COMPLEX MESH DEFORMATIONS IN OPENFOAM:

A CUSTOM BOUNDARY CONDITION FOR PRESCRIBED MESH MOTION

André Da Luz Moreira / Linköping University

CFD with OpenSource Software 2022 (Chalmers University of Technology)

LINKÖPING UNIVERSITY

CONTENTS

```
presentationContents
{
    introduction();                         // Motivation for the work

    meshMotionInOpenFOAM
    (
        basics,                             // dynamicMeshDict settings; boundary updates
        laplacianMotionSolvers,             // Laplace's equation, velocity and displacement solvers
        diffusivity                         // Intro to diffusivity models
    );

    timeVaryingMotionInterpolation
    (
        function,                           // Purpose, how it works
        compilation,                        // Compilation instructions; file list
        usage,                              // How to use
        implementation                      // How it is coded
    );

    tutorials
    (
        airfoil,                            // A 2D deforming airfoil
        deformingCylinder                   // A 3D deforming cylinder, in parallel
    );
}
```

INTRODUCTION

OpenFOAM does not have a boundary condition for generic/arbitrary boundary motion. This is useful for experimental measurments, image registration, among others.

Example: simulations of biological flows using geometrical data from medical images.

For this purpose, a generic boundary condition for moving walls has been developed: `timeVaryingMotionInterpolation`

It can be used to extract motion information varying in time, from either unstructured data or regular spaced points, and interpolate this to CFD boundaries in OpenFOAM v2206.

MESH MOTION AND DEFORMATION IN OPENFOAM

OpenFOAM has multiple tools to deal with dynamic meshes. This can be used for geometries that change in time (morphing), to deal with overset meshes, sliding meshes, moving bodies and even adaptive mesh refinement.

Dynamic meshes are mainly controled by inputs in `dynamicMeshDict` and appropriate boundary conditions.

For mesh motion, an appropriate type of `dynamicFvMesh` is used, which in turn uses solvers and depends on boundary conditions defined for the CFD domain.

EXAMPLE `dynamicMeshDict`:

Example with `dynamicMotionSolverFvMesh` with a single Lagrangian motion solver and an inverse distance diffusivity model based on chosen boundary or boundaries.

```
/*--------------------------------*- 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");

```

LINKÖPING UNIVERSITY

From dynamicMotionSolverFvMesh.C:

```cpp
bool Foam::dynamicMotionSolverFvMesh::update()
{
    fvMesh::movePoints(motionPtr_->newPoints());

    volVectorField* Uptr = getObjectPtr<volVectorField>("U");

    if (Uptr)
    {
        Uptr->correctBoundaryConditions();
    }

    return true;
}
```

From motionSolver.C:

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

LAPLACE'S EQUATION FOR MESH DEFORMATION

Calculate a mesh deformation field in the mesh.

$$\nabla \cdot (\mathbf{\Gamma}_c \nabla \boldsymbol{V}_c) = 0$$

Defined at cell centres!

- $\mathbf{\Gamma}_c \Rightarrow$ mesh deformation diffusivity vector
- $\boldsymbol{V}_c \Rightarrow$ deformation (**velocities** or **displacements**)

Advantages: Simple to solve, bounded, non-uniform, smooth.

Drawback in OpenFOAM implementation:

Requires interpolation to cell points, which can cause problems in the mesh!

MOTION SOLVERS IN `fvMotionSolvers`:

- Velocity:
  - `velocityComponentLaplacian`
  - `velocityLaplacian`
- Displacement:
  - `displacementComponentLaplacian`
  - `displacementLaplacian`
  - `displacementSBRStress`
  - `solidBodyDisplacementLaplacian`
  - `surfaceAlignedSBRStress`

LAPLACIAN MOTION SOLVERS IN `fvMotionSolvers`:

- `velocityComponentLaplacian` and `displacementComponentLaplacian` solve equations for one component (**scalar elds**).
- `velocityLaplacian` and `displacementLaplacian` solve the velocities/displacements in 3D (**vector elds**) and are more complete than the component version.

We will go through the two complete Lagrangian solvers to understand their function and differences. We will start with the equations they solve, moving to their implementation in OpenFOAM, focusing on their constructors, as well as the `solve()` and `curPoints()` functions.

VELOCITY SOLVER: `velocityLaplacian`*

- Solves Laplace's equation for a velocity field $\boldsymbol{U}_c$:

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

- Inherits from `velocityMotionSolver`` and fvMotionSolve.
- Interpolates data from `cellMotionU_`` and pointMotionU_.
- Updates point coordinates using velocity and time step:

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

* Located at src/fvMotionSolver/fvMotionSolvers/velocity

LINKÖPING UNIVERSITY

CONSTRUCTOR:

- Received references to a `polyMesh` and a `IOdictionary`.

From `velocityLaplacianFvMotionSolver.C`

```
Foam::velocityLaplacianFvMotionSolver::velocityLaplacianFvMotionSolver
(
    const polyMesh& mesh,
    const IOdictionary& dict
)
:
{}
```

CONSTRUCTOR:

From velocityLaplacianFvMotionSolver.C

- Received references to a polyMesh and a IOdictionary.
- Initialises inheritances velocityMotionSolver and fvMotionSolver.
- Creates the cellMotionU_, interpolationPtr_ and diffusivityPtr_ objects.

```
)
:
    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"))
    )
{}
```

LINKÖPING UNIVERSITY

`solve()`:

- `movePoints` doesn't do anything.
- Update diffusivity.
- `updateCoeffs()` is used to perform updates, as will be seen for the developed BC later on.
- `fvOptions` used for `constrain()` and `correct()`.
- `nNonOrthCorr` retrieved.

From `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)
    );
}
```

LINKÖPING UNIVERSITY

solve():

- The equation for the cell centre velocities is defined and solved.
- Non-orthogonal correctors used here but not for displacement solver. But why??

From velocityLaplacianFvMotionSolver.C

```
    (
        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_);
    }
}
```

curPoints():

- Velocities interpolated from cell centres to mesh points.
- Current points are updated using previously shown equation.

From 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;
}
```

DISPL. SOLVER: `displacementLaplacian` *

- Solves Laplace's equation for a displacement field $\boldsymbol{\Delta X}_c$:

$$\nabla \cdot (\Gamma \, \nabla(\boldsymbol{\Delta X}_c)) = 0$$

- Inherits from `displacementMotionSolver`` and fvMotionSolver.
- Interpolates data from `cellDisplacement_`` to pointDisplacement_.
- Updates points from original coordinates:

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

* Located at src/fvMotionSolver/fvMotionSolvers/displacement/laplacian

CONSTRUCTOR:

- Received references to a `polyMesh` and a `IOdictionary`.

From `displacementLaplacianFvMotionSolver.C`

```
Foam::displacementLaplacianFvMotionSolver::displacementLaplacianFvMotionSolver
(
    const polyMesh& mesh,
    const IOdictionary& dict
)
:
{}
```

LINKÖPING UNIVERSITY

CONSTRUCTOR:

- Received references to a `polyMesh` and a `IOdictionary`.
- Initialises inheritances `displacementMotionSolve` and `fvMotionSolver`.
- Creates the `cellDisplacement_`, `interpolationPtr_` and `diffusivityPtr_` objects.

From `displacementLaplacianFvMotionSolver.C`

```
)
:
    displacementMotionSolver(mesh, dict, typeName),
    fvMotionSolver(mesh),
    cellDisplacement_
    (
        IOobject
        (
            "cellDisplacement",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        fvMesh_,
        dimensionedVector(pointDisplacement_.dimensions(), Zero),
        cellMotionBoundaryTypes<vector>(pointDisplacement_.boundaryField())
    ),
    interpolationPtr_
    (
        coeffDict().found("interpolation")
      ? motionInterpolation::New(fvMesh_, coeffDict().lookup("interpolation"))
      : motionInterpolation::New(fvMesh_)
    ),
    diffusivityPtr_
    (
        motionDiffusivity::New(fvMesh_, coeffDict().lookup("diffusivity"))
    )
{}
```

CONSTRUCTOR:

- Additional tasks:
- Initialises `pointLocation_` and `frozenPointsZone_`.
- `pointLocation_` used when a applying BCs to points (beying scope of this report).
- `frozenPointsZone_` will not be discussed here.

From `displacementLaplacianFvMotionSolver.C`

```
        fvMesh_,
        dimensionedVector(pointDisplacement_.dimensions(), Zero),
        cellMotionBoundaryTypes<vector>(pointDisplacement_.boundaryField())
    ),
    pointLocation_(nullptr),
    interpolationPtr_
    (
        coeffDict().found("interpolation")
      ? motionInterpolation::New(fvMesh_, coeffDict().lookup("interpolation"))
      : motionInterpolation::New(fvMesh_)
    ),
    diffusivityPtr_
    (
        motionDiffusivity::New(fvMesh_, coeffDict().lookup("diffusivity"))
    ),
    frozenPointsZone_
    (
        coeffDict().found("frozenPointsZone")
      ? fvMesh_.pointZones().findZoneID
        (
            coeffDict().get<word>("frozenPointsZone")
        )
      : -1
    )
{
    IOobject io
    (
        "pointLocation",
        fvMesh_.time().timeName(),
```

LINKÖPING UNIVERSITY

CONSTRUCTOR:

- Additional tasks:
- Initialises `pointLocation_` and `frozenPointsZone_`.
- `pointLocation_` used when a applying BCs to points (beying scope of this report).
- `frozenPointsZone_` will not be discussed here.

From `displacementLaplacianFvMotionSolver.C`

```
          ? fvMesh_.pointZones().findZoneID
            (
                coeffDict().get<word>("frozenPointsZone")
            )
          : -1
    )
{
    IOobject io
    (
        "pointLocation",
        fvMesh_.time().timeName(),
        fvMesh_,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    );

    if (debug)
    {
        Info<< "displacementLaplacianFvMotionSolver:" << nl
            << "    diffusivity       : " << diffusivityPtr_().type() << nl
            << "    frozenPoints zone : " << frozenPointsZone_ << endl;
    }


    if (io.typeHeaderOk<pointVectorField>(true))
    {
        pointLocation_.reset
        (
            new pointVectorField
```

CONSTRUCTOR:

- Additional tasks:
- Initialises `pointLocation_` and `frozenPointsZone_`.
- `pointLocation_` used when a applying BCs to points (beying scope of this report).
- `frozenPointsZone_` will not be discussed here.

From `displacementLaplacianFvMotionSolver.C`

```
        Info<< "displacementLaplacianFvMotionSolver:" << nl
            << "    diffusivity       : " << diffusivityPtr_().type() << nl
            << "    frozenPoints zone : " << frozenPointsZone_ << endl;
    }


    if (io.typeHeaderOk<pointVectorField>(true))
    {
        pointLocation_.reset
        (
            new pointVectorField
            (
                io,
                pointMesh::New(fvMesh_)
            )
        );

        if (debug)
        {
            Info<< "displacementLaplacianFvMotionSolver :"
                << " Read pointVectorField "
                << io.name()
                << " to be used for boundary conditions on points."
                << nl
                << "Boundary conditions:"
                << pointLocation_().boundaryField().types() << endl;
        }
    }
}
```

LINKÖPING UNIVERSITY

solve():

- `movePoints` doesn't do anything.
- Update diffusivity.
- `updateCoeffs()` is used to perform updates, as will be seen for the developed BC later on.
- `fvOptions` used for `constrain()` and `correct()`.

From displacementLaplacianFvMotionSolver.C

```
void Foam::displacementLaplacianFvMotionSolver::solve()
{
    // 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 inbetwee
    // 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_);
```

solve():

- The equation for the cell centre velocities is defined and solved.
- This solver does not use non-orthogonal correctors. But why??

From displacementLaplacianFvMotionSolver.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 inbetwee
    // 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_);
}
```

curPoints():

- Displacements interpolated from cell centres to mesh points.
- Current points are updated using previously shown equation.

From displacementLaplacianFvMotionSolver.C

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

    if (pointLocation_)
    {
        if (debug)
        {
            Info<< "displacementLaplacianFvMotionSolver : applying
                << " boundary conditions on " << pointLocation_().n
                << " to new point location."
                << endl;
        }

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

        pointLocation_().correctBoundaryConditions();

        // Implement frozen points
        if (frozenPointsZone_ != -1)
        {
            const pointZone& pz = fvMesh_.pointZones()[frozenPoints
```

LINKÖPING UNIVERSITY

curPoints():

- Displacements interpolated from cell centres to mesh points.
- Current points are updated using previously shown equation.

From displacementLaplacianFvMotionSolver.C

```

        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)
        {
            const pointZone& pz = fvMesh_.pointZones()[frozenPoints

            forAll(pz, i)
            {
                curPoints[pz[i]] = points0()[pz[i]];
            }
        }

        twoDCorrectPoints(curPoints);

        return tcurPoints;
    }
}
```

DIFFUSIVITY MODELS:

Available in folder `src/fvMotionSolver/motionDiffusivity/`.

- `uniformDiffusivity`
- `inverseDistance`
- `inverseFaceDistance`
- `inversePointDistance`
- `inverseVolume`
- `directional`
- `motionDirectional`
- `file`

DIFFUSIVITY MODELS:

- Very poor descriptions in source files.
- Available models may be modified with a **square** or an **exponential** function.
- Selecting an appropriate model:

  VERY IMPORTANT! MODEL SPECIFIC!
- Inverse distance methods tend to outperform uniform models [1] [2]

[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.

`timeVaryingMotionInterpolation:`

Developed for seamless integration with `fvMotionSolvers`.

Compiled as part of a new library names `myFvMotionSolvers`, which can be used in `dynamicMeshDict`.

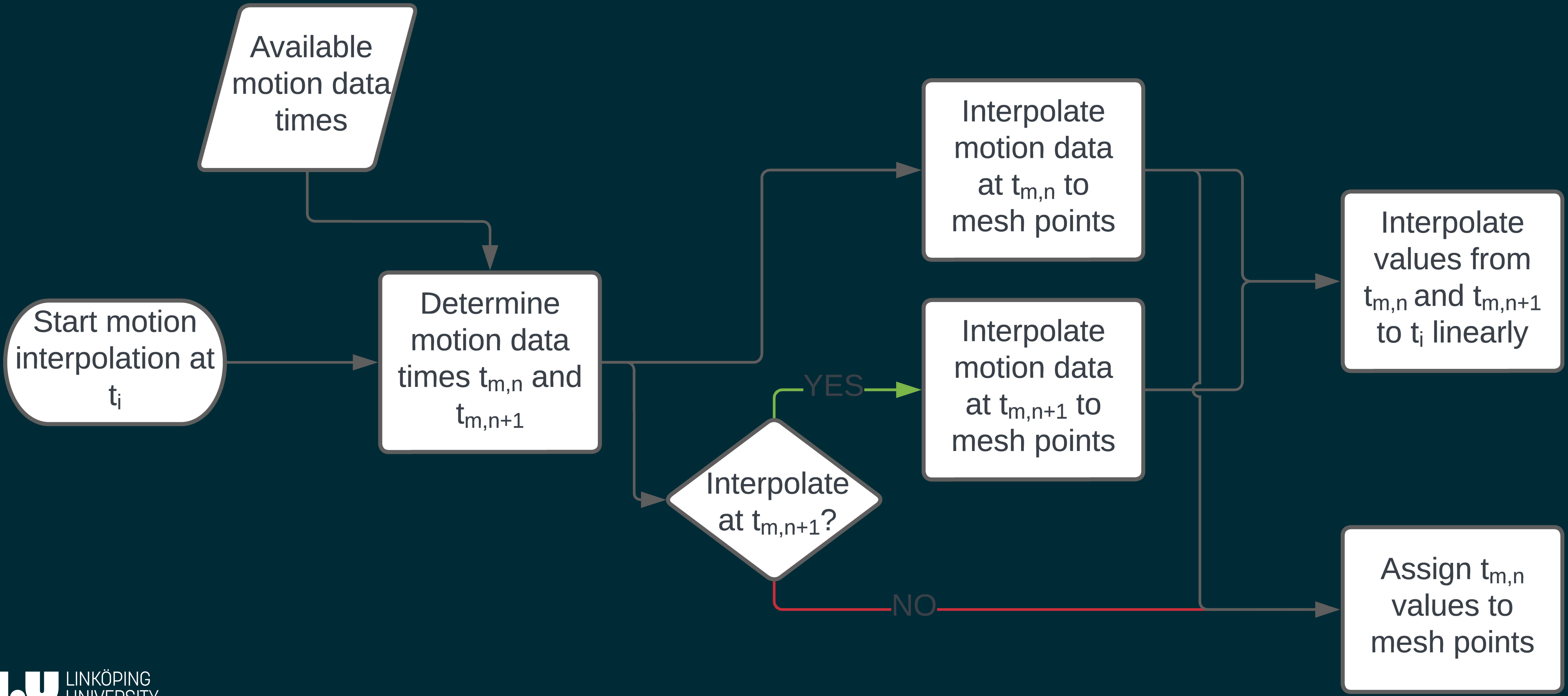
Allows arbitrary motion information to be applied to boundaries.

Motion/deformation may be obtained experimentally, from image registration, etc.

We'll go through it in steps!

FLOWCHART FOR

`timeVaryingMotionInterpolation`:



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

COMPILATION OF `timeVaryingMotionInterpolation`:

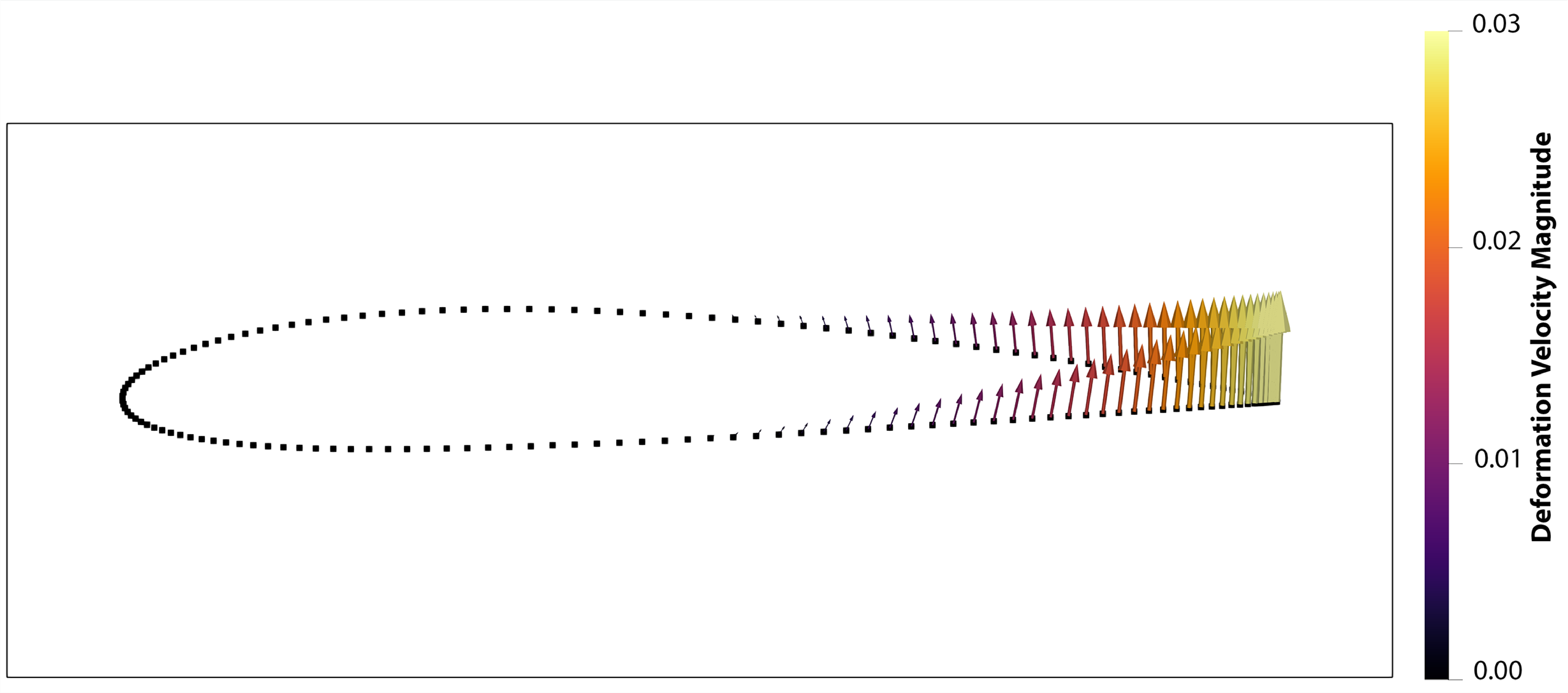
Provided files:

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

Compilation either using `wmake` or the provided script `Allwmake`.

Add **`motionSolverLibs ("myFvMotionSolvers.so");`** to `dynamicMeshDict`.

EXAMPLE OF UNSTRUCTURED MOTION DATA:



UNSTRUCTURED MOTION DATA INPUTS:

Files for point coordinates and motion (velocity or displacement).

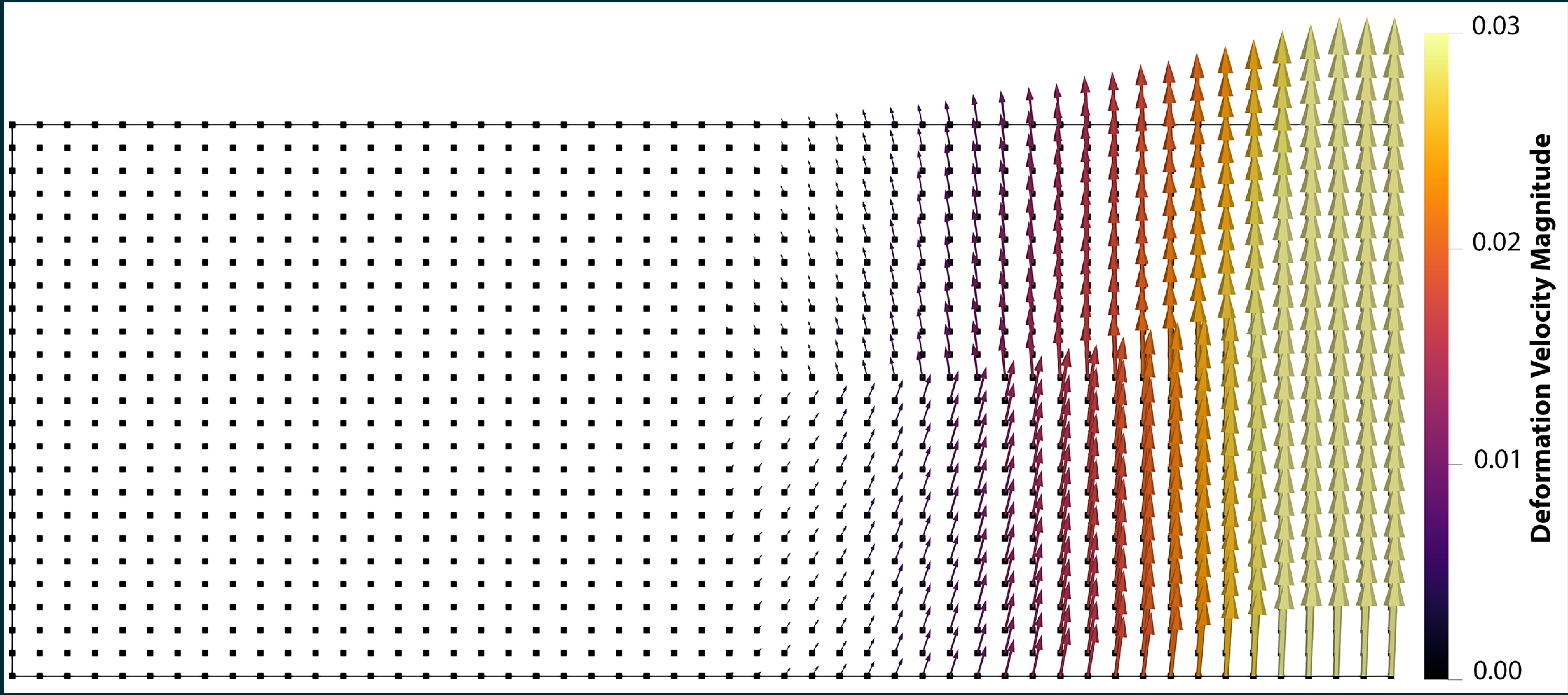
```
// Example points file
N
(
(pX_0        pY_0        pZ_0     )
(pX_1        pY_1        pZ_1     )
(pX_2        pY_2        pZ_2     )

...

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

```
// 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   )
)
```

```
`-- inputFolderName
    |-- 0.0000
    |   |-- pointDisplacement
    |   `-- points
    |-- 0.0500
    |   |-- pointDisplacement
    ...
    `-- 9.9500
        |-- pointDisplacement
        `-- points
```

Example folder structure.

EXAMPLE OF STRUCTURED MOTION DATA:



STRUCTURED MOTION DATA INPUTS:

Motion data files (velocity or displacement) and matrix information file.

```
// Example domainMatrixInfo file
// 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 )
(d_X    d_Y     d_Z   )
(N_x    N_y     N_z   )
)
```

```
// 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   )
)
```

```
`-- inputFolderName
    |-- 0.0000
    |   `-- pointMotionU
    |-- 0.0500
    |   `-- pointMotionU
    ...
    |-- 9.9500
    |   `-- pointMotionU
    `-- domainMatrixInfo
```

Example folder structure.

LINKÖPING UNIVERSITY

STRUCTURED MATRIX FORMAT:

For a point with indices $i_{x,p}$, $i_{y,p}$ and $i_{z,p}$, its coordinates are

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

```
// Example domainMatrixInfo file
// 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 )
(d_X    d_Y     d_Z   )
(N_x    N_y     N_z   )
)
```

OpenFOAM reads data as a list, so an index m_p is needed:

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

PRIVATE VARIABLES:

From `timeVaryingMotionInterpolationPointPatchField.H`

```
template<class Type>
class timeVaryingMotionInterpolationPointPatchField
:
    public fixedValuePointPatchField<Type>
{
    // Private data

        //- Name of the field data table, defaults to the name of the field
        word fieldTableName_;

        //- List of boundaryData time directories
        instantList sampleTimes_;

        //- Current starting index in sampleTimes
        label startSampleTime_;

        //- Interpolated values from startSampleTime
        Field<Type> startSampledValues_;

        //- Current end index in sampleTimes
        label endSampleTime_;

        //- Interpolated values from endSampleTime
        Field<Type> endSampledValues_;

```

PRIVATE VARIABLES:

From `timeVaryingMotionInterpolationPointPatchField.H`

```

        //- Input data type
        word inputType_;

        //- Interpolation type
        word interpolationType_;

        //- Custom boundary subfolder
        word inputFolderName_;

        //- Use custom boundary folder?
        Switch useCustomFolder_;

        //- Outside bounds action
        Switch intOutsideBounds_;

        //- Inverse distance search radius
        scalar inverseDistRadius_;

        //- Sample data coordinates at start
        pointField startSamplePoints_;

        //- Sample data values at start
        Field<Type> startSampleData_;

```

PRIVATE VARIABLES:

From `timeVaryingMotionInterpolationPointPatchField.H`

```
        Switch intOutsideBounds_;

        //- Inverse distance search radius
        scalar inverseDistRadius_;

        //- Sample data coordinates at start
        pointField startSamplePoints_;

        //- Sample data values at start
        Field<Type> startSampleData_;

        //- Sample data coordinates at end
        pointField endSamplePoints_;

        //- Sample data values at end
        Field<Type> endSampleData_;

        //- Reference point for sample data coordinates
        vector domainRefPt_;

        //- Dimensions (number of voxels in x, y, z) from sample data
        vector domainMatDm_;

        //- Voxel sizing for sample data voxels (x, y, z)
        vector domainVoxSz_;
```

LINKÖPING UNIVERSITY

updateCoeffs()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
template<class Type> 57
void Foam::timeVaryingMotionInterpolationPointPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;                 // Check if already updated
    }

    checkTable();               // Call checkTable() function

    // 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_);  // Assign startSampledValues_
    }
    else
```

updateCoeffs()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
    {
        return;                 // Check if already updated
    }

    checkTable();               // Call checkTable() function

    // 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_);  // Assign startSampledValues_
    }
    else
    {
        scalar start = sampleTimes_[startSampleTime_].value();            // t_{m,n}
        scalar end = sampleTimes_[endSampleTime_].value();                // t_{m,n+1}

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

updateCoeffs()

From timeVaryingMotionInterpolationPointPatchField.C

```
        }

        this->operator==(startSampledValues_);  // Assign startSampledValues_
    }
    else
    {
        scalar start = sampleTimes_[startSampleTime_].value();          // t_{m,n}
        scalar end = sampleTimes_[endSampleTime_].value();              // t_{m,n+1}

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

        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_); // Interpolate in time
    }


    if (debug)
    {
```

updateCoeffs()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
        scalar end = sampleTimes_[endSampleTime_].value();                  // t_{m,n+1}

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

        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_); // Interpolate in time
    }


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

    fixedValuePointPatchField<Type>::updateCoeffs();    // Run updateCoeffs() from parent class
}
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
template<class Type>
void Foam::timeVaryingMotionInterpolationPointPatchField<Type>::checkTable()
{
    const Time& time = this->db().time();                               // Reference to time

    const polyMesh& pMesh = this->patch().boundaryMesh().mesh()();   // Reference to boundary mesh

    // Read the initial point position
    pointField meshPts;                                                 // Point coordinates

    if (pMesh.pointsInstance() == pMesh.facesInstance())
    {
        meshPts = pointField(pMesh.points(), this->patch().meshPoints());
    }
    else
    {
        // Load points from facesInstance
        if (debug)
        {
            Info<< "Reloading points0 from " << pMesh.facesInstance()
                << endl;
        }

        pointIOField points0
        (
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
    }
    else
    {
        // Load points from facesInstance
        if (debug)
        {
            Info<< "Reloading points0 from " << pMesh.facesInstance()
                << endl;
        }

        pointIOField points0
        (
            IOobject
            (
                "points",
                pMesh.facesInstance(),
                polyMesh::meshSubDir,
                pMesh,
                IOobject::MUST_READ,
                IOobject::NO_WRITE,
                false
            )
        );
        meshPts = pointField(points0, this->patch().meshPoints());
    }
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
        );
        meshPts = pointField(points0, this->patch().meshPoints());
    }

    // Initialise
    if (startSampleTime_ == -1 && endSampleTime_ == -1)
    {
        // Structured domain properties file path
        // (Defined here because it's path is used anyways)
        const fileName domainInfoFile
        (
             time.path()
            /time.caseConstant()
            /"boundaryData"
            /inputFolderName_
            /"domainMatrixInfo"
        );

        // Read the times for which data is available

        const fileName samplePointsDir = domainInfoFile.path();
        sampleTimes_ = Time::findTimes(samplePointsDir);            // Find available times

        if (debug)
        {
            Info<< "timeVaryingMotionInterpolationPointPatchField : In directory "
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
        // Read structed data domain properties
        if (inputType_ == "structured")
        {
            IOobject ioDomainInfoFile                          // Read the domain information
            (
                domainInfoFile,    // absolute path
                time,
                IOobject::MUST_READ,
                IOobject::NO_WRITE,
                false,                 // no need to register
                true                   // is global object (currently not used)
            );
            const rawIOField<point> domainInformationData(ioDomainInfoFile, false);
            if (domainInformationData.size() != 3 && domainInformationData.size() != 6)
            {
                FatalErrorInFunction
                    << "Length of file 'domainMatrixInfo' (" << domainInformationData.size()
                    << ") differs from allowed values (3 and 6) in file "
                    << domainInfoFile << exit(FatalError);
            }
            domainRefPt_ = domainInformationData[0];    // Reference point
            domainVoxSz_ = domainInformationData[1];    // Matrix spacing
            domainMatDm_ = domainInformationData[2];    // Matrix dimensions
            // Create a variable with the structured point coordinates
            // THIS IS NOT EFFICIENT, BUT SHOULD NOT BE FORBIDDEN!!!
```

LINKÖPING UNIVERSITY

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
if (interpolationType_ != "trilinear")
{
    int m;
    int matSize = domainMatDm_.x()*domainMatDm_.y()*domainMatDm_.z();
    tmp<pointField> tstrPts(new pointField(matSize));
    pointField& strPts = tstrPts.ref();
    for(int ii = 0; ii<domainMatDm_.x(); ii++)
    {
        for(int jj = 0; jj<domainMatDm_.y(); jj++)
        {
            for(int kk = 0; kk<domainMatDm_.z(); kk++)
            {
                m = (kk)*(domainMatDm_.x()*domainMatDm_.y()) +
                    (jj)*(domainMatDm_.x()) +
                    (ii);
                strPts[m] = point
                (
                    domainRefPt_.x()+ii*domainVoxSz_.x(),
                    domainRefPt_.y()+jj*domainVoxSz_.y(),
                    domainRefPt_.z()+kk*domainVoxSz_.z()
                );
            }
        }
    }
    startSamplePoints_ = tstrPts;   // Calculate a matrix with all structured points
```

LINKÖPING UNIVERSITY

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
                }
            }
            startSamplePoints_ = tstrPts;   // Calculate a matrix with all structured points
        }
    }
}

// Find current time in sampleTimes
label lo = -1;
label hi = -1;

bool foundTime = pointToPointPlanarInterpolation::findTime // Find indices of time folders
(
    sampleTimes_,
    startSampleTime_,
    time.value(),
    lo,                         // Index for t_{m,n}
    hi                          // Index for t_{m,n+1}
);

if (!foundTime)
{
    FatalErrorInFunction
        << "Cannot find starting sampling values for current time "
        << time.value() << nl
        << "Have sampling values for times "
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
    // Update START sampled data fields.
    if (lo != startSampleTime_)
    {
        startSampleTime_ = lo;

        if (startSampleTime_ == endSampleTime_)
        {
            // No need to reread since are end values
            if (debug)
            {
                Pout<< "checkTable : Setting startValues to (already read) "
                    <<   "boundaryData"
                      /inputFolderName_
                      /sampleTimes_[startSampleTime_].name()
                    << endl;
            }
            startSampleData_ = endSampleData_;      // Previous t_{m,n+1} for t_{m,n}
            if (inputType_ == "unstructured")
            {
                startSamplePoints_ = endSamplePoints_;
            }
        }
        else
        {
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
                        << endl;
                }

                // Reread field values at points
                const fileName valsFile
                (
                    time.path()
                   /time.caseConstant()
                   /"boundaryData"
                   /inputFolderName_
                   /sampleTimes_[startSampleTime_].name()
                   /fieldTableName_
                );
                IOobject ioField
                (
                    valsFile,           // absolute path
                    time,
                    IOobject::MUST_READ,
                    IOobject::NO_WRITE,
                    false,              // no need to register
                    true                // is global object (currently not used)
                );
                startSampleData_ = rawIOField<Type>(ioField, false);    // Read motion data at t_{m,n}

                // Reread field points coordinates for unstructured
                if (inputType_ == "unstructured")
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
            // Reread field points coordinates for unstructured
            if (inputType_ == "unstructured")
            {
                // Reread mask data
                const fileName pointsFile
                (
                    time.path()
                   /time.caseConstant()
                   /"boundaryData"
                   /inputFolderName_
                   /sampleTimes_[startSampleTime_].name()
                   /"points"
                );
                IOobject ioPoints
                (
                    pointsFile,          // absolute path
                    time,
                    IOobject::MUST_READ,
                    IOobject::NO_WRITE,
                    false,               // no need to register
                    true                 // is global object (currently not used)
                );
                startSamplePoints_ = rawIOField<point>(ioPoints, false);    // Read motion points at t_{m,n}

                if (startSampleData_.size() != startSamplePoints_.size())
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
            }
        }

        // Update END sampled data fields.
        if (hi != endSampleTime_)
        {
            endSampleTime_ = hi;

            if (endSampleTime_ == -1)
            {
                // endTime no longer valid. Might as well clear endValues.
                if (debug)
                {
                    Pout<< "checkTable : Clearing endValues" << endl;
                }
                endSampledValues_.clear();          // Clear values if current time> t_{m,end}
                endSampleData_.clear();
                if (inputType_ == "unstructured")
                {
                    endSamplePoints_.clear();
                }
            }
            else
            {
                if (debug)
                {
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
        }
    }

    // Do the interpolation of the data to mesh coordinates
    scalar deltaTime = time.value() - sampleTimes_[startSampleTime_].value();
    bool interpolateEnd = (endSampleTime_ != -1  && deltaTime > SMALL);

    if (interpolationType_ == "trilinear")
    {
        applyTrilinearInterpolation
        (
            meshPts,
            interpolateEnd
        );
    }
    else if (interpolationType_ == "nearest")
    {
        applyNearestValues
        (
            meshPts,
            interpolateEnd
        );
    }
    else if (interpolationType_ == "inverseDist")
    {
        applyInverseDistanceInterpolation
```

checkTable()

From `timeVaryingMotionInterpolationPointPatchField.C`

```
        );
    }
    else if (interpolationType_ == "nearest")
    {
        applyNearestValues
        (
            meshPts,
            interpolateEnd
        );
    }
    else if (interpolationType_ == "inverseDist")
    {
        applyInverseDistanceInterpolation
        (
            meshPts,
            interpolateEnd
        );
    }
    else
    {
        FatalErrorInFunction
            << "Illegal interpolation option."
            << abort(FatalError);
    }
}
```

TYPES OF INTERPOLATION:

NEAREST VALUE:

Direct assignment of the value at the closest data point to the boundary points.

May lead to poor quality results in cases with a coarse point cloud.

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.

Hypothetical ideal case: one data point for each mesh boundary point.

Recommendation: use a point cloud with spacing similar to the mesh.

TYPES OF INTERPOLATION:

NEAREST VALUE:

```
template<class Type>
void Foam::timeVaryingMotionInterpolationPointPatchField<Type>::applyNearestValues
(
    const pointField& meshPts,
    const bool& interpolateEnd
)
{
    // Create a generic interpolator pointer
    autoPtr<pointToPointPlanarInterpolation> interpPtr;

    // Always interpolate start values
    interpPtr.reset
    (
        new pointToPointPlanarInterpolation
        (
        startSamplePoints_, // sourcePoints
        meshPts,            // destPoints
        0,                  // perturb (not used)
        true                // nearestOnly
        )
    );
    startSampledValues_ = interpPtr().interpolate(startSampleData_);

```

TYPES OF INTERPOLATION:

NEAREST VALUE:

```
    );
    startSampledValues_ = interpPtr().interpolate(startSampleData_);

    // Interpolate end
    if (interpolateEnd)
    {
        if (inputType_ == "unstructured")
        {
            interpPtr.reset
            (
                new pointToPointPlanarInterpolation
                (
                endSamplePoints_,   // sourcePoints
                meshPts,            // destPoints
                0,                  // perturb (not used)
                true                // nearestOnly
                )
            );
        }

        endSampledValues_ = interpPtr().interpolate(endSampleData_);
    }
}
```

TYPES OF INTERPOLATION:

INVERSE DISTANCE:

Follows method described by Shepard with a distance criterion: [3]

$$V_p = \begin{cases} \dfrac{\sum_{i=1}^{n} w_i V_i}{\sum_{i=1}^{n} w_i} & \text{, for all } i \text{ where } \mathrm{dist}(\boldsymbol{X}_p, \boldsymbol{X}_i) \leq R_{\min} \text{ ,} \\ V_i & \text{, for any } i \text{ where } \mathrm{dist}(\boldsymbol{X}_p, \boldsymbol{X}_i) = 0 \text{ .} \end{cases}$$

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

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

LINKÖPING UNIVERSITY

TYPES OF INTERPOLATION:

INVERSE DISTANCE:

```
template<class Type> 1210
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)                                    // Iterate through boundary points
    {
        // Assign new values to SortableList and sort it
        distSorted = mag(meshPts[pp]-samplePoints);
        distSorted.sort();

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

```

LINKÖPING UNIVERSITY

TYPES OF INTERPOLATION:

INVERSE DISTANCE:

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

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

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

        forAll(distSorted,jj)                                  // Iterate through distances (ascending)
        {
            scalar pointD = distSorted[jj];
            if (pointD>inverseDistRadius_)                     // Stop loop if distance > inverseDistRadius_
            {
                break;
            }

            label pointI = distSorted.indices()[jj];
            if (pointD < SMALL)                                    // dist=0 condition
            {
```

TYPES OF INTERPOLATION:

INVERSE DISTANCE:

```
            if (pointD>inverseDistRadius_)               // Stop loop if distance > inverseDistRadius_
            {
                break;
            }

            label pointI = distSorted.indices()[jj];
            if (pointD < SMALL)                              // dist=0 condition
            {
                interpNum = sampleData[pointI];
                interpDen = 1.0;
                break;
            }

            scalar pointInvD = 1/(pointD);                   // Inv. dist weight (w_i)

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

        if ( interpDen == 0)
        {
            fld[pp] = Type(Zero);                            // No division by 0
        }
```

TYPES OF INTERPOLATION:

INVERSE DISTANCE:

```
                    interpNum = sampleData[pointI];
                    interpDen = 1.0;
                    break;
                }

                scalar pointInvD = 1/(pointD);                  // Inv. dist weight (w_i)

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

            if ( interpDen == 0)
            {
                fld[pp] = Type(Zero);                           // No division by 0
            }
            else
            {
                fld[pp] = interpNum/interpDen;                  // Final value at point
            }
        }

    return tfld;                                                // Return field
}
```

LINKÖPING UNIVERSITY

TYPES OF INTERPOLATION:

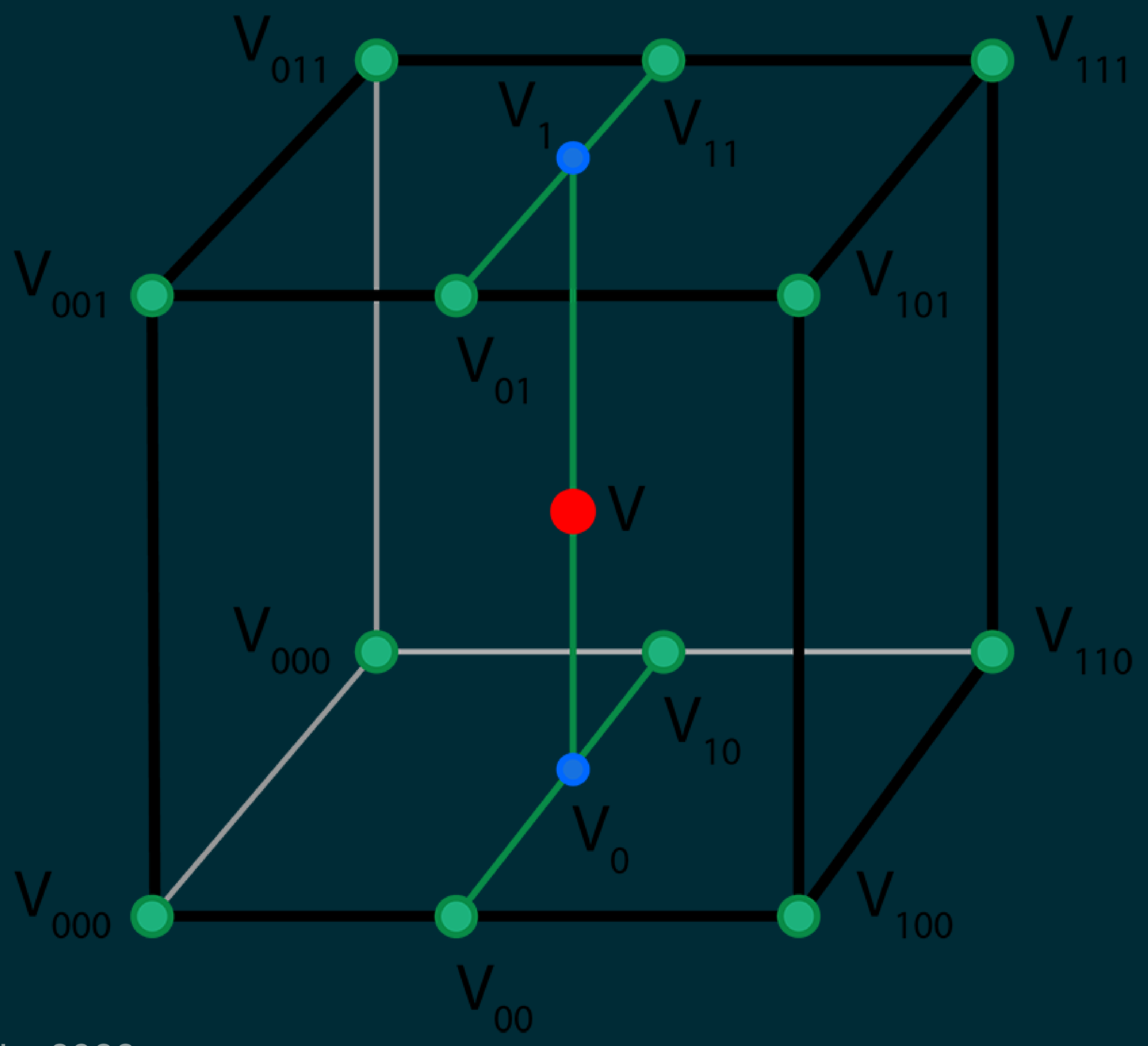
TRILINEAR INTERPOLATION:

A series of 7 linear interpolations in a regular lattice, using the 8 nearest data vertices to a mesh point.

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

$$V_0 = V_{00}\,(1 - l_y) + V_{10} l_y \ ,$$
$$V_1 = V_{01}\,(1 - l_y) + V_{11} l_y \ ,$$

$$V = V_0\,(1 - l_z) + V_1 l_z .$$



TYPES OF INTERPOLATION:

TRILINEAR INTERPOLATION:

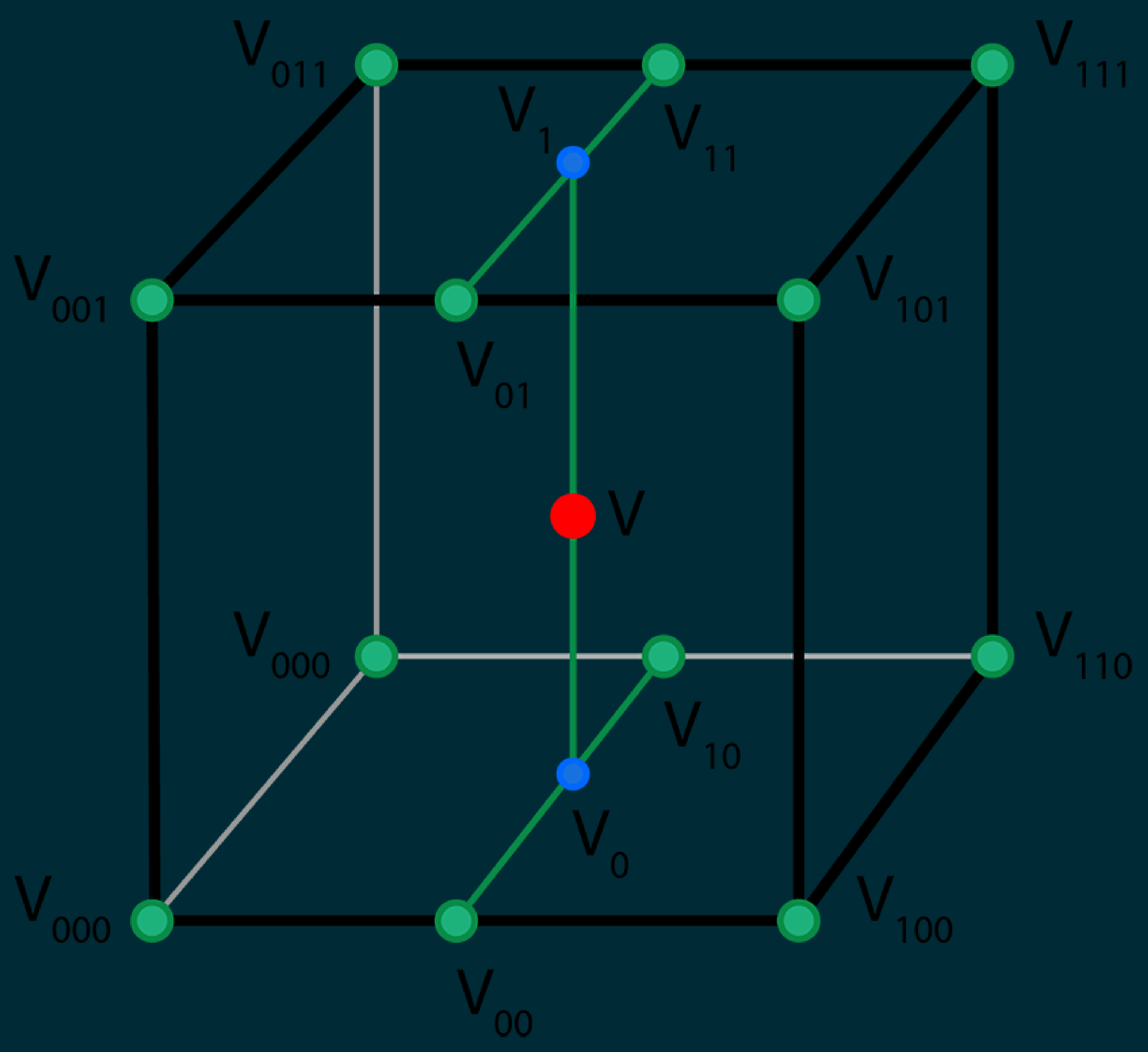
Interpolation factors l_x, l_y and l_z represent normalised terms for the linear interpolations ranging from 0 to 1 and calculated as

$$l_x = \frac{x_p - x_0}{x_1 - x_0} \, ,$$

$$l_y = \frac{y_p - y_0}{y_1 - y_0} \, ,$$

$$l_z = \frac{z_p - z_0}{z_1 - z_0} \, ,$$

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.



TYPES OF INTERPOLATION:

LINEAR INTERPOLATION:

Given a linear interpolation factor l_f, interpolate between two values.

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

TYPES OF INTERPOLATION:

BILINEAR INTERPOLATION:

Given two linear interpolation factors $l_{f,1}$ and $l_{f,2}$, interpolate between 4 values, in two dimensions.

```
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);
}
```

TYPES OF INTERPOLATION:

TRILINEAR INTERPOLATION:

Given three linear interpolation factors $l_{f,1}$, $l_{f,2}$ and $l_{f,3}$, perform two bilinear interpolations in axes 1 and 2 and a linear interpolation in axis 3.

```
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
```

TYPES OF INTERPOLATION:

TRILINEAR INTERPOLATION:

Given three linear interpolation factors $l_{f,1}$, $l_{f,2}$ and $l_{f,3}$, perform two bilinear interpolations in axes 1 and 2 and a linear interpolation in axis 3.

```
    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
```

TYPES OF INTERPOLATION:

TRILINEAR INTERPOLATION:

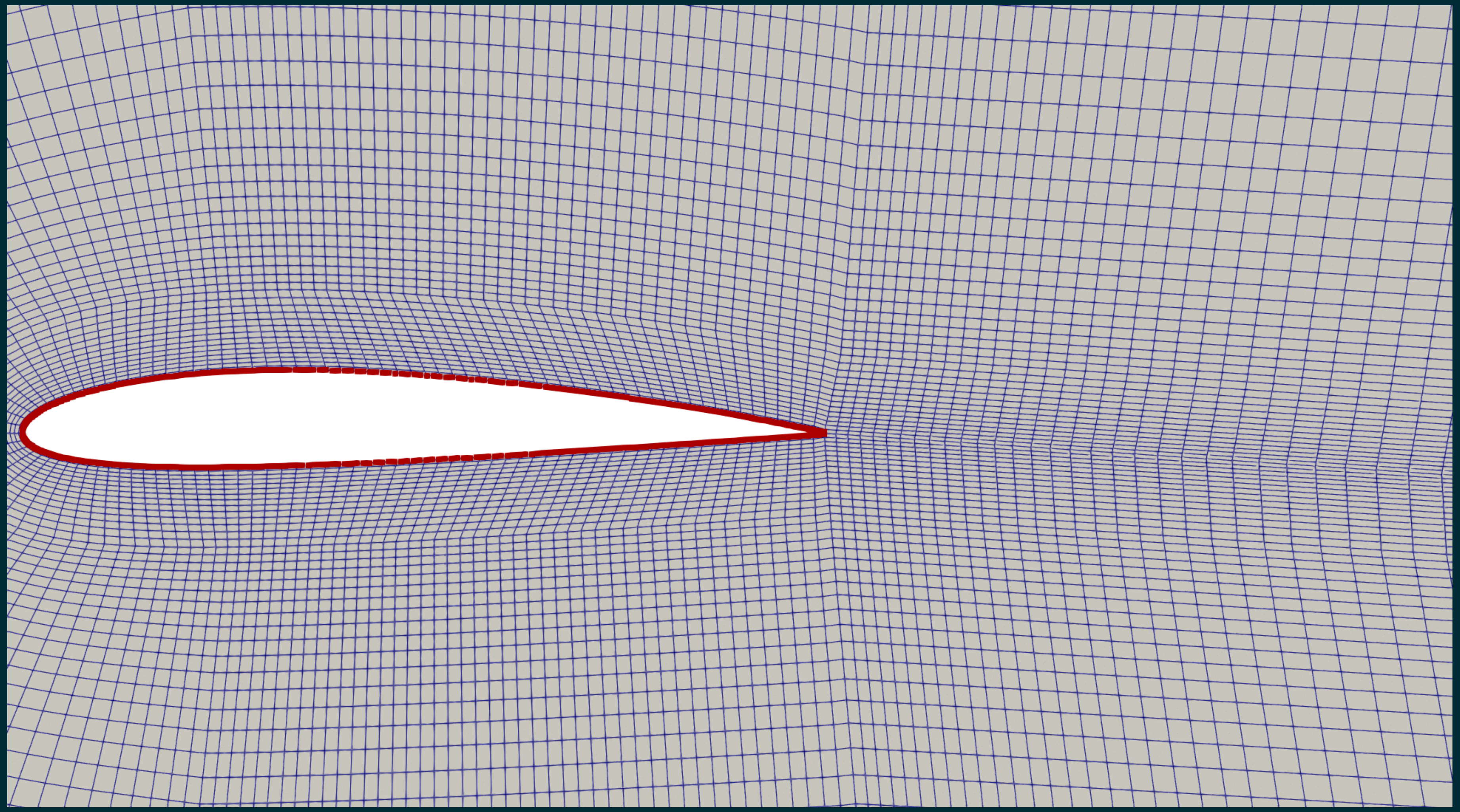
Given three linear interpolation factors $l_{f,1}$, $l_{f,2}$ and $l_{f,3}$, perform two bilinear interpolations in axes 1 and 2 and a linear interpolation in axis 3.

```
            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);
    }
```

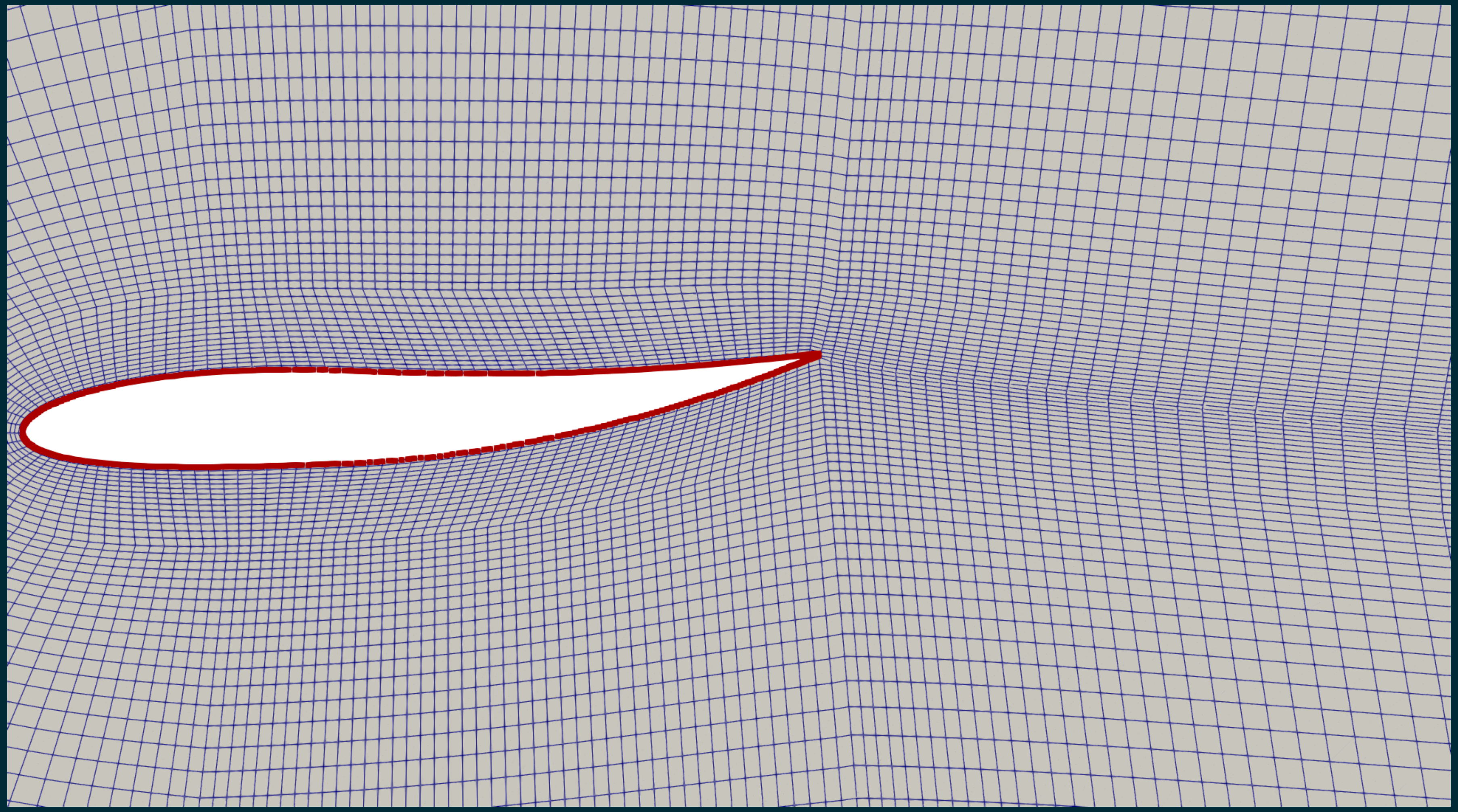
LINKÖPING UNIVERSITY

TUTORIAL 01: DEFORMING 2D AIRFOIL

User-defined NACA 4 digit airfoil. See files in `tutorials/airfoil` and file `tutorials/Allrun_airfoil`.



(a) Initial mesh.



(b) Deformed mesh.

TUTORIAL 01: DEFORMING 2D AIRFOIL

We will follow `tutorials/Allrun_airfoil`.

Then go through the details of one of the simulation folders.

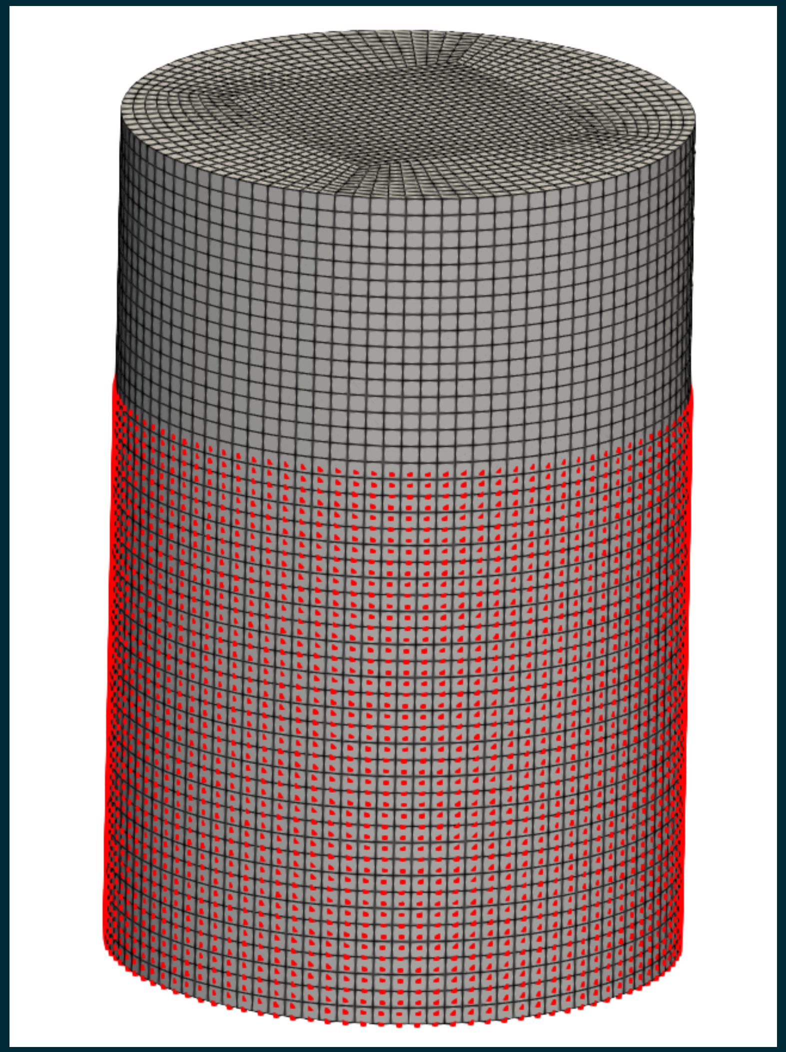
PRACTICAL SESSION!

```
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
```

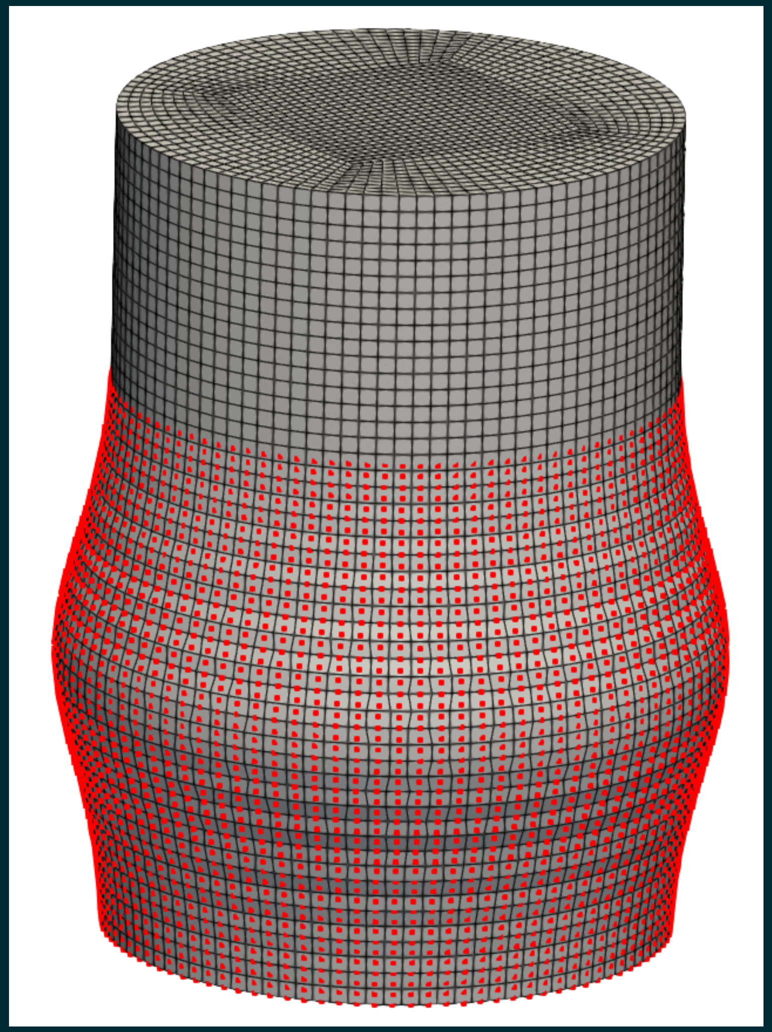
TUTORIAL 02: DEFORMING 3D CYLINDER

A cylinder with radial deformation as a function of height (z) and time.

See files in folder `tutorials/deformingCylinder` and file `tutorials/Allrun_deformingCylinder`.



(a) Initial mesh.



(b) Deformed mesh.

TUTORIAL 02: DEFORMING 3D CYLINDER

We will follow `tutorials/Allrun_deformingCylinder`.

Then go through the details of one of the simulation folders.

PRACTICAL SESSION!

```
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
```

FINAL REMARKS:

`timeVaryingMotionInterpolation` can be improved in many ways:

- `pointToPointPlanarInterpolation` can also be used for 2D Delaunay interpolation (its original purpose).
- Improve non-trilinear interpolation methods for `structured` data, so that a matrix with point coordinates is not needed.
- Allow for structured grids with motion data that are not aligned with global coordinate system.
- Improve temporal interpolation. Linear is not great and can produce motions that not accurately follow the desired motion.

I will have all BC files and tutorials in a git repository (currently exists, but it is still empty). I hope to add updates/changes in the future.

THANK YOU!