Cite as: Tisovska, P.: Description of the overset mesh approach in ESI version of OpenFOAM. In Proceedings of CFD with OpenSource Software, 2019, Edited by Nilsson. H., `http://dx.doi.org/10.17196/OS_CFD#YEAR_2019`

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Description of the overset mesh approach in ESI version of OpenFOAM

Developed for OpenFOAM v1906
Requires: overPimpleDyMFoam

Author:
TISOVSKA PETRA
Technical University of Liberec
petra.tisovska@gmail.com

Peer reviewed by:
CONSTANTIN SULA
MUYE GE

December 22, 2019

Learning outcomes

The main requirements of a tutorial 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:

- description of how to run a tutorial case simpleRotor with overPimpleDyMFoam
- modification of this case for moving overset mesh
- different approaches for mesh generation

The theory of it:

- the principle of overset is described
- the inverse distance interpolation scheme is analyzed

How it is implemented:

- differences between of simpleFoam and overSimpleFoam are described from overset point of view
- the structure of the chapter 3 follows the process of implementation of the overSimpleFoam solver
- the implementation of the interpolation scheme is analyzed

How to modify it:

- the complete procedure of modifying a function in the current library is described according to OpenFOAM principles
- the modification of stencilWeights function is given
- neighbors of original donors are added into account for weights computation

Prerequisites

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

- to run standard document tutorials like damBreak tutorial and to understand the meaning of files in this tutorial
- to understand the blockMesh utility
- to have a a basic knowledge of the finite volume method and how the discretization process works
- to have basic understanding of object oriented programming.

Contents

Introduction

The overset mesh approach is an implementation which uses disconnected meshes in OpenFOAM. It can be also called chimera framework. The CFD solution on the system of meshes requires interpolation between the overlapped regions of the meshes, which do not require connectivity. The overset grid approach can be highly useful in cases, when a rigid body moves in the fluid domain and when the displacements are too high to be captured by mesh deformation. It can also be used for mesh optimization or for simplified mesh generation. Its main disadvantage is extension of computational costs. Benefits of overset are described in more detail in chapter 2.

This project report explores the overset grid approach implemented in OpenFOAM v1906 and is divided into four parts. The first chapter contains the description of all different approaches used in tutorials, which are available in OpenFOAM v1906. The simpleRotor tutorial is described in detail. The different ways to generate overset mesh are given and described in detail. The last part of the chapter describes a modification of one of the tutorials.

The second chapter explains the principle of the overset method. The theory of this framework is based on program code in OpenFOAM. This chapter also includes the theory of the inverseDistance interpolation scheme. The description of program code is given in the third chapter, which also compares the differences between the simpleFoam and overSimpleFoam solvers.

The last chapter proposes a modification of the library and it gives the step-by-step tutorial. It is based on OpenFOAM principle that the user should not interfere with original files. The modification extends the inverseDistance interpolation scheme. The interpolated value computation is extended to neighbours of neighbours, instead of original form, which takes to the computation just the neighbours of given cell.

Chapter 1

Tutorials for overset mesh

This chapter is divided into three sections. The first section is about tutorial provided in OpenFOAM v1906 simpleRotor. The second section is about different ways to generate and use overset mesh. The last section is modification of one tutorial case.

Recommendation: For better understanding it is recommended to copy the tutorial that is described to your user directory and see the files that are described below. After the description of one tutorial it is recommended to run it and see results in paraFoam. In case of uncerternity how to run it correctly, the commands are listed in relevant Allrun or Allrun.pre script.

1.1 The simpleRotor tutorial

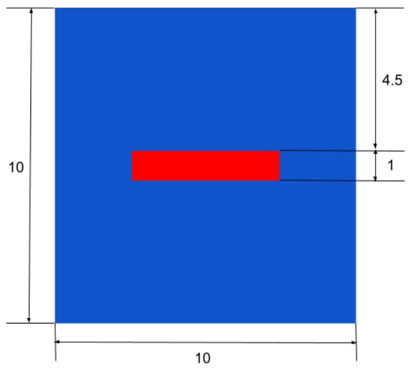
This tutorial is a two-dimensional case with moving overset mesh, which is run by solver overPimpleDyMFoam. In tutorial, the Allrun, Allrun.pre and Allclear scripts are provided. The commands of Allrun and Allrun.pre scripts is described in following sections. If you want to run the tutorial, you should copy it to your user directory by executing

```
OFv1906
cp -r $FOAM_TUTORIALS/incompressible/overPimpleDyMFoam/simpleRotor $FOAM_RUN/
```

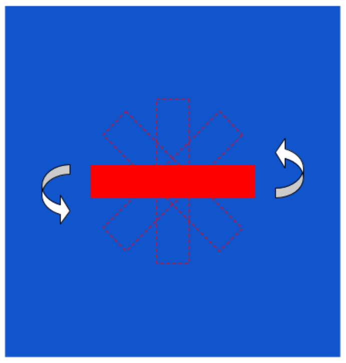
and run it by a command

```
./Allrun
```

The rotor is rotating in domain and creates fluid motion. The dimensions are shown in figure 1.1a and the movent is implied in figure 1.1b. The fluid is colored blue and the rotor is red.



(a) Domain dimensions



(b) Rotor motion

Figure 1.1: Schematic view of simpleRotor tutorial case

1.1.1 Mesh generation

In this section the generation of mesh with overmesh region is described. All commands described here can be found in Allrun.pre script. Figure 1.2 shows mesh after running

```
cd $FOAM_RUN/simpleRotor
blockMesh
```

The blockMesh file is set up to create two overlapped blocks – fluid region and overset mesh. No rotor part is introduced yet.

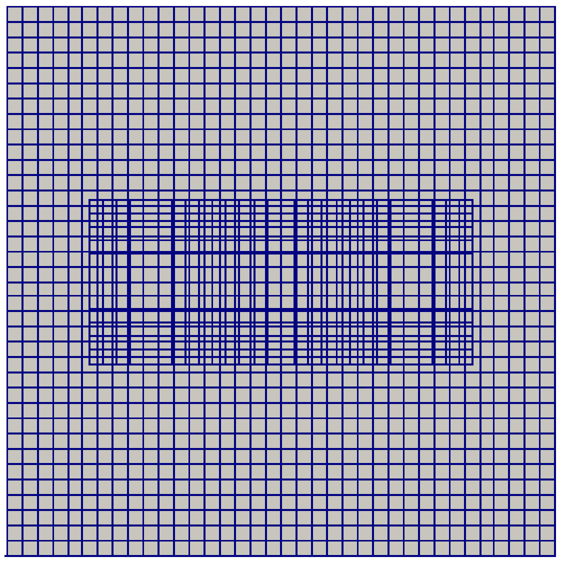


Figure 1.2: Overlapped mesh after blockMesh command

The next step is to run command

```
topoSet
```

It has two functions. The first part of topoSet file is

```
    {
        name    c0;
        type    cellSet;
        action  new;
        source  regionToCell;
        insidePoints ((0.001 0.001 0.001));
    }

    {
        name    c1;
        type    cellSet;
        action  new;
        source  cellToCell;
        set     c0;
    }

    {
        name    c1;
        type    cellSet;
        action  invert;
    }
```

The purpose of this part is to divide the mesh into two parts: c0 and c1. The c0 is the mesh in background. The c1 is the mesh around the rotor. Another step is to establish the space, where the rotor is supposed to be. The box is created in the second part of topoSet file.

```
    {
        name    box;
        type    cellSet;
        action  new;
        source  cellToCell;
        set     c1;
    }

    {
        name    box;
        type    cellSet;
        action  subset;
        source  boxToCell;
        box     (0.0025 0.0045 -100)(0.0075 0.0055 100);
    }

    {
        name    box;
        type    cellSet;
        action  invert;
    }
```

When the box is establish, the utility subsetMesh is used

```
subsetMesh box -patch hole -overwrite
```

This command creates a new patch from the region box with name hole and the old mesh is overwritten. After the creation of patch for the rotor, the topoSet command is run again to set the zones c0 and c1. Figure 1.3 shows the outcome from this settings.

The zero folder is restored from 0.orig. It can be done by command in Allrun.pre script `restore0Dir` or without the script by

```
cp -r 0.orig 0
```

The last command in Allrun.pre script is `setFields`. In this case the command is used for setting zones for overset. The ID of zone c0 is zero and the ID for zone c1 is 1. These values will be described in chapter 2.

1.1.2 Folder zero

In this section the main focus is on zoneID and pointDisplacement. The knowledge of standard files with boundary condition such as k, epsilon, nut, nuTilda, U, p is assumed.

The file pointDisplacement specifies the conditions for points movement for each boundary. In this case, this file tells the solver that there should be no moving except for hole and oveset patch type. The type of the condition specified here is zeroGradient.

The file zoneID defines the type oveset for overset patch and zeroGradient for each of other boundaries. Above this declaration, the file containts include statement

```
#includeEtc "caseDicts/setConstraintTypes"
```

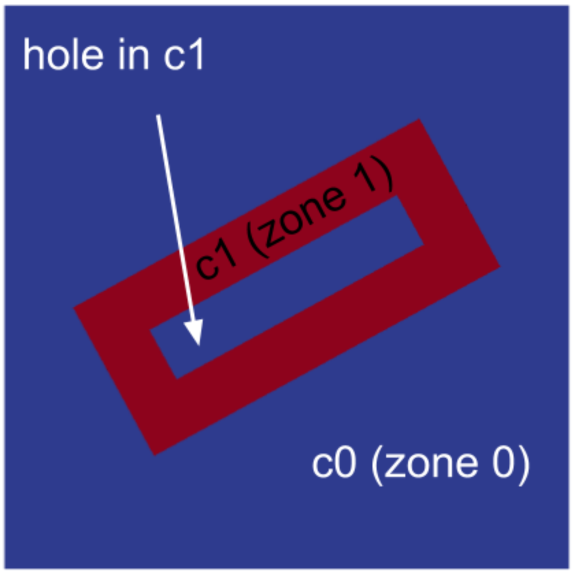


Figure 1.3: The zones of mesh distinguished by color

The reason for using this include statement is to shorten the zoneID file. This included file has following contain.

```
cyclic
{
    type  cyclic;
}

cyclicAMI
{
    type  cyclicAMI;
}

cyclicACMI
{
    type  cyclicACMI;
    value $internalField;
}

cyclicSlip
{
    type  cyclicSlip;
}

empty
{
    type  empty;
}

nonuniformTransformCyclic
{
    type  nonuniformTransformCyclic;
}

processor
{
    type  processor;
    value $internalField;
}

processorCyclic
{
    type  processorCyclic;
    value $internalField;
}

symmetryPlane
{
    type  symmetryPlane;
}

symmetry
{
    type  symmetry;
}

wedge
{
    type  wedge;
}

overset
{
    type  overset;
}
```

This file is changed by running the setFields command. The zone ID is assigned to each of internal cells according the settings in setFieldsDict file.

1.1.3 Folder constant

The turbulenceProperties defines the type of flow, in this case the laminar option is chosen. The file RASProperties is intended for turbulence model specification. With the laminar flow option, this file is not used.

The transportProperties specifies physical properties of fluid and transport model. The coefficients mentioned in this file are described in table 1.1.

Symbol	name
DT	diffusion coefficient
nu	viscosity

Table 1.1: Constants of transport properties file

The newtonian model is selected by

```
transportModel  Newtonian;
```

The user of the tutorial can choose different transport models for whose the coefficients are provided as

```
CrossPowerLawCoeffs
BirdCarreauCoeffs
```

The Newtonian model assumes that viscosity is constant. The other models estimate the viscosity as infinity value plus correction, which is affected by coefficients provided by user.

The file dynamicMeshDict is discussed in detail in this section. Its main purpose is to determine the mesh motion. In this case the solver `multiSolidBodyMotionSolver` is selected. This solver tells the overPimpleDyMFoam solver how to move with the selected zone. In this case the zone named movingZone is supposed to rotate by defined parameters: origin, axis and omega. The name of the zone is set in the blockMeshDict file.

The file with the same name and purpose can be found in tutorial cases for pimpleDyMFoam. These types of motion can be described: rotation about an axis, linear movement, combination of both, oscillating linear or rotational motion and finally motion described by tabular data.

Different solvers for motion can also be selected. For example the solver sixDoFRigidBodyMotion.

1.1.4 Folder system

Most of contents of this file is already mentioned in section 1.1.1 because they are assigned with mesh generation. The remaining files are: controlDict, decomposeParDict, fvSchemes and fvSolution. The controlDict control the whole simulation. The decomposeParDict is used for running the simulation in parallel.

The file fvSolution is used for setting up the numerical parameters of the simulation. There is nothing specific for the overset mesh approach. The file fvSchemes has to be set interpolation schemes for overset, in this case it is the inverseDistance scheme.

1.1.5 Running the case

The final step is to run the case with command

```
cd $FOAM_RUN/simpleRotor
overPimpleDyMFoam
```

After running the results can be displayed in Paraview by executing the command `paraFoam`.

1.2 Different approaches of mesh generation for overset mesh

In this section, the possibilities how to generate overset meshes is discussed. A different tutorial cases are explored. The solver using overset mesh can be recognised by the fact that it is name starting with over. The solvers can be found with terminal command

```
find $FOAM_SOLVERS -iname "over*"
```

The ones existing in OpenFOAM v1906 are listed below.

- compressible – overRhoPimpleDyMFoam
- compressible – overRhoSimpleFoam
- basic – overLaplacianDyMFoam
- basic – overPotentialFoam
- multiphase – overInterDyMFoam
- incompressible – overSimpleFoam
- incompressible – overPimpleDyMFoam
- heatTransfer – overBuoyantPimpleDyMFoam

To be able to run these solvers, the case must have at least one patch of type overset. It is strongly recommended that the overset patch is introduced first. The outcome of the interpolation is an assymetric matrix. The solver in fvSolution has to be adjusted. All asymmetric solvers except from GAMG are supported. In practice a good choice is the smooth (guide-solvers-smooth-smooth) solver with symGaussSeidel smoother for transport equations (U, k, etc.) and PBiCGStab with DILU preconditioner for elliptic equations (p, yPsi).

The approaches used for creating the mesh are divided into three parts: simple mesh, merged meshes and special cases. The tutorial cases are listed and the methods are described in detail in each section bellow.

1.2.1 Simple mesh

The blockMesh utility alone is used for creation of the mesh. In blockMeshDict user can found two blocks that are overlapping each other. This method is used for simpleRotor tutorial which is described in detail in section 1.1.

The same setup for the mesh (except dimensions – size of the computational domain) is used for another two cases: heatTransfer and movingBox. The main difference here is that the prescribed movement of the hole is linear and the case is solved by different solver. The name, solver and the path are displayed in table 1.2.

Case	solver	path
simpleRotor	pimpleDyMFoam	`$FOAM_TUTORIALS/incompressible/overPimpleDyMFoam/`
heatTransfer	overLaplacianDyMFoam	`$FOAM_TUTORIALS/basic/overLaplacianDyMFoam`
movingBox	overBuoyantPimpleDyMFoam	`$FOAM_TUTORIALS/heatTransfer/overBuoyantPimpleDyMFoam`

Table 1.2: Basic information about tutorial cases for simple mesh

A slight modification of simpleRotor mesh can be found in cases named twoSimpleRotor. According to the name, a second rotor is added into the mesh. This task is accomplished in the exact same way as the first rotor was added, so in blockMeshDict the second movingZone is added. The hole defined in topoSet is extended by second coordinates and the second oversetMesh zone is defined by c2 cell set. The cases where this method is applied are listed in table 1.3.

Case	solver	path
twoSimpleRotors	overRhoPimpleDyMFoam	`$FOAM_TUTORIALS/compressible/overRhoPimpleDyMFoam`
twoSimpleRotors	overInterDyMFoam	`$FOAM_TUTORIALS/multiphase/overInterDyMFoam/`
twoSimpleRotors	overPimpleDyMFoam	`$FOAM_TUTORIALS/incompressible/overPimpleDyMFoam`

Table 1.3: Basic information about tutorial cases twoSimpleRotors

1.2.2 Merged meshes

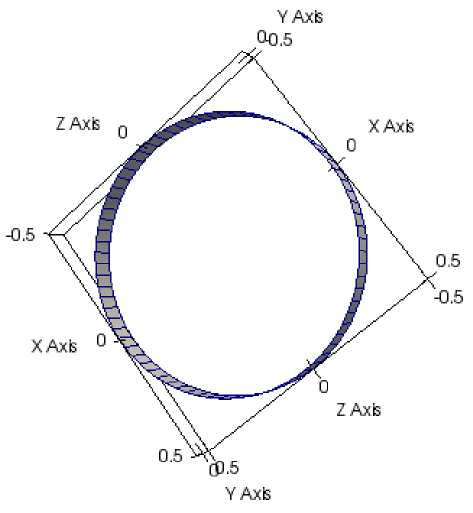
Another way to create overset mesh is to use OpenFOAM utility mergeMeshes. This approach is used in case cylinder, which can be found for following solvers listed in table 1.4.

Case	solver	path
hotCylinder	overRhoSimpleFoam	`$FOAM_TUTORIALS/compressible/overRhoSimpleFoam/`
cylinder	overPotentialFoam	`$FOAM_TUTORIALS/basic/overPotentialFoam/`
cylinder	overPimpleDyMFoam	`$FOAM_TUTORIALS/incompressible/overPimpleDyMFoam/`

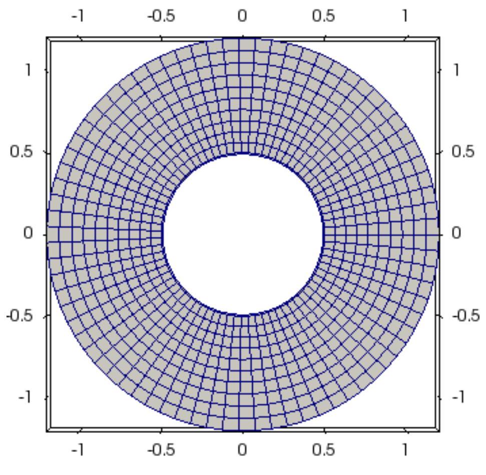
Table 1.4: Basic information about tutorial cases cylinder

The case folder contains two another case folders: cylinderMesh, cylinderAndBackground plus Allrun and Allclean scripts. First the mesh around cylinder is created, so the folder cylinderMesh is entered. No zero folder is present because this is just for creating the mesh around the cylinder.

The creation of the mesh is not governed by blockMeshDict. The cylinder is predefined surface saved in `constant/triSurface`. It can be displayed in paraview and it is shown in figure 1.4a. The mesh is created by command `extrudeMesh` and the result is displayed in figure 1.4b.



(a) Cylinder surface



(b) Mesh around cylinder

Figure 1.4: Cylinder surface and mesh created around it

The shortened file extrudeMeshDict for command `extrudeMesh` with only the necessary commands is printed below. If any change is needed to be made by the user, the whole file can be found in the tutorial case with all the options.

```
constructFrom surface;

surface "constant/triSurface/cylinder.vtk";
```

```
// Flip surface normals before usage. Valid only for extrude from surface or
// patch.
flipNormals false;

//- Linear extrusion in point-normal direction
extrudeModel        linearNormal;

nLayers             10;

expansionRatio      1.02;


linearNormalCoeffs
{
    thickness       0.7;
}

mergeFaces false;

// Merge small edges. Fraction of bounding box.
mergeTol 0;
```

This file specifies that the surface is cylinder.vtk and its location in file structure. The extrude model should be linearNormal, which mean that the extrudion should be made in normal vector direction. The linearNormal model really simplifies number of parameters that have to be specified. Parameter nLayers stands for number of elements along the normal vector. The thickness affects the length of mesh in normal vector direction.

The last task in the cylinderMesh file is to name the patches. This is made by

```
createPatch -overwrite
```

The file createPatchDict is specified. The extrudeMesh named the patches sides, originalPatch and otherSide. The createPatchDict rename this patches as overset (before otherSide), walls (before originalPatch) and frontAndBack (before sides).

The tutorial continues in second file of the case cylinderAndBackground. The mesh of background is created by blockMesh. The two meshes are merged by

```
mergeMeshes . ../cylinderMesh -overwrite
```

which merges both meshes and overwrites the existing mesh in current folder. The overset mesh is created and the zones ID are determined in the same way as was described for simpleRotor tutorial 1.1.

1.2.3 Special cases

In this section special cases from mesh perspective will be described. The list of them is provided in table 1.5.

Case	solver	path
boatAndPropeller	overInterDyMFoam	`$FOAM_TUTORIALS/multiphase/overInterDyMFoam/`
floatingBody	overInterDyMFoam	`$FOAM_TUTORIALS/multiphase/overInterDyMFoam/`
aeroFoil	overSimpleFoam	`$FOAM_TUTORIALS/incompressible/overSimpleFoam/`

Table 1.5: Basic information about tutorial cases with complex mesh settings

The case boatAndPropeller is just an extension of simpleRotor tutorial, but it is 3D case and it is expensive in terms of memory and computational time. The princip for overset mesh is the same. One mesh is created by blockMesh. The mesh is divided into 4 parts: background, hullBox, propeller and rudder. The overset is set for each of the part except background mesh.

After the creation of the mesh, the refineMesh is used on each of the overset mesh. The name of cellSet in refineMeshDict stays the same, the cellSet name is set by topoSet, so in Allrun.pre script, these two utilities are called multiple times with different files. After the refinement is done, the topoSet is used for setting the zones for overset, which was already explained.

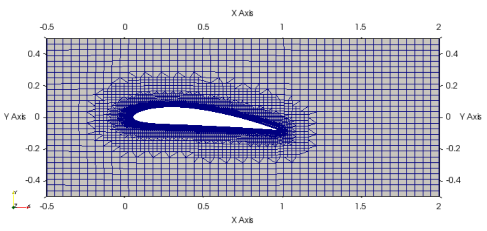
The floatingBody is a 3D case. Its structure is similar to the cylinder case, which is described in section 1.2.2. The last tutorial is aerofoil. This case uses snappyHexMesh, whose description is not in focus of this project, so it is not described here.

The structure of this case is similar to the cylinder 1.2.2, but it is a litle bit complicated. The case contains four folders listed in order for running.

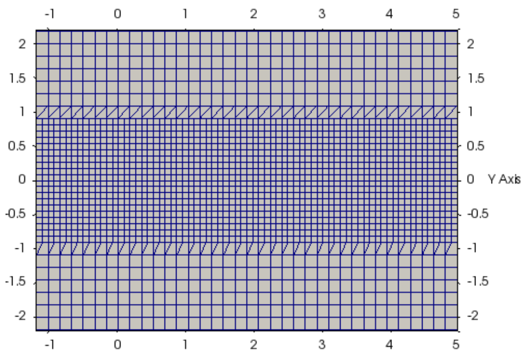
- aeroFoil_snappyHexMesh
- aeroFoil_overset
- background_snappyHexMesh
- background_overset

In the first folder, the three dimensional mesh with aeroFoil is created. In aeroFoil_overset folder, the command extrudeMesh creates a mesh from source patch named symFront. And the names of patches are overwriten by createPatch. This mesh is displayed in figure 1.5a.

The same trick is repeated with background mesh, which is displayed in figure 1.5b. The meshes are merged together.



(a) Aerofoil mesh



(b) Background mesh

Figure 1.5: SnappyHexMesh for aerofoil and background before merge

These special cases are various extensions (three dimensions, refinements of mesh or snappyHexMesh utillity) but the principle of overset was the same as for the discussed cases.

Comment: Tips and tricks

For overset to work, the user has to set up the zone ID. That is accomplished in tutorials by topoSet and setFields utilities, so these two files are neded. This ID is automatically set up by running checkMesh. This can save some time, but it does not have to work in each case.

1.3 Modification of twoSimpleRotors

The tutorial twoSimpleRotors is modified to be more general. The concept of one blockMesh for all mesh parts is transformed into separate files for each mesh, that is used for cylinder tutorial case. The case is copied into user folder by executing

```
cp -r $FOAM_TUTORIALS/incompressible/overPimpleDyMFoam/twoSimpleRotors/ $FOAM_RUN
```

The case folder after modification is (can be obtained by command `tree`)

```
-- Allclean
-- Allrun
-- background
|   -- 0.orig
|   |   -- epsilon
|   |   -- k
|   |   -- nut
|   |   -- p
|   |   -- pointDisplacement
|   |   -- U
|   |   -- zoneID
|   -- Allclean
|   -- Allrun
|   -- Allrun.pre
|   -- constant
|   |   -- dynamicMeshDict
|   |   -- transportProperties
|   |   -- turbulenceProperties
|   -- system
|       -- blockMeshDict
|       -- controlDict
|       -- decomposeParDict
|       -- fvSchemes
|       -- fvSolution
|       -- setFieldsDict
|       -- topoSetDict
-- rotorOne
|   -- Allclean
|   -- Allrun.pre
|   -- constant
|   -- system
|       -- controlDict
|       -- blockMeshDict
-- rotorTwo
    -- Allrun.pre
    -- constant
    -- system
        -- blockMeshDict
        -- controlDict
```

In comparison with original twoSimpleRotors, the folders background, rotorOne and rotorTwo are created. The background folder is the main forlder and the computation is executed here. The folders rotorOne and rotorTwo are similar, so just rotorOne is discussed here. The Allrun.pre script containts one command `runApplication blockMesh`. The only purpose of this command is to create meshes. The blockMeshDict was modified to create just one block – the overset one.

After the meshes are created for overset rotors, the background folder is entered. The blockMesh here was also adjusted to create just the background mesh. The Allrun.pre script was modified as

```
runApplication blockMesh

#merge meshes
runApplication -s 1 mergeMeshes . ../rotorOne/ -overwrite
runApplication -s 2 mergeMeshes . ../rotorTwo/ -overwrite

# Select cellSets
runApplication -s 1 topoSet

runApplication subsetMesh box -patch hole -overwrite

# Select cellSets
runApplication -s 2 topoSet

restore0Dir

# Use cellSets to write zoneID
runApplication setFields
```

The case can be run from case folder by command `./Allrun` and can be cleaned up by `./Allclean`.

Chapter 2

Theory

The overset framework in OpenFOAM is a generic implementation for the use of disconnected (also called Chimera) meshes. Both static and dynamic meshes are suported. The meshes (background and overset) are not connected. The overset approach is particularly useful for cases with mesh motion and interactions. It avoids the problems and instabilities associated with deforming meshes, but according to Houzeaux [1] it can be also used in following cases such as

Simplified mesh generation: It could be easier to generate not a whole mesh, but split the domain into more parts that are simpler for meshing. In this way more suitable elements can be used, for example rectangles instead of triangles.

Local refinement: The example for this application is given in overSimpleFoam tutorial, where the mesh around cylinder is refined and the background mesh is simply generated from one block (see 1.2.2).

Moving parts: This application is clearly visible in the twoSimpleRotor tutorial, where two meshes move independently. Rotation of a single rotor could be simulated using the Arbitrary mesh interface (AMI) functionality, but the interaction of two rotors could be otherwise done only by remeshing.

Optimization: This could be used for moving various parts of mesh without the need of remeshing the domain.

The basic principle of the overset method is to solve the governing partial differential equations on both the background mesh and the overset mesh. Within the background domain, the elements corresponding to the interior of the overset domain are marked as holes and removed from the computational domain. The values on the boundary of the overset domain are then interpolated to the background mesh. Similar procedure is realized for the overset domain.

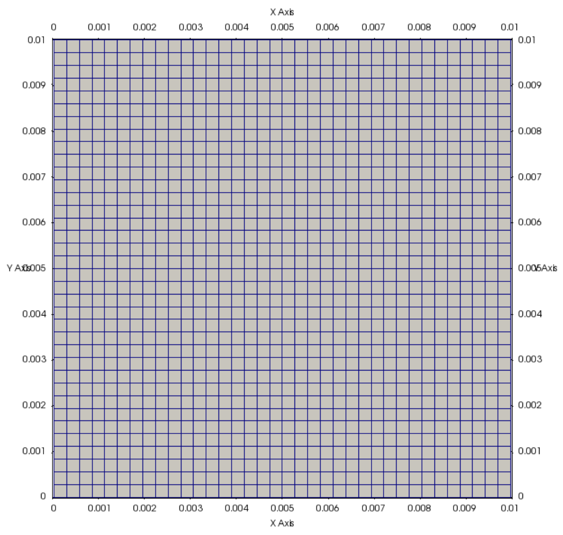
As a result, in each timestep, each cell of the background and overset mesh is marked as one of the following types

Calculated: For this type of cells, the equations are solved.

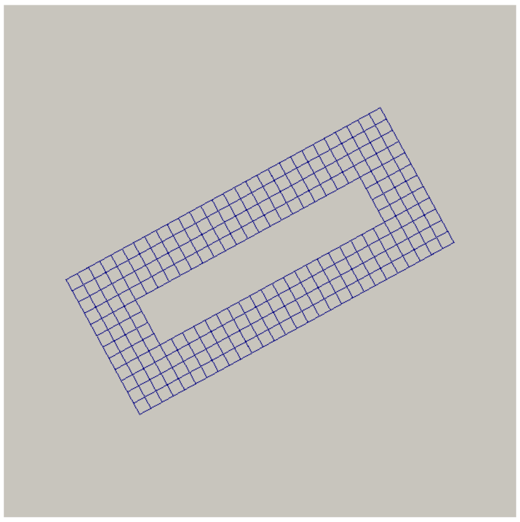
Interpolated: The values in these cells are computed by interpolation from the nearest elements of the second domain (background elements for the overset elements, and vice versa).

Holes: No solution is computed here.

The type of mesh (zone ID – background or overset) is specified in case folder, usually by running topoSet and setFields utility. The whole domain is divided into the three parts mentioned above, one of them, the holes, does not have to be used. The zones for the simpleRotor tutorial (see 1.1) are displayed in figures 2.2a and 2.2b. For better orientation, the separate meshes are displayed in figures 2.1a for the background mesh and 2.1b for overset mesh.

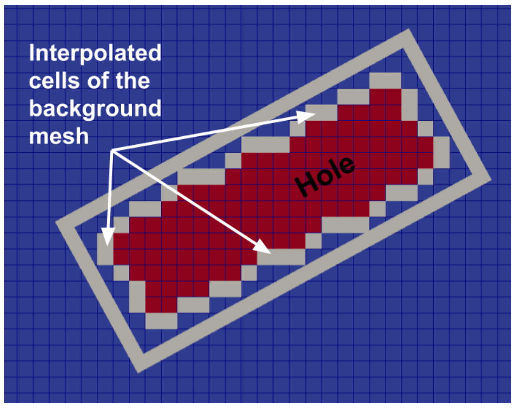


(a) Background mesh

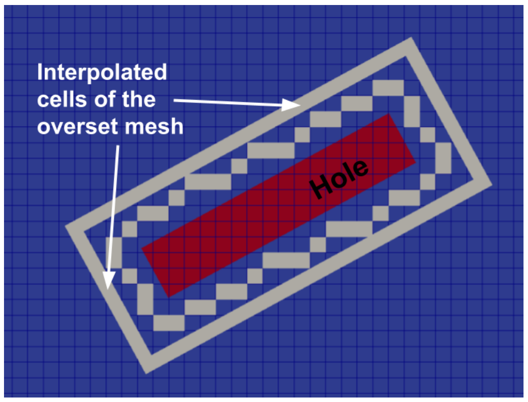


(b) Overset mesh

Figure 2.1: Meshes for simple rotor tutorial case



(a) Hole (red cells) and interpolated cells (white cells) defined for background mesh



(b) Hole (red cells) and interpolated cells (white cells) defined for overset mesh

Figure 2.2: Display of interpolation cells (white) for simpleRotor tutorial

The zones are distinguished by color. The zone zero is blue and it is meant to be calculated. The red zone is for holes and it differs for each mesh. Each mesh contains interpolation cells. The cells around the rotor (the hole in overset mesh) belong to the background mesh and their donors are from overset mesh. It can be seen that the cells are oriented in way of background mesh orientation. The white cells in the end of mesh for the rotor belong to the overset mesh and their donors are found in the background mesh. The meshes are independent and the values are influenced through the interpolation cells only.

The task of finding and assigning the right cells to right types is a technical issue. The interpolation can be realized by various interpolation schemes

- cellVolumeWeight
- inverseDistance
- leastSquares
- trackingInverseDistance

The principles of inverseDistance are described in section 2.1.

2.1 Inverse distance

The acceptor cells are the ones that are interpolated. Their donors are calculated cells which provide the correct values that are interpolated into acceptors cells. Donors are determined from the nearest cells by mesh indices. Then they are checked for better (closest) donors. For one acceptor the donor is determined as its closest neighbour. The values from donors are interpolated into the acceptor element. This interpolation can be found in in the markDonors() function, which uses the betterDonors() function to check if the current cell is better donor for target than the the one already set.

The weights for interpolation are determined from distances of cell centers (this is explained in subsection stencilWeights 3.3.14). The situation for one acceptor is displayed in figure 2.3.

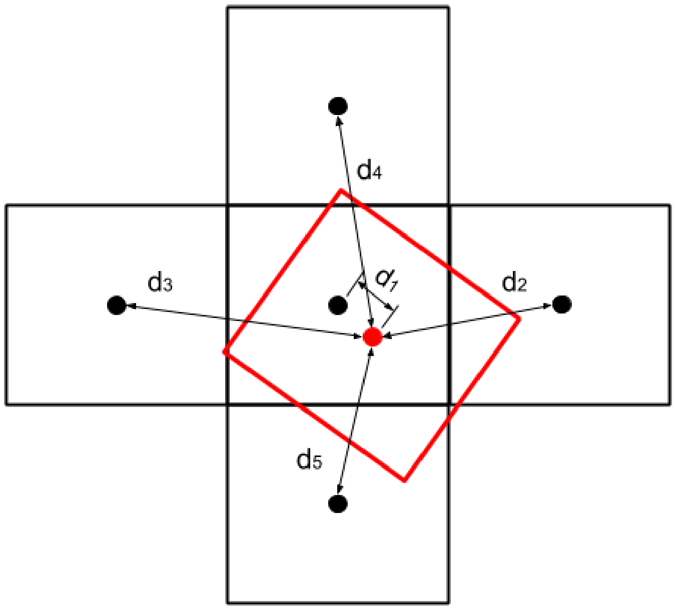


Figure 2.3: Distance of cell centers for determination of the weights, the red cell is acceptor and the black ones are donors

To determine the weights the sum S of of their inverse distances is needed

$$S = \sum_{i}^{n} \frac{1}{|d_i|}, \tag{2.1}$$

where n is number of donors and d_i is the distance between their centers and center of aceptor. Then the weights w_i are

$$w_i = \frac{\frac{1}{|d_i|}}{S}. \tag{2.2}$$

Finally, the interpolated value is obtained for one cell by sum over all neighbours

$$\varphi = \sum_{i}^{n} w_i \varphi_i, \tag{2.3}$$

where φ is the field that is interpolated, for example the pressure.

2.1.1 Logarithm inverse distance

The inverse distance interpolation scheme is changed in section 4.1.3. The weights computation is amended. According to theory of differential equations which can be found in Evans [2] the fundamental solution of laplace differential equation for two dimensional case is logarithm.

Chapter 3

Description of the overset library

This chapter is divided into two main parts. The first part of the chapter documents the differences between the implementation of the standard simpleFoam solver, and its overset variant called overSimpleFoam. The second part focuses on detailed description of used functions in overSimpleFoam with interpolation scheme inverseDistance. For better orientation in report text, the reader is advised to follow the contents of discussed files.

The overset method in OpenFOAM is implemented in an implicit, fully parallel way. The interpolation (from donor to acceptor) is inserted as an adapted discretisation on the donor cells, such that the resulting matrix can be solved using the standard linear solvers.

The overset method is implemented in OpenFOAM as a library. The source code can be found in folder `$FOAM_SRC/overset`.

3.1 Structure of the library

The overset library contains 9 directories.

- cellCellStencil
- dynamicOversetFvMesh
- fvMeshPrimitiveLduAddressing
- include
- lduPrimitiveProcessorInterface
- oversetAdjustPhi
- oversetPolyPatch
- regionsToCell
- Make

The Make folder contains information for the compiler. It specifies the name of the library and connects all header files used in the library. All of the other directories contain header and source files.

3.2 Overset and simpleFoam solver

In this section the differences in implementation between simpleFoam and overSimpleFoam are discussed. The code can be compared by executing following commands in OFv1906 environment.

```
cd $FOAM_APP/solvers/incompressible/simpleFoam/
diff -y simpleFoam.C overSimpleFoam/overSimpleFoam.C
```

3.2.1 overSimpleFoam.C

The main difference for this file is in the include statements. Before main function beginning, the following statements are added.

- `#include "dynamicFvMesh.H"`
- `#include "cellCellStencilObject.H"`
- `#include "localMin.H"`
- `#include "interpolationCellPoint.H"`
- `#include "fvMeshSubset.H"`
- `#include "transform.H"`
- `#include "oversetAdjustPhi.H"`

Because the statements are added before declaration of the main function, they do nothing. They just contain declarations of functions that are used in further code. Brief description of header functions that do not directly belong into overset library is provided below.

The dynamicFvMesh.H is added because the simpleFoam can not be used on dynamic meshes and the mesh for overset can be dynamic. The localMin.H belongs into finiteVolume family and it is interpolation scheme for faces. It returns a field of values for each of faces. The value is the smallest value from either neighbor or owner cell. The header file interpolationCellPoint.H has one function: interpolate. This function takes cell center and point (vertex) value and and decomposes them into tetrahedron and does the linear interpolation within them. The fvMeshSubset.H is associated with dymanicMesh. Given the original mesh and the list of selected cells, it creates the mesh consisting only of the desired cells, with the mapping list for points, faces, and cells. The last header file transform.H contains transformation operations for 3D tensors.

Compared to the include statements of the standard simpleFoam solver, the overSimpleFoam implementation has following extra entries:

- `#include "setRootCaseLists.H"`
- `#include "createUpdatedDynamicFvMesh.H"` (subsection 3.2.2)
- `#include "createFields.H"` (subsection 3.2.3)
- `#include "createOversetFields.H"` (subsection 3.2.4)
- `#include "createFvOptions.H"`

The files setRootCaseLists.H and createFvOptions.H have no close connection to the overset library, so they are not described in this report. The simple loop itself looks the same for simpleFoam as for overSimpleForm. The differences are hidden inside files for solving equations: UEqn.H and pEqn.H. These files are described in separate subsections 3.2.5 and 3.2.6.

3.2.2 createUpdatedDynamicFvMesh.H

This file creates the dynamic mesh with initial mesh to mesh mapping. The IOobject `meshPtr` is created. The pointer `mesh` is introduced and assigned to `meshPtr`. The mapping is done by single function `mesh.update()`. The function `update` for dynamicFvMesh is discussed in detail.

The `update` is declared in dynamicFvMesh.H as

```
virtual bool update() = 0;
```

This means that it is a member function, so it is supposed to be defined in derived classes. The classes that contain the definition of this function are implemented in

- topoChangerFvMesh
- dynamicOversetFvMesh
- dynamicRefineFvMesh
- movingConeTopoFvMesh
- mixerFvMesh
- linearValveFvMesh
- linearValveLayersFvMesh
- dynamicMotionSolverFvMesh
- dynamicInkJetFvMesh
- staticFvMesh
- rawTopoChangerFvMesh
- dynamicMultiMotionSolverFvMesh
- dynamicMotionSolverTopoFvMesh
- dynamicMotionSolverListFvMesh

In this case the derived class is dynamicMotionSolverFvMesh. This class contains the function update and it is virtual as well (it has the same definition). From this class the derived class is dynamicOversetFvMesh. The correct implementation for the update function is chosen by user in the dictionary constant/dynamicMeshDict by entering the dynamicFvMesh solver as dynamicOversetFvMesh.

The update function is defined in dynamicOversetFvmesh.C at line 588, which can be found in file

```
$FOAM_SRC/overset/dynamicOversetFvMesh/dynamicOversetFvMesh.C
```

Its purpose is to call two functions

- `updateAddressing();`
- `interpolateFields();`

which are part of the overset library. The first function calculates the extended addressing for lduMatrix. Some extra faces has to be added due to the overset. And the corrections for parallel computing are made. The second function creates the interpolation for appropriate fields for selected cells. If these operations are done correctly, the bool true is returned. Special section is devoted to detailed descriptions of implementations of important functions from the oveset library. This two functions can be found in sections 3.3.5 and 3.3.6.

3.2.3 createFields.H

The difference between these files can be obtained by

```
cd $FOAM_APP/solvers/incompressible/simpleFoam/
diff -y createFields.H overSimpleFoam/createFields.H
```

The only difference is include statement for createFvOptions.H file. For the overSimpleFoam solver, this file is included in the overSimpleFoam.C file. The reason is that fields for overset must be introduced before the fvOptions are created.

3.2.4 createOversetFields.H

This file has no partner in simpleFoam to be compared with. In the beginning of the file the variable nonInt, type wordHashSet, is created and set. The type meant that the variable nonInt is an associative container that contains set of unique objects of type Key. The nonInt variable contains the name of fields that are not supposed to be interpolated anywhere in the computational process. The definition is shown below.

```
{
    wordHashSet& nonInt =
        const_cast<wordHashSet&>(Stencil::New(mesh).nonInterpolatedFields());

    nonInt.insert("HbyA");
    nonInt.insert("grad(p)");
    nonInt.insert("surfaceIntegrate(phi)");
    nonInt.insert("surfaceIntegrate(phiHbyA)");
    nonInt.insert("cellMask");
    nonInt.insert("cellDisplacement");
    nonInt.insert("interpolatedCells");
    nonInt.insert("cellInterpolationWeight");
}
```

The file continues with two include statements

```
#include "createCellMask.H"
#include "createInterpolatedCells.H"
```

These two files are part of the overset library. They are specified in detail in subsection 3.3.3 and 3.3.4. In short they create scalar fields for hole and interpolated cells and they also set their values. The end of the file consists of these lines:

```
bool adjustFringe
    ( simple.dict().lookupOrDefault("oversetAdjustPhi", false) );
bool massFluxInterpolation
    ( simple.dict().lookupOrDefault("massFluxInterpolation", false) );
```

The purpose of them is to define the way how specific functions are supposed to be handled. In the case folder, the user can choose from various ways or let them be handled by default settings. These two variables are used in file pEqn.H which is described in subsection 3.2.6.

3.2.5 UEqn.H

The comparison of the two files can be made by following commands.

```
cd $FOAM_APP/solvers/incompressible/simpleFoam/
diff -y UEqn.H overSimpleFoam/UEqn.H
```

Two changes are made in UEqn.H file for overSimpleFoam. The right part of velocity equation is multiplied by cellMask

```
solve(UEqn == -cellMask*fvc::grad(p));
```

The field cellMask has value one for each mesh cell except the hole. So if the hole is present the right part of the equation is equal zero. The second change is that the correction for velocity field `fvOptions.correct(U);` is made independent of the usage of the momentum predictor.

3.2.6 pEqn.H

The comparison of the two files can be made by following commands.

```
cd $FOAM_APP/solvers/incompressible/simpleFoam/
diff -y pEqn.H overSimpleFoam/pEqn.H
```

The first step is to create a mask for the faces from the cellMask interpolation.

```
surfaceScalarField faceMask(localMin<scalar>(mesh).interpolate(cellMask));
```

The field of diagonal coefficients of the matrix resulting from the discretisation of the momentum equation rAU is introduced. The surface scalar field rAUf which was interpolated from rAU is multiplied by faceMask. The principle is the same as for previous case – to introduce the hole faces into system.

```
    volScalarField rAU(1.0/UEqn.A());
    surfaceScalarField rAUf("rAUf", faceMask*fvc::interpolate(rAU));
```

The same principle is applied for HbyA (matrix H divided by A), but of course it is multiplied by cellMask field. The file interpolatedFaces.H is located in overPimpleDyMFoam code file. This file is not used during overSimpleFoam simulation because the field specified in file createOversetFields.H (subsection 3.2.4) is set to "false". The same goes for adjustFringe, which turns off the special setting for adjusting phi by `oversetAdjustPhi(phiHbyA, U);`.

Non-orthogonal pressure corrector loop is the same for both solvers. The last difference is in momentum corrector, where the part of computation of velocity field is also multiplied by cellMask.

```
U = HbyA - rAU*cellMask*gradP;
```

3.3 Selected parts of the overset library

This section is devoted to description of selected parts of the overset library. The described content is used for the inverse distance interpolation scheme and the call of triggering function update() is made in overSimpleFoam.

First the addressing is set, so for example cellStencil private variable contains a list of neighbors for each cell that is supposed to be interpolated. This neighbors have the same index as the cell they belong to. For each field that is supposed to be interpolated, the function interpolate is called. This function also corrects the boundary condition and calls the functioncellInterpolationMap which calls function update.

The function update takes care of the final interpolation and calls another function. Its purpose is to declare valid donors, set interpolated, calculated and hole cells. It also calls globalCellCells and stencilWeights function. The first one sets valid donors for each acceptor and the second one creates the weights. The weights are used in function interpolate to finalize the calculation.

3.3.1 Class cellCellStencil

This class is determined for calculation of interpolation stencils. The definition and declaration is located in `$FOAM_SRC/overset/cellCellStencil/cellCellStencil/`.

The class has public and protected variables. The public are two enumerated types. They are a data type used in computer programming to map a set of names to numeric values. Enumerated data type variables can only have values that are previously declared. First one is patchCellType and the possibilities are

OTHER value 0: everywhere except boundary and overset boundary

PATCH value 1: defines uncouplet boundary

OVERSET value 2: defines overset boundary

The second public variable cellType can have one of the following values

CALCULATED value 0: normal computation on this cells

INTERPOLATED value 1: interpolation is needed

HOLE value 2: the hole is present

The protected data are related to the cellType variable defined in public section, the reference to mesh is also created and set of fields that should not be interpolated, which is type of wordHashSet. Two constructors are introduced. The first one creates cellCellStencil from fvMesh. The second does the same and returns the pointer for the object. The function declared as

```
virtual const labelUList& cellTypes() const = 0;
```

which is a virtual function whose purpose is to return the cell type list. The data type label is integer of size 64 or 32, the size is specified by preprocessor macro WM_LABEL_SIZE.

3.3.2 Class cellCellStencilObject

The definition and declaration is located in `$FOAM_SRC/overset/cellCellStencil/cellCellStencil/`. First the class declares new data type: Stencil.

```
class cellCellStencilObject;
typedef MeshObject
<
    fvMesh,
    Foam::MoveableMeshObject,
    cellCellStencilObject
> Stencil;
```

The Stencil is a mesh object, who consists of fvMesh, MoveableMeshObject and cellCellStencilObject. This class inherits from class Stencil and class cellCellStencil. A pointer is declared as a private variable stencilPtr_.

Construction of the cellCellStencilObject is introduced. The mesh is needed for creation. The pointer for the object is returned from creation of cellCellStencil and the pointer is assigned to the private pointer created before. During the construction of the object, the interpolation method is set by looking up the phrase ”oversetInterpolation” in system dictionary.

After the constructor and destructor are introduced, list of virtual functions follows. For example: update, movePoints, cellSTencil and so on.

Each of this virtual functions are implemented in class that inherits from this one. As example the different implementation of function update() can be found in classes: inverseDistance, cellVolumeWeight and trackingInverseDistance. The reason is that all updates of stencil are made differently for each of the approaches.

3.3.3 File createCellMask.H

This file can be found in `$FOAM_SRC/overset/include` together with createInterpolatedCells.H (3.3.4). The field cellMask is created as IOobject. The field is dimensionless scalar and its purpose is to define the hole mesh for interpolations. This file also contains include statement

```
#include "setCellMask.H"
```

The file setCellMask.H is also part of the include folder. It sets all the cell in cellMask scalar field to zero value if the cellType of the cell in question is defined as HOLE.

3.3.4 File createInterpolatedCells.H

This file can be found in `$FOAM_SRC/overset/include` together with createCellMask.H (3.3.3). The field interpolatedCells is created as IOobject. The field is dimensionless scalar and its purpose is to define the mesh cells for interpolations. This file also contains include statement

```
#include "setInterpolatedCells.H"
```

The file setInterpolatedCells.H is also part of the include folder. It sets all the cell in interpolatedCells scalar field to zero value if the cellType of the cell in question is defined as INTERPOLATED.

3.3.5 Function updateAddressing

The function declaration is in file dynamicOversetFvMesh.H. The definition can be found in file dynamicOversetFvMesh.C, lines 44 – 357. The location of this two files is as follows.

```
$FOAM_SRC/overset/dynamicOversetFvMesh
```

The purpose of this function is to set addressing for parallel and local computation, to hand out the interfaces between processors and to find out if one processor cares about the same face as other processor. The function updateAddressing takes no arguments and returns bool. The first line

```
const cellCellStencilObject& overlap = Stencil::New(*this);
```

creates or addresses existing mesh object Stencil to have a pointer named overlap. The data type of Stencil is meshObject, the data type is derived from templated abstract base-class. The Stencil is defined in file cellCellStencilObject.H (see 3.3.2). The variable overlap is supposed to be type cellCellStencilObject.

The label list named stencil is created by calling function cellStencil with no argument. This function is part of cellCellStencilObject.

```
 const labelListList& stencil = overlap.cellStencil();
```

The function cellStencil() is part of inverseDistanceCellCellStencil.H and its definition is at line 313.

```
virtual const labelListList& cellStencil() const
{
     return cellStencil_;
}
```

Its purpose is to return a private variable cellStencil_. This variable is a list of cells where the neighbors for cell meant to be interpolated are listed. Index of the list is relevant to concrete cell. The following example shows the part of the field printed out.

```
...
0()
0()
0()
5(1382 1383 1410 1354 1381)
0()
0()
0()
0()
0()
0()
0()
0()
4(1395 1396 1367 1394)
4(1396 1397 1368 1395)
5(1369 1370 1397 1341 1368)
5(1370 1371 1398 1342 1369)
...
```

Next variable is baseAddr and it is created by

```
 const lduAddressing& baseAddr = dynamicMotionSolverFvMesh::lduAddr();
```

It creates a variable baseAddr which contains the addressing for lduMatrix. The lduMatrix contains three arrays with lower triangle, upper triangle and diagonal coefficients. Variables are declared. The integer nExtraFaces is created. One dimensional arrays of integer for lowerAddr, upperAddr are introduced. Two two integer arrays are defined with names: localFaceCells and remoteFaceCells. The constant globalNumbering of data type globalIndex is created with size of baseAddr variable.

Array globalCellIDs is constructed with size of cellInterpolationMap. This array is filled with cells ID in global addressing. This globalCellIDs is distributed into overlap. All variables that were declared so far are added into reverseFaceMap, which is protected data from dynamic motion solver for fvmesh. So the addressing was extended and the new and old number of faces are printed in debug version. The result (for simpleRotor tutorial) changes because the mesh does move.

```
Time = 0.000294118
extended addressing from nFaces:2968 to nFaces:3586 nExtraFaces:618
Time = 0.000630252
extended addressing from nFaces:2968 to nFaces:3594 nExtraFaces:626
```

The next part of this function is to hand out faces by processors. For each entry in stencil the patch and face needs to be set. The patch is a value of patch or -1 for internal faces. The face is either an internal face index or a patch face index.

The private variable stencilPatches_ size is set according to size of private variable stencilFaces_. Two arrays of dynamic lists for processor owner and dynamic processor neighbor (procOwner and dynProcNeighbour) are set. DynamicList creates one dimensional vector that resizes itself as necessary to accept the new objects. The arrays are constructed with given size, such as the number of parallel processes in run.

Now the loop is made. OpenFOAM has its own loops declared. The most used is loop over all members of an array. In this case the loop is created over all arrays in stencil variable, the index is named celli and it is declared by the loop itself. Auxiliary variables are made or changed.

```
const labelList& nbrs = stencil[celli];
stencilPatches_[celli].setSize(nbrs.size());
stencilPatches_[celli] = -1;
```

The nbrs is one dimensional array with values from stencil with index celli. The size of private variable stencilPatches_ is set according to the number of neighbors of each cell. The array is set to have value -1, which means that it is internal face.

Inner loop is created to declare the correct value for stencilPatches_. If the face of stencil in question are internal faces (`if (stencilFaces_[celli][nbri] == -1)`, the following command are meant to be executed.

```
const label nbrCelli = nbrs[nbri];
label globalNbr = globalCellIDs[nbrCelli];
label proci = globalNumbering.whichProcID(globalNbr);
label remoteCelli = globalNumbering.toLocal(proci, globalNbr);

// Overwrite the face to be a patch face
stencilFaces_[celli][nbri] = procOwner[proci].size();
stencilPatches_[celli][nbri] = proci;
procOwner[proci].append(celli);
dynProcNeighbour[proci].append(remoteCelli);
```

The global ID of neighbour is assigned into globalNbr variable. Some work for parallel processes is to be done. And the faces are overwritten to be a patch faces. The faces from stencil are added according to the processor.

The loop at line 135 assigns the values from matrix dynProcNeighbour into newly created matrix (type labelListList) named procNeighbour. A command std::move is used for this assignation. The std::move is not part of the foam namespace. This command tells the compilator that it can assign space occupied by the dynProcNeighbour to newly created field, because the original one is not used any longer.

Matrix mySendCells of integers is created by calling exchange function from the Pstream namespace. This matrix contains numbers for cells for each of the processors. The declaration is made as follows.

```
labelListList mySendCells;
Pstream::exchange<labelList, label>(procNeighbour, mySendCells);
```

The integer nbri is increased of one if the procOwner or mySendCells respectively to processor has other value that zero. According the value of nbri, the size of private array remoteStencilInterfaces_ is set. So if there are one to much or more processors to handle the cell, they are find out and the correction is set.

Next an interface is added for process if there were two processes that did handle the cell and the processor is the first one by numbering (starting with master – process number zero). Than the interface is added for the second process.

At line 258 the stencil patches are rerouted because the actual interface is known. The last step is to set correct addressing for all interfaces.

3.3.6 Function interpolateFields

This function is defined in dynamicOversetFvMesh.C at lines 618 – 640. The file can be found at following path.

```
$FOAM_SRC/overset/dynamicOversetFvMesh
```

A suppression list is introduced. All fields that are supposed not to be interpolated are added. These fields are specified in code where solver is created and they are also specified in fvSchemes file in case folder for user to add another fields. The specification for the solver are stored in file createOversetFields.H. This is described in subsection 3.2.4. The last step of this function is to call interpolation for every possible field except the suppressed ones.

```
interpolate<volScalarField>(suppressed);
interpolate<volVectorField>(suppressed);
interpolate<volSphericalTensorField>(suppressed);
interpolate<volSymmTensorField>(suppressed);
interpolate<volTensorField>(suppressed);
```

3.3.7 Function interpolate

The function interpolate is a templated function declared in dynamicOversetFvMesh.H at line 121. The functionality is set in file dynamicOversetFvMeshTemplates.C at lines 86 – 111, which is located in folder

```
$FOAM_SRC/overset/dynamicOversetFvMesh
```

The function finds out all geometric fields that are the type that was called by the function, for example: volScalarField, volVectorField, and so on. For example for overPimpleFoam in simpleRotor tutorial the interpolated fields are: nu, p, U, U_0 and the skipped ones are: cellMask, interpolatedCells and cellMask_0.

The classification according the data type is as follows:

- volScalarField:

  - nu
  - cellMask
  - interpolatedCells
  - p
  - cellMask_0

- volVectorField:
  - U
  - U_0

The interpolation function for each of these fields is called.

```
interpolate(fldPtr->primitiveFieldRef());
```

The fldPtr is a pointer to the field in questions and primitiveFieldRef() returns a reference to the field. Another interpolate function is called and the boundaries of the fields which were interpolated are corrected. The function is called at lines 77 – 82 in the same file dynamicOversetFvMeshTemplates.C.

```
template<class GeoField>
void Foam::dynamicOversetFvMesh::interpolate(GeoField& psi) const
{
    interpolate(psi.primitiveFieldRef());
    psi.correctBoundaryConditions();
}
```

The final interpolate function is the one where the interpolation is done. It is located in the same file (dynamicOversetFvMeshTemplates.C) as the two interpolate functions before, but the location is between lines 38 – 75.

First, variable overlap of type cellCellStencil and variable stencil from overlap by calling overlap.cellStencil() are created. This cellStencil() function was already discussed in 3.3.5.

Another variables are constructed.

```
const mapDistribute& map = overlap.cellInterpolationMap();
const List<scalarList>& wghts = overlap.cellInterpolationWeights();
const labelList& cellIDs = overlap.interpolationCells();
const scalarList& factor = overlap.cellInterpolationWeight();
```

These functions are called according the type of interpolation scheme. They are explained in detail in following subsections: cellInterpolationMap – 3.3.8, cellInterpolationWeights – 3.3.16, interpolationCells – 3.3.17 and cellInterpolationWeight – 3.3.18. After these functions are executed, the interpolated map is set. Each cell knows if it is supposed be interpolated, calculated or if it is a hole. Every interpolated cell (acceptor) has its donors and the donors have weights. The work is distributed for parallel processes.

The interpolation is executed as follows for all interpolated cell named cellIds with index i.

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

        const scalarList& w = wghts[celli];
        const labelList& nbrs = stencil[celli];
        const scalar f = factor[celli];

        T s(pTraits<T>::zero);
        forAll(nbrs, nbrI)
        {
            s += w[nbrI]*work[nbrs[nbrI]];
        }
        //Pout<< "Interpolated value:" << s << endl;
        //T oldPsi = psi[celli];
        psi[celli] = (1.0-f)*psi[celli] + f*s;
        //Pout<< "psi was:" << oldPsi << " now:" << psi[celli] << endl;
    }
```

The array of type double of weights is named w, array of type integer is named nbrs and stands for neighbors of current cell. The type double f holds zero or one. The zero is for calculated and the one is for interpolated cells.

The variable s is sum over all neighbors. The sum contains weight for the neighbour that is multiplied by the value of the field in questions for the relevant neighbour.

The last step from overset library is to change the field. It is done by using the factor variable. If the field should be computed, the factor is zero and no change to the field is made. If the factor is one and the field is supposed to be interpolated the new value is set as the s.

3.3.8 Function cellInterpolationMap

This function declaration can be found in three files:

- inverseDistanceCellCellStencil.H (line 302)
- cellVolumeWeightCellCellStencil.H (line 205)
- cellCellStencilObject.H (line 231)

In this report the inverse distance interpolation scheme is discussed, so the first declaration is taken in account.

```
        virtual const mapDistribute& cellInterpolationMap() const
        {
            if (!cellInterpolationMap_.valid())
            {
                const_cast<inverseDistance&>(*this).update();
            }
            return cellInterpolationMap_();
        }
```

The function update for inverseDistance interpolation stencil is called, this function is described in detail in subsection 3.3.9.

3.3.9 Function update

This function is written in file inverseDistanceCellCellStencil.C from line 1740 to 2341. Its return type is bool and it does not take any arguments. It returns false if nothing changed.

```
bool Foam::cellCellStencils::inverseDistance::update()
```

Since this file is really long it is advised to follow the code.

After various variables are declared, the determination of zone meshes and bounding boxes is set. Loop over all mesh parts creates new sub meshes with zone ID. Early evaluation of mesh dimension is triggered in case there are no cells in mesh. If the submesh has any point the boundary box for process in question is added from list. In other way the bounding box is created. The local bounding boxes are moved per indexing for mesh.

Next step is to determine patch for bounding boxes. They can be either global and provided by user or processor-local as a copy of the mesh bounding box. It is possible for all zones to have the same bounding box.

Variable globalDivs of data type three dimensional integer is created. It is read from file, which is specified by user under ”searchBoxDivisions”. Variable patchDivisions is assigned with this value. If the value is not specified by user, it is determined from mesh. If they are not defined by user, the following process is used. First the dimension of the case is determined. The variable type vector dim is made by calling function geometricD() from mesh as

```
const labelVector& dim = mesh_.geometricD();
```

So the dim is a vector with values 1, that indicates unconstrained direction and -1 indicating a constrained direction.

Variable nDivs is filled with

- number of mesh cells for 1D
- square root as double from mesh cells for 2D
- cubic root as double from mesh cells for 3D

Vector v is filled with values ndivs `labelVector v(nDivs, nDivs, nDivs);`.

The check is done. If there are no values -1 in dim, the v is set as one. And the patchDivision is v, which are the borders of searchBoxDivisions.

The patchParts are set with patchDivisions that are multiplied for each of direction for zone and the boundaries are marked. The printing statement is introduced at line 1902.

```
Info<< type() << " : detected " << nZones  << " mesh regions" << endl;
Info<< incrIndent;
forAll(nCellsPerZone, zoneI)
{
     Info<< indent<< "zone:" << zoneI
     << " nCells:" << nCellsPerZone[zoneI]
     << "  voxels:" << patchDivisions[zoneI]
     << " bb:" << patchBb[zoneI][Pstream::myProcNo()]
     << endl;
}
Info<< decrIndent;
```

The new boundaries according to the sub meshes were marked.

One dimensional array allCellTypes with length of number of cells and values of CALCULATED is set. So the array allCellTypes for simpleRotor tutorial is 1552 members with value 0. The second is allStencil which is matrix of 1552 arrays with array filled with zero. The last declared is array AllDonorID with length of number of cells in mesh minus one.

Data type globalIndex is used for variable globalCells constructed with number of mesh cells. The globaIndex data type is a unique integer. A buffer pBufs is constructed for inter-processor communications.

The loop for marking holes in allCellTypes starts at line 1930. Two for loops over meshParts size creates srcI (source index) and tgtI (target index) integers as counter. The function markPatchesAsHoles is used. This function is described in subsection 3.3.10.

Donors are found. This loops starts at line 1967. The counters scrI and tgtI are used again. In the loops, the function markDonors (subsection 3.3.12) is called.

Next part of the code sets all cell types where is overset and no HOLE to type INTERPOLATED (require interpolation). If there are no donors, the cells can be HOLE or CALCULATED. Now it is set to be HOLE.

At line 2058 the check is present. Variables allCellTypes and private cellTypes_ are compared. If allCellTypes is CALCULATED and the relevant cellType_ is hole, two possibilities are presented. If the allStencil size is not zero the allCellType is reset to HOLE and the allStencil is cleared. Otherwise the allCellType is set to INTERPOLATED. The walkFront function is called, found purpose is to surround holes with layers of interpolated cells.

The stencil in compact notation is set for interpolated cells. The size of array for interpolation weights is set according the mesh size and the value is 1, but it will be overwritten. The boundary field for interpolation weights array is corrected.

The final act of the overlap function except printing some statements is to call createStencil function. This function is described in subsection 3.3.13. Its purpose is to extend stencil to get inverse distance weighted neighbors.

3.3.10 Function markPatchesAsHoles

This function is written in file inverseDistanceCellCellStencil.C from line 312 to 364.

This function checks if source cell patch boundary box overlaps the target patch boundary box. If this is true, the bool function overlaps (subsection 3.3.11) is called. This function gets the information about source cell and boundary box called cBb from target cell. If true value is returned from overlap function, allCellTypes of this cell index is assigned as HOLE.

The rest of the function is meant for parallel computing with the same principle as the first part dedicated to one processor or local processor computing.

3.3.11 Function overlaps

This function is written in file inverseDistanceCellCellStencil.C from line 268 to 309.

This function tests given boundary boxes, if they overlap by looping from minimal index to maximal one. If they do overlap and the patch type of source cell is PATCH, the function returns true, otherwise false is returned.

3.3.12 Function markDonors

This function is implemented in file inverseDistanceCellCellStencil.C from line 480 to 656.

Its purpose is to determine donors for all target cells. The addressing is calculated by function calculate from namespace waveMethod. By this the variable tgtToScrAddr is created from target sub mesh and source one. The loop over all target cell map is created. Variable srcCelli is equal to the addressing with index relevant to target cell. If this variable is not equal to minus one and its type is not HOLE the donor is found. The mesh is check for better donor. The final donor is set.

The rest of the function is meant for parallel computing with the same principle as the first part dedicated to one processor or local processor computing.

3.3.13 Function createStencil

This function is written in file inverseDistanceCellCellStencil.C from line 1467 to 1648. Its purpose is to extend stencil to get inverse distance weighted neighbors.

The first step is the creation of bool list with name isValidDonor, all values are set to true and the size of the list is number of cells. Some members of this field are set to be false if the type of cell is equal to HOLE. The HOLE can not be a valid donor. If the acceptor has been already handled, it is marked in special array of interpolation cells size which is named doneAcceptor and its data type is bitSet. The bitSet stores bits – elements with two states. It behaves lartely like a list and supports variety of bit-set operations.

```
bitSet doneAcceptor(interpolationCells_.size());
```

Endless cycle while (true) is set. This cycle goes to the end of this function with exception of function that does the map distribution once again. The setting of donorAcceptor is checked for all interpolation cells. If the variable have not been set yet, the fields are rewrittens to delete any old contents. Field of points is created with number of elements which is equal to construct size of cell interpolation map and the value is set for each of the point as 3D values GREAT, which are a great values, namely 1e+15.

```
 pointField samples(cellInterpolationMap().constructSize(), greatPoint);
```

In short the code from line 1509 to line 1534 overwrites the field named samples with values according to

```
  minMagSqrEqOp<point>()(samples[elemI], cc);
```

This function takes two arguments and returns void. It alters first parameter according the condition in its name.

```
  EqOp(minMagSqrEq, x = (magSqr(x) <= magSqr(y) ? x : y))
```

So the smaller magnitude of sqrt is written into samples variable for the cell in question. The slot represents the cell stencil. The other variable is cell center. The cell center is the same for one slot (slot index). The cell stencil contains the addresses of cells, there can be any number of them and their points are compared with the center of the cell in question.

The only way how to break the never ending loop is to fulfill this condition.

```
if (returnReduce(nSamples, sumOp<label>()) == 0)
{
       break;
}
```

This function gathers data from different processors and returns the summation of the variable nSamples. This variable is set to zero in the start of the loop and it is incremented when the samples variable is set which was described in lines above, if the acceptor is not done yet.

The function distribute (called at line 1543) can be found at line 629 of file mapDistributeBaseTemplates.C. Its purpose is to distribute data and take care that multiple processes do not write the same. All of the donors have a valid cell center and the stencil for them is constructed.

```
DynamicList<label> donorCells(mesh_.nCells());
forAll(samples, cellI)
{
      if (samples[cellI] != greatPoint)
      {
          donorCells.append(cellI);
      }
}
```

If the samples have been changed in the beginning of this function, the donorCell array is extended of its value. The fields globalCells, donorCells, donorCellCells and donorCellCentres are changed in function globalCellCells (see subsection 3.3.15. Variable donorWeights is introduced as matrix of double with size of numbers of mesh cells.

For all donor cells the function stencilWeights is called as

```
stencilWeights
(
      samples[cellI],
      donorCentres,
      donorWeights[cellI]
);
```

and the field donorWeights is set. For details see 3.3.14. After the weights are obtained the cell interpolation map is updated and donor cells and weights are added.

Final check is done. If the stencil is actually for the correct cell in general mesh, the private variables cellStenci_ and cellInterpolationWeights_ are updated and this cell is set as done. The loop repeats until all interpolation cells have its donor and weights.

3.3.14 Function stencilWeights

This function is written in file inverseDistanceCellCellStencil.C from line 1433 to 1464.

This function takes three arguments: cell that is interpolated (acceptor), list of cells that are donors and list of double which is supposed to be filled with weights. It is done just for one acceptor. So the function is called in loop.

Loop is created over all donors in the list. Double variable d is introduced and its value is the distance between the centroids of the donor and acceptor. If this value d is bigger then the smallest number excepted in OpenFOAM the weight is not computed and its value is 1. Otherwise it is computed according

```
weights[i] = 1.0/d;
sum += weights[i];
```

and finally all of the weights are divided by whole sum of all weights, which is implemented as

```
forAll(weights, i)
{
     weights[i] /= sum;
}
```

3.3.15 Function globalCellCells

This function is written in file cellCellStencil.C from line 183 to 297. Its function is to create cell-cell addressing in global numbering.

First the function determines the global cell number on other side of coupled patches. Then the cells and all the neighbors are collected. First entry in stencil is the cell itself. Others entries are cell neighbors.

3.3.16 Function cellInterpolationWeights

This function is written in file inverseDistanceCellCellStencil.H (lines 320 – 323) and its purpose is to return private variable cellInterpolationWeights_.

3.3.17 Function interpolationCells

This function is written in file inverseDistanceCellCellStencil.H (lines 297 – 300) and its purpose is to return private variable interpolationCells_.

3.3.18 Function cellInterpolationWeight

This function is written in file inverseDistanceCellCellStencil.H (lines 327 – 330) and its purpose is to return private variable cellInterpolationWeight_.

This variable indicates the interpolation factor:

- zero for calculated
- one for interpolated

Chapter 4

Modification of inverse distance interpolation scheme

This chapter will contain two modifications of the overset library. The first modification will show how to change the weighting function in the inverse distance interpolation scheme. The second modification will change the input data for this function. The neighbors of neighbours will be added. This process is accompanied by checking if the more distant neighbors have already been added.

Both modifications will be described in detail. The inclusion of this new code into openFoam installation without the risk of destroying it will be showed.

4.1 Logarithmic distance

The purpose for this modification is to set weights for the 2D case as a logarithmic function. The old code

```
weights[i] = 1.0/d;
sum += weights[i];
```

is replaced with

```
if (mesh_.nGeometricD() == 2)
{
    weights[i] = -1*Foam::log(d);
}
else
{
    weights[i] = 1.0/d;
}
sum += weights[i];
```

If the case is run in two dimensions, the weights are set as a logarithmic function of the distance d, which is the distance between cell centers of acceptor and donor. Otherwise the distance is calculated as in the inverse distance interpolation scheme.

4.1.1 Incorporation into OpenFOAM

Because only one function is modified, it is not necessary to create a new overset library or inverseDistance interpolation scheme. The easiest way is to change leastSquares interpolation schemes, because it consists of only this one function. The least squares interpolation scheme is derived from inverse distance by changing the weights function. This process is described step by step.

First the terminal in Linux is open and the openFoam environment is loaded.

```
OFv1906
```

The second step is to copy the leastSquare file and rename it to logarithmInverseDistance. And the Make file is created.

```
ufoam
mkdir applications  %DONT DO IF THE FOLDER ALREADY EXISTS
cd applications
cp -r $FOAM_SRC/overset/cellCellStencil/leastSquares logarithmInverseDistance
mkdir logarithmInverseDistance/Make
```

Folder Make should contain two files: options and files. The options has following content.

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/overset/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -loverset
```

The libraries that are used for the code have to be specified. It is possible to copy this file from the overset library, in that case the overset library has to be included as well.

The second file files contains

```
logarithmInverseDistance.C

LIB = $(FOAM_USER_LIBBIN)/liblogarithmInverseDistance
```

In case of copying this file, it is really important to check if the library is compiled into FOAM_USER_LIBBIN unlike all standard libraries, which are compiled into FOAM_LIBBIN, so the user can overwrite some important things if he do not change this.

The files leastSquares.C and leastSquares.H should be renamed.

```
mv leastSquaresCellCellStencil.C logarithmInverseDistance.C
mv leastSquaresCellCellStencil.H logarithmInverseDistance.H
```

The structure of file logarithmInverseDistance is

```
logarithmInverseDistance
   |--- logarithmInverseDistance.C
   |--- logarithmInverseDistance.H
   |--- Make
         |--- files
         |--- options
```

The word leastSquare has to be changed inside the files by executing

```
sed -i 's/leastSquaresCellCellStencil.H/logarithmInverseDistance.H/g' logarithmInverseDistance.C
sed -i 's/leastSquares/logarithmInverseDistance/g' logarithmInverseDistance.C
sed -i 's/leastSquares/logarithmInverseDistance/g' logarithmInverseDistance.H
```

This is a good time to check if the code works for OpenFOAM case. So far no modification to leastSquares code was made except the name.

4.1.2 Check before modification

The code is compiled by `wmake` in logarithmInverseDistance directory. The output should be checked for errors. The tutorial simpleRotor is copied into run folder.

```
cp -r $FOAM_TUTORIALS/incompressible/overPimpleDyMFoam/simpleRotor/ $FOAM_RUN/.
```

The case has to be slightly modified. The overset interpolation scheme in fvSchemes file has to be changed and the user library has to be added into controlDict.

```
sed -i 's/inverseDistance/logarithmInverseDistance/g' $FOAM_RUN/simpleRotor/system/fvSchemes
sed -i 's/"libfvMotionSolvers.so"/"libfvMotionSolvers.so" "liblogarithmInverseDistance.so"/g'\
$FOAM_RUN/simpleRotor/system/controlDict
```

The case can be run now by the Allrun script. Check the log.overPimpleDyMFoam if the logarithmInverseDistance was used and if the computation run without errors.

4.1.3 Modification to logarithmInverseDistance

In file logarithmInverseDistance.C the new code for function stencilWeights is

```
void Foam::cellCellStencils::logarithmInverseDistance::stencilWeights
(
    const point& sample,
    const pointList& donorCcs,
    scalarList& weights
) const
{
    // Inverse-distance weighting

    weights.setSize(donorCcs.size());
    scalar sum = 0.0;
    forAll(donorCcs, i)
    {
        scalar d = mag(sample-donorCcs[i]);

        if (d > ROOTVSMALL)
        {
    if (mesh_.nGeometricD() == 2)
    {
     weights[i] = -1*Foam::log(d);
            }
    else
    {
             weights[i] = 1.0/d;
            }
            sum += weights[i];
        }
        else
        {
            // Short circuit
            weights = 0.0;
            weights[i] = 1.0;
            return;
        }
    }
    forAll(weights, i)
    {
        weights[i] /= sum;
    }
}
```

The declaration of the function was updated from inverseDistance to logarithmInverseDistance and weights were changed according the code written above. Make sure to copy over the whole function stencilWeights, not over the whole file.

After saving the file, the library should be compiled with `wmake` and the case can be run again. (in case folder: `./Allclean` and `./Allrun`)

4.2 Neighbors inverse distance

The purpose of this modification is to extend the number of donors. In the inverse distance interpolation scheme, the donors are found around the acceptor cell. This new function finds the neighbors of donors and adds them to the original stencil as well as their cell centers. This extension is displayed in figure 4.1. The function also checks if the neighbor was already added and in this case it skips the neighbor.

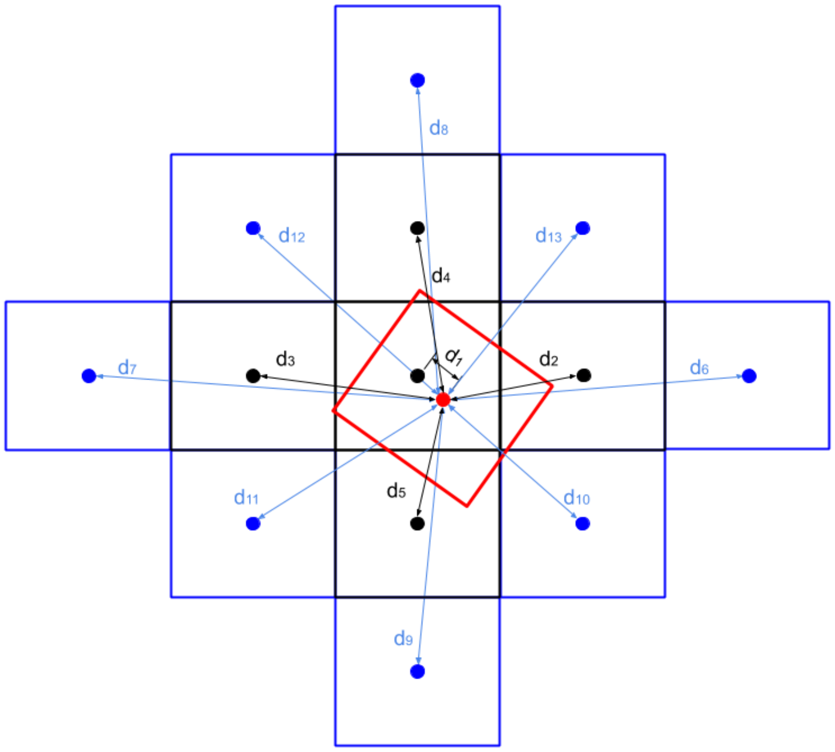


Figure 4.1: Distance of cell centers for determination of the weights, the red cell is acceptor, the black ones are original donors and the blue cells are newly added donors

The code of the extension is

```
forAll(donorCells,cellI)
{
   //index of donors
   label someCell = donorCells[cellI];
   //neighbors for acceptor, finding their neighbors
   labelList neighborCells = donorCellCells[someCell];

   //new field for new neighbours
   labelListList nbDonorCellCells(mesh_.nCells());
   //new field for cell centers of new nb
   pointListList nbDonorCellCentres(mesh_.nCells());
   //finds neibours of "neighborCells" and fills following arrays with cells and cell centers
   globalCellCells
   (
      globalCells,
      mesh_,
      isValidDonor,
      neighborCells,
      nbDonorCellCells,
      nbDonorCellCentres
   );

   for(label index = 1; index < neighborCells.size();index++)
   {
      label someNbCell = donorCellCells[someCell][index];
      forAll(nbDonorCellCells[someNbCell], k)
      {
         bool addNb = true;
         forAll(donorCellCells[someCell], j)
         {
            if(donorCellCells[someCell][j] ==  nbDonorCellCells[someNbCell][k])
            {
               addNb = false;
            }
         }
         if (addNb)
         {
            donorCellCentres[someCell].append(nbDonorCellCentres[someNbCell][k]);
            donorCellCells[someCell].append(nbDonorCellCells[someNbCell][k]);
         }
      }
   }
}
```

The first loop goes over all of the neighbors. The second loop goes over all neighbors for one acceptor and the third checks if the neighbor is or is not in the current field. If the neighbor was not found as a neighbor, it is added, so as its cell centers.

4.2.1 Incorporation into OpenFOAM

The process is almost the same as in subsecion 4.1.1 but with different name. So the terminal commands are listed without description.

```
OFv1906
ufoam
mkdir applications  %DONT DO IF THE FOLDER ALREADY EXISTS
cd applications
cp -r $FOAM_SRC/overset/cellCellStencil/leastSquares nbInverseDistance
mkdir nbInverseDistance/Make
```

Folder Make should contain two files: options and files. The options has following content.

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/overset/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -loverset
```

The second file files contains

```
nbInverseDistance.C

LIB = $(FOAM_USER_LIBBIN)/libnbInverseDistance
```

The files leastSquares.C and leastSquares.H should be renamed. In dictionary nbInverseDistance do

```
mv leastSquaresCellCellStencil.C nbInverseDistance.C
mv leastSquaresCellCellStencil.H nbInverseDistance.H
```

The structure of dictionary logarithmInverseDistance is

```
nbInverseDistance
   |--- nbInverseDistance.C
   |--- nbInverseDistance.H
   |--- Make
         |--- files
         |--- options
```

The word leastSquare has to be changed inside the files by executing

```
sed -i 's/leastSquaresCellCellStencil.H/nbInverseDistance.H/g' nbInverseDistance.C
sed -i 's/leastSquares/nbInverseDistance/g' nbInverseDistance.C
sed -i 's/leastSquares/nbInverseDistance/g' nbInverseDistance.H
```

Now it should work with simpleRotor case.

4.2.2 Check before modification

The check is the same as for the case before. The code is compiled by `wmake` in nbInverseDistance directory. The output should be checked for errors. The tutorial simpleRotor is copied into run folder.

```
cp -r $FOAM_TUTORIALS/incompressible/overPimpleDyMFoam/simpleRotor/ $FOAM_RUN/.
```

The case has to be slightly modified. The overset interpolation scheme in fvSchemes file has to be changed and the user library has to be added into controlDict.

```
sed -i 's/inverseDistance/nbInverseDistance/g' $FOAM_RUN/simpleRotor/system/fvSchemes
sed -i 's/"libfvMotionSolvers.so"/"libfvMotionSolvers.so" "libnbInverseDistance.so"/g' \
$FOAM_RUN/simpleRotor/system/controlDict
```

The case can be run now by Allrun script. Check the log.overPimpleDyMFoam if the lnbInverseDistance was used and if the computation run without errors.

4.2.3 Adding the extension into code

One additional change has to be done. The code has to be added in (the whole procedure is explained below step by step, so do not add it now)

```
$FOAM_SRC/overset/cellCellStencil/inverseDistance/inverseDistanceCellCellStencil.C
```

and it should start at line 1576 after

```
// Get neighbours (global cell and centre) of donorCells.
labelListList donorCellCells(mesh_.nCells());
pointListList donorCellCentres(mesh_.nCells());
globalCellCells
(
    globalCells,
    mesh_,
    isValidDonor,
    donorCells,
    donorCellCells,
    donorCellCentres
);
```

To do so, the file has to be opened and the whole function createStencil (line 1467 – 1650) has to be copied and put instead of the stencilweights function in nbInverseDistance.C.

The code above needs to be localized in nbInverseDistance.C file and the modification has to be added. After the added code

```
// Determine the weights.
scalarListList donorWeights(mesh_.nCells());
forAll(donorCells, i)
```

should follow. The function header should change from

```
void Foam::cellCellStencils::inverseDistance::createStencil
```

to

```
void Foam::cellCellStencils::nbInverseDistance::createStencil
```

The last modification has to be added into nbInverseDistance.H. Since the name of function had changed, the declaration has to be changed as well. Instead of

```
//- Calculate lsq weights for single acceptor
virtual void stencilWeights
(
     const point& sample,
     const pointList& donorCcs,
     scalarList& weights
) const;
```

The declaration from file inverseDistanceCellCellStencil.H must be added.

```
virtual void createStencil(const globalIndex&);
```

The library now have to be compiled by `wmake` and the functionality can be checked by running the simpleRotor tutorial modified above.

The difference for original inverse distance interpolation scheme and the new one is displayed for simpleRotor tutorial. The figure 4.2 shows the red line over which are magnitudes of velocity plotted. The left contures belong to original inverse distance and the right ones are for newly implemented scheme. The magnitudes of velocity over this lines are displayed in figure 4.3, the red one is for original scheme, and the blue one is for newly inplemented scheme.

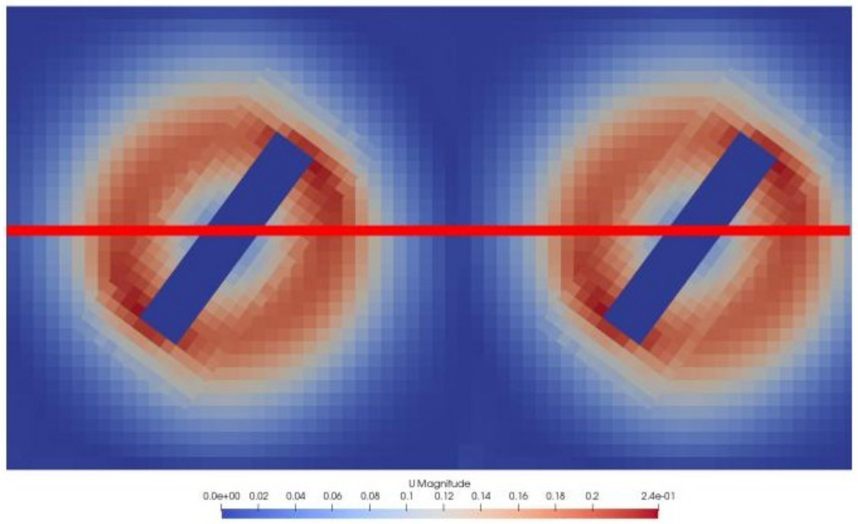


Figure 4.2: The contours for simpleRotor tutorial, left one is for usage of original the inverse distance interpolation scheme and the right one is for newly implemented nbInverseDistance, the red line is displayed for plotting purposes

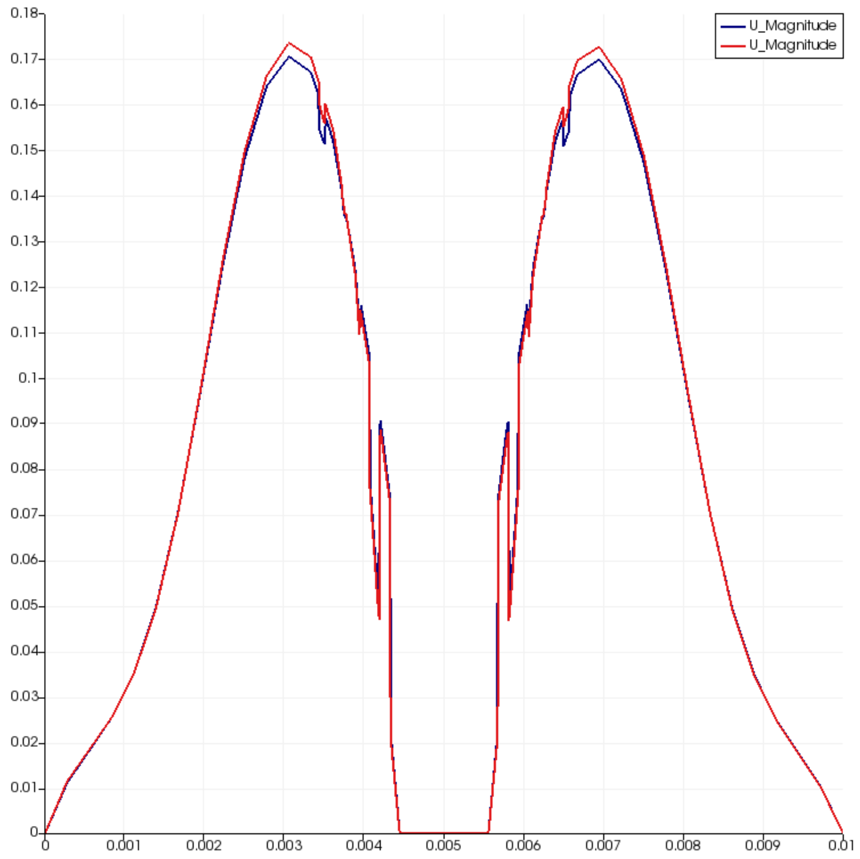


Figure 4.3: The magnitudes of velocity for original inverseDistance interpolation scheme (red) and for nbInverseDistance (blue) plotted over red line displayed in 4.2 (the cases overlap for the plotting)

Study questions

1. How many blockMesh files do you need to create overset meshes?
2. What is the maximum number of donors for one acceptor for structured mesh in standard inverse distance interpolation scheme? Consider 2D cases only.
3. The private variable cellStencil_ has data type labelListList. What does it mean and what data does this variable store?
4. The purpose of neighbors inverse distance implementation is to find new donors (neighbors of standard donors). The function globalCellCells is used for finding the new neighbors. Why the function that returns neighbors of given cell is not used instead?

Bibliography

[1] G. Houzeaux, B. Eguzkitza, R. Aubry, H. Owen, and M. Vazques. A chimera method for the incompressible navier-stokes equations. *International Journal for Numerical Methods in Fluids*, 75(3):155–183, 2014.

[2] L.C. Evans and American Mathematical Society. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 1998.