Combination of reactingFoam and chtMultiRegionFoam as a first step toward creating a multiRegionReactingFoam, suitable for solid/gas phase reactions

Developed for OpenFOAM-1906

Author:
Seyed Morteza MOUSAVI
Lund University
morteza.mousavi@energy.lth.se

Peer reviewed by:
J. L. ALEJANDRO BARBA Piña
MUYE GE

Licensed under CC-BY-NC-SA, https://creativecommons.org/licenses/

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

January 14, 2020
Learning outcomes

The reader will learn:

How to use it:

• how to use the reactingFoam solver.
• how to use the chtMultiRegionFoam solver.

The theory of it:

• the theory of the governing equations of the gas phase (reactingFoam) which are continuity, momentum, species, and energy equations.
• the theory of energy equation which is solved in the solid phase (chtMultiRegionFoam).
• the theory of the boundary condition at the interface between solid and gas phase.

How it is implemented:

• how reactingFoam solver is implemented.
• how chtMultiRegionFoam solver is implemented.
• how turbulentTemperatureCoupledBaffleMixed boundary condition is implemented.

How to modify it:

• how to modify and add multi-region capability to the reactingFoam solver, in order to create the multiRegionReactingFoam.
Prerequisites

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

- Fundamentals of Heat and mass Transfer, Book by Frank P. Incropera [1].
- Dimensional groups like Nusselt and Reynolds Numbers.
- Be able to run standard documentation tutorials, like damBreak or cavity tutorial cases.
- It is recommended to be familiar with biomass or solid-fuel combustion.
Chapter 1

Introduction

1.1 Background and motivation

CFD is proved to be a powerful tool to study combustion and reacting flows, and OpenFOAM has different solvers related to this subject where each solver is designed for a specific purpose. Among the combustion solvers in OpenFOAM, reactingFoam is one being widely used to study the transient or steady-state flows with chemical reactions. Since the combustion itself is a gas phase phenomena, this solver only solves the governing equations for the gas phase. However, in some problems such as solid fuels or biomass combustion, it is required to study the solid phase as well which is not possible in the current implementation of reactingFoam solver. In this report, it is explained how to overcome this limitation and how to add new regions to the original reactingFoam solver, where the governing equations for solid phase can be solved.

It should be noted that there are other solvers such as reactingMultiphaseEulerFoam already available in the OpenFOAM package which can be used to study the multi-phase liquid-gas reacting flows, but even those solvers are not appropriate to solve the governing equations for the solid-phase fuel. Another relevant solver is the coalChemistryFoam which is specially developed for the combustion of pulverized solid fuels e.g. coal with very small dimensions. This solver is an Eulerian-Lagrangian solver which assumes particles are small enough to have a constant thermo-physical properties all over each particle. This is usually a good assumption for solid fuels with diameters smaller than a few hundred micrometers (depending on the conditions), but this solver cannot be used usually for larger particles such as biomass pellets which are larger than a few millimeters in diameter.

For larger solid fuels, the gradients of temperature and other properties inside the particle cannot be neglected. Therefore, an appropriate mesh should be created for the solid domain as well as the gas phase domain to solve the discretized governing equations in that region. However, the governing equations for the solid and gas phase are different hence we need to create a mesh for more than one region which is called multiRegion in OpenFOAM terminology. The chtMultiRegionFoam solver is used to learn how multiRegion works in OpenFOAM and how we can add a second region to the reactingFoam. In reality, the solid fuels are usually porous and reacting, but in this project, only the energy equation is solved in the solid region and the fluid motion and reactions in the solid phase are not included in this report. In order to couple the energy equations in solid and gas phase, a proper boundary condition at the interface is required which is explained in detail in this report.
1.2 Report layout

After this introduction, the theory behind the reactingFoam solver will be explained in chapter 2. In chapter 3, two simple tutorials one for reactingFoam and another for the chtMultiRegion will be explained in order to learn how to use these two solvers. For the chtMultiRegion case a special focus will be on the boundary condition at the interface between the solid and gas phase regions for the energy equation. The implementation of reactingFoam, parts of the chtMultiRegionFoam related to multiRegion, and the interface boundary condition will be explained in chapter 4. Finally, in chapter 5 a step by step guide to modify the reactingFoam solver for adding a new solid region is presented and a simple tutorial case for the new solver is also explained.
Chapter 2

Theory

In this chapter, the governing equations from the reactingFoam solver which are solved for the gas phase region are explained. Furthermore, the energy equation for the solidPhase from the chtMultiRegionFoam is presented and at the end, the boundary condition for coupling the energy equations in solid and gas phase regions are explained.

2.1 Governing equations

2.1.1 Fluid region

The continuity, momentum, energy, and species transport equations are solved within the fluid region. Here, the basic equations which are considered in reactingFoam solver will be explained without considering the special sources terms (such as gravity or radiation) which can be added to these equations through fvOptions. The governing equations are

continuity equation
\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho U) = 0,
\]  

(2.1)

momentum equation
\[
\frac{\partial (\rho U)}{\partial t} + \nabla \cdot (\rho U U) = -\nabla p + \nabla \cdot \tau,
\]

(2.2)

\[
\tau = \mu \left[ (\nabla U + \nabla U^T) - \frac{2}{3} \nabla \cdot U I \right],
\]

(2.3)

species transport equation
\[
\frac{\partial (\rho Y_i)}{\partial t} + \nabla \cdot (\rho U Y_i) = \nabla \mu_{eff} \nabla Y_i + \dot{R}_i,
\]

(2.4)

and energy equation
\[
\frac{\partial}{\partial t} (\rho h) + \nabla \cdot (\rho U h) + \nabla \cdot (\rho K) + \nabla \cdot (\rho U K) - \frac{\partial p}{\partial t} = \nabla \alpha_{eff} \nabla h + \dot{R}_{heat}.
\]

(2.5)

In the above equations, \( \rho \) is density, \( t \) is time, \( U \) is the velocity vector, and \( p \) is pressure. \( \mu \) is the dynamic viscosity of the fluid mixture and \( \mu_{eff} \) is the effective dynamic viscosity which is calculated based on the turbulence model. In species transport equation, \( Y_i \) is the mass fraction of the i-th species and \( \dot{R}_i \) is the production rate of the same species due to reactions. The energy equation is solved for internal enthalpy, \( h \), in which \( K \), \( \alpha_{eff} \), and \( \dot{R}_{heat} \) are the kinematic energy, effective thermal diffusivity, and heat generation due to reactions, respectively.
2.1.2 Solid region

In the solid region, since we do not have any fluid motion, only the heat conduction equation will be solved which is in form of

\[
\frac{\partial}{\partial t}(\rho h) = \nabla \alpha_{eff} \nabla h,
\]

(2.6)

where \( \rho \), \( h \), and \( \alpha_{eff} \) are the density, enthalpy, and effective thermal diffusivity of the solid, respectively. In this project, the movement of the fluid movement inside the pores of the solid and also the radiative heat transfer is neglected.

2.2 Boundary conditions

At the interface between solid and gas regions, an appropriate boundary condition is required to couple the energy equations in solid and gas phase regions. To derive the equations for this boundary condition, consider the two cells at either side of the interface such as in Figure 2.1.

![Figure 2.1: Schematic of two cells at the interface between solid and gas regions.](image)

As it is displayed in Figure 2.1, \( T_c \) is the temperature at the cell center and \( T_p \) is the temperature on the patch for both sides. \( q''_1 \) is the heat flux out of side 1, and \( q''_2 \) is the heat flux which enters the side 2. Since there is no surface reaction, the energy conservation indicates that the temperature at both sides of the interface and the heat fluxes on both sides of the interface should be equal (in magnitude), therefore

\[
T_{p,1} = T_{p,2} = T_p, \tag{2.7}
\]

and

\[
q''_1 = q''_2 = q''. \tag{2.8}
\]

The value of the heat fluxes in the eq. 2.8 can be calculated from the one dimensional Fourier’s law which gives

\[
-k_1 \frac{\partial T}{\partial n} \bigg|_{side1} = -k_2 \frac{\partial T}{\partial n} \bigg|_{side2}, \tag{2.9}
\]
where $k$ is the thermal conductivity on each side and $n$ is the direction normal to the boundary. Assuming a first order discretization of the temperature gradient on the previous equation, gives us

\[ k_1 \Delta_1 (T_{c,1} - T_p) = k_2 \Delta_2 (T_p - T_{c,2}). \tag{2.10} \]

Based on the above equation the values of the $T_p$ and $q''$ at the interface can be calculated.

\[ T_p = \frac{k_1 \Delta_1 T_{c,1} + k_2 \Delta_2 T_{c,2}}{k_1 \Delta_1 + k_2 \Delta_2}. \tag{2.11} \]

\[ q'' = k_1 \Delta_1 (T_{c,1} - T_p) = k_2 \Delta_2 (T_p - T_{c,2}). \tag{2.12} \]

This boundary condition is already available in the OpenFOAM package under the name of "turbulentTemperatureCoupledBaffleMixed" which we will describe its implementation later in this report.
Chapter 3

Tutorials

In this chapter, we will go through two simple tutorial cases which are already available in OpenFOAM package, in order to understand which files and options are required to run a case using reactingFoam solver. Moreover, we will go through a simple tutorial case for chtMultiRegionFoam in order to find out which files are required to have a multiRegion simulation.

3.1 reactingFoam tutorial

In this part, we will try to explain how to use the reactingFoam solver. In order to start, copy the "SandiaD>LTS" tutorial case to your run directory and go to that directory using the following commands:

run

cp -r $FOAM_TUTORIALS/combustion/reactingFoam/RAS/DLR_A_LTS/ .

cd DLR_A_LTS

After doing this, you can simply run the case by using ./Allrun command, but here, we will first go through each directory available in this case to explain the purpose of each file. Then in the latest subsection, we will run the commands available in the Allrun script one by one to see the final results. To get a view of all the available directories and files inside this tutorial, you can use the tree command if you have the package installed. You can see the results also in Figure 3.1.

tree .
3.1. REACTINGFOAM TUTORIAL

3.1.1 The 0 directory

In this directory, the initial values and boundary conditions of all field variables are located in separate files. In order to understand the geometry and boundary conditions, the geometry is displayed in Figure 3.2. This case uses an axis-symmetric geometry, where the fuel and oxidizer enter the domain through two separate inlets and the combustion products exit the domain through an outlet at the top. In the later subsections, we will explain how this geometry is created using the blockMesh utility.

The alphat, epsilon, k, and nut files are related to the turbulence modelling and in case you are going to run a laminar combustion case, you can remove these files. The CH4, H2, H2O, N2, and O2, are the files related to the mass fraction of these species. In this case a diffusion flame is going to be studied where the fuel is a mixture of H2 and CH4 diluted in N2, and the oxidizer is air. You can check this by looking at the species mass fractions in these files, for the "inletfuel" and "inletair" boundaries. This case uses the GRI3.0 mechanism which involves 36 species, but we have only specified the initial and boundary conditions for 5 species! The conditions of the rest of the species are defined in the Ydefault file, where we have defined the initial values and the boundary values of all other species are equal to zero. However, these species might be produced when we run the case, because of the reactions.

The remaining files in the 0 directory are p, U, and T.orig files which contain the initial and boundary conditions for pressure, velocity, and temperature respectively. It is important to note that in fact, the reactingFoam solver looks for "T" instead of "T.orig" in the 0 directory. We will later copy this T.orig file to create T file and modify it by using the setFields utility. So the T.orig file is only a backup file for the time we want to clean the case and run everything back to normal.
3.1.2 The chemkin directory

There are three files in this directory which are related to reaction mechanism that we are going to use. In this tutorial the GRI 3.0 mechanism [2] has been used. The first file "grimech30.dat" contains the name of elements, species, and reactions in the CHEMKIN format and the second file, "thermo30.dat" contains the thermophysical properties of the species. The third file contains the transport properties of species using Sutherland’s formulation. If you look inside the last file in this directory, you will see that the same values are used for all gas species using the wildcard ".*", but the values of H2 and CO2 species are overwritten separately. A more detailed description of how to use chemkin files and how to modify/add/remove the reactions, is presented in [3].

3.1.3 The constant directory

In this directory, some of the main settings of the reactingFoam solver can be defined and later when we use the blockMesh utility, the polyMesh directory will be created here as well. In this section, we will briefly describe how each file in this directory is used to change the settings of the solver.

In the chemistryProperties file, you can turn the chemistry calculations on or off and you can set the chemistry solver settings. In the first block in this file, you can see the chemistryType where the solver and method names are defined. Like many other OpenFOAM dictionaries, since we are going to use ode solvers for chemistry calculations, we need to define the odeCoeff as well. In odeCoeff block, you can change the odeSolver type (default odeSolver in this case is Rosenbrock34) and set the absolute and relative tolerances. The next block is the reduction, where you can use a reduction method such as DAC to reduce the size of your reaction mechanism. Please note that reduction methods would increase the simulation speed at the cost of accuracy and sometimes they might remove some reaction pathways which are important for your case of study. Therefore, for each specific problem, you need to ascertain that using the reduction methods leads to the same results as the original reaction mechanism without reduction. The latest block is the tabulation settings which are also used to enhance the simulation speed and it is usually used with large mechanisms such as GRI3.0. In the constant directory, there is also another file named chemistryProperties.test.
which is almost the same file but with another settings for tabulation. You can use the diff command to get the differences between these two file.

diff constant/chemistryProperties constant/chemistryProperties.test

In the results of this command you can see the following settings of the tabulation have been changed in the chemistryProperties.test file.

tolerance 1e-4; -> 0.003;
Temperature 1000; -> 10000;
deltaT 0.5; -> 1;
maxNLeafs 2000; -> 5000;
chPMaxLifeTime 100; -> 1000;
maxGrowth 10; -> 100;
checkEntireTreeInterval 5; -> 500;
variableTimeStep true; -> removed

In the combustionProperties file, you can turn combustion on or off and change the combustion model which is EDC1 by default. Please note that if you change the combustion model to any other model e.g. laminar, you have to create a block named laminarCoeffs otherwise the solver would complain.

In the g file, the vector of the gravitational acceleration has been defined and the $\kappa-\epsilon$ turbulence model is selected in the turbulenceProperties file. In the thermophysicalProperties file, the properties of the mixture and the energy equation and equation of state are identified. It is also stated that the only two other files in this directory i.e. reactionsGRI and thermo.compressibleGasGRI should be read by foamChemistryReader in order to get the reactions and species properties. These two files actually contain the same data as the three files in the chemkin directory, but they are in the OpenFOAM format, so the foamChemistryReader can read them. You can use the chemkinToFoam command to convert the files in the chemkin directory to these two files in the OpenFOAM format. Now it becomes obvious that the solver, does not actually use the contents of the chemkin folder and only the files in OpenFOAM format are used here. You can test this by removing the chemkin directory (or by moving it to a backup directory) and running the solver.

3.1.4 The system directory

The blockMeshDict, controlDict, fvSchemes, and fvSolution files in this directory are required for most tutorials, so you should already be familiar with all of them if you have already followed any other tutorial e.g. for cavity case. The decomposeParDict is used to decompose the solution domain into several subdomains for parallel processing. Please note that the default numberOfSubdomains in this file is 6 in this case, which is more than the available number of CPUs in some computers. Therefore, if you get an error during decomposition, you might need to reduce the number of CPUs and change the coeffs block in this file accordingly.

The only other file in this directory is setFieldsDict which is used to change the initial values of a specific field. in this case, it sets the initial values of temperature in the field to 298K everywhere. Then overwrites the initial values of temperature, in a specific box where the fuel and oxidizer meet, to a high value of 2200K. This is a common approach for combustion simulations to ignite the flame at a specific point by this method, and after a few iterations, the effects of this initial condition will wear off, so we can get the steady or final state conditions.

---

1Eddy Dissipation Concept
3.1.5 Running the case

As it has been mentioned, you can run the case by running the Allrun script, but it is good if we go through the script and run the commands one by one to understand what is the purpose of each command.

First of all, we have to remove the current T file in the 0 directory (if it is available) and replace it by a clean copy from T.orig file.

```bash
rm -f 0/T
cp 0/T.orig 0/T
```

The next step would be to translate the chemkin files to the OpenFOAM format by using the chemToFoam utility. This utility needs five file names as inputs in a specific order to operate. The first three are the names (and local addresses) of the mechanism file, thermodynamic file, and the transportProperties file in the chemkin format. Next two names are the names of the output reaction and thermodynamic files in the OpenFOAM format. Here is a sample of how to use this command for this tutorial:

```bash
chemkinToFoam \
chemkin/grimech30.dat chemkin/thermo30.dat chemkin/transportProperties \
constant/reactionsGRI constant/thermo.compressibleGasGRI
```

Next, we can create the mesh using the blockMesh utility and set the initial hot zone in the domain for ignition of the flame, using the setField utility. You should already know that these two utilities use the blockMeshDict and setFieldsDict dictionaries in the system directory in order to operate.

```bash
blockMesh
setFields
```

At this stage, you should be able to see the geometry, mesh, and the initial condition of temperature in order to check everything is working fine. The initial condition of the temperature field is presented in Figure 3.3.

Figure 3.3: Initial temperature field for the DLR_A_LTS case after using the setFields utility.
Now you can run the case simply by using the reactingFoam command. However, this case uses a detailed reaction mechanism for methane combustion and the simulation requires a very long time to converge, even if you run in parallel (more than 1 day on a core-i7 PC using 6 cores). Therefore, we can use a very simple global reaction instead just to test the case. Here, we are going to use the global reaction which is used in the membrane tutorial case. Using the following commands, we will first move the original chemistry and reaction files to the constant/BACKUP directory and then we will copy the new files from the membrane tutorial case. The chemistry is originally off in the new files, so we will have to use the sed command to turn it on.

```
mkdir constant/BACKUP
mv constant/chemistryProperties,thermophysicalProperties,\ reactionsGRI,thermo.compressibleGasGRI} constant/BACKUP

$FOAM_TUTORIALS/combustion/reactingFoam/RAS/membrane/constant/{chemistryProperties,\ thermophysicalProperties, reactions, thermo.compressibleGas} constant/

esed -i s/"chemistry .*"/"chemistry on,;"/g constant/chemistryProperties
```

The case is now ready and we can run the simulations with global reaction mechanism in a few seconds. First, we decompose the case using decomposePar utility. The "-force" flag indicates that it will recreate the processor directories even if they are already exist. Then we should use mpirun like the following commands, to run the case in parallel using 6 cores (only if we have this number of CPUs available! for instance, I could not use more than 4 cores on my laptop, so I had to reduce the number of cores in both decomposeParDict and mpirun command). Finally, we can run the reconstructPar utility to attach the decomposed regions together and then we will be able to see the results in paraview. The following commands are almost identical to what you can see in the Allrun script, but without using the runFunctions so you can normally run them in the terminal one by one.

```
decomposePar -force
mpirun -n 6 reactingFoam >& log.reactingFoam &
reconstructPar
```

After the simulation is finished, you can observe the results using paraFoam. Some of the results at the final time-step are also presented in Figure 3.4.
3.2  chtMultiRegionFoam tutorial

In this section we are going to use one of the chtMultiRegionFoam tutorials to understand how multiRegion works in openFoam. The focus is only on the multiRegion part and more details on other parts can be found in [4]. The multiRegionHeater tutorial case will be used here to clarify how we can use more than one region in OpenFOAM. Use the following commands to copy the tutorial case to the run directory and change to that directory:

```bash
run
cp -r $FOAM_TUTORIALS/heatTransfer/chtMultiRegionFoam/multiRegionHeater/ .
cd multiRegionHeater
```

3.2.1  Directories

We can get an overview of the contents of this tutorial case by the following command:

```bash
tree -L 2
```

The output of the above command can also be observed in Figure 3.5. It is interesting to note that compared to single region tutorial cases, there are more directories inside constant and system directories. These directories which are named "bottomWater, heater, leftSolid, rightSolid, and topAir", are related to the five regions with the same names which are available in this case. The geometry and the mesh of these regions are presented in Figure 3.6. We can see that we have an air region at the top and a water region at the bottom, with three different solid regions in the middle, which separate the two fluid regions. The left and right solid regions are planar, but a part of the heater is going down into the water which cannot be seen in this figure. The idea of this section is to understand how OpenFOAM handles multiRegion simulations and how different regions are coupled to each other.
In the system directory, every region has its own directory where you can find four files. The first three files are *fvSchemes*, *fvSolution*, and *decomposeParDict* which we already know their purpose from other tutorials. You can either have specific files with specific schemes and solution methods for each region, or you can use symbolic links to other files from other sub-directories. For instance,
in the system/topAir directory, decomposeParDict and fvSchemes are symbolic links to the same
files in other subdirectories, but fvSolution and changeDictionaryDict are normal independent files.
You can use the "ls -l" command in a directory to see the difference between symbolic links and
normal files. The last file in each region directory is the changeDictionaryDict, which is not neces-
sary for multiRegion simulations, but it is used here for convenience to set the boundary conditions
each region. You can take a look at this dictionary for each region to see how different boundary
conditions are set in this tutorial. An interesting boundary condition for us in this report is the
temperature interface between different regions. For instance if you look at the T sub-dictionary in
"system/topAir/changeDictionaryDict", you will find that they have used a wild-card "topAir_to.*"
to set the same boundary condition at the interface of the topAir to other neighbour regions (the
three solid regions in middle, see Figure 3.6). We will get back to this boundary type later in this
report.

In the constant directory, a similar pattern to system directory can be observed. There are five
directories available specific to each region, where the radiationProperties, thermoPhysicalProper-
ties, and turbulenceProperties (only for fluid regions) files are located. The purpose of these files
are similar to other single regions tutorials, so we will not go deep into them, but again, you should
know that you can use either normal properties files or symbolic links to identify these properties for
each region. Other than these directories, there are two other files in the constant directory, i.e. "g"
and "regionProperties". The "g" file is the gravity specification and it is used similar to all other
single region tutorials. But regionProperties is a required file for all the multiRegion simulations.
There is only one sub-dictionary in this file which is named regions where the user should specify
the fluid and solid regions. In this tutorial case, the sub-dictionary looks like this:

```plaintext
regions
(
    fluid (bottomWater topAir)
    solid (heater leftSolid rightSolid)
);
```

The only other directory which is initially available is the 0.orig directory, which contains the
original field files. These files, do not belong to any specific region and the boundary conditions are
not correct. However, a copy of these files will be generated for each region later, and the boundaries
will be modified by the changeDictionaryDict which are located in the system subdirectories and
discussed earlier.

### 3.2.2 Preparing the multi-region case

One can simply run the Allrun.pre script to prepare the case for multiRegion simulations. However,
in order to understand the required steps for this kind of simulation, we will read through this script
line by line. You can open the script with your favorite text-editor and follow this part of tutorial.
The first three lines are similar to most run scripts in OpenFOAM tutorials, where we indicate we
are using sh script, we run from this directory, and we source the OpenFOAM’s RunFunctions.

```plaintext
#!/bin/sh
cd ${0%/*} || exit 1               # Run from this directory
. $WM_PROJECT_DIR/bin/tools/RunFunctions # Tutorial run functions
```

In the following two lines, first we will use the blockMesh utility to create the initial mesh which
is a simple 3D box (with no multiRegion capability). The next step will be to use topoSet utility to
create required cellZones for multiRegion simulations.

```plaintext
runApplication blockMesh
runApplication topoSet
```
It is important to know how we can use topoSet to create cellZones. It is also required to know that in OpenFOAM terms, cellSet, cellZone, and region are different, even though they are closely related. A cellSet is a named list of specific cell indices, which can be used for data sampling or creating the cellZones. The cellZone is an extension of cellSet, which contains more mesh related information, and can be used e.g. for MRF or porous zones. Finally, a region is a type of independent mesh, which which we can use to solve specific governing equations using specific boundary conditions. In order to create a region, we can start by creating a cellSet and then convert it to a cellZone. We can check the system/topoSetDict to find more about this. In this file, you can find a sub-dictionary named actions which contains two or more actions to generate a cellZone for each region. For instance, at the top of the file, you will find three different actions to generate the heater cellZone.

```
actions
{
    // Heater
    {
        name heaterCellSet;
        type cellSet;
        action new;
        source boxToCell;
        box (-0.01001 0 -100) (0.01001 0.00999 100);
    }
    {
        name heaterCellSet;
        type cellSet;
        action add;
        source boxToCell;
        box (-0.01001 -100 -0.01001) (0.01001 0.00999 0.01001);
    }
    {
        name heater;
        type cellZoneSet;
        action new;
        source setToCellZone;
        set heaterCellSet;
    }
}
```

The first block in the above actions, creates a new cellSet named "heaterCellSet" and adds all of the mesh cells in a specific box to that cellSet (notice the third line "action new;"). The second block, adds more cells which are within another box to the same cellSet (notice "action add;"). The last block is used to convert the heaterCellSet to a new cellZone which is named "heater". A cellZone, can later be converted to a mesh region, which we need for multiRegion simulations, by using the splitMeshRegions utility, but we will get back to that later.

The next line in the Allrun.pre script calls the restore0Dir function. Based on the comment above this line, and the comments in the RunFunctions, it simply overwrite the 0/ with the contents of 0.orig/ if it exists.

```
# Restore initial fields
restore0Dir
```

Now we can get to create actual mesh regions, based on the cellZones that we previously generated. This is simply done by using the splitMeshRegions utility in the following way.

```
runApplication splitMeshRegions -cellZones -overwrite
```
You can check the contents of constant sub-directories before and after running this command. This command actually creates a new mesh (new region), for each cellZone that we had earlier. It also creates automatic boundaries at the interface between two regions. For instance if you check the constant/topAir/polyMesh/boundary file, you will see there are three new boundaries, named "topAir_to_rightSolid", "topAir_to_heater", and "topAir_to_leftSolid". There is no boundary between air and water in this tutorial, because they have no face in common. This utility also changes the contents of the 0 directory. It creates a new sub-directory for each region in the 0 directory, and creates similar field files for each region. As it was mentioned before, the boundary conditions are not fixed yet. Only p and T fields are required for the solid regions, so we will get rid of other field files i.e. epsilon, k, p_rgh, and U by using the next few lines of the Allrun.pre script:

```
# Remove fluid fields from solid regions (important for post-processing)
for region in $(foamListRegions solid)
    do
        rm -f 0/$region/{nut,alphat,epsilon,k,U,p_rgh}
        rm -f processor*/0/$region/{nut,alphat,epsilon,k,U,p_rgh}
    done
```

You can check that everything has worked fine by using "tree 0" command, to see the content of each sub-directory. The next and final step in the preparation of this multiRegion tutorial case is to fix the boundary conditions in every file, for every region. This is done by running the changeDictionary utility for each region. The next few lines of the script, uses the changeDictionaryDict from the system sub-directories which are mentioned before.

```
for region in $(foamListRegions)
    do
        runApplication -s $region changeDictionary -region $region
    done
```

By running the last commands, the case will be ready for simulation. You can also check the temperature fields for each region (e.g. 0/topAir/T) to find the boundary settings for the interfaces. You can see that for all interfaces, the boundary type is compressible::turbulentTemperatureCoupledBaffleMixed which we will discuss later in this report.

### 3.2.3 Running the case

Now that the case is completely prepared, you can run the simulation by running the chtMultiRegion solver. However, if you want to run in parallel, you can use the commands from the Allrun script which are presented in the following.

```
# Decompose
runApplication decomposePar -allRegions
# Run
runParallel $(getApplication)
# Reconstruct
runApplication reconstructPar -allRegions
```

It is important to note that both decomposePar and reconstructPar utilities, require -allRegions flags to work for multiRegion problems. After a short while, when the simulation is done, you can see the results by paraFoam. The final temperature distribution in the solid regions are displayed in Figure 3.7. It can be observed that the heat from the bottom part of the heater, is transferred to the rightSolid, but not to the leftSolid. This is mainly because there is a heat resistance in the interface between heater and leftSolid (check 0/leftSolid/T). Furthermore, in the fluid regions, we have air and water flow towards the x direction, which probably enhances the heat transfer to the
rightSolid and cools the leftSolid down.

Figure 3.7: The final temperature of the solid regions in the multiRegionHeater tutorial case.
Chapter 4

Implementation of original solvers and boundary condition

In the first section of this chapter the implementation of reactingFoam solver is explained. The details of the high-level code as well as the purpose of each part of the code of this solver is presented. In the next section, parts of chtMultiRegionFoam which are related to the multiRegion simulations are investigated. This knowledge is required to modify the reactingFoam solver to include solid regions as well. At the last section of this chapter, the implementation of the turbulentTemperatureCoupledBaffleMixed boundary condition is explained, which is used to couple the energy equation for different regions.

4.1 reactingFoam

The code for this solver is located among the combustion solvers which can be accessed by:

sol
cd combustion/reactingFoam

There are three directories and nine files located in this directory. The first directory is Make, which is required for the compilation of the code and the other two directories are related to rhoReactingFoam and rhoReactingBouyantFoam, which are derived from reactingFoam but are not the topic of this report. The files in this directory and their main purpose are listed below:

- reactingFoam.C : The main code of the solver, which calls the other .H files from this directory.
- createFields.H : Creates the field variables (e.g. p, U, T, ...).
- createFieldRefs.H : Create references to some field variables.
- setRDeltaT.H : Adjusts the local timestep, only if the local time stepping (LTS) is used.
- UEqn.H : Defines the momentum equation and perform the momentum predictor step.
- YEqn.H : Defines and solves the species transport equation.
- EEqn.H : Defines and solves the energy equation.
- pEqn.H : Defines and solves the pressure equation and does the pressure-velocity coupling based on pimple algorithm.
4.1. REACTINGFOAM

In order to understand the code of the solver, we will go through the main file (reactingFoam.C) block-by-block, and explain the purpose of each block. The first section of the main file, is where the libraries are included. The code presented below contains the libraries which are used in this solver. These libraries are different compared to other .H files which are included inside the main function. These libraries contain the definition of classes and their functions, and they are included only once before the main function. However, the other .H files which are included inside the main function are only parts of code, aimed to perform a specific task.

```
#include "fvCFD.H"
#include "turbulentFluidThermoModel.H"
#include "psiReactionThermo.H"
#include "CombustionModel.H"
#include "multivariateScheme.H"
#include "pimpleControl.H"
#include "pressureControl.H"
#include "fvOptions.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"
```

After including the required libraries, the main function starts with the following lines:

```cpp
int main(int argc, char *argv[])
{
    argList::addNote
    (    
        "Solver for combustion with chemical reactions"
    );

    #include "postProcess.H"

    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
```

The first line above is the definition of the main function, and the next lines are used in most solvers to do some common tasks, such as preparing the solver for postProcess utility or checking case options for dry-run (single timestep) or other flags. You can easily find the contents of each of these included files using doxygen. There are seven other .H files included after these lines which are presented in the following:

```
#include "createTime.H"
#include "createMesh.H"
#include "createControl.H"
#include "createTimeControls.H"
#include "initContinuityErrs.H"
#include "createFields.H"
#include "createFieldRefs.H"
```

The first three lines are responsible for creating the time variable named "runTime", the mesh variable and a reference to that variable named "mesh", and finally the pimple control variable named "pimple". These three variables are used in different parts throughout the solver code. createTimeControls.H reads the data related to adjustable run time from the controlDict dictionary in the case and stores them in corresponding variables. initContinuityErrs.H creates a scalar field to store the continuity errors during the iterative solution. The mentioned .H files up to here are shared files which are used by many solvers. That is why they are not located in the solver directory. However, the final two .H files are the createFields.H and createFieldRefs.H which are located in the
4.1. REACTINGFOAM

CHAPTER 4. IMPLEMENTATION OF ORIGINAL SOLVERS AND BOUNDARY CONDITION

solver directory, and are specific to this solver. These two files will be explained in detail.

The first lines in createFields.H are displayed below:

```c++
#include "createRDeltaT.H"

Info<< "Reading thermophysical properties\n" << endl;
autoPtr<psiReactionThermo> pThermo(psiReactionThermo::New(mesh));
psiReactionThermo& thermo = pThermo();
thermo.validate(args.executable(), "h", "e");

basicSpecieMixture& composition = thermo.composition();
PtrList<volScalarField>& Y = composition.Y();
```

The first line is a call to createRDeltaT.H, which creates the required variables for local time stepping (LTS). LTS is an algorithm for steady state simulations, where you can have different timesteps at different cells, based on their local Courant number. The user can identify whether he wants to use LTS or not in the fvSchemes file in the case directory. If you want to use LTS for your simulation, you should use keyword "localEuler" as the default value of the ddtSchemes, and if your simulation is not steady-state or you do not want to use LTS, you can use "Euler" keyword instead.

For instance, the ddtSchemes from the DLR_A_LTS is presented below:

```c++
ddtSchemes
{
    default localEuler;
}
```

The next four lines in the createFields.H code (after #include "createRDeltaT.H"), are used to create the thermodynamic model. pThermo is a pointer to the new thermodynamic model, and thermo is the name of the reference to that variable. The next two lines define two variables named composition and Y. The first one is a reference to the mixture composition, and the second one is a list of references to the mass fraction of each species.

The next few lines which are presented below, look for the inertSpecie which should be defined in the thermophysicalProperties dictionary. If the inertSpecie (which is required for this solver) is not defined correctly, it would abort the program execution with a fatal error.

```c++
const word inertSpecie(thermo.get<word>("inertSpecie"));
if (!composition.species().found(inertSpecie))
{
    FatalIOErrorIn(args.executable().c_str(), thermo)
        << "Inert specie " << inertSpecie << " not found in available species "
        << composition.species() << exit(FatalIOError);
}
```

In the next few lines, the scalar fields rho (density), the vector field U (velocity), and a reference to scalar field p (pressure) are defined. There are different ways to define a new scalar/vector field which are not going to be explained here. However, it is good to take a look at the definitions between lines 19 to 44 of the createFields.H (not repeated here) to see how rho, U, and p fields are defined. The field rho is created based on thermodynamics, so a field file is not required in 0 directory when we create this field. The field U is created with IOobject::MUST_READ flag, which means the initial and boundary values must be identified in the time directories. The p variable here is only a reference to thermodynamic pressure (thermo.p()), but in order to create the thermo variable, the p file in time directory is required.

The next three lines in the same file are presented below:
4.1. REACTINGFOAM

CHAPTER 4. IMPLEMENTATION OF ORIGINAL SOLVERS AND BOUNDARY CONDITION

#include "compressibleCreatePhi.H"

define PressureControl pressureControl(p, rho, pimple.dict(), false);

mesh.setFluxRequired(p.name());

The first line above is used in many compressible solvers, and it creates a surface scalar field variable named phi, which contains the normal mass flux values for all the internal and external faces of the mesh. The flux is calculated by the following equation on each surface,

\[ \phi = \rho(\vec{U} \cdot \vec{S}_f) \]  

(4.1)

where \( \vec{U} \) is the velocity vector on each surface, estimated by linear interpolation, and \( \vec{S}_f \) is the surface normal vector multiplied to the surface area. The dot sign is used to show the inner product of the two vectors. The second line above is to define a pressureControl, which is used to control the pressure if pMax or pMin are specified in pimple settings in the fvSolution dictionary. The next line adds pressure to the list of variables which their flux should be calculated. This is required by the pimple algorithm for p-U coupling.

Through the lines 52 to 68 of createFields.H, a pointer to turbulence model, and another pointer to the combustion model are created, which are named "turbulence" and "reaction", respectively. Then through lines 70 to 76, the references to mass fraction fields and specific enthalpy field, are added to a table named fields (apparently for interpolation purpose). Then the Qdot (heat generation per second per unit of volume) scalar field is defined with the "IOobject::READ_IF_PRESENT" flag. It means that if a file named Qdot was available in the time directory, the code will read that, otherwise it gives zero values to this fields.

In the last few lines of the createFields.H, four other .H files are included which are also presented below.

#include "createDpdt.H"
#include "createK.H"
#include "createMRF.H"
#include "createFvOptions.H"

The first two files create the dpdt and K variables which are required for the energy equation (see eq. 2.5). The third line is used only if moving reference frame (MRF) is required which is not the topic of this report. Finally, the last line creates the fvOptions variable, which can be used to add some terms such as gravity or radiation in governing equations, without modifying the source code.

After reading the last line of the createFields.H, the compiler returns to the main file, where the createFieldRefs.H is included. This file only contains three lines of code which is presented below:

const volScalarField& psi = thermo.psi();
const volScalarField& T = thermo.T();
const label inertIndex = composition.species().inertSpecie();

The above lines, create a reference to thermodynamic compressibility named "psi", a reference to gas temperature named T, and a new label named inertIndex which will contain the label of the inert species in the mixture. After this, we should return to the main file again.

In the next few lines of the main file, which are repeated below, the solver initially calls the turbulence->validate() function. You can find the definition of this function in OpenFOAM source.
4.1. REACTING FOAM

CHAPTER 4. IMPLEMENTATION OF ORIGINAL SOLVERS AND BOUNDARY CONDITION

directory, in the eddyViscosity.C. This function corrects the turbulent dynamic viscosity \( \mu_t \) based on the turbulence model that is used during the runtime. The next two lines inside the if block calculate the compressible Courant number and sets the initial time step of the simulation, based on the settings of the controlDict. The LTS variable is true if local time stepping is used (localEuler ddtScheme is used), and false otherwise. Therefore these two lines will be used only if the local time stepping is off.

\[ \text{turbulence->validate();} \]

\[ \text{if (!LTS)} \]

\[ \text{#include "compressibleCourantNo.H"} \]
\[ \text{#include "setInitialDeltaT.H"} \]

In the next block of the code, the time loop starts. The while loop is controlled by the values that we set in the controlDict of the case. In the first line of the loop, readTimeControls.H is included, which simply updates the time settings of in case the controlDict is modified during the run time. In the next few lines, there is an if statement to set the time step. If LTS (localEuler ddtScheme) is used, then setRDeltaT.H is called, which calculates the local time step based on the local Courant number. If LTS is not used, then the compressibleCourantNo.H is called to calculate the maximum Courant number. Then the setDeltaT.H is called to set the global time step based on the maximum Courant number. After the time step size is identified, the ++runTime command is used to increase the simulation time based on current time step size.

\[ \text{Info<< "\nStarting time loop\n" } \]
\[ \text{while (runTime.run())} \]

\[ \text{#include "readTimeControls.H"} \]

\[ \text{if (LTS)} \]

\[ \text{#include "setRDeltaT.H"} \]
\[ \text{else} \]
\[ \text{#include "compressibleCourantNo.H"} \]
\[ \text{#include "setDeltaT.H"} \]

\[ \text{++runTime;} \]

\[ \text{Info<< "Time = " } \]
\[ \text{runTime.timeName()} \]
\[ \text{nl} \]
\[ \text{endl;} \]

It is fairly easy to understand the purpose of the code in the rest of the main loop. In fact the remaining lines of the main loop are used to solve the governing equations. The governing equations for a reacting flow, which are explained in Chapter 2, are continuity equation (rhoEqn.H), momentum equation (UEqn.H), species transport equation (YEqn.H), energy equation (EEqn.H), and a pressure equation for pressure-velocity coupling (pEqn.H, or pcEqn.H if using SIMPLEC). The call to the .H files which solve each of these equations are presented in the following lines.

\[ \text{#include "rhoEqn.H"} \]
\[ \text{while (pimple.loop())} \]

{
#include "UEqn.H"
#include "YEqn.H"
#include "EEqn.H"

// --- Pressure corrector loop
while (pimple.correct())
{
    if (pimple.consistent())
    {
        #include "pcEqn.H"
    }
    else
    {
        #include "pEqn.H"
    }
}
if (pimple.turbCorr())
{
    turbulence->correct();
}

rho = thermo.rho();
runTime.write();
runTime.printExecutionTime(Info);

As it can be seen the first equation to be solved outside the pimple loop is the rhoEqn.H. This file is not available in the code directory, but instead you can find it in the OpenFOAM source directory
(./finiteVolume/cfdTools/compressible/rhoEqn.H). The rhoEqn.H for compressible flows are located
in the source directory because the same equation is used for many compressible solvers. It is based
on the compressible continuity equation (eq. 2.1) with addition of fvOptions so the user will be able
to modify it without changing the solver code. The code for rhoEqn.H is presented below.

{  
    fvScalarMatrix rhoEqn
    {
        fvm::ddt(rho) + fvc::div(phi) == fvOptions(rho);
    }
    fvOptions.constrain(rhoEqn);
    rhoEqn.solve();
    fvOptions.correct(rho);
}

After solving the rhoEqn.H for the first time (it will be called again in the pEqn.H or pcEqn.H),
the pimple loop starts. The details of the pimple method for pressure-velocity coupling is not pre-
sented here. In pimple loop, initially UEqn.H is called, which forms, but not solves, the momentum
equation (eq. 2.2) without the pressure gradient term. If the momentumPrediction is active in
the fvSolution of the case directory, it will add the pressure gradient term and solves the equation
to estimate the intermediate velocity of the current time step. This intermediate velocity will be
updated during the pimple loop. The code of UEqn.H is presented below. All terms starting with
"MRF" are related to moving reference frame which is not discussed in this report, and if you are
not using MRF, they will be zero.
MRF.correctBoundaryVelocity(U);

```
tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
    + MRF.DDt(rho, U)
    + turbulence->divDevRhoReff(U)

    ==
    fvOptions(rho, U)
);
fvVectorMatrix& UEqn = tUEqn.ref();
UEqn.relax();
fvOptions.constrain(UEqn);

if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
    fvOptions.correct(U);
    K = 0.5*magSqr(U);
}
```

The next equation to be solved is the species transport equation (eq. 2.4). The whole part of
the YEqn.H is not presented here, but it basically solves the reaction equations to find the source
terms and then solves the following equation for each species, except the inert species. The mass
fraction of the inert species is calculated by using the fact that the summation of mass fraction of
all species are equal to unity.

```
fvScalarMatrix YiEqn
(
    fvm::ddt(rho, Yi)
    + mvConvection->fvmDiv(phi, Yi)
    - fvm::laplacian(turbulence->muEff(), Yi)

    ==
    reaction->R(Yi)
    + fvOptions(rho, Yi)
);
```

Then, the energy equation will be solved (eq. 2.5). It is easy to find that the first four terms on
the left hand side of the equation are similar to those in eq. 2.5. The fifth term is in fact a kind of
if statement in the C++ syntax, which indicates different terms should be used depending on the
name of the energy variable i.e. he.name(). If the energy is presented in form of internal energy,
"e", then the value before ":" will be used, else the value after ":" will be used. In our case, we are
solving the energy equation for the specific enthalpy "h", hence the value of the fifth term will be
"-dpdt" which is in agreement to eq. 2.5.

```
fvScalarMatrix EEqn
(
    fvm::ddt(rho, he) + mvConvection->fvmDiv(phi, he)
    + fvc::ddt(rho, K) + fvc::div(phi, K)
    + (he.name() == "e"?
        fvc::div
    )
```

The next equation to be solved is the species transport equation (eq. 2.4). The whole part of
the YEqn.H is not presented here, but it basically solves the reaction equations to find the source
terms and then solves the following equation for each species, except the inert species. The mass
fraction of the inert species is calculated by using the fact that the summation of mass fraction of
all species are equal to unity.

Then, the energy equation will be solved (eq. 2.5). It is easy to find that the first four terms on
the left hand side of the equation are similar to those in eq. 2.5. The fifth term is in fact a kind of
if statement in the C++ syntax, which indicates different terms should be used depending on the
name of the energy variable i.e. he.name(). If the energy is presented in form of internal energy,
"e", then the value before ":" will be used, else the value after ":" will be used. In our case, we are
solving the energy equation for the specific enthalpy "h", hence the value of the fifth term will be
"-dpdt" which is in agreement to eq. 2.5.
4.2 CHTMULTIREGIONFOAM

As it is mentioned before, we do not go through all parts of the implementation of the chtMultiRegionFoam. The focus here is only on the parts of the code which aid us to run a multiRegion simulation. We can start by changing to the solver directory using the following commands:

```
sol
cd heatTransfer/chtMultiRegionFoam/
```

In this directory, you can find that there are only five files (main file plus four \texttt{.H} files). Other than Make directory, there are "fluid", "solid", and "include" directories which belong to this solver. The fluid directory contains all sorts of files related to fluid regions from creating the mesh to solving
the governing equations. The same goes to the solid directory and the solid regions, but less files are available in that region because only the heat transfer equation is solved in solid regions. The include directory, contains two files related to the multiRegion time step size. Finally, there are also two other directories containing two solvers which are derived from chtMultiRegionFoam but they are not the topic of this report. We will start investigating the implementation by looking at the main file i.e. chtMultiRegionFoam.C.

The main file starts with the section where the libraries are included. Many of the libraries are similar to those used in reactingFoam and you might also be familiar with some other libraries from other solvers. However, there are only the following two libraries which are related to multiRegion and we are going to explain them.

```cpp
#include "solidRegionDiffNo.H"
#include "regionProperties.H"
```

The first library, solidRegionDiffNo.H, contains only the definition to a new scalar function with the same name which calculates the diffusion number of the solid region. It will be used later to determine the time step size, similar to how courantNo in fluid regions determines the time step. The next library contains the regionProperties class which holds the names of the solid and fluid regions. You can find regionProperties.H either through doxygen or by using find command in the OpenFOAM source directory. There are three public member functions in this class with the following details:

```cpp
// Member Functions
//- Total count of all region names.
label count() const;
//- The region names. Sorted by region type.
wordList names() const;
//- The region names in sorted order.
wordList sortedNames() const;
```

After including the libraries, the next interesting part in the main file and in the main function, is how the postProcess.H is called:

```cpp
#define NO_CONTROL
#define CREATE_MESH createMeshesPostProcess.H
#include "postProcess.H"
```

We are not going into the details of the postProcess.H and you can take a look at it later if you are interested. However, it is important to know how the two lines before including the postProcess.H affects how the postProcess utility works for this solver. By default, the postProcess.H calls the ”createControl.H” and ”createMesh.H” files, which we already discussed their purpose. However, in a multiRegion solver, those two files are not implemented. Therefore, ”#define NO_CONTROL” means that createControl.H should not be used in postProcess and ”#define CREATE_MESH createMeshesPostProcess.H” means that createMeshesPostProcess.H should be used instead of createMesh.H.

In the next set of .H files which are included in the main directory, the following .H files are interesting if we are going to create a multiRegion solver:

```cpp
...  
#include "createMeshes.H"
...  
#include "readSolidTimeControls.H"
#include "compressibleMultiRegionCourantNo.H"
#include "solidRegionDiffusionNo.H"
#include "setInitialMultiRegionDeltaT.H"
```
The first file, createMeshes.H, can be found in the solver directory and it contains only the following three lines:

```c++
regionProperties rp(runTime);
#include "createFluidMeshes.H"
#include "createSolidMeshes.H"
```

The rp variable which is created in the first line, reads the regionProperties files in the constant directory of the case. Hence, it stores the names of the fluid and solid regions (see the chtMultiRegionFoam tutorial section). The rp variable is used in the next two .H files mentioned above, to create a mesh in each of the fluid and solid regions. You can find those .H files in the fluid and solid sub-directories of the solver.

Back to the main file, the readSolidTimeControls.H, located in solid sub-directory, reads the maxDi (max diffusion number) from the controlDict of the case. The compressibleMultiRegionCourantNo.H, located in fluid sub-directory, finds the maximum Courant number (called CoNum) in all of the fluid regions. The solidRegionDiffusionNo.H is located in the solid sub-directory and should not be confused with the "solidRegionDiffNo.H" which is a header and it was included before the main function. The solidRegionDiffusionNo.H is part of the code which uses the solidRegionDiffNo function for each solid region to find the maximum diffusion number (called DiNum) in all of the solid regions. Finally, setInitialMultiRegionDeltaT.H, located in the include sub-directory, adjusts the initial time step size, based on the CoNum and DiNum which are calculated before.

After this section in the main function, the time loop starts with the following .H files being called in the time loop:

```c++
while (runTime.run())
{
    #include "readTimeControls.H"
    #include "readSolidTimeControls.H"
    #include "readPIMPLEControls.H"
    #include "compressibleMultiRegionCourantNo.H"
    #include "solidRegionDiffusionNo.H"
    #include "setMultiRegionDeltaT.H"
```

The purpose of the first three .H files above, is to update the time controls, maxDi, and pimple controls in case they have been modified during the run time. The next three .H files are used to recalculate the time step based on CoNum and DiNum, similar to what was explained before.

The rest of the main loop is mainly consist of the pimple loop, in which the governing equations are solved first for the fluid regions and then for the solid regions. Since the energy equation is coupled between fluid and solid regions, there is a possibility to perform additional iterations for energy solution only. In order to use this additional iterations for energy equation, one can use a value greater than one for the pimple nOuterCorrectors, in the fvSolution of the case.

This has been all we wanted to know about a multiRegion solver. Solving the governing equations for each region is not the focus of this report, evenhough some of the governing equations are similar to those explained for the reactingFoam solver. In the final chapter of this report, similar parts of code to what were discussed here are going to be used to convert reactingFoam to multiRegionReactingFoam.
4.3 turbulentTemperatureCoupledBaffleMixed

In this section, the implementation of the turbulentTemperatureCoupledBaffleMixed boundary condition (BC) is explained. You can find the source code for this boundary condition by the following commands:

```bash
src find . -iname "turbulentTemperatureCoupledBaffleMixed"
```

There will be only one result for this find command, and we can change to that directory to find the source code of this boundary condition. In the .H file you can find the description of this boundary condition: "Mixed boundary condition for temperature, to be used for heat-transfer on back-to-back baffles. Optional thin thermal layer resistances can be specified through thicknessLayers and kappaLayers entries.". The mixed boundary condition in OpenFOAM has a specific meaning, which can be found in the "mixedFvPatchField.H". In a mixed boundary patch, the values on the patch are determined by a combination of a fixed value and a fixed gradient, in the form of

\[
x_p = wx + (1 - w) \left( x_c + \frac{\nabla_T x}{\Delta} \right),
\]

where \( x_p \) is the patch value which is calculated by the BC code, \( w \) is the weight field named "valueFraction" in the code, \( x \) is the fixed value named "refValue" in the code, and \( \nabla_T x \) is the patch normal gradient named "refGrad" in the code. The \( x_c \) is the patch internal cell values and the \( \Delta \) is the inverse distance from face centre to internal cell centre. Therefore, for any boundary condition such as turbulentTemperatureCoupledBaffleMixed which is derived from the mixedFvPatchField, we need to identify the refValue, refGrad, and valueFraction to set the boundary condition. For instance, \( w = 0 \) is identical to fixedGradient and \( w = 1 \) is identical to fixedValue boundary conditions.

Back to the header file turbulentTemperatureCoupledBaffleMixedFvPatchScalarField.H, there are four private variables in the class, with their purpose explained in the comments at at the top of the header file. The variables are defined as:

```cpp
// Private data

// Name of field on the neighbour region
const word TnbrName_;

// Thickness of layers
scalarList thicknessLayers_;

// Conductivity of layers
scalarList kappaLayers_;

// Total contact resistance
scalar contactRes_;
```

Where the contact resistance is calculated based on the thickness and conductivity of the layers. And the example of the boundary condition specification is as follows:

```plaintext
<patchName>
{
    type compressible::turbulentTemperatureCoupledBaffleMixed;
    Tnbr T;
    thicknessLayers (0.1 0.2 0.3 0.4);
    kappaLayers (1 2 3 4);
    kappaMethod lookup;
    kappa kappa;
    value uniform 300;
}
```
Similar to most boundary conditions, you can find six constructors, an updateCoeffs() function, and a write() function. The main code which calculates the boundary condition values, is in fact the updateCoeffs() function. You can find the definition of this function in the .C file in the same directory. In the first few lines of this function there is an if statement to check whether the boundary values are already updated or not, and some fixing related to processor communications. Then you can find the following lines, which are used to get the values from the neighbour mesh:

```c
// Get the coupling information from the mappedPatchBase
const mappedPatchBase& mpp =
    refCast<const mappedPatchBase>(patch().patch());
const polyMesh& nbrMesh = mpp.sampleMesh();
const label samplePatchi = mpp.samplePolyPatch().index();
const fvPatch& nbrPatch =
    refCast<const fvMesh>(nbrMesh).boundary()[samplePatchi];
```

Since this is a coupled boundary condition, we need the above lines to have access to the field values on the neighbour mesh. the nbrPatch variable above is a reference to the neighbour patch on the other side of the coupled boundary condition. We have two options in this boundary condition, the first one is to assume a specific contact resistance on the boundary, and the second one is to use the neighbour heat conductivity, \( k \), and \( \Delta \). The next few lines which are presented below are related to those two options:

```c
tmp<scalarField> nbrIntFld(new scalarField(nbrField.size(), Zero));
tmp<scalarField> nbrKDelta(new scalarField(nbrField.size(), Zero));
if (contactRes_ == 0.0)
{
    nbrIntFld.ref() = nbrField.patchInternalField();
    nbrKDelta.ref() = nbrField.kappa(nbrField)*nbrPatch.deltaCoeffs();
}
else
{
    nbrIntFld.ref() = nbrField;
    nbrKDelta.ref() = contactRes_;
}
```

```
mpp.distribute(nbrIntFld.ref());
mpp.distribute(nbrKDelta.ref());
```

```
tmp<scalarField> myKDelta = kappa(*this)*patch().deltaCoeffs();
```

The value of the contactRes_ variable is calculated when constructor is called, based on the thickness and conductivity of the layers. If it is equal to zero, it means that there is no thermal resistance between two regions and the boundary value is based on \( k \) and \( \Delta \) of the neighbour cell. At the end, we have calculated the nbrIntFld which is the temperature of the neighbour cell, nbrKDelta which is the \( k\Delta \) of the neighbour field (or the contact resistance), and myKDelta which is the \( k\Delta \) of the current cell. These three variables will be used to set the boundary condition. It should be noted that the same code will be used for both sides on a coupled patch, so equal values and gradients should be obtained by running this code for both sides of the patch.

Finally, using the values calculated above, the values of the refValue, refGrad, and valueFraction for the mixed boundary patch can be identified as following, and the boundary can be updated:

```c
this->refValue() = nbrIntFld();
this->refGrad() = 0.0;
```


this->valueFraction() = nbrKDelta()/ (nbrKDelta() + myKDelta());

mixedFvPatchScalarField::updateCoeffs();

We want to make sure that the above code has the same physical meaning as the eq. 2.11 which was explained in chapter 2. Assume we are running the code for side 1, so myKDelta is $k_1 \Delta_1$, nbrKDelta is $k_2 \Delta_2$, and nbrIntFld is $T_{c,2}$. Therefore, based on the above code, the $x$, $\nabla_{\perp} x$, and $w$ in eq. 4.2 are identified as:

$$x = refValue = T_{c,2} \quad (4.3)$$

$$\nabla_{\perp} x = refGrad = 0.0 \quad (4.4)$$

$$w = valueFraction = \frac{k_2 \Delta_2}{k_1 \Delta_1 + k_2 \Delta_2} \quad (4.5)$$

Inserting these three values in eq. 4.2 and having $x_p = T_p$ leads to:

$$T_p = \frac{k_2 \Delta_2}{k_1 \Delta_1 + k_2 \Delta_2} T_{c,2} + \left(\frac{k_1 \Delta_1}{k_1 \Delta_1 + k_2 \Delta_2}\right) (T_{c,1} + 0.0) \quad (4.6)$$

Which simply can be shown that it is identical to eq. 2.11. If we write the equations for the second side, we will get exactly the same results, therefore this boundary gives identical temperature and temperature gradients on both sides of the coupled patch.
In this chapter, we are going to implement a new solver named multiRegionReactingFoam. This solver is based on the reactingFoam solver with the ability to couple one solid region to the main fluid domain. The step-by-step guide to create this solver is presented in the first section and in the second section, using the solver to run a simple tutorial case is explained.

5.1 Implementation

Start by copying the original reactingFoam solver to the user application directory and removing the other two solver directories inside.

```
foam
cp --parents -r applications/solvers/combustion/reactingFoam/ $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/applications/solvers/combustion/reactingFoam/
rm -r rhoReacting*
```

Change the name of the directory and the main file, and update the Make/files accordingly. Then compile the code to make sure it works in the current state.

```
cd ..
mv reactingFoam multiRegionReactingFoam
cd multiRegionReactingFoam
mv reactingFoam.C multiRegionReactingFoam.C
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
sed -i s/reactingFoam/multiRegionReactingFoam/g Make/files
wclean
wmake
```

Open the main file (multiRegionReactingFoam.C) and add the following libraries to the list of libraries included before the main function.

```
// Solid region libraries
#include "solidRegionDiffNo.H"
#include "solidThermo.H"
```

In the main function, after the argList::addNote and before the turbulence->validate, there is a set of .H files which are included. Replace them with the following lines below. Only three lines are modified in the new set, and a comment is added before each modification so you can easily find them.
// To be able to use postProcess
#define CREATE_MESH createMeshes.H
#include "postProcess.H"
#include "addCheckCaseOptions.H"
#include "setRootCaseLists.H"
#include "createTime.H"

// To create more than one mesh
#include "createMeshes.H"
#include "createControl.H"
#include "createTimeControls.H"

// To read the max diffusion number
#include "readSolidTimeControls.H"
#include "initContinuityErrs.H"
#include "createFields.H"
#include "createFieldRefs.H"

Based on the above lines, we need two new .H files, named createMeshes.H and readSolidTimeControls.H. Create a new file named createMeshes.H and put the following code inside it and save the file.

Foam::Info
<< "Create fluid mesh for time = "
<< runTime.timeName() << Foam::nl << Foam::endl;

Foam::fvMesh mesh
(
    Foam::IOobject
    ("fluid",
     runTime.timeName(),
     runTime,
     Foam::IOobject::MUST_READ)
);

Foam::Info
<< "Create pyrolysis mesh for time = "
<< runTime.timeName() << Foam::nl << Foam::endl;

Foam::fvMesh pyrolysisMesh
(
    Foam::IOobject
    ("pyrolysisRegion",
     runTime.timeName(),
     runTime,
     Foam::IOobject::MUST_READ)
);

The above code creates two mesh variables. The first one is called "mesh" similar to normal createMesh file. However, the name of the mesh region is "fluid", so in the constant directory of the case, instead of polyMesh, the code will look for a fluid directory which contains the mesh of this
The second mesh variable is named "pyrolysisMesh" and the region name is "pyrolysisRegion" (the word pyrolysis is used for solid region because the idea is to expand the code later for pyrolysis simulation).

The second .H file can be copied from the chtMultiRegionFoam solver. Since we need also some other files from the same solver, we can create a solid directory here and copy them altogether.

```
mkdir solid
cp $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/solid/readSolidTimeControls.H solid/
cp $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/solid/solidRegionDiffNo.H solid/
cp $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/solid/solidRegionDiffNo.C solid/
cp $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/solid/solidRegionDiffusionNo.H solid/
cp $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/include/setMultiRegionDeltaT.H solid/
```

Continue with the modification of the main function. Inside the time loop, the if statement for deltaT setting should be replaced by the following lines. The original setDeltaT.H is replaced by two other .H files which are marked by a comment line in the new code.

```
if (LTS)
{
    #include "setRDeltaT.H"
}
else
{
    #include "compressibleCourantNo.H"
    // To include solidRegion in decision over deltaT
    #include "solidRegionDiffusionNo.H"
    #include "setMultiRegionDeltaT.H"
}
```

Add a line to include solidEEqn.H inside the pimple loop so the initial lines of the pimple loop will be like the following.

```
while (pimple.loop())
{
    #include "UEqn.H"
    #include "YEqn.H"
    #include "EEqn.H"
    // To solve the solid energy equation
    #include "solidEEqn.H"
```

Create a new empty file named solidEEqn.H in the solid sub-directory and copy the following lines into that file. This code will solve the eq. 2.6 for the solid region.

```
volScalarField& h = pyroThermo.he();
pyroRho = pyroThermo.rho();

eqnlScalarMatrix hEqn
{
    fvm::ddt(pyroRho, h)
    - fvm::laplacian(pyroThermo.alpha(), h)
    == fvOptions(pyroRho, h);
}
```
hEqn.relax();
fvOptions.constrain(hEqn);
hEqn.solve();
fvOptions.correct(h);

pyroThermo.correct();

Info<< "Solid Min/max T:" << min(pyroThermo.T()).value() << " "
      << max(pyroThermo.T()).value() << endl;
}

We still need to add the field variables for the solid region. Create another empty file named createSolidFields.H in the solid sub-directory and copy the following lines into that file.

Info<< "Reading solid thermophysical properties\n" << endl;
autoPtr<solidThermo> pPyroThermo(solidThermo::New(pyrolysisMesh));
solidThermo& pyroThermo = pPyroThermo();

volScalarField pyroRho
(
    IOobject
    (
        "solidRho",
        runTime.timeName(),
        pyrolysisMesh
    ),
    pyroThermo.rho()
);

Add the following two lines to the last line of the original createFields.H to call the recently created file.

// To create the solid fields
#include "createSolidFields.H"

You also need to replace the contents of the solid/solidRegionDiffusionNo.H file with the following lines, which is the simplified version of the original file, for single solid region.

scalar DiNum = -GREAT;
{
    tmp<volScalarField> magKappa;
    magKappa = pyroThermo.kappa();

    tmp<volScalarField> tcp = pyroThermo.Cp();
    const volScalarField& cp = tcp();

    tmp<volScalarField> trho = pyroThermo.rho();
    const volScalarField& rho = trho();

    DiNum = solidRegionDiffNo
    (}
5.1. IMPLEMENTATION

The final step is to modify the Make/files and Make/options in order to be able to compile the new solver. Replace the contents of the Make/files with the following (or simply add the first line).

```plaintext
solid/solidRegionDiffNo.C
multiRegionReactingFoam.C
```

```plaintext
EXE = $(FOAM_USER_APPBIN)/multiRegionReactingFoam
```

And then modify the contents of the Make/options with the following (or add the first two lines in EXE_INC and the first line in the EXE_LIBS).

```plaintext
EXE_INC = \
-$(LIB_SRC)/solidThermo/inc \n-$(LIB_SRC)/finiteVolume/inc \n-$(LIB_SRC)/meshTools/inc \n-$(LIB_SRC)/TurbulenceModels/TurbulenceModels/inc \n-$(LIB_SRC)/TurbulenceModels/compressible/inc \n-$(LIB_SRC)/TurbulenceModels/specie/inc \n-$(LIB_SRC)/TurbulenceModels/reactionThermo/inc \n-$(LIB_SRC)/transportModels/compressible/inc \n-$(LIB_SRC)/transportModels/basic/inc \n-$(LIB_SRC)/thermophysicalModels/chemistryModel/inc \n-$(LIB_SRC)/ODE/inc \n-$(LIB_SRC)/combustionModels/inc
```

```plaintext
EXE_LIBS = \
-lsolidThermo \n-lfiniteVolume \n-lfvOptions \n-lmeshTools \n-lsampling \n-lTurbulenceModels \n-lcompressibleTurbulenceModels \n-lreactionThermophysicalModels \n-lspecie \n-lcompressibleTransportModels \n-lfluidThermophysicalModels \n-lchemistryModel \n-lODE \n-lcombustionModels
```

You should be able to clean and compile the code now by the following commands.

```plaintext
wclean
wmake
```

If there is no error, you can follow the steps in the next section to run a simple tutorial case.
5.2 Run a tutorial case

The tutorial case is similar to the DLR.A.LTS which is already discussed in chapter 3. However, some changes are made to the initial geometry to make it more simple so the simulation time will be shorter. In the new case which is named multiRegionDLR, a large solid particle is placed in front of the fuel and air jets. The idea is to investigate the reactions around the solid particle and the heat transfer from hot gas to the solid. You can download the tutorial case, and run the case simply by using the Allrun script and check the results with paraview.

```
./Allrun
paraFoam &
```

With the default settings, the simulation might take around two minutes to finish, using six cores on a core-i7 computer. There is also a state file named "paraViewState.pvsm" in the case directory, which you can use to view some of the simulation results. For instance, the results at the final timestep with the default settings (at $t = 0.5\ s$) are presented in Figure 5.1.

![Figure 5.1: Final results of the simulation of the tutorial case. From left to right, temperature, velocity, pressure, and O2 mass fraction.](image)

The solid rectangular block can be easily detected in the above figures. The temperature field and pressure fields are actually available for the solid region, but since there are no U and O2 fields in the solid region, paraview decides the color of that region. The pressure is always constant for the solid region because we did not solve a pressure equation for that region. However, the temperature in the solid region is increasing because it is coupled to the fluid region. Due to the short duration of the simulation ($t = 0.5\ s$), the temperature raise in the solid region is so little and it cannot be observed in Figure 5.1. Therefore, the temperature of only the solid region, at the same time but in a different scale is presented in Figure 5.2.
5.2. RUN A TUTORIAL CASE

CHAPTER 5. MULTIREGIONREACTINGFOAM

Figure 5.2: Final temperature distribution in the large solid particle.
Bibliography


Study questions

1. What is the purpose of different terms in momentum equation in reactingFoam?

2. What is the purpose of different terms in energy equation in reactingFoam?

3. Explain the main purpose of each file in constant and system directories, for a tutorial case of the reactingFoam.

4. Compare the momentum and energy equations between the fluid regions in chtMultiRegion-Foam to those in the reactingFoam.

5. In a reactingFoam tutorial case, how do you change the reaction mechanism which is used?

6. How do you modify a currently existing mesh to create new regions (using utilities)?

7. What is the purpose of the regionProperties file in the constant directory of a multiRegion case?

8. Which new libraries are necessary to be added to reactingFoam, to have more than one regions?

9. How do you modify the createMesh.H if you want to add another region to your solver code?