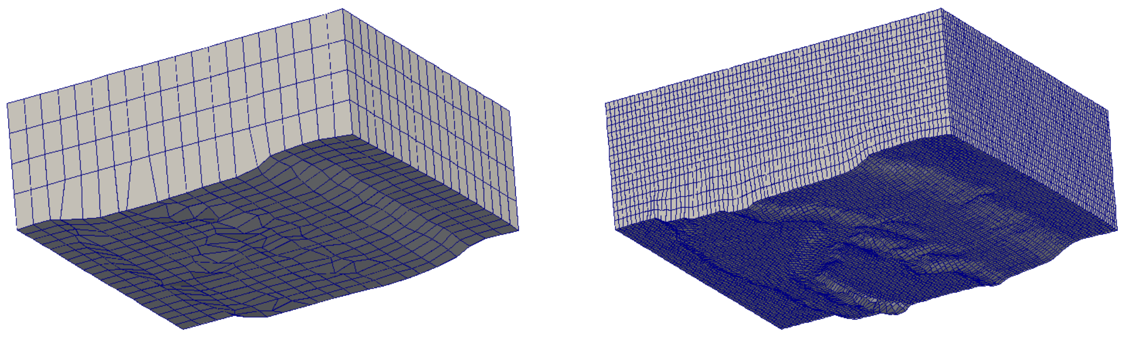

Figure 7, nSolveIter=1, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

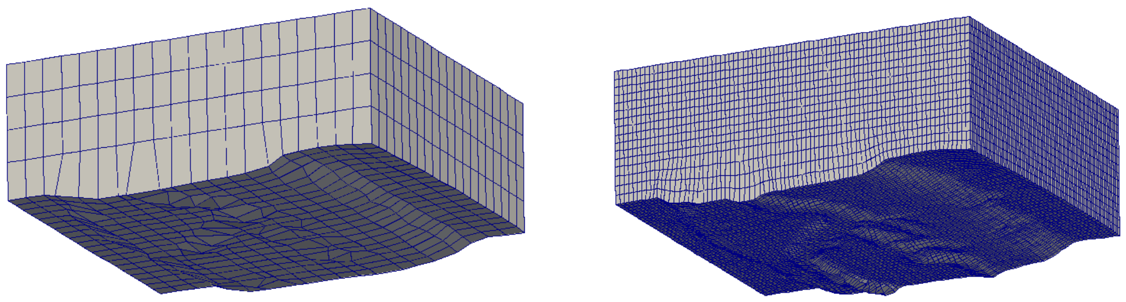

Figure 8, nSolveIter=100, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

As can be seen there is no difference between the two figures, this is because the surface is rather flat and the option would have no influence.

2.3.4 nRelaxIter

In the location
`/opt/openfoam170/src/autoMesh/autoHexMesh/autoHexMeshDriver/snapParameters/`
a file called *snapParameters.C* can be found, this piece of code is there to read the *snapControls* subdictionary that is specified in *snappyHexMeshDict*. The *snapParameters* file reads the dictionary and saves the four options, the option *nRelaxIter,* is in *snapParameters* saved as *nSnap*. *nSnap* can be found in the *autoSnapDriver*.C Below you can see the code where *nSnap* is used. As can be seen *nSnap* is used as the upper limit for a for-loop.

from autoSnapDriver

```
Info<< "Moving mesh ..." << endl;
    for (label iter = 0; iter < 2*snapParams.nSnap(); iter++)
    {
        Info<< nl << "Iteration " << iter << endl;

        if (iter == snapParams.nSnap())
        {
            Info<< "Displacement scaling for error reduction set to 0."
<< endl;
            oldErrorReduction = meshMover.setErrorReduction(0.0);
        }

        if (meshMover.scaleMesh(checkFaces, baffles, true,
nInitErrors))
        {
            Info<< "Successfully moved mesh" << endl;

            break;
        }
        if (debug)
        {
            const_cast<Time&>(mesh.time())++;
            Pout<< "Writing scaled mesh to time " <<
meshRefiner_.timeName()
                << endl;
            mesh.write();

            Pout<< "Writing displacement field ..." << endl;
            meshMover.displacement().write();
            tmp<pointScalarField>
magDisp(mag(meshMover.displacement()));
            magDisp().write();
        }
    }
```

This option changes how many times the mesh will run a relaxing script that removes some bad mesh points; however *snappyHexMesh* should stop before, when the correct mesh is created. In figure 9 and 10 a mesh can be seen where in figure 9 the value where *nRelaxIter* is set to 1, and figure 10 shows how it could look with *nRelaxIter* =10.

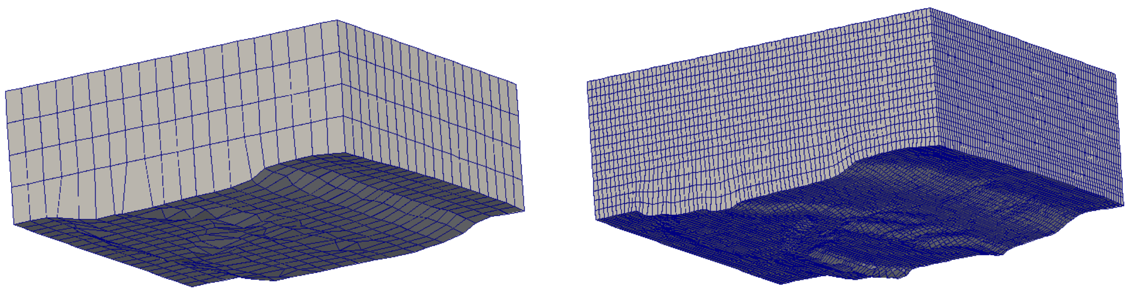

Figure 9, nRelaxIter=1, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

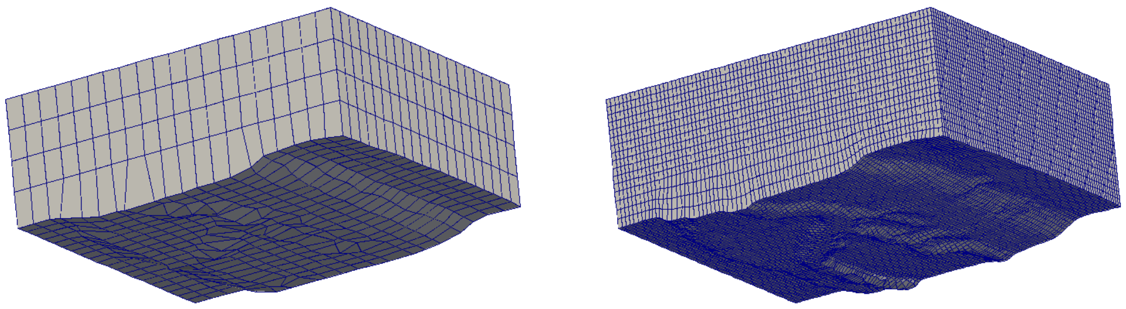

Figure 10, nRelaxIter=10, to the left a coarse mesh can be seen whereas the right figure shows a much finer mesh.

It might be possible to see in the figures that the option has little to no impact on the coarser mesh, whereas a small difference can be seen between the two finer meshes.

3 Theoretical background: the Basics of moveDynamicMesh

moveDynamicMesh is used in *mysnappyHexMesh* to move the internal boundary field, because it is used, a brief description of what *moveDynamicMesh* is and how it can be used follows in this chapter.

moveDynamicMesh is a mesh manipulation utility, just as *snappyHexMesh* and the code is located in the following directory:

`$WM_PROJECT_DIR/applications/utilities/mesh/manipulation/moveDynamicMesh`.

moveDynamicMesh is a very big and wide spread utility that has several functionalities. For example it is used in all sorts of simulations where the mesh or the boundary is moving, and can also be used to manipulate a static mesh, because of the big range of how *moveDynamicMesh* is used. This project will only briefly mention the capability which this utility has, but puts a higher focus on the *dynamicMotionSolverFvMesh* and the library *libfvMotionSolvers.so* that goes with it, which later on is used to modify the internal mesh of the projection.

moveDynamicMesh can be split up in to two different mesh manipulation categories the automatic mesh motions (*dynamicFvMesh*) and the topological mesh changes (*topoChangerFvMesh*) which will be explained further in chapter 3.1.

Each of these utilities can be solved in different manners. There are three solvers for this which is called *displacementLaplacian*, *velocityLaplacian* and *SBRStress* more on these can be found in chapter 3.2.

When using *moveDyanamicMesh* one also has to specify the *diffusivity* of the mesh, e.g. how easy or hard it is to move the internal mesh points, the different options available is described in chapter 3.3.

For more information about *dynamicMeshDict* and the different options [2] is a very good source.

3.1 The different classes used in moveDynamicMesh

In general one could say that for time varying, and big mesh changes, such as a moving object through the mesh or a rotating mesh of the kind that's used in for example turbomachinery, the *libDynamicTopoChanger* should be used. This library links to classes such as *linearValveFvMesh*, *linearValveLayersFvMesh*, *mixerFvMesh* and *movingConeTopoFvMesh*. The functionality of these classes can be described as:

1. *linearValveFvMesh*, which can be found in the folder `$WM_PROJECT_DIR/src/topoChangerFvMesh/linearValveFvMesh/` should for example be used if an object is moving linearly compared to another piece of mesh. The program works in such a way that 2 or more boundaries are defined, and as the time changed the boundaries are decoupled and can move freely. As the calculations starts again, the mesh is reattached to the new position the object has.
2. *linearValveLayersFvMesh*, which can be found in the folder `$WM_PROJECT_DIR/src/topoChangerFvMesh/linearValveLayersFvMesh/` is extremely similar to the above, however layer addition and removal is also included. This is very good for cases where you have a big deformation of the mesh, such as a piston in a diesel engine.
3. *mixerFvMesh*, which can be found in the folder `$WM_PROJECT_DIR/src/topoChangerFvMesh/mixerFvMesh/` is used for rotation of meshes and is specifically used in turbo machinery calculations. Similarly to the other mesh methods, when time is changing the mesh boundaries will be decoupled and then reattached when calculations are started.

For smaller mesh displacements, *DynamicFvMesh* should be used. This contains mainly three classes, which are *staticFvMesh*, *dynamicMotionSolverFvMesh* and *dynamicInkJetFvMesh*.

1. *staticFvMesh*, which can be found in the folder `/opt/openfoam170/src/dynamicFvMesh/staticFvMesh` is pretty much as it sounds, static, using this in the dictionary when running *moveDynamicMesh*, and nothing will happen.
2. *dynamicMotionSolverFvMesh*, which can be found in the folder `/opt/openfoam170/src/dynamicFvMesh/dynamicMotionSolverFvMesh/` is the main class for this library. For small changes or movements in the boundary layer, this will move the internal mesh points according to how the user specifies it should behave.
3. *dynamicInkJetFvMesh*, which can be found in the folder `/opt/openfoam170/src/dynamicFvMesh /dynamicInkJetFvMesh` modifies the mesh based on harmonic motions around a user-defined reference plane.

When using *mysnappyHexMesh dynamicMotionSolverFvMesh* should be used.

3.2 The different solvers used in moveDynamicMesh

As said before there are three solvers in moveDynamicMesh the solvers solve the mesh motion as if the mesh would have been a solid, using FEM and the solvers are called *displacementLaplacian*, *velocityLaplacian* and *SBRStress.*

1. *displacementLaplacian* solves the mesh motion as a displacement in mesh points, which for each timestep is moved to different points, and a prescribed boundary displacement is needed. The code for this solver can be found in the location `/opt/openfoam170/src/fvMotionSolver/fvMotionSolvers/displacement/laplacian/`

2. *velocityLaplacian* solves the mesh motion as a velocity in the mesh points, and therefore a prescribed velocity is needed on the boundary. The code for this solver can be found in the location`/opt/openfoam170/src/fvMotionSolver/fvMotionSolvers/velocity/laplacian/`

3. *SBRStress* works like *displacementLaplacian*, but also solves for rotation. The code for this solver can be found in the location `/opt/openfoam170/src/fvMotionSolver/fvMotionSolvers/displacement/SBRStress`

3.3 The diffusivity used in moveDynamicMesh

In the *dynamicMeshDict* which can be found in the constant directory the diffusivity is also described, the diffusivity can be explained similarly to a springs “k” constant, the smaller diffusivity the easier it will be to move that point. In the diffusivity there are several different ways to define how the diffusiveness is changed along the mesh, for example diffusivity uniform; will have the diffusivity equal along the mesh and the mesh will then be deformed equally along the entire mesh. This is compared to for example diffusivity InverseDistance 1(maxZ); which will decrease the diffusivity the further away from the boundary specified (in this case maxZ). The latter option should be used, when using *moveDynamicMesh* in *mysnappyHexMesh*.

3.4 dynamicMeshDict

A typical *dynamicMeshDict* could look like the code below:

```
FoamFile
{
    version         2.0;
    format          ascii;
    class           dictionary;
    object          dynamicMeshDict;
}
dynamicFvMesh dynamicMotionSolverFvMesh;
motionSolverLibs ("libfvMotionSolvers.so");
solver velocityLaplacian;
diffusivity  quadratic inverseDistance 1 (maxZ);
```

4 Tutorial of how to implement mysnappyHexMesh

Below a short tutorial of how to change *snappyHexMesh.C* in to *mysnappyHexMesh.C* will be described.

4.1 Creation of mysnappyHexMesh

MysnappyHexMesh is based on four different utilities, these four are

- snappyHexMesh.C
- refineMesh.C
- moveDynamicMesh.C
- refineWallLayer.C

Start by copying all these utilities to OpenFoams user folder by typing these lines:

```
cp -r $WM_PROJECT_DIR/applications/utilities/mesh/generation/snappyHexMesh/ \
$WM_PROJECT_USER_DIR/.
```

```
cp -r $WM_PROJECT_DIR/applications/utilities/mesh/manipulation/refineMesh/ \
$WM_PROJECT_USER_DIR/snappyHexMesh/.
```

```
cp -r $WM_PROJECT_DIR/applications/utilities/mesh/manipulation/moveDynamicMesh/ \
$WM_PROJECT_USER_DIR/snappyHexMesh/.
```

```
cp -r $WM_PROJECT_DIR/applications/utilities/mesh/advanced/refineWallLayer/ \
$WM_PROJECT_USER_DIR/snappyHexMesh/.
```

Enter the user folder by typing:

```
cd $WM_PROJECT_USER_DIR
```

Rename the *snappyHexMesh* folder just copied to *mysnappyHexMesh* and enter it by writing:

```
mv snappyHexMesh mysnappyHexMesh
```

```
cd mysnappyHexMesh
```

Rename snappyHexMesh.C to mysnappyHexMesh.C by typing:

```
mv snappyHexMesh.C mysnappyHexMesh.C
```

To be able to compile the new program the file Make/files needs to be edited slightly. Do this by typing:

```
vi Make/files
```

The file should look something like this:

```
snappyHexMesh.C
```

```
EXE = $(FOAM_APPBIN)/snappyHexMesh
```

Change it so it looks like this:

```
mysnappyHexMesh.C

EXE = $(FOAM_USER_APPBIN)/mysnappyHexMesh
```

Save and exit and assuming everything have been done correctly it should now be possible to compile the code and run a case for *snappyHexMesh*, using *mysnappyHexMesh* instead.

Compile it by typing:

```
wclean
wmake
```

4.2 Create header files of the other utilities

Move the other utilities from their copied location to *mysnappyHexMesh* folder.

```
mv refineMesh/refineMesh.C refineMesh.H
cp refineMesh.H refineMeshStuff.H
mv refineWallLayer/refineWallLayer.C refineBoundaryWallLayer.H
mv moveDynamicMesh/moveDynamicMesh.C moveDynamicMesh.H
```

To be able to use these utilities in the code, the compiler needs to know where all the important files are located; this information is specified in the *Make/options* file. Therefore all the lines in the different options files for all utilities needs to be incorporated in to *mysnappyHexMesh's* options file.

For *snappyHexMesh.C* the options file (located in *Make/options*) look like this:

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

EXE_LIBS = \
    -lfiniteVolume \
    -ldecompositionMethods \
    -lmeshTools \
    -ldynamicMesh \
    -lautoMesh
```

To make it possible for all the utilities to be run -ldynamicFvMesh \ should be added to this, add this by typing

```
vi Make/options
```

Change it to this:

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


EXE_LIBS = \
    -lfiniteVolume \
    -ldecompositionMethods \
    -lmeshTools \
    -lengine \
    -ldynamicMesh \
    -ldynamicFvMesh \
    -ldynamicMesh \
    -llduSolvers \
    -lautoMesh
```

Save and exit.

Similarly to what has been done to the options file, the include header files that are written at the start of each program also needs to be added.

Start by opening mysnappyHexMesh.C by typing:

```
vi mysnappyHexMesh.C
```

And the code below should look like this:

```
Application
    snappyHexMesh

Description
    Automatic split hex mesher. Refines and snaps to surface.

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

#include "argList.H"
#include "Time.H"
#include "fvMesh.H"
#include "autoRefineDriver.H"
#include "autoSnapDriver.H"
#include "autoLayerDriver.H"
#include "searchableSurfaces.H"
#include "refinementSurfaces.H"
#include "shellSurfaces.H"
#include "decompositionMethod.H"
#include "fvMeshDistribute.H"
#include "wallPolyPatch.H"
#include "refinementParameters.H"
```

Just below this i.e. on line 47, add:

```
#include "dynamicFvMesh.H"
#include "polyMesh.H"
#include "undoableMeshCutter.H"
#include "hexCellLooper.H"
#include "cellSet.H"
#include "twoDPointCorrector.H"
#include "directions.H"
#include "OFstream.H"
#include "multiDirRefinement.H"
#include "labelIOList.H"
#include "wedgePolyPatch.H"
#include "plane.H"
#include "cellCuts.H"
#include "cellSet.H"
#include "meshCutter.H"
#include "directTopoChange.H"
#include "mapPolyMesh.H"
#include "fvCFD.H"
#include "pointFields.H"
#include "Istream.H"
#include "pointMesh.H"
```

Save and exit.

To test so nothing has been changed compile by typing:

```
wclean
wmake
```

All that's been done now is to let the compiler know what files that needs to be read for the final code to work.

4.3 Changes to mysnappyHexMesh.C

Open mysnappyHexMesh.C again, and go down to line 380 and the piece of code below should be seen:

```
    // Now do the real work -refinement -snapping –layers
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Switch wantRefine(meshDict.lookup("castellatedMesh"));
    Switch wantSnap(meshDict.lookup("snap"));
    Switch wantLayers(meshDict.lookup("addLayers"));
```

These lines creates *booleans* (true or false) of what the user has written in *snappyHexMeshDict* (i.e. if the user has put "snap true;" in the dictionary, this will be read in to the code and the *boolean* value "*wantSnap*" will be put to true). In the new code, *mysnappyHexMesh*, more options like this should be added, so, after the line Switch wantLayers(meshDict.lookup("addLayers"));

Add:

```
    Switch wantProjection(meshDict.lookup("projection"));
    Switch wantDynMesh(meshDict.lookup("DynMesh"));
    Switch wantRefineMesh(meshDict.lookup("refineMesh"));
    Switch wantBoundaryLayer(meshDict.lookup("boundaryLayer"));
```

Also add:

```
    if (wantRefineMesh)
    {
            scalar nRefs=readScalar(meshDict.lookup("nRefinements"));
    }
    else
    {
            scalar nRefs=0;
    }

    bool notLast=true;
   if (Testing==nRefs+1)
    {
            Testing=10;
            notLast=false;
    }
```

The last thing added is to enable another option, namely to be able to run the refinement several times.

Around line 437 the code below should exist.

```
  if (wantSnap)
    {
        cpuTime timer;
        autoSnapDriver snapDriver
        (
            meshRefiner,
            globalToPatch
        );

        // Snap parameters
        snapParameters snapParams(snapDict);
        if (!overwrite)
        {
            const_cast<Time&>(mesh.time())++;
        }

        snapDriver.doSnap(snapDict, motionDict, snapParams);

        writeMesh
        (
            "Snapped mesh",
            meshRefiner,
            debug
        );

        Info<< "Mesh snapped in = "
            << timer.cpuTimeIncrement() << " s." << endl;
    }
```

Change all of the above to the lines below:

```
if (wantSnap || wantProjection)
   {
     if (wantProjection)
     {
      pointMesh pMesh(mesh);
      pointVectorField pointMotionU
      (
             IOobject
             (
                   "pointMotionU",
                   runTime.timeName(),
                   mesh,
                   IOobject::MUST_READ,
                   IOobject::AUTO_WRITE
               ),
               pMesh
      );
     }
         cpuTime timer;
         autoSnapDriver snapDriver
         (
               meshRefiner,
               globalToPatch
         );
         // Snap parameters
         snapParameters snapParams(snapDict);
         if (!overwrite)
         {
               const_cast<Time&>(mesh.time())++;
         }
         snapDriver.doSnap(snapDict, motionDict, snapParams);
         writeMesh
         (
               "Snapped mesh",
               meshRefiner,
               debug
         );
         Info<< "Mesh snapped in = "
               << timer.cpuTimeIncrement() << " s." << endl;
   }
   if (wantRefineMesh && notLast)
         {
                 #   include "refineMesh.H"
         }
   if (wantDynMesh && Testing==10)
       {
                 scalar nDynIter=readScalar(meshDict.lookup("nDynIter"));
              #   include "moveDynamicMesh.H"
       }
   if (wantBoundaryLayer && Testing==10)
       {
                 #   include "createWeights.H"
              //splitting the closest row of cells in to nLayer amount of
cells. using the
              //souce code from refineWallLayer.C.
       scalar weight=0;
       for(int i=0;i<nLayers-2;i++)
       {
```

```
                //calculating the weighted place to cut the mesh for this
itteration.
                Info << "weight =" << Li[i] << endl;
                weight=Li[i];
                #   include "refineBoundaryWallLayer.H"
        }
     }
```

By adding this code all the new options are added, i.e. it is now (almost) possible to run the program and all the new utilities can be used. There is however three more things needed to be done.

First since this code is supposed to be able to run several times, a for-loop needs to be added, this for-loop goes around most of the main program, so around line 140, the code below can be seen:

```
int main(int argc, char *argv[])
{
                argList::validOptions.insert("overwrite", "");
#   include "setRootCase.H"
#   include "createTime.H"
    runTime.functionObjects().off();
#   include "createMesh.H"
```

Add here a for-loop so it looks like below:

```
int main(int argc, char *argv[])
{
    for (int Testing=1;Testing<=10;Testing++)
    {
                argList::validOptions.insert("overwrite", "");
#   include "setRootCase.H"
#   include "createTime.H"
    runTime.functionObjects().off();
#   include "createMesh.H"
```

Then at the bottom of the code, at around line 675 the code below should be seen.

```
    Info<< "Finished meshing in = "
        << runTime.elapsedCpuTime() << " s." << endl;

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

    return(0);

}
```

Add a } here to close the for-loop so it looks like this:

```
    Info<< "Finished meshing in = "
        << runTime.elapsedCpuTime() << " s." << endl;
    Info<< "End\n" << endl;
    }
    return(0);

}
```

In the refinement part of the program, *refineMesh.H* is called for, and *refineMesh.H* in turn uses a lot of functions that should be outside of the main function. To add all these functions go to line 108 in *mysnappyHexMesh.C* and add the code below:

```
#include "refineMeshStuff.H"
```

I.e. it should look like this:

```
            << exit(FatalError);
    }
    return mergeDist;
}

#include "refineMeshStuff.H"

// Write mesh and additional information

void writeMesh
(
    const string& msg,
```

Now all the editing in *mysnappyHexMesh.C* is done, let's continue with the .H files.

4.4 Creation of createWeights.H

In the code a file called *createWeights.H* is called for, but this file is not created yet, to do this, write:

```
vi createWeights.H
```

copy the following lines in to it.

```
//reading the dictionary for the layer properties.
scalar nLayers=readScalar(meshDict.lookup("nLayers"));
scalar stretching=readScalar(meshDict.lookup("stretching"));
nLayers++;
// label patchName=readLabel(meshDict.lookup("patch"));
//Calculating the length of the cell that's going to be refined (in weighted
terms).

scalar L[100];
scalar Li[100];

L[0]=0;
L[1]=1;

for(int j=1;j<nLayers-1;j++)
{
            L[j+1]=stretching*(L[j]-L[j-1])+L[j];
}
```

```
int k=0;
for(int i=nLayers-2;i>=1;i--)
{
            Li[k]=1-(L[i+1]-L[i])/L[i+1];
            Info << "weightedthings =" << Li[k] << endl;
            k=k+1;
}
```

4.5 Changes to refineMesh.H

Open *refineMesh.H* by typing:

```
vi refineMesh.H
```

Remove the first 298 lines of code by typing

```
298dd
```

This will remove all the functions and the first lines in the main function, the file should now start like this:

```
#   include "createPolyMesh.H"
    const word oldInstance = mesh.pointsInstance();
```

After the lines shown above, copy

```
            pointMesh pMesh(mesh);
            pointVectorField pointMotionU
            (
                IOobject
             (
                     "pointMotionU",
                     runTime.timeName(),
                     mesh,
                     IOobject::MUST_READ,
                     IOobject::AUTO_WRITE
             ),
                 pMesh
            );
```

After that, go down to the bottom of the code, and remove the last '}'.

And then save and exit.

Open *refineMeshStuff.H* by typing:

```
vi refineMeshStuff.H
```

Go to line 288 by typing

```
:288
```

Delete all the code below, i.e. delete all the code that's in the main function, including the main function itself.

Save and exit.

4.6 Changes to moveDynamicMesh.H

Edit the moveDynamicMesh file by typing:

```
vi moveDynamicMesh.H
```

Remove the 43 first lines by typing:

```
43dd
```

The file will now look like this:

```
#   include "setRootCase.H"
#   include "createTime.H"
#   include "createDynamicFvMesh.H"

    while (runTime.loop())
    {
        Info<< "Time = " << runTime.timeName() << endl;

        if (isDir(runTime.path()/"VTK"))
        {
            Info << "Clear VTK directory" << endl;
            rmDir(runTime.path()/"VTK");
        }

        mesh.update();

#       include "checkVolContinuity.H"

        mesh.checkMesh(true);

        runTime.write();

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

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

    return 0;
}


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

Change the while-loop:

```
    while (runTime.loop())
```

To a for-loop:

```
    for(int i=0;i<nDynIter;i++)
```

Comment or removed the include continuity header file:

```
#       include "checkVolContinuity.H"
```

Too:

```
//#       include "checkVolContinuity.H"
```

And remove the last **}**.

The changes to moveDynamicMesh.H are now done, just Save and Exit.

4.7 Changes to refineBoundaryWallLayer.H

Open *refineBoundaryWallLayer.*H by typing:

```
vi refineBoundaryWallLayer.H
```

Remove the 54 first lines by typing

```
54dd
```

This will make the first lines look like:

```
#   include "setRootCase.H"
#   include "createTime.H"
    runTime.functionObjects().off();
#   include "createPolyMesh.H"
    const word oldInstance = mesh.pointsInstance();

    word patchName(args.additionalArgs()[0]);
    scalar weight(readScalar(IStringStream(args.additionalArgs()[1])()));
    bool overwrite = args.optionFound("overwrite");
```

Change these lines so it says:

```
//reading the dictionary for the layer properties.
    scalar nLayers=readScalar(meshDict.lookup("nLayers"));
nLayers++;

//Calculating the length of the cell that's going to be refined (in weighted
terms).

#   include "setRootCase.H"
#   include "createTime.H"
    runTime.functionObjects().off();
#   include "createPolyMesh.H"
    const word oldInstance = mesh.pointsInstance();

    word patchName
   (
        meshDict.lookup("patch")
    );

    bool overwrite = args.optionFound("overwrite");
```

After that, go down to the bottom of the code, and remove the last '}'.

Save and exit.

With this change the code is complete, and it's just to compile it by typing:

```
wclean
wmake
```

There are now a lot of files that isn’t used by the program, which can therefore be deleted, to do this type:

```
rm -r refineWallLayer
rm -r refineMesh
rm -r moveDynamicMesh
```

5 Explanation of the changes to mysnappyHexMesh

Below the changes to *mysnappyHexMesh* will be explained in further detail.

5.1 mysnappyHexMesh.C

mysnappyHexMesh.C is the base of the new code and is obviously based on *snappyHexMesh.C*. The changes or additions to *snappyHexMesh.C* are rather extensive.

In *mysnappyHexMesh.C* the lines shown below were added;

```
    Switch wantProjection(meshDict.lookup("projection"));
    Switch wantDynMesh(meshDict.lookup("DynMesh"));
    Switch wantRefineMesh(meshDict.lookup("refineMesh"));
    Switch wantBoundaryLayer(meshDict.lookup("boundaryLayer"));
```

The code above makes it so that the program reads the *snappyHexMeshDict* for how to deal with the new options. In the code, a bit further down a bunch of "if's" can be seen, i.e. if (wantRefine), if (wantSnap) and if (wantLayer).

If (wantSnap) was changed to if (wantSnap || wantProjection) this means that the projection option is largely the same as the snap option. However just below this change is another addition, namely

```
if (wantProjection)
      {
            pointMesh pMesh(mesh);
            pointVectorField pointMotionU
            (
                IOobject
                (
                     "pointMotionU",
                     runTime.timeName(),
                     mesh,
                     IOobject::MUST_READ,
                     IOobject::AUTO_WRITE
                 ),
                 pMesh
            );
       }
```

This piece of code shows that if the "*projection*" option is used, a file called *pointMotionU* needs to exist in the 0/ directory. The *pointMotionU* file is needed for the *moveDynamicMesh* to move the internal mesh points and the code just reads the information in the 0/ time directory, and writes it in the next time directories as well.

The if (wantRefineMesh && notLast) is there to activate the *refineMesh.H* code if *refineMesh* is wanted and it is not the last iteration of refinements. More on the *refineMesh.H* code can be found in chapter 5.1.1. The reason *refineMesh* should not be used in the last iteration is due to how the code is placed (below wantProjection). If this restriction was not there *refineMesh* would always be used after a snap, which isn't really ideal for the last iteration, because the mesh would then be very fine, but might not follow the surface to the degree it should be able too.

The code inside *moveDynamicMesh.H* is explained further in chapter 5.1.2, the *calculateWeights.H* code is explained in chapter 5.1.3 and the *refineBoundaryWallLayer* code is explained in chapter 5.1.4.

5.1.1 refineMesh.H

RefineMesh.H is pretty much the same as *refineMesh.C,* except the initial lines are removed. *refineMesh.H*, just as *refineMesh.C* refines the mesh, if a *refineMeshDict* is present in the system folder the user can choose to refine in only one direction, but base-line all directions are refined equally (all hexagons are split in the middle, i.e. 1 hexagon becomes 8 hexagons) the *refineMesh.H* file can be seen in Appendix 1 and a figure of how *refineMesh.C* works can be seen in figure 11.

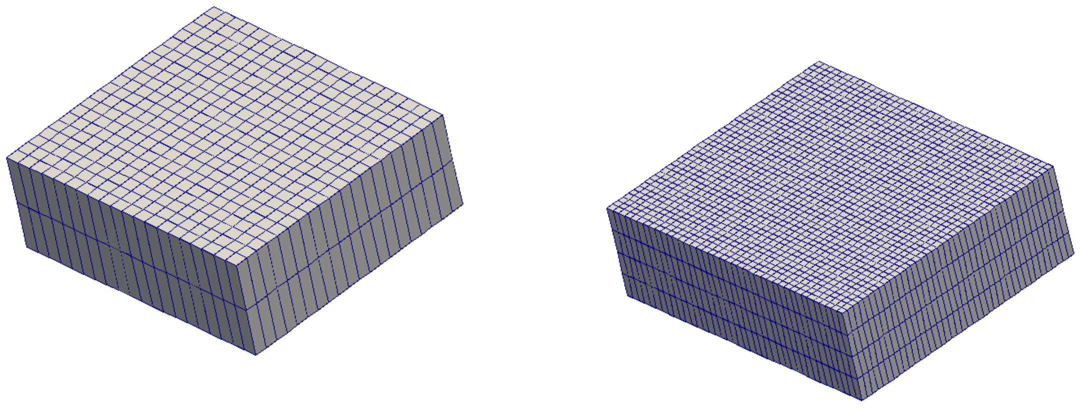

Figure 11, the figure to the left shows a coarse mesh whereas the right figure shows how the coarse mesh would look after refineMesh.C is used.

refineMesh.C can be found in the folder `$WM_PROJECT_DIR/applications/utilities/mesh/manipulation/refineMesh` whereas the *refineMesh.H* should be placed in the *mysnappyHexMesh* folder.

5.1.2 moveDynamicMesh.H

moveDynamicMesh.H utilizes the same code as *moveDynamicMesh.C* except the .C file has a while-loop that breaks on time specified in the *controlDict*, and the .H file has a for-loop that runs specified on the nDynIter value specified in *snappyHexMeshDict*; the code for the .H file can be seen in Appendix 2.

moveDynamicMesh.C can be found in the folder `$WM_PROJECT_DIR/ applications/ utilities/ mesh/ manipulation/ moveDynamicMesh` whereas the *moveDynamicMesh.H* should be placed in the *mysnappyHexMesh* folder.

5.1.3 createWeights.H

refineWallLayer.C needs two input parameters, the boundary on which to create a new layer and the weighting of where the new layer will be created. The weighting factor should be a value between 0 and 1 and defines how much of the first cell that should be cut i.e. if the current cells should be split in half, one would enter 0.5, the Figure 12 below shows how a square block is refined depending on the weighting value added.

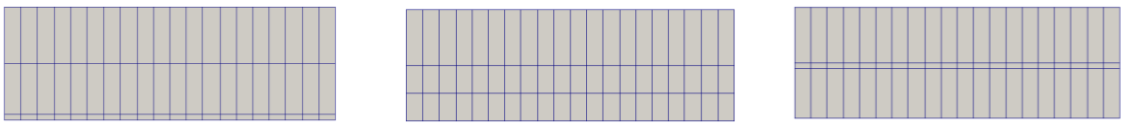
Figure 12, these figures above show how refineWallLayer.C works on a mesh when the weighting factor is set to 0.1 0.5 and 0.9 respectively.

To take the code from *refineWallLayer.C* to create a full on boundary layer, the weighting factor needs to be calculated for several points and then refined one by one. This is done by two for-loops, one which calculates the distance for each new cell, assuming the first cell height is 1, and then the other for-loop calculates the weighting factor for each new layer. This code needs two input parameters, the number of layers that should be created (*nLayers*) and how much each cell is allowed to grow (*shearing*), so with a shearing factor of 1, the layer closest to the snapped surface will be divided in to *nLayers* of equal sized cells; Below the code for the *createWeights.H* can be seen.

```
            //reading the dictionary for the layer properties.
            scalar nLayers=readScalar(meshDict.lookup("nLayers"));
            scalar stretching=readScalar(meshDict.lookup("stretching"));
            nLayers++;
            // label patchName=readLabel(meshDict.lookup("patch"));
            //Calculating the length of the cell that's going to be refined
(in weighted terms).


            scalar L[100];
            scalar Li[100];
            L[0]=0;
            L[1]=1;

            for(int j=1;j<nLayers-1;j++)
            {
                        L[j+1]=stretching*(L[j]-L[j-1])+L[j];
            }

            int k=0;

            for(int i=nLayers-2;i>=1;i--)
            {
                        Li[k]=1-(L[i+1]-L[i])/L[i+1];
                        Info << "weightedthings =" << Li[k] << endl;
                        k=k+1;

            }

            //splitting the closest row of cells in to nLayer amount of
cells. using the souce code from refineWallLayer.C.
```

So for example if one wants to create a boundary layer which has 5 layers, and with an allowed shearing of 1.2 a square block would look like the Figure 13.

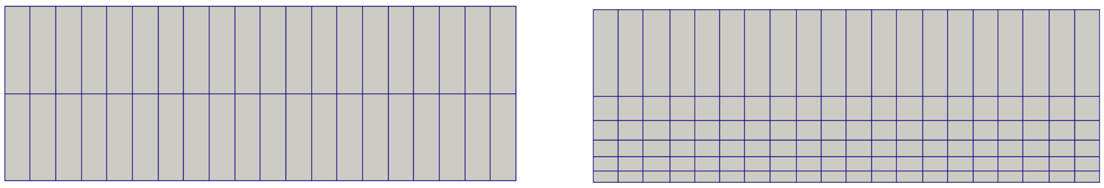

Figure 13, the figure above shows how a box (to the left) looks when refineBoundaryWallLayer is used on it (to the right).

5.1.4 refineBoundaryWallLayer.H

RefineBoundaryWallLayer.H is pretty much the same as *refineWallLayer.*C except the initial lines are removed; the code can be seen in Appendix 3.

refineWallLayer.C can be found in the folder `$WM_PROJECT_DIR/applications/utilities/mesh/advanced/refineWallLayer` whereas the *refineBoundaryWallLayer*.H should be placed in the *mysnappyHexMesh* folder.

6 Creation of snappyTestCase

To test the code of *mysnappyHexMesh* out a test case can be created, how to create it will be described in this chapter, and then further explanations of the files and dictionaries needed, can be read in chapter 7.

snappyTestCase needs a lot of different dictionaries, most of them, including the .stl surface used can be found in the tutorial SnakeRiverCanyon, therefore start by copying the tutorial.

```
cp -r $WM_PROJECT_DIR/tutorials/mesh/moveDynamicMesh/SnakeRiverCanyon
$WM_PROJECT_USER_DIR/run/snappyTestCase
```

To enter the test case type:

```
cd $WM_PROJECT_USER_DIR/run/snappyTestCase
```

The directory should look something like this:

```
snappyTestCase
-0/
--pointDisplacement
-constant/
--dynamicMeshDict
--transportProperties
--polyMesh/
---blockMeshDict
---boundary
--triSurface/
---ACROSSCYN.JPG
---AcrossCyn.XYZ
---AcrossRiver.stl.gz
-system/
--controlDict
--decomposeParDict
--fvSchemes
--fvSolution
```

An important file that's missing from this directory is the *snappyHexMeshDict* which should be placed in the system/ directory. It can be grabbed from any of the *snappyHexMesh* tutorials, or the provided dictionary that's existing with the *snappyHexMesh* code. To grab the *snappyHexMeshDict* from the *snappyHexMesh* code simply type:

```
cp
$WM_PROJECT_DIR/applications/utilities/mesh/generation/snappyHexMesh/sn
appyHexMeshDict $WM_PROJECT_USER_DIR/run/snappyTestCase/system/.
```

6.1 Changes to snappyHexMeshDict

Open *snappyHexMeshDict* by typing:

```
vi system/snappyHexMeshDict
```

Go down to line 22 and add:

```
projection            true;
DynMesh               true;
refineMesh            true;
boundaryLayer         true;

stretching            1.2;
nLayers               3;
nRefinements          2;
nDynIter              10;
patch                 AcrossRiver_patch0;
```

These are the options that were specified in the code, and will be described further in chapter 7.3.

Set *castellatedMesh* and *snap* to false by changing:

```
castellatedMesh true;
snap            true;
addLayers       false;
```

To:

```
castellatedMesh false;
snap            false;
addLayers       false;
```

Some more changes that need to be done is changing the geometry sub-dictionary, i.e.

```
geometry
{
    box1x1x1
    {
        type searchableBox;
        min (1.5 1 -0.5);
        max (3.5 2 0.5);
    }

    sphere.stl
    {
        type triSurfaceMesh;

           //tolerance   1E-5;   // optional:non-default tolerance on
intersections
        //maxTreeDepth 10;    // optional:depth of octree. Decrease only in
case
                             // of memory limitations.

        // Per region the patchname. If not provided will be
<name>_<region>.
```

```
        regions
        {
            secondSolid
            {
                name mySecondPatch;
            }
        }
    }

        sphere2
    {
        type searchableSphere;
        centre  (1.5 1.5 1.5);
        radius  1.03;
    }
};
```

This should be changed to:

```
geometry
{
    AcrossRiver.stl
    {
        type triSurfaceMesh;
        name AcrossRiver;
    }

    refinementBox
    {
        type searchableBox;
        min (659531  4.7513e+06   1028);
        max (662381  4.75454e+06  1200);
    }
};
```

Inside the *castellatedMeshControls* at the end of the controls there's a option called location in mesh, looking like this:

```
    // Mesh selection
    // ~~~~~~~~~~~~~~

    // After refinement patches get added for all refinementSurfaces and
    // all cells intersecting the surfaces get put into these patches. The
    // section reachable from the locationInMesh is kept.
    // NOTE: This point should never be on a face, always inside a cell,
even
    // after refinement.
    locationInMesh (5 0.28 0.43);
```

Change that too:

```
    locationInMesh (659535 4.7513e+06 1328);
```

That is all that needs to change in *snappyHexMeshDict*, save and exit.

6.2 Changes to dynamicMeshDict

Open *dynamicMeshDict* by typing:

```
vi constant/dynamicMeshDict
```

The information below should then be seen.

```
dynamicFvMesh dynamicMotionSolverFvMesh;


motionSolverLibs ("libfvMotionSolvers.so");

solver displacementSBRStress;   //displacementLaplacian;
//displacementSBRStress;
diffusivity quadratic quadratic inverseDistance 1(minZ);

//solver velocityComponentLaplacian z;
//diffusivity  uniform;
//diffusivity  directional (1 200 0);
// diffusivity  motionDirectional (1 1000 0);
// diffusivity  file motionDiffusivity;
diffusivity  quadratic inverseDistance 1(minZ);
// diffusivity  exponential 2000 inverseDistance 1(movingWall);
```

First of all the solver should be changed, i.e. where it says:

```
solver displacementSBRStress;   //displacementLaplacian;
//displacementSBRStress;
```

should be changed to:

```
solver velocityLaplacian;
```

The diffusivity just below should be removed or commented, i.e.

```
diffusivity quadratic quadratic inverseDistance 1(minZ);
```

should be written as:

```
//diffusivity quadratic quadratic inverseDistance 1(minZ);
```

Then instead of it saying:

```
diffusivity  quadratic inverseDistance 1(minZ);
```

minZ should be changed for maxZ, i.e. it should say:

```
diffusivity  quadratic inverseDistance 1(maxZ);
```

The final version of *dynamicMeshDict* should then look something like this:

```
dynamicFvMesh dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver velocitytLaplacian;
```

```
//diffusivity quadratic quadratic inverseDistance 1(minZ);

//solver velocityComponentLaplacian z;
//diffusivity  uniform;
//diffusivity  directional (1 200 0);
// diffusivity  motionDirectional (1 1000 0);
// diffusivity  file motionDiffusivity;
diffusivity  quadratic inverseDistance 1(maxZ);
// diffusivity  exponential 2000 inverseDistance 1(movingWall);
```

The changes to *dynamicMeshDict* are now done, just save and quit.

6.3 Changes to blockMeshDict

The only change needed in *blockMeshDict* is to change the boundary *minZ* seen below:

```
patches
(
    wall maxX
    (
        (3 7 6 2)
    )
    wall minZ
    (
        (0 4 7 3)
    )
```

To:

```
patches
(
    wall maxX
    (
        (3 7 6 2)
    )
    wall AcrossRiver_patch0
    (
        (0 4 7 3)
    )
```

6.4 Changes of pointDisplacement

First of all since the solver *velocityLaplacian* is supposed to be used the *pointDisplacement* file should actually be a *pointMotion* file, start by renaming the *pointDisplacement* file to *pointMotionU* by typing:

```
mv 0/pointDisplacement 0/pointMotionU
```

Now it's time to edit the *pointMotionU* file, start by opening the file by typing:

```
vi 0/pointMotionU
```

Since the *pointMotionU* file was changed from a displacement file, the dimensions will be wrong, change the dimensions from:

```
dimensions      [0 1 0 0 0 0 0];
```

Too:

```
dimensions      [0 1 -1 0 0 0 0];
```

On line 29 the boundary condition for *minZ* will start and look like below:

```
    minZ
    {
        type            surfaceDisplacement;
        value           uniform (0 0 0);

        // Clip displacement to surface by max deltaT*velocity.
        velocity            (10 10 10);

        geometry
        {
            AcrossRiver.stl
            {
                type triSurfaceMesh;
            }
        };

        // Find projection with surface:
        //     fixedNormal : intersections along prespecified direction
        //     pointNormal : intersections along current pointNormal of
patch
        //     nearest     : nearest point on surface
        // Other
        projectMode fixedNormal;

        // if fixedNormal : normal
        projectDirection (0 0 1);

        //- -1 or component to knock out before doing projection
        wedgePlane      -1;

        //- Points that should remain fixed
        //frozenPointsZone fixedPointsZone;
    }
```

Since the lower boundary should be fixed all of the above should be changed to:

```
    AcrossRiver_patch0
    {
            type            fixedValue;
            value           uniform (0 0 0);

    }
```

Above that the boundary condition for *maxZ* should stand as well, looking like the below:

```
    maxZ
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
```

Since the upper boundary should move in a up and down motion, change the above too:

```
    maxZ
    {

            type timeVaryingUniformFixedValue;
            fileName "velocity";
            outOfBounds repeat;

    }
```

The *pointMotionU* file is now correct, just save and exit.

6.5 Creation of the "velocity" file

In the *pointMotionU* file a file called velocity was referenced to how the *maxZ* boundary should move to create this file type, it should be located straight in the case folder:

```
vi velocity
```

And then just copy paste the below in to the file:

```
(
 (4.99 (0 0 -70))
 (10.0 (0 0 -70))
 (10.001 (0 0 70))
 (15.0 (0 0 70))
)
```

Save and quit.

6.6 Changes to the controlDict

Now most changes are done, the only thing remaining is to change a small thing in the *controlDict*, do this by typing:

```
vi system/controlDict
```

Change the:

```
startFrom       startTime;
```

To:

```
startFrom       latestTime;
```

That's all that needs to be changed in this file, just save and quit.

The test case is now done, and the directory structure should look something like this:

```
snappyTestCase
-0/
--pointMotionU
-constant/
--dynamicMeshDict
--transportProperties
--polyMesh/
---blockMeshDict
```

```
---boundary
--triSurface/
---ACROSSCYN.JPG
---AcrossCyn.XYZ
---AcrossRiver.stl.gz
-system/
--snappyHexMeshDict
--controlDict
--decomposeParDict
--fvSchemes
--fvSolution
-velocity
```

It is now just to run the case by typing:

```
blockMesh
mysnappyHexMesh
```

7 Explanation of the test case for mysnappyHexMesh

In this chapter the different files and dictionaries used in *mysnappyHexMesh* will be explained in more detail so the user can get an understanding of what the files do, and why they are there.

To utilize *mysnappyHexMesh* there are several files needed, all of them have been copied in the test case. First of all, of course a .stl surface of the topology that should be snapped is needed. Then a *blockMeshDict* file is needed to create an initial mesh. *mysnappyHexMesh* requires its own dictionary called *snappyHexMeshDict*, If internal mesh motion is needed, a *dynamicMeshDict* is also required.

7.1 .stl surface

The .stl surface should be placed in a folder called trisurface in the constant directory, i.e. $caseFolder/constant/trisurface/<surfaceName>.stl should exist.

A stl surface is a surface built out of a lot of triangular surfaces which together creates a complex geometry, these kind of files can be created by various mean, for example, most 3d-generation software's or meshing software's can convert a surface to a .stl file.

7.2 blockMeshDict

A *blockMeshDict* needs to be created in a *polyMesh* folder in the constant directory, i.e. $caseFolder/constant/polyMesh/blockMeshDict should exist. In the *blockMeshDict* there are some things that needs to be specified, first of all, eight points or "vertices" needs to be specified which will be the corners of the square that's the base of the mesh.

The points then needs to be connected as a hex, and the different boundaries will need to be specified, an example would be the code below.

One important thing for *mysnappyHexMesh* to be working is to have one of the boundaries (the one closest to the surface) should be named <NameOfThe.stlFile>_<NameOftheSurfaceInThe.stlFile>. I.e. for this case, where the .stl file is called AcrossRiver.stl and the surface in the .stl file is called patch0, hence the boundary should be called "AcrossRiver_patch0".

```
convertToMeters 1;

vertices
(
    ( 659531  4.7513e+06   900)
    ( 659531  4.7513e+06   2100)
    ( 662381  4.7513e+06   2100)
    ( 662381  4.7513e+06   900)
    ( 659531  4.75454e+06  900)
    ( 659531  4.75454e+06  2100)
    ( 662381  4.75454e+06  2100)
    ( 662381  4.75454e+06  900)
);
blocks
(
    hex (0 1 2 3 4 5 6 7) (4 20 20)  simpleGrading (1 1 1)
);

edges
(
);

patches
(
    wall maxX
    (
        (3 7 6 2)
    )
    wall AcrossRiver_patch0
    (
        (0 4 7 3)
    )
    wall maxZ
    (
        (2 6 5 1)
    )
    wall minX
    (
        (1 5 4 0)
    )
    wall minY
    (
        (0 3 2 1)
    )
    wall maxY
    (
        (4 5 6 7)
    )
);

mergePatchPairs
(
);
```

7.3 snappyHexMeshDict

Since *snappyHexMesh* is the basis of the operation, ofcourse a *snappyHexMeshDict* is needed to specify how the mesh snapping should be handled. The *snappyHexMeshDict* should be located in the system directory, i.e. $caseFolder/system/snappyHexMeshDict is needed. As seen above *snappyHexMeshDict* has had some additions to it, and the options below are the new ones.

```
projection            true;
```

The projection option activates (true) or deactivates (false) the *projection* feature, i.e. if the boundary should be snapped to the surface.

```
DynMesh              true;
```
The DynMesh option activates (true) or deactivates (false) the *moveDynamicMesh* feature, i.e. if the internal mesh points should be moved or not.

```
refineMesh           true;
```
The refineMesh option activates (true) or deactivates (false) the *refineMesh* feature, i.e. if the mesh should be refined or not.

```
boundaryLayer        true;
```
The boundaryLayer option activates (true) or deactivates (false) the *boundaryLayer* feature, i.e. if the mesh should get a boundary layer on the surface or not.

```
stretching           1.2;
```
Defines how much stretching is allowed when using the boundary wall layer feature i.e. how much bigger each cell-layer are allowed to become (if the first layer has the height of 1 unit, next layer will be 1.2 units high).

```
nLayers              5;
```
Defines how many layers that should be created when using the boundary wall layer feature.

```
nRefinements         2;
```
Defines how many times the mesh should be refined, and snapped to the surface.

```
nDynIter             10;
```
Defines how many time steps the moveDynamicMesh feature should do, always needs to be an even number.

```
patch                AcrossRiver_patch0;
```
Defines which patch the boundary layer should be put on, in general this means that where "patch" stands the name of the boundary which is supposed to be snapped should be written here, but if for some reason a person want boundary layer somewhere else, for example on the top boundary it's just to type maxZ instead.

In *snappyHexMeshDict* there are other things that are important for the mesh quality, and it's the settings for the snapping, in the *snapControls* subdictionary:

```
snapControls
{
    //- Number of patch smoothing iterations before finding correspondence
    //  to surface
    nSmoothPatch 3;

    //- Relative distance for points to be attracted by surface feature
point
    //  or edge. True distance is this factor times local
    //  maximum edge length.
    tolerance 4.0;

    //- Number of mesh displacement relaxation iterations.
    nSolveIter 30;

    //- Maximum number of snapping relaxation iterations. Should stop
    //  before upon reaching a correct mesh.
    nRelaxIter 5;
}
```

The importance of these values is explained in chapter 2.3.

7.4 dynamicMeshDict

The *dynamicMeshDict* is only needed if the internal mesh needs to be moved, i.e. the *DynMesh* option in *snappyHexMeshDict* is enabled. If the dictionary is needed it should be placed in the constant directory i.e. $caseFolder/constant/dynamicMeshDict should exist. More about the dynamicMeshDict is explained in the basics of *moveDynamicMesh* chapter. However the options that's needed or preferred for this case is displayed below.

```
dynamicFvMesh dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solver velocityLaplacian;

diffusivity  quadratic inverseDistance 1 (maxZ);
//diffusivity  exponential -0.01 inverseDistance 1 (AcrossRiver_patch0);
```

When using *dynamicMesh* with the "*velocityLaplacian*" solver, an additional file is needed, namely the *pointMotionU* file, this file describes the motion of the boundaries and will be explained further in chapter 6.2.5.

7.5 pointMotionU

The *pointMotionU* file needs to be placed in the 0/ directory and similarly to a regular U file that's used when solving fluid motion, the *pointMotionU* file is there to specify the boundary conditions of the mesh motion. Something special about this though is the fact that it's a point file, i.e. all values are specified at the points, instead of the cell centers where the information usually is located. How this file should look can be seen in the code below.

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    maxX
    {
        type            slip;
    }
    maxY
    {
        type            slip;
    }
    minY
    {
        type            slip;
    }
    maxZ
    {
        type timeVaryingUniformFixedValue;
        fileName "velocity";
        outOfBounds repeat;

    }
    AcrossRiver_patch0
    {
         type            fixedValue;
         value           uniform (0 0 0);

    }
    minX
    {
        type            slip;
    }
}
```

As can be seen this is a usual field file for OpenFoam where the dimension, internal field and external field is specified. The two first options are nothing special, the dimension is specified as m/s because it's *velocityLaplacian* that's used, and the internal field is set to uniform 0. For the external field comes some important settings though. In general, the boundary which is snapped should get a locked boundary condition, i.e. a fixed value of uniform 0 like shown below.

```
    AcrossRiver_patch0
    {
         type            fixedValue;
         value           uniform (0 0 0);

    }
```

The boundary opposite to the snapped boundary is most important though. Namely the boundary opposite of the snapped boundary should get a condition that changes with time more specifically that it varies between a positive and a negative value, this can be achieved in many ways, and only one will be mentioned here.

By putting the boundary condition as a *timeVayingUniformFixedValue* the boundary motion can be controlled in an external file, how this file should look will be mentioned in chapter 6.2.6. The code would then look like below.

```
    maxZ
    {

        type timeVaryingUniformFixedValue;
        fileName "velocity";
        outOfBounds repeat;

    }
```

Where the "*fileName*" option specifies what file should be used, and the option "*outOfBounds*" specifies how OpenFoam should handle times which isn't defined in the file.

All other boundaries should be selected with a slip condition, like the code below.

```
    minY
    {
        type            slip;
    }
```

7.6 Velocity boundary file

The "*velocity*" file, or the file which specifies how the boundary opposite of the snapped surface should move needs to be placed directly in the case folder i.e. a file $CaseFolder/velocity should exist. And should look something like the code below

```
(
 (5 (0 0 -70))
 (10.0 (0 0 -70))
 (10.001 (0 0 70))
 (15.0 (0 0 70))
)
```

This file varies greatly on the settings used in *snappyHexMeshDict* and the two options that need to be considered when writing this file is the "*nRefinements*" option and the "*nDynIter*" option.

The *nRefinements* option defines at what time the dynamic iterations start, this time can be calculated as 1+2**nrefinements*, so for the example used here where *nRefinements* is 2, the starting time would be 5.

The *nDynIter* option defines at what time the dynamic iterations ends and also at what time the boundary should start moving upwards, the end time for the dynamic iterations can be calculated as 1+2**nRefinements* + *nDynIter*, so for the example used here where *nRefinements* is 2, *nDynIter* is 10 the end time would be 15.

For a general case the velocity file could be written as

```
(
 (1+2*nRefinements                          (0 0 -70))
 (1+2*nRefinements + nDynIter/2             (0 0 -70))
 (1+2*nRefinements + nDynIter/2 +0.001      (0 0 70))
 (1+2*nRefinements + nDynIter               (0 0 70))
)
```

The last option in this file is the velocity at which the boundary should move, this value depends on how big the mesh is, and how many *nDynIter* that's going to be made. For the test case provided here, 70 worked well, if the amount of dynamic iterations would increase or if the mesh size itself was smaller, the value would have to be smaller.

The reason the boundary needs to go down is to move the internal mesh points so they are closer to the projected surface, however doing this the entire mesh get's smaller, and to keep the size ratio of the mesh, the boundary needs to be moved upwards.

7.7 Running the case

When all these things are set-up the only thing needed to do is to type >> *blockMesh* to generate the base mesh, and then type >> *mysnappyHexMesh* in the command window, and the mesh will be generated.

When the command *mysnappyHexMesh* has been written, the mesh manipulation will start, for the test case above, the starting mesh and the first snapping will look like the figure 14.

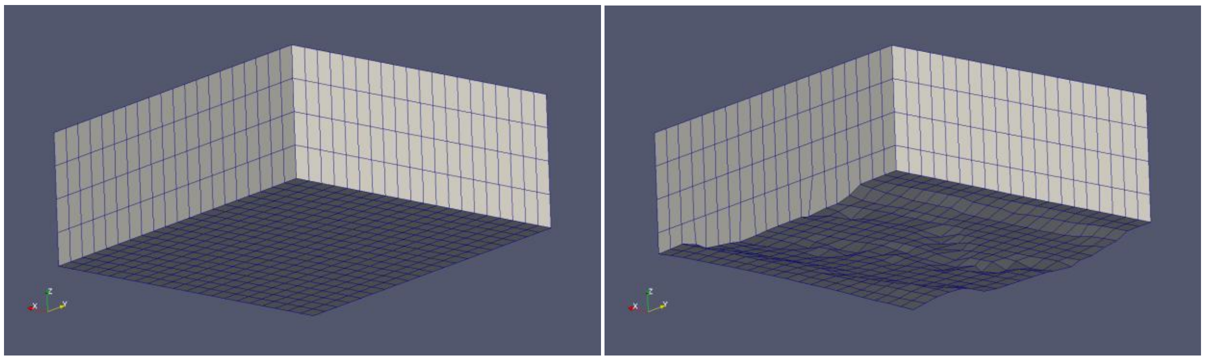

Figure 14, the figure to the left shows how the mesh generated by blockMesh looks, i.e. this is the initial mesh of which mysnappyHexMesh does its manipulations on, and the figure to the right shows how the mesh looks like after the first snapping .

After the first snapping *mysnappyHexMesh* does the first refinement, and after that the second snapping is done, this can be seen in figure 15.

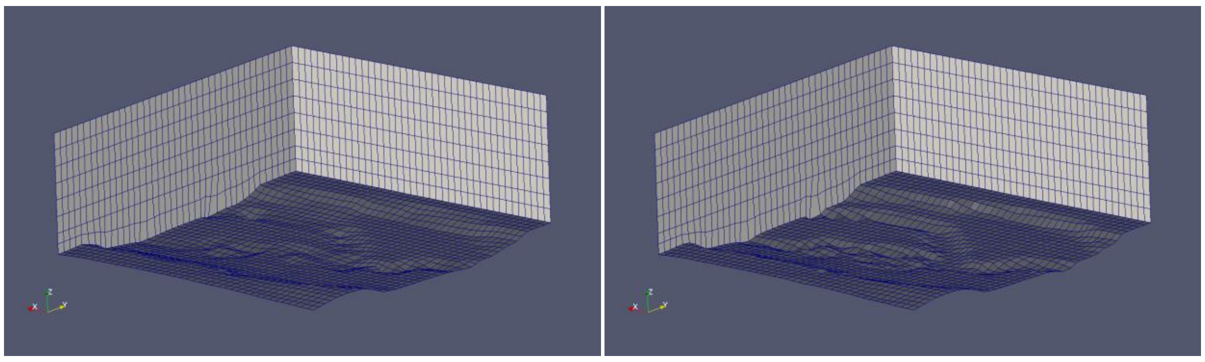

Figure 15, the figure to the left shows how the mesh looks after the first refinement, and the figure to the right shows how the mesh looks like after the second snapping.

When the second snapping is done, the second refinement and the third snapping begins, this can be seen in figure 16.

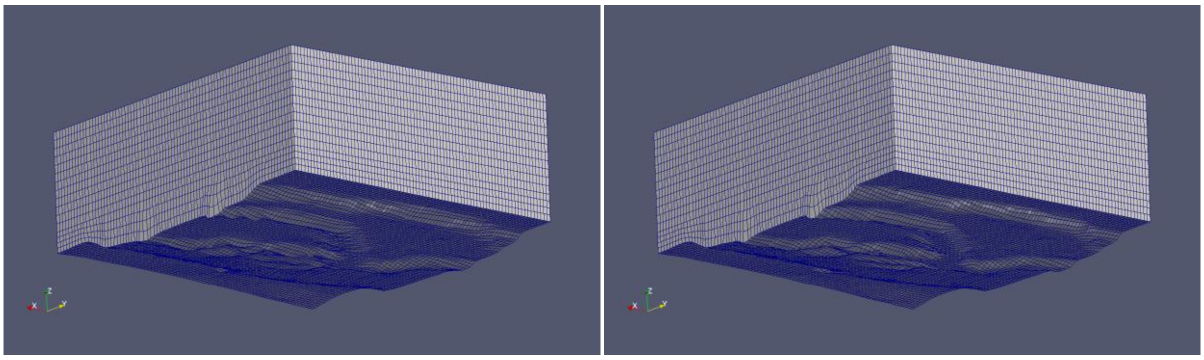
Figure 16, the figure to the left shows how the mesh looks after the second refinement, and the figure to the right shows how the mesh looks like after the third snapping.

With the third snapping, using a *nRefinements* of 2 the snapping and refining is now done, and the internal mesh points needs to be moved slightly, this is done with the *moveDynamicMesh* feature in a up and down motion, the two extreme points of the mesh motion can be seen in figure 17.

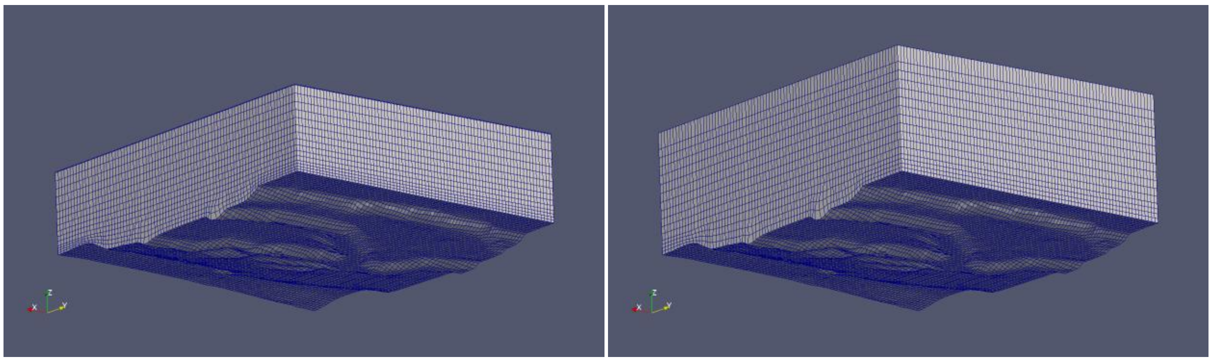
Figure 17, the figure to the left shows how the mesh looks after half of the dynamic iterations and the figure to the right shows how the mesh looks like after all dynamic iterations.

As can be seen the mesh moves down so that it is very compressed and then moves up to its original size, with the internal mesh being much more compressed in the lower part of the boundary compared to before.

When the internal mesh is moved all that's left is to create the boundary wall layer, this is done layer by layer and how it looks can be seen in figure 18.

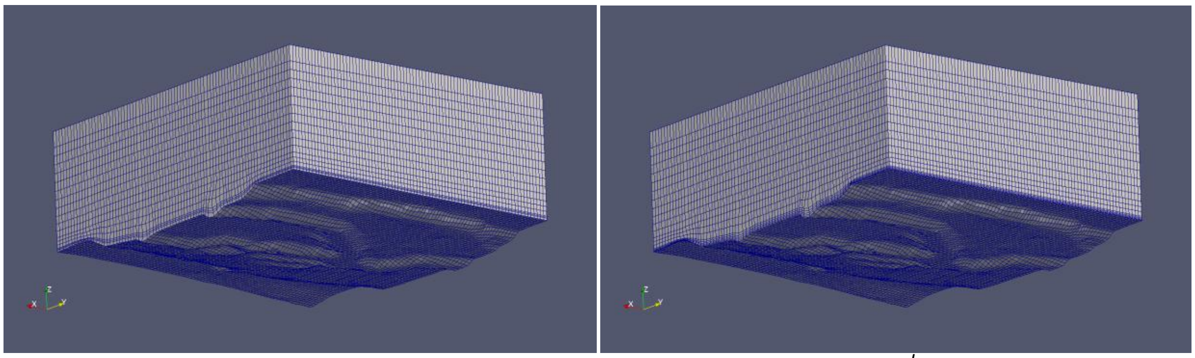

Figure 18, the figure to the left shows how the mesh looks after the 2nd layer is created and the figure to the right shows how the mesh looks after all layers have been created.

The mesh is now complete and some bigger figures can be seen below, where figure 19 shows the whole mesh and figure 20 is zoomed in on one part of the border to give a better picture of how the boundary wall layer looks like.

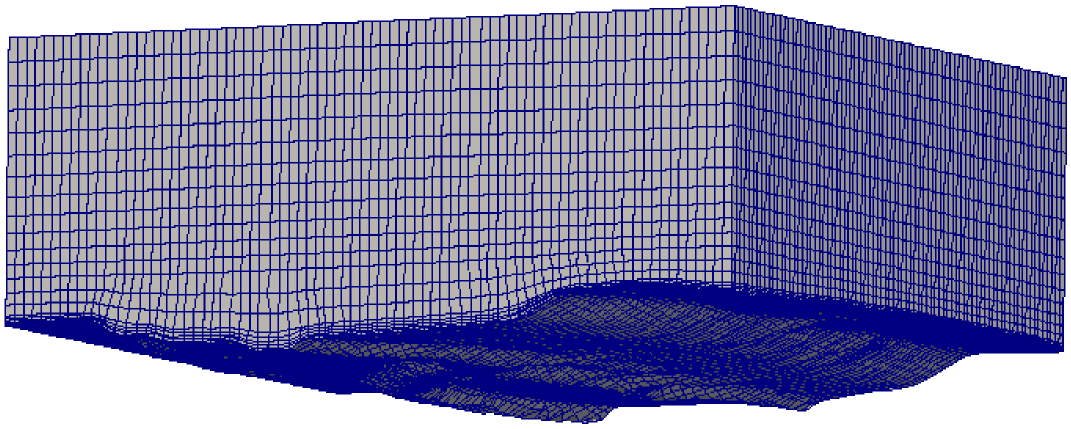

Figure 19, one of the sides of the mesh generated by mysnappyHexMesh.

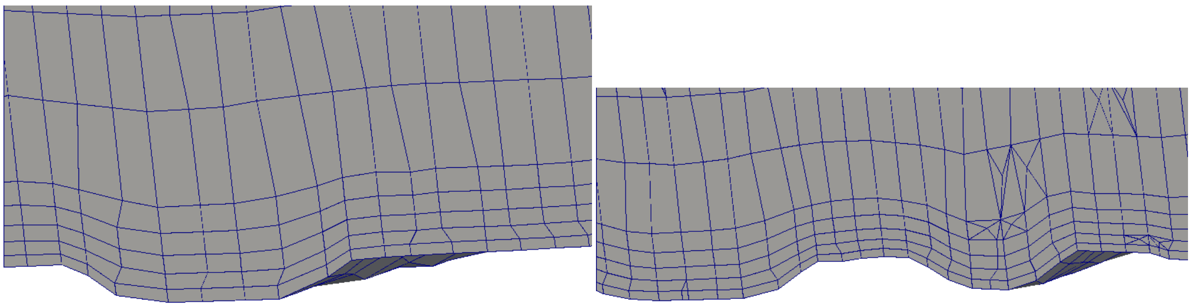

Figure 20, the mesh generated by mysnappyHexMesh can be seen in close up to the left and then to the right the mesh is split in half to show how the internal mesh looks like.

8 Best Practices

This program, *mysnappyHexMesh* is very sensitive to changes in the initial settings of the case. Therefore a chapter like this is most likely needed.

mysnappyHexMesh is built to snap the mesh to a surface, but due to the way *snappyHexMesh* is programmed (snap to closest surface point) the quality of the mesh will depend highly on how similar the initial mesh is to the surface that should be snapped. Therefore, a rougher initial mesh (in the x- and y-direction), and a couple of mesh-refinements during the snapping is a good way to go.

In general the *moveDynamicMesh* is bad, and will create a less than stellar mesh (i.e. the mesh can become skewed and for example the boundary layer will not be of equal height along the boundary). In general *moveDynamicMesh* should only be used if necessary due to how the surface looks. The finer initial mesh (in the z-direction) the more the *moveDynamicMesh* will need to be used, and also, the worse the final mesh becomes, therefore a course mesh is better in the z-direction as well.

If the mesh is good in the x- and y-direction but not in the z-direction (in terms of amounts of cells) then *refineMesh* with *refineMeshDict* can be used to refine only in the z-direction (this is preferably done before creating the boundary layer).

If the initial mesh (after the first snap) has a bad shape, i.e. the mesh looks skewed, the value *nRelaxIter* in *snappyHexMeshDict* should be increased, and this will lead to a longer mesh creation time, but may solve the issue.

If the initial mesh (after the first snap) has some points which hasn’t snapped, either increase the tolerance, or create a rougher initial mesh.

The surface can never cross an internal mesh point in the z-direction, but should also be placed in such way that the mesh boundary needs to move as little as possible.

9 Further work for future improvements

As been mentioned earlier in chapter 7, the *moveDynamicMesh* feature has a chance to also "destroy" the mesh if used incorrectly. This is because at the moment internal mesh motion through *moveDynamicMesh* is exclusively done by an external motion in one of the boundaries, this leads to a lesser degree of control.

Instead of using an existing feature (in this case *dynamicMotionSolverFvMesh*) one could write a totally new one, which focused on moving internal mesh. Not due to a movement in the boundary, but rather because of a "force" on the internal mesh point, where the force on the point depends on how close it is to a specific boundary.

There are several improvements in the coding that could be done, for example, at the moment; even if you're not going to use *moveDynamicMesh* the *pointMotionU* file in the `0/` directory is still required.

References

[1] Xabier Pedruelo Tapia 2009: “Modelling of wind flow over complex terrain using OpenFoam”
http://hig.diva-portal.org/smash/get/diva2:228936/FULLTEXT01
[2] Pirooz Moradnia 2007: “A tutorial on how to use Dynamic Mesh solver IcoDyMFOAM”
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2007/PiroozMoradnia/OpenFOAM-rapport.pdf

Appendix

Appendix 1 - refineMesh.H

```
#   include "createPolyMesh.H"
    const word oldInstance = mesh.pointsInstance();
//cpuTime timer;
    printEdgeStats(mesh);

pointMesh pMesh(mesh);
           pointVectorField pointMotionU
           (
               IOobject
               (
                    "pointMotionU",
                    runTime.timeName(),
                    mesh,
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                pMesh
           );


    //
    // Read/construct control dictionary
    //

    bool readDict = args.optionFound("dict");
    bool overwrite = args.optionFound("overwrite");

    // List of cells to refine
    labelList refCells;

    // Dictionary to control refinement
    dictionary refineDict;

    if (readDict)
    {
        Info<< "Refining according to refineMeshDict" << nl << endl;

        refineDict =
            IOdictionary
            (
                IOobject
                (
                    "refineMeshDict",
                    runTime.system(),
                    mesh,
                    IOobject::MUST_READ,
                    IOobject::NO_WRITE
                )
            );

        word setName(refineDict.lookup("set"));

        cellSet cells(mesh, setName);

        Pout<< "Read " << cells.size() << " cells from cellSet "
```

```
            << cells.instance()/cells.local()/cells.name()
            << endl << endl;

        refCells = cells.toc();
    }
    else
    {
        Info<< "Refining all cells" << nl << endl;

        // Select all cells
        refCells.setSize(mesh.nCells());

        forAll(mesh.cells(), cellI)
        {
            refCells[cellI] = cellI;
        }


        // Set refinement directions based on 2D/3D
        label axisIndex = twoDNess(mesh);

        if (axisIndex == -1)
        {
            Info<< "3D case; refining all directions" << nl << endl;

            wordList directions(3);
            directions[0] = "tan1";
            directions[1] = "tan2";
            directions[2] = "normal";
            refineDict.add("directions", directions);

            // Use hex cutter
            refineDict.add("useHexTopology", "true");
        }
        else
        {
            wordList directions(2);

            if (axisIndex == 0)
            {
                Info<< "2D case; refining in directions y,z\n" << endl;
                directions[0] = "tan2";
                directions[1] = "normal";
            }
            else if (axisIndex == 1)
            {
                Info<< "2D case; refining in directions x,z\n" << endl;
                directions[0] = "tan1";
                directions[1] = "normal";
            }
            else
            {
                Info<< "2D case; refining in directions x,y\n" << endl;
                directions[0] = "tan1";
                directions[1] = "tan2";
            }

            refineDict.add("directions", directions);

            // Use standard cutter
            refineDict.add("useHexTopology", "false");
        }

        refineDict.add("coordinateSystem", "global");

        dictionary coeffsDict;
        coeffsDict.add("tan1", vector(1, 0, 0));
        coeffsDict.add("tan2", vector(0, 1, 0));
        refineDict.add("globalCoeffs", coeffsDict);

        refineDict.add("geometricCut", "false");
        refineDict.add("writeMesh", "false");
    }


    string oldTimeName(runTime.timeName());

    if (!overwrite)
    {
        const_cast<Time&>(mesh.time())++; //runTime++;
    }

    // Multi-directional refinement (does multiple iterations)
    multiDirRefinement multiRef(mesh, refCells, refineDict);


    // Write resulting mesh
    if (overwrite)
    {
        mesh.setInstance(oldInstance);
    }
    mesh.write();



    // Get list of cell splits.
    // (is for every cell in old mesh the cells they have been split into)
    const labelListList& oldToNew = multiRef.addedCells();


    // Create cellSet with added cells for easy inspection
    cellSet newCells(mesh, "refinedCells", refCells.size());

    forAll(oldToNew, oldCellI)
    {
        const labelList& added = oldToNew[oldCellI];

        forAll(added, i)
        {
            newCells.insert(added[i]);
        }
    }

    Pout<< "Writing refined cells (" << newCells.size() << ") to cellSet "
        << newCells.instance()/newCells.local()/newCells.name()
        << endl << endl;

    newCells.write();
```

```
//
// Invert cell split to construct map from new to old
//

labelIOList newToOld
(
    IOobject
    (
        "cellMap",
        runTime.timeName(),
        polyMesh::meshSubDir,
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh.nCells()
);
newToOld.note() =
    "From cells in mesh at "
  + runTime.timeName()
  + " to cells in mesh at "
  + oldTimeName;


forAll(oldToNew, oldCellI)
{
    const labelList& added = oldToNew[oldCellI];

    if (added.size())
    {
        forAll(added, i)
        {
            newToOld[added[i]] = oldCellI;
        }
    }
    else
    {
        // Unrefined cell
        newToOld[oldCellI] = oldCellI;
    }
}

Info<< "Writing map from new to old cell to "
    << newToOld.objectPath() << nl << endl;

newToOld.write();


// Some statistics.

printEdgeStats(mesh);
```

Appendix 2 – moveDynamicMesh.H

```
if (wantDynMesh && Testing==10)
           {
                      #   include "setRootCase.H"
                      #   include "createTime.H"
                      #   include "createDynamicFvMesh.H"

                      for(int i=0;i<nDynIter;i++)
                      {
                      if (!overwrite)
                      {

           const_cast<Time&>(mesh.time())++;
                      }
                      Info<< "Time = " << runTime.timeName() << endl;

                      if (isDir(runTime.path()/"VTK"))
                      {
                                            Info << "Clear VTK directory" <<
endl;
                                            rmDir(runTime.path()/"VTK");
                      }

                      mesh.update();

                                 //#       include "checkVolContinuity.H"

                      mesh.checkMesh(true);

                                 runTime.write();

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




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

Appendix 3 – refineBoundaryWallLayer.H

```
//reading the dictionary for the layer properties.
    scalar nLayers=readScalar(meshDict.lookup("nLayers"));
nLayers++;

//Calculating the length of the cell that's going to be refined (in
weighted terms).


#   include "setRootCase.H"
#   include "createTime.H"
    runTime.functionObjects().off();
#   include "createPolyMesh.H"
    const word oldInstance = mesh.pointsInstance();


    word patchName
    (
        meshDict.lookup("patch")
    );


    bool overwrite = args.optionFound("overwrite");


    label patchID = mesh.boundaryMesh().findPatchID(patchName);

    if (patchID == -1)
    {
        FatalErrorIn(args.executable())
            << "Cannot find patch " << patchName << endl
            << "Valid patches are " << mesh.boundaryMesh().names()
            << exit(FatalError);
    }
    const polyPatch& pp = mesh.boundaryMesh()[patchID];


    // Cells cut

    labelHashSet cutCells(4*pp.size());

    const labelList& meshPoints = pp.meshPoints();

    forAll(meshPoints, pointI)
    {
        label meshPointI = meshPoints[pointI];

        const labelList& pCells = mesh.pointCells()[meshPointI];

        forAll(pCells, pCellI)
        {
            cutCells.insert(pCells[pCellI]);
        }
    }

    Info<< "Selected " << cutCells.size()
        << " cells connected to patch " << pp.name() << endl << endl;

    //
    // List of cells to refine
    //

    bool useSet = args.optionFound("useSet");

    if (useSet)
    {
        word setName(args.option("useSet"));

        Info<< "Subsetting cells to cut based on cellSet" << setName <<
endl
            << endl;

        cellSet cells(mesh, setName);

        Info<< "Read " << cells.size() << " cells from cellSet "
            << cells.instance()/cells.local()/cells.name()
            << endl << endl;

        for
        (
            cellSet::const_iterator iter = cells.begin();
            iter != cells.end();
            ++iter
        )
        {
            cutCells.erase(iter.key());
        }
        Info<< "Removed from cells to cut all the ones not in set " <<
setName
            << endl << endl;
    }

    // Mark all meshpoints on patch

    boolList vertOnPatch(mesh.nPoints(), false);

    forAll(meshPoints, pointI)
    {
        label meshPointI = meshPoints[pointI];

        vertOnPatch[meshPointI] = true;
    }


    // Mark cut edges.

    DynamicList<label> allCutEdges(pp.nEdges());

    DynamicList<scalar> allCutEdgeWeights(pp.nEdges());

    forAll(meshPoints, pointI)
    {
        label meshPointI = meshPoints[pointI];

        const labelList& pEdges = mesh.pointEdges()[meshPointI];

        forAll(pEdges, pEdgeI)
        {
            label edgeI = pEdges[pEdgeI];

            const edge& e = mesh.edges()[edgeI];
            label otherPointI = e.otherVertex(meshPointI);

            if (!vertOnPatch[otherPointI])
            {
                allCutEdges.append(edgeI);

                if (e.start() == meshPointI)
                {
                    allCutEdgeWeights.append(weight);
                }
                else
                {
                    allCutEdgeWeights.append(1 - weight);
                }
            }
        }
    }

    allCutEdges.shrink();
    allCutEdgeWeights.shrink();

    Info<< "Cutting:" << endl
        << "    cells:" << cutCells.size() << endl
        << "    edges:" << allCutEdges.size() << endl
        << endl;

    // Transfer DynamicLists to straight ones.
    scalarField cutEdgeWeights;
    cutEdgeWeights.transfer(allCutEdgeWeights);
    allCutEdgeWeights.clear();


    // Gets cuts across cells from cuts through edges.
    cellCuts cuts
    (
        mesh,
        cutCells.toc(),     // cells candidate for cutting
        labelList(0),       // cut vertices
        allCutEdges,        // cut edges
        cutEdgeWeights      // weight on cut edges
    );

    directTopoChange meshMod(mesh);

    // Cutting engine
    meshCutter cutter(mesh);

    // Insert mesh refinement into directTopoChange.
    cutter.setRefinement(cuts, meshMod);

    // Do all changes
    Info<< "Morphing ..." << endl;

    if (!overwrite)
    {
        runTime++;
    }

    autoPtr<mapPolyMesh> morphMap = meshMod.changeMesh(mesh, false);
    if (morphMap().hasMotionPoints())
    {
        mesh.movePoints(morphMap().preMotionPoints());
    }

    // Update stored labels on meshCutter.
    cutter.updateMesh(morphMap());

    if (overwrite)
    {
        mesh.setInstance(oldInstance);
    }

    // Write resulting mesh
    Info << "Writing refined morphMesh to time " << runTime.timeName() <<
endl;

    mesh.write();

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