Cite as: Li, Y.: Implementation of multiple time steps for the multi-physics solver based on chtMultiRegionFoam. 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:

Implementation of multiple time steps for the multi-physics solver based on chtMultiRegionFoam

Developed for FOAM-extend-4.0

Author:
Yuzhu Pearl Li
University of Stavanger
yuzhu.li@uis.no

Peer reviewed by:
Alasdair MacKenzie
Turo Välikangas
Gregor Cvijetić
Håkan Nilsson

February 16, 2017

Learning outcomes

The reader will learn:

How to use it

- how the tutorial case is set up before running the `chtMultiRegionFoam` solver in FOAM-extend-4.0 (Section 2.2).
- how to visualize the `chtMultiRegionFoam` cases in FOAM-extend-4.0 by paraView (Section 5.1).

The theory of it

- the code structure of the `chtMultiRegionFoam` solver (Section 2.1 and 3.1).
- the theory of the PIMPLE algorithm (Section 3.2).
- the equations of the conjugate heat transfer in fluid and solid regions (Section 3.3-3.4).
- the boundary condition coupling algorithm for multiple regions/physics (Section 3.5).

How it is implemented

- how to implement multiple time steps for multiple regions in `chtMultiRegionFoam` solver by two different approaches (Section 4.1).
- how to test the implemented solvers (Section 4.2.2 and Section 4.3.2).

How to modify it

- how to modify the PIMPLE loop and other corresponding files to introduce an extra time step for the solid regions (Section 4.2-4.3).
- how to modify the current temperature-coupled boundary condition into a new pressure-coupled boundary condition (Chapter 6).

Acknowledgements

I would like to thank Professor Håkan Nilsson for providing the course of CFD with open-source software OpenFOAM and the continuous support from him during this project. I am greatly benefited from this course. I would like to express my special gratitude to Gregor Cvijetić for his help and very useful advices for my project. I also appreciate the communication and knowledge sharing from Turo Välikangas, for we were both working on the conjugate heat transfer solvers.

I am immensely grateful to my supervisor Professor Muk Chen Ong in the University of Stavanger; without his support, I would not have the chance to go for this course.

Many thanks to all the classmates in this course for we have shared a good time and knowledge with each other.

Contents

Chapter 1

Introduction

This work is to demonstrate the structure of the `chtMultiRegionFoam` solver in FOAM-extend-4.0 and to implement modifications upon it. The installation instruction of FOAM-extend-4.0 can be found on the website [1]. In this report, the structure of the multi-region solver `chtMultiRegionFoam` and the boundary coupling methods between multiple regions will be illustrated.

The solver modification part in this report is regarded as an initial step to develop a wave-structure-soil (WSS) interaction solver. The WSS multi-physics solver solves the interaction between wave, structure and soil domains[1] , as shown in Figures 1.1 and 1.2 (with and without the existence of a structure). Wave pressure imposes directly on the soil and on the structure, triggering stresses and displacements; and the structure stresses impose on the soil which is an indirect effect from the wave pressure. The interaction between the multiple physical domains is achieved by boundary coupling at the interfaces. The boundary coupling method will be explained in section 3.5.

The development of the WSS solver is based on `chtMultiRegionFoam` since they have the following features in common:

- The `chtMultiRegionFoam` solver is a multi-regional (multi-physics) solver for the conjugate heat transfer (CHT) between fluid and solid regions. Similar to this, the wave-structure-soil interaction solver will also solve the fluid (wave) and solid (structure and soil) regions. A difference is that in WSS, the wave is solved using a solver for incompressible two-phase flows containing a free surface, while in conjugate heat transfer the air is solved as a compressible one-phase fluid.

- Anther common part between conjugate heat transfer and wave-structure-soil interaction is that the displacement and deformation of the solid regions can be negligible. For conjugate heat transfer, it is straightforward of such assumption; For wave-structure-soil interaction, it is assumed that the displacement and deformation of the solids (structure and soil) are minor compared to the wave length, thus they are negligible and the mesh will not be deformed. This common part provides the possibility of using different time steps and different cell sizes for the fluid and solid regions respectively.

- The `chtMultiRegionFoam` is a transient solver that performs the calculation of the CHT during each time step until the convergence is achieved, while solving the WSS interaction is a same process.

To implement a multi-physics solver, it is worthwhile to consider that different properties may have various demands for convergence and stability. In the engineering practice, for example, for solving the interaction between fluid (wave) and solid (soil) regions, the time step and mesh size for the solid region(s) can be much larger than what is needed for solving fluid region(s).

[1]Note that the term *domain* is different from the term *region* in this report: a *domain* relates to a *physics* that contains multiple *region(s)* with the same physical property. For example, in chtMultiRegionFoam, there are only two *domains* (fluid and solid), while there are multiple *regions* (such as the topAir, bottomAir) for each *domain*.

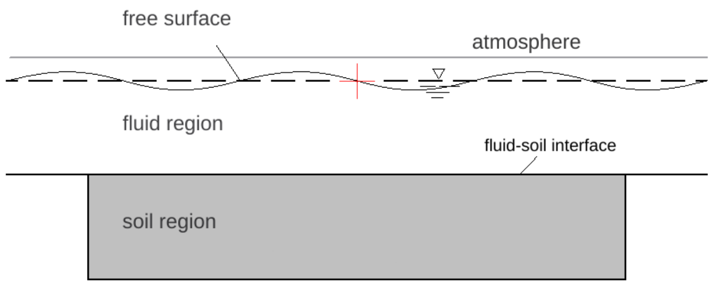

Figure 1.1: A demonstration of the fluid wave-soil interaction.

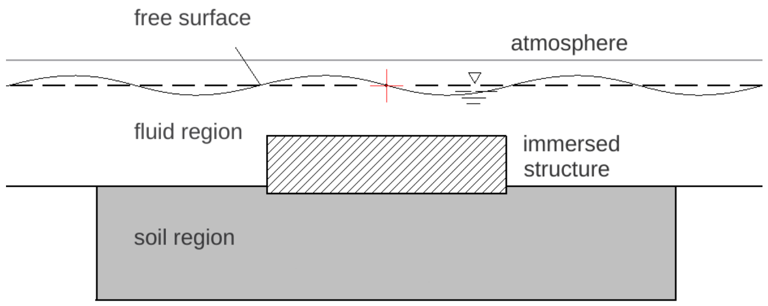

Figure 1.2: A demonstration of the wave-structure-soil interaction.

For solving the unsteady fluid regions in OpenFOAM, to achieve temporal accuracy and numerical stability, a Courant number less than 1 is required [2]. The Courant number is defined for one cell as

$$Co = \frac{\delta t |U|}{\delta x} \tag{1.1}$$

where δt is the time step, $|U|$ is the magnitude of the velocity through that cell and δx is the length scale of the cell in the direction of the velocity. Therefore, the setting of the time step δt should be small enough to satisfy the Courant number requisition. What is more, for solving the free surface or turbulent problems, the cell size needs to be small enough to achieve more accurate results. On the contrary, in solid mechanics, the mesh size and time step can be relatively larger compared to solving the fluids. Therefore, for some multi-physics problems, it is not efficient to calculate the multiple regions with the same mesh size and time step.

The `chtMultiRegionFoam` solver has interface mesh mapping that allows non-conformal meshes and it has access to interface coupling functionality. However, the fluid and solid regions are solved with the same time step in a loop, which needs to be modified to our case. To achieve multiple time steps for each physical domain based on the `chtMultiRegionFoam` solver, the following modifications need to be implemented:

- First, a different time step for the solid regions is to be introduced.
- Then the multi-region PIMPLE loop is to be modified to solve different domains in various time steps.
- Two approaches to implement the multiple time steps are to be demonstrated: one approach allows the adjustment of the time step while the other does not.
- Info statements for debugging purposes are to be added into the codes.
- Case studies are to be conducted to verify the new solvers.

In addition to the implementation of the multiple time steps, a new boundary condition called `solidWallMixedPressureCoupled` which applies a pressure mapping between the fluid and solid regions for WSS interaction is developed. The boundary condition is tested but not verified at this stage.

To finalize the WSS solver, further work needs to be performed, such as the modification of the solid solver and fluid solver. Due to the time limitation of this project, these modifications will not be performed at this stage. Future work is discussed in the last chapter.

Chapter 2

The chtMultiRegionFoam Solver

The `chtMultiRegionFoam` solver is a multi-region (multi-physics) solver that solves transient conjugate heat transfer between solid regions and fluid regions.

2.1 File structure of the solver

Since the `chtMultiRegionFoam` solver deals with multiple domains in different physical properties, the file structure of the solver is different from the single-physics solvers. The file structure of the `chtMultiRegionFoam` solver is shown as the tree below:

```
|----- chtMultiRegionFoam.C
|----- derivedFvPatchFields
|   |----- solidWallHeatFluxTemperature
|   |   |----- solidWallHeatFluxTemperatureFvPatchScalarField.C
|   |   |----- solidWallHeatFluxTemperatureFvPatchScalarField.H
|   |----- solidWallMixedTemperatureCoupled
|       |----- solidWallMixedTemperatureCoupledFvPatchScalarField.C
|       |----- solidWallMixedTemperatureCoupledFvPatchScalarField.H
|----- fluid
|   |----- compressibleContinuityErrors.H
|   |----- compressibleCourantNo.C
|   |----- compressibleCourantNo.H
|   |----- compressibleMultiRegionCourantNo.H
|   |----- createFluidFields.H
|   |----- createFluidMeshes.H
|   |----- initContinuityErrs.H
|   |----- readFluidMultiRegionPIMPLEControls.H
|   |----- readFluidMultiRegionPISOControls.H
|   |----- setRegionFluidFields.H
|   |----- solveFluid.H
|   |----- storeOldFluidFields.H
|   |----- UEqn.H
|   |----- hEqn.H
|   |----- pEqn.H
|----- include
|   |----- setInitialMultiRegionDeltaT.H
|   |----- setMultiRegionDeltaT.H
|----- Make
|   |----- files
|   |----- options
```

```
|----- readPIMPLEControls.H
|----- regionProperties
|   |----- regionProperties.C
|   |----- regionProperties.H
|----- solid
    |----- createSolidFields.H
    |----- createSolidMeshes.H
    |----- readSolidMultiRegionPIMPLEControls.H
    |----- readSolidMultiRegionPISOControls.H
    |----- readSolidTimeControls.H
    |----- setRegionSolidFields.H
    |----- solidRegionDiffNo.C
    |----- solidRegionDiffNo.H
    |----- solidRegionDiffusionNo.H
    |----- solveSolid.H
```

where

- `chtMultiRegionFoam.C`, the main source file, calls the needed files and solvers.
- `regionProperties/`, the subdirectory contains files that read fluid and solid region names set in the constant/regionProperties of the case files.
- `fluid/`, the subdirectory contains source files that solve equations for continuity momentum, enthalpy, pressure for the fluid regions.
- `solid/`, the subdirectory contains source files for solving heat conduction equation in solid regions.
- `derivedFvPatchFields/`, the subdirectory contains files that set new boundary conditions for the coupling between solid and fluid domains.
- `include/`, the subdirectory contains files to set/reset the multi-region time step.
- `Make/`, the subdirectory contains files for compilation purpose.

An insight of the code structure of the `chtMultiRegionFoam` solver will be presented in Chapter 3.

2.2 Case setup

A typical OpenFOAM case directory consists of the following three folders:

- 0
- constant
- system

This general case structure is also kept for multi-regional cases, but in each of those directories there is one additional directory for each region. An example below is taken from the tutorial case in `$FOAM_TUTORIALS/heatTransfer/chtMultiRegionFoam/multiRegionHeater` in FOAM-extend-4.0. The structure of the multi-regional case before running the solver is shown as below, where the the region sub-directories for the time directories are created by the `Allrun` script.

```
|--- 0
   |--- cellToRegion
    |--- cp
    |--- epsilon
    |--- k
    |--- Kappa
    |--- p
    | --- rho
    | --- T
    | --- U
|--- 0.001
   | --- bottomAir
   |    |--- cellToRegion
   |    |--- cp
   |    |--- epsilon
   |    |--- k
   |    |--- Kappa
   |    |--- p
   |    |--- rho
   |    |--- T
   |    |--- U
   |    |--- polyMesh
   |       | --- ...
   | --- heater
   |    |--- cellToRegion
   |    |--- cp
   |    |--- epsilon
   |    |--- k
   |    |--- Kappa
   |    |--- p
   |    |--- rho
   |    |--- T
   |    |--- U
   |    |--- polyMesh
   |       | ---...
   | --- leftSolid
   |    |--- cellToRegion
   |    |--- cp
   |    |--- epsilon
   |    |--- k
   |    |--- Kappa
   |    |--- p
   |    |--- rho
   |    |--- T
   |    |--- U
   |    |--- polyMesh
   |       | --- ...
   | --- rightSolid
   |    |--- cellToRegion
   |    |--- cp
   |    |--- epsilon
   |    |--- k
   |    |--- Kappa
   |    |--- p
   |    |--- rho
   |    |--- T
   |    |--- U
   |    |--- polyMesh
   |       | --- ...
   | --- topAir
        |--- cellToRegion
        |--- cp
        |--- epsilon
        |--- k
        |--- Kappa
        |--- p
        |--- rho
        |--- T
        |--- U
        |--- polyMesh
           | --- ...
|--- constant
   | --- polyMesh
   |    |--- blockMeshDict
   |    |--- ...
   | --- bottomAir
   |    |--- g
   |    |--- RASProperties
   |    |--- thermophysicalProperties
   |    |--- turbulenceProperties
   | --- topAir
        |--- g -> ../bottomAir/g
        |--- RASProperties
        |--- thermophysicalProperties -> ../bottomAir/thermophysicalProperties
        |--- turbulenceProperties -> ../bottomAir/turbulenceProperties
   | --- regionProperties
   | --- cellToRegion
|--- system
   | --- controlDict
   | --- fvSchemes
   | --- fvSolution
   | --- bottomAir
   |    |--- changeDictionaryDict
   |    |--- fvSchemes
   |    |--- fvSolution
   | --- heater
   |    |--- changeDictionaryDict
   |    |--- fvSchemes
   |    |--- fvSolution
   | --- leftSolid
   |    |--- changeDictionaryDict
   |    |--- fvSchemes -> ../heater/fvSchemes
   |    |--- fvSolution -> ../heater/fvSolution
   | --- rightSolid
   |    |--- changeDictionaryDict
   |    |--- fvSchemes -> ../heater/fvSchemes
   |    |--- fvSolution -> ../heater/fvSolution
   | --- topAir
        |--- changeDictionaryDict
        |--- fvSchemes -> ../bottomAir/fvSchemes
        |--- fvSolution -> ../bottomAir/fvSolution
|--- Allrun
```

In the tree, the symbol `->` denotes the file is linked to another file.

- In the `0/` directory, boundary condition files for all desired fields have to be created by the user. This directory is only needed when setting up the tutorial case which is not needed by the solver.
- In the `0.001/` directory, the files are created by the `Allrun` script. Variables and the mesh files in this directory are actually needed by the solver. The tutorial case is started to be solved from the time step of 0.001.
- In the `constant/` directory, `polyMesh/` defines the initial geometry for the full domain, and it is only needed when setting up the tutorial case. It is important to notice that, in this tutorial, the region meshes are created by first making a mesh for the entire domain, then splitting it up into regions using `setSet`, `setsToZones`, and `splitMeshRegions`. After that the mesh in `constant/polyMesh/` is not used anymore.

  The file `regionProperties` specifies region names and assigns the physical phase to each region: either fluid or solid.

  For the fluid regions (bottomAir and topAir), there is a `thermophysicalProperties` file containing the properties of the fluid, and also `RASProperties` and `turbulenceProperties` files, which provide settings and parameters of the turbulent model.
- In the `system` directory, the `changeDictionaryDict` file in each folder contains details about the necessary fields in the region. The OpenFOAM application `changeDictionary` will look up for the dictionary files in the `system/regionName` folders and then create initial, boundary and coupling conditions for all fields existing in `0` directory for all regions. The `changeDictionaryDict` file is specially needed for setting up this tutorial case and it is not required for running the solver.

  There should be one `fvSolution` file for each region, for example `system/bottomAir/fvSolution`, since each region has its own settings for the solution in that region. However, a dummy file `system/fvSolution` is also required (the reason will be explained in the PIMPLE loop in Section 3.2). The same settings have to be done in both files.

  The `fvSchemes` for each region, for example, `system/bottomAir/fvSchemes` is required but the `system/fvSchemes` file can actually be removed.
- The `./Allrun` script for the tutorial case `multiRegionHeater` calls a series of additional functions before running the solver. Its main part is shown below:

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

#Pre-processing the tutorial case
rm -rf constant/polyMesh/sets
runApplicationAndReportOnError blockMesh
runApplicationAndReportOnError setSet -batch makeCellSets.setSet
rm constant/polyMesh/sets/*_old
runApplicationAndReportOnError setsToZones -noFlipMap
runApplicationAndReportOnError splitMeshRegions -cellZones
cp 0/* 0.001/bottomAir/
cp 0/* 0.001/heater/
cp 0/* 0.001/leftSolid/
cp 0/* 0.001/rightSolid/
cp 0/* 0.001/topAir/
runApplication changeDictionary -region bottomAir
mv log.changeDictionary log.changeDictionary_bottomAir
runApplication changeDictionary -region topAir
mv log.changeDictionary log.changeDictionary_topAir
runApplication changeDictionary -region heater
mv log.changeDictionary log.changeDictionary_heater
runApplication changeDictionary -region leftSolid
mv log.changeDictionary log.changeDictionary_leftSolid
runApplication changeDictionary -region rightSolid
mv log.changeDictionary log.changeDictionary_rightSolid

#Running the solver
runApplicationAndReportOnError chtMultiRegionFoam
```

It is seen that `blockMesh`, `setSet`, `setsToZones`, `splitMeshRegions` and `changeDictionary` were run to finish the case preparation before running the `chtMultiRegionFoam` solver. The purposes of each step is explained as below:

`blockMesh`: creates a mesh and defines the geometry for the full domain.

`setSet`: uses the makeCellSets.setSet in the top-level case directory to create and define the cellSets.

`setsToZones`: converts the cellSets to the the cellZones that define the regions.

`splitMeshRegions`: splits the mesh into multiple regions, according to the cellZones.

`changeDictionary`: defines initial, boundary and coupling conditions for all fields of all regions.

The `runApplicationAndReportOnError` is defined in the `RunFunctions` in the header of the `Allrun` script to run the applications and to report the errors. It is a part of the test-loop, so that the test loop can run the tutorial and see if it works or not.

The case setup for `chtMultiRegionFoam` is not a main tutorial in this project. A detailed case setup for `chtMultiRegionFoam` has been demonstrated in the report of Singal [5]. It is worthwhile to mention that the case setting of `multiRegionHeater` in OpenFOAM-4.0x differs from the setting in FOAM-extend-4.0.

Chapter 3

A walk through the solver

The general steps of the `chtMultiRegionFoam` solver are [6]:

- Define multiple meshes, one for each 'region'
- Create field variables on each mesh
- Solve separate governing equations on each mesh
- Apply a multi-region coupling at the boundary interface
- Iterate until the coupled solution is fully converged

In this chapter, we will walk through the solvers in the `chtMultiRegionFoam` including the main source file `chtMultiRegionFoam.C`, the fluid solver, the solid solver and the derived coupling boundary conditions.

3.1 The main file

We had an overview of multi-physics problems and the case setup of the `chtMultiRegionFoam` solver. Now, from a developer's perspective, how is the `chtMultiRegionFoam` solver constructed?

First, let us get into the top-level directory and look into the main source file, `chtMultiRegionFoam.C`. Open a new terminal window and source FOAM-extend-4.0 by:

```
f40NR
```

or by other commands such as `fe40`, depending on the alias set by the user for initialising FOAM-extend-4.0. Then,

```
cd $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam
vi chtMultiRegionFoam.C
```

The source file of `chtMultiRegionFoam.C`, shown in the box below, provides an overview of how the solver is constructed:

```
\*---------------------------------------------------------------------------*/
#include "fvCFD.H"
#include "basicPsiThermo.H"
#include "turbulenceModel.H"
#include "fixedGradientFvPatchFields.H"
#include "regionProperties.H"
#include "compressibleCourantNo.H"
#include "solidRegionDiffNo.H"

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

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

    regionProperties rp(runTime);

    #include "createFluidMeshes.H"
    #include "createSolidMeshes.H"

    #include "createFluidFields.H"
    #include "createSolidFields.H"

    #include "initContinuityErrs.H"

    #include "readTimeControls.H"
    #include "readSolidTimeControls.H"


    #include "compressibleMultiRegionCourantNo.H"
    #include "solidRegionDiffusionNo.H"
    #include "setInitialMultiRegionDeltaT.H"

    while (runTime.run())
    {
        #include "readTimeControls.H"
        #include "readSolidTimeControls.H"
        #include "readPIMPLEControls.H"


        #include "compressibleMultiRegionCourantNo.H"
        #include "solidRegionDiffusionNo.H"
        #include "setMultiRegionDeltaT.H"

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        if (nOuterCorr != 1)
        {
            forAll(fluidRegions, i)
            {
                #include "setRegionFluidFields.H"
                #include "storeOldFluidFields.H"
            }
        }
        // --- PIMPLE loop
        for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
        {
            forAll(fluidRegions, i)
            {
                Info<< "\nSolving for fluid region "
                    << fluidRegions[i].name() << endl;
                #include "setRegionFluidFields.H"
                #include "readFluidMultiRegionPIMPLEControls.H"
                #include "solveFluid.H"
            }

            forAll(solidRegions, i)
            {
                Info<< "\nSolving for solid region "
                    << solidRegions[i].name() << endl;
                #include "setRegionSolidFields.H"
                #include "readSolidMultiRegionPIMPLEControls.H"
                #include "solveSolid.H"
            }
        }

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info << "End\n" << endl;

    return 0;
}
// ************************************************************************* //
```

Type `:q!` to exit the opened text in the terminal.

The purpose of each included header file [1] can be seen in their descriptions at the top of that file. The paths of the files can be found by the command (where `<filename>` is the name of that particular file):

```
find $WM_PROJECT_DIR -name <filename>
```

For example, to look for the path of the `turbulenceModel.H` included in the main source file, type the following command:

```
find $WM_PROJECT_DIR -name turbulenceModel.H
```

Then, four paths are shown up as below:

```
$FOAM_SRC/turbulenceModels/incompressible/turbulenceModel/lnInclude/turbulenceModel.H
$FOAM_SRC/turbulenceModels/incompressible/turbulenceModel/turbulenceModel.H
$FOAM_SRC/turbulenceModels/compressible/turbulenceModel/lnInclude/turbulenceModel.H
```

[1]Header file: the *.H files included before the main function

`$FOAM_SRC/turbulenceModels/compressible/turbulenceModel/turbulenceModel.H`

The lines containing ‘lnInclude’ are just linking to the other file with a similar path. We can see from above that in the OpenFOAM library, there are `turbulenceModel.H` files for both incompressible and compressible models according to their paths. To check which `turbulenceModel.H` is included, we can take a look at the `Make/options` file. Exit the opened text in the terminal by `:q!`, and type

```
vi ./Make/options
```

It is shown in the last line of `EXE_INC` in the text that the the following directory is included:

`-I$(LIB_SRC)/turbulenceModels/compressible/turbulenceModel`

We understand that the `turbulenceModel.H` file of the compressible model is included. Open the file by

```
vi  $FOAM_SRC/turbulenceModels/compressible/turbulenceModel/turbulenceModel.H
```

The description of the file shows that the purpose of the `turbulenceModel.H` is to declare ‘abstract base class for compressible turbulence models (RAS, LES and laminar)’. It includes the source files of `turbulenceModel.C`.

Above is an example of how to look for the information of each included file. The purposes/descriptions of the other included files in the `chtMultiRegionFoam.C` are provided as follows:

`fvCFD.H` – A standard file for finite volume method.

`basicPsiThermo.H` – To declare basic thermodynamic properties based on compressibility.

`turbulenceModel.H` – To declare and define abstract base class for compressible turbulence models (RAS, LES and laminar).

`fixedGradientFvPatchFields.H` – To make patch type as field type and declare the primitive field types, such as scalar, tensor, vector, etc.

`regionProperties.H` – To declare simple class to hold region information for coupled region simulations.

`compressibleCourantNo.H` – To calculate and output the mean and maximum Courant Numbers for the fluid regions.

`solidRegionDiffNo.H` –To calculate and output the mean and maximum Diffusion Numbers for the solid regions.

In the `main()` function, the following files are included before running the loop. These files are not proper header files; they only contain pieces of code that are inserted at each location.

`setRootCase.H` – To check the folder structure of the case.

`createTime.H` – To check runtime according to the controlDict and initiates time variables.

`createFluidMeshes.H` – To create fluid mesh for region(s).

`createSolidMeshes.H` – To create solid mesh for region(s).

`createFluidFields.H` – To create the fields for the fluid region: Reading fluid mesh thermophysical properties rho, kappa, U, phi, g, turbulence, DpDtFluid.

`createSolidFields.H` – To create the fields for the solid region: Reading solid mesh thermophysical properties rho, cp, kappa, T.

`initContinuityErrs.H` – To declare and initialise the cumulative continuity error.

`createTimeControls.H` – To read the control parameters used by setDeltaT.

`readSolidTimeControls.H` – To read the control parameters diffusion number (DiNum) used in the solid, lookup the 'maxDi' or use the default value.

`compressibleMultiRegionCourantNo.H` – To calculate and output the mean and maximum Courant Numbers for the fluid regions.

`solidRegionDiffusionNo.H` – To calculate the DiNum for all the solid regions.

`setInitialMultiRegionDeltaT.H` – To set the initial timestep for the chtMultiRegionFoam solver.

Then in the `while (runTime.run())` loop, initialising files are executed and some of them are executed again. Purposes of the files are described as follows:
`readTimeControls.H` – To read the control parameters used by setDeltaT.

`readSolidTimeControls.H` – To read the control parameters used in the solid.

`readPIMPLEControls.H` – To read the nOuterCorrectors in fvSolution(this parameter only defined for the fluid regions).

`compressibleMultiRegionCourantNo.H` – To calculate CoNum for fluid regions.

`solidRegionDiffusionNo.H` – To calculate DiNum for solid regions.

`setMultiRegionDeltaT.H` – To reset the time step to maintain a constant maximum courant number(CoNum) and diffusion Numbers(DiNum). The time step is reset according to the DiNum and CoNum calculated from compressibleMultiRegionCourantNo.H and solidRegionDiffusionNo.H.

3.2 The PIMPLE loop

The `chtMultiRegionFoam` solver uses the PIMPLE algorithm. PIMPLE algorithm is a combination of the pressure-implicit split-operator (PISO) and the semi-implicit method for pressure-linked equations (SIMPLE) algorithms. Most fluid dynamics solver applications in OpenFOAM use either the PISO, SIMPLE or the combined PIMPLE algorithm. These algorithms are iterative procedures for coupling equations for momentum and mass conservation, PISO and PIMPLE being used for transient problems and SIMPLE for steady-state. More explanation can be seen in section 4.5 of the OpenFOAM user guide [2].

The PIMPLE looping is controlled by the following input parameters:

- `nCorrectors`: it sets the number of times the algorithm solves the pressure equation and momentum corrector in each step; typically set to 2 or 3.
- `nNonOrthogonalCorrectors`: it specifies repeated solutions of the pressure equation and is used to update the explicit non-orthogonal correction of the Laplacian term $\nabla \cdot \frac{(1/A)}{\nabla P}$, described in section 4.4.4 of the OpenFOAM user guide [2]; typically set to 0 (particularly for steady-state) or 1.

- `nOuterCorrectors`: it enables looping over the entire system of equations within on time step, representing the total number of times the system is solved; must be larger than or equal to 1 and is typically set to 1, replicating the PISO algorithm.
- `momentumPredictor`: the looping algorithms optionally begins each step by solving the momentum equation: the so-called momentum predictor. This parameter is a switch that controls solving of the momentum predictor; typically set to `off` for some flows, including low Reynolds number and multiphase.

In the `chtMultiRegionFoam` solver, the PIMPLE looping control parameters are specified in the files of:
`./fluid/readFluidMultiRegionPIMPLEControls.H` and `./readPIMPLEcontrols.H`.
The `readFluidMultiRegionPIMPLEControls.H` file declares the following PIMPLE parameters:

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

    int nCorr(readInt(pimple.lookup("nCorrectors")));

    int nNonOrthCorr =
        pimple.lookupOrDefault<int>("nNonOrthogonalCorrectors", 0);

    bool momentumPredictor =
        pimple.lookupOrDefault<Switch>("momentumPredictor", true);
```

In `readPIMPLEcontrols.H`, `nOuterCorrectors` is declared:

```
    fvSolution solutionDict(runTime);

    const dictionary& pimple = solutionDict.subDict("PIMPLE");

    int nOuterCorr(readInt(pimple.lookup("nOuterCorrectors")));
```

Regarding the case settings, the PIMPLE control parameters are set both in the `system/fvSolution` and `system/fluidRegionName/fvSolution`, where `fluidRegionName` stands for the directory named by the fluid region(s). These two files set the same parameters for the PIMPLE loop, which means that there are (at least) two entries of the PIMPLE settings. The redundant setting in the file `system/fvSolution` may due to a lack of tidying up the code of the solver. A safe and simple way of fixing the setting is to give the same settings in both files.
An example of the PIMPLE loop setting in `chtMultiRegionFoam/multiRegionHeater/system/topAir/fvSolution` in the tutorial case of FOAM-extend-4.0 is presented in the box below:

```
PIMPLE
{
    momentumPredictor        off;
    nOuterCorrectors         1;
    nCorrectors              2;
    nNonOrthogonalCorrectors 1;
    pRefCell                 0;
    pRefValue                0;
}
```

Back to the main source file `chtMultiRegionFoam.C`, before the PIMPLE loop and after the `runTime++`:

```
        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        if (nOuterCorr != 1)
        {
            forAll(fluidRegions, i)
            {
                #include "setRegionFluidFields.H"
                #include "storeOldFluidFields.H"
            }
        }
```

The `#include "setRegionFluidFields.H"` is to set the mesh and the field values that are used in the calculation for the next time step. If `nOuterCorr != 1`, the same time step of the fluid domain will be calculated more than once; therefore, `#include "storeOldFluidFields.H"` is to retrieve the `pressure` and `rho` from the previous time step.

After that, the PIMPLE loop part is shown in the box below:

```
        // --- PIMPLE loop
        for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
        {
            forAll(fluidRegions, i)
            {
                Info<< "\nSolving for fluid region "
                    << fluidRegions[i].name() << endl;
                #include "setRegionFluidFields.H"
                #include "readFluidMultiRegionPIMPLEControls.H"
                #include "solveFluid.H"
            }

            forAll(solidRegions, i)
            {
                Info<< "\nSolving for solid region "
                    << solidRegions[i].name() << endl;
                #include "setRegionSolidFields.H"
                #include "readSolidMultiRegionPIMPLEControls.H"
                #include "solveFluid.H"
            }
        }
```

It is shown that, within one time step, the `fluidRegions` are calculated first. Then the `soildRegions` are computed based on the field data that are transferred from the `fluidRegions`. Both `fluidRegions` and `SolidRegions` are looping by the same time step, while they are converged by separate iterations within one time step.

In the subdirectories of `fluid/` and `solid/`, the fluid solver and solid solver are defined in the `solveFluid.H` and `solveSolid.H` files respectively. Those are described in the following sections.

3.3 Fluid solver

To get to know a solver, it is a good start to understand the known and unknown variables and what equations it solves first.

3.3.1 Unknown variables

The fluid equations solve density `rho`, velocity `U`, pressure `p` and the derivative of pressure `DpDt`. The thermal solver for the fluid region is essentially the same as for the solid part, although the variable solved for is the thermal energy h rather than the temperature `T`:

$$h = c_p \cdot dT \tag{3.1}$$

where c_p is the specific heat capacity at a constant pressure. The conventions differ between solid mechanics and fluid mechanics.

3.3.2 Equations

The fluid solver solves four equations:

```
if (oCorr == 0)
{
    #include "rhoEqn.H"  //Solve the continuity for density.
}

#include "UEqn.H" //Solve the momentum equation

#include "hEqn.H" //Solve the thermal energy

// --- PISO loop
for (int corr = 0; corr < nCorr; corr++)
{
    #include "pEqn.H" //solve pressure
}

turb.correct();

rho = thermo.rho();
```

The `rhoEqn.H` is included from the foam source library, not from the local solver directory. It can be viewed by:

```
vi $FOAM_SRC/finiteVolume/lnInclude/rhoEqn.H
```

The `rhoEqn` in the C++ format is as below:

```
\*---------------------------------------------------------------------------*/

{
    solve(fvm::ddt(rho) + fvc::div(phi));
}

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

It solves the density of the compressible fluid according to the following equation (`rhoEqn`):

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{3.2}$$

The other three equations `UEqn.H`, `hEqn.H` and `pEqn.H` can be seen in the `./fluid/` directory.
`UEqn.H` solves the momentum equation. In C++, the code of the momentum equation is shown below:

```
    tmp<fvVectorMatrix> UEqn
       (
           fvm::ddt(rho, U)
         + fvm::div(phi, U)
         + turb.divDevRhoReff()
       );

       UEqn().relax();

       if (momentumPredictor)
       {
            solve
            (
               UEqn()
            ==
               fvc::reconstruct
               (
                   fvc::interpolate(rho)*(g & mesh.Sf())
                 - fvc::snGrad(p)*mesh.magSf()
               )
           );
       }
```

It demonstrates the momentum equation:

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) + \nabla \cdot (\mu \nabla \mathbf{u}) + \nabla \cdot \left(\mu \left[(\nabla \mathbf{u})^T - \frac{2}{3} \mathrm{tr}(\nabla \mathbf{u})^T \mathbf{I} \right] \right) = \rho \mathbf{f} - \nabla p \tag{3.3}$$

where $\mathbf{u}$ denotes the velocity vector, $\mathbf{f}$ denotes the body force per unit mass acting on the fluid element. The function turb.divDevRhoReff() denotes the full viscous stress tensor in compressible flow:

$$\text{turb.divDevRhoReff()} = \nabla \cdot (\mu \nabla \mathbf{u}) + \nabla \cdot \left(\mu \left[(\nabla \mathbf{u})^T - \frac{2}{3} \mathrm{tr}(\nabla \mathbf{u})^T \mathbf{I} \right] \right) \tag{3.4}$$

where tr denotes the trace of the tensor in three dimensions.

`hEqn.H` solves the thermal energy h:

```
\\hEqn.H
{
    fvScalarMatrix hEqn
    (
        fvm::ddt(rho, h)
      + fvm::div(phi, h)
      - fvm::laplacian(turb.alphaEff(), h)
     ==
        DpDt
    );
    if (oCorr == nOuterCorr-1)
    {
        hEqn.relax();
        hEqn.solve(mesh.solutionDict().solver("hFinal"));
    }
    else
    {
        hEqn.relax();
        hEqn.solve();
    }

    thermo.correct();

    Info<< "Min/max T:" << min(thermo.T()).value() << ' '
        << max(thermo.T()).value() << endl;
}
```

The corresponding mathematical equation is:

$$\rho \cdot \frac{\partial h}{\partial t} + \nabla \cdot (\rho \mathbf{u} h) - \nabla(\alpha h) = \frac{Dp}{Dt} \tag{3.5}$$

where α is the laminar thermal diffusivity in the unit of [kg/m/s], which is defined in the file:

`$FOAM_SRC/thermophysicalModels/basic/basicThermo/basicThermo.H`

In heat transfer analysis, thermal diffusivity is the thermal conductivity divided by density and specific heat capacity at constant pressure [8]. It measures the rate of transfer of heat of a material from the hot side to the cold side.

$$\alpha = \frac{K}{\rho c_p} \tag{3.6}$$

where K denotes `Kappa` which is thermal conductivity $[\mathrm{W}/(\mathrm{m}\cdot\mathrm{K})]$, ρ is density $[\mathrm{kg}/\mathrm{m}^3]$, c_p is specific heat capacity $[\mathrm{J}/(\mathrm{kg}\cdot\mathrm{K})]$.

`pEqn.H` solves for pressure and updates the other variable fields.

```
\\pEqn.H part(1)
{
    bool closedVolume = p.needReference();

    rho = thermo.rho();

    volScalarField rUA = 1.0/UEqn().A();
    surfaceScalarField rhorUAf("(rho*(1|A(U)))", fvc::interpolate(rho*rUA));

    U = rUA*UEqn().H();

    surfaceScalarField phiU
    (
        fvc::interpolate(rho)
       *(
            (fvc::interpolate(U) & mesh.Sf())
          + fvc::ddtPhiCorr(rUA, rho, U, phi)
        )
    );

    phi = phiU + fvc::interpolate(rho)*(g & mesh.Sf())*rhorUAf;

    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvc::div(phi)
          - fvm::laplacian(rhorUAf, p)
        );
```

The part above interpolates the density (`rho`) and solves the `pEqn`.

```
\\pEqn.H part(2)
        if
        (
            oCorr == nOuterCorr-1
         && corr == nCorr - 1
         && nonOrth == nNonOrthCorr
        )
        {
            pEqn.solve(mesh.solutionDict().solver(p.name() + "Final"));
        }
        else
        {
            pEqn.solve(mesh.solutionDict().solver(p.name()));
        }

        if (nonOrth == nNonOrthCorr)
        {
            phi += pEqn.flux();
        }
    }

    // Correct velocity field
    U += rUA*fvc::reconstruct((phi - phiU)/rhorUAf);
    U.correctBoundaryConditions();

    // Update pressure substantive derivative
    DpDt = fvc::DDt(surfaceScalarField("phiU", phi/fvc::interpolate(rho)), p);

    // Solve continuity
    #include "rhoEqn.H"

    // Update continuity errors
    #include "compressibleContinuityErrors.H"

    // For closed-volume cases adjust the pressure and density levels
    // to obey overall mass continuity
    if (closedVolume)
    {
        p += (massIni - fvc::domainIntegrate(psi*p))
            /fvc::domainIntegrate(psi);
        rho = thermo.rho();
    }

    // Update thermal conductivity
    Kappa = thermoFluid[i].Cp()*turb.alphaEff();
}
```

The part above mainly updates pressure substantive derivative `DpDt` and the thermal conductivity `Kappa`.

3.4 Solid solver

3.4.1 Unknown variables

The unknown variable for the solid regions is the temperature T. Variables that are needed for solving the temperature equation are: thermal conductivity (`Kappa`), specific heat capacity (`cp`), and solid density (`rho`).

3.4.2 Equations

The solid solver solves the `TEqn` for solid regions.

```
\\solveSolid.H
{
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        tmp<fvScalarMatrix> TEqn
        (
            fvm::ddt(rho*cp, T)
          - fvm::laplacian(Kappa, T)
        );
        TEqn().relax();
        TEqn().solve();
    }

    Info<< "Min/max T:" << min(T) << ' ' << max(T) << endl;
}
```

The corresponding mathematical formula for the `TEqn` is:

$$\rho \cdot c_p \frac{\partial T}{\partial t} - \nabla(KT) = 0 \tag{3.7}$$

where K denotes `Kappa`.

3.5 Interface boundary coupling

In the solver, there are two alternatives for interface boundary conditions:

- solidWallHeatFluxTemperature: it allows introducing constant heat flux to a patch. It is important to notice that it has nothing to do with the boundary coupling. The function of this boundary condition is equal to the 'fixedGradient' boundary condition in OpenFOAM.
- solidWallMixedTemperatureCoupled: this is a mixed boundary condition for coupling the temperature at the interface, to be used by the conjugate heat transfer solver. Both sides use a mix of zero gradient and neighbour value.

The `solidWallMixedTemperatureCoupled` directory defines the mixed coupling boundary condition that the multi-regions are coupled via Dirichlet-Neumann partitioning strategy at the coupled interface:

- Dirichlet boundary condition:

$$\Gamma_1 = \Gamma_2 \tag{3.8}$$

  where the neighbour patches have the same field value.

- Neumann boundary condition:

$$\frac{\partial \Gamma_1}{\partial n_1} = \frac{\partial \Gamma_2}{\partial n_2} \tag{3.9}$$

  where the neighbour patches agree on the gradient of the field value.

The Dirichlet-Neumann partitioning strategy satisfies both Dirichlet and Neumann boundary conditions.

In the folder of `solidWallMixedTemperatureCoupled/` in the solver, a `solidWallMixedTemperatureCoupledFvPatchScalarField.H` file and a `solidWallMixedTemperatureCoupledFvPatchScalarField.C` file declares and defines the coupled boundary condition.

In the `solidWallMixedTemperatureCoupledFvPatchScalarField.H`, an example of how to use this boundary condition is given in the description part.

```
    Example usage:
        myInterfacePatchName
        {
            type                solidWallMixedTemperatureCoupled;
            neighbourFieldName  T;
            Kappa               Kappa;
            value               uniform 300;
        }
```

The boundary type name is specified in the file as `solidWallMixedTemperatureCoupled`.

```
public:
    //- Runtime type information
    TypeName("solidWallMixedTemperatureCoupled");
```

`solidWallMixedTemperatureCoupledFvPatchScalarField.C` file defines the mixed type of the boundary condition with the following three components: Line 33 to line 49 in `solidWallMixedTemperatureCoupledFvPatchScalarField.C`

```
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
Foam::solidWallMixedTemperatureCoupledFvPatchScalarField::
solidWallMixedTemperatureCoupledFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    mixedFvPatchScalarField(p, iF),
    neighbourFieldName_("undefined-neighbourFieldName"),
    KappaName_("undefined-Kappa")
{
    this->refValue() = 0.0;
    this->refGrad() = 0.0;
    this->valueFraction() = 1.0;
}
```

A comment in the `solidWallMixedTemperatureCoupledFvPatchScalarField.C` illustrates the mixed boundary condition: Line 199 to line 212 in `solidWallMixedTemperatureCoupledFvPatchScalarField.C`.

```
// We've got a degree of freedom in how to implement this in a mixed bc.
// (what gradient, what fixedValue and mixing coefficient)
// Two reasonable choices:
// 1. specify above temperature on one side (preferentially the high side)
//    and above gradient on the other. So this will switch between pure
//    fixedvalue and pure fixedgradient
// 2. specify gradient and temperature such that the equations are the
//    same on both sides. This leads to the choice of
//    - refGradient = zero gradient
//    - refValue = neighbour value
//    - mixFraction = nbrKappaDelta / (nbrKappaDelta + myKappaDelta())
```

The description denotes that user can choose the boundary condition between Dirichlet type (pure fixedvalue, agree on refValue), Neumann type (pure fixedgradient, zero gradient) or a mixed type agree on both temperature and its gradient. This is done by looking up for the keyword `refValue` as shown in the box below: Line 100 to line 113 in `solidWallMixedTemperatureCoupledFvPatchScalarField.C`.

```
    if (dict.found("refValue"))
    {
        // Full restart
        refValue() = scalarField("refValue", dict, p.size());
        refGrad() = scalarField("refGradient", dict, p.size());
        valueFraction() = scalarField("valueFraction", dict, p.size());
    }
    else
    {
        // Start from user entered data. Assume fixedValue.
        refValue() = *this;  //set the boundary condition in the time directory
        refGrad() = 0.0;
        valueFraction() = 1.0;
    }
```

For pressure mapping between the regions, which we have mentioned in Chapter 1, only the Dirichlet boundary condition needs to be specified because the pressure mapping is an agreement between the neighbour patches on the pressure value only, which means

$$p_1 = p_2 \tag{3.10}$$

The implementation of the pressure coupled boundary condition will be demonstrated in Chapter 6.

Chapter 4

Implementation of multiple time steps

4.1 Two approaches of modification

The goal of the multi-region loop modification is to introduce an extra time step for the solid regions in addition to the original time step for the fluid regions. The reason of such implementation was discussed in the Chapter 1: to improve the efficiency of the whole system when the time step requirements varies between different physical regions. From a physical point of view, in most of the cases, the time step needed for the solid regions can be much larger than needed for the fluid region while there is no displacement or solid deformation considered. For example, for the interaction between wave and soil, if no deformation or failure of the soil is considered, the solid region time step can be 10-100 times larger than the fluid region time step.

The loop modification with multiple time steps will prevent the solid solvers looping unnecessarily, thus improve the efficiency of the whole system. To introduce multiple time steps into the system, two ways of implementation were conducted and discussed in this report. Both methods have their own pros and cons.

4.1.1 The first approach

The first approach to modify the loop is to introduce a time step for the solid regions (`solidRegionDeltaT`) which is an integral multiple of the fluid region time step (`deltaT`), i.e.

$$\mathsf{solidRegionDeltaT} = n \cdot \mathsf{deltaT} \tag{4.1}$$

where n is an integer. In this way, we can specify the time step of the solid regions according to the solid region property and the fluid region property. The n times of the `deltaT` indicates that the solid regions will be solved once after every n times we solve the fluid regions.

However, for such implementation, an important thing needs to be done is that the switch of 'adjustTimeStep' for fluid regions must be turned off in the `controlDict` when setting a case. It means that we will not allow the system to prolong the time step of the fluid regions according to the Courant number. Otherwise, the `solidRegionDeltaT` will not remain the integral multiple of the fluid region `deltaT`, which will cause an error to the system. Details upon this point will be discussed in the next section.

The first approach of implementation enables a stable time step for the solid regions that can be specified by the user. When doing the case setting, only one extra variable (`solidRegionDeltaT`) needs to be specified. One shortcoming of such an implementation is that it might reduce the efficiency of the fluid region computations, since the 'adjustTimeStep' switch is turned off.

4.1.2 The second approach

The second approach will allow the adjustment of the time step (set 'adjustTimeStep yes' in the controlDict) for the fluid regions. A solid region time step reference value `solidRegionDeltaTRef` will be specified. This value is a reference that every time the runTime difference in equation 4.2 passes such a value, the solid regions will be solved, checked by

$$\mathsf{runTime(i)} - \mathsf{solidRegionRunTime(j\text{-}1)} > \mathsf{solidRegionDeltaTRef} \qquad (4.2)$$

If that happens, also the `solidRegionRunTime` is updated to prepare for the next check by equation 4.2, as

$$\mathsf{solidRegionRunTime(j)} = \mathsf{runTime(i)} \qquad (4.3)$$

For example, set the `solidRegionDeltaTRef = 0.01`, and the fluid region original `deltaT = 0.001`. Then, after running for a little while, let us assume current fluid region time step is at $i = 0.0095421$, where i is the time step index for solving the fluid region. Since i is less than 0.01, we haven't solved the solid regions yet. Initial `solidRegionRunTime` is at $j = 0$, where j is the time step index for solving the solid region. A detailed illustration from the time step i is presented as follows :

- time step i.

  Current time step is $i = 0.0095421$: it is smaller than `solidRegionDeltaTRef` 0.01, so the fluid regions will continually be solved for the next time step.

  The solid regions have not been solved yet, $j = 0$.

- time step $i + 1$.

  Time step adjusted according to Courant number, current time step is $i + 1 = 0.014511$: $(i+1)-j = 0.014511-0 > 0.01$. The time difference is larger than the `solidRegionDeltaTRef`.

  Therefore, the solid regions are solved at $j + 1 = 0.014511$.

- time step $i + 2$.

  Time step adjusted according to Courant number, current time step is $i + 2 = 0.019573$: the difference between current time step and the last time we solve the solid region is $(i + 2) - (j + 1) = 0.005062 < 0.01$. The time difference is smaller than the `solidRegionDeltaTRef`.

  Therefore, the solid regions are not solved at this time step.

- time step $i + 3$.

  current time step is $i + 3 = 0.0201234$, the difference between current time step and the last time we solve the solid region is $(i + 3) - (j + 1) = 0.024634 - 0.014511 = 0.010123 > 0.01$. The time difference is larger than the `solidRegionDeltaTRef`.

  The solid regions should be solved at this time step $j + 2 = 0.0201234$.

The main benefit of doing in this way is to allow the adjustment of the time step for the fluid regions. However, it is important to notice that, this approach of implementation is based on the assumption that: the fluid region `deltaT` will not be adjusted too much according to the Courant number, so that after it is subtracted by the last `solidRegionRunTime`, the time difference will not be too large for solving the solid regions.

4.2 The first approach of modification

4.2.1 Modify the solver

The first step of the loop modification is to introduce an extra time step for the solid regions. Then the PIMPLE loop is to be modified to achieve looping upon different time steps. The process of implementing such modification is seen as follows:

1. Rename the solver as `chtMultiRegionMultiDeltaTFoam`. We want to classify our new solver as a multiphysics solver, so a `multiphysics/` directory is created for it.

   ```
   mkdir -p $WM_PROJECT_USER_DIR/applications/solvers/multiphysics/
   cd $WM_PROJECT_USER_DIR/applications/solvers/multiphysics/
   cp -r $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam .
   mv chtMultiRegionFoam chtMultiRegionMultiDeltaTFoam
   cd chtMultiRegionMultiDeltaTFoam
   mv chtMultiRegionFoam.C chtMultiRegionMultiDeltaTFoam.C
   ```

   Make sure that the binary file ends up in the user directory, following modifications need to be implemented in the `Make/files`,

   ```
   sed -i s/chtMultiRegionFoam/chtMultiRegionMultiDeltaTFoam/g Make/files
   sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
   ```

2. Now we need to create a new variable to control the solid region time step, let us call it `solidRegionDeltaT`. The creating of this variable should be implemented in `solid/readSolidTimeControls.H`. Add the following commands after the declaration of scalar `maxDi` in the `readSolidTimeControls.H` file:

   ```
   scalar solidRegionDeltaT =
   runTime.controlDict().lookupOrDefault<scalar>("solidRegionDeltaT",
   runTime.deltaT().value());
   ```

   The command above indicates that, if the `solidRegionDeltaT` is not set in the `controlDict` file in the case/system directory, it will be set equal to the original fluid region `deltaT`.

3. To our case, the `solidRegionDeltaT` should be set as an integral multiple of the fluid region `deltaT`. Therefore, a checking sentence should be written to avoid wrong user setting. The following commands need to be added after step 2, to see if the `solidRegionDeltaT` is integral multiple of the `deltaT`. If not, the program will stop running instead of generating garbage results.

   ```
   if(fmod(solidRegionDeltaT, runTime.deltaT().value()) > SMALL)
   {
       Info<<"Error: solid region delta T is not an integral multiple of the \\
       original delta T." << endl;
       return 0;
   }
       Info << "Solid region deltaT set to: "
            << solidRegionDeltaT
            << " s."
            << endl;
   ```

   See above, an Info line is also added for the convenience of future debugging. It is worthwhile to mention that `fmod()` is a C++ function declared in math.h/cmath.h for calculating the remainder between float or double type of numbers. The % operator in C++ only works for calculating the remainder between integers.

4. Based on the modification of step 3, before the first command line of the `chtMultiRegionMultiDeltaTFoam.C`, Add

```
//for the computation of the remainder between two floating numbers.
#include <math.h>
```

5. Modify the PIMPLE loop. This is the main step of the modification. The PIMPLE loop in the `chtMultiRegionMultiDeltaTFoam.C` needs to be replaced. The following is the original loop in the `chtMultiRegionFoam.C`.

```
        // --- PIMPLE loop
          for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
          {
              forAll(fluidRegions, i)
              {
                  Info<< "\nSolving for fluid region "
                      << fluidRegions[i].name() << endl;
                  #include "setRegionFluidFields.H"
                  #include "readFluidMultiRegionPIMPLEControls.H"
                  #include "solveFluid.H"
              }
              forAll(solidRegions, i)
              {
                  Info<< "\nSolving for solid region "
                      << solidRegions[i].name() << endl;
                  #include "setRegionSolidFields.H"
                  #include "readSolidMultiRegionPIMPLEControls.H"
                  #include "solveSolid.H"
              }
          }
```

The original loop code in the box above needs to be replaced by the modified code below:

```
        // --- PIMPLE loop
         for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
         {
             forAll(fluidRegions, i)
             {
                 Info<< "\nSolving for fluid region "
                     << fluidRegions[i].name() << endl;
                 #include "setRegionFluidFields.H"
                 //Read correctors for the fluid regions.
                 #include "readFluidMultiRegionPIMPLEControls.H"
                 #include "solveFluid.H"
             }
             //To write the time variables for debugging purpose.
             Info << "If statement remainder: "
                  << fmod(runTime.value(), solidRegionDeltaT)
                  << nl
                  << "RunTime, solidDeltaT:" << nl
                  << runTime.value() << ", " << solidRegionDeltaT
                  << endl;
             //To check if the solid solver is to be run.
             if((fmod(runTime.value(), solidRegionDeltaT) < SMALL)||
                  << (fmod(runTime.value(), solidRegionDeltaT)
                  <<  - solidRegionDeltaT < SMALL))
             {
                 forAll(solidRegions, i)
                 {
                     Info<< "\nSolving for solid region "
                        << solidRegions[i].name() << endl;
                     #include "setRegionSolidFields.H"
                     #include "readSolidMultiRegionPIMPLEControls.H"
                     #include "solveSolid.H"
                 }
              }
         }
```

It is important to be aware of using the `fmod()` in C++. The float number calculation has accuracy limitation. For example, it may return a 0.1 when we calculate the remainder between 1.5 and 0.1. The IF statement

```
if((fmod(runTime.value(), solidRegionDeltaT) < SMALL)||
 << (fmod(runTime.value(), solidRegionDeltaT)
 <<  - solidRegionDeltaT < SMALL))
```

is written to do multiple check in order to avoid C++ floating-point arithmetic error.

6. Compile the new solver.

```
cd $WM_PROJECT_USER_DIR/applications/solvers/multiphysics/
cd chtMultiRegionMultiDeltaTFoam/
wmake
```

Then type the new solver name to check if it is compiled completely.

4.2.2 Test the solver

Now, the new solver is ready to be applied. To test the new solver, we will do some modification on the settings of a tutorial case, and then run the new solver. Copy the tutorial chtMultiRegion-Foam/multiRegionHeater and rename it.

```
mkdir -p $WM_PROJECT_USER_DIR/run/tutorials/chtMultiRegionFoam
cd $WM_PROJECT_USER_DIR/run/tutorials/chtMultiRegionFoam
cp -r $FOAM_TUTORIALS/heatTransfer/chtMultiRegionFoam/multiRegionHeater .
mv multiRegionHeater multiRegionHeaterMultiDeltaT
cd multiRegionHeaterMultiDeltaT
```

In the system/controlDict, add the new created variable `solidRegionDeltaT` and set it to 0.0015; (deltaT is 0.001, this setting is to check if the errors can be output.) Insert the following line into the system/controlDict file.

```
solidRegionDeltaT  0.0015;
```

In the Allrun file, change the last command 'runApplicationAndReportOnError chtMultiRegion-Foam' to 'runApplicationAndReportOnError chtMultiRegionMultiDeltaTFoam'.

```
sed -i s/chtMultiRegionFoam/chtMultiRegionMultiDeltaTFoam/g Allrun
```

Then, type

```
./Allrun
```

The program will crash very fast after running and report errors. Then in the `log.chtMultiRegionMultiDeltaTFoam` file, an error can be seen:

```
Error: solid region delta T is not an integral multiple of the original delta T.
```

This error is just as what we expected due to our setting!

Then we will try to run a successful case. In controlDict, change 'solidRegionDeltaT 0.0015' to 'solidRegionDeltaT 0.01' (10 times of the deltaT); then, run the solver by

```
sed -i 's/solidRegionDeltaT 0.0015/solidRegionDeltaT 0.01/g' system/controlDict
./Allclean
./Allrun
```

The case stopped running at 0.001 second and the same error occurred again. This error is not what we expected:

```
Error: solid region delta T is not an integral multiple of the original delta T.
```

Check the `log.chtMultiRegionMultiDeltaTFoam` file, we can see that after adjustment of the time step, the fluid region `deltaT` becomes 0.1 rather than keeping the original value 0.001. Therefore, we understand that the `solidRegionDeltaT` is once again not the multiple of the `deltaT`.

To solve this error, we need to set the 'adjustTimeStep' in the controlDict from 'yes' to 'no', this is very important.

```
sed -i 's/adjustTimeStep yes/adjustTimeStep no/g' system/controlDict
./Allclean
./Allrun
```

Then after a longer while, the case will finish running successfully with a stable time step. It is shown in the log.chtMultiRegionMultiDeltaTFoam file that the solid regions are not solved until the runtime is 0.01s. After every 0.01s, the solid regions are to be solved at 0.02s, 0.03s, 0.04s, etc.

```
Time = 0.01

Solving for fluid region bottomAir...

Solving for fluid region topAir...

If statement remainder: 1.734723e-18
RunTime, solidDeltaT:
0.01, 0.01

Solving for solid region heater...

Solving for solid region leftSolid...

Solving for solid region rightSolid...
```

The box above shows that at 0.01s, the solid regions were run after the remainder was checked to be zero. Information regarding the calculation outputs were replaced by the ellipsis mark for a better view of the structure. Note that, in C++ the remainder is calculated by floating point computation with limited accuracy, the small number 1.734723e-18 is regarded as zero. Till now, the first approach of loop modification was implemented and verified.

4.3 The second approach of modification

4.3.1 Modify the solver

In the second approach, the initial steps of the modification are similar to the first approach. The detailed process to implement such modification is presented as follows:

1. Give the name to the new solver as `chtMultiRegionMultiDeltaTRefFoam`.

   ```
   cd $WM_PROJECT_USER_DIR/applications/solvers/multiphysics/

   cp -r $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam .

   mv chtMultiRegionFoam chtMultiRegionMultiDeltaTRefFoam

   cd chtMultiRegionMultiDeltaTRefFoam

   mv chtMultiRegionFoam.C chtMultiRegionMultiDeltaTRefFoam.C
   ```

   Make sure that the binary file ends up in the user directory, following modifications need to be implemented in the make/files,

   ```
   sed -i s/chtMultiRegionFoam/chtMultiRegionMultiDeltaTRefFoam/g Make/files
   sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
   ```

2. Now two new variables need to be introduced. `solidRegionDeltaTRef` and `solidRegionRunTime`. The latter one is used to store the last time step of solving the solid regions. The creation of these variables should be done in `solid/readSolidTimeControls.H`. Add the following commands after the declaration of scalar `maxDi`:

```
scalar solidRegionDeltaTRef = runTime.controlDict().
lookupOrDefault<scalar>("solidRegionDeltaTRef", runTime.deltaT().value());

scalar solidRegionRunTime;
```

   Then, an Info statement was added to provide setting information in the running log file.

```
    Info << "solidRegionDeltaTRef set to: "

         << solidRegionDeltaTRef

         << " s."

         << endl;
```

   Now, we will move to the modification of the main source file `chtMultiRegionMultiDeltaTRefFoam.C`. Before the line 'while (runTime.run())', add

```
solidRegionRunTime=0.0;
```

   This is to set the initial value of the solid region run time. The value of `solidRegionRunTime` will be updated each time when the solid region is run. Then, before 'runTime.write();', replace the PIMPLE loop by the following:

```
        // --- PIMPLE loop

         for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
         {
             forAll(fluidRegions, i)
             {
                 Info<< "\nSolving for fluid region "
                     << fluidRegions[i].name() << endl;
                 #include "setRegionFluidFields.H"
                 #include "readFluidMultiRegionPIMPLEControls.H"
                 #include "solveFluid.H"
             }

             //To write the time step variables in the log file.
             Info << "RunTime-solidRegionRunTime: "
                  << runTime.value()-solidRegionRunTime
                  << nl
                  << "RunTime, solidRegionRunTime, solidRegionDeltaTRef:"
                  << nl
                  << runTime.value() << ", " << solidRegionRunTime
                  << ", " << solidRegionDeltaTRef
                  << endl;

             //To check if the solid solver is to be run:
             //if the difference between current runTime and the last
             //solidRegionRunTime is larger than solidRegionDeltaTRef

             if(runTime.value()-solidRegionRunTime > solidRegionDeltaTRef)
             {
                 //set the current runTime as solidRegionRunTime  and
                 //store it in the variable for the next comparison
                 solidRegionRunTime = runTime.value();
                 forAll(solidRegions, i)
                {
                     Info<< "\nSolving for solid region "
                        << solidRegions[i].name() << endl;
                     #include "setRegionSolidFields.H"
                     #include "readSolidMultiRegionPIMPLEControls.H"
                     #include "solveSolid.H"
                 }
             }
         }
```

3. Run `wmake` in the top-level of the new solver, and then type the new solver name to check if it is compiled completely.

4.3.2 Test the solver

The commands below are to test the solver. Copy the tutorial case `chtMultiRegionFoam/multiRegionHeater` into run directory and rename it as `multiRegionHeaterMultiDeltaTRef`.

```
cd $WM_PROJECT_USER_DIR/run/tutorials/chtMultiRegionFoam
cp -r $FOAM_TUTORIALS/heatTransfer/chtMultiRegionFoam/multiRegionHeater .
mv multiRegionHeater multiRegionHeaterMultiDeltaTRef
cd multiRegionHeaterMultiDeltaTRef
```

In the `system/controlDict`, add the new variable `solidRegionDeltaTRef` and set it to 0.01. Add the following line in to the `controlDict` file:

```
solidRegionDeltaTRef 0.01;
```

In the Allrun file, change the last command 'runApplicationAndReportOnError chtMultiRegionFoam' to 'runApplicationAndReportOnError chtMultiRegionMultiDeltaTRefFoam'.

```
sed -i s/chtMultiRegionFoam/chtMultiRegionMultiDeltaTRefFoam/g Allrun
```

Then, type

```
./Allrun
```

The case should be running faster with 'adjustTimeStep' than without. Check the `log.chtMultiRegionMultiDeltaTRefFoam` file. It shows that, the solid regions were run every time when the difference between the `runTime` and `solidRegionRunTime` is larger than 0.01. For example, at the first time step,

```
deltaT = 0.1
Time = 0.101

Solving for fluid region bottomAir...

Solving for fluid region topAir...

RunTime, solidRegionRunTime, solidRegionDeltaTRef:
0.101, 0, 0.01
RunTime-solidRegionRunTime: 0.101

Solving for solid region heater...

Solving for solid region leftSolid...

Solving for solid region rightSolid...
```

Calculation results were replaced by the symbol of ... for a better view of the structure. It shows that at the first time step, the `deltaT` was adjusted to 0.101s, therefore, `RunTime` minus `solidRegionRunTime` was 0.101, which was larger than `solidRegionDeltaTRef`(0.01), so the solid regions were computed at this time step.

At the next time step,

```
deltaT = 1.061063e-05
Time = 0.101011

Solving for fluid region bottomAir...

Solving for fluid region topAir...

RunTime, solidRegionRunTime, solidRegionDeltaTRef:
0.1010106, 0.101, 0.01
RunTime-solidRegionRunTime: 1.061063e-05
```

It is shown the difference between current run time and the last time solving the solid regions is only 1.061063e-05, which is smaller than 0.01, therefore the solid regions will not be solved at this time step. The fluid regions will be continually solved at the next time step.

Until now, the implementation of the multiple `deltaT`s for fluid regions and solid regions have been done. In the next chapter, the graphical results will be compared between the modified solver and the original solver based on the tutorial case `multiRegionHeater` in FOAM-extend-4.0.

Chapter 5

Visualization and comparison

5.1 Visualization

To the author's knowledge, the `multiRegionHeater` tutorial case solved by chtMultiRegionFoam does not work very well with paraview visualization in FOAM-extend-4.0. The current solution for this problem is to use paraview in OpenFOAM-4.0 instead. After running the `./Allrun` script, open a new terminal, and source OpenFOAM-4.0 by:

```
 OF4x
```

or other alias depending on the user setting in their machines.
Then create files for each region for paraview post-processing by:

```
touch multiRegionHeater{bottomAir}.OpenFOAM

touch multiRegionHeater{heater}.OpenFOAM

touch multiRegionHeater{leftSolid}.OpenFOAM

touch multiRegionHeater{rightSolid}.OpenFOAM

touch multiRegionHeater{topAir}.OpenFOAM
```

Then

```
paraview
```

In the paraview window, select File and Open. Then in the window find the option 'Files of type', drag the list to the end and select All files(*). Choose:

```
multiRegionHeater{bottomAir}.OpenFOAM
multiRegionHeater{heater}.OpenFOAM
multiRegionHeater{leftSolid}.OpenFOAM
multiRegionHeater{rightSolid}.OpenFOAM
multiRegionHeater{topAir}.OpenFOAM
```

Press OK and in the main window, press the green button Apply for each region. Note that for the solid regions, only temperature T should be selected as a volume field, otherwise, paraview may crash due to wrong volume fields selection.

In the window of paraview, the mesh of the multiRegionHeater is shown in Figure 5.1.

The blue meshes form the top air region, the red meshes form the bottom air region. In the middle, the white meshes are the solid regions including left solid, heater and right solid.

Figure 5.2 and 5.3 show the density and temperature distribution of the topAir at 180s for the original tutorial case.

Figure 5.4 present the results of the modified cases: `chtMultiRegionMultiDeltaT` and `chtMultiRegionMultiDeltaTRef`. The density distributions `rho` of the topAir at 180s are exactly the same between the two modified cases and the original case, therefore, only one figure is presented here. It indicates that the modification of the loop with multiple time steps does not change the theory and the final results.

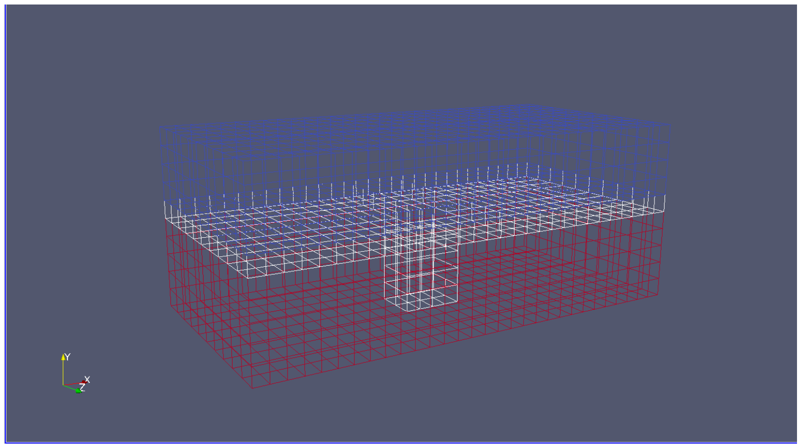

Figure 5.1: Meshes of the multiRegionHeater in paraview.

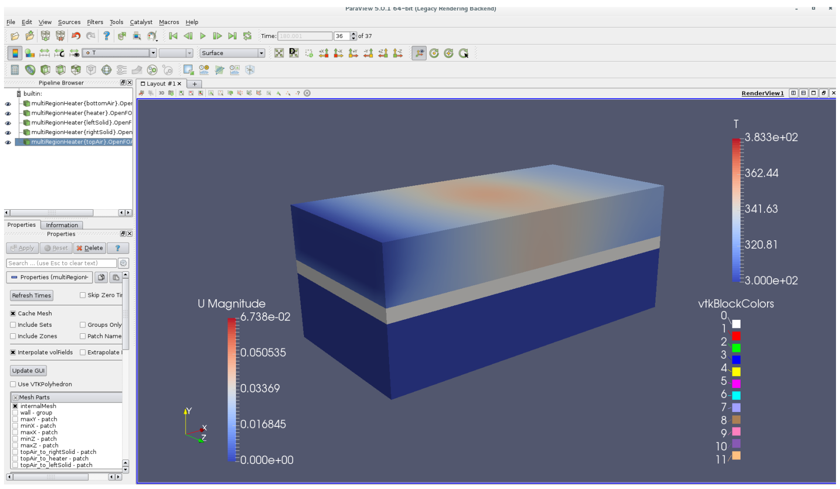

Figure 5.2: Temperature distribution of the topAir at 180s for MultiRegionHeater.

5.2 Comparison

Figures 5.5 and 5.6 compare the changes of maximum temperature in the air and solid regions along time between three cases: the original `multiRegionHeater` tutorial case, the `multiRegionHeaterMultiDeltaT` case calculated by the first approach (approach 1) and the `multiRegionHeaterMultiDeltaTRef` case calculated by the second approach (approach 2).

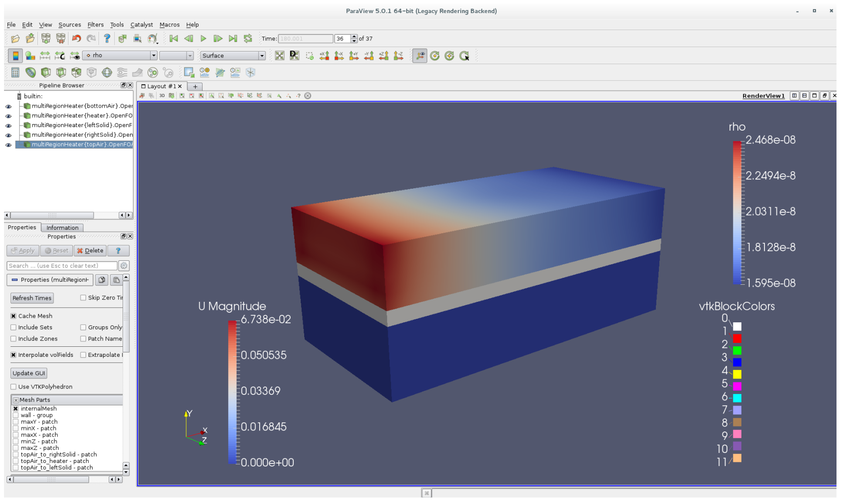
ParaView 5.0.1 64-bit (Legacy Rendering Backend)
Pipeline Browser
multiRegionHeater{bottomAir}.OpenFOAM
multiRegionHeater{heater}.OpenFOAM
multiRegionHeater{leftSolid}.OpenFOAM
multiRegionHeater{rightSolid}.OpenFOAM
multiRegionHeater{topAir}.OpenFOAM
rho
2.468e-08
2.2494e-8
2.0311e-8
1.8128e-8
1.595e-08
U Magnitude
6.738e-02
0.050535
0.03369
0.016845
0.000e+00
vtkBlockColors
0
1
2
3
4
5
6
7
8
9
10
11

Figure 5.3: Density distribution of the topAir at 180s of the MultiRegionHeater.

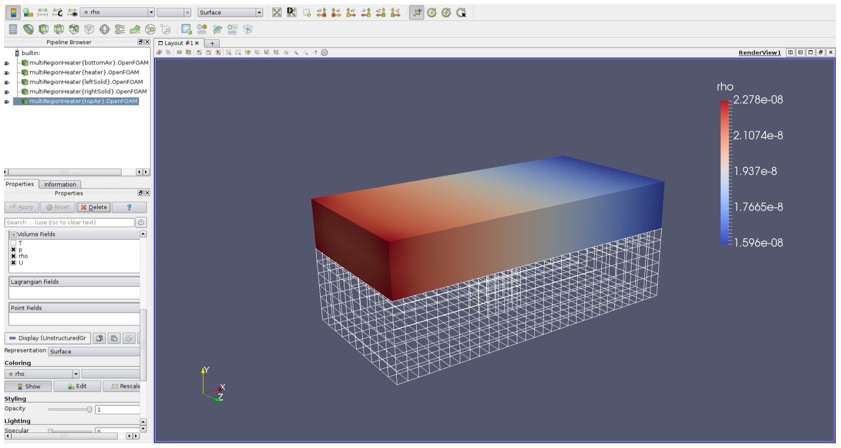
Pipeline Browser
multiRegionHeater{bottomAir}.OpenFOAM
multiRegionHeater{heater}.OpenFOAM
multiRegionHeater{leftSolid}.OpenFOAM
multiRegionHeater{rightSolid}.OpenFOAM
multiRegionHeater{topAir}.OpenFOAM
rho
2.278e-08
2.1074e-8
1.937e-8
1.7665e-8
1.596e-08

Figure 5.4: Density distribution of the topAir at 180s from modified cases: MultiRegionHeather-MultiDeltaT and MultiRegionHeatherMultiDeltaTRef (Results are exactly the same between two modified cases and the original case).

- Approach 1 provides exact the same result as the original case. It is because that in approach 1, the time step of the solid regions is set as 0.01, which is small enough to achieve a very accurate result. However, since the adjustment of time step is not allowed in approach 1, the computation time is longer than the approach 2. The computation time can be reduced by increasing the time steps of the fluid regions and solid regions.
- Approach 2 provides a relatively accurate result. It has a small variance compared to the original tutorial case in the initial time steps; after some time, it reaches the same result as the others.
- Three curves merges at the end. The computational time of approach 2 (`chtMultiRegionMultiDeltaTRefFoam`) is the shortest among the three.

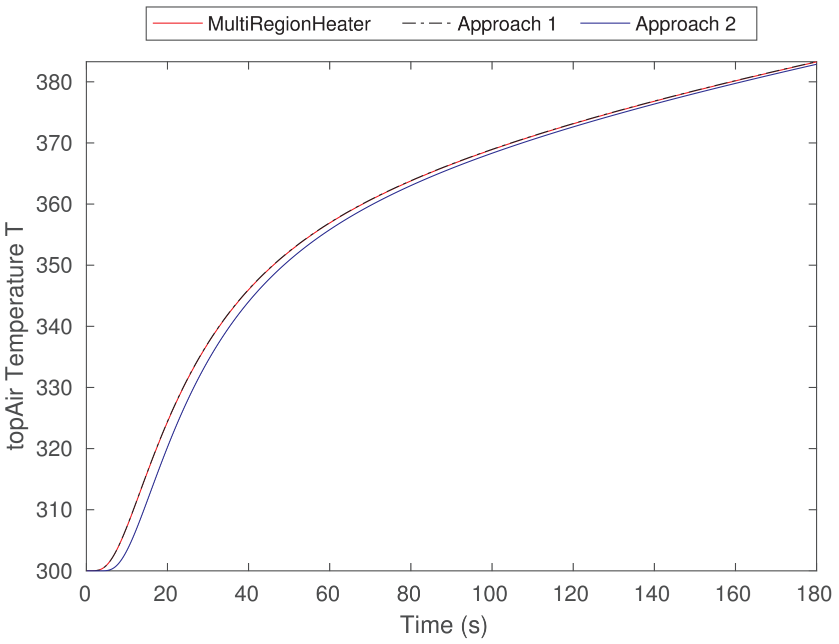

Figure 5.5: Comparison of maximum temperature of the topAir along time.

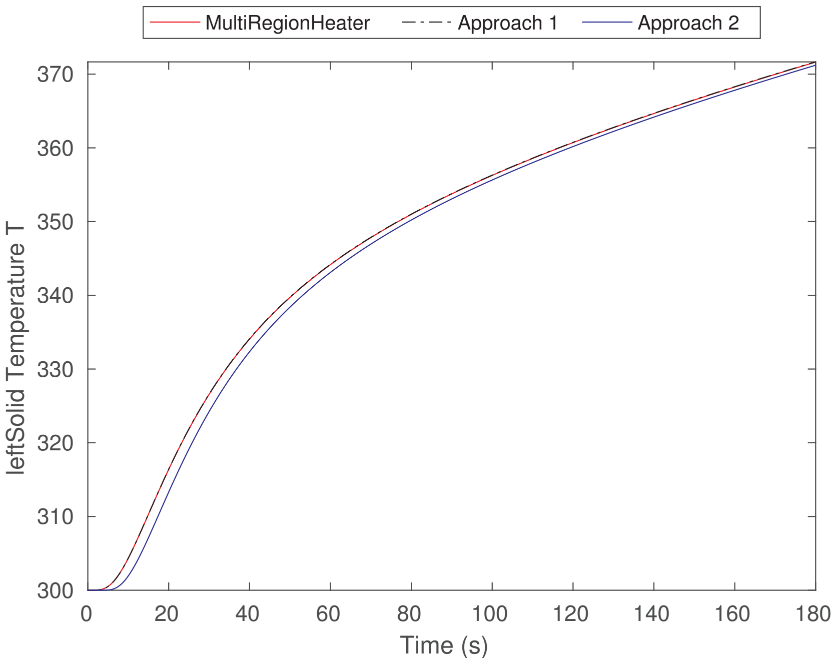

Figure 5.6: Comparison of maximum temperature of the leftSolid along time.

Chapter 6

Implementation of the pressure coupled boundary condition

6.1 Boundary condition implementation

As we mentioned before, a pressure coupled boundary condition for multi-physics solver was developed. It was tested by a case but has not been verified at this stage. The steps of implementing the pressure coupled boundary condition are presented in this chapter.

Get into the top-level directory of the `chtMultiRegionFoam` solver. In the folder of `derivedFvPatchFields`, create a new boundary condition folder named `solidWallPressureCoupled`. This step can be done by copying from the `solidWallMixedTemperatureCoupled` and modifying upon it. We would like to call the new solver `multiRegionPressureFoam`.

Commands are as follows:

```
f40NR
cd $WM_PROJECT_USER_DIR/applications/solvers/multiphysics/
cp -r $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam .
mv chtMultiRegionFoam multiRegionPressureFoam
cd multiRegionPressureFoam/derivedFvPatchFields
mkdir solidWallPressureCoupled
cp solidWallMixedTemperatureCoupled/* solidWallPressureCoupled
```

Rename the files:

```
cd solidWallPressureCoupled
mv *Temperature*.C solidWallPressureCoupledFvPatchScalarField.C
mv *Temperature*.H solidWallPressureCoupledFvPatchScalarField.H
```

Although for pressure coupling only the Dirichlet type boundary condition is needed, the Neumann type and Dirichlet-Neumann partitioning strategy are still kept in case of future extension.

In `solidWallPressureCoupledFvPatchScalarField.H` and `solidWallPressureCoupledFvPatchScalarField.C` files, replace all the 'MixedTemperature' by 'Pressure'.

```
sed -i s/MixedTemperature/Pressure/g *.H
sed -i s/MixedTemperature/Pressure/g *.C
```

In `chtMultiRegionFoam`, the temperature coupling is conducted by mapping the variable `Kappa`. The thermal conductivity `Kappa` is defined as a scalar in the solver, the same as pressure.

In the declaration file `solidWallPressureCoupledFvPatchScalarField.H`, pressure field: pressure and pressure gradient are used as mapping variables. Corresponding revision were implemented in the source file `solidWallPressureCoupledFvPatchScalarField.C`. After some detailed revision, a new boundary condition for pressure coupling was created.

The code of the new boundary condition `solidWallPressureCoupled` can be found in the attachment. Since the code for such boundary condition is relatively long, it will not be presented in this report.

Before compiling the new boundary condition in the solver, following steps need to be implemented. Add the following line into Make/files:

```
derivedFvPatchFields/solidWallPressureCoupled/solidWallPressureCoupledFvPatchScalarField.C
```

Do not forget to do the following routines before compiling the solver by `wmake`.

```
cd $WM_PROJECT_USER_DIR/applications/solvers/multiphysics/
cd multiRegionPressureFoam
mv chtMultiRegionFoam.C multiRegionPressureFoam.C
sed -i s/chtMultiRegionFoam/multiRegionPressureFoam/g Make/files
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
```

The solver with the newly implemented boundary condition was renamed to `multiRegionPressureFoam`. After compiling, it can be tested by a case: a small change was made in the case `multiRegionHeater` by changing the pressure boundary condition of the `bottomAir` in `/system/bottomAir/changeDictionaryDict`. Replace the following condition in the field `p`:

```
            bottomAir_to_rightSolid
            {
                type            buoyantPressure;
                value           uniform 1e+05;
            }
```

by

```
            bottomAir_to_rightSolid
            {
                type            solidWallPressureCoupled;
                neighbourRegionName rightSolid;
                neighbourPatchName rightSolid_to_bottomAir;
                neighbourFieldName p;
                p           p;
                value           uniform 1e+05;
            }
```

Then in the `Allrun`, change the solver name to `multiRegionPressureFoam`.

However, after running, the case will stop with an error. It complains that:

```
request for volScalarField p from objectRegistry rightSolid failed
 available objects of type volScalarField are
 5
 (
 cp
 Kappa
 rho
 T
 rhosCps
 )
```

This is because a pressure field was not created for the solid regions. A volume field `p` needs to be created in order to use the pressure coupled boundary condition.

In `solid/createSolidFields.H`, add the following line in the first part of initialising the solid field pointer lists:

```
PtrList<volScalarField> ps(solidRegions.size());
```

In the solid filed pointer lists, add:

```
        Info<< "    Adding to ps\n" << endl;
         ps.set
         (
             i,
             new volScalarField
             (
                 IOobject
                 (
                     "p",
                     runTime.timeName(),
                     solidRegions[i],
                     IOobject::MUST_READ,
                     IOobject::AUTO_WRITE
                 ),
                 solidRegions[i]
             )
         );
```

In solid/setRegionSolidFields.H, add the follwing line at the end,

```
volScalarField& p = ps[i];
```

Then compile the solver again by `wmake`.

6.2 Test the boundary condition

Copy the tutorial case `multiRegionHeater` and rename it as `multiRegionHeaterPressureCoupled`. Following changes were implemented. Since we have created a `p` field for the solid regions, the boundary condition of the pressure needs to be specified in all the solid regions.

1. Add the boundary conditions of `p` in the `system/solidRegions/changeDict`. For each patch, specify the interface boundary condition of `p` as follows:

   ```
                       type            buoyantPressure;
                       value           uniform 1e+05;
   ```

   For example, in the `leftSolid` region, the boundary condition of `p` as follows should be added:

```
p
{
    boundaryField
    {
        minX
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }
        maxX
        {
            type            waveTransmissive;
            field           pd;
            U               U;
            phi             phi;
            rho             rho;
            psi             psi;
            gamma           1.4;    // cp/cv
            fieldInf        0;
            lInf            0.40;   // double length of domain
            inletOutlet     off;
            correctSupercritical off;
            value           uniform 1e+05;
        }
        minY
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }
        minZ
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }
        maxZ
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }

        leftSolid_to_bottomAir
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }
        leftSolid_to_heater
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }
        leftSolid_to_topAir
        {
            type            buoyantPressure;
            value           uniform 1e+05;
        }
    }
}
```

2. To test the pressure coupled boundary condition, do following changes of `p` in `system/bottomAir/changeDict`:

```
            bottomAir_to_rightSolid
            {
                type            solidWallPressureCoupled;
                neighbourRegionName rightSolid;
                neighbourPatchName rightSolid_to_bottomAir;
                neighbourFieldName p;
                p           p;
                value           uniform 1e+05;
            }
```

   and in system/rightSolid/changeDict

```
            rightSolid_to_bottomAir
            {
                type            solidWallPressureCoupled;
                neighbourRegionName bottomAir;
                neighbourPatchName bottomAir_to_rightSolid;
                neighbourFieldName p;
                p           p;
                value           uniform 1e+05;
              }
```

3. Then in the top-level directory of the case, do

```
    ./Allclean
    ./Allrun
```

Now the case is able to run with the use of `solidWallPressureCoupled` boundary condition. After running, we can check a time step in the case directory, for example:

```
ls 150.001/leftSolid/
```

It is shown that there are five files including: `cp`, `Kappa`, `rho`, `T` and `p`, instead of four files `cp`, `Kappa`, `rho`, `T` in the original case. A `p` file is created in the solid region time directories, but without any calculated values. This is because that current solid solver only solves temperature `T`.

Now the boundary condition has been successfully created. To verify such boundary condition, the pressure field need to be computed and output by the solid solver.

However, the verification of such boundary condition has not been done at this stage. The solid solver needs to be modified in order to solve the pressure/stress to verify the coupled pressure data. This will leave to the future work.

Chapter 7

Summary and future work

7.1 Summary

The `chtMultiRegionFoam` solver has been studied and modified in this report. The structure of the `chtMultiRegionFoam` solver was discussed elaborately. Two approaches of loop modifications were implemented and verified. A new boundary condition for pressure coupling was implemented.

Here are some basic notes for doing solver modifications:

- Firstly, when modifying a solver, it is important to copy and rename the original solver at the first step. It is a good habit to do the implementation of the new solver in the user directory and make sure that the binary file ends up in the user directory.
- Secondly, when doing the modification, it is good to use Info statement to write the newly defined variable and other relevant information into the log file. When the results turn to be wrong or the case does not run properly, the information in the log file can be tracked for the debugging purpose.
- Finally, after modification, case study needs be to performed for verifications.

7.2 Future work

The modification in this report is an initial step for a multi-physics solver. More work needs to be done such as the verification of the pressure coupled boundary condition, the solid solver modification in order to compute the displacement and stress and the fluid solver modification in order to solve the incompressible two-phase fluids.

Study questions

1. Why does the author choose to do the modification based on the `chtMultiRegionFoam` solver for his/her purpose?
2. In the tutorial case `multiRegionHeater`, what is the function of the `changeDictionaryDict` file? Is it required for running the solver?
3. In the case setup of the tutorial case `multiRegionHeater`, why there are `fvSolution` files in both `system` directory and `system/fluidRegionName/` directory? Are they redundant?
4. In which file has the coupled boundary condition been defined in the `chtMultiRegionFoam` solver?
5. What modifications have been done for the new solvers?
6. What is the main difference between two approaches of implementing multiple time steps? What are the pros and cons for each approach?
7. What is the reason for using multiple time steps in the multi-physics solver?

Bibliography

[1] Nilsson, Håkan. *Install FOAM-extend-4.0* `http://pingpong.chalmers.se/public/courseId/7056/lang-en/publicPage.do?item=3112079`, 2016

[2] OpenCFD. *OpenFOAM user guide version 4.0*, OpenFOAM Foundation, 2016.

[3] Moradnia, P. *A description of how to do Conjugate Heat Transfer in OpenFOAM*, CFD with open source software, 2008.

[4] Järvstråt, N. *Adding electric conduction and Joule heating to chtMultiRegionFoam*, CFD with open source software, 2009.

[5] Singhal, A. *Tutorial to set up a case for chtMultiRegionFoam in OpenFOAM 2.0.0*, University of Luxembourg, 2014

[6] Craven, B.A. and Campbell,R.L. *Multi-Region Conjugate Heat/Mass Transfer, MRconjugate-HeatFoam: A Dirichlet-Neumann partitioned multi-region conjugate heat transfer solver*, 6th OpenFOAM Workshop, 2011.

[7] van der Tempel, M. *A chtMultiRegionSimpleFoam tutorial*, 6th OpenFOAM Workshop, 2012.

[8] Lide, D.R., ed. *CRC Handbook of Chemistry and Physics (90th ed.)*, Boca Raton, Florida: CRC Press, 2009.