Cite as: Minghao Li.: Implement interFoam as a fluid solver in FSI package. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H.,
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Implement interFoam as a fluid solver in the FSI package

Developed for foam-extented-4.0
Requires: FSI-package

Author:
Minghao Li
Chalmers University of Technology
lmhao1014@hotmail.com

Peer reviewed by:
Yuzhu Li
Håkan Nilsson

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

October 5, 2017

Learning outcomes

This tutorial will mainly teach four aspects: How to use it, The theory of it, How to implement it, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- How to solve a fluid solid interaction problem with the FSI package.
- How the solver folder, library folder and case folder are organized.

The theory of it:

- The basic principle of solving fluid solid interaction problems.

How it is implemented:

- The difference between a top level solver and a class.
- The connection between single fluid or solid model and the fluid solid interaction (fsiFoam) solver.

How to modify it:

- How to add a new fluid solver i.e. interFluid to the fluid models.
- How to setup a case to verify the implemented fluid solver.

Contents

Chapter 1

Introduction

This tutorial aims to provide a thorough interpretation of the structures and connections of Fluid-Solid-Interaction FSI package based on `foam-extend-4.0` plus a detailed procedure of introducing a new interFluid (fluid solver) class. Similar as class inheritance, this project can be regarded as inherited, or a continuation of previous work. A detailed introduction to the FSI package has been presented in the project report "The implementation of interFoam solver as a flow model of the fsiFoam solver for strong fluid-structure interaction" of Thomas Vyzikas in 2014 [1]. Therefore what leaves to the present work is mainly the connections of how the fsiFoam solver recalls the fluid or solid solvers and how the variables are transferred between the fluid and solid sides.

In addition, a comparison of the FSI package structure between two versions i.e.`foam-extend-3.2` and `4.0` will be carried out to see the difference. The most important modification to turn interFoam to interFluid will be described in details including: 'Make' directory modification, constructor construction and member function declarations and definitions. The validation of this interFluid fluid solver is based on the tutorial case `BeamInCrossFlow` with the corresponding adjustment for the interFluid solver.

The target tutorial case is illustrated in Figure 1.1.

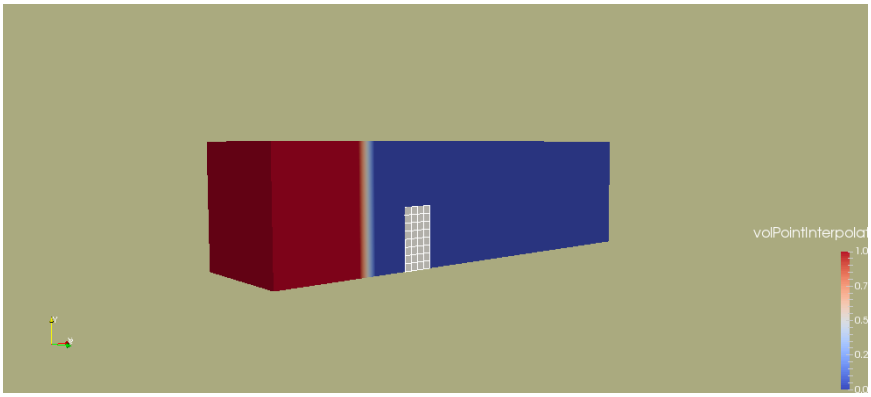

Figure 1.1: FSI with interFluid

Chapter 2

Structure of the FSI package

2.1 Overview of the FSI package

The FSI package can be easily downloaded and installed on `foam-extend-4.0` by

```
f40NR
run
wget https://openfoamwiki.net/images/d/d6/Fsi_40.tar.gz
tar -xzf Fsi_40.tar.gz

cd FluidSolidInteraction/src/
./Allwmake
```

After unpacking, a folder named "FluidSolidInteraction" will be generated which contains two main sub-directories i.e. *run* directory for the tutorial cases and *src* directory for the source files.

Since the previous work is based on `foam-extend-3.1`, we first start by a short files comparison between these two versions. The corresponding FSI package of `foam-extend-3.1` is downloaded by the following commands and the files structure is listed below.

```
f40NR
run
wget http://openfoamwiki.net/images/5/52/Fsi_31.tar.gz
tar zxvf Fsi_31.tar.gz
```

```
foam-extend-4.0/FluidSolidInteraction/src:
|-- fluidSolidInteraction
|--  |-- fluidSolidInterface
|--  |-- fluidSolvers
|--  |--  |-- consistentIcoFluid
|--  |--  |-- finiteVolume
|--  |--  |-- fluidSolver
|--  |--  |-- fvPatchFields
|--  |--  |-- icoFluid
|--  |--  |-- pisoFluid
|--  |-- include
|--  |-- lnInclude
|--  |-- Make
|--  |-- solidSolvers
|--      |-- finiteVolume
|--      |-- solidModels
```

```
foam-extend-3.1/FluidSolidInteraction/src:
 |-- fluidStructureInteraction
 |--  |-- fluidStructureInterface
 |--  |-- flowModels
 |--  |--  |-- consistentIcoFlow
 |--  |--  |-- flowModel
 |--  |--  |-- fvPatchFields
 |--  |--  |-- icoFlow
 |--  |--  |-- pisoFlow
 |--  |-- lnInclude
 |--  |-- Make
 |--  |-- numerics
 |--  |--  |-- backwardD2dt2Scheme
 |--  |--  |-- ddtSchemes
 |--  |--  |-- findRefCell
 |--  |--  |-- fvc
```

```
|--      |-- solidSolver
|--      |-- unsIncrTotalLagrangianSolid
|--      |-- unsTotalLagrangianSolid
|-- solvers
|--  |-- ampFsiFoam
|--  |-- fluidFoam
|--  |-- fsiFoam
|--  |-- solidFoam
|--  |-- thermalSolidFoam
|--  |-- weakFsiFoam
|-- ThirdParty
|--  |-- eigen3
|-- utilities
```

```
|--  |--  |-- fvMeshSubset
|--  |--  |-- fvMotionSolvers
|--  |--  |-- ggi
|--  |--  |-- leastSquaresSkewCorrected
|--  |--  |--
↪  leastSquaresVolPointInterpolation
|--  |--  |-- quadraticReconstruction
|--  |--  |-- skewCorrectedSnGrad
|--  |--  |-- skewCorrectedVectorSnGrad
|--  |-- stressModels
|--      |-- componentReference
|--      |-- constitutiveModel
|--      |-- fvPatchFields
|--      |-- materialInterfaces
|--      |--
↪  pRveUnsIncrTotalLagrangianStress
|--      |--
↪  pRveUnsTotalLagrangianStress
|--      |-- simpleCohesiveLaws
|--      |-- solidInterfaces
|--      |-- stressModel
|--      |--
↪  unsIncrTotalLagrangianStress
|--      |-- unsTotalLagrangianStress
|-- solvers
|--  |-- crackStressFoam
|--  |-- flowFoam
|--  |-- fsiFoam
|--  |-- stressFoam
|--  |-- thermalStressFoam
|--  |-- weakFsiFoam
|-- utilities
```

Similar to the OpenFOAM src directory structure, there are solvers, utilities and libraries in the FSI src directory. The *fluidSolidInteraction* directory contains the most fundamental sources to support the solvers e.g. fsiFoam. The *solvers* directory shows the available solvers with the end words "Foam". Comparing these two *src* directories, one can figure out the differences are:

- The name of "flowModels/stressModels" in V3.1 has been changed to "fluidSolvers/solidSolvers" in V4.0, which is more obvious for understanding.
- The "numerics" directory in V3.1 is removed and the contents are spreaded into "finiteVolume" sub-directory under "fluidSolvers/solidSolvers".
- A new directory "include" is created in V4.0 with the purpose of avoiding repetition of identical codes.
- A "ThirdParty" directory is added in V4.0, increasing the flexibility.
- The "crackStressFoam" solver in V3.0 is replaced by "ampFsiFoam" solver in V4.0.
- There are also some updates regarding "numeric" and "stressModels" of V3.0 but they are out of the present work.

2.2 A quick look at the classes

The `Make` directory under `fluidSolidInteraction` indicates that the latter directory is the top level of a library. *Make/files* lists all the classes that will be compiled. Below is only a part of the file.

```
...
fluidSolvers/fluidSolver/fluidSolver.C
fluidSolvers/fluidSolver/newFluidSolver.C
fluidSolvers/icoFluid/icoFluid.C
fluidSolvers/pisoFluid/pisoFluid.C
fluidSolvers/consistentIcoFluid/consistentIcoFluid.C
...
fluidSolidInterface/fluidSolidInterface.C
...
solidSolvers/solidSolver/solidSolver.C
solidSolvers/solidSolver/newSolidSolver.C
solidSolvers/unsTotalLagrangianSolid/unsTotalLagrangianSolid.C
solidSolvers/unsTotalLagrangianSolid/unsTotalLagrangianSolidSolve.C
solidSolvers/unsIncrTotalLagrangianSolid/unsIncrTotalLagrangianSolid.C
solidSolvers/unsIncrTotalLagrangianSolid/unsIncrTotalLagrangianSolidSolve.C

LIB = $(FOAM_USER_LIBBIN)/libfluidSolidInteraction
```

It can be seen that each class is compiled independently and that all the compiled classes are stored in the library named `libfluidSolidInteraction` under `$(FOAM_USER_LIBBIN)`. If a solver needs to link to this library, it is done by adding `EXE_LIBS =-lfluidSolidInteraction \` to the *fsiFoam/Make/options* file. However at this stage, let us take a round look at the basic classes. The fluidSolidInterface class declaration and part of the member data and functions are shown in Table 2.1.

fluidSolidInterface.H	

Header files	`#include "fluidSolver.H"`
	`#include "solidSolver.H"`
	`#include "IOdictionary.H"`
	`#include "patchToPatchInterpolation.H"`
	`#include "dynamicFvMesh.H"`
	`#include "ggiInterpolation.H"`
	`#include "movingWallVelocityFvPatchVectorField.H"`
	`#include "RBFInterpolation.H"`
	`#include "TPSFunction.H"`

Member data	`autoPtr<fluidSolver> flow_;`
	`autoPtr<solidSolver> stress_;`
	`Switch predictor_;`

Member functions	`//- Return fluid solver`
	`fluidSolver& flow() {return flow_();}`
	`//- Return solid solver`
	`solidSolver& stress() {return stress_();}`
	`//- Initialize fields`
	`void initializeFields();`
	`//- Update interface force`
	`void updateForce();`
	`//- Calculate interface displacement`
	`void updateDisplacement();`
	`//- Initialize fields`
	`void initializeFields();`
	`//- Return preditor on/off`
	`const Switch& predictor() const{return predictor_;}`
	`//- Return current outer iteration`
	`label& outerCorr(){return outerCorr_;}`
	`//- Move fluid mesh`
	`void moveFluidMesh();`

Table 2.1: fluidSolidInterface class

The green lines will be discussed further in the following sections. The listed member data and functions will be used to serve for the solver fsiFoam as discussed in section 2.3. Therefore it is good to have a glimpse at this information before we go deeper into the solver.

Since `fluidSolver.H` is included in the fluidSolidInterface class, we move on to take a look at this class and summarize the useful information in Table 2.2.

fluidSolver.H	

Header files	`#include "fvMesh.H"`
	`#include "IOdictionary.H"`
	`#include "autoPtr.H"`
	`#include "runTimeSelectionTables.H"`

Member data	`dictionary fluidProperties_;` `const fvMesh& mesh_;`
Selectors	`static autoPtr<fluidSolver> New(const fvMesh& mesh);`

Member functions	`//- Return flow properties dictionary`
	`const dictionary& fluidProperties() const`
	`{return fluidProperties_;}`
	`//- Return mesh`
	`const fvMesh& mesh() const {return mesh_;}`
	`//- Patch viscous force (N/m2)`
	`virtual tmp<vectorField> patchViscousForce`
	`(const label patchID) const = 0;`
	`//- Patch pressure force (N/m2)`
	`virtual tmp<scalarField> patchPressureForce`
	`(const label patchID ) const = 0;`
	...
	`//- Evolve the fluid solver`
	`virtual void evolve() = 0;`
	`//- Update fields`
	`virtual void updateFields() {}`

Table 2.2: fluidSolver class

This fluidSolver is the base class where the common settings are defined. Some member functions are defined as zero as initialization at this stage. The upcoming sub-classes will update these initialized member functions based on their specific new definitions. One additional thing to mention here is the selector of fluidSolver class, this `New` selector is used to select which fluid solver will be used to solve the fluid side and the definition is found in *newFluidSolver.C*, which is a subfile under the *fluidSolver* directory.

Next we will pick up one fluidSolver, for example the pisoFluid class, to see the details as shown in Table 2.3. A very important feature of the pisoFluid is that it is a class rather than a solver.

pisoFluid.H	
Inheritance	fluidSolver

Header files	`#include "fluidSolver.H"`
	`#include "volFields.H"`
	`#include "surfaceFields.H"`
	`#include "#include "singlePhaseTransportModel.H"`
	`#include "turbulenceModel.H"`

Member data	`volVectorField U_`
	`volScalarField p_`
	`volVectorField gradp_;`
	`volTensorField gradU_;`
	`surfaceScalarField phi_;`
	`singlePhaseTransportModel laminarTransport_;`
	`autoPtr<incompressible::turbulenceModel> turbulence_;`
	`dimensionedScalar rho_;`

Member functions	`//- Return velocity field`
	`volVectorField& U() { return U_; }`
	`//- Return pressure field`
	`volScalarField& p() { return p_; }`
	...
	`//- 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 fluid solver`
	`virtual void evolve();`
	`//- Update fields`
	`virtual void updateFields() {}`

Table 2.3: pisoFluid class

It is important to be aware that the class `pisoFluid` is inherited from the class `fluidSolver`, thus the member functions from the class `fluidSolver` as shown in Table 2.2 are also in the class `pisoFluid`. The class `pisoFluid` is derived from the solver `pisoFoam`, with some similarity and differences that are illustrated as:

- Both the `pisoFluid` class and the `pisoFoam` solver include `singlePhaseTransportModel.H` and `turbulenceModel.H`, because the transport model and the turbulence model need to be specified here.
- The member data in the `pisoFluid` class corresponds to the `createField.H` file under `pisoFoam` but the format is different. An underscore is normally taken in class as a convention for the member data.
- There is no function in the `pisoFoam` solver to calculate the pressure and viscous forces but they exist in the `pisoFluid` class.
- The `evolve()` function in the `pisoFluid` class is acting the same as in the `pisoFoam` solver, but the code is purely written line-by-line without inclusion files due to the class properties.

For a detailed line-by-line code comparison of `pisoFluid` class and `pisoFoam` solver, please refer to section 3.2.2 in Thomas Vyzikas's project "The implementation of interFoam solver as a flow model of the fsiFoam solver for strong fluid-structure interaction" [1].

2.3 An insight of fsiFoam

The next step is to understand how the solver, fsiFoam in this case, is connected with the `fluidSolvers`, `solidSolvers` and `fluidSolidInterface` classes under *fluidSolidInteraction* library. As a starting point, an insight look of the code file `fsiFoam.C` is taken.

```
                                fsiFoam.C
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "fluidSolidInterface.H"

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

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

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

    fluidSolidInterface fsi(mesh, solidMesh);

    Info<< "\nStarting time loop\n" << endl;

    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Time = " << runTime.value()
            << " (dt = " << runTime.deltaT().value() << ")" << nl << endl;

        fsi.initializeFields();

        fsi.updateInterpolator();

        scalar residualNorm = 0;

        if (fsi.predictor())
        {
            fsi.updateForce();

            fsi.stress().evolve();

            residualNorm =
                fsi.updateResidual();
        }

        do
        {
            fsi.outerCorr()++;

            fsi.updateDisplacement();  // Using selected coupling scheme
```

```
49
50              fsi.moveFluidMesh();
51
52              fsi.flow().evolve();
53
54              fsi.updateForce();  // Face ggi interpolation
55
56              fsi.stress().evolve();
57
58              residualNorm =
59                  fsi.updateResidual();// point ggi interpolation
60          }
61          while
62          (
63              (residualNorm > fsi.outerCorrTolerance())
64           && (fsi.outerCorr() < fsi.nOuterCorr())
65          );
66
67          fsi.flow().updateFields();
68          fsi.stress().updateTotalFields();
69
70          runTime.write();
71
72          Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
73              << "  ClockTime = " << runTime.elapsedClockTime() << " s"
74              << nl << endl;
75      }
76
77      Info<< "End\n" << endl;
78
79      return(0);
80  }
81
82  // ************************************************************************* //
```

Starting with the header files, one can see that the `fluidSolidInterface.H` which is a class of the FSI package is included and it is of great importance. `dynamicFvMesh.H` enables the use of dynamic moving mesh. Then the main code starts from the key word `int main(){}`, the inclusions from line 12 to 15 are the common code for case set up. If we take a overall view of the functional code, we can notice most of code is based on the constructed object `fsi` of class `fluidSolidInterface` in line 19. How the problem is initialized, calculated and converged are done by evoking member functions of object `fsi`. The following content will describe the solving of the flow as well as the purpose of calling each function and how are they connected with the `fluidSolvers` and `solidSolvers`.

- Lines 28 - 32: make the initialization by calling the member functions specified in Table 2.1.
- Lines 34 - 41: make a prediction of initial forces and evoke solid solver and update the residual to prepare for the calculation loop.
- Lines 46 -58: loop the fluid-solid interaction procedure which is firstly updating the solid displacement based on the solid solver and then transfer the deformed mesh to fluid side followed by asking fluid solver to calculate the pressure and velocity. Again the pressure and viscous forces are updated and transported to the solid side to solve new velocity and displacement[2].
- Lines 61 - 63: check residual and iterate the loop until the requiement is met.

- Lines 67 - 70: the final flow field and stress field are written to the output.

The next step is to go through the details of several important lines and check what happens there.

- `fluidSolidInterface fsi(mesh, solidMesh);`

  As mentioned above, the object `fsi` belongs to the class `fluidSolidInterface` and is constructed by two arguments `mesh, solidMesh`. The declaration can be found in *FluidSolidInteraction/src/fluidSolidInteraction/fluidSolidInterface/fluidSolidInterface.H*

```
        //- Construct from components
        fluidSolidInterface
        (
            dynamicFvMesh& fluidMesh,
            fvMesh& solidMesh
        );
```

  and the definition is in *fluidSolidInterface.C* with the main function of reading basic settings in fluid side and solid side and setting the patches and faces index. Some interesting codes are listed below

```
    Foam::fluidSolidInterface::fluidSolidInterface
(
    dynamicFvMesh& fMesh,
    fvMesh& sMesh
)
:
    IOdictionary
    (
        IOobject
        (
            "fsiProperties",
            fMesh.time().constant(),
            fMesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    ),
    fluidMesh_(fMesh),
    flow_(fluidSolver::New(fluidMesh_)),
    solidMesh_(sMesh),
    stress_(solidSolver::New(solidMesh_)),
    couplingScheme_(lookup("couplingScheme")),
    relaxationFactor_(readScalar(lookup("relaxationFactor"))),
    aitkenRelaxationFactor_(relaxationFactor_),
    outerCorrTolerance_(readScalar(lookup("outerCorrTolerance"))),
    nOuterCorr_(readInt(lookup("nOuterCorr"))),
    coupled_(lookup("coupled")),
    predictor_(false), //(lookup("predictor")),
    rbfInterpolation_(false), //(lookup("predictor"))
    ...
```

  Firstly the IO dictionary "fsiProperties" is created, where many parameters are set, such as couplingScheme, relaxationFactor, nOuterCorr and predictors. Another notice is the member data `flow_` and `stress_` as they need to call another member function. Now take `flow_` as

an example, it is initialized by the `fluidSolver::New(fluidMesh_)` selector of the fluidSolver class.

FluidSolidInteraction/src/fluidSolidInteraction/fluidSolvers/fluidSolver/fluidSolver.H

```
    // Selectors

        //- Select constructed from mesh
        static autoPtr<fluidSolver> New(const fvMesh& mesh);
```

The definition of selector `New` is found in *newFluidSolver.C* under *fluidSolver* directory.

```
    Foam::autoPtr<Foam::fluidSolver> Foam::fluidSolver::New(const fvMesh& mesh)
{
    word fluidSolverTypeName;

    // Enclose the creation of the dictionary to ensure it is
    // deleted before the flow is created otherwise the dictionary
    // is entered in the database twice
    {
        IOdictionary fluidProperties
        (
            IOobject
            (
                "fluidProperties",
                mesh.time().constant(),
                mesh,
                IOobject::MUST_READ,
                IOobject::NO_WRITE
            )
        );

        fluidProperties.lookup("fluidSolver") >> fluidSolverTypeName;
    }
    ...
}
```

In the New selector, IOdictionary "fluidProperties" is created again. Which fluidSolver (icoFluid.C or pisoFluid.C) to use is specified by reading the key word after "fluidSolver" subdirectory. One can take "pisoFluid" fluid solver as an example again, the TypeName key word is defined in pisoFluid.C as:

FluidSolidInteraction/src/fluidSolidInteraction/fluidSolvers/pisoFluid/pisoFluid.C

```
defineTypeNameAndDebug(pisoFluid, 0);
addToRunTimeSelectionTable(fluidSolver, pisoFluid, dictionary);
```

So far, it is clear that the `flow_` member data is defined by calling the `New` selector in the `fluidSolver` class and that this selector reads the "fluidProperties" dictionary to look up the predefined type key word of each fluid solver. The same principle goes for the connection between `stress_` and solidSolvers, which will not be discussed here.

- `fsi.flow().evolve();`

  An interpretation of this line is that the object fsi will call a member function `flow()` where another member function `evolve()` will be called continuously. One can see from Table 2.1 that member function `flow()` will return the member data `flow_` which is thoroughly described

above. This `flow_` will select a fluid solver for instance pisoFluid, then the `evolve()` member function in pisoFluid as shown in Table 2.3 is recalled by `flow().evolve()`. The `evolve()` function is equivalent as `pisoFoam` to solve pressure and velocity in pisoFluid.

- `fsi.stress().evolve();`

  The identical principle goes for the solidSolver. The object fsi first evokes the `stress()` function which returns the `stress_` data and another function `evolve` is recalled by `stress()`.

- `fsi.updateForce();`

  This function is to update the interface pressure and viscous force from the fluid side. The definition can be found in *FluidSolidInteraction/src/fluidSolidInterface/fluidSolidInterface.C* with the starting line being like `void Foam::fluidSolidInterface::updateForce()`. The definition code is quite long and since the focus of this tutorial is to clear the connections, only the related code is listed.

```
    void Foam::fluidSolidInterface::updateForce()
{
    Info << "Setting traction on solid patch" << endl;

    vectorField fluidZoneTraction =
        flow().faceZoneViscousForce
        (
            fluidZoneIndex(),
            fluidPatchIndex()
        );

    scalarField fluidZonePressure =
        flow().faceZonePressureForce(fluidZoneIndex(), fluidPatchIndex());
    ...
}
```

  To transport pressure and viscous forces from the fluid side to the solid side, member functions `faceZoneViscousForcefrom()` and `faceZonePressureForce()` of the flow model are recalled to set traction on the solid patch.

- `fsi.updateDisplacement();`

  Similar with `fsi.updateForce();`, this function calculates the new displacement due to pressure and viscous forces and consequently deformed mesh. Based on the definition, one can see that the displacement calculation depends on which coupling scheme i.e. FixedRelaxation, Aitken or IQN-ILS is specified in the case `constant/fsiProperties` directory. Again the interesting part here is how the displacement is related to the solvers. It is obvious to see member function `faceZoneAccelerationunder()` under `stress()` is used during the calculation.

```
void Foam::fluidSolidInterface::updateDisplacement()
{
    ...
            Info << "Setting acceleration at fluid side of the interface" << endl;

        vectorField solidZoneAcceleration =
            stress().faceZoneAcceleration
            (
                solidZoneIndex(),
                solidPatchIndex()
```

```
            );
    ...
}
```

- `fsi.moveFluidMesh();`

  The procedure of moving fluid mesh is to first get the fluid patch displacement from the fluid zone, and a mesh motion solver type should be selected, either "motionU" or "pointMotionU" in order to move the interface mesh.

To summarize, the code of `fsiFoam` solver is not complex but the complexity lies in the library `fluidSolidInteraction` and especially the main class `fluidSolidInterface`, where the information from both fluid and solid sides communicate and exchange. The base classes i.e. fluidSolver sets the base parameters, constructors and member functions while the subclasses e.g. pisoFluid inherits from the base class and specializes itself by an unique evolve() function.

Chapter 3

Adding the interFluid fluidmodel

3.1 Introduction of interFluid

Now the target is to add another fluidSolver class, i.e. `interFluid`, into the *FluidSolidInteraction/src/fluidSoidInteraction/fluidSolvers* directory to widen the applications of the FSI package.

The basic procedure of adding a new interFluid fluid solver is to find an existing similar fluid solver that does almost what one expects, copy and rename it and do the modifications there. In this case, the pisoFluid fluid solver is chosen as a starting point.

```
f40NR
run # change to the user run directory
cd FluidSolidInteraction/src/fluidSolidInteraction/fluidSolvers
cp -r pisoFluid interFluid
cd interFluid
mv pisoFluid.C interFluid.C
mv pisoFluid.H interFluid.H
sed -i s/pisoFluid/interFluid/g interFluid.*
```

So far interFluid is identical to pisoFluid with only name difference. The next step is to go back to `fluidSolidInteraction` library and add this solver to `Make/files` in order to compile it. For compilation, there is no changes of inclusion and library paths in `Make/opotions` as interFluid has not been modified in this stage.

```
cd ../..
sed -i '32i fluidSolvers/interFluid/interFluid.C' Make/files
wmake libso >& log
```

After compilation, the prompt shows up
"`$FOAM_USER_BIN/linux64GccDPOpt/libfluidSolidInteraction.so is up to date`"

For a double check, one can check the tutorial cases by setting *dummy* of the fluidSolver in *fluid/constant/fluidProperties* dictionary.

```
open a new terminal
f40NR
run
cd FluidSolidInteraction/run/fsiFoam/beamInCrossFlow
chmod 755 *               # make sure they are all executable
sed -i s/tcsh/sh/g *Links # change to bash shell
cd fluid
```

```
sed -i s/consistentIcoFluid/dummy/g constant/fluidProperties
./Allclean
./Allrun
```

Take a look at "log.fsiFoam" file, one will see the error but *interFluid* has been added to the valid fluidSover types, which is what we expected.

```
valid fluidSolver types are:
4
(consistentIcoFluid
interFLuid
pisoFluid
icoFluid
)
```

3.2 Modify interFluid

As interFluid has been successfully added as a fluidSolver, the next step is to modify the class code to make sure it performs the identical function as `interDyMFoam` which 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.* A brief comparison of these two directories is carried out in Table 3.1.
Note: the *.H* files in italic are the inclusion of `interFoam` solver but they are also essential to `interDyMFoam`.

interDyMFoam	Description
createFields.H	create the calculation fields
correctPhi.H	make sure the total flux is conserved
createControls	create time controls, boolean correctPhi and and checkMeshCourantNo
readControls	read time controls, correctPhi and checkMeshCourantNo
pEqn.H	PIMPLE algorithm for sloving pressure equation
interDyMFoam.C	the main source code
alphaEqnSubCycle.H	*perform a subCycle on alpha field*
alphaEqn.H	*resolve volumetric flux with alpha*
UEqn.H	*solve velocity equation*
interFluid	Description
interFluid.H	Declaration of member data, constructors and member functions
interFluid.C	Definition of constructors and member functions.

Table 3.1: Comparison of interDyMFoam and interFluid

As we discussed earlier, `interFluid` is a class and because of this no inclusion except the header files should be made in the code. The implementation of the `interDyMFoam` solver into the `interFluid` class is divided into three subsections i.e. Headers and Make files, Constructor and Member function.

3.2.1 Headers and Make files

The header files in `interDyMFoam` and `interFluid` are shown in Table 3.2. Please note that interFluid is still a copy of pisoFluid, i.e. the header files of that are exactly the same as for pisoFluid at the moment. For the modification, we can simply add all the header files under `interDyMFoam.C` into `interFluid.H`. This is because what really matters for the declaration is the indispensable required *.H* files rather than how many files are added. But to be efficient with declarations, the best way is to remove all the red lines in interFluid and add the blue lines from `interDyMFoam.C` to `interFluid.H` and the remaining black lines to `interFluid.C`.

Header files		
interDyMFoam.C	interFluid.C	interFluid.H
fvCFD.H	interFluid.H	fluidSolver.H
dynamicFvMesh.H	volFields.H	volFields.H
MULES.H	fvm.H	surfaceFields.H
subCycle.H	fvc.H	singlePhaseTransportModel.H
interfaceProperties.H	fvMatrices.H	turbulenceModel.H
twoPhaseMixture.H	findRefCell.H	
turbulenceModel.H	adjustPhi.H	
pimpleControl.H	fluidSolidInterface.H	
	addToRunTimeSelectionTable.H	
	fixedGradientFvPatchFields.H	

Table 3.2: Header files of interDyMFoam and interFluid

The explanation is that we remove singlePhaseTransportModel.H since we never use it in interFluid, instead we need interfaceProperties.H and twoPhaseMixture.H to declare the fields `interface_` and `twoPhaseProperties_`. fvCFD.H is added since it contains "uniformDimensionedVectorFields.H" which is used to declare the gravity field `g_`. fvCFD.H also contains fvm.H fvc.H fvMatrices.H so they can be removed as well. The black lines in interDyMFoam.C are moved to interFluid.C because they are mainly used for the declaration of data in member function `evolve()` of interFluid class.

Make/options	
interDyMFoam	fluidSolidInteraction
`EXE_LIBS = \`	`EXE_LIBS = \`
`-linterfaceProperties \`	
`-lincompressibleTurbulenceModel \`	`-lincompressibleTurbulenceModel \`
`-lincompressibleRASModels \`	`-lincompressibleRASModels \`
`-lincompressibleLESModels \`	`-lincompressibleLESModels \`
`-lincompressibleTransportModels \`	`-lincompressibleTransportModels \`
`-lfiniteVolume \`	`-lfiniteVolume \`
`-ldynamicFvMesh \`	`-ldynamicFvMesh \`
`-ldynamicMesh \`	`-ldynamicMesh \`
`-lmeshTools`	`-lmeshTools`
`-llduSolvers \`	
`-L$(MESQUITE_LIB_DIR) -lmesquite`	

Table 3.3: Make/options of interDyMFoam and interFluid

We now move to the `Make` directory for some adjustments. The contents of Make/options file is listed in Table 3.3. One should be aware that interFluid does not have its own Make directory, as it belongs to the `fluidSolidInteraction` library. All the new inclusion paths and linking libraries should thus be added in the library Make/options file. Compared to Make/options of interDyMFoam, three libraries should be added to fluidSolidInteraction. As to the `EXE_INC = \` paths, two more are needed as well i.e.

```
-I$(LIB_SRC)/transportModels/incompressible/lnInclude \
-I$(LIB_SRC)/transportModels/interfaceProperties/lnInclude \
```

3.2.2 Constructor

The constructor is declared in `interFluid.H` as `pisoFluid(const fvMesh& mesh)` and defined in `interFluid.C`. It is constructed from components and used to initialise the private data such as velocity and pressure fields. The private data consists of the initial fields, density, transport and

turbulence model etc and they are quite similar with `createFields.H` file of `interDyMFoam`. The differences are pointed out by Thomas Vyzikas [1]:

- The declaration and initialization of private data are separately stored in *.H* and *.C* files but the sequence should correspond with each other.
- As a convention, member data of class usually ends with an underscore i.e. `U_`, `p_` and `rho_`.
- `runTime.timeName()` in `createFields.H` is changed to `runTime().timeName()` in `interFluid`.
- There should not be any "Info" lines in the constructor.
- The symbol used to separate variables changes from ";" of `createFields.H` to "," in the constructor.

Then it is straight-forward to copy `createFields.H` and paste to interFluid.H and *.C* with the above modifications. Additionally, three variables i.e. gravity `g_` velocity gradient `gradU_` and pressure field `gradp_` should be added. The gravity field `g_` is used to prepare the calculation of total pressure `p` which is equal to dynamic pressure `pd` plus static pressure `rho*gh`. Therefore uniformDimensionedVectorField `g_` is declared and define before the volScalarField `gh` and the code of `g_` could be found in readGravitationalAcceleration.H by command `locate readGravititionalAcceleration.H`. and the path should be:
./foam/foam-extend-4.0/finiteVolume/cfdTools/general/include/readGravitationalAcceleration.H

The gradient fields `gradU_` and `gradp_`are declared after velocity field `U_` and pressure field `p_` and initialized by `gradU_(fvc::grad(U_))` and `gradp_(fvc::grad(p_))`respectively. The final version of the constructor is given in the attached files.

3.2.3 Member function

As known `interFluid.H` is a sub-class of `fluidSolver.H`, so it will inherit the member functions from `fluidSolver.H` which is the same as what `pisoFluid.H` does. So there should not be much difference for the member functions between `pisoFluid.H` and `interFluid.H`. We can simply keep everything unchanged except for the `rho()` function since `interFluid.H` is copied from `pisoFluid.H` . The reason is that function `rho()` returns member data `rho_` that is defined as a dimensionedScalar in pisoFluid but a volScalarField in interFluid. A declaration type shift is necessary in this occasion.

The most difficult part goes for the `interFluid.C` file where all the declared virtual functions of `interFluid.H` are defined. Identical to `pisoFluid` there are totally 8 virtual functions as shown in Table 2.3 in `interFluid.C`. It is quite similar for the first 7 virtual function since they are mainly to return constant fields such as pressure and velocity or calculate patch and zone forces with the consideration of turbulence effect. A tiny change is again the shift type of member data `rho_` from dimensionedScalar to volScalarField. Initially `rho_.value()` is used for the calculation of patch viscous force, but when `rho_` is declared as a volScalarField in interFluid now, there would be no member function `value()` any more but instead scalar `rho_` is used here directly. Besides the usage in patchViscoursForce function, the usage of `rho_.value()` in other functions should be change to `rho_` as well.

Then it comes to modify the `evolve()` member function which is as described a modified version of the `interDyMFoam` solver. In principle all the inclusions of interDyMFoam should be expanded and have their original codes directly implemented into this `evolve()` function. But to make the conversion simple and efficient, some useless or not that important files are removed and some statements are changed. The detailed interDyMFoam.C file with comments is shown below:

```
1                                   interDyMFoam.C
2  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

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

    pimpleControl pimple(mesh);
//- Comment of Line 9
/*- object pimple with argument "mesh" cannot be used in interFluid.C class
the constructor is declared as below in pimpleControl.H

   //- Construct from mesh and the name of control sub-dictionary
   pimpleControl(fvMesh& mesh, const word& dictName="PIMPLE");

but the mesh of interFluid.C is defined as
    const fvMesh& mesh = fluidSolver::mesh();
therefore a "const mesh" can not be used to construct the object pimple
and the following code where a function of pimple is evoked should rewrite
according to its original functions.*/

#   include "readGravitationalAcceleration.H"
#   include "initContinuityErrs.H"
#   include "createFields.H"
#   include "createControls.H"
#   include "correctPhi.H"
#   include "CourantNo.H"
#   include "setInitialDeltaT.H"
//- Comment of Line 23-29
/* gravity, createFields and correctPhi have been implemented in the constructor
of interFluid.C the others will be discussed below.*/

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
#       include "readControls.H"
//- Comment of Line 39
/* there is another inclusion "readTimeControls.H" inside and it is used to read
the control parameters used by setDeltaT*. Other controls such as
"adjustTimeStep", "maxCo", "correctPhi" are also specified.

This inclusion will be skipped as setDeltaT is removed later on and the other
parameters will be set by default.*/

#       include "CourantNo.H"
//- Comment of Line 48
/* This inclusion could be found by "locate CourantNo.H" and it is used to
calculate the mean and maximum Counrant Numbers, should be implemented.*/

        // Make the fluxes absolute
        fvc::makeAbsolute(phi, U);

#       include "setDeltaT.H"
```

```
57  //- Comment of Line 56
58  /* This is for resetting timestep to maintain a constant maximum courant number.
59  For simplicity, "adjustTimeStep" will be false then there is no need to keep it.*/
60
61          runTime++;
62
63          Info<< "Time = " << runTime.timeName() << nl << endl;
64
65          bool meshChanged = mesh.update();
66          reduce(meshChanged, orOp<bool>());
67
68  #       include "volContinuity.H"
69  //- Comment of Line 67
70  /* It is to calculate volume continuity errors and should be kept.*/
71
72          volScalarField gh("gh", g & mesh.C());
73          surfaceScalarField ghf("ghf", g & mesh.Cf());
74
75          if (correctPhi && meshChanged)
76          {
77  #           include "correctPhi.H"
78          }
79  //- Comment of Line 65-78
80  /* romove line 72-73 as "gh" and "ghf" have been implemented in the constructor.
81  remove line 65-66 and if statement as correctPhi is true by default and mesh
82  will definitely change in FSI problems. Only "correctPhi.H" will be unfolded
83  in interFluid.C.*/
84
85          // Make the fluxes relative to the mesh motion
86          fvc::makeRelative(phi, U);
87
88          if (checkMeshCourantNo)
89          {
90  #           include "meshCourantNo.H"
91          }
92  //- Comment of Line 88-91
93  /* remove these lines as "checkMeshCourantNo" is false by default.*/
94
95          // Pressure-velocity corrector
96          while (pimple.loop())
97  //- Comment of Line 96
98  /* As mentioned, pimple cannot be constructed in interFluid, the member function
99  "loop()" cannot be recalled consequently. The function of "loop()" is defined in
100 "pimpleControl.C" which is mainly reading the solutions controls and starting
101 the loop. It can be simply omitted here since the solution controls of interFluid.C
102 is set by looking up the "fluidProperties" dictionary.*/
103
104         {
105             twoPhaseProperties.correct();
106
107 #           include "alphaEqnSubCycle.H"
108
109 #           include "UEqn.H"
110
```

```
111             // --- PISO loop
112             while (pimple.correct())
113 //- Comment of Line 112
114 /*The "correct()" function is an inline function which can be found in
115 "pimpleControlI.H" and it funtions the PISO loop. The code could be changed to
116
117             for (int corr=0; corr<nCorr; corr++)                          */
118             {
119                     #           include "pEqn.H"
120             }
121
122             p = pd + rho*gh;
123
124             if (pd.needReference())
125             {
126                 p += dimensionedScalar
127                 (
128                     "p",
129                     p.dimensions(),
130                     pRefValue - getRefCellValue(p, pdRefCell)
131                 );
132             }
133
134             turbulence->correct();
135         }
136
137         runTime.write();
138
139         Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
140             << "  ClockTime = " << runTime.elapsedClockTime() << " s"
141             << nl << endl;
142     }
143
144     Info<< "End\n" << endl;
145
146     return 0;
147 }
148 // ************************************************************************* //
```

After the above interpretation and reasonable modifications, we will move to the required inclusions i.e. correctPhi.H in line 77, alphaEquSubCycle.H in line 106, UEqn.H in line 108 and pEqn.H in line 118, they are all located in either *applications/solvers/multiphase/interDyMFoam* or *interFoam* directory and their descriptions are shown in Table 3.1. To implement these inclusions, several modifications are still necessary and discussed below.

1. The recalling of function `while (pimple.correctNonOrthogonal())` in both correctPhi.H and pEqn.H should be reverted a for loop to do the non-orthogonal correction.

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

   Another function `if (pimple.finalNonOrthogonalIter())` will be changed to

   ```
   if (nonOrth == nNonOrthCorr)
   ```

2. The code of looking up the number of alpha corrections and subCycles in alphaEquSubCycle.H should be changed since function `pimple.dict()` is invalid here.

```
label nAlphaCorr
(
  readLabel(pimple.dict().lookup("nAlphaCorr"))
);

label nAlphaSubCycles
(
    readLabel(pimple.dict().lookup("nAlphaSubCycles"))
);
```

   The new code is inspired by readPIMPLEControls.H where a pimple dictionary is defined as the sub-dictionary "PIMPLE" of fvSolution in which the subCycle parameters can be read.

```
dictionary pimple = mesh.solutionDict().subDict("PIMPLE");

label nAlphaCorr
(
   readLabel(pimple.lookup("nAlphaCorr"))
);

label nAlphaSubCycles
(
  readLabel(pimple.lookup("nAlphaSubCycles"))
);
```

   One should also be aware that alphaEqn.H is a sub-inclusion of alphaEquSubCycle.H and it needs to be unfolded as well.

3. The function `if (pimple.momentumPredictor())` in UEqn.H should be deleted. There are two reasons: one is as usual taht the pimple function is not allowed to be used here and another is that this function actually always return true by default. If one would like to look at the details, the momentumPredictor() function is an inline bool member function of solutionControl.H which is the base class of pimpleControl.H.

4. Besides the usage of `pimple.correctNonOrthogonal()` and `pimple.finalNonOrthogonalIter()`, pEqn.H contains `pimple.finalInnerIter()` as well. The definition is in pimpleCountrolI.H and it checks the final correction. A key word "pdFinal" can be used instead.

```
pdEqn.solve
(
   //mesh.solutionDict().solver(pd.select(pimple.finalInnerIter()))
   mesh.solutionDict().solver("pdFinal")
);
```

To summarize, the majority of modifications are how to rewrite the code to function identically as the invalid pimple object and its member functions. The completed version of member functions is provided in the attached files.

3.3 Compile interFluid

As we have already added "fluidSolvers/interFluid/interFluid.C" into Make/files under *FluidSolidInteraction/src/fluidSoidInteraction* directory during the introduction of interFluid, the compilation is straight-forward by issuing the command `wmake libso` in the *fluidSolidInteraction* directory. If everything goes right, the fluidSolidInteraction.so will be updated including the new interFluid fluidSolver model.

It is highly recommended that one performs the modification and compilation simultaneously and reads carefully about the errors in the log file.

3.4 Compile myfsiFoam

In section 3.2.1, we discussed that `interfaceProperties.so` is added to *fluidSolidInteraction/Make/options* to link this library for interFluid. Meanwhile, the solver fsiFoam requires the linking to this library as well.

```
cd FluidSolidInteraction/src/solvers
cp -r fsiFoam myfsiFoam
cd myfsiFoam
mv fsiFoam.C myfsiFoam.C
sed -i s/fsiFoam/myfsiFoam/g Make/files
echo "-linterfaceProperties" >> Make/options # check by vim again and be careful with the syntax
wmake
```

This added "myfsiFoam" will be used in the tutorial case test.

3.5 Suggested future improvement

The above implementation of interDyMFoam into interFluid `evolve()` member function is quite complex and one main reason is the invalid usage of the object pimple. As mentioned, the constructor of pimpleControl class requires an argument of non-constant `fvMesh& mesh` while the mesh in our case is defined as constant which fails the construction. Instead of rewriting the identical functional codes as what was done previously, another smart way is to convert the constant mesh into non-constant with command `const_cast`. Consequently the pimple object and its member functions could remain the same as before without modifications.

Chapter 4

Validation of interFluid

4.1 Tutorial case beamInCrossFlow

The tutorial cases are all located in the *FluidSolidInteraction/run/fsiFoam/* directory. beamInCrossFlow is chosen as an example in this tutorial but the others have the same file structure and running principle. There are **fluid** folder, **solid** folder, **setInletVelocity** folder and ***Links** files under the beamInCrossFlow case. The fluid folder and solid folder contains all the settings for fluid and solid respectively such as the time directory 0, constant and system dictionaries. The setInletVelovity is a library folder required by the functions defined in *fluid/system/controlDict*. The *Links files are used to link the fluid and solid solvers.

In the fluid folder, the fluidSolver (e.g. icoFluid) and the corresponding coefficients (e.g. nCorrectors) are specified in *fluid/constant/fluidProperties*, and the interface settings (e.g. couplingScheme) are specified in *fluid/constant/fsiProperties*. Similar to the solid folder.

To run the case, one should go to the fluid directory where the Allrun script is located and execute it. One more step is to change the *Links files shell mode from *bcsh* to *sh* and the commands are:

```
f40NR
run # change to the user run directory
cd FluidSolidInteraction/run/fsiFoam/beamInCrossFlow
sed -i s/tcsh/sh/g *Links
cd fluid
./Allrun
touch beamInCrossFlow{fluid}.foam
touch beamInCrossFlow{solid}.foam
paraview
```

For the visualization, the reader needs to open both beamInCrossFlowfluid.foam and beamInCrossFlowsolid.foam in paraview and select the "Mesh Regions" to be internalMesh and solid/internalMesh respectively. The tool "WarpByVector" is applied to beamInCrossFlowsolid.foam in the next step. After that, one would clearly observe that the solid beam deflects towards the fluid flowing direction in paraview after simulation.

4.2 Case setup with interFluid solver

To avoid ruining the original tutorial cases, the first step is to copy beamInCrossFlow and rename it to interFluidBeamInCrossFlow. Then one needs to specify interFluid will be used in *fluid/constant/fluidProperties* as well as the coefficient subdirectory.

```
fluidSolver interFluid;

interFluidCoeffs
{
   nCorrectors 3;
   nNonOrthogonalCorrectors 1;
   nOuterCorrectors 1;
}
```

Currently the settings for **constant** and **system** directory are still based on the old icoFluid case but the parameters are not sufficient anymore for interFluid. The adjustments includes changing new boundaries in time directory, adding gravity and turbulence properties in constant directory, adding setFieldsDict and adjusting schemes and solvers in solution directory. Most of the modifications refer to the tutorial `$FOAM-TUTORIALS/multiphase/interDymFOam/ras/damBreakWithObstacle` case.

1. 0 directory

   VolScalarField alpha1 is needed and copied from damBreakWithObstacle/0 but the boundary fields need to be adjusted to the current geometry.

   Pressure field p first needs to be renamed to pd corresponding to the fvScheme and fvSolution setting later on. In this case simulation, the outlet boundary is treated as a wall, `type zeroGradient` is used on the outlet. The top boundary is adjusted to the same setting `type totalPressure` as the atmosphere boundary in damBreakWithObstacle/0/pd. The others remain the initial setting with `type extrapolatePressure`, which is defined in the source directory *fluidSolvers/fvPatchFields*.

   Velocity field U should have an outlet boundary `type fixedValue; value uniform (0 0 0)` as it is a wall and a `type pressureInletOutletVelocity; value uniform (0 0 0)` on the top boundary field.

2. constant directory

   There are three more files namely turbulenceProperties, g and RASProperties needed in the constant directory. TransportProperties file needs to specify two phase properties.

   ```
   cd interFluidBeamInCrossFlow/fluid
   cp $FOAM_TUTORIALS/multiphase/interDymFoam/ras/damBreakWithObstacle/
   \ constant/turbulenceProperties constant/
   cp ... # the same to RASProperties and g
   ```

3. system directory

   The application name in system/controlDict should be changed from the old "fsiFoam" to the new compiled "myfsiFoam". The time step needs to be adjusted to a smaller one.

   The setFieldsDict is copied from the damBreakWithObstacle case and the water region is set as the left part.

   system/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
      );
 }
)
```

In fvScheme, the divSchemes and laplacianSchemes are changed to exactly the same as damBreakWithObstacle/system/fvSchemes.

system/fvSchemes

```
divSchemes
{
default         none;
// div(phi,U)   Gauss linearUpwind cellLimited Gauss linear 1;
div(phi,U)   Gauss upwind;
div(rho*phi,U)  Gauss upwind;
div(phi,alpha)  Gauss vanLeer;
div(phirb,alpha) Gauss interfaceCompression;
}

laplacianSchemes
{
default        Gauss linear corrected;
// laplacian(nu,U) Gauss linear skewCorrected 1;
// laplacian((1|A(U)),p) Gauss linear skewCorrected 1;
// laplacian(diffusivity,cellMotionU) Gauss linear skewCorrected 1;
}
```

The original Gauss linearUpwind and Gauss linear skewCorrected schemes are more accurate for bad mesh quality but also computational consuming.

In fvSolution, pd is used as the pressure symbol as mentioned in 0 directory. The solvers are mainly identical to damBreakWithObstacle/system/fvSolution as well both for the existing pd, U and the adding k, B, nuTilda etc. The control parameters in PIMPLE sub-directory is also defined in fvSolution.

system/fvSolution

```
pd
{
    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 30;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
```

If one is interested in the GAMG method, one can investigate more on the result sensitivity to the parameter changing,l for example a higher mergeLevels number should lead to a faster solution.

4.3 Running the case

One should first take a look at the Allrun script in the fluid folder and pay attention to the runApplication command. Section 3.4 compiles a new solver "myfsiFoam" for the tutorial case, then one should change the default setting "application fsiFoam" in *fluid/system/controlDict* to "application myfsiFoam"

Since alpha1 field is needed, `setFields` command should be added into `Allrun` script.

fluid/Allrun

```
#!/bin/sh
# Source tutorial run functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions
# Get application name
application=`getApplication`
...
cd fluid
runApplication setFileds     #add setFields command
runApplication $application
```

Another step is to comment the functions part in *system/controlDict* because it brings some complains to the solver.

Run the case with command `./Allrun` and one can always monitor the lasted computational stage by the command `less log.fsiFoam`.

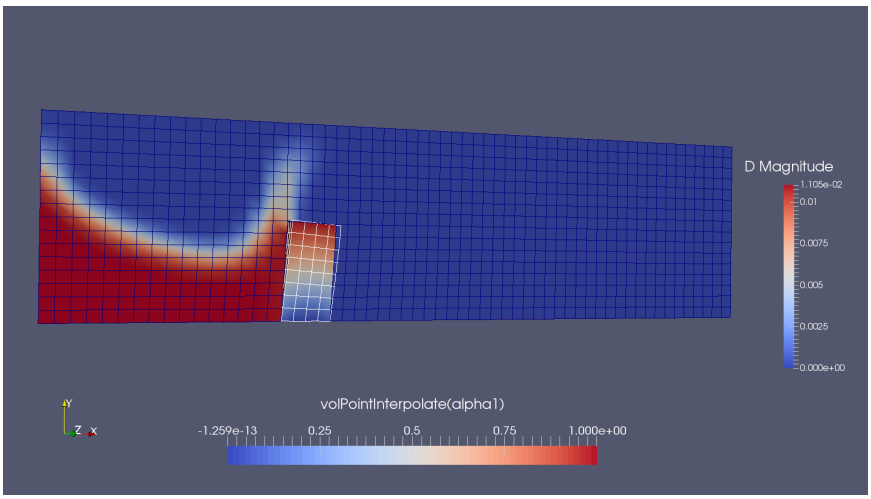

Figure 4.1: Large deformation case

In the current condition, the solver will crash at some time when the flow flushes the beam as shown in Figure 4.1. The capital letter D stands for displacement and it has a larger value in the top face of the beam. The crash reason could be the relatively too large deformation of the beam which produces a mesh gap between the fluid and solid sides (zoom in to see the mesh gap along the interface).

To fix this problem, one can increase the beam stiffness in *solid/constant/rheologyProperties*.

```
                    solid/constant/rheologyProperties
rheology
{
     type linearElastic;
     rho rho [1 -3 0 0 0 0 0] 1000;
     E E [1 -1 -2 0 0 0 0] 1.4e10;  // orig. 1.4e6
     nu nu [0 0 0 0 0 0 0] 0.4;//poissson's ratio
}
```

One can also try to decrease the tolerance lever or increase the maximum iteration number in *fluid/constant/fsiProperties* to make the solution more converged.

```
                    fluid/constant/fsiProperties
outerCorrTolerance 1e-6; // tolerance
nOuterCorr 40; // maximum iteration
...
```

The new result with higher stiffness and more converged level is shown in Figure 4.2. The arrow indicates the displacement and the displacement is rather small as seen in the D magnitude bar, but at least it works to finish the simulation and the result is decent. The solid stress distribution is illustrated as well and the bottom edges undertakes the largest stress due to the water induced bending moment. As expected, one can see the second cycle of beam deformation when the flow hits the wall and returns back although the displacement will be even smaller this time.

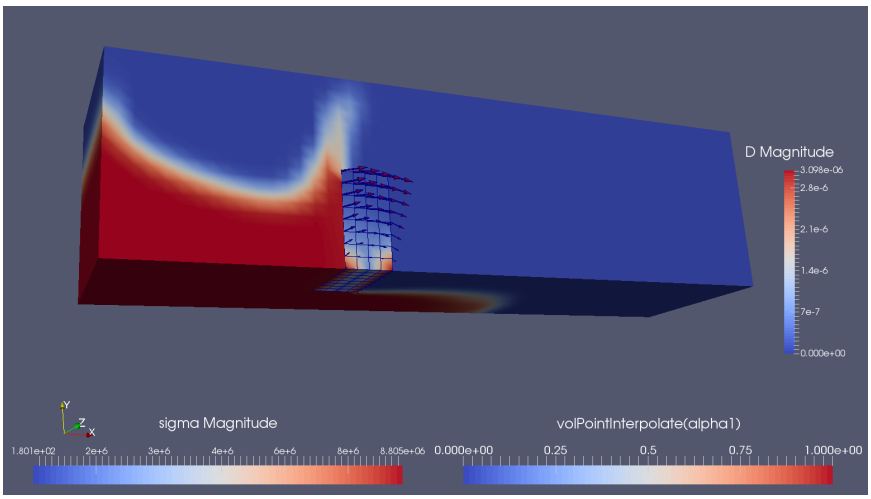

Figure 4.2: Small deformation case

A tip of testing new added interFluid is to use fluidFoam first to purely evoke fluid solver. If fluidFoam works well, it indicates interFluid setting is completed then try fsiFoam to solving the interaction problem.

Chapter 5

Conclusion and future work

This tutorial provides a general overview of fluid solid interaction (FSI) package and clearly illustrates the connection between solvers e.g. fsiFoam and classes e.g. pisoFluid. It is not that difficult to implement interFoam into interFluid in FSI package but several limitations are set, for example, the omission of `setDeltaT.H` that could adjust time steps. Similarly any other fluid or solid solver can be implement in the same process as interFluid implementation. The FSI package will be greatly enriched by this way.

As to the validation case, fsiFoam with interFluid as fluid solver is only suitable for small displacement problems which should be investigated further to fix the mesh slip problems. In addition, it would be interesting to see how the boundary conditions affect the interactions for example, what would happen if we change solid boundary from fixedDisplacement to timeVaryingFixedDisplacement.

Study questions

How to use it:

- Where to find the fsiFoam solver and the pisoFluid fluid solver?
- Is Allrun script in the case folder located in the fluid directory or the solid directory and what does it do?

The theory of it:

- What information is required from fluid and solid sides respectively and how does it transferred and exchanged?

How it is implemented:

- What is the main difference between a top-lever fluid solver and an implemented fluid solver class e.g. pisoFoam VS pisoFluid?
- How the fsiFoam solver understands which fluid and solid solvers are specified in the case?

How to modify it:

- What should be modified when a new fluid solver model is added?
- How the case is set up to verify the implemented fluid solver?

Bibliography

[1] Thomas Vyzikas. *The implementation of interFoam solver as a flow model of the fsiFoam solver for strong fluid-structure interaction*. Tech. rep. CFD with OpenSource software Course, 2014.

[2] Hua-Dong Yao. *SIMULATION OF FLUID-STRUCTURAL INTERACTION USING OPENFOAM*. Tech. rep. CFD with OpenSource software Course, 2014.