Cite as: Da Luz Moreira, A.: Complex mesh deformations in OpenFOAM: a custom boundary condition for prescribed mesh motion. In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2022

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Complex mesh deformations in OpenFOAM: a custom boundary condition for prescribed mesh motion

Developed for OpenFOAM-v2206
Requires: Python with `numpy` and `scipy` (for tutorials)

Author:
André Da Luz Moreira
Linköping University
andre.da.luz.moreira@liu.se

Peer reviewed by:
Jonas Lantz
Yuchen Zhou
Saeed Salehi

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 16, 2023

Learning outcomes

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

The reader will learn:

How to use it:

- How to use deforming dynamic meshes in OpenFOAM.
- How to to prescribe custom deformations interpolated to a boundary in OpenFOAM using the boundary condition `timeVaryingMotionInterpolation`.

The theory of it:

- How mesh deformation is calculated in OpenFOAM with Laplacian solvers.
- How arbitrary deformation information may be interpolated into boundary points in different ways.

How it is implemented:

- How deformable boundaries are implemented in the new boundary condition presented here.

How to modify it:

- How to modify an existing boundary condition for new purposes.
- How to implement different interpolation algorithms to apply motion data values boundary points.

Prerequisites

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

- How to run standard documented tutorials in OpenFOAM and modify them.
- Have a basic knowledge of how static and dynamic meshes function in OpenFOAM.
- Know how to compile and use custom libraries and boundary conditions in OpenFOAM.
- Be able to understand how some interpolation techniques work to transfer information from a provided field into mesh points.

Contents

Nomenclature

Acronyms

BC	Boundary Condition
CFD	Computational Fluid Dynamics
VOF	Volume Of Fluids

English symbols

$\boldsymbol{U}_i$	Cartesian velocity vector	[m/s]
$\boldsymbol{V}_i$	Cartesian arbitrary field vector	[-]
$\boldsymbol{X}_i$	Cartesian coordinates vector	[m]
i_i	3D matrix index	
l_i	Interpolation factor	
m_i	List index in OpenFOAM	
N_i	3D matrix dimension	
p	Inverse distance exponent term	
V_i	Arbitrary field value	[-]
w_i	Inverse distance weight factor	
x_i	Cartesian x-coordinates	[m]
y_i	Cartesian y-coordinates	[m]
z_i	Cartesian z-coordinates	[m]

Greek symbols

$\boldsymbol{\Gamma}_c$	Cartesian motion diffusivity at cell centres	[m^2/s]
Δ	Change in quantity/parameter (delta)	
Γ	Motion diffusivity at cell centres	[m^2/s]

Subscripts

0	Initial
c	Mesh cell centre parameter
i	Arbitrary subscript
p	Mesh point parameter
t	Time
min	Minimum
ref	3D matrix reference point

Other symbols

∇	Vector differential operator (nabla)

Chapter 1

Introduction

Deformations of computational fluid dynamic (CFD) meshes are frequently used to simulate flows in domains with variable geometries. This allows for a multitude of studies where dynamic systems are more accurately represented. Several types of motion may be used in CFD, so that not only the flow may be driven by a change in geometry, but also the results of a flow field may be used to compute an updated geometry or position. A flow driven by a piston in an engine cylinder with positions known *a priori* is an example of the first. For the second type, an example would be simulations of airfoils whose position and angle of attack are changed by the flow.

In most CFD software packages, including OpenFOAM, the existing boundary conditions (BCs) and utilities intended for mesh deformations are developed with more "standard" types of motion in mind. They are easily capable of deforming meshes for rigid body motions in one or more of its boundaries, as well as for deformations than can be described mathematically. However, complex and arbitrary deformation of a boundary is not a straightforward task, often requiring that boundary conditions are developed and customised for each case.

For a generic case of complex mesh deformation, where the motion of the boundary points is known (*e.g.* from experiments or measurements) but cannot be described using a mathematical function, the solution to applying this motion field to a computational mesh is to interpolate the values into the mesh points. An example of such a case would be simulations of biological flows, which may be performed using direct measurements of complex and irregular geometries with spatial and temporal information. These may come from computed tomography scans, magnetic resonance imaging, high resolution cameras, to mention a few examples.

This, however, is not something readily available in OpenFOAM. The aim of this report is to describe a generic boundary condition for mesh deformations based on case-specific motion information. This is accomplished by interpolating the data, be it from a point cloud or a structured grid, into the deformable CFD boundaries. This BC should be easily integrated into existing OpenFOAM code and should be able to take advantage of established dynamic mesh types, motion solvers and other utilities, without compromises or any loss of functionalities.

To lay the foundations required to understand the function of this boundary condition, a general explanation of the existing mesh deformation functionalities in **OpenFOAM v2206** are presented in Chapter 2. The developed code is then presented in Chapter 3, where the most important aspects required for understanding its usage will be explained. Some tutorial cases using this BC are explained in Chapter 4. These can be used as inspiration for more advanced and complex deformation cases the user may wish to study.

A complete list of files accompanying is included in Appendix A for easy referencing and preparation for following the contents of this report.

Chapter 2

Deforming meshes in OpenFOAM

The requirements for mesh deformation in OpenFOAM are included in multiple libraries in its source code, such as `dynamicMesh`, `fvMotionSolvers` and `dynamicFvMesh`. These features are possible in applications that enable dynamic meshes, such as `pimpleFoam`, `interFoam` and `rhoPimpleFoam`, as some noteworthy examples.

This chapter includes a brief explanation of how mesh deformation is triggered and controlled in OpenFOAM during execution. It will start with a generic description of the steps required for creating a deformation-capable dynamic mesh, followed by a more detailed description of some components of the `fvMotionSolvers` library. The matters discussed are the ones relevant to boundary motion and mesh deformation. This description aims at building the foundations for the topics in Chapter 3 and is not intended as a complete overview of this library nor of dynamic meshes in OpenFOAM.

2.1 Dynamic meshes and mesh motion in OpenFOAM

As an example of the handling of dynamic meshes in OpenFOAM, the source code of the solver `pimpleFoam` can be used as an example. The steps described below are not exclusive to this application, and are also found in other solvers with dynamic mesh capabilities.

The first step in using dynamic meshes in OpenFOAM is the creation of a `dynamicFvMesh` object, performed by including the line `#include"createDynamicFvMesh.H"` in the case setup. During this step, the information contained in the dictionary `dynamicMeshDict` is read, and the specified `dynamicFvMesh` type is used. Dynamic meshes allow for multiple advanced features in OpenFOAM. Some examples are mesh motion, deformation, overset meshes and sliding meshes. An example dictionary is shown below.

Example `dynamicMeshDict`

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2206                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh       dynamicMotionSolverFvMesh;
```

```
motionSolverLibs    ("libfvMotionSolvers.so");

motionSolver        displacementLaplacian;

diffusivity         quadratic inverseDistance (boundaryName);

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

OpenFOAM contains multiple types of `dynamicFvMesh` classes for various types of geometry or topology changes in meshes, which will not be discussed in detail in this report. For mesh deformation cases, a mesh of type `dynamicMotionSolverFvMesh`[1] seen above can be created. As its name suggests, this type of dynamic mesh requires an appropriate motion solver to be selected, as well as the library containing it. The selected `motionSolver` is responsible for the description of the dynamic behaviour of the mesh, as well as initialisation of the required fields for the deformation. The appropriate boundary condition (BC) files are read, verifying that information consistent with the contents of the dynamic mesh dictionary is found and used.

The fourth input above is not a mandatory requirement for all cases, but is used by motion solvers based on Laplace's equation, which need a description of how the deformation diffusivity is calculated.

The OpenFOAM library containing the utilities for mesh deformations relevant for this report is `fvMotionSolvers`, and its source code is located in `src/fvMotionSolver`. It builds on existing libraries for dynamic meshes and comprises mainly, but not only, of motion solvers, diffusivity models and BCs that account for moving boundaries in a mesh and how their deformation is distributed in this domain in a continuous manner. Additional information on selected types of `motionSolver` and `diffusivity` contained in `fvMotionSolvers` will be presented in Section 2.3.

Although other tasks specific to dynamic meshes are performed, the most important step is seen performed when the function `update()` inside the solution loop for a time step[2]. For a `dynamicMotionSolverFvMesh`, this update moves the mesh points using the output of the function `newPoints()` in the base class for all motion solvers `motionSolver` class. This function is seen in the code excerpt below.

Function `newPoints` in `src/dynamicMesh/motionSolvers/motionSolver/motionSolver.C`

```
Foam::tmp<Foam::pointField> Foam::motionSolver::newPoints()
{
    solve();
    return curPoints();
}
```

The functions `solve()` and `curPoints()` are solver-specific, but are originally declared on the base class `motionSolver`. This allows access to the correct sub-class at execution through dynamic binding to the correct function, specific to the chosen solver. These function are responsible for calculating the displacement information for the mesh's point according to each solver, and returning the new point coordinates for the mesh update. The implementation of these two functions will be discussed in Section 2.3 for the two relevant motion solvers in OpenFOAM using Laplace's equation.

2.2 Laplace's equation for mesh deformation

The two motion solvers discussed in this report use Laplace's equation to determine a mesh deformation field at cell centres and, for this reason, a brief introduction to this topic is given here. The generalised Laplace equation equation for a deformation field in three dimensions is given by

[1] Other types of `dynamicFvMesh` classes allow for more complex mesh motion and deformation combinations. Some examples are as `dynamicMotionSolverListFvMesh`, which allows multiple different motion solvers to be selected; and `dynamicMultiMotionSolverFvMesh`, for which solvers for separate `cellZones` in a mesh may be selected. These are beyond the scope of this report and will not be described.

[2] The function `controlledUpdate()` may be used, as is the case in `pimpleFoam`. This functions ultimately triggers `update()`

$$\nabla \cdot (\boldsymbol{\Gamma}_c \, \nabla \boldsymbol{V}_c) = 0 \, . \tag{2.1}$$

The terms $\boldsymbol{\Gamma}_c$ and $\boldsymbol{V}_c$ are the mesh deformation diffusivity and an arbitrary deformation vector fields, defined at mesh cell centres. This arbitrary field will depend on the selected Lagrangian motion solver and can be chosen to be either the cell centres' displacements or their velocities. The diffusivity may be specified in different manners, and despite the fact that most models for this parameter are scalar, the generalised vectorial form $\boldsymbol{\Gamma}_c$ will be used. The different types of diffusivity models in OpenFOAM 2206, selected by the user in `dynamicMeshDict`, will be shortly presented in Section 2.4..

Given that appropriate motion boundary conditions are specified for the CFD domain, the solution to this equation is a deformation field in which motion in the whole mesh is calculated to accommodate these BCs while preserving the mesh validity and quality [1]. Advantages to using this approach include this equation's easy and rapid solution, with bounded results for a smooth non-uniform distribution of $\boldsymbol{V}_c$.

Since the final mesh deformation must be calculated not at cell centres, but at the mesh points, the main disadvantage of this approach is the need for interpolation. This step will be seen in the `curPoints` functions for both motion solvers described in the following section. Studies on this matter have shown that for complex deformations this results in flipping and degeneration of mesh cells, as well as problems in corner points belonging to one cell only [1]. These problems are commonly solved by periodically remeshing the simulation domain, which can be a costly computational task. Although mesh deformations based on equations defined at points instead of cell centres exist, they will not be discussed here as these are not implemented in OpenFOAM 2206.

2.3 Motion solvers in `fvMotionSolvers`

The motion solvers, located in the library sub-folder named `fvMotionSolvers`, are primarily divided in two types: displacement and velocity solvers. All of them ultimately determine displacements for the points in the mesh, but different approaches are used for this, according to the type of mesh motion selected. Displacement solvers fulfil this task by first calculating a displacement field in the cell centres of the mesh, which is then interpolated to the all points and then added to their initial coordinates for an update in position. The velocity solvers, on the other hand, first solve for a velocity field in the cells, which is interpolated to points, and the points' displacement is calculated by the product of this velocity and the simulation time step size.

2.3.1 Velocity solvers

The two mesh velocity solvers available in OpenFOAM as part of `fvMotionSolvers` are based on the solution of a mesh velocity field, using Laplace's equation. These are:

- `velocityComponentLaplacian`[3]
  Description: *Mesh motion solver for an fvMesh. Based on solving the cell-centre Laplacian for the given component of the motion velocity.*

- `velocityLaplacian`[4]
  Description: *Mesh motion solver for an fvMesh. Based on solving the cell-centre Laplacian for the motion velocity.*

As seen in the description text above, copied directly from their header files in the source code, they are used in quite similar manners to solve the mesh motion and the differences between them are only related to the description of the cell centres' velocity field (`cellMotionU`) as a vector field or as one of three components of the velocity. This allows for simplifications and faster computations in cases where the mesh boundaries are only displaced on one direction. For this reason, only

[3] Located in `src/fvMotionSolver/fvMotionSolvers/componentVelocity/componentLaplacian`.
[4] Located in `src/fvMotionSolver/fvMotionSolvers/velocity/laplacian`.

velocityLaplacian will be described here, as all of the considerations made for it are also applicable to the more limited velocityComponentLaplacian.

The solver velocityLaplacian is built upon two others: velocityMotionSolver[5], from the dynamic mesh library, and the generalised base class fvMotionSolver[6]. The first is used to include references to the mesh through a polyMesh object, access to the information provided by the user in dynamicMeshDict, while also creating the field pointMotionU during object construction, which represents the points velocity used for the displacement calculations. The second class, fvMotionSolver, is the base for all motion solvers in the fvMotionSolvers library and provides access to the mesh by a fvMesh object (obtained by dynamically casting a polyMesh object)[7].

The Laplacian velocity solver complements its dependencies by creating the cell centre velocity field cellMotionU, along with class members and functions for performing all the necessary steps to solve the cell velocity distribution equation and deform the mesh according to this equation's results. More details are available in the OpenFOAM header file velocityLaplacianFvMotionSolver.H[8].

Some noteworthy details of this class are its constructor and the functions curPoints and solve (see Section 2.1). These will be briefly described here. The function updateMesh, also part of velocityLaplacian, is important in cases of topology updates, but is beyond the scope of this report and will not be discussed here.

Constructor for velocityLaplacian object:

As seen in the text excerpt for velocityLaplacian's constructor[9], the Laplacian velocity solver receives a reference to the mesh through a polyMesh object and through a dictionary (more specifically, to dynamicMeshDict).

Constructor from velocityLaplacianFvMotionSolver.C

```
Foam::velocityLaplacianFvMotionSolver::velocityLaplacianFvMotionSolver
(
    const polyMesh& mesh,
    const IOdictionary& dict
)
:
    velocityMotionSolver(mesh, dict, typeName),
    fvMotionSolver(mesh),
    cellMotionU_
    (
        IOobject
        (
            "cellMotionU",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        fvMesh_,
        dimensionedVector(pointMotionU_.dimensions(), Zero),
        cellMotionBoundaryTypes<vector>(pointMotionU_.boundaryField())
    ),
    interpolationPtr_
    (
        coeffDict().found("interpolation")
      ? motionInterpolation::New(fvMesh_, coeffDict().lookup("interpolation"))
      : motionInterpolation::New(fvMesh_)
    ),
    diffusivityPtr_
    (
        motionDiffusivity::New(fvMesh_, coeffDict().lookup("diffusivity"))
    )
```

[5] See contents of src/dynamicMesh/motionSolvers/velocity.
[6] See contents of src/fvMotionSolver/fvMotionSolvers/fvMotionSolver.
[7] See the class constructor in fvMotionSolver.C
[8] Full path: src/fvMotionSolver/fvMotionSolvers/velocity/laplacian/velocityLaplacianFvMotionSolver.H.
[9] See src/fvMotionSolver/fvMotionSolvers/velocity/laplacian/velocityLaplacianFvMotionSolver.C.

```
{}
```

After providing the required inputs for its inheritances, three important objects in this solver are created. First, the previously mentioned field `cellMotionU` is initialised or read from the case's time folders. Then, an interpolator object is created, which is used for transferring the calculated cell centre velocities to the mesh points. Finally, a motion diffusivity object is created. The diffusivity is a key parameter in distributing the deformation in the internal mesh points during the equation solution process and will be further explained in Section 2.4.

Function `solve`:

The `solve` function is responsible for solving an equation for the velocities in the cell centres, from which deformation is ultimately calculated. This takes the form of Laplace's equation,

$$\nabla \cdot (\boldsymbol{\Gamma}_c \, \nabla \boldsymbol{U}_c) = 0 \, , \tag{2.2}$$

where $\boldsymbol{U}_c$ represents the cell centre velocities, similar to Equation (2.1).

As one may see in the code for this function, it starts with updating some terms, including the diffusivity field and the point velocities field at the mesh boundaries using the specified boundary conditions. It is worth noting that the `movePoints` function below, inherited from the `velocityMotionSolver` class, does not perform any tasks in this case. This may be verified by inspecting the function code in file `velocityMotionSolver.C`. The line `pointMotionU_ .boundaryFieldRef().updateCoeffs();` will be of importance in Chapter 3 (Section 3.4.2), as it is responsible for triggering each boundary's own `updateCoeffs` function, where the user specified boundary conditions are updated. Then, after defining the `fvOptions` object and obtaining the number of non orthogonal corrector steps defined in the simulation inputs, it goes on to define and solve the deformation velocity equation.

Function `solve` in `velocityLaplacianFvMotionSolver.C`

```
void Foam::velocityLaplacianFvMotionSolver::solve()
{
    // The points have moved so before interpolation update
    // the fvMotionSolver accordingly
    movePoints(fvMesh_.points());

    diffusivityPtr_->correct();
    pointMotionU_.boundaryFieldRef().updateCoeffs();

    fv::options& fvOptions(fv::options::New(fvMesh_));

    const label nNonOrthCorr
    (
        getOrDefault<label>("nNonOrthogonalCorrectors", 1)
    );

    for (label i=0; i<nNonOrthCorr; ++i)
    {
        fvVectorMatrix UEqn
        (
            fvm::laplacian
            (
                dimensionedScalar("viscosity", dimViscosity, 1.0)
              * diffusivityPtr_->operator()(),
                cellMotionU_,
                "laplacian(diffusivity,cellMotionU)"
            )
         ==
            fvOptions(cellMotionU_)
        );

        fvOptions.constrain(UEqn);
        UEqn.solveSegregatedOrCoupled(UEqn.solverDict());
        fvOptions.correct(cellMotionU_);
    }
```

```
}
```

Function curPoints:

In curPoints, the point velocities $\boldsymbol{U}_p$ are first obtained from the cell centre values $\boldsymbol{U}_c$ (calculated in solve) by using the class' interpolator object. Then, the updated mesh point coordinates are calculated as

$$\boldsymbol{X}_{t_n} = \boldsymbol{X}_{t_{n-1}} + \Delta t \cdot \boldsymbol{U}_p \ , \tag{2.3}$$

where $\boldsymbol{X}_{t_n}$ and $\boldsymbol{X}_{t_{n-1}}$ are the point coordinates at the current and previous simulated times, Δt is the time step size. The function ultimately returns these values as a tmp<Foam::pointField> object used to update the mesh point coordinates, as seen in the following code excerpt.

Function curPoints in velocityLaplacianFvMotionSolver.C

```
Foam::tmp<Foam::pointField>
Foam::velocityLaplacianFvMotionSolver::curPoints() const
{
    interpolationPtr_->interpolate
    (
        cellMotionU_,
        pointMotionU_
    );

    tmp<pointField> tcurPoints
    (
        fvMesh_.points()
      + fvMesh_.time().deltaTValue()*pointMotionU_.primitiveField()
    );

    twoDCorrectPoints(tcurPoints.ref());

    return tcurPoints;
}
```

2.3.2 Displacement solvers

Unlike the velocity solvers, the list of mesh displacement solvers includes options that go beyond the use of Laplace's equation for the mesh deformation. The existing motion solvers for displacment in the fvMotionSolvers library, as of OpenFOAM's version 2206, are:

- displacementComponentLaplacian[10]
  Description: *Mesh motion solver for an fvMesh. Based on solving the cell-centre Laplacian for the given component of the motion displacement.*
- displacementLaplacian[11]
  Description: *Mesh motion solver for an fvMesh. Based on solving the cell-centre Laplacian for the motion displacement.*
- displacementSBRStress[12]
  Description: *Mesh motion solver for an fvMesh. Based on solving the cell-centre solid-body rotation stress equations for the motion displacement.*
- solidBodyDisplacementLaplacian[13]
  Description: *Applies Laplacian displacement solving on top of a transformation of the initial points using a solid-body motion.*

[10] See contents of src/fvMotionSolver/fvMotionSolvers/componentDisplacement/componentLaplacian.
[11] See contents of src/fvMotionSolver/fvMotionSolvers/displacement/laplacian.
[12] See contents of src/fvMotionSolver/fvMotionSolvers/displacement/SBRStress.
[13] See contents of src/fvMotionSolver/fvMotionSolvers/displacement/solidBodyDisplacementLaplacian.

- surfaceAlignedSBRStress[14]
  Description: *Mesh motion solver for an fvMesh. Based on solving the cell-centre solid-body rotation stress equations for the motion displacement. The model calculates the necessary rotation to align the stl surface with the closest mesh face normals and it calculates the respective source term for the SBRStress equation.*

As their names and descriptions indicate, displacementComponentLaplacian is similar to the aforementioned Laplacian component solver, while displacementLaplacian is similar in function the more complete velocityLaplacian. The two SBRStress displacement solvers are based on the solution of rotation stresses, while solidBodyDisplacementLaplacian is based on deformations based on rigid body motions. As the aim of this report is to describe a novel boundary condition used for interpolating arbitrary displacements into a mesh, only the first two solvers will be described here. As with the cased of the solvers based on mesh deformation velocities, only the more general displacementLaplacian will be described here. Several similarities exist between displacementLaplacian and velocityLaplacian, and the reader is referred to 2.3.1 in such cases.

As its velocity counterpart, displacementLaplacian is built from two classes: the general displacement solver displacementMotionSolver[15], and fvMotionSolver. The displacement solver is similar to velocityMotionSolver, providing access to the mesh through a polyMesh object, access to the dynamicMeshDict dictionary, and more importantly, creates the field pointDisplacement for the mesh deformation. The Laplacian displacement motion solver class expands these with the creation of the cell centre displacement field (cellDisplacement), along with members and functions pertaining to the solution of this field and calculation of updated mesh point coordinates using the field's results.

Similar to the description in Section 2.3.1, the relevant constructor and the two functions solve and curPoints for displacementLaplacian will be discussed here.

Constructor for displacementLaplacian object:

The two available constructors for displacementLaplacian[16]perform more tasks than the constructor for velocityLaplacian, but their main function is similar. Due to the length of these constructor functions, and the fact that not significant differences exist between them and the constructor for velocityLaplacian presented in Section 2.3.1, they will not be included here. The reader is referred to the source files for detailed inspection of the constructor code.

After providing inputs to the classes from which it is built, the motion field cellDisplacement is created, as are the interpolator and diffusivity. Two additional optional objects exist for the solver displacementLaplacian, pointLocation_ and frozenPointsZone_, are used for position boundary conditions and to fix certain points in place. These two objects will not be discussed here and the reader is referred to the solver's source code for additional information.

Function solve:

The solution of the cell displacement field $\boldsymbol{\Delta X}_c$ follows a similar process as the one seen in the previous section for the cell velocities. It uses Laplace's equation for a displacement distribution,

$$\nabla \cdot (\Gamma \ \nabla(\boldsymbol{\Delta X}_c)) = 0. \tag{2.4}$$

The excerpt from displacementLaplacianFvMotionSolver.C shows that the steps taken are similar the those seen in the solution of the mesh velocity field in velocityLaplacian. After updating the diffusivity field and the boundary conditions, the displacement equation is created and solved. The main difference in this function when compared to the Laplacian velocity solver is that no corrector steps for mesh non-orthogonality are employed.

Function solve in displacementLaplacianFvMotionSolver.C

```
void Foam::displacementLaplacianFvMotionSolver::solve()
```

[14]See contents of src/fvMotionSolver/fvMotionSolvers/displacement/surfaceAlignedSBRStress.
[15]See contents of src/dynamicMesh/motionSolvers/displacement/displacement.
[16]See src/fvMotionSolver/fvMotionSolvers/displacement/laplacian/displacementLaplacianFvMotionSolver.H and .C.

```
{
    // The points have moved so before interpolation update
    // the motionSolver accordingly
    movePoints(fvMesh_.points());

    diffusivity().correct();
    pointDisplacement_.boundaryFieldRef().updateCoeffs();

    fv::options& fvOptions(fv::options::New(fvMesh_));

    // We explicitly do NOT want to interpolate the motion inbetween
    // different regions so bypass all the matrix manipulation.
    fvVectorMatrix TEqn
    (
        fvm::laplacian
        (
            dimensionedScalar("viscosity", dimViscosity, 1.0)
           *diffusivity().operator()(),
            cellDisplacement_,
            "laplacian(diffusivity,cellDisplacement)"
        )
     ==
        fvOptions(cellDisplacement_)
    );

    fvOptions.constrain(TEqn);
    TEqn.solveSegregatedOrCoupled(TEqn.solverDict());
    fvOptions.correct(cellDisplacement_);
}
```

Function `curPoints`**:**

The function `curPoints` shown below, although at a first glance more complicated that the one in seen for the `velocityLaplacian` solver, does not present major differences for cases where the optional `pointLocation_` and `frozenPointsZone_` are not used.

Function `curPoints` in `displacementLaplacianFvMotionSolver.C`

```
Foam::tmp<Foam::pointField>
Foam::displacementLaplacianFvMotionSolver::curPoints() const
{
    interpolationPtr_->interpolate
    (
        cellDisplacement_,
        pointDisplacement_
    );

    if (pointLocation_)
    {
        if (debug)
        {
            // Not shown here for conciseness sake
        }

        pointLocation_().primitiveFieldRef() =
            points0()
          + pointDisplacement_.primitiveField();

        pointLocation_().correctBoundaryConditions();

        // Implement frozen points
        if (frozenPointsZone_ != -1)
        {
            // Not shown here for conciseness sake
        }

        twoDCorrectPoints(pointLocation_().primitiveFieldRef());
```

```
        return tmp<pointField>(pointLocation_().primitiveField());
    }
    else
    {
        tmp<pointField> tcurPoints
        (
            points0() + pointDisplacement_.primitiveField()
        );
        pointField& curPoints = tcurPoints.ref();

        // Implement frozen points
        if (frozenPointsZone_ != -1)
        {
            // Not shown here for conciseness sake
        }

        twoDCorrectPoints(curPoints);

        return tcurPoints;
    }
}
```

Similar to the previous case, this function begins with the interpolation of the calculated cell-centred data ($\boldsymbol{\Delta X}_c$) to the mesh point locations($\boldsymbol{\Delta X}_p$). The first `if`-statement in the function is only used for the aforementioned position boundary conditions. When this type of boundary condition is not used, the point coordinates are updated using the very straightforward relation

$$\boldsymbol{X}_{t_n} = \boldsymbol{X}_{t_0} + \boldsymbol{\Delta X}_p. \tag{2.5}$$

One important aspect for the displacement solver implementation's is that the returned field is not calculated based on the mesh points' previous coordinates, but on their initial coordinates $\boldsymbol{X}_{t_0}$. After calculation, the function returns the coordinates as a `tmp<Foam::pointField>` object, just as in `velocityLaplacian`.

2.4 Mesh deformation diffusivity in OpenFOAM

The selection of an appropriate diffusivity model is critical depending on the type of mesh motion used. Different models for the diffusivity may be applicable in different cases, and the ones available in the `fvMotionSolvers` library are listed below.

- `uniformDiffusivity`[17]
  Description:
- `inverseDistance`[18]
  Description: *Inverse distance to the given patches motion diffusivity.*
- `inverseFaceDistance`[19]
  Description: *Inverse distance to the given patches motion diffusivity.*
- `inversePointDistance`[20]
  Description: *Inverse distance to the given patches motion diffusivity.*
- `inverseVolume`[21]
  Description: *Inverse cell-volume motion diffusivity.*

[17] See contents of `src/fvMotionSolver/motionDiffusivity/uniform`.
[18] See contents of `src/fvMotionSolver/motionDiffusivity/inverseDistance`.
[19] See contents of `src/fvMotionSolver/motionDiffusivity/inverseFaceDistance`.
[20] See contents of `src/fvMotionSolver/motionDiffusivity/inversePointDistance`.
[21] See contents of `src/fvMotionSolver/motionDiffusivity/inverseVolume`.

- `directional`[22]
  Description: *Directional finite volume mesh motion diffusivity.*

- `motionDirectional`[23]
  Description: *MotionDirectional finite volume mesh motion diffusivity.*

- `file`[24]
  Description: *Motion diffusivity read from given file name.*

The descriptions included above were copied from the diffusivity models' source files, and unfortunately do not provide detailed information about the models. The 2006 paper by Jasak and Tuković provide interesting insights about differences between diffusivity models [1], but do not fully describe the models listed here. For a more accurate understanding of each diffusivity model's behaviour, their source code should be analysed. This, however, is beyond the scope of this report, since the selection of an appropriate diffusivity model should be simulation-specific.

The diffusivity models may also be combined with two available manipulator classes, included in `src/fvMotionSolver/motionDiffusivity/manipulators`. The calculated values from the above models may be modified with a square or an exponential function, by adding a keyword between the model name (see the example `dynamicMeshDict` in Section 2.1).

For arbitrary mesh motions the distance based diffusivity models outperform the uniform diffusivity model, as reported by Löner and Yang [2] as well as by Jasak and Tuković [1]. Similar results were obtained in tests during the preparation of this report, with results showing better mesh quality, with lower non-orthogonality and skewness, among other indicators. The reader is encouraged to investigate these by altering the diffusivity parameters in the example tutorial later presented in Section 4.1. The inverse volume, directional, motion directional and `file` diffusivity models were not investigated.

[22] See contents of `src/fvMotionSolver/motionDiffusivity/directional`.
[23] See contents of `src/fvMotionSolver/motionDiffusivity/motionDirectional`.
[24] See contents of `src/fvMotionSolver/motionDiffusivity/file`.

Chapter 3

Description of implemented boundary condition

The boundary condition `timeVaryingMotionInterpolation` was developed for seamless integration with the existing library `fvMotionSolvers`. It has been made by extensive modifications to the existing BC `timeVaryingMappedFixedValue`[1]. The boundary condition can be compiled as part of a new library that includes all the existing functionalities of `fvMotionSolvers` using the provided `Make` folder and files, named `myFvMotionSolvers`. The complete list of included files as well as instructions for compilation are presented in Section 3.1.

This novel BC utilises motion information provided as point coordinates and corresponding motion data (displacements or velocities), which may or may not vary in time. This motion may be obtained in different manners, such as experimental measurements, registration of image data, among others. The flowchart in Figure 3.1 illustrates the overall function of `timeVaryingMotionInterpolation`.

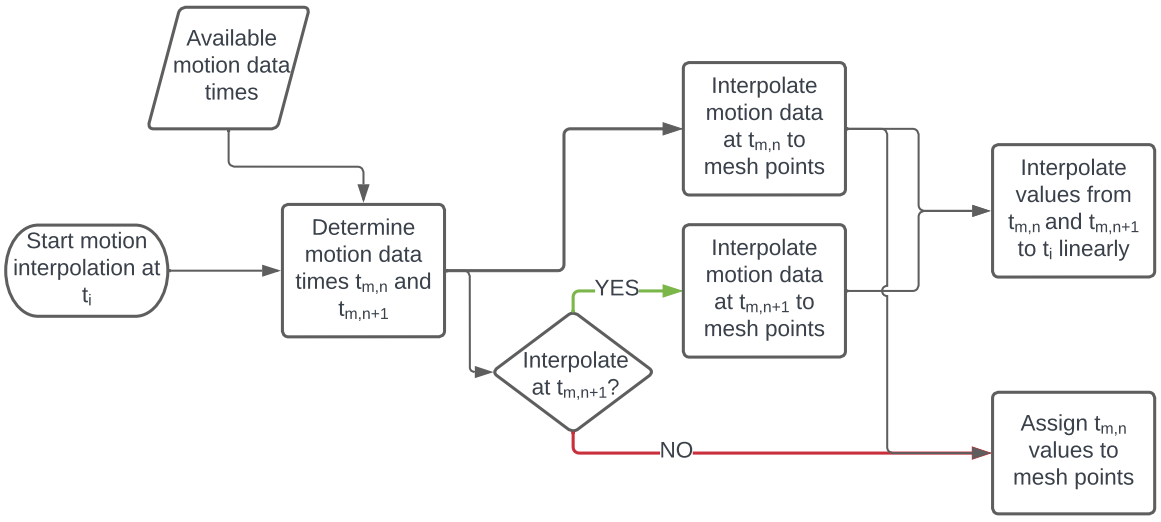
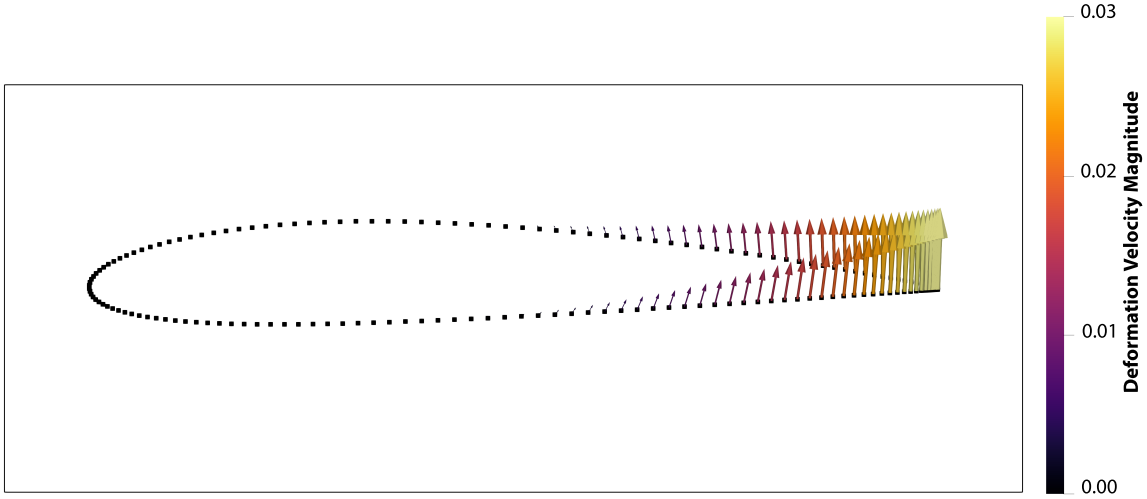


Figure 3.1: Flowchart for data interpolation in Boundary Condition.

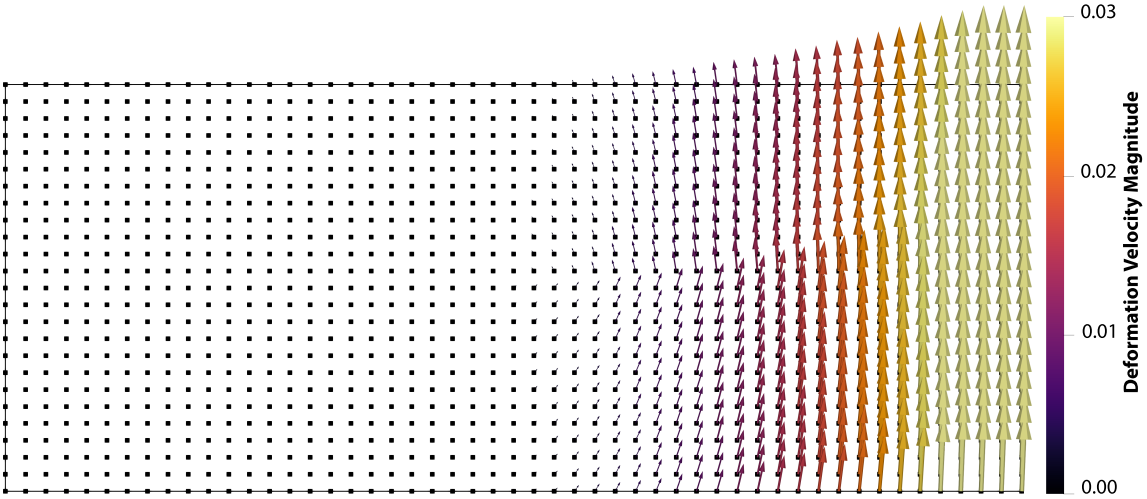
As we may see, for a given simulation time t_i, the BC first verifies the time values at which motion data are available. If the simulation time is the same as one of the available motion information times, $t_{m,n}$, the corresponding data are interpolated into the boundary points coordinates and these values are assigned to the boundary. If t_i lies between two available motion data values, $t_{m,n}$ and $t_{m,n+1}$, the data are first interpolated in space for these two time values and ultimately interpolated linearly in time.

[1] Located in `src/fvMotionSolver/pointPatchFields/derived/timeVaryingMappedFixedValue`.

Although conceptually simple, the key parts of `timeVaryingMotionInterpolation` concern different types of input data formats, as well as the different interpolation methods that may be used. The provided motion data may be of two types: structured or unstructured. Unstructured data is given by a set of points arbitrarily distributed in space where motion information is provided. Structured data, on the other hand, is given in the form of a uniformly spaced grid, not requiring a list of coordinates but instead the information characterising a matrix containing the motion data. Unstructured and structured data are shown in Figure 3.2 for a two-dimensional example.



(a) Unstructured.



(b) Structured.

Figure 3.2: Examples of motion velocity data for a 2D case.

Figure 3.2a shows the displacement velocity for a deforming airfoil trailing edge, corresponding to the tutorial for usage of `timeVaryingMotionInterpolation`, later seen in Section 4.1. showing either points or a grid with motion information. In Figure 3.2b this same airfoil displacement is shown as a field in a structured 2D grid. Further information on the use of `timeVaryingMotionInterpolation` is included in Section 3.2, with details and examples of the types of motion data files accepted.

The spatial interpolation from the motion data points to the boundary points may be performed using different techniques. These will first be introduced in Section 3.2, but their methodology will be explained with more detail in Section 3.3. Finally, in Section 3.4, the boundary condition

code is presented in more detail, with some example excerpts from the boundary condition files `timeVaryingMotionInterpolationPointPatchField.H` and `.C` for important tasks, as well as the implementations for the interpolation types.

This report is not aimed at guiding the reader on choosing between the provided interpolation methods as this is a problem specific task and will depend on countless factors. However, some remarks based on previous experience will be made here. As is to be expected, the results of the data interpolation are highly dependant on the amount of data points provided, their location and density with regards to the target boundary points, as well as the complexity of the mesh motion or deformation. These factors, associated with some of the drawbacks of the Lagrangian equation method presented in Chapter 2, may require that the CFD domain is remeshed multiple times during the simulation to preserve mesh quality. The two tutorials presented in Chapter 4 should be used as inspiration and grounds for experimentation on the effects of the input data, interpolation types and other parameters to the mesh motion results.

One final remark will be made about the aforementioned linear interpolation in time. This is the simplest interpolation possible between two available data time values, and a smoother mesh motion could potentially be achieved with a more complex interpolation scheme in time. This, however, would greatly increase the complexity of the boundary condition and will not be implemented or investigated here.

3.1 Compilation of implemented boundary condition

The source code provided for the boundary condition `timeVaryingMotionInterpolation` follows the folder structure in the original `fvMotionSolvers`. The scripts containing its code lie in the folder `myFvMotionSolver/pointPatchFields/derived/timeVaryingMotionInterpolation`. The files `timeVaryingMotionInterpolationPointPatchFields.H` and `.C` do not contain any relevant code, but are responsible for correct compilation of the boundary condition and integration with OpenFOAM. The files `timeVaryingMotionInterpolationPointPatchField.H` and `.C` contain the actual boundary condition code, such as variables, constructors, member functions, etc.

The compilation instructions and dependencies are included in the files `Make/files` and `Make/options`. By running the OpenFOAM compilation command `wmake` or running the provided script `Allwmake` the boundary condition is compiled and the library `myFvMotionSolvers` is created. This library may be used by setting its name in the `dynamicMeshDict` field `motionSolverLibs`. The complete folder structure for the library `myFvMotionSolvers` is shown below, with all files and sub-folders.

Contents of folder `myFvMotionSolver`

```
myFvMotionSolver
|-- Allwclean
|-- Allwmake
|-- Make
|   |-- files
|   `-- options
`-- pointPatchFields
    `-- derived
        `-- timeVaryingMotionInterpolation
            |-- timeVaryingMotionInterpolationPointPatchField.C
            |-- timeVaryingMotionInterpolationPointPatchField.H
            |-- timeVaryingMotionInterpolationPointPatchFields.C
            `-- timeVaryingMotionInterpolationPointPatchFields.H
```

3.2 Usage of implemented boundary condition

The usage of `timeVaryingMotionInterpolation` requires that some parameters are specified in the correct boundary condition input file, as exemplified here:

Inputs to boundary condition with default values

```
boundaryName
{
    type                timeVaryingMotionInterpolation;

    // Optional entries
    inputType           unstructured;
    interpolationType   nearest;
    inverseDistRadius   0.01;
    inputFolderName     boundaryName;
    intOutsideBounds    true;
    value               $internalField;
}
```

The entries shown above are used to define the type of motion data, type of interpolation used, and other parameters. Their default values and possible inputs are listed in Table 3.1 and explained briefly below.

Table 3.1: Inputs to `timeVaryingMotionInterpolation`

Field	Default	Accepted values
`inputType`	`unstructured`	`unstructured`, `structured`.
`interpolationType`	`nearest`	`nearest`, `inverseDist`, `trilinear`.
`inverseDistRadius`	-	Any float value.
`inputFolderName`	Boundary name.	A folder name.
`intOutsideBounds`	`true`	`true`, `false`.
`value`	-	Any value of the field type.

- `inputType`
  Determines the type of input data provided for the interpolation of the deformation to the mesh points. All deformation data should be saved in the folder named according to `inputFolderName` (see explanation below), with sub-folders for each time value where data are available.
  - `unstructured`: data points with unstructured coordinates (see 3.2.1.1).
  - `structured`: structure input data organised in a 3-dimensional matrix (see 3.2.1.2).

  For the `unstructured` input, there should be a file name `points` containing the the point coordinates for the data, and a file named after the applicable field (`pointMotionU`, `pointDisplacement`, etc.), containing the available values corresponding to the point coordinates in the time sub-directories. For the `structured` input the `points` files are not needed, but instead a single file named `domainMatrixInfo` in `inputFolderName` is required.

- `interpolationType`
  Specifies how the input data values will be assigned to the boundary points.
  - `nearest`: Applies the nearest value in the input point data to the boundary point location (see 3.3.1).
  - `inverseDist`: Performs an inverse distance interpolation from the point data to the boundary point (see 3.3.2).
  - `trilinear`: Performs a trilinear interpolation inside a uniform lattice grid, which is only applicable for `inputType` options `structured` (see 3.3.3).

- `inverseDistRadius`
  A radius value utilised only when the `inverseDist` interpolation method is selected. If this method is chosen, this becomes a mandatory input.

- `inputFolderName`
  The input data to be interpolated to the mesh must be stored in a folder inside `constant/boundaryData/inputFolderName`. If a valid value is not provided, the boundary condition will default to a folder named after the specific boundary. Using a custom folder name may be useful if the same input files are used for multiple boundaries in the same simulation case.

- `intOutsideBounds`
  Only applicable if `inputType` set to `structured` and `interpolationType` is `trilinear`. In case of a structured input, determines how points outside of the input matrix should be treated. If set to `false`, the field values outside of the input matrix will be set to zero. Otherwise, values will be determined for this points based on the linear or bilinear interpolation (see 3.3.3).

- `value`
  Standard OpenFOAM input. If a value is provided the initial field is set to this value on all points.

The motion interpolation to be used shall be stored in the folder `constant/boundaryData/inputFolderName`. Data files for point coordinates and data must be stored in sub-folders named after the time value, as shown in the examples in Sections 3.2.1.1 and 3.2.1.2.

3.2.1 Types of input data

3.2.1.1 Unstructured input data

If `inputType` is set to `unstructured`, the input motion data is assumed to be a point cloud with no regularity in the points distribution. For this reason, the boundary condition requires that for each time value where data is available two files should exist. The first, named `points`, is read by OpenFOAM as a `pointField`. The file should start with an integer representing the size of this `pointField`, followed by the values of the point coordinates.

Example of file `points`

```
// Example points file
N
(
(pX_0       pY_0        pZ_0    )
(pX_1       pY_1        pZ_1    )
(pX_2       pY_2        pZ_2    )

...

(pX_N-1     pY_N-1      pZ_N-1  )
)
```

The second file should be named either `pointDisplacement` or `pointMotionU`, according to the type of motion data provided. The example below shows a formatting similar to that used in the `points` file. In case the motion data is provided in one component only, the file shall be adjusted accordingly.

Example of file `pointMotionU` or `pointDisplacement`

```
// Example motion data file
N
(
(mX_0       mY_0        mZ_0    )
(mX_1       mY_1        mZ_1    )
(mX_2       mY_2        mZ_2    )

...

(mX_N-1     mY_N-1      mZ_N-1  )
)
```

The file tree below is an example of a case with `unstructured` deformation data.

Example of folder structure for `unstructured` deformation displacement data

```
`-- inputFolderName
    |-- 0.0000
    |   |-- pointDisplacement
    |   `-- points
    |-- 0.0500
    |   |-- pointDisplacement
    |   `-- points
    |-- 0.1000
    |   |-- pointDisplacement
    |   `-- points

    ...

    |   |-- 9.9000
    |   |-- pointDisplacement
    |   `-- points
    `-- 9.9500
        |-- pointDisplacement
        `-- points
```

3.2.1.2 Structured input data

For `structured` deformation data, the boundary condition does not require a `points` file. Instead, a file containing information about the structured data matrix is read at the beginning of the simulation. An example of this input file, which should be named `domainMatrixInfo` and stored in `constant/boundaryData/inputFolderName`, is seen below.

Example of file `domainMatrixInfo`

```
// Motion data matrix information.
// Line 1: reference point      (x,y,z).
// Line 2: voxels sizing        (dx,dy,dz).
// Line 3: matrix dimensions    (nx,ny,nz).
3
(
(x_ref  y_ref   z_ref )     // reference point coordinates
(d_X    d_Y     d_Z   )     // spacing between matrix voxels
(N_x    N_y     N_z   )     // dimensions of the 3-dimensional matrix
)
```

The first input above is a reference point for the matrix, corresponding to the coordinates of its first element, $(x_{\text{ref}}, y_{\text{ref}}, z_{\text{ref}})$. The second input indicates the matrix spacing (or voxel sizes) in each direction, Δx, Δy and Δz. The last input corresponds to the 3D matrix dimensions, N_x, N_y N_z.

For an arbitrary point in this matrix, characterised by three indices $i_{x,p}$, $i_{y,p}$ and $i_{z,p}$, ranging from 0 to $(N_x - 1)$, $(N_y - 1)$, $(N_z - 1)$, its coordinates x_p, y_p and z_p are

$$\begin{aligned} x_p &= x_{\text{ref}} + i_{x,p}(\Delta x) \\ y_p &= y_{\text{ref}} + i_{y,p}(\Delta y) \\ z_p &= z_{\text{ref}} + i_{z,p}(\Delta z) \ . \end{aligned} \tag{3.1}$$

Since the motion data in the `pointDisplacement` or `pointMotionU` files must be provided as a list to OpenFOAM, an single index m_p must be calculated for this point in order to retrieve its value. For this reason, it is important that the values in the list are sorted correctly, so that $i_{x,p}$, $i_{y,p}$ and $i_{z,p}$ may be used to calculate m_p as

$$m_p = i_{x,p} + i_{y,p}(N_x) + i_{z,p}(N_x \cdot N_y) \ . \tag{3.2}$$

This equation appears multiple times in `timeVaryingMotionInterpolationPointPatchField.C` when retrieving data for `structured` inputs. Following Equation (3.2) is of importance when the motion data is processed as a 3D matrix and later converted to a list. Examples of this task are seen in the tutorials described in Chapter 4 utilising Python and `numpy` matrices.

The example folder structure for the deformation data of type `structured` is seen below, with the `domainMatrixInfo` at the root of the `inputFolderName` folder.

Example of folder structure for `structured` deformation velocity data

```
`-- inputFolderName
    |-- 0.0000
    |   `-- pointMotionU
    |-- 0.0500
    |   `-- pointMotionU
    |-- 0.1000
    |   `-- pointMotionU

    ...

    |-- 9.9000
    |   `-- pointMotionU
    |-- 9.9500
    |   `-- pointMotionU
    `-- domainMatrixInfo
```

3.3 Types of motion interpolation

In this section the three available interpolation types available in `timeVaryingMotionInterpolation` will be briefly described. Details about their implementation in the code are later provided in Section 3.4.4.

3.3.1 Nearest value

The `interpolationType` of type `nearest` is not technically an interpolation method, but works by directly assigning the value at the closest data point to the boundary points. Although fast in comparison to the other available methods, the values are assigned to the boundary without any considerations for how close or far the mesh points are to the data point. This may lead to poor quality results in cases where a coarse point cloud with motion data is used. This method is useful when the motion information is provided as a point cloud with sufficient density, so that interpolation between multiple data points does not alter the results significantly.

In a hypothetical ideal case, no two or more mesh points would use the same data point for motion information, while no unused data points would exist. For complex motion this may not be easy of even possible, and the recommended usage of this method corresponds to a point cloud with spacing similar or smaller to that of the boundary mesh.

3.3.2 Inverse distance interpolation

For the `inverseDist` option, the interpolation of the data is performed by an averaging weighted by the inverse of the distance between a given mesh point and the data points within a user specified maximum distance (`inverseDistRadius`). For a given mesh point $\boldsymbol{X}_p$, the distance to all data points $\boldsymbol{X}_i$ is calculated. All data points whose distance is smaller than `inverseDistRadius` $R_{\min}$ are used to calculate the interpolated value

$$V_p = \frac{\sum_{i=1}^{n} w_i V_i}{\sum_{i=1}^{n} w_i} \text{ for all } i \text{ where } \operatorname{dist}(\boldsymbol{X}_p, \boldsymbol{X}_i) \leq R_{\min} \ , \tag{3.3}$$

where V_i is the field value at each used data point i and w_i is the corresponding weight factor

$$w_i = \left(\frac{1}{\operatorname{dist}(\boldsymbol{X}_p, \boldsymbol{X}_i)} \right)^p \ . \tag{3.4}$$

with p being an arbitrary exponent ≥ 1 that affects the weighting and with this the smoothness of the calculated results. If any point in the motion data coincides with a boundary point's coordinates ($\operatorname{dist}(\boldsymbol{X}_p, \boldsymbol{X}_i) = 0$), the value at the data point is directly assigned to the boundary point. Although

this is unlikely to happen for complex, unstructured meshes, it could occur in structured meshes using structured motion data sets. This methodology was first described by Shepard and follows the suggested selection of nearby points using an arbitrary distance criterion [3].

3.3.3 Trilinear interpolation

For a point inside a the three-dimensional matrix in the `structured` motion data, a value for the interpolated variable may be calculated by a trilinear interpolation (interpolation along x, y and z-axes). This can be accomplished with a series of seven linear interpolations using the eight nearest vertices, as seen in Figure 3.3. The algorithm for the trilinear interpolation presented here follows the same methodology presented by Kang in Chapter 9 of the book *Computational Color Technology* [4].

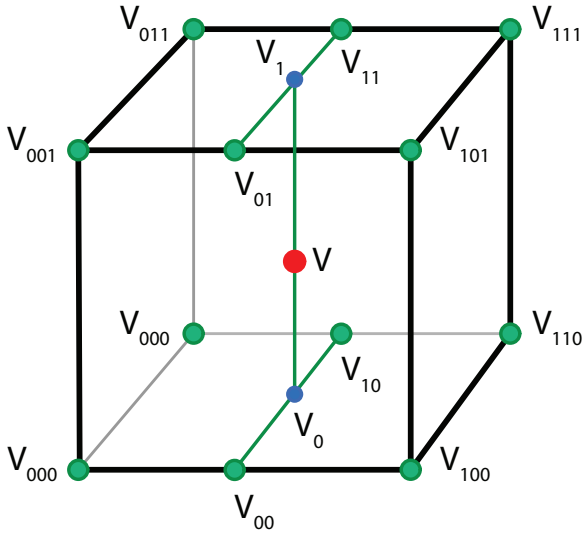


Figure 3.3: Trilinear interpolation to a point from 8 neighbour values[2].

For a point p inside a given voxel with edges values V_{XXX} numbered as seen in Fig. 3.3, the first four linear interpolations are performed along the x-axis

$$\begin{aligned} V_{00} &= V_{000}\,(1-l_x) + V_{100} l_x \ , \\ V_{10} &= V_{010}\,(1-l_x) + V_{110} l_x \ , \\ V_{01} &= V_{001}\,(1-l_x) + V_{101} l_x \ , \\ V_{11} &= V_{011}\,(1-l_x) + V_{111} l_x \ , \end{aligned} \tag{3.5}$$

followed by two interpolations along y

$$\begin{aligned} V_0 &= V_{00}\,(1-l_y) + V_{10} l_y \ , \\ V_1 &= V_{01}\,(1-l_y) + V_{11} l_y \ , \end{aligned} \tag{3.6}$$

and one along z

$$V = V_0\,(1-l_z) + V_1 l_z. \tag{3.7}$$

The interpolation factors l_x, l_y and l_z represent normalised terms for the linear interpolations ranging from 0 to 1 and calculated as

$$\begin{aligned} l_x &= \frac{x_p - x_0}{x_1 - x_0} \ , \\ l_y &= \frac{y_p - y_0}{y_1 - y_0} \ , \\ l_z &= \frac{z_p - z_0}{z_1 - z_0} \ , \end{aligned} \tag{3.8}$$

[2]Modified from original figure available at https://upload.wikimedia.org/wikipedia/commons/9/97/3D_interpolation2.svg, licensed under CC BY-SA 3.0.

where (x_0, y_0, z_0) and (x_1, y_1, z_1) represent the coordinates of points 000 and 111 in the interpolation lattice, and (x_p, y_p, z_p) represents the point at which data will interpolated.

For a simpler implementation of this method, the interpolations may be grouped differently, with V calculated as the linear interpolation along z of two bilinear interpolations in x and y performed on the lower $(z = z_0)$ and upper $(z = z_1)$ faces of the lattice. This is the approach used in the implementation of this method, described in Section 3.4.4.3.

3.4 Detailed description of `timeVaryingMotionInterpolation`'s code

3.4.1 Constructor for `timeVaryingMotionInterpolation`

The multiple types of constructors available for the developed boundary condition, detailed in `timeVaryingMotionInterpolationPointPatchField.H`, are the same already present in the class `timeVaryingMappedFixedValue`. They were adapted to deal with the private data included in `timeVaryingMotionInterpolation`, accounting for elements that were either removed from the original code or added in for the new one. Among these constructors, the one relevant for the boundary condition initialisation is the one based on a patch, an internal field and a dictionary, with its declaration seen below.

Constructor declaration for `timeVaryingMotionInterpolation`

```
//- Construct from patch, internal field and dictionary
timeVaryingMotionInterpolationPointPatchField
(
    const pointPatch&,
    const DimensionedField<Type, pointMesh>&,
    const dictionary&
);
```

The relevant code in this function is given in `timeVaryingMotionInterpolationPointPatchField.C` and will not be included in this report due to its length. It is responsible for initialisation of the boundary field values, as well as extraction and verification of the input variables included in the boundary condition specifications in the case time folders. The compatibility between inputs is also verified, assuring, for example, that the `trilinear` interpolation type is not used when unstructured motion data is provided, or that a valid `inverseDistRadius` value is given for the inverse distance interpolation method. The last step in this function is the initialisation of the field values in case the `value` parameter is used in the boundary condition specifications.

3.4.2 The `updateCoeffs` function

As seen in Sections 2.3.1 and 2.3.2, describing the `solve` function in both `velocityLaplacian` and `displacementLaplacian`, the boundary conditions are updated through a function standardly named `updateCoeffs` in OpenFOAM. The contents of this function, are included below, copied from `timeVaryingMotionInterpolationPointPatchField.C`

Function `updateCoeffs` for `timeVaryingMotionInterpolation`

```
template<class Type>
void Foam::timeVaryingMotionInterpolationPointPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    checkTable();
```

```
    // Interpolate between the sampled data
    scalar deltaTime = this->db().time().value() - sampleTimes_[startSampleTime_].value();
    if (endSampleTime_ == -1 || deltaTime < SMALL)
    {
        // only start value
        if (debug)
        {
            Pout<< "updateCoeffs : Sampled, non-interpolated values"
                << " from start time:"
                << sampleTimes_[startSampleTime_].name() << nl;
        }

        this->operator==(startSampledValues_);
    }
    else
    {
        scalar start = sampleTimes_[startSampleTime_].value();
        scalar end = sampleTimes_[endSampleTime_].value();

        scalar s = (this->db().time().value()-start)/(end-start);

        if (debug)
        {
            Pout<< "updateCoeffs : Sampled, interpolated values"
                << " between start time:"
                << sampleTimes_[startSampleTime_].name()
                << " and end time:" << sampleTimes_[endSampleTime_].name()
                << " with weight:" << s << endl;
        }
        this->operator==((1-s)*startSampledValues_ + s*endSampledValues_);
    }


    if (debug)
    {
        Pout<< "updateCoeffs : set fixedValue to min:" << gMin(*this)
            << " max:" << gMax(*this)
            << " avg:" << gAverage(*this) << endl;
    }

    fixedValuePointPatchField<Type>::updateCoeffs();
}
```

When called, this function first determines whether a boundary condition update is required. If so, the member function `checkTable` is called, updating the values the variables `startSampledValues_` and `endSampledValues_`. These represent the motion data at the available time folder that are smaller and larger than the simulation time, respectively. Finally, the simulation time value is compared to the available motion data at time `sampleTimes_[startSampleTime_].value()` in the time folders of the motion data. If the simulation time overlaps with a motion data time, or if it is larger than the last available data time, the values of `startSampledValues_` are assigned to the boundary points. In case the simulation time lies between two available data times, the values interpolated in space are linearly interpolated in time, in `((1-s)*startSampledValues_ +s*endSampledValues_)`. The final step of this function consists in calling the `updateCoeffs` function in the class `fixedValuePointPatchField<Type>`, which will not be discussed further here.

3.4.3 The `checkTable` function

The function `checkTable` in `timeVaryingMotionInterpolation` is responsile for reading all the motion data in `constant/boundaryData/inputFolderName`, as well as determining which interpolation function to use based on inputs. It will be described here in a general manner and the reader is referred to `timeVaryingMotionInterpolationPointPatchField.C` for deeper inspection.

3.4.3.1 Initialisation of re-usable parameters

During the first time `checkTable` is used this function initialises some relevant variables and inspects the contents of folder `inputFolderName` for the available time sub-folders. This is performed by the code following the `if`-statement `if(startSampleTime_==-1&&endSampleTime_==-1)`. If the `inputType` is set to `unstructured`, the only task performed is the initialisation of the member variable `sampleTimes_` by inspection of data folder contents. If, instead, `inputType` is `structured`, the file `domainMatrixInfo` will be read for extraction of the structured matrix information. Additionally, for `structured` data where the `trilinear` option is not chosen for `interpolationType`, a `pointField` representing all points in the structured data matrices is assigned to `startSamplePoints_` using Equation (3.1). This is required for the `nearest` and `inverseDist` methods, which for structured data are less efficient than the `trilinear` method.

3.4.3.2 Reading of motion data files

The next steps in `checkTable` are related to loading the provided motion data relevant for the simulation time step. This is performed by defining the data times prior and after the simulaion time, assigning their values to `startSampleData_` and `endSampleData_`, respectively.

To determine which data time folders will be read, the function `findTime` is used. Its main purpose it to determine the two time values in `sampleTimes_` nearest to the simulation time value, to determine the motion data files which should be read and used in the data interpolation. This function and the other time-related function `timeNames` are static members of the OpenFOAM class `pointToPointPlanarInterpolation`, allowing them to be invoked without an instance of this class.

Once the data times between which the simulation time lies are known, the necessary files are read and the values are assigned to `startSampleData_` and `endSampleData_`. For `unstructured` data, files containing the point coordinates are read and values are assigned to the `startSamplePoints_` and `endSamplePoints_`. In case the simulation time becomes greater than the last available data time, the variables `endSampleData_`, `endSamplePoints_` and `endSampledValues_` are instead cleared during this step, as they are no longer used.

3.4.3.3 Data interpolation

The last step in this function is defining the variable `interpolateEnd` and calling the appropriate interpolation function. The Boolean `interpolateEnd` determines whether data from both `startSampleData_` and `endSampleData_` should be interpolated, or if only `startSampleData_` should be used. This is required not only if the simulation time is outside the provided data time bounds (and the motion data has been cleared), but is also used to accelerate the interpolation process when the current time matches one of the provided data time values and no interpolation in time must be performed in `updateCoeffs`.

The choice of interpolation function is made based on the value of `interpolationType`. For all three available options, a reference to the boundary points is provided, as well as the value of `interpolateEnd`. The functions `applyNearestValues`, `applyInverseDistanceInterpolation` and `applyTrilinearInterpolation` are used for the three interpolation methods described in Section 3.3 and their implementation will be briefly described in Section 3.4.4.

All three of these functions assign motion values corresponding to the mesh points to the variables `startSampledValues_` and `endSampledValues_`, used for the final field values in `updateCoeffs`.

3.4.4 Implementations of motion interpolation

3.4.4.1 Nearest value

This method, as described in Section 3.3.1, is implemented in the function `applyNearestValues`, which utilises OpenFOAM's existing class `pointToPointPlanarInterpolation` to create interpolator objects. Despite the fact that this class' original purpose is to perform interpolation from one

set of unstructured points to another using 2D Delaunay triangulation[3], it can conveniently be used for the nearest value assignment if constructed as seen in the example code below.

Example `pointToPointPlanarInterpolation` object for nearest value

```
// Create a generic interpolator pointer
autoPtr<pointToPointPlanarInterpolation> interpPtr;

interpPtr.reset
(
    new pointToPointPlanarInterpolation
    (
    sourcePoints,       // [sourcePoints]   Source points
    meshPts,            // [destPoints]     Target points
    0,                  // [perturb]        Perturbation (not used for nearest)
    true                // [nearestOnly]
    )
);

valuesAtMeshPoints = interpPtr().interpolate(sourceData);
```

The creation of a pointer for this interpolator instead of an instance of this class, seen above, allows for simpler code and less usage of memory. By resetting this pointer with `sourcePoints` as either `startSamplePoints_` or `endSamplePoints_`, it may be used to calculate values at the mesh points for the corresponding `sourceData` values.

3.4.4.2 Inverse distance interpolation

The function `applyInverseDistanceInterpolation` is used to interpolate motion data to boundary points using the inverse distance method. This function does not contain the actual interpolation code, but is responsible for calling the appropriate version another function called `inverseDistance` for the existing motion data values based on the data type (`structured` or `unstructured`) and the provided value of `interpolateEnd`.

Two version of `inverseDistance` exist in the code for different data types, built by overloading it with different inputs. The more general case of this function uses the boundary point coordinates, the motion data coordinates and the motion data values to return a `Foam::tmp<Foam::Field<Type>>` containing the motion data values interpolated to the mesh points. This first version is used for `unstructured` motion data or whenever `interpolateEnd` is `false` in `structured` motion data. The second version of `inverseDistance` does not return any values and is used to assign values directly to both `startSampledValues_` and `endSampledValues_` with `structured` motion data when `interpolateEnd` is `true`. In this case the function takes advantage of the fact that only one set of motion data points exists in `structured` data to accelerate the process. As the relevant aspects of these two functions do not differ significantly, only the first will be discussed here and the differences between the two implementations may be observed in detail in `timeVaryingMotionInterpolationPointPatchField.C`.

As seen in the following code excerpt, the distances to all motion data points is calculated for each point in the boundary. This task is performed by using the `SortableList<scalar>` object named `distSorted`. This class is updated at each mesh point and sorts its values while storing the original indices corresponding to the source array. This way, a `for`-loop can quickly calculate the inverse distance terms going through the distance values from closest for farthest to a boundary point, interrupting the loop as soon as the first distance greater than `inverseDistRadius` is found. During this loop the variables `interpNum` and `interpDen` represent the numerator and denominator in Equation (3.3). As seen in the variable `pointInvD` below, the implementation of the inverse distance method presented used an exponent $p = 1$ for simplicity. The case for $\text{dist}(\boldsymbol{X}_p, \boldsymbol{X}_i) = 0$ is covered using the `if`-statement `if(pointD<SMALL)` seen below. The final value for the motion data at each mesh point is given by `interpNum/interpDen` at the end. In the case seen below the `Foam::tmp<Foam::Field<Type>>` returned by `inverseDistance` is assigned by `applyInverseDistanceInterpolation` to the correct variable.

[3] See `src/meshTools/triSurface/triSurfaceTools/pointToPointPlanarInterpolation.H`.

Function `inverseDistance`

```
template<class Type>
Foam::tmp<Foam::Field<Type>> Foam::timeVaryingMotionInterpolationPointPatchField<Type>::
    inverseDistance
(
    const pointField& meshPts,
    const pointField samplePoints,
    const Field<Type> sampleData
) const
{
    tmp<Field<Type>> tfld(new Field<Type>(meshPts.size()));
    Field<Type>& fld = tfld.ref();

    // A sortable list for distances from mesh to point data
    SortableList<scalar> distSorted(samplePoints.size());

    forAll(meshPts, pp)
    {
        // Assign new values to SortableList and sort it
        distSorted = mag(meshPts[pp]-samplePoints);
        distSorted.sort();

        Type interpNum(Zero);
        scalar interpDen(Zero);

        forAll(distSorted,jj)
        {
            scalar pointD = distSorted[jj];
            if (pointD>inverseDistRadius_)
            {
                break;
            }

            label pointI = distSorted.indices()[jj];
            if (pointD < SMALL)
            {
                interpNum = sampleData[pointI];
                interpDen = 1.0;
                break;
            }

            scalar pointInvD = 1/(pointD);

            interpNum += pointInvD * sampleData[pointI];
            interpDen += pointInvD;
        }

        if ( interpDen == 0)
        {
            fld[pp] = Type(Zero);
        }
        else
        {
            fld[pp] = interpNum/interpDen;
        }
    }

    return tfld;
}
```

3.4.4.3 Trilinear interpolation

As mentioned in Section 3.3.3, `timeVaryingMotionInterpolation` uses the trilinear interpolation as a linear interpolation along z of two bilinear interpolations in x and y. This is accomplished by using the three member functions `linearInterpolation`, `bilinearInterpolation` and

`trilinearInterpolation` described here.

The first, `linearInterpolation`, receives two values and the normalised linear factor for the scaling of these values (l_x, l_y or l_z). It may be used for any linear interpolation between two values and is the base for the calculations returned for the other two function.

Function `linearInterpolation`

```
template<class Type>
Type Foam::timeVaryingMotionInterpolationPointPatchField<Type>::linearInterpolation
(
    const Type& edgeVal0,
    const Type& edgeVal1,
    const scalar& lf
)
{
    return edgeVal0*(1-lf)+edgeVal1*lf;
}
```

`bilinearInterpolation` performs a similar task to `linearInterpolation`, but in two dimensions. As seen below, it performs two linear interpolations using the first of two interpolation factors, and ultimately returns a linear interpolation between these using the second factor. This bilinear interpolation may be performed along any face of a chosen voxel and can be conveniently used to interpolate data to mesh points whose coordinates lie outside of the domains of the motion data matrix.

Function `bilinearInterpolation`

```
template<class Type>
Type Foam::timeVaryingMotionInterpolationPointPatchField<Type>::bilinearInterpolation
(
    const Type& edgeVal00,
    const Type& edgeVal10,
    const Type& edgeVal01,
    const Type& edgeVal11,
    const scalar& lf1,
    const scalar& lf2
)
{
    Type linA = linearInterpolation(edgeVal00,edgeVal10,lf1);
    Type linB = linearInterpolation(edgeVal01,edgeVal11,lf1);
    return linearInterpolation(linA,linB,lf2);
}
```

Finally, the function `trinearInterpolation` makes use of the previous two to calculate the final interpolated value. It performs two bilinear interpolations using the first two factors, and ultimately returns a linear interpolation between these.

Function `trilinearInterpolation`

```
template<class Type>
Type Foam::timeVaryingMotionInterpolationPointPatchField<Type>::trilinearInterpolation
(
    const Type& edgeVal000,
    const Type& edgeVal100,
    const Type& edgeVal010,
    const Type& edgeVal110,
    const Type& edgeVal001,
    const Type& edgeVal101,
    const Type& edgeVal011,
    const Type& edgeVal111,
    const scalar& lf1,
    const scalar& lf2,
    const scalar& lf3
)
{
    // First bilinear interpolation
    Type biLinA = bilinearInterpolation
    (
        edgeVal000,
        edgeVal100,
        edgeVal010,
        edgeVal110,
        lf1,
        lf2
    );
    // Second bilinear interpolation
    Type biLinB = bilinearInterpolation
    (
        edgeVal000,
        edgeVal100,
        edgeVal010,
        edgeVal110,
        lf1,
        lf2
    );
    // Interpolate linearly between the two previous
    return linearInterpolation(biLinA,biLinB,lf3);
}
```

Although the implementation of `trilinearInterpolation` does not require the interpolations to be performed in any order of the Cartesian axes, its usage is only performed in the same sequence described in Section 3.3.3, so that `lf1`, `lf2` and `lf3` are in fact l_x, l_y and l_z.

These functions are called by the main interpolation function `applyTrilinearInterpolation`, which determines the position of mesh points within the structured point grid and provides the inputs to the required functions. Due to its length and repetitiveness, the code for the function `applyTrilinearInterpolation` will not be presented. The reader may refer to the provided file `timeVaryingMotionInterpolationPointPatchField.C` for details.

The first part of `applyTrilinearInterpolation` comprises of initialising variables used throughout the function and verifying if the input data is three-dimensional or not. In case a 2D or 1D dataset is provided, data are interpolated to all mesh points, even if `intOutsideBounds_` is `false`. It then proceeds to evaluate values for each point in the boundary mesh.

The indices corresponding to point P_{000} in Figure 3.3 (`iX`, `iY`, `iZ` in the code) are obtained as

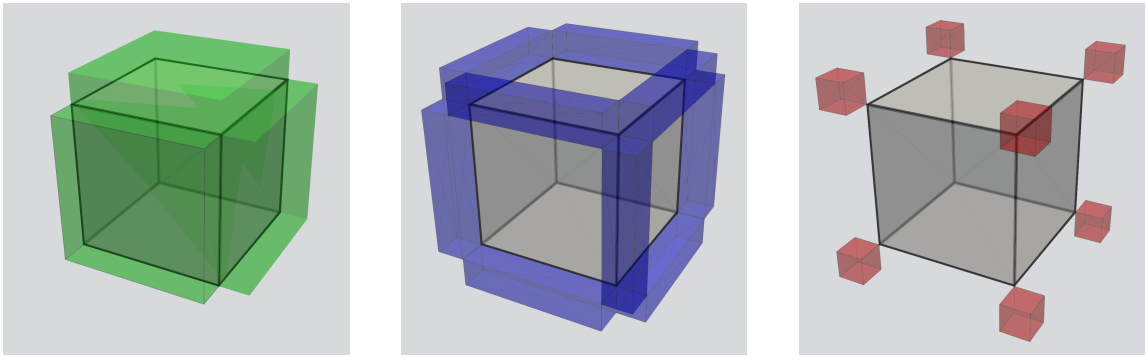
$$\begin{aligned} i_{x,000} &= (x_p - x_{\text{ref}})/\Delta x \ , \\ i_{y,000} &= (y_p - y_{\text{ref}})/\Delta y \ , \\ i_{z,000} &= (y_p - z_{\text{ref}})/\Delta z \ . \end{aligned} \tag{3.9}$$

If the calculated indices are within a 3D matrix' bounds, the `trilinearInterpolation` function is called after the indices for the remainign points in Figure 3.3 are determined and l_x, l_y, l_z are calculated.

If the conditions for `trilinearInterpolation` are not fulfilled, the next step is verifying whether any further calculations are needed and, if so, which method to use. If the motion data is not three-dimensional or if `intOutsideBounds_=true`, the values $i_{x,000}$, $i_{y,000}$ and $i_{z,000}$ are compared to the matrix dimensions to determine the mesh point's position with regards to the data matrix. One of the three options below is employed for any point p outside of the matrix bounds, depending on this verification:

1. If only one of the indices is outside the 3D matrix bounds, a bilinear interoplation is used (Figure 3.4a).
2. If two of the indices are outside the 3D matrix bounds, a linear interoplation is used (Figure 3.4b).
3. If all indices are outside the 3D matrix bounds, no interpolation is used and the value of the nearest edge is assigned (Figure 3.4c).

This options are illustrated in Figure 3.4 the regions in green, blue and red represent possible locations for a mesh point with regards to the three-dimensional matrix region, shown in grey.



(a) Bilinear. (b) Linear interpolation. (c) Direct value assignment.

Figure 3.4: Exception cases in the trilinear interpolation function.

Chapter 4

Description of provided tutorials

In this chapter the two provided tutorials will be briefly introduced. The purpose of these tutorials is the demonstration of the boundary condition `timeVaryingMotionInterpolation` for mesh deformation fields in OpenFOAM. Their settings were intended for faster computations, not at accurate simulations of the flow. It should be noted that adjustments had to be made to the time step sizes in both cases to guarantee that the mesh deformation fields were well distributed in the mesh. Although not presented here, the reader is encouraged to investigate the effects of different simulation time steps in the results and mesh quality.

The provided tutorials folder contains the scripts and folders shown below.

Contents of folder `tutorials`

```
tutorials
|-- Allclean_airfoil
|-- Allclean_deformingCylinder
|-- Allrun_airfoil
|-- Allrun_deformingCylinder
|-- airfoil
|-- deformingCylinder
|-- supportFunctions.py
`-- supportFunctions.sh
```

The folders `airfoil` and `deformingCylinder` contain the base-cases for the two example tutorials. The scripts `Allrun_airfoil`, `Allrun_deformingCylinder`, `Allclean_airfoil` and `Allclean_deformingCylinder` may be used for batch running the tutorials with multiple deformation settings, as well as removing all the simulation data. The automation of tasks in the first two of these scripts depends on the functions included in `supportFunctions.sh`. The tutorials also require the additional file `supportFunctions.py`, which contains functions used for exporting the data in a format compatible with the descriptions in Section 3.2.1, as well as function for part of the geometrical calculations.

4.1 Deforming airfoil

The first case that exemplifies the use of `timeVaryingMotionInterpolation` is a deforming two-dimensional airfoil. Although a simple case at a first glance, the difficulties in mesh deformation cases become evident once tests with this tutorial are performed. All data required for executing the simulations are included in the provided `airfoil` folder, with contents seen below.

Clean folder structure for airfoil tutorial

```
airfoil
|-- 0_orig
|   |-- U
|   |-- nuTilda
|   |-- nut
```

```
|   |-- p
|   |-- pointDisplacement
|   `-- pointMotionU
|-- Allclean
|-- Allrun
|-- Allrun_prepare
|-- README
|-- constant
|   |-- dynamicMeshDict
|   |-- transportProperties
|   `-- turbulenceProperties
|-- createNaca4dig.py
|-- curiosityFluidsAirfoilMesher.py
`-- system
    |-- controlDict
    |-- fvSchemes
    `-- fvSolution
```

4.1.1 Creation of geometry and motion data

This tutorial utilises two provided Python scripts to generate the geometry information for a NACA 4-digit airfoil, following information and instructions from the book *Theory of Wing Sections, Including a Summary of Airfoil Data* by Abbot [5].

The geometrical and motion parameters are created by the script `createNaca4dig.py`. The *Inputs* section of this file is shown below.

Inputs to *airfoil* tutorial in `createNaca4dig.py`

```
# ============================================================================
# %% Import dependencies
# ============================================================================
import numpy as np
import sys
import scipy.optimize as optimize
# appending the tutorials root for support functions
sys.path.append('../')
import supportFunctions as sF

# ============================================================================
# %% Input values
# ============================================================================
# Define a NACA 4 digit's parameters
mm = 2                  # Maximum camber (percentage of chord)
pp = 4                  # Location of maximum thickness (tenths of chord)
tt = 12                 # Maximum thickness (percentage of chord)

# DEFORMATION PARAMETERS
defPeri = 20                    # Period for the deformation motion [s]
defAmpl = 0.1                   # Amplitude for the motion
# (this is used to modify the camber equation for the aifoil)

# SIMULATION TIME PARAMETERS
staTime = 0                     # Start time [s]
endTime = defPeri + staTime     # End time (better if >= defPeri+staTime)

# DISCRETISATION PARAMETERS
discAir = 200           # Discretisation of the airfoil surface
discTim = 81            # Discretisation of the time for a full motion period

# STRUCTURED OUTPUT PARAMETERS (only used when structuredData is True)
structData = True
structDisX = 101                # Discretisation in X
structDisY = 51                 # Discretisation in Y
```

The first portion of the inputs is used to determine the NACA code of the selected airfoil. For ease of use, the parameters for the commonly used NACA 2412 airfoil have been provided. The

following parameters refer to the motion of the airfoil's trailing edge computed using a modified version of the standard NACA camber equation. By changing the y-coordinate of the trailing edge point in this equation, the airfoil shape is altered. The script guarantees that the length of the airfoil is maintained. The discretisation parameters are used for spatial and time subdivisions. Another relevant parameter is the Boolean variable `structData`, which determines if the output motion data is structured or unstructured. If set to `True`, the motion data originally calculated at points is interpolated to a two-dimensional uniform grid. The parameter `exportVelocities` is also of interest, as it determines whether the motion data will be exported as displacements (`pointDisplacement` files) or velocities (`pointMotionU` files). Last, the Boolean `compData` does not alter the output values, but if enabled applies compression to the exported file, thus reducing their size. When executed, `createNaca4dig.py` exports a geometry file for the chosen airfoil called `naca4digit`, saved to the case folder containing the x and y-coordinates of the airfoil surface. The motion data is exported to the path set in the `outputPath` variable.

The second provided Python script, `curiosityFluidsAirfoilMesher.py`[1], was published and described in the website *curiosityFluids* [6]. It is used to read the file `naca4digit` and create a `blockMeshDict` with instructions for a structured mesh around this airfoil.

The example mesh for the NACA 2412 airfoil created with `blockMesh` for this tutorial is shown in Figure 4.1.

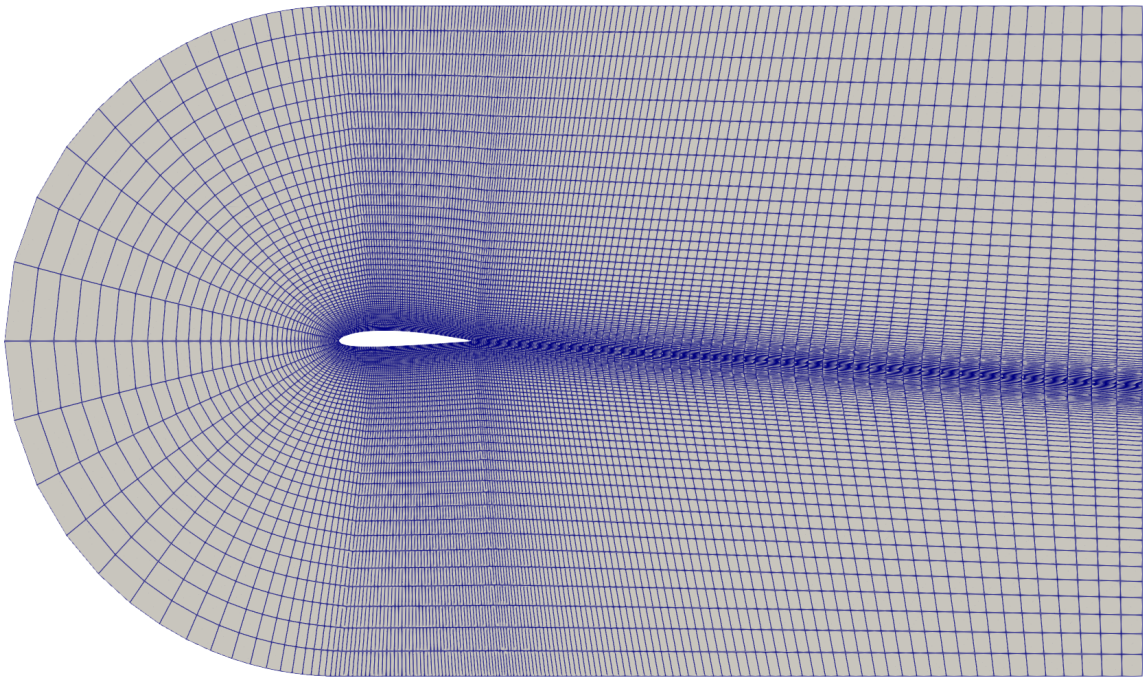


Figure 4.1: Trilinear interpolation to a point from 8 neighbour values.

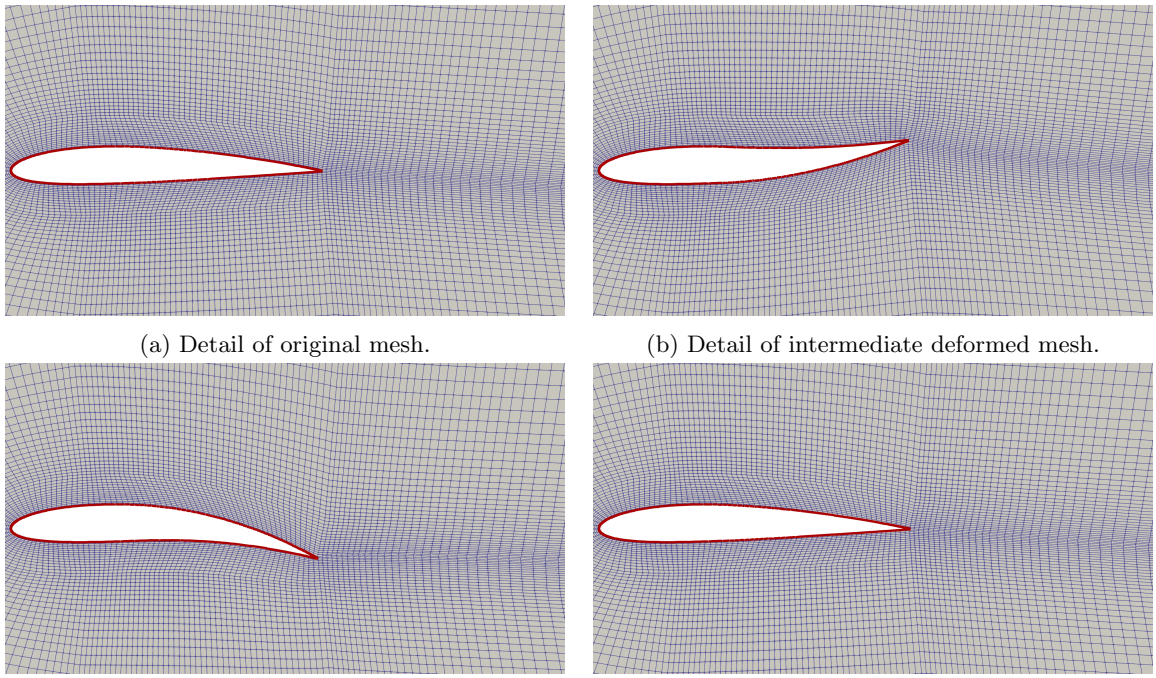
The steps described here, including execution of Python scripts and the mesh creation generation, were automated in the provided file `Allrun_prepare`.

4.1.2 Airfoil mesh deformation

The deformation of the mesh may be accomplished separately with the utility `moveDynamicMesh` or while performing the flow calculations with `pimpleFoam`. The motion solver in `dynamicMeshDict` must be adjusted to `displacementLaplacian` or `velocityLaplacian` depending on the choice of

[1] Available at https://github.com/curiosityFluids/curiosityFluidsAirfoilMesher and licensed under GNU General Public License v3.0.

motion data type in `createNaca4dig.py`. For ease of use, the `Allrun` file provided executes all tasks required. Detail images of the mesh in four different instances are shown in Figure 4.2. Figure 4.2a shows the initial mesh as well as the points created by `createNaca4dig.py`, seen in red. The following three figures show the same mesh and points during and at the end of simulations, illustrating the maintenance of the mesh quality throughout the deformation process.



(a) Detail of original mesh.

(b) Detail of intermediate deformed mesh.

(c) Detail of intermediate deformed mesh.

(d) Detail of final deformed mesh.

Figure 4.2: Example mesh deformation for the airfoil tutorial.

4.1.3 Batch execution

The provided script `Allrun_airfoil` may be used to create multiple copies of the base `airfoil` case and execute them with different settings for the data file types (unstructured/structured), types of interpolation and types of data (velocity or displacement).

4.2 Deforming cylinder

The second provided tutorial is based on the deformation of a cylindrical vessel in a axisymmetric manner. The points on the side of the cylinder are displaced radially in a sinusoidal motion in time. All required files are included in the base case folder, are seen below.

Clean folder structure for deformingCylinder tutorial

```
deformingCylinder
|-- 0_orig
|   |-- U
|   |-- alpha.water
|   |-- p_rgh
|   |-- pointDisplacement
|   `-- pointMotionU
|-- Allclean
|-- Allrun
|-- Allrun_prepare
|-- README
```

```
|-- constant
|   |-- dynamicMeshDict
|   |-- g
|   |-- transportProperties
|   `-- turbulenceProperties
|-- createMotion.py
`-- system
    |-- blockMeshDict.m4
    |-- controlDict
    |-- decomposeParDict
    |-- fvSchemes
    |-- fvSolution
    `-- setFieldsDict
```

4.2.1 Creation of geometry and motion data

The motion information for the simulations is created by the script `createMotion.py`, with inputs seen below.

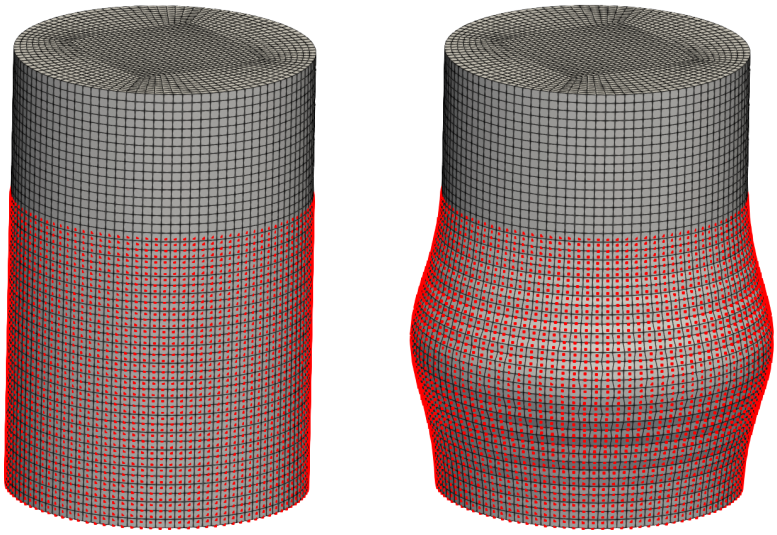
Inputs to *deformingCylinder* tutorial in `createMotion.py`

```
# ============================================================================
# %% Import dependencies
# ============================================================================
import numpy as np
import sys
from scipy.stats import norm
# appending the tutorials root for support functions
sys.path.append('../')
import supportFunctions as sF

# ============================================================================
# %% Input values
# ============================================================================
# GEOMETRICAL PAARAMETERS (MATCHING CFD MESH)
cylRadius_ini = 0.5             # Cylinder radius [m]
cylHeight_sta = 0.0             # Minimum height value (Z axis) [m]
cylHeight_end = 1.0             # Maximum height value (Z axis) [m]

# SIMULATION TIMES VALUES
timeVal_sta = 0.0               # Initial deformation time []
timeVal_end = 20.0              # Final deformation time [s]

# DEFORMATION SHAPE PARAMETER
deform_StDev = 0.2              # Normal distribution curve std dev. [m]
deform_Ampli = 0.05             # Cylinder deformation amplitude [m]
deformPeriod = 3.0              # Cylinder deformation period [s]
deformStartT = 0
# The deformation is based on a normal distribution curve with a mean in the
# middle of the cylinder height. The bell shape width in the height direction
# determined by the standard deviation value "deform_StDev".
# The deformation oscillates in time using a sinus function with period
# "deformPeriod" and amplitue "deform_Ampli" representing the displacement at
# the peak of the normal distribution curve.

# DISCRETISATION VALUES
cylHeight_dis = 51              # Discretisation in cylinder height
cylAngles_dis = 100             # Discretisation in cylinder angle coordinate
deforTime_dis = 61              # Discretisation in time

# Displacements or velocities?
exportVelocities = True         # If false, displacement values are exported
```

The script requires geometrical and motion information that must be compatible with the simulation settings in the OpenFOAM dictionaries. The deformation of the cylinder's sides is performed using the shape of a normal distribution curve varying in time. Changes to the parameters `deform_StDev`

and `deform_Ampli` affect the shape of the normal distribution curve and the deformation amplitude, respectively. The period of the deformation motion may be adjusted using the parameter `deformPeriod`. The same variables as in the `airfoil` tutorial are used to define whether displacement of velocity information is created, to define if the data will be interpolated to a uniform grid or output as a point cloud, as well as determining if compression will be applied to the files. In case structured data is exported, the motion of the cylinder's surface is interpolated to a three dimensional uniform grid.

Execution of `createMotion.py` will output the necessary motion data to the correct path. The cylinder mesh may be generated using the `blockMesh` utility. The provided `blockMeshDict.m4` script should be used to create a `blockMeshDict`, and should be changed in case the geometry information in `createMotion.py` is altered. As in the `airfoil` tutorial, all steps described here may be executed by running the `Allrun_prepare` script from the `deformingCylinder` case. Examples of the original and deformed meshes for this case are shown in Figures 4.3a and 4.3b.



(a) Initial cylinder mesh and points.

(b) Deformed cylinder mesh and points.

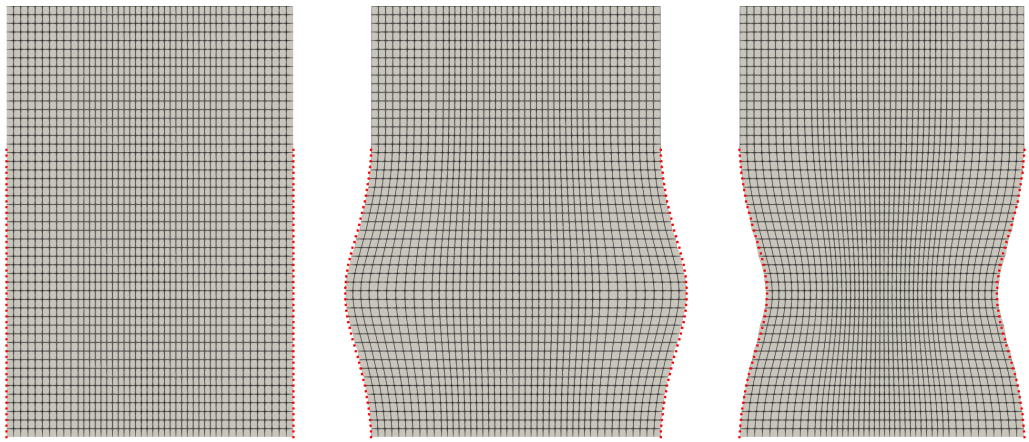
Figure 4.3: Cylinder mesh and point at initial and deformed conditions.

4.2.2 Cylinder mesh deformation

The mesh motion for the `deformingCylinder` may be performed either by `moveDynamicMesh` or running the flow simulations with the Volume Of Fluids (VOF) solver `interFoam`, which may be run using the script `Allrun` inside the case folder. The settings for the type of motion solver in `dynamicMeshDict` must be compatible with the type of motion data exported. If `interFoam` is used, the cylinder is treated as a vessel partially filled with liquid, so that the deformations affect the level of this liquid inside the container. Figure 4.4 shows slices of the cylindrical mesh in different instants, to illustrate the deformation's effects to the mesh.

4.2.3 Batch execution

The script `Allrun_deformingCylinder` may be used to create multiple copies of the base `deformingCylinder` case and execute them with different settings for the data file types (unstructured/structured), type of interpolation and types of data (velocity or displacement).



(a) Initial mesh and points. (b) Deformed mesh and points. (c) Deformed mesh and points.

Figure 4.4: Example mesh deformation for the deforming cylinder.

Bibliography

[1] H. Jasak and Z. Tukovic, "Automatic mesh motion for the unstructured finite volume method," *Transactions of FAMENA*, vol. 30, no. 2, pp. 1–20, 2006.

[2] R. Löhner and C. Yang, "Improved ALE mesh velocities for moving bodies," *Communications in Numerical Methods in Engineering*, vol. 12, no. 10, pp. 599–608, 1996.

[3] D. Shepard, "A two-dimensional interpolation function for irregularly-spaced data," in *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, (New York, NY, USA), pp. 517–524, Association for Computing Machinery, Jan. 1968.

[4] Kang and H. R, *Computational Color Technology*. Bellingham, Wash: SPIE Publications, May 2006.

[5] I. H. Abbott and A. E. V. Doenhoff, *Theory of Wing Sections, Including a Summary of Airfoil Data*. Dover Publications, Jan. 1959.

[6] curiosityFluids, "Automatic Airfoil C-Grid Generation for OpenFOAM – Rev 1." https://curiosityfluids.com/2019/04/22/automatic-airfoil-cmesh-generation-for-openfoam-rev-1/, Apr. 2019.

Study questions

How to use it:

- How can dynamic meshes be used for arbitrary deformation in OpenFOAM with different types of motion and motion solvers?
- How can we prescribe information from a deformation field into a CFD mesh?

The theory of it:

- How are deformations to a dynamic mesh accomplished in OpenFOAM?
- How are mesh deformation calculated in OpenFOAM with Laplacian solvers?
- How can arbitrary mesh deformations be interpolated from data points to mesh points?

How it is implemented:

- How is the calculation of boundary points motion implemented in `timeVaryingMotionInterpolation`?

How to modify it:

- How can the provided boundary condition code be modified or expanded?
- How can a new interpolation algorithm be integrated into `timeVaryingMotionInterpolation`?

Appendix A

Accompanying files

The acompanying files to this report are included in two main folders, `myFvMotionSolver` and `tutorials`. Their contents are included below for easy referencing:

A.1 Contents of `myFvMotionSolver`

This folder contains the boundary developed condition code and instructions to compile it as part of a library named `myFvMotionSolvers`, which also includes all the items from the original OpenFOAM `fvMotionSolvers`.

Contents of folder `myFvMotionSolver`

```
myFvMotionSolver
|-- Allwclean
|-- Allwmake
|-- Make
|   |-- files
|   `-- options
`-- pointPatchFields
    `-- derived
        `-- timeVaryingMotionInterpolation
            |-- timeVaryingMotionInterpolationPointPatchField.C
            |-- timeVaryingMotionInterpolationPointPatchField.H
            |-- timeVaryingMotionInterpolationPointPatchFields.C
            `-- timeVaryingMotionInterpolationPointPatchFields.H
```

A.2 Contents of `tutorials`

The `tutorials` folder contains all the required files for the testing of the `timeVaryingMotionInterpolation` boundary condition. The two folders `airfoil` and `deformingCylinder` are base cases which may be run independently or with multiple different settings using the provided `Allrun_airfoil` and `Allrun_deformingCylinder` scripts.

Contents of folder `myFvMotionSolver`

```
tutorials
|-- Allclean_airfoil
|-- Allclean_deformingCylinder
|-- Allrun_airfoil
|-- Allrun_deformingCylinder
|-- airfoil
|   |-- 0_orig
|   |   |-- U
|   |   |-- nuTilda
|   |   |-- nut
```

```
|   |   |-- p
|   |   |-- pointDisplacement
|   |   `-- pointMotionU
|   |-- Allclean
|   |-- Allrun
|   |-- Allrun_prepare
|   |-- README
|   |-- constant
|   |   |-- dynamicMeshDict
|   |   |-- transportProperties
|   |   `-- turbulenceProperties
|   |-- createNaca4dig.py
|   |-- curiosityFluidsAirfoilMesher.py
|   `-- system
|       |-- controlDict
|       |-- fvSchemes
|       `-- fvSolution
|-- deformingCylinder
|   |-- 0_orig
|   |   |-- U
|   |   |-- alpha.water
|   |   |-- p_rgh
|   |   |-- pointDisplacement
|   |   `-- pointMotionU
|   |-- Allclean
|   |-- Allrun
|   |-- Allrun_prepare
|   |-- README
|   |-- constant
|   |   |-- dynamicMeshDict
|   |   |-- g
|   |   |-- transportProperties
|   |   `-- turbulenceProperties
|   |-- createMotion.py
|   `-- system
|       |-- blockMeshDict.m4
|       |-- controlDict
|       |-- decomposeParDict
|       |-- fvSchemes
|       |-- fvSolution
|       `-- setFieldsDict
|-- supportFunctions.py
`-- supportFunctions.sh
```