CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

The implementation of interFoam solver as a flow model of the fsiFoam solver for strong fluid-structure interaction

---

Developed for foam-extend-3.1
Requires: Paraview

Author:
Thomas Vyzikas

Peer reviewed by:
Håkan Nilsson
Jelena Andric

January 22, 2015

Contents

Chapter 1

Introduction

1.1 Basics of the Fluid-Structure Interaction problem

The Fluid-Structure Interaction (FSI) problem is a very commonly faced engineering problem spanning from aerospace engineering to biomechanics. The approaches that OpenFOAM has incorporated for solving the interaction between the fluid and the structure were presented during the third occasion of the "CFD with OpenSource software" course by Hua-Dong Yao [3] in 2014.

The idea behind the FSI problem is that the flow deforms an elastic structure, which by deforming influences the flow around it. As a result, FSI is a problem of both fluid mechanics and solid mechanics.

The FSI simulation approaches are discussed in Željko Tuković's presentation in a previous OpenFOAM course [7]. The methods that can be considered are split in two approaches:

1. The simultaneous or **monolithic** approach, where the interaction of the fluid and the structure at the mutual interface is treated synchronously, through a unified mathematical model and a unified discretisation method. The system of equations is solved iteratively, which guarantees unconditional stability. In general, monolithic schemes are accurate and stable, but computationally expensive. The monolithic approach is ideal for strong coupling between the fluid and the solid.

2. The **partitioned** approach, where there is a separate mathematical model for the fluid and the solid. The fluid and the solid continua are solved independently and they "communicate" through their common interface. The coupling is achieved by enforcing the kinematic and dynamic conditions via the boundary conditions at the interface. The solving process is sequential and progressed from solid to fluid in each time step as it can be seen in Figure 1.1. In principle, partitioned schemes are less accurate and stable than monolithic, but they are computationally efficient. The partitioned method is used for weak coupling between the fluid and the solid.

A comparison between the partitioned and monolithic procedures for the numerical simulation of FSI problems is discussed in Michler's study [5]. The two methods are compared in terms of stability, accuracy and computational cost, using a simple numerical piston as an experimental benchmark.

The FSI package in `foam-extend` has employed the partitioned approach for the coupling between the fluid and the solid. However, instead of the "traditional" explicit coupling, which is suitable only for weak interaction, it provides an implicit coupling method, which can account for strong interactions [3]. The core difference between implicit (strong) and explicit (weak) coupling is that the first has an extra loop in the solution algorithm, which guarantees that after each time step the mesh deformation is consistent with the kinematic and dynamic conditions at the interface. Figure

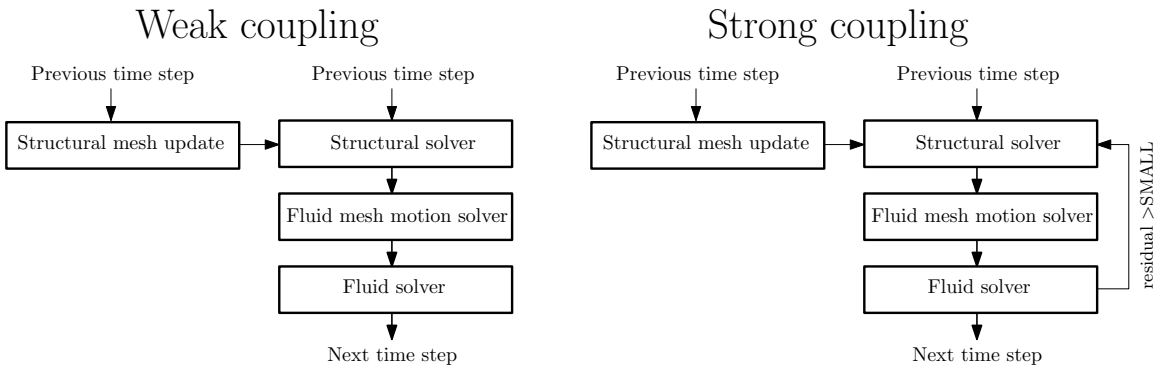


Figure 1.1: FSI solving process with weak and strong coupling using the partitioned approach

1.1 demonstrates the difference between the weak and the strong coupling. The solution strategy of the explicit method is analytically described in Hua-Dong Yao's presentation [3]. It is also important to underline the use of the `Aitken` adaptive under-relaxation technique to accelerate the coupling process.

In the FSI package, the solver used for the strong coupling between the fluid and the solid with the explicit partitioned approach is called `fsiFoam`. The solver comes with three tutorials. Examples of two of the `fsiFoam` tutorials `beamInCrossFlow` (left) and `HronTurekFsi3` (right) provided with the FSI package are shown in Figure 1.2.

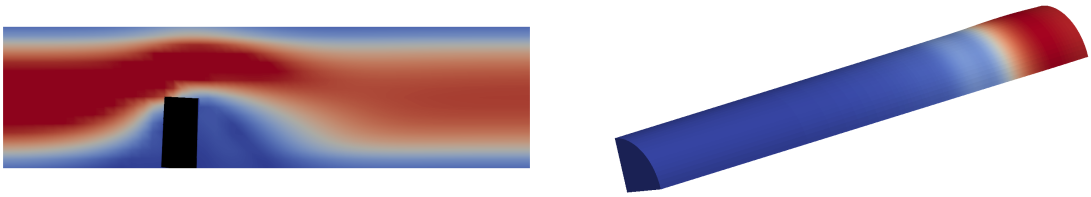


Figure 1.2: Examples of FSI tutorials

The mesh is coloured according to the velocity field with red representing the areas off the high magnitude of velocity. The bending of the vertical beam in the `beamInCrossFlow` and the deformation of the rod in `HronTurekFsi3` can be clearly seen.

1.2 Scope of the work

The latest version of `foam-extend` has not incorporated the FSI package in the main distribution. However, this package can be found in the `extend-bazaar` and it can be compiled independently. The FSI package is still under development and the "philosophy" underlying the way it is created differs to some degree from the other solvers of OpenFOAM. The reason for that is that it has to solve both the fluid and the structure sequentially using two different solvers that exchange information continuously. The way the FSI package is set up at the time of writing of this report, treats the flow solvers as classes that constitute the fluid-structure interaction library and not as executables, as usually happens in OpenFOAM. It is expected that the future versions of `foam-extend` will incorporate the FSI package in the main distribution, but it is not yet known if the structure

of the package will remain the same or change substantially to conform with the OpenFOAM norms.

The FSI problem has great applications for ocean and coastal engineering. The response of flexible structures for coastal protection, wave & tidal energy harnessing and oil industry under the wave impact is of significant importance for the survivability of these structures. However, none of the existing flow models in the FSI package allows free surface modelling. This becomes possible with the inclusion of a two phase flow model in the FSI package. Therefore, this study focuses on the implementation of a two phase flow solver in the FSI package for `foam-extend-3.1`.

In this work, the flow solver used as the basis to build a two-phase flow model is `interFoam`. It is a solver designed for two incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach, allowing for mesh motion (`interDyMFoam`) and turbulence modelling (i.e. laminar, Reynolds Averaged Simulation (RAS) and Large Eddy Simulation (LES)). The new flow model, namely `interFlow`, is part of the fluid-structure interaction library and it has similar structure as the existing flow models: `icoFlow`, `consistentIcoFlow` and `pisoFlow`.

The new flow model is tested with the `fsiFoam` solver. The tutorial used as a benchmark for the tests between the different flow models is the `beamInCrossFlow` tutorial, thanks to the similarities that it has with the well known `damBreak` tutorial, where the `interFoam` solver is usually tested. Instead of having an one phase fluid flow in the `beamInCrossFlow` tutorial, the collapsing of water column will be simulated with `interFlow` and its impact on the beam will be presented.

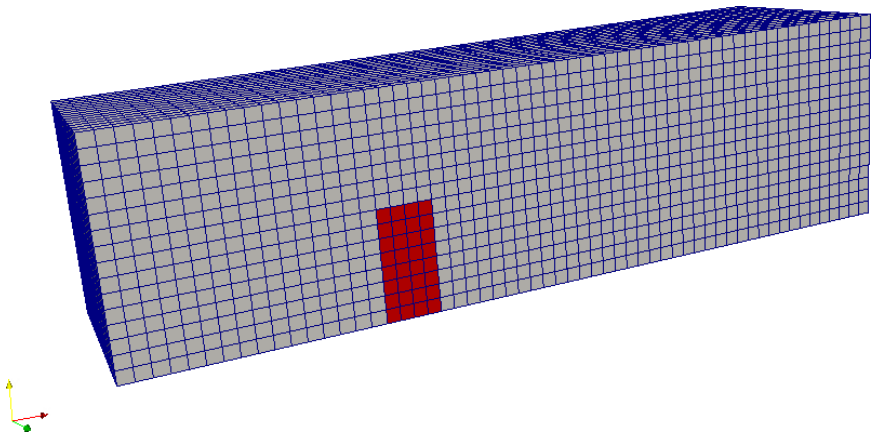


Figure 1.3: Perspective view of the mesh of the beamInCrossFlow tutorial. The solid is pained with red colour.

Chapter 2

The FSI package

This section begins with the necessary guidelines needed to install the FSI package for `foam-extend-3.1` and continues with the overview of the structure of the FSI package.

2.1 Installation of the FSI package

The FSI package can be downloaded independently from `http://openfoamwiki.net/index.php/Extend-bazaar/Toolkits/Fluid-structure_interaction` or it can be found in the `$WM_PROJECT_DIR/extend-bazaar/FluidStructureInteraction` directory.

Provided that `foam-extend-3.1` is downloaded and compiled, the user has to compile the FSI package independently. It is recommended to copy the FSI package in the user directory and compile it there to avoid damaging the central installation of `foam`, to keep the original package untouched and to avoid messing with permission rights. One can also compile the package at the `$WM_PROJECT_DIR` and the libraries and executables will still be created in the user's directory. To compile FSI in `$WM_PROJECT_DIR` execute the following commands:

```
. $HOME/foam/foam-extend-3.1/etc/bashrc
cd $WM_PROJECT_DIR/extend-bazaar
./Allwmake
```

This basically downloads and compiles the FSI package and cfMesh, which is not necessary. As stated above, the package can be downloaded from the repository and later unpacked, which is actually what happens when the previous `./Allwmake` script is executed. The commands are:

```
wget http://openfoamwiki.net/images/5/52/Fsi_31.tar.gz
tar zxvf Fsi_31.tar.gz
rm Fsi_31.tar.gz
```

To compile the FSI package at the user's directory, copy it and compile it there:

```
cd $WM_PROJECT_USER_DIR
cp -r $WM_PROJECT_DIR/extend-bazaar/FluidStructureInteraction/ .
```

Now that the FSI package is copied in the user's directory, the compilation can be done. Before starting, make sure that all the executables will be created in the `$(FOAM_USER_APPBIN)` and all the libraries in the `$(FOAM_USER_LIBBIN)`.

To compile the package in the user's directory execute the following commands:

```
cd $WM_PROJECT_USER_DIR/FluidStructureInteraction/src
chmod 755 *
./Allwclean
./Allwmake
```

At this point, it is important to mention that the FSI package contains numerous source files that have the same name with files of the central installation. Some of these files contain significant differences and some others are identical. However, not all the files contained in the FSI package are compiled and some of the corresponding original OpenFOAM files are used, despite the fact that some of them contain substantial modifications. Moreover, by looking inside the source files of the FSI package, one can notice that there are many parts commented out. It is expected that a cleaning up of the FSI packae will take place in the future before it becomes part of the main distribution.

A second point is that there is a chance that one faces problems compiling the `fluidStructure Interaction` library. If this library is not created further problems will occur in the compilation. The conflict has to do with the `IOReferencer` in the `constitutiveModel.C`. `constitutiveModel.C` is also included in the main installation in the `$FOAM_SOURCE/solidModels/constitutiveModel/` with a few changes. A rough solution is to use the latter file. There are of course some differences, between the two constitutive models, however, the `fsiFoam` can still run without using the file from the FSI package. This is achieved my commenting out the `stressModels/constitutiveModel/ constitutiveModel.C` file in the `src/fluidStructureInterface/Make/files` like that:
`/*stressModels/constitutiveModel/constitutiveModel.C*/`.
No further changes are required and the file
`~/foam /foam-extend-3.1/src/solidModels/constitutiveModel/constitutiveModel.C`
is used instead.

2.2 Overview of the FSI package

The `FluidStructureInteraction` directory contains two subdirectories: `run`, where all the tutorials are located, and `src`, where the source code is found. `src` contains three directories: `fluidStructureInteraction`, `solvers`, and `utilities`. The compilation of the `fluidStructure Interaction` library is prerequisite for the compilation of the solvers and some utilities.

After compiling the FSI package there is a series of solvers, utilities and libraries that are created. Taking a look at the `./Allwmake` script gives a good idea of the result of the compilation.

`Allwmake`

```
#!/bin/sh
set -x

wmake libso fluidStructureInteraction

wmake solvers/fsiFoam
wmake solvers/weakFsiFoam
wmake solvers/flowFoam
wmake solvers/stressFoam
wmake solvers/crackStressFoam
wmake solvers/thermalStressFoam

wmake utilities/set2dMeshThickness
wmake utilities/decomposePar
wmake utilities/reconstructPar
```

```
wmake libso utilities/foamCalcFunctions
wmake libso utilities/meshTools
wmake libso utilities/functionObjects/pointHistory
wmake libso utilities/functionObjects/energyHistory
wmake libso utilities/functionObjects/patchAvgTractionHistory
```

As it can be seen, the solvers `fsiFoam`, `weakFsiFoam`, `flowFoam`, `stressFoam`, `crackStressFoam` and `thermalStressFoam` are created and the utilities `set2dMeshThickness`, `decomposePar`, `reconstructPar`, `foamCalcFunctions`, `meshTools`, `pointHistory`, `energyHistory` and `patchAvgTractionHistory` are compiled.

The result of the compilation of the FSI package is the creation of the executables and libraries in the `$FOAM_USER_APPBIN` and `$FOAM_USER_LIBBIN` directories respectively, as listed in Table 2.1.

Table 2.1: Application and libraries created with the FSI package

Applications	**Libraries**
crackStressFoam	libenergyHistory.so
decomposeParFsi	libfluidStructureInteraction.so
flowFoam	libfoamStressCalcFunctions.so
fsiFoam	libmeshTools.so
reconstructParFsi	libpatchAvgTractionHistory.so
set2dMeshThickness	libpointHistory.so
stressFoam	
thermalStressFoam	
weakFsiFoam	

2.2.1 Overview of the fluidStructureInteraction library

As mentioned before, this study focuses on the `fsiFoam` and the `fluidStructureInteraction` library. Actually, all the modifications in the source code will take place in the `fluidStructureInteraction` directory. However, it is strongly recommended not to try to isolate these applications and compile them separately from the FSI package, because they are strongly bonded with many other source files and libraries.

The `fluidStructureInteraction` directory contains the following directories: `flowModels`, `fluidStructureInterface`, `numerics` and `stressModels`. The contents of these directories are shown in Table 2.2. It also contains the `Make` directory, which indicates that it can be compiled using the command `wmake libso`.

Each of the subdirectories listed in the Table 2.2 contains further subdirectories or source files. It is out of the scope of this document to present the contents in detail and the links between them. However, as it was mentioned before, not all of these files in the FSI package are compiled. Therefore, it is important to know which of these files are actually compiled. For this reason, a look at the `Make/files` is useful in order to realise which of them are actually compiled:

`fluidStructureInteraction/Make/files`

```
flowModels/flowModel/flowModel.C
flowModels/flowModel/newFlowModel.C
flowModels/icoFlow/icoFlow.C
flowModels/pisoFlow/pisoFlow.C
flowModels/consistentIcoFlow/consistentIcoFlow.C
flowModels/fvPatchFields/extrapolatedPressure/extrapolatedPressureFvPatchScalarField.C
flowModels/fvPatchFields/transitionalParabolicVelocity/
```

```
transitionalParabolicVelocityFvPatchVectorField.C

stressModels/fvPatchFields/tractionDisplacement/
tractionDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/tractionDisplacementIncrement/
tractionDisplacementIncrementFvPatchVectorField.C
stressModels/fvPatchFields/symmetryDisplacement/symmetryDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/fixedDisplacement/
fixedDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/fixedNormalDisplacement/
fixedNormalDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/fixedNormalDisplacementIncrement/
fixedNormalDisplacementIncrementFvPatchVectorField.C
stressModels/fvPatchFields/componentMixed/componentMixedPointPatchFields.C
stressModels/fvPatchFields/cohesiveZoneIncrementalModeI/
cohesiveZoneIncrementalFvPatchVectorField.C
stressModels/fvPatchFields/planeContactDisplacement/
planeContactDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/directionMixedDisplacement/
directionMixedDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/pRveTractionDisplacement/
pRveTractionDisplacementFvPatchVectorField.C
stressModels/fvPatchFields/pRveTractionDisplacementIncrement/
pRveTractionDisplacementIncrementFvPatchVectorField.C
stressModels/fvPatchFields/timeVaryingFixedNormalDisplacement/
timeVaryingFixedNormalDisplacementFvPatchVectorField.C
stressModels/stressModel/stressModel.C
stressModels/stressModel/newStressModel.C
stressModels/unsTotalLagrangianStress/unsTotalLagrangianStress.C
stressModels/unsTotalLagrangianStress/unsTotalLagrangianStressSolve.C
stressModels/unsIncrTotalLagrangianStress/unsIncrTotalLagrangianStress.C
stressModels/unsIncrTotalLagrangianStress/unsIncrTotalLagrangianStressSolve.C
stressModels/pRveUnsTotalLagrangianStress/pRveUnsTotalLagrangianStress.C
stressModels/pRveUnsIncrTotalLagrangianStress/pRveUnsIncrTotalLagrangianStress.C
stressModels/constitutiveModel/plasticityStressReturnMethods/plasticityStressReturn/
plasticityStressReturn.C
stressModels/constitutiveModel/plasticityStressReturnMethods/plasticityStressReturn/
newPlasticityStressReturn.C
stressModels/constitutiveModel/plasticityStressReturnMethods/aravasMises/aravasMises.C
stressModels/constitutiveModel/plasticityStressReturnMethods/newAravasMises/
newAravasMises.C
/*stressModels/constitutiveModel/constitutiveModel.C*/

stressModels/solidInterfaces/solidInterfaceTL/solidInterfaceTL.C
stressModels/solidInterfaces/solidInterfaceITL/solidInterfaceITL.C
stressModels/materialInterfaces/materialInterface/materialInterface.C
stressModels/materialInterfaces/TLMaterialInterface/TLMaterialInterface.C
stressModels/materialInterfaces/ITLMaterialInterface/ITLMaterialInterface.C
stressModels/simpleCohesiveLaws/simpleCohesiveLaw/simpleCohesiveLaw.C
stressModels/simpleCohesiveLaws/linear/linearSimpleCohesiveLaw.C
stressModels/simpleCohesiveLaws/Dugdale/DugdaleSimpleCohesiveLaw.C


numerics/ddtSchemes/backwardDdtScheme.C
numerics/leastSquaresSkewCorrected/leastSquaresSkewCorrected.C
numerics/leastSquaresVolPointInterpolation/leastSquaresVolPointInterpolation.C
numerics/skewCorrectedVectorSnGrad/skewCorrectedVectorSnGrad.C
numerics/skewCorrectedSnGrad/skewCorrectedSnGrads.C
numerics/backwardD2dt2Scheme/backwardD2dt2Schemes.C
```

```
numerics/fvMotionSolvers/velocityLaplacianFvMotionSolver.C
numerics/findRefCell/findRefCellVector.C
numerics/quadraticReconstruction/quadraticReconstruction.C
numerics/fvMeshSubset/fvMeshSubset.C

fluidStructureInterface/fluidStructureInterface.C

LIB = $(FOAM_USER_LIBBIN)/libfluidStructureInteraction /$
```

NB: The `constitutiveModel.C` is commented out to allow compiling.

Table 2.2: Contents of the fluidStructureInteraction directory

flowModels	consistentIcoFlow flowModel fvpatchFields icoFlow pisoFlow
fluidStructureInteraction	fluidStructureInteraction.C and fluidStructureInteraction.H
Make	files options
numerics	backwardD2dt2Scheme ddtSchemes findRefCell fvc fvMeshSubset fvMotionSolvers ggi leastSquaresSkewCorrected leastSquaresVolPointInterpolation quadraticReconstruction skewCorrectedSnGrad skewCorrectedVectorSnGrad
stressModels	componentReference constitutiveModel fvPatchFields materialInterfaces pRveUnsIncrTotalLagrangianStress pRveUnsTotalLagrangianStress simpleCohesiveLaws solidInterfaces stressModel unsIncrTotalLagrangianStress unsTotalLagrangianStress

Chapter 3

An insight of the flow models

3.1 The fsiFoam and its connection to the flow models

As reported before, the scope of this work is to introduce a two phase flow model in the FSI package that can be used with `fsiFoam` for strong fluid structure interaction. Therefore the best place to start looking the code is the source files of the `fsiFoam` solver. The source code of the solver can be found in `FluidStructureInteraction/src/solvers/fsiFoam`. The files `createStressMesh.H`, `fsiFoam.C` and the directory `Make` are located there. As it can be seen from `Make/files`, only the `fsiFoam.C` is compiled. The author of the code is Zeljko Tukovic. A look at the code will help the reader proceed further.

fsiFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "fluidStructureInterface.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
int main(int argc, char *argv[])
{
#   include "setRootCase.H"
#   include "createTime.H"
#   include "createDynamicFvMesh.H"
#   include "createStressMesh.H"

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

    fluidStructureInterface fsi(mesh, stressMesh);
    Info<< "\nStarting time loop\n" << endl;
    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        fsi.initializeFields();
        fsi.updateInterpolator();
        scalar residualNorm = 0;
        do
        {
```

```
            fsi.outerCorr()++;
            fsi.updateDisplacement();
            fsi.moveFluidMesh();
            fsi.flow().evolve();
            fsi.updateForce();
            fsi.stress().evolve();
            residualNorm =
                fsi.updateResidual();
        }
        while
        (
            (residualNorm > fsi.outerCorrTolerance())
         && (fsi.outerCorr() < fsi.nOuterCorr())
        );
        fsi.stress().updateTotalFields();
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return(0);
}


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

The first thing to notice is the inclusion files. These files should be looked together with the `Make/options` file.

`fsiFoam/Make/options`

```
EXE_INC = \
    -I../../fluidStructureInteraction/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/finiteArea/lnInclude \
    -I$(LIB_SRC)/lagrangian/basic/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/tetFiniteElement/lnInclude \
    -I$(LIB_SRC)/solidModels/lnInclude

EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lfluidStructureInteraction \
    -lfiniteVolume \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lincompressibleTransportModels \
    -ldynamicFvMesh \
    -ldynamicMesh \
    -ltopoChangerFvMesh \
    -lmeshTools \
    -lsolidModels
```

It is easy to see where each of the included files are located, by using the command `locate <filename>` in the terminal. If a file exists in more than one locations, like it often happens in the FSI package, the file `Make/options` defines which of these files will be used, according to the order of the list of locations stated there. It can be seen from `fsiFoam/Make/options` that the files contained in `FluidStructureInteraction/`

`src/fluidStructureInteraction/lnInclude` are taken into account first. The location of the source files of `fsiFoam` are shown bellow.

fvCFD.H → $(LIB_SRC)/finiteVolume/lnInclude
dynamicFvMesh.H → $(LIB_SRC)/dynamicMesh/dynamicFvMesh/lnInclude
fluidStructureInterface.H → FluidStructureInteraction/src/fluidStructureInteraction/lnInclude
setRootCase.H → $(LIB_SRC)/foam/lnInclude
createDynamicFvMesh.H → $(LIB_SRC)/dynamicMesh/dynamicFvMesh/lnInclude
createStressMesh.H → fsiFoam directory (Header file)

All the other files apart from `createStressMesh.H` and `fluidStructureInterface.H` are located in the central installation of OpenFOAM. If there are any extra inclusions can only exist in the two files mentioned. `createStressMesh.H` does not include extra files. The case is not the same for `fluidStructureInterface.H`. It is useful to take a look at this file and its inclusions. The included files and their locations are listed below.

flowModel.H → /src/fluidStructureInteraction/flowModels/flowModel
stressModel.H → /src/fluidStructureInteraction/stressModels/stressModel
IOdictionary.H → $(LIB_SRC)/foam/lnInclude
patchToPatchInterpolation.H → $(LIB_SRC)/foam/lnInclude
dynamicFvMesh.H → $(LIB_SRC)/dynamicMesh/dynamicFvMesh/lnInclude
ggiInterpolation.H → $(LIB_SRC)/foam/lnInclude
extendedGgiInterpolation.H → src/fluidStructureInteraction/numerics/ggi/ExtendedGGIInterpolation

NB: The `Make/options` file for the `fluidStructureInterface.H` is one level up from the location that it is found, i.e. in the `src/fluidStructureInteraction` directory.

As it can be seen, most of the included files in `fluidStructureInterface.H` belong to the FSI package. If one continues looking into each one of these files, he would realise that they point to other files in the FSI package. Keeping in mind that the scope is to adjust the flow models, the most reasonable step is to further look only in the `flowModel.H`, which is the header file of `flowModel.C`, and both of them are located in `/src/fluidStructureInteraction/flowModels/flowModel` directory. In this directory, the source file `newFlowModel.C` exists, which recognises the available flow models and ensures that the dictionary is not entered twice in the database.

The `flowModel.H` is used for the declaration of the `flowModel` class. This file dictates that the dictionary `flowProperties` is read, where the flow model in use is stated. At the end of `flowModel.H` the member function `evolve` is defined. At this point, one should return to the `fsiFoam.C`, where the only reference to the fluid solver is the `fsi.flow().evolve()` and realise how the two files are linked.

The connection between the `fsiFoam` solver and the flow models should be clear:
The `fsiFoam.C` file includes the `fluidStructureInterface.H`, which includes the `flowModel.H`.

Now, the question is how a flow model is compiled and becomes available in the FSI package. Going back to the `fluidStructureInteraction/Make/files` file, one can see that the flow models are compiled independently: `icoFlow.C`, `pisoFlow.C` etc. The corresponding header files of the flow models (`icoFlow.H`, `pisoFlow.H` etc) declare the classes of the flow models. In the declaration the name of each model is defined (`icoFlow`, `pisoFlow` etc) which is recognised as a flow model through the `flowModel.H`.

3.2 The existing flow models

3.2.1 Description of the existing flow models

The available flow models in the FSI package are `icoFlow`, `pisoFlow` and `consistentIcoFlow`. This can be confirmed by looking at the `fluidStructureInteraction/flowModels` direcory or by putting a dummy as a flow model in the `flowProperties` file in the `fluid/constant` directory of one of the existing tutorials of the FSI package. Unfortunately, all the flow models lack description at the beginning of the source files.

The **icoFlow** flow model is based on the `icoFoam` solver, which is a transient solver for incompressible, laminar flow of Newtonian fluids.

The **consistentIcoFlow** flow model is again based on the `icoFoam` solver, as it is declared in the `consistentIcoFlow.H` file. However, there are significant differences between `icoFlow.C` and `consistentIcoFlow.C`. This can be easily seen by comparing the source file using the tool `kompare`. Assuming that the user is in the `FluidStructureInteractionsrc/fluidStructureInteraction/flowModels` the `kompare` program is executed as follows:

```
kompare icoFlow/icoFlow.C consistentIcoFlow/consistentIcoFlow.C
```

The same can be done for the `*.H` files. An interesting and easy thing to notice from the previous comparison is the inclusion files, listed in Table 3.1. From the inclusion files of `consistentIcoFlow.H`, it becomes obvious that `consistentIcoFlow` is derived from the `icoFlow`. The latter is also supported from the class declaration of `consistentIcoFlow`:

`consistentIcoFlow.H`

```
namespace Foam
{
namespace flowModels
{

/*---------------------------------------------------------------------------*\
                        Class consistentIcoFlow Declaration
\*---------------------------------------------------------------------------*/

class consistentIcoFlow
:
    public icoFlow
```

Instead of the previous name, the public name in `icoFlow.H` is `flowModel`.

Table 3.1: Comparison of the source files of the flow models

icoFlow.C	consistentIcoFlow.C	icoFlow.H	consistentIcoFlow.H
icoFlow.H	consistentIcoFlow.H	flowModel.H	icoFlow.H
volFields.H	volFields.H	volFields.H	volFields.H
fvm.H	fvm.H	surfaceFields.H	surfaceFields.H
fvc.H	fvc.H		
fvMatrices.H	fvMatrices.H		
addToRunTimeSelectionTable.H	addToRunTimeSelectionTable.H		
findRefCell.H	findRefCell.H		
adjustPhi.H	adjustPhi.H		
fluidStructureInterface.H			
fixedGradientFvPatchFields.H			

It is easy to check that `icoFlow` has very similar structure to `icoFoam`. However, since a solver with name like `constistentIcoFoam` cannot be found in `$WM_PROJECT_DIR/applications/solvers`, a side by side comparison is not possible.

The **pisoFlow** flow model is based on the `pisoFoam` solver, which is a a transient solver for incompressible flow allowing generic turbulence modelling, i.e. laminar, RAS or LES may be selected. A quick comparison between `icoFlow.H` and `pisoFlow.H` reveals that the difference is in the selection of the turbulence model. In `pisoFlow.H` the files `singlePhaseTransportModel.H` and `turbulenceModel.H` are included complementary to the files included in `icoFlow.H`. In the class declaration of `pisoFlow` the transport model is `singlePhaseTransportModel laminarTransport_`, while in `icoFlow` the transport properties are read from the `transportProperties_` dictionary. Similar things can be noticed also when looking at the `*.C` files.

3.2.2 Comparison of the flow models with the corresponding solvers

In this section, a side by side comparison of the source code of the flow model and the corresponding solver will be presented. For this scope, `pisoFlow` and `pisoFoam` are compared. As it has been mentioned earlier in this document, the flow models are classes of the `fluidStructureInteraction` library, which mainly distinguishes them from the solvers that compile independently to executables. This implies substantial differences to the source code. Despite these differences, one can follow the two codes almost line by line as shown below.

The overall comparison reveals some noticeable differences:

- `pisoFoam.C` includes the files `createFields.H, readPISOControls.H, CourantNo.H` and `continuityErrs.H` that contain the required source code. For `pisoFlow` instead, all the source code is contained in `pisoFlow.C` .
- The nomenclature of the fields changes. The fields are named with the same variable as before followed by "_". For example: `U_`, `p_`, `rho_`, `phi_`, `turbulence_`, `laminarTransport_` etc.
- The type of the field (`volVectorField, surfaceScalarField, dimensionedScalar` etc) is not declared together with the field. This is done in the `pisoFlow.H` file.
- There are other small changes like ";", "," and "()" that one should take into account.

pisoFoam.C

```
\*---------------------------------------------------------*/
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulenceModel.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
int main(int argc, char *argv[])
{
#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"
#   include "createFields.H"
#   include "initContinuityErrs.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

createFields.H contains:

```
    Info<< "Reading field U\n" << endl;
    volVectorField U
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
    Info<< "Reading field p\n" << endl;
```

pisoFlow.C

```
\*---------------------------------------------------------*/
#include "pisoFlow.H"
#include "volFields.H"
#include "fvm.H"
#include "fvc.H"
#include "fvMatrices.H"
#include "addToRunTimeSelectionTable.H"
#include "findRefCell.H"
#include "adjustPhi.H"
#include "fluidStructureInterface.H"
#include "fixedGradientFvPatchFields.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace flowModels
{
// * * * * * * * * Static Data Members * * * * * * * * * //
defineTypeNameAndDebug(pisoFlow, 0);
addToRunTimeSelectionTable(flowModel,pisoFlow,dictionary);
// * * * * * * * * * * Constructors * * * * * * * * * * * //
pisoFlow::pisoFlow(const fvMesh& mesh)
:
    flowModel(this->typeName, mesh),
    U_
    (
        IOobject
        (
            "U",
            runTime().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    ),
```

```
    volScalarField p
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );

#   include "createPhi.H"
```

createPhi.H contains:

```
Info<<"Reading/calculating face flux field phi\n"<<endl;
surfaceScalarField phi
(
    IOobject
    (
        "phi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    linearInterpolate(U) & mesh.Sf()
);
```

end of *createPhi.H*

```
    label pRefCell = 0;
    scalar pRefValue = 0.0;
    setRefCell(p, mesh.solutionDict().subDict("PISO"),
    pRefCell, pRefValue);
    singlePhaseTransportModel laminarTransport(U, phi);
    autoPtr<incompressible::turbulenceModel> turbulence
    (
        incompressible::turbulenceModel::New(U, phi,
        laminarTransport)
    );
```

rho is not required in an incompressible solver, but it is needed for the calculation of the viscous and the pressure forces

end of *createFields.H*

there are no member functions in pisoFoam to calculate pressure and viscous forces

```
    p_
    (
        IOobject
        (
            "p",
            runTime().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    ),
    gradp_(fvc::grad(p_)),
```

```
    phi_
    (
        IOobject
        (
            "phi",
            runTime().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        fvc::interpolate(U_) & mesh.Sf()
    ),
```

```
    laminarTransport_(U_, phi_),
    turbulence_
    (
        incompressible::turbulenceModel::New(U_, phi_,
        laminarTransport_)
    ),
    rho_
    (
        IOdictionary
        (
            IOobject
            (
                "transportProperties",
                runTime().constant(),
                mesh,
                IOobject::MUST_READ,
                IOobject::NO_WRITE
            )
        ).lookup("rho")
    )
{}
// * * * * * * * * * Member Functions  * * * * * * * * * //

const volVectorField& pisoFlow::U() const
{
```

```
    return U_;
}
const volScalarField& pisoFlow::p() const
{
    return p_;
}
//- Patch viscous force (N/m2)
tmp<vectorField> pisoFlow::patchViscousForce(const label
patchID)
const
{
    tmp<vectorField> tvF
    (
        new vectorField(mesh().boundary()[patchID].size(),
         vector::zero)
    );
    tvF() =
        rho_.value()
       *(
            mesh().boundary()[patchID].nf()
          & turbulence_->devReff()().boundaryField()
          [patchID]
        );
//     vectorField n = mesh().boundary()[patchID].nf();
//     tvF() -= n*(n&tvF());
    return tvF;
}
//- Patch pressure force (N/m2)
tmp<scalarField> pisoFlow::patchPressureForce(const label
patchID) const
{
    tmp<scalarField> tpF
    (
        new scalarField(mesh().boundary()[patchID].size(), 0)
    );
    tpF() = rho_.value()*p().boundaryField()[patchID];
    return tpF;
}
//- Patch viscous force (N/m2)
tmp<vectorField> pisoFlow::faceZoneViscousForce
(
    const label zoneID,
    const label patchID
)
const
{
    vectorField pVF = patchViscousForce(patchID);
    tmp<vectorField> tvF
    (
        new vectorField(mesh().faceZones()[zoneID].size(),
        vector::zero)
    );
    vectorField& vF = tvF();
    const label patchStart =
        mesh().boundaryMesh()[patchID].start();
    forAll(pVF, i)
    {
        vF[mesh().faceZones()[zoneID].whichFace(patchStart
        + i)] =  pVF[i];
```

```
    }
    // Parallel data exchange: collect pressure field on
    all processors
    reduce(vF, sumOp<vectorField>());
    return tvF;
}
//- Patch pressure force (N/m2)
tmp<scalarField> pisoFlow::faceZonePressureForce
(
    const label zoneID,
    const label patchID
)
const
{
    scalarField pPF = patchPressureForce(patchID);

    tmp<scalarField> tpF
    (
        new scalarField(mesh().faceZones()[zoneID].size(), 0)
    );
    scalarField& pF = tpF();
    const label patchStart =
        mesh().boundaryMesh()[patchID].start();
    forAll(pPF, i)
    {
        pF[mesh().faceZones()[zoneID].whichFace(patchStart
        + i)] = pPF[i];
    }
    // Parallel data exchange: collect pressure field on
    all processors
    reduce(pF, sumOp<scalarField>());
    return tpF;
}
tmp<scalarField> pisoFlow::faceZoneMuEff
(
    const label zoneID,
    const label patchID
)
const
{
    scalarField pMuEff = rho_.value()*turbulence_
    ->nuEff()().boundaryField()[patchID];
    tmp<scalarField> tMuEff
    (
        new scalarField(mesh().faceZones()[zoneID].size(), 0)
    );
    scalarField& muEff = tMuEff();
    const label patchStart =
        mesh().boundaryMesh()[patchID].start();
    forAll(pMuEff, i)
    {
        muEff[mesh().faceZones()
        [zoneID].whichFace(patchStart + i)] =
            pMuEff[i];
    }
    // Parallel data exchange: collect pressure field on
    all processors
    reduce(muEff, sumOp<scalarField>());
    return tMuEff;
```

```
Info<< "\nStarting time loop\n" << endl;
while (runTime.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;
#   include "readPISOControls.H"
```

readPISOControls.H contains:

```
dictionary piso = mesh.solutionDict().subDict("PISO");
int nCorr(readInt(piso.lookup("nCorrectors")));
int nNonOrthCorr = piso.lookupOrDefault<int>
  ("nNonOrthogonalCorrectors", 0);
bool momentumPredictor = piso.lookupOrDefault<Switch>
  ("momentumPredictor", true);
bool transonic = piso.lookupOrDefault<Switch>
  ("transonic", false);
int nOuterCorr = piso.lookupOrDefault<int>
  ("nOuterCorrectors", 1);
```

end of *readPISOControls.H*

```
#   include "CourantNo.H"
```

CourantNo.H contains:

```
scalar CoNum = 0.0;
scalar meanCoNum = 0.0;
scalar velMag = 0.0;
if (mesh.nInternalFaces())
{
    surfaceScalarField magPhi = mag(phi);
    surfaceScalarField SfUfbyDelta =
        mesh.surfaceInterpolation::deltaCoeffs()*magPhi;
    const scalar deltaT = runTime.deltaT().value();
    CoNum = max(SfUfbyDelta/mesh.magSf()).value()*deltaT;

    meanCoNum = (sum(SfUfbyDelta)/sum(mesh.magSf())).
      value()*deltaT;
    velMag = max(magPhi/mesh.magSf()).value();
}
Info<< "Courant Number mean: " << meanCoNum
    << " max: " << CoNum
    << " velocity magnitude: " << velMag
    << endl;
```

end of *CourantNo.H*

```
// Pressure-velocity PISO corrector
{
    // Momentum predictor
    fvVectorMatrix UEqn
    (
       fvm::ddt(U)
```

```
}
```

the info statements exist in fsiFoam.C

```
void pisoFlow::evolve()
{
    Info << "Evolving flow model" << endl;
    const fvMesh& mesh = flowModel::mesh();



    int nCorr(readInt(flowProperties().lookup("nCorrectors")));
    int nNonOrthCorr =
        readInt(flowProperties().
        lookup("nNonOrthogonalCorrectors"));



// Prepare for the pressure solution
label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p_, flowProperties(), pRefCell, pRefValue);
if(mesh.moving())
{
    // Make the fluxes relative
    phi_ -= fvc::meshPhi(U_);
}

// CourantNo
{
   scalar CoNum = 0.0;
   scalar meanCoNum = 0.0;
   scalar velMag = 0.0;
   if (mesh.nInternalFaces())
   {

      surfaceScalarField SfUfbyDelta =
      mesh.surfaceInterpolation::deltaCoeffs()*mag(phi_);

   CoNum = max(SfUfbyDelta/mesh.magSf()).
     value()*runTime().deltaT().value();
   meanCoNum = (sum(SfUfbyDelta)/sum(mesh.magSf())).
     value()*runTime().deltaT().value();
   velMag = max(mag(phi_)/mesh.magSf()).value();
   }
Info<< "Courant Number mean: " << meanCoNum
    << " max: " << CoNum
    << " velocity magnitude: " << velMag << endl;
}



fvVectorMatrix UEqn
(
   fvm::ddt(U_)
```

```
      + fvm::div(phi, U)
      + turbulence->divDevReff(U)
    );
    UEqn.relax();
    if (momentumPredictor)
    {
        solve(UEqn == -fvc::grad(p));
    }


    // --- PISO loop
    for (int corr = 0; corr < nCorr; corr++)
    {
        volScalarField rUA = 1.0/UEqn.A();
        U = rUA*UEqn.H();
        phi = (fvc::interpolate(U) & mesh.Sf())
          + fvc::ddtPhiCorr(rUA, U, phi);
        adjustPhi(phi, U, p);
      // Non-orthogonal pressure corrector loop
      for (int nonOrth=0; nonOrth<=nNonOrthCorr;nonOrth++)
      {
         // Pressure corrector
         fvScalarMatrix pEqn
         (
             fvm::laplacian(rUA, p) == fvc::div(phi)
         );
         pEqn.setReference(pRefCell, pRefValue);
         if
         (
            corr == nCorr-1
            && nonOrth == nNonOrthCorr
         )
         {
            pEqn.solve(mesh.solutionDict().
            solver("pFinal"));
         }
         else
         {
            pEqn.solve();
         }
         if (nonOrth == nNonOrthCorr)
         {
            phi -= pEqn.flux();
         }
      }

#  include "continuityErrs.H"
```

continuityErrs.H contains:

```
{
    volScalarField contErr = fvc::div(phi);
    sumLocalContErr = runTime.deltaT().value()*
        mag(contErr)().weightedAverage(mesh.V()).value();
    globalContErr = runTime.deltaT().value()*
        contErr.weightedAverage(mesh.V()).value();
    cumulativeContErr += globalContErr;
    Info<< "time step continuity errors : sum local ="
     << sumLocalContErr
        << ", global = " << globalContErr
```

```
      + fvm::div(phi_, U_)
      + turbulence_->divDevReff(U_)
    );



     solve(UEqn == -gradp_);
```

gradp_ is defined in the constructors as fvc::grad(p_)

```
// --- PISO loop
volScalarField rUA = 1.0/UEqn.A();
for (int corr=0; corr<nCorr; corr++)
{
   U_ = rUA*UEqn.H();
   phi_ = (fvc::interpolate(U_) & mesh.Sf());



   for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
   {

      fvScalarMatrix pEqn
      (
         fvm::laplacian(rUA, p_) == fvc::div(phi_)
      );
      pEqn.setReference(pRefCell, pRefValue);
      if
      (
         corr == nCorr-1
         && nonOrth == nNonOrthCorr
      )
      {
         pEqn.solve(mesh.solutionDict().
         solver("pFinal"));
      }
      else
      {
         pEqn.solve();
      }
      if (nonOrth == nNonOrthCorr)
      {
          phi_ -= pEqn.flux();
      }
   }



// Continuity error
{
   volScalarField contErr = fvc::div(phi_);
   scalar sumLocalContErr = runTime().deltaT().value()*
       mag(contErr)().weightedAverage(mesh.V()).value();
   scalar globalContErr = runTime().deltaT().value()*
       contErr.weightedAverage(mesh.V()).value();

Info<< "time step continuity errors : sum local = "
    << sumLocalContErr
    << ", global = " << globalContErr
```

```
        << ", cumulative = " << cumulativeContErr
        << endl;
}
```

end of *continuityErrs.H*

```
   U -= rUA*fvc::grad(p);
   U.correctBoundaryConditions();
   }
}
        turbulence->correct();

   runTime.write();
   Info<< "ExecutionTime = " << runTime.elapsedCpuTime()
    << " s"
       << "  ClockTime = " << runTime.elapsedClockTime()
    << " s"
       << nl << endl;
}
   Info<< "End\n" << endl;
   return 0;
}
// ************************************************* //
```

```
    << endl;
}

   gradp_ = fvc::grad(p_);
   U_ -= rUA*gradp_;
   U_.correctBoundaryConditions();
}

    turbulence_->correct();

}

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

} // End namespace flowModels
} // End namespace Foam

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

A part of the `pisoFlow.H` file is presented below in order to demonstrate how the declaration of the type of the fields is performed. It should be underlined that the fields in `pisoFlow.H` and `pisoFlow.C` must be in the same order. It should be noted that the class contains also constructors and destructors that have been omitted here. Also notice that the private data and the member functions are in the same order as well.

`pisoFlow.H`

```
\*---------------------------------------------------------------------------*/
#ifndef pisoFlow_H
#define pisoFlow_H
#include "flowModel.H"
#include "volFields.H"
#include "surfaceFields.H"
#include "singlePhaseTransportModel.H"
#include "turbulenceModel.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace flowModels
{
/*---------------------------------------------------------------------------*\
                        Class pisoFlow Declaration
\*---------------------------------------------------------------------------*/
class pisoFlow
:
    public flowModel
{
    // Private data
        //- Velocity field
        volVectorField U_;
        //- Pressure field
        volScalarField p_;
        //- Pressure field
        volVectorField gradp_;
        //- Flux field
        surfaceScalarField phi_;
```

```
        //- Transport model
        singlePhaseTransportModel laminarTransport_;
        ...


    // Member Functions
        // Access
            //- Return velocity field
            virtual const volVectorField& U() const;
            //- Return velocity field
            volVectorField& U()
            {
                return U_;
            }
            //- Return pressure field
            virtual const volScalarField& p() const;
            //- Return pressure field
            volScalarField& p()
            {
                return p_;
            }
            //- Return pressure gradient
            volVectorField& gradp()
            {
                return gradp_;
            }
            //- Return flux field
            surfaceScalarField& phi()
            {
                return phi_;
            }
            ...


            //- Patch viscous force (N/m2)
            virtual tmp<vectorField> patchViscousForce
            (
                const label patchID
            ) const;
            //- Patch pressure force (N/m2)
            virtual tmp<scalarField> patchPressureForce
            (
                const label patchID
            ) const;
            ...

            //- Evolve the flow model
            virtual void evolve();
};
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace flowModels
} // End namespace Foam
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif
// ************************************************************************* //
```

3.3 Running the beamInCrossFlow tutorial with all the existing flow models

In this section the `beamInCrossFlow` tutorial will be run with the all the available flow models in order to demonstrate the extra files that are required and the amendments needed to the existing files. The reason for selecting this tutorial is explained in Section 1.2. All the tutorials for the `fsiFoam` are located in the `fluidStructureInteraction/run/fsiFoam` directory. The tutorial used in the presentation made by Hua-Dong Yao [3] was the `3dTube` tutorial with the `consistenIcoFlow` as a flow model. Useful details about running the tutorial can be found in the presentation.

The tutorial provided in the FSI package is designed for the `consistenIcoFlow` flow model. The tutorials in the FSI package differ from the common tutorials in OpenFOAM because of the fact that two solvers are linked together. The `beamInCrossFlow` with `consistenIcoFlow` will be used to demonstrate how the tutorial runs.

3.3.1 beamInCrossFlow with icoFoam and consistentIcoFlow

As `consistenIcoFlow` is derived from `icoFlow`, there are only minor changes on the tutorial. For this reason the tutorial is presented once for the two flow models. The tree of the directories and files of the examined tutorial shown in Figure 3.1 gives a good overview of its structure. The tutorial contains a fluid and a solid directory that have the necessary files for the flow and the structural simulation respectively. These two folders are linked as demonstrated later with the scripts contained in the `beamInCrossFlow` directory.

The first steps to run the tutorial is to make sure that the fluid and the solid are coupled. This is achieved my executing the `makeSerialLinks` script located in the `beamInCrossFlow` directory. If the terminal is bash, the following steps should take place. It is recommended to destroy the existing links to avoid any problems by executing the `removeSerialLinks` script.

```
chmod 755 *
sed -i s/tcsh/sh/g *Links
./removeSerialLinks fluid solid
./makeSerialLinks fluid solid
```

At this point it is useful to see what happens when the `makeSerialLinks` script is executed. The $1 refers to the first directory stated after the `./makeSerialLinks`, i.e. fluid, and the $2 refers to the second directory, i.e. solid. It can be seen that by running the `makeSerialLinks` script soft links pointing to `solid/0`, `solid/constant` and `solid/system` are created in `fluid/0`, `fluid/constant` and `fluid/system` respectively under a directory named `solid`.

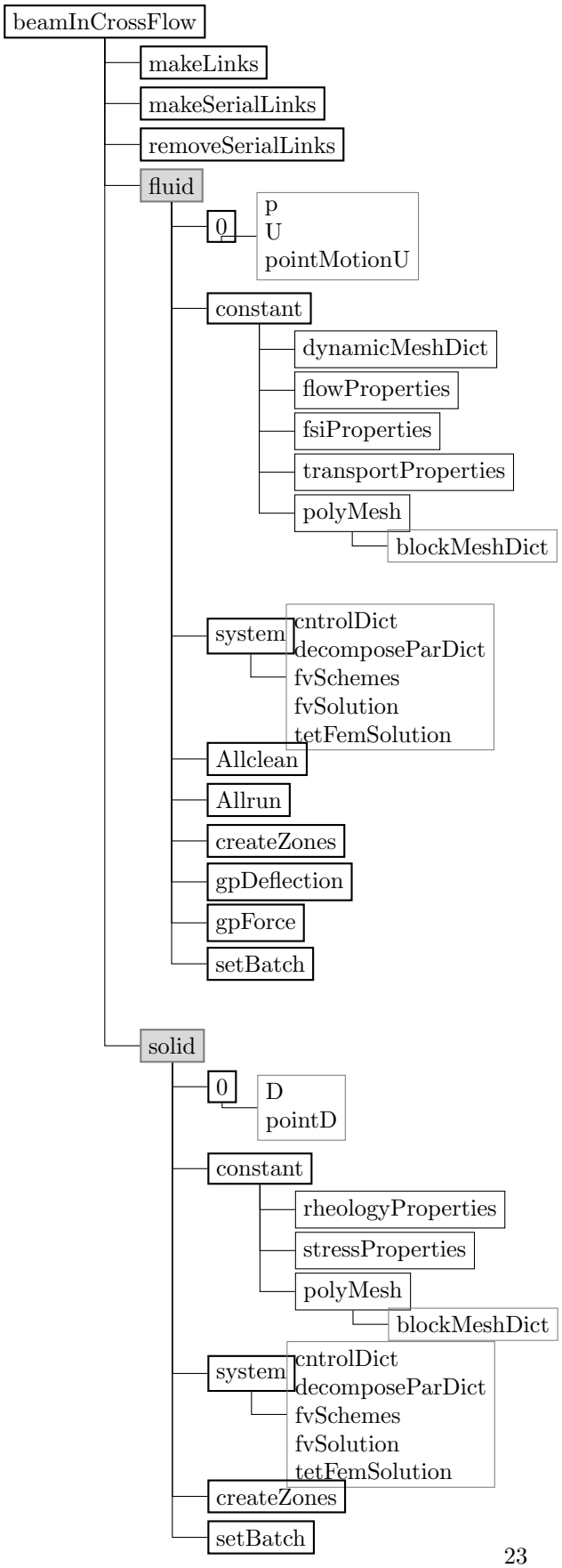
`makeSerialLinks`

```
#!/bin/sh
cd $1
cd constant
ln -s ../../$2/constant solid
cd ../system
ln -s ../../$2/system solid
cd ../0
ln -s ../../$2/0 solid
cd ../../
```

The script `makeLinks` does exactly the same as `makeSerialLinks`, but it is used for parallel simulations.

After the links are created, the tutorial is ready to run. This can be done with the `Allrun` script in the `fluid` directory. It is also recommended to run the `Allclean` script before the `Allrun` in order to clean the case from previous runs.

Figure 3.1: Tree of files and directories of beamInCrossFlow tutorial



It is useful to have a look at the `Allrun` script here.

`Allrun`

```
#!/bin/sh

# Source tutorial run functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions

# Get application name
application=`getApplication`

runApplication -l log.blockMesh.solid blockMesh -region solid
runApplication -l log.setSet.solid setSet -case ../solid -batch ../solid/setBatch
runApplication -l log.setToZones.solid setsToZones -case ../solid -noFlipMap

runApplication blockMesh
runApplication setSet -batch setBatch
runApplication setsToZones -noFlipMap

# Build setInletVelocity function object
wmake libso ../setInletVelocity

runApplication $application
```

The application name, i.e. the solver, is defined through the `getApplication`. This looks into the `system/controlDict` where the application name is declared. In this case the application is the `fsiFoam`. The coupling process requires the information of the FSI zones and this is done through the `setSet` and `setToZones` commands. More details can be found in Hua-Dong Yao's presentation[3].

Another thing to notice is that the compilation of the `setInletVelocity` library takes place every time that `Allrun` script is executed. This is a function object that sets a three-dimensional parabolic velocity profile at the channel inlet. Once compiled, there is a library created in the `USER_LIBBIN` named `libsetInletVelocity.so`. There is no need to compile the library every time. For all the test cases that will be tried later, it is recommended to comment out the compilation of the `setInletVelocity` library from the `Allrun` script using the # sign before `wmake`. This is done for the sake of time saving.

The use of `setInletVelocity` library needs to be declared in the `controlDict` as a function. There are also another two functions used in the tutorial examined. These are the `pointHistory` and `forces` also declared in the `controlDict` as follows.

`fluid/system/controlDict`

```
functions
(
   beamReport
   {
       type pointHistory;
       functionObjectLibs  ("libpointHistory.so");
       refHistoryPoint (0.45 0.15 -0.15);
       region solid;
   }
   setInletVelocity
   {
       type setInletVelocity;
       functionObjectLibs  ("libsetInletVelocity.so");
   }
   forces
   {
       type               forces;
```

```
        functionObjectLibs  ( "libforces.so" );
        outputControl       timeStep;
        outputInterval      1;
        patches             (interface);
        pName               p;
        UName               U;
        rhoName             rhoInf;
        log                 true;
        rhoInf              1000;
        CofR                (0.5 0.1 0);
    }
);
```

The last thing to check before running the tutorial is the `flowProperties` file. As mentioned before, this is where the flow model and its coefficients are declared. In this case, if the flow model is `icoFlow`, then only the `icoFlowCoeffs` has to be defined. However, if the the selected flow model is `consistentIcoFlow` then both `icoFlowCoeffs` and `consistentIcoFlowCoeffs` have to be defined. This is because `consistentIcoFlow` is derived from `icoFlow`. It is out of the scope of the document to explain the coefficients of the flow models.

`fluid/constant/flowProperties`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      flowProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
flowModel consistentIcoFlow;
icoFlowCoeffs
{
    nCorrectors 3;
    nNonOrthogonalCorrectors 1;
}
consistentIcoFlowCoeffs
{
    nCorrectors 3;
    nNonOrthogonalCorrectors 1;
}
```

NB:One can leave as many flow coefficients as he wants in the `flowProperties` file. The flow model will only read those that correspond to it.

Now, in order to run the tutorial case simply execute `./Allrun` in the fluid directory.

3.3.2 beamInCrossFlow with pisoFlow

In order to run the `beamInCrossFlow` tutorial with the `pisoFlow` flow model, the tutorial will be copied and saved with a new name and the necessary changes in the files will be presented.

First the tutorial is copied to a different folder:

```
cp -r beamInCrossFlow beamInCrossFlow_pisoFlow
```

Then in the case directory, the `fluid/constant/flowProperties` file is changed so that the `pisoFlow` model is called and its coefficients are declared.

fluid/constant/flowProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      flowProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
flowModel pisoFlow;
pisoFlowCoeffs
{
    nCorrectors 3;
    nNonOrthogonalCorrectors 1;
}
```

The next thing to be done is to define the transport model in the `fluid/constant/transportProperties`. For that purpose, the first line is added to the `transportProperties` file. The Newtonian model is selected. The `transportProperties` file should look like this:

fluid/constant/transportProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
transportModel  Newtonian;
nu              nu [0 2 -1 0 0 0 0] 1e-3;
rho             rho [1 -3 0 0 0 0 0] 1000;
```

Moreover, the turbulence properties need to be defined in the `constant/turbulenceProperties` file. For convenience, this file can be copied from existing tutorials in the case directory.

```
cp $FOAM_TUTORIALS/incompressible/pisoFoam/ras/cavity/constant/turbulenceProperties
 fluid/constant/
```

For the sake of simplicity, the `laminar` model is selected. Of course, a RAS or a LES model can also be used with the appropriate modifications. The `turbulenceProperties` file should look like this:

fluid/constant/turbulenceProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType  laminar;
```

Some additions also need to be made in the `system/fvSchemes` file. The `divSchemes` should include the `div((nuEff*dev(grad(U).T()))` and the `laplacianSchemes` should include the `laplacian(nuEff,U)` as well. While the rest of the file remains the same, the `divSchemes` and `laplacian(nuEff,U)` should be like

that:

fluid/system/fvSchemes

```
divSchemes
{
    default         none;
    div(phi,U)   Gauss linear;
    div((nuEff*dev(grad(U).T()))) Gauss linear;
}

laplacianSchemes
{
    default         none;
    laplacian(nu,U) Gauss linear skewCorrected 1;
    laplacian((1|A(U)),p) Gauss linear skewCorrected 1;
    laplacian(diffusivity,cellMotionU) Gauss linear skewCorrected 1;
    laplacian(nuEff,U) Gauss linear skewCorrected 1;
}
```

NB: It is out of the scope of this work to suggest which schemes are more appropriate.

The last thing to change is the `system/fvSolution` file in order to include the `pFinal` in the solvers. One can be advised from the existing tutorials to see how the `fvSolution` should be like. In this case though, for the sake of consistency with the other solvers, `pFinal` gets the same properties as `p`:

fluid/system/fvSolution

```
solvers
{
...
 pFinal
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0;
        minIter         1;
        maxIter         1000;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        nFinestSweeps   2;
        scaleCorrection true;
        directSolveCoarsest false;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 20;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
...
}
```

After the previous changes, the `beamInCrossFlow` tutorial can run with `pisoFlow` as the flow model simply by executing `./Allrun` in the fluid directory.

Chapter 4

Building and testing interFlow

In this section, the creation and compilation of a new flow model is demonstrated. The necessary steps for creating a tutorial for two phase flow will be presented at the end of the chapter.

4.1 Introduction of a new flow model

As it was briefly discussed in the last paragraph of Section 3.1, a flow model is compiled independently and it belongs to the `fluidStructureInteraction` library. The source code for any flow model consists only of the *.C and *.H files. The compilation of a new flow model does not affect the existing models.

For this case the `interFlow` flow model will be compiled, but one can use exactly the same process to compile the flow model of his needs. The only thing that needs to be changed is the name. It is important to remember that once the FSI package is compiled, only the `fluidStructureInteraction` library has to be recompiled to include the new flow model and not the whole package. To start with, the new flow model will be the exact copy of an existing one. The names of the source files and the class names will be updated with the new names. The process is showed below:

```
cd fluidStructureInteraction/src/fluidStructureInteraction/flowModels
cp -r pisoFlow interFlow
cd interFlow
mv pisoFlow.C interFlow.C
mv pisoFlow.H interFlow.H
sed -i s/pisoFlow/interFlow/g interFlow.*
```

Up to this point, the source files for the `interFlow` flow model exist. Of course they are identical to the `pisoFlow` flow model. Now the compiler needs the extra information required to include `interFlow` in the flow models. One can manually add the `flowModels/interFlow/interFlow.C` in the `Make/files` of the `fluidStructureInteraction` library or use the terminal command below and compile the library assuming that he is in the `src/fluidStructureInteraction` directory.

```
echo "flowModels/interFlow/interFlow.C" >> Make/files
wmake libso
```

The `fluidStructureInteraction` library is now updated containing the `interFlow` flow model. No changes to the `Make/Options` were required, since the flow model is a copy of an existing one and all the essential paths already exist. In the next section, the the `Make/Options` file will be updated according to the changes in the source code of the flow model.

The reason that `pisoFlow` is selected as a starting point is that it allows turbulence modelling, which will appear to be useful in the next steps of the work.

NB: One can change the name of the library at the bottom of the `Make/files`, however it has not been done

here for two reasons: a) the library is already compiled in a `$FOAM_USER_LIBBIN`, so there is no way to harm the central installation and b) the library already exists and even in the case that the new flow models fail to compile the existing can still work. One can change the name of the library, but then he needs to include the name of the new library in other parts of the FSI package, for example the `Make/Options` file of the `fsiFoam` solver. For convenience the original name of the library is kept here.

4.2 Introduction of the two phase fluid flow

Probably, the best way to start working on the introduction of two phase flow in the existing flow model, is to find an existing two phase flow solver. Having a look at `$FOAM_SOLVERS/multiphase` there is a list of different solvers. The `twoPhaseEulerFoam` and the `interFoam` solver appear to be the most obvious choices.

The `interFoam` solver is selected as a well tested solver for free surface modelling. More precisely the `interDyMFoam` solver, which is similar to the `interFoam`, but it also allows mesh motion that is necessary for the FSI problem. According to its description, `interDyMFoam` is a solver for 2 incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach, with optional mesh motion and mesh topology changes including adaptive re-meshing.

4.2.1 Comparison of the source code of interDyMFoam and interFlow

In this section, the changes required to the source code of the recently generated `interFlow` flow model will be presented. For the sake of clarity, when reference is done to the `interFlow` code, it will correspond to the current version of `interFlow` and not its final functional one. It is reminded that at this stage, `interFlow` is just a copy of `pisoFlow`.

The first thing to notice when comparing the directory of the source code of a flow model (`interFlow`) and a flow solver (`interDyMFoam`) is that the flow model has only the *.C and *.H files, while the solver has many more *.H files and its own `Make` directory, since it compiles independently to an executable. The strategy that is going to be followed is to merge all the necessary code contained in the *.H files of the flow solver to the only two files of the flow model, i.e.`interFlow.C` and `interFlow.H`. The *.H files contained in the `interDyMFoam` source code directory are listed in Table 4.1. It is also important to notice that there are a few *.H files that are used from the `interFoam` source code directory. The latter are in *italics*.

Table 4.1: Source files of interDyMFoam and interFlow

interDyMFoam	**interFlow**
interDyMFoam.C	interFlow.C
correctPhi.H	interFlow.H
createFields.H	
pEqn.H	
readControls.H	
alphaEqnSubCycle.H	
UEqn.H	

It is important to mention that `alphaEqn.H` is not included directly in the `interDyMFoam` source code, but it is included in the `alphaEqnSubCycle.H` file and therefore it should not be omitted.

As mentioned before `interFlow` is a class and because of that, it cannot include any other source files after the `namespace` in the class declaration, static data members, constructors and member functions. That means that the necessary content of the *.H and *.C files should be copied into the `interFlow.C` and `interFlow.H` files. Another reason that one cannot directly use the existing code of `interDyMFoam` is that many variables have different names in the FSI package. This will be shown later.

The next thing to take into account is the additions required in the `fluidStructureInteraction/Make/options` file in order to include all the necessary paths that the two phase flow model will need for compiling. The corresponding `Make/options` file of the `interDyMFoam` solver is presented bellow. The files in blue are those that are not initially listed in `fluidStructureInteraction/Make/options`. These files need to be added.

The same applies for the libraries.

fluidStructureInteraction/Make/options

```
EXE_INC = \
    -I../interFoam \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels/interfaceProperties/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/incompressible/turbulenceModel \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/dynamicFvMesh/lnInclude

EXE_LIBS = \
    -linterfaceProperties \
    -lincompressibleTransportModels \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lfiniteVolume \
    -ldynamicMesh \
    -lmeshTools \
    -ldynamicFvMesh \
    -ltopoChangerFvMesh \
    -llduSolvers \
    -L$(MESQUITE_LIB_DIR) -lmesquite
```

One should be careful with the use of \in the `Make/options` file.

4.2.2 Modifications in the interFlow.C interFlow.H file

The `interFlow.C` and `interFlow.H` files are quite lengthy and for this reason their presentation is going to be split in two parts: the constructors and the member functions.

4.2.2.1 Modifications in the constructors

To begin with, two extra files have to be included at the beginning of `interFlow.C` after the `#include "fixedGradientFvPatchFields.H"` line:

```
#include "interfaceProperties.H"
#include "twoPhaseMixture.H"
```

The `createFields.H` of the `interDyMFoam` contains all the necessary fields that need to be read for running the solver. These fields are: `pd, alpha1, U, phi, rho, rho*phi and p`. In fact, `phi` is not directly contained in `createFields.H`, but it is contained in the `createPhi.H` file, which is included in `createFields.H`. This file is located at `$FOAM_SRC/finiteVolume/lninclude` and the `phi` declared there is similar to the one already existing in `interFlow`.

There are some minor changes that are required to the way the fields are defined in order to conform with that of the `interFlow.C`:

- In `createFields.H` the fields are separated by ";", while in `interFlow.C` this is done with ","
- The line "`Info<< "Reading field alpha1\n" << endl;`" has to be omitted
- The "`runTime.timeName()`" has to be changed to "`runTime().timeName()`" for every occasion
- The variables of the fields (`U, rho, p` etc) should include an "_" at the end: `U_, rho_, p_` etc
- The names of the fields (`volScalarField, volVectorField, surfaceScalarField`) have to be declared separately in the `interFlow.H` file

The names and types of the fields have to be consistent between `interFlow.C` and `interFlow.H`. Moreover, they must have the same order in the two files. It is recommended to add the fields one by one and to compile the `fluidStructureInteraction` library after every change. Any errors can be seen in a log file if the "`wmake libso > make.log 2>&1`" is executed for compiling.

It is straight forward to include the fields `pd_` and `alpha1_`. Regarding the field `phi`, the only difference is in the interpolation of the velocity. In `createPhi.H` the `linearInterpolate(U) & mesh.Sf()` is used, while in `interFlow.C` the `fvc::interpolate(U_) & mesh.Sf()` is used. Therefore, `phi` is left as it is originally in `interFlow.C`. The code of `createPhi.H` is presented below. It can be seen that it includes `continuityErrs.H`. However, no further changes are needed, because this code already exists in `interFlow.C` (see Section 3.2.2).

correctPhi.C

```
{
#   include "continuityErrs.H"
    volScalarField pcorr
    (
        IOobject
        (
            "pcorr",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar("pcorr", pd.dimensions(), 0.0),
        pcorrTypes
    );
    dimensionedScalar rAUf
    (
        "(1|A(U))",
        dimTime/rho.dimensions(),
        runTime.deltaT().value()
    );
    phi = (fvc::interpolate(U) & mesh.Sf());
    adjustPhi(phi, U, pcorr);
    for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pcorrEqn
        (
            fvm::laplacian(rAUf, pcorr) == fvc::div(phi)
        );
        pcorrEqn.setReference(pdRefCell, pdRefValue);
        pcorrEqn.solve();
        if (nonOrth == nNonOrthCorr)
        {
            phi -= pcorrEqn.flux();
        }
    }
#   include "continuityErrs.H"
#   include "CourantNo.H"
    // Recalculate rhoPhi from rho
    rhoPhi = fvc::interpolate(rho)*phi;
}
```

After the additions the constructors in `interFlow.C` should look like:

constructors of interFlow.C

```
#include "interFlow.H"
#include "volFields.H"
#include "fvm.H"
#include "fvc.H"
#include "fvMatrices.H"
#include "addToRunTimeSelectionTable.H"
#include "findRefCell.H"
#include "adjustPhi.H"
#include "fluidStructureInterface.H"
#include "fixedGradientFvPatchFields.H"
#include "interfaceProperties.H"
#include "twoPhaseMixture.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace flowModels
{
// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
defineTypeNameAndDebug(interFlow, 0);
addToRunTimeSelectionTable(flowModel, interFlow, dictionary);
// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //
interFlow::interFlow(const fvMesh& mesh)
:
    flowModel(this->typeName, mesh),
    U_
    (
        IOobject
        (
            "U",
            runTime().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    ),
    pd_
    (
        IOobject
        (
            "pd",
            runTime().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    ),
    p_
    (
        IOobject
        (
            "p",
            runTime().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
```

```
        mesh
    ),
    gradp_(fvc::grad(p_)),
    alpha1_
    (
        IOobject
        (
            "alpha1",
            runTime().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    ),
    phi_
    (
        IOobject
        (
            "phi",
            runTime().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        fvc::interpolate(U_) & mesh.Sf()
    ),
    laminarTransport_(U_, phi_),
    turbulence_
    (
        incompressible::turbulenceModel::New
        (
            U_, phi_, laminarTransport_
        )
    ),
    rho_
    (
        IOdictionary
        (
            IOobject
            (
                "transportProperties",
                runTime().constant(),
                mesh,
                IOobject::MUST_READ,
                IOobject::NO_WRITE
            )
        ).lookup("rho")
    )
{}
```

After the additions the private data in `interFlow.H` should look like:

constructors of interFlow.H

```
class interFlow
:
    public flowModel
{
```

```
    // Private data
        //- Velocity field
        volVectorField U_;
        //- Pressure field
        volScalarField pd_;
        //- Pressure field
        volScalarField p_;
        //- Pressure field
        volVectorField gradp_;
        //- Phase field
        volScalarField alpha1_;
        //- Flux field
        surfaceScalarField phi_;
        //- Transport model
        singlePhaseTransportModel laminarTransport_;
        //- Turbulence model
        autoPtr<incompressible::turbulenceModel> turbulence_;
        //- Density
        dimensionedScalar rho_;
```

4.2.2.2 Modifications in the member functions

If one looks at the source code of `interDyMFoam`, he will notice that there are some files included in the solution process. As it was demonstrated for `pisoFlow` in Section 3.2.2, these files cannot be included to the source code of the flow model, but their content should be copied in the whole and the names of the variables used need to be changed as well. The files that have to be added from `interDyMFoam` are `pEqn.H, readControls.H, alphaEqnSubCycle.H, UEqn.H` and `alphaEqn.H`. Their content is presented below:

`pEqn.H`

```
{
    volScalarField rAU = 1.0/UEqn.A();
    surfaceScalarField rAUf = fvc::interpolate(rAU);
    U = rAU*UEqn.H();
    surfaceScalarField phiU("phiU", (fvc::interpolate(U) & mesh.Sf()));
    if (pd.needReference())
    {
        adjustPhi(phi, U, pd);
    }
    phi = phiU +
    (
        fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
      - ghf*fvc::snGrad(rho)
    )*rAUf*mesh.magSf();
    for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pdEqn
        (
            fvm::laplacian(rAUf, pd) == fvc::div(phi)
        );
        pdEqn.setReference(pdRefCell, pdRefValue);
        if (corr == nCorr - 1 && nonOrth == nNonOrthCorr)
        {
            pdEqn.solve(mesh.solutionDict().solver(pd.name() + "Final"));
        }
        else
        {
            pdEqn.solve(mesh.solutionDict().solver(pd.name()));
```

```
        }
        if (nonOrth == nNonOrthCorr)
        {
            phi -= pdEqn.flux();
        }
    }
    U += rAU*fvc::reconstruct((phi - phiU)/rAUf);
    U.correctBoundaryConditions();
    #include "continuityErrs.H"
    // Make the fluxes relative to the mesh motion
    fvc::makeRelative(phi, U);
}
```

As it can be seen, `continuityErrs.H` is included in `pEqn.H`. The source code of `continuityErrs.H` is already contained in `interFlow.C` as it can be seen from the comparison between `pisoFlow.C` and `pisFoam.C` in Section 3.2.2.

`readControls.H`

```
#   include "readTimeControls.H"
#   include "readPIMPLEControls.H"
    bool correctPhi = true;
    if (pimple.found("correctPhi"))
    {
        correctPhi = Switch(pimple.lookup("correctPhi"));
    }
    bool checkMeshCourantNo = false;
    if (pimple.found("checkMeshCourantNo"))
    {
        checkMeshCourantNo = Switch(pimple.lookup("checkMeshCourantNo"));
    }
```

As it can be seen, `readTimeControls.H` is included in `readControls.H`, which defines the controls for the adjustable time step. The code that has to be included can be found in `$FOAM_SRC/finiteVolume/lnInclude`.

The same applies for `readPIMPLEControls.H`, which has the controls for the PIMPLE algorithm. In Section 3.2.2 PISO algorithm was employed, therefore the necessary changes have to be made to `interFlow.C`. The source code for `readPIMPLEControls.H` can be found in `$FOAM_SRC/finiteVolume/lnInclude` directory.

`alphaEqnSubCycle.H`

```
label nAlphaCorr
(
    readLabel(pimple.lookup("nAlphaCorr"))
);
label nAlphaSubCycles
(
    readLabel(pimple.lookup("nAlphaSubCycles"))
);
if (nAlphaSubCycles > 1)
{
    dimensionedScalar totalDeltaT = runTime.deltaT();
    surfaceScalarField rhoPhiSum = 0.0*rhoPhi;
    for
    (
        subCycle<volScalarField> alphaSubCycle(alpha1, nAlphaSubCycles);
        !(++alphaSubCycle).end();
    )
    {
#       include "alphaEqn.H"
```

```
        rhoPhiSum += (runTime.deltaT()/totalDeltaT)*rhoPhi;
    }
    rhoPhi = rhoPhiSum;
}
else
{
#       include "alphaEqn.H"
}
interface.correct();
rho == alpha1*rho1 + (scalar(1) - alpha1)*rho2;
```

As it can be seen, `alphaEqn.H` is included in `alphaEqnSubCycle.H` (see Table 4.1 and comments). The source to be added can be found in `$FOAM_APP/solvers/multiphase/interFoam` and reads as follows:

`alphaEqn.H`

```
{
    word alphaScheme("div(phi,alpha)");
    word alpharScheme("div(phirb,alpha)");
    surfaceScalarField phic = mag(phi/mesh.magSf());
    phic = min(interface.cAlpha()*phic, max(phic));
    surfaceScalarField phir = phic*interface.nHatf();
    for (int aCorr=0; aCorr<nAlphaCorr; aCorr++)
    {
        surfaceScalarField phiAlpha =
            fvc::flux
            (
                phi,
                alpha1,
                alphaScheme
            )
          + fvc::flux
            (
                -fvc::flux(-phir, scalar(1) - alpha1, alpharScheme),
                alpha1,
                alpharScheme
            );
        MULES::explicitSolve(alpha1, phi, phiAlpha, 1, 0);
        rhoPhi = phiAlpha*(rho1 - rho2) + phi*rho2;
    }
    Info<< "Liquid phase volume fraction = "
        << alpha1.weightedAverage(mesh.V()).value()
        << "  Min(alpha1) = " << min(alpha1).value()
        << "  Max(alpha1) = " << max(alpha1).value()
        << endl;
}
```

`alphaEqn.H` does not have any extra inclusions and after the necessary changes. Its source code can be included in the `interFlow.C` file.

`UEqn.H`

```
    surfaceScalarField muEff
    (
        "muEff",
        twoPhaseProperties.muf()
      + fvc::interpolate(rho*turbulence->nut())
    );
    fvVectorMatrix UEqn
    (
```

```
        fvm::ddt(rho, U)
      + fvm::div(rhoPhi, U)
      - fvm::laplacian(muEff, U)
      - (fvc::grad(U) & fvc::grad(muEff))
    //- fvc::div(muEff*(fvc::interpolate(dev(fvc::grad(U))) & mesh.Sf()))
    );
    UEqn.relax();
    if (momentumPredictor)
    {
        solve
        (
            UEqn
         ==
            fvc::reconstruct
            (
                (
                    fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
                  - ghf*fvc::snGrad(rho)
                  - fvc::snGrad(pd)
                ) * mesh.magSf()
            )
        );
    }
```

As it can be seen from the source file `UEqn.H`, the expression for `U` is more complicated for a two phase fluid simulation than it is for a single fluid (see `pisoFlow.C` in Section 3.2.2). This means that the code above has to be used.

After including the source code of the files mentioned in this section in the `interFlow.C` and `interFlow.H`, the `interFlow` flow model should be able to handle two phase flow simulations. Once the `fluidStructureInteraction` library is recompiled, a tutorial is prepared to test `interFlow` as presented in the next section.

4.3 Preparation of the interFlow tutorial

In this chapter, the necessary changes required in the `beamInCrossFlow` tutorial in order to make it run with `interFlow` will be presented, as it was done for `icoFlow` and `pisoFlow` (see Section 3.3).

In order to run the `beamInCrossFlow` tutorial with the `interFlow` flow model, the tutorial will be copied and saved with a new name:

```
cp -r beamInCrossFlow beamInCrossFlow_interFlow
```

The rest of the work in this chapter will be done in the `beamInCrossFlow_interFlow/fluid` directory, since no changes are going to be applied to the solid.

As it was mentioned in the introduction, the tutorial for `interFlow` will be based on the `damBreakWithObstacle` tutorial that can be found in `$FOAM_TUTORIALS/multiphase/interDyMFoam/ras/`.

4.3.1 Introducing the fields

The computational mesh will essentially remain the same as in the original `beamInCrossFlow` tutorial, however the two different fields, namely the water and the air, have to be defined. The two fields are defined in the `setFieldsDict` dictionary located in the `system` directory. This file does not exist and it suggested to be copied from the `damBreakWithObstacle` tutorial:

```
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/system/setFieldsDict
fluid/system
```

The `setFieldsDict` file has to be modified so that the region is in the computational mesh and in front of the flexible beam. The modified file should look like this:

`setFieldsDict`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      setFieldsDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
defaultFieldValues
(
    volScalarFieldValue alpha1 0
    volVectorFieldValue U ( 0 0 0 )
);
regions
(
    boxToCell
    {
        box ( 0 0 -0.4 ) ( 0.3 0.4 0.0 );
        fieldValues (volScalarFieldValue alpha1 1);
    }
 );
```

One should also add the command "`runApplication setFields`" in the `Allrun` script before the execution of `fsiFoam` (`runApplication $application`), in order to set the fields. However, before executing the `setFields` command, one should make some changes in the `0` directory (see Section 4.3.2).

4.3.2 Modifications in the boundary conditions

Here, two are the main changes that need to be made. The first one is of course about inserting the two different fields in the initial conditions. The second one is about the top wall boundary condition that it is recommended to change to an atmospheric boundary condition.

Starting from the second change, an atmospheric boundary condition is set at the top patch to allow water to leave the domain and to make air able to leave and enter the domain through the top part. This is done because when the dam breaks and splashes, there is common to observe some quantity of water leaving the domain and be replaced with air. The boundary conditions for the top patch are inherited from `damBreakWithObstacle` tutorial. All the patch names remain the same though, as defined in `beamInCrossFlow` tutorial.

The top patch in `0/U` should be modified from `fixed value` to `pressureInletOutletVelocity` as follows:

`0/U`

```
    top
    {
        type            pressureInletOutletVelocity;
        phi             phi;
        value           uniform (0 0 0);
    }
```

The `pressureInletOutletVelocity` boundary condition is a combination of `pressureInletVelocity` and `inletOutlet`, which means that when `p` is is known at the inlet, `U` is evaluated from the flux, normal to the patch and `U` and `p` can be switched automatically from `fixedValue` to `zeroGradient` and vice versa depending on the direction of `U` respectively [6].

The fields are defined in the initial conditions through the `0/alpha1` file in the `0` directory. The file can be copied from the `damBreakWithObstacle` tutorial, but requires further modifications to include all the patches of the `beamInCrossFlow` tutorial.

```
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/0.org/alpha1 fluid/0
```

The `0/alpha1` fields should read:

`fluid/0/alpha1`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      alpha1;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 0 0 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    interface
    {
        type            zeroGradient;
    }
    outlet
    {
        type            zeroGradient;
    }
    inlet
    {
        type            zeroGradient;
    }
    bottom
    {
        type            zeroGradient;
    }
    top
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
    left
    {
        type            zeroGradient;
    }
    symmetry
    {
        type            symmetryPlane;
    }
}
```

Another noticeable difference between the `0` diretories of `damBreakWithObstacle` and `beamInCrossFlow` tutorials is that in the first `pd` can be found, instead of `p` in the second. This is not necessarily a problem, because there are different constructors for the pressure for `interDyMFoam` and `interFlow`. Of course, if one wants `pd` to be read, appropriate changes are required in the source code, fvSolution and fvSchemes.

After these changes, one should be able to see the fields `beamInCrossFlow` tutorial, as shown in Figure 4.1. The mesh is coloured according to `alpha1` field at `Time=0`. The water column is in red colour. The solid in the figure is transparent.

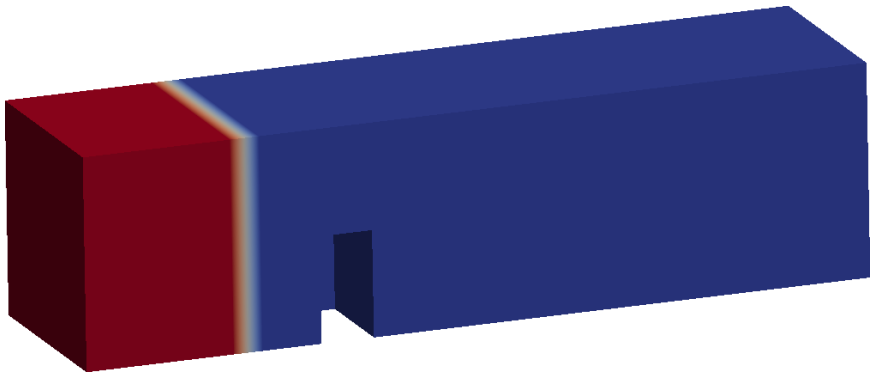


Figure 4.1: Perspective view of the mesh of the beamInCrossFlow tutorial with the two phases.

4.3.3 Modifications in the constant directory

There are various things that need to be added or modified in the constant directory. To get an idea of the files that are required one should look at `damBreakWithObstacle` tutorial.

The first thing to do is to tell the solver which flow model to use and its coefficients. This is done in the `flowProperties file`. For the sake of simplicity, the same coefficients as before (see Section 3.3.1).

`fluid/constant/flowProperties`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      flowProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
flowModel interFlow;
interFlowCoeffs
{
    nCorrectors 3;
    nNonOrthogonalCorrectors 1;
}
```

The next thing that needs to be done is the definition of the gravitational acceleration. This is done in the `g` file, which can be directly copied from the `damBreakWithObstacle` tutorial:

```
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/constant/g
fluid/constant
```

The `g` file should look like this, assuming that $g = 9.81 m/s^2$:

`fluid/constant/g`

```
FoamFile
{
    version     2.0;
    format      ascii;
```

```
    class       uniformDimensionedVectorField;
    location    "constant";
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 1 -2 0 0 0 0];
value           ( 0 0 -9.81 );
```

An important modification is required in the `transportProperties` dictionary. Now, there are two fluid phases and their transport properties, i.e. the transport model, the cross power law coefficients and the bird Carreau coefficients, should be declared separately. For simplicity the `transportProperties` can be copied from the `damBreakWithObstacle` tutorial.

```
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/constant/transportPro
perties fluid/constant
```

The `transportProperties` file with the two phases should look like this:

`fluid/constant/transportProperties`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
phase1
{
    transportModel  Newtonian;
    nu              nu [ 0 2 -1 0 0 0 0 ] 1e-06;
    rho             rho [ 1 -3 0 0 0 0 0 ] 1000;
    CrossPowerLawCoeffs
    {
        nu0             nu0 [ 0 2 -1 0 0 0 0 ] 1e-06;
        nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
        m               m [ 0 0 1 0 0 0 0 ] 1;
        n               n [ 0 0 0 0 0 0 0 ] 0;
    }
    BirdCarreauCoeffs
    {
        nu0             nu0 [ 0 2 -1 0 0 0 0 ] 0.0142515;
        nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
        k               k [ 0 0 1 0 0 0 0 ] 99.6;
        n               n [ 0 0 0 0 0 0 0 ] 0.1003;
    }
}
phase2
{
    transportModel  Newtonian;
    nu              nu [ 0 2 -1 0 0 0 0 ] 1.48e-05;
    rho             rho [ 1 -3 0 0 0 0 0 ] 1;
    CrossPowerLawCoeffs
    {
        nu0             nu0 [ 0 2 -1 0 0 0 0 ] 1e-06;
        nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
        m               m [ 0 0 1 0 0 0 0 ] 1;
```

```
        n               n [ 0 0 0 0 0 0 0 ] 0;
    }
    BirdCarreauCoeffs
    {
        nu0             nu0 [ 0 2 -1 0 0 0 0 ] 0.0142515;
        nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
        k               k [ 0 0 1 0 0 0 0 ] 99.6;
        n               n [ 0 0 0 0 0 0 0 ] 0.1003;
    }
}

sigma           sigma [ 1 0 -2 0 0 0 0 ] 0.07;
```

It is easy to see from the density (`rho`) that `phase1` corresponds to water and `phase2` to air. Note that the kinematic viscosity (`nu`) and `nu0` of the fluid have different values from the original `beamInCrossFlow` tutorial.
NB: The `CrossPowerLawCoeffs` and `BirdCarreauCoeffs` are only used when non-Newtonian transport models are selected.

The turbulence properties should also be defined in the `turbulenceProperties` file. For simplicity, the simulation is `laminar` and the `fluid/constant/turbulenceProperties` dictionary is identical to the one defined for `pisoFlow` (see Section 3.3.2).

Another file that should be added in the `constant` directory is the `RASProperties` dictionary. This can be copied directly from the `damBreakWithObstacle` tutorial and the RAS model is set to laminar for simplicity.

```
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/constant/RASProperties
fluid/constant
```

`fluid/constant/RASProperties`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
RASModel        laminar;
turbulence      on;
printCoeffs     off;
```

The `constant/dynamicMeshDict` remains the same as in the original `beamInCrossFlow` tutorial.

4.3.4 Modifications in the system directory

The first addition to the `fluid/system` directory is the `setFieldsDict` dictionary, as it was discussed at the beginning of the current chapter (see 4.3.1). The other files that need to be changed in the `system` directory are the `controlDict`, `fvSchemes` and `fvSolution`.

The changes suggested for the `controlDict` refer to the application of an adjustable time step. Working experience with the `interFoam` solver indicates that if fixed time step is selected, the simulation is likely to become unstable. A way to fix this is to control the time step through the Courant–Friedrichs–Lewy condition. The switch `adjustTimeStep` should be turned to `yes` and `maxCo` has to be defined. The time step might become very small and therefore it is recommended to change the `writeControl` from `timeStep` to `adjustableRunTime`. This will also result to have the same output interval. Last but not least, `deltaT`, which is now an indicative value for the first time step, should be changed to a lower value that $0.1s$ and

maxDeltaT should be defined as the highest possible value for the time step. After the previous modifications, the controlDict should look like:

fluid/system/controlDict

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
application     fsiFoam;
startFrom       latestTime;
startTime       0;
stopAt          endTime;
endTime         4;
deltaT          0.001;
writeControl    adjustableRunTime;
writeInterval   0.02;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression uncompressed;
timeFormat      general;
timePrecision   6;
adjustTimeStep  yes;
maxCo           0.1;
maxDeltaT       1;
functions
( ...
```

The next file to modify is fvSolution. The solvers for pcorr, pFinal, k, B and nuTilda should be included, together with the PISO and PIMPLE controls for the pressure–velocity coupling. It is important to note again that the interDyMFoam tutorial uses pd instead of p. Depending on the constructors of the new flow model, either p or pd is required to be read. For simplicity, the solvers and the PISO controls are identical to those used in the damBreakWithObstacle tutorial. After the proposed modifications, fvSolution should be like:

fluid/system/fvSolution

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
solvers
{
    pcorr
    {
        solver          PCG;
        preconditioner
        {
            preconditioner  GAMG;
            tolerance       0.001;
            relTol          0;
```

```
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        nBottomSweeps   2;
        cacheAgglomeration false;
        nCellsInCoarsestLevel 10;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
    tolerance       0.0001;
    relTol          0;
    maxIter         100;
}
p
{
    solver          GAMG;
    tolerance       1e-06;
    relTol          0;
    minIter         1;
    maxIter         1000;
    smoother        GaussSeidel;
    nPreSweeps      0;
    nPostSweeps     2;
    nFinestSweeps   2;
    scaleCorrection true;
    directSolveCoarsest false;
    cacheAgglomeration true;
    nCellsInCoarsestLevel 20;
    agglomerator    faceAreaPair;
    mergeLevels     1;
}
pFinal
{
    solver          PCG;
    preconditioner
    {
        preconditioner  GAMG;
        tolerance       1e-08;
        relTol          0;
        nVcycles        2;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        nFinestSweeps   2;
        cacheAgglomeration false;
        nCellsInCoarsestLevel 10;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
    tolerance       1e-08;
    relTol          0;
    maxIter         20;
}
cellMotionU
{
    solver          GAMG;
    tolerance       1e-06;
    relTol          0.001;
    minIter         1;
```

```
        maxIter         100;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        nFinestSweeps   2;
        scaleCorrection true;
        directSolveCoarsest false;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 20;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
        minIter         1;
    }
    k
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
    }
    B
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
    }
    nuTilda
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
    }
}
PISO
{
    cAlpha          1;
}
PIMPLE
{
    momentumPredictor yes;
    nOuterCorrectors 1;
    nCorrectors     4;
    nNonOrthogonalCorrectors 0;
    nAlphaCorr      1;
    nAlphaSubCycles 3;
    cAlpha          1;
    pRefPoint      (0.51 0.51 0.51);
    pRefValue      0;
}
```

The last file to modify is `fvSchemes`. Once again, a comparison between the two tutorials (using the program `kompare`) will reveal the additions required. The main change is that the field `alpha1` has been introduced, together with `rho*phi` (see the new constructors at Section 4.2.2.1). This affects the `gradSchemes`, `divSchemes` and the `fluxRequired`. After the necessary modifications the `fvSchemes` dictionary should read in whole:

`fluid/system/fvSchemes`

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
ddtSchemes
{
    default Euler;
}
gradSchemes
{
    default         Gauss linear;
    grad(U)         Gauss linear;
    grad(alpha)     Gauss linear;
}
divSchemes
{
    default           none;
    div(phi,U)        Gauss linear;
    div(rho*phi,U)    Gauss upwind;
    div(phi,alpha)    Gauss vanLeer;
    div(phirb,alpha)  Gauss interfaceCompression;
}
laplacianSchemes
{
    default         none;
    laplacian(nu,U) Gauss linear skewCorrected 1;
    laplacian((1|A(U)),p) Gauss linear skewCorrected 1;
    laplacian(diffusivity,cellMotionU) Gauss linear skewCorrected 1;
}
interpolationSchemes
{
    default linear;
    interpolate(y) linear;
    interpolate(U) skewCorrected linear;
}
snGradSchemes
{
    default         skewCorrected 1;
}
fluxRequired
{
    default         no;
    p;
    pcorr;
    alpha;
}
```

Of course, if `pd` is used instead of `p`, appropriate changes should be made.

Chapter 5

Conclusions and future work

This work showed a detailed presentation of the new FSI package created for `foam-extend-3.1`. As it has been discussed, the FSI package is still under development and it is expected to be included in main distribution of `foam-extend`. In this case, there is a chance that substantial changes to the structure of the package will take place. Therefore, any future users of the FSI package that want to consult this document should be aware of the changes made after the present date.

Future work should be carried out to complete the modifications required in the `interFlow` package to make it functional. Future tutorials should include other classic problems like free surface current and wave loading on a beam. This requires implementation of new boundary conditions in the FSI package that will expand its applicability. Boundary conditions for waves and currents can be found in the packages `waves2Foam` [4] and `IHFOAM` [2].

Bibliography

[1] Robert L. Campbell (2010): Fluid-Structure Interaction and inverse design simulations for flexible turbomachinery, PhD Thesis, Pennsylvania State University, College of Engineering

[2] Pablo Higuera and Javier L. Lara and Inigo J. Losada (2013): Realistic wave generation and active wave absorption for Navier Stokes models: Application to OpenFOAM, Coastal Engineering, Volume 71, Pages 102 - 118

[3] Hua-Dong Yao: Simulation of Fluid-Structural Interaction using OpenFOAM, Presentation (2014) `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/OFLecFSI-1.pdf`

[4] Jacobsen, N G and Fuhrman, D R and Fredse, J (2014): A Wave Generation Toolbox for the Open-Source CFD Library: OpenFoam®, Int. J. Numerl. Meth. Fluids, Volume 70, Pages 1073-1088

[5] C. Michler, S.J. Hulshoff, E.H. van Brummelen, R. de Borst (2003): A monolithic approach to fluid-structure interaction, Preprint submitted to Elsevier Science, 25 March 2003

[6] OpenFOAM Foundation (2012): OpenFOAM, The Open Source CFD Toolbox User Guide, Version 2.1.1, 16 May 2012

[7] Željko Tuković: FVM for FSI with large structural displacements, Presentation (2009) `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2009/FSIslides.pdf`