CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

A chtMultiRegionSimpleFoam tutorial

Developed for OpenFOAM-2.1.x
Requires: ..

Author:
Maaike van der Tempel

Peer reviewed by:

October 22, 2012

Introduction

This tutorial describes how to simulate heat transfer between solids and fluids in OpenFOAM. It shows how to pre-process, run and post-process a basic steady state case involving the conjugate heat transfer between different regions in a 2D domain. A link between theory and model will be made.
We will take a closer look at the used solver `chtMultiRegionSimpleFoam`, and based on the theoretical background we will define situations where for buoyancy in fluid the Boussinesq approximation for incompressible fluids can be used.
An approach fo a new solver, `chtMultiRegionSimpleBoussinesqFoam` is presented, and possible other ways of creating it.

Chapter 1

Heat Transfer

1.1 Heat Transfer

Heat transfer occurs through three mechanisms: conduction, radiation and convection.
Steady heat conduction is described by a Laplace equation while unsteady conduction is governed by the heat equation.
In most applications, heat conduction in a solid needs to be considered along with heat convection in an adjacent fluid: the **conjugate heat transfer** problems. These problems must be solved by iterating between the equations describing the two types of heat transfer must. In addition radiation can be added.
In flows accompanied by heat transfer, the fluid properties are normally temperature-dependent, such as the density. Variations in density can be the cause of the fluid motion.
In such a case, the properties are calculated using the temperature on its current iteration, then the temperature is updated, etc. More important is that for these compressible flows, energy and momentum equations are coupled and must be solved simultaneously.
When density variation stays limited, the density can be treated as constant in the unsteady and convection terms, and as variable only in the gravitational term: this is the **Boussinesq approximation**. The density is then assumed to vary linearly with temperature. Density variations due to thermal expansion are then given by:

$\Delta\rho = \rho_0 \beta \Delta T$
where
ρ_0 is the reference density
β is the coefficient of thermal expansion, and
ΔT is the temperature difference across the fluid.

This approximation allows to solve the equations with methods for incompressible flow. It introduces errors of the order of 1 % if the temperature difference is smaller then 15° for air. For natural ventilation cases, where temperature differences remain small, is the Boussinesq approximation thus a valid approximation to speed up numerical simulation.

1.2 Heat Transfer in OpenFOAM

In OpenFOAM 2.1.x, several solvers can be found to simulate heat transfer.
Heat transfer in solids can be simulated by solving the Laplace equation for the conductive heat transfer in a solid (`laplacianFoam`).
For fluids, different approaches are possible depending on if the fluid is considered compressible or not. The buoyancy problems in compressible flows can be solved using the `buoyantPimpleFoam`

solver for transient cases and `buoyantSimpleFoam` solver for steady state cases.
When the fluid is presumed to be an incompressible flow using the Boussinesq approximation the `buoyantBoussinesqPimpleFoam` solver (transient) `buoyantBoussinesqSimpleFoam` solver (steady state) are present in OpenFOAM 2.1.x.

Solvers that combine other solvers for solid and fluid regions to calculate the conjugated heat transfer are also present. In 2.1.x, a transient (`chtMultiRegionFoam`) and a steady state solver (`chtMultiRegionSimpleFoam`) are already included. These two solvers consider the fluid regions as compressible flows.

For small temperature differences - as f.e. in natural convection, the Boussinesq approximation is a very accurate approximation. It is therefore interesting to create a new solver that considers the fluid region as an incompressible flow, based on the `chtMultiRegionSimpleFoam` solver and the `buoyantBoussinesqSimpleFoam` solver already present in the solver library.

In the following we will only focus on the steady state solver `chtMultiRegionSimpleFoam`. We will demonstrate the solver using a well known plane wall problem that can be found on the following website to start.

`http://openfoamwiki.net/index.php/Getting_started_with_chtMultiRegionSimpleFoam`
`_-_planeWall2D`

Chapter 2

chtMultiRegionSimpleFoam

The solver chtMultiRegionSimpleFoam is the steady-state version of chtMultiRegionFoam. Heat conduction in solids (laplacianFoam) is combined with a buoyantFoam solver for conjugate heat transfer between solid region and fluid regions.

2.1 Solver: chtMultiRegionSimpleFoam

Go to the solver's directory

```
cd $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/chtMultiRegionSimpleFoam
ls
```

2.1.1 chtMultiRegionSimpleFoam.C

The directory contains the main file chtMultiRegionSimpleFoam.C, and a \Make ,fluid and \solid directory. Several *.H files are included in the main file, each containing code included by default in OpenFOAM.

```
\*---------------------------------------------------------------------------*/
#include "fvCFD.H"
#include "basicRhoThermo.H"
#include "turbulenceModel.H"
#include "fixedGradientFvPatchFields.H"
#include "regionProperties.H"
#include "basicSolidThermo.H"
#include "radiationModel.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"
...
// ************************************************************************* //
```

A list of the included .H files and their purposes is given below:

fvCFD.H - standard file for for finite volume method
`$FOAM_SRC/finiteVolume/cfdTools/general/include/fvCFD.H`

basicRhoThermo.H - Basic thermodynamic properties based on density
`$FOAM_SRC/thermophysicalModels/basic/rhoThermo/basicRhoThermo/basicRhoThermo.H`

turbulenceModel.H - Abstract base class for compressible turbulence models
`$FOAM_SRC/turbulenceModels/compressible/turbulenceModel/turbulenceModel.H`

fixedGradientFvPatchFields.H `$FOAM_SRC/finiteVolume/fields/fvPatchFields/basic/fixedGradient /fixedGradientFvPatchFields.H`

regionProperties.H - Simple class to hold region information for coupled region simulations
`$FOAM_SRC/turbulenceModels/compressible/turbulenceModel/derivedFvPatchFields /turbulentTemperatureCoupledBaffle/regionProperties.H`

basicSolidThermo.H - Basic solid thermodynamic properties
`$FOAM_SRC/thermophysicalModels/basicSolidThermo/basicSolidThermo/basicSolidThermo.H`

radiationModel.H - Namespace for radiation modelling
`$FOAM_SRC/thermophysicalModels/radiationModels/radiationModel/radiationModel/radiationModel.H`

setRootCase.H - checks folder structure of the case
`$FOAM_SRC/OpenFOAM/include/setRootCase.H`

createTime.H - check runtime according to the controlDict and initiates time variables
`$FOAM_SRC/OpenFOAM/include/createTime.H`

createFluidMeshes.H - defines fluid region in domain
`$FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/chtMultiRegionSimpleFoam/fluid/createFluidMeshes.H`

createFluidFields.H - creates the fields for the fluid region: rho, kappa, U, phi, g, turbulence, gh, ghf, p rgh
`$FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/chtMultiRegionSimpleFoam/fluid/createFluidFields.H`

initContinuityErrs.H - declare and initialise the cumulative continuity error
`$FOAM_SRC/finiteVolume/cfdTools/general/include/initContinuityErrs.H`

The second part of the `chtMultiRegionSimpleFoam.C` file contains the solver loop. Here the runTime field is set. Then, for each region the region fields are set using the last timestep (`setRegionFluidFields.H`), the algorithm is checked (`readFluidMultiRegionSIMPLEControls.H`) and the equations are loaded and solved by the solver (in `solveFluid.H` and `solveSolid.H`).

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
    while (runTime.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        forAll(fluidRegions, i)
        {
            Info<< "\nSolving for fluid region "
                << fluidRegions[i].name() << endl;
            #include "setRegionFluidFields.H"
            #include "readFluidMultiRegionSIMPLEControls.H"
            #include "solveFluid.H"
        }
        forAll(solidRegions, i)
        {
            Info<< "\nSolving for solid region "
                << solidRegions[i].name() << endl;
            #include "setRegionSolidFields.H"
            #include "readSolidMultiRegionSIMPLEControls.H"
            #include "solveSolid.H"
        }

        runTime.write();

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

2.1.2 SolveFluid.H

```
//  Pressure-velocity SIMPLE corrector
    p_rgh.storePrevIter();
    rho.storePrevIter();
    {
        #include "UEqn.H"
        #include "hEqn.H"
        #include "pEqn.H"
    }

    turb.correct();
```

In `solveFluid.H` we see that the fluid region is solved using `buoyantSimpleFoam`. For each iteration, the density en pressure from the former iteration is used.
The momentum equation is solved in `UEqn.H`.

```
    // Solve the Momentum equation
    tmp<fvVectorMatrix> UEqn
    (
        fvm::div(phi, U)
      + turb.divDevRhoReff(U)
    );
    UEqn().relax();
    solve
    (
        UEqn()
     ==
        fvc::reconstruct
        (
            (
              - ghf*fvc::snGrad(rho)
              - fvc::snGrad(p_rgh)
            )*mesh.magSf()
        )
    );
```

The heat equation in `hEqn.H`.

```
{
    fvScalarMatrix hEqn
    (
        fvm::div(phi, h)
      - fvm::Sp(fvc::div(phi), h)
      - fvm::laplacian(turb.alphaEff(), h)
     ==
      - fvc::div(phi, 0.5*magSqr(U), "div(phi,K)")
      + rad.Sh(thermo)
    );
    hEqn.relax();
    hEqn.solve();
    thermo.correct();
    rad.correct();
    Info<< "Min/max T:" << min(thermo.T()).value() << ' '
        << max(thermo.T()).value() << endl;
}
```

The pressure equation can be found in `pEqn.H`. It solves the pressure, corrects the momentum velocities according to the new pressure field and adjusts the pressure level to obey overall mass continuity. It ends by updating the thermal conductivity.

```
{
    rho = thermo.rho();
    rho = max(rho, rhoMin[i]);
    rho = min(rho, rhoMax[i]);
    rho.relax();

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

    p_rgh.boundaryField().updateCoeffs();
    U = rAU*UEqn().H();
    UEqn.clear();

    phi = fvc::interpolate(rho)*(fvc::interpolate(U) & mesh.Sf());
    bool closedVolume = adjustPhi(phi, U, p_rgh);
    dimensionedScalar compressibility = fvc::domainIntegrate(psi);
    bool compressible = (compressibility.value() > SMALL);

    surfaceScalarField buoyancyPhi(rhorAUf*ghf*fvc::snGrad(rho)*mesh.magSf());
    phi -= buoyancyPhi;

    // Solve pressure
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rhorAUf, p_rgh) == fvc::div(phi)
        );

        p_rghEqn.setReference
        (
            pRefCell,
            compressible ? getRefCellValue(p_rgh, pRefCell) : pRefValue
        );

        p_rghEqn.solve();

        if (nonOrth == nNonOrthCorr)
        {
            // Calculate the conservative fluxes
            phi -= p_rghEqn.flux();

            // Explicitly relax pressure for momentum corrector
            p_rgh.relax();

            // Correct the momentum source with the pressure gradient flux
            // calculated from the relaxed pressure
            U -= rAU*fvc::reconstruct((buoyancyPhi + p_rghEqn.flux())/rhorAUf);
            U.correctBoundaryConditions();
        }
    }

    p = p_rgh + rho*gh;

    #include "continuityErrs.H"

    // For closed-volume cases adjust the pressure level
    // to obey overall mass continuity
    if (closedVolume && compressible)
    {
        p += (initialMass - fvc::domainIntegrate(thermo.rho()))
            /compressibility;
        p_rgh = p - rho*gh;
    }

    rho = thermo.rho();
    rho = max(rho, rhoMin[i]);
    rho = min(rho, rhoMax[i]);
    rho.relax();

    Info<< "Min/max rho:" << min(rho).value() << ' '
        << max(rho).value() << endl;

    // Update thermal conductivity
    kappa = thermo.Cp()*turb.alphaEff();
}
```

2.1.3 SolveSolid.H

The solid region is solved by solving the laplacian equation.

```
{
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix tEqn
        (
            -fvm::laplacian(kappa, T)
        );
        tEqn.relax();
        tEqn.solve();
    }

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

2.2 Differences between cht in fluid and buoyantSimpleFoam

In order to understand how the coupling between the fluid and solid regions is handled in `chtMultiRegionSimpleFoam`, the differences and analogy between the implementation of the fluid part of the solver and `bouyantSimpleFoam` will be discussed in this section.

2.2.1 C-files

When comparing the two main files of the solvers, differences can be found in the included header files.
In `chtMultiRegionSimpleFoam`, different turbulence models can be used, while in `buoyantSimpleFoam`, the RASmodel is specified.
In `chtMultiRegionSimpleFoam.C`, the used thermodynamic properties are based on density (`basicRhoThermo.H`), while in `buoyantSimpleFoam.C` the used thermodynamic properties are based on compressibility (`basicPsiThermo.H`). `basicRhoThermo.H` is valid for both liquids and gases, but in the case of ideal gases, the only difference between the two is that the density rhoThermo is updated once every time step and psiThermo will always use the up-to-date pressure field. rhoThermo also includes the gravitational acceleration, so no extra `readGravitationalAcceleration.H` file is included in the cht-solver.
Additional information on the two different models can be found on the OpenFOAM® -Extend Discussion Board

`http://www.extend-project.de/user-forums/extend-groups/`
`16-special-interest-group-on-general-chemistrythermo-group-forum`

For `chtMultiRegionSimpleFoam`, the convergence information and checks for the SIMPLE loop are done inside the loop and read from `readFluidMultiRegionSIMPLEControls.H`.
In both solvers, the mesh and the fields must be assigned. This is done in the cht-solver for each region individually and tin a way that the solver knows where to look in the structure of a case for the specific region through `regionProperties.H`.

```
    #include "createFluidMeshes.H"
    #include "createSolidMeshes.H"
    #include "createFluidFields.H"
    #include "createSolidFields.H"
```

In `createFluidFields.H` in `chtMultiRegionSimpleFoam/fluid/` the pointer lists are initialized and populated so that the correct thermophysical properties can be assigned.

```
// Initialise fluid field pointer lists
    PtrList<basicRhoThermo> thermoFluid(fluidRegions.size());
    PtrList<volScalarField> rhoFluid(fluidRegions.size());
    PtrList<volScalarField> kappaFluid(fluidRegions.size());
    PtrList<volVectorField> UFluid(fluidRegions.size());
...
 // Populate fluid field pointer lists
    forAll(fluidRegions, i)
    {
        Info<< "*** Reading fluid mesh thermophysical properties for region "
            << fluidRegions[i].name() << nl << endl;

        Info<< "    Adding to thermoFluid\n" << endl;

        thermoFluid.set
        (
            i,
            basicRhoThermo::New(fluidRegions[i]).ptr()
        );

        Info<< "    Adding to rhoFluid\n" << endl;
        rhoFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                    "rho",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
                ),
                thermoFluid[i].rho()
            )
        );
                ...
        }
```

The thermodynamical properties are read from the startTime directory of the region and set or computed if not available. The `setRegionFluidFields.H` file is the crucial step before starting to solve the equations. As this header file is included in the loop, the properties will be updated taking into account the adjacent region before the next loop. The fluid fields are now written to the variables of the different equations.

```
    const fvMesh& mesh = fluidRegions[i];

    basicRhoThermo& thermo = thermoFluid[i];
    volScalarField& rho = rhoFluid[i];
    volScalarField& kappa = kappaFluid[i];
    volVectorField& U = UFluid[i];
    surfaceScalarField& phi = phiFluid[i];
    compressible::turbulenceModel& turb = turbulence[i];

    volScalarField& p = thermo.p();
    const volScalarField& psi = thermo.psi();
    volScalarField& h = thermo.h();

    const dimensionedScalar initialMass
    (
        "initialMass",
        dimMass,
        initialMassFluid[i]
    );

    radiation::radiationModel& rad = radiation[i];

    const label pRefCell = pRefCellFluid[i];
    const scalar pRefValue = pRefValueFluid[i];

    volScalarField& p_rgh = p_rghFluid[i];
    const volScalarField& gh = ghFluid[i];
    const surfaceScalarField& ghf = ghfFluid[i];
```

To further understand how the`chtMultiRegionSimpleFoam` solver works, we will run and discuss a basic case in heat transfer: the Plane Wall.

2.3 Tutorial Plane Wall case

In this tutorial, we will solve a one-dimensional, steady state situation where heat transferred.
This case is an example shown in the book *Fundamentals of Heat and Mass Transfer* by Frank P. Incropera et. al, chapter 3, section 3.1 "The Plane Wall". It is basically a wall, with hot and warm air flowing next to it, which results in a temperature gradient in the wall. In a 1D systems, temperature gradients (and thus heat transfer) occur only in 1 direction.

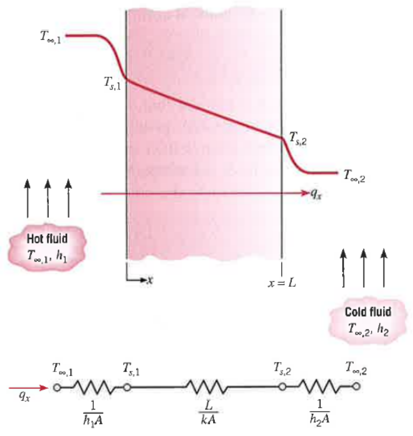

FIGURE 3.1 Heat transfer through a plane wall. (*a*) Temperature distribution. (*b*) Equivalent thermal circuit.

Figure 2.1: PlaneWall, chapter 3.1,p.89

The heat flux through the wall is then given by
$q = h_1(T_w - T_{ws}) = \frac{k}{L}(T_{ws} - T_{cs}) = h_2(T_{cs} - T_c)$
where
h_1 is the heat transfer coefficient at the surface of the wall with the warm fluid
h_2 is the heat transfer coefficient at the surface of the wall with the cold fluid
T_w is the temperature of the warm fluid
T_{ws} is the temperature of the surface at the warm fluid side
T_{wc} is the temperature of the surface at the cold fluid side
T_c is the temperature of the cold fluid
k is the thermal conductivity of the solid
L is the width of the solid wall

2.3.1 Getting started

Copy the tutorial planeWall2D.tar.gz to the run directory, unpack and go to the case.

```
tar xzf planeWall2D.tar.gz
cd planeWall2D
tree
```

```
|---0
|   |---epsilon
|   |---k
|   |---p
|   |---p_rgh
|   |---T
|   |---U
|   |---Ychar
|   |---Ypmma
|---Allclean
|---Allrun
|--- constant
|   |---bottomAir
|   |   |---g
|   |   |---radiationProperties
|   |   |---RASProperties
|   |   |---thermophysicalProperties
|   |   |---turbulenceProperties
|   |---polyMesh
|   |   |---blockMeshDict
|   |   |---boundary
|   |---regionProperties
|   |---topAir
|   |   |---g
|   |   |---radiationProperties
|   |   |---RASProperties
|   |   |---thermophysicalProperties
|   |   |---turbulenceProperties
|   |---wall
|   |   |---solidThermophysicalProperties
|--- system
    |---bottomAir
    |   |---changeDictionaryDict
    |   |---decomposeParDict
    |    |---fvSchemes
    |    |---fvSolution
    |---controlDict
    |---decomposeParDict
    |---fvSchemes
    |---fvSolution
    |---README
    |---topAir
    |    |---changeDictionaryDict
    |    |---decomposeParDict
    |    |---fvSchemes
    |    |---fvSolution
    |---topoSetDict
    |---wall
        |---changeDictionaryDict
        |---decomposeParDict
        |---fvSchemes
        |---fvSolution
```

The file structure of the planeWall2D case is similar to other OpenFOAM tutorials where the case directory has a `/0`, `/constant` and `/system` directory.

The `/0` folder contains all empty initial and conditions of fluid and solid regions. It is only after creating the different regions, that this folder will contain subfolders per region with all initial conditions.

In the `/constant` folder we can find the mesh setup in `/constant/polyMesh`. We also find directories per region with all transport, thermophysical properties and turbulence properties specific for that region. The `/regionProperties` file contains the different fluidRegionNames and solidRegionNames needed to find the different regions when preprocessing and running the case.

As usual in OpenFOAM tutorials; the solver-, write- and time-control can be found in the `/system`. As different regions will be defined, a `toposetsDict` file is included in the `/system` directory.
The initial conditions per region can be found in `/system/regionname/changeDictionaryDict`. When running the application changeDictionary, OpenFOAM will look here for all initial conditions for that region and change them per region in the /verb+/0+ directory.
Below, the initial conditions for the velocity U for the fluid region bottomAir as mentioned in `/system/bottomAir/changeDictionaryDict`. are given

```
dictionaryReplacement
{
    U
    {
        internalField   uniform (0.1 0 0);

        boundaryField
        {
            leftLet
            {
                type            fixedValue;
                value           uniform ( 0.1 0 0 );
            }
            rightLet
            {
                type            inletOutlet;
                inletValue      uniform ( 0 0 0 );
                value           uniform ( 0.1 0 0 );
            }

            bottomAir_bottom
            {
                type            symmetryPlane;
            }

            "bottomAir_to_.*"
            {
                type            fixedValue;
                value           uniform (0 0 0);
            }
        }
    }
}
```

The bottomAir_to.* refers to the interface between the bottomAir-region and the solid region Wall. These are the baffles in the orginial mesh with mappedWall patches.

2.3.2 Geometry and Initial Conditions

The planeWall case in OpenFOAM consists of an infinite solid and uniform wall with generic fluids flowing on both sides of the wall, at different temperature. The geometry is based on the cavity case geometry, and consists of a 1m x 1m x 0.1m mesh divided into 3 regions: two fluid regions and one solid region.
The 3 regions are defined as follows:

topAir 0.6 to 1m - where the air is colder at 300K and flows from the left to the right at 0.1m/s. The top patch is a symmetry plane.

wall 0.4 to 0.6m - where a solid wall is placed, initiated at 300K and has a high conduction factor.

bottomAir 0 to 0.4m - where the air is hot at 500K and flows from the left to the right at 0.1m/s. The bottom patch is a symmetry plane.

2.3.3 Run and Post processing

Run the case in the terminal using the Allrun file.

```
runApplication blockMesh
runApplication topoSet
runApplication splitMeshRegions -cellZones -overwrite

remove fluid fields from solid regions
remove solid fields from fluid regions

for i in bottomAir topAir wall
do
   changeDictionary -region $i > log.changeDictionary.$i
done
runApplication chtMultiRegionSimpleFoam
```

```
paraFoam -touchAll
```

With `blockMesh` the base mesh is created. With `topoSet` the different cellsets per region are made, by `splitMeshRegions` the different regions are created. As in the `/0` directory both solid and fluid fields were included, the fluid fields must be removed from the solid regions and vice versa.
In a next step, the initial conditions are changed. The case is now ready to run the solver.

We now take a look at the log file `log.chtMultiRegionSimpleFoam` created while solving the case.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Create time
Create fluid mesh for region bottomAir for time = 0
Create fluid mesh for region topAir for time = 0
Create solid mesh for region wall for time = 0

*** Reading fluid mesh thermophysical properties for region bottomAir
    Adding to thermoFluid
Selecting thermodynamics package
hRhoThermo<pureMixture<constTransport<specieThermo<hConstThermo<perfectGas>>>>>
    Adding to rhoFluid
    Adding to kappaFluid
    Adding to UFluid
    Adding to phiFluid
    Adding to gFluid
    Adding to turbulence
Selecting turbulence model type laminar
    Adding to ghFluid
    Adding to ghfFluid
Selecting radiationModel none

*** Reading fluid mesh thermophysical properties for region topAir
    Adding to thermoFluid
Selecting thermodynamics package
hRhoThermo<pureMixture<constTransport<specieThermo<hConstThermo<perfectGas>>>>>
    Adding to rhoFluid
    Adding to kappaFluid
    Adding to UFluid
    Adding to phiFluid
    Adding to gFluid
    Adding to turbulence
Selecting turbulence model type laminar
    Adding to ghFluid
    Adding to ghfFluid
Selecting radiationModel none

*** Reading solid mesh thermophysical properties for region wall
    Adding to thermos
Constructed constSolidThermo with
    rho         : rho [1 -3 0 0 0 0 0] 8000
    Cp          : Cp [0 2 -2 -1 0 0 0] 450
    K           : K [1 1 -3 -1 0 0 0] 80
    Hf          : Hf [0 2 -2 0 0 0 0] 1
    emissivity  : emissivity [0 0 0 0 0 0 0] 1
    kappa       : kappa [0 -1 0 0 0 0 0] 0
    sigmaS      : sigmaS [0 -1 0 0 0 0 0] 0
```

```

Time = 1

Solving for fluid region bottomAir
DILUPBiCG:  Solving for Ux, Initial residual = 1,
Final residual = 0.0003272254, No Iterations 1
DILUPBiCG:  Solving for Uy, Initial residual = 1,
Final residual = 0.0003192866, No Iterations 1
DILUPBiCG:  Solving for h, Initial residual = 1,
Final residual = 0.0117141, No Iterations 1
Min/max T:300.0643 500.0003
GAMG:  Solving for p_rgh, Initial residual = 0.7403903,
Final residual = 0.007175254, No Iterations 4
time step continuity errors : sum local = 0.0139927,
global = 0.001807746, cumulative = 0.001807746
Min/max rho:0.6951731 1.158375

Solving for fluid region topAir
DILUPBiCG:  Solving for Ux, Initial residual = 1,
Final residual = 0.0001938523, No Iterations 1
DILUPBiCG:  Solving for Uy, Initial residual = 1,
Final residual = 1.477592e-05, No Iterations 1
DILUPBiCG:  Solving for h, Initial residual = 1,
Final residual = 0.0008823631, No Iterations 1
Min/max T:299.9991 300.0006
GAMG:  Solving for p_rgh, Initial residual = 0.8957774,
Final residual = 0.005316061, No Iterations 3
time step continuity errors : sum local = 0.03498322,
global = 0.01049165, cumulative = 0.01229939
Min/max rho:1.15862 1.158626

Solving for solid region wall
DICPCG:  Solving for T, Initial residual = 1,
Final residual = 0.01573355, No Iterations 2
Min/max T:300 300.1412
ExecutionTime = 0.37 s  ClockTime = 1 s

...
```

We see that in the first the different fields per region are initialized and the different models (thermodynamic, turbulence, radiation) are added to the region. Then, the solver starts running.

As the different regions are solved, all regions should be added together to post-process the results and visualize them with paraview. The following files for post-processing were created:

```
planeWall2D{bottomAir}.OpenFOAM
planeWall2D{topAir}.OpenFOAM
planeWall2D{wall}.OpenFOAM
```

We can now post-process by opening paraview in the terminal and by loading the files in paraview instead of using the application paraFoam.

```
paraview
```

We must open the 3 different regions, and load the internal mesh and the temperature T. Select all 3 files in the Pipeline Browser, and apply a filter GroupDatasets to handle the data in the different regions together. Now we can visualize the temperature profile in the Y-access over the wall by applying the Filter PlotOverLine.

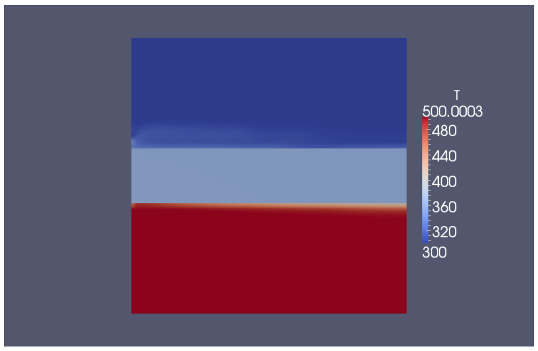

Figure 2.2: Temperature in Plane Wall case

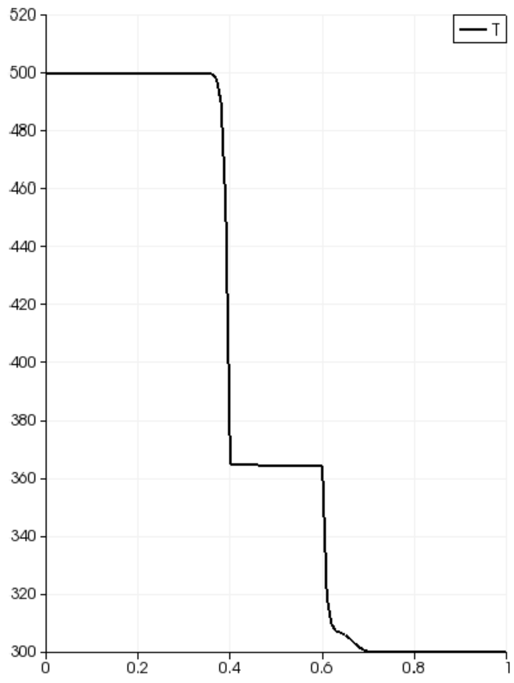

Figure 2.3: Temperature profile through Plane Wall case

We now see that the temperature gradient in the wall is very small, this means that our wall is highly conductive. The two curves in the graph are the result of heat convection at the surfaces of the wall.
The temperature at the warm side of the surface is
$T_{0.6} = 364.207K$
while the temperature at the cold side of the surface is
$T_{0.4} = 364.863K$
In `/constant/wall/solidThermophysicalProperties` we find the conductivity of our solid. The thermal conductivity is 80 W/mK.
As our wall is 0.2m thick, we can now easily calculate the heat flux through the wall:
$q = \frac{k}{L}(T_{0.4} - T_{0.6})$ to find the heat transfer coefficients on surfaces.

Here, this results in $h_warm = 2W/m^2K$, $h_cold = 4W/M^2K$ for a flow velocity of 0.1 m/s.

2.3.4 Using the case

This case can be used to check the simulated results of heat transfer coefficients along walls in buildings using CFD.

We will rerun the case , making some adjustments and see how a smaller temperature difference, a lower thermal conductivity of the wall en different wind speeds affect the heat transfer coefficients at both sides of the wall.

topAir 0.6 to 1m - where the air is colder at 273K and flows from the left to the right at 1m/s and at 4m/s. The top patch is a symmetry plane.

wall 0.4 to 0.6m - where a solid wall is placed, initiated at 273K and has a thermal conductivity of 5 W/mK, density of 2400 kg/m$^3 andthermalcapacityof880J/kgK(averageforconcrete).0to0.4m - wheretheairishotat293Kandflowsfromthelefttotherightat1m/sandat4m/s.Thebottompatchisasymmetryplane.$

bottomAir

wind speed m/s	Surface heat transfer W/m^2K	
U	h$_ws$	h$_c$
1	1.0	0.9
4	4.77	4.79

Table 2.1: Heat transfer coefficients on surfaces

Now we can revisualize the temperature profile in the Y-access over the wall by applying the Filter PlotOverLine.
The lower thermal conductivity results in a procentual bigger temperature gradient in the wall.

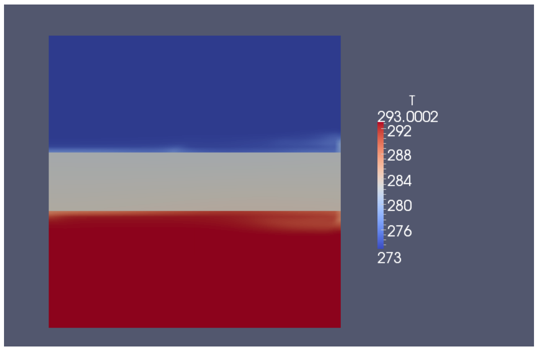

Figure 2.4: Temperature in Plane Wall case with 1m/s

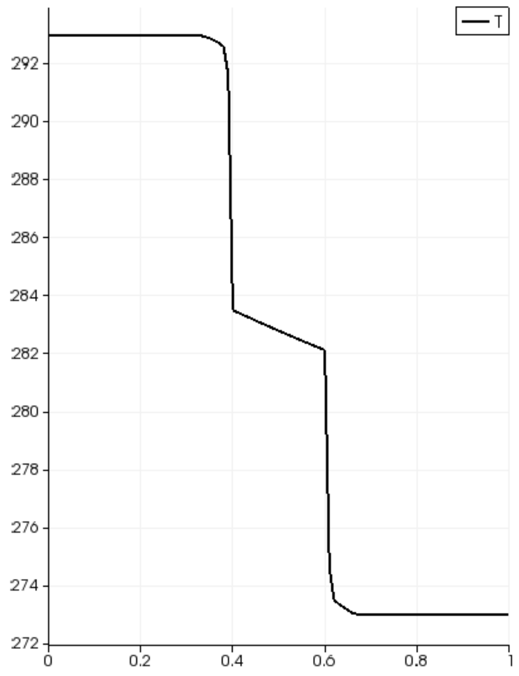

Figure 2.5: Temperature profile through new Plane Wall case at 1m/s

In European standards used in building physics for (ISO 6946:2007), values for the heat convection coefficients are given to calculate the heat flux through a wall for surfaces with a normal (high) emissivity, which can be used to benchmark a case.

Surface heat transfer	Direction of heat flow		
W/m^2K	Upwards	Horizontal	Downwards
h$_{si}$	10	7.7	6
h$_{se}$	25	25	25

Table 2.2: Standard heat transfer coefficients on surfaces

Chapter 3

chtMultiRegionSimpleBoussinesqFoam

As mentioned in our heat transfer introduction, the Boussinesq approximation is valid in heat transfer cases with air where temperature differences stay limited, as is the case for natural ventilation. It is therefore interesting to create a new solver that considers the fluid region as an incompressible flow, based on the `chtMultiRegionSimpleFoam` solver and the `buoyantBoussinesqSimpleFoam` solver already present in the solver library. This is a steady-state solver for buoyant, turbulent flow of incompressible fluids. It uses the Boussinesq approximation

$\rho_{eff} = 1 - \beta(T - T_{ref})$
where
ρ_{eff} is the effective (driving) density
β is the coefficient of thermal expansion [1/K]
T = temperature [K]
T_{ref} = reference temperature [K]
It is valid when $\rho_{eff} << 1$

3.1 buoyantBoussinesqSimpleFoam

We will take a closer look at the `buoyantBoussinesqSimpleFoam` solver first.

```
cd $FOAM_SOLVERS/heatTransfer/buoyantBoussinesqSimpleFoam
```

The main file of the solver `buoyantBoussinesqSimpleFoam.C` has a similar construction to the `buoyantSimpleFoam` solver, although some other header files are included and the heat equation `hEqn.H` has become the temperature equation `TEqn.H`.
There is no thermodynamic model included (as the fluid is incompressible).

```
/*---------------------------------------------------------------------------*\
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "RASModel.H"
#include "simpleControl.H"

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

    simpleControl simple(mesh);

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

    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        // Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "TEqn.H"
            #include "pEqn.H"
        }

        turbulence->correct();

        runTime.write();

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

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

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

singlePhaseTransportModel.H - A simple single-phase transport model based on viscosityModel, used by the incompressible single-phase solvers like simpleFoam, turbFoam etc.
`$FOAM_SRC/transportModels/incompressible/lnInclude/singlePhaseTransportModel.H`

RASModel.H - Namespace for incompressible RAS turbulence models.
`$FOAM_SRC/turbulenceModels/incompressible/RAS/RASModel/RASModel.H`

These header files should thus be included in the `chtMultiRegionSimpleBousinesqFoam`, and in the entire solver, the fluid should be considered as incompressible. All header-files etc. must thus be adapted to the incompressible fluid approach.

3.2 Changing chtMultiRegionSimpleFoam

Copy the `chtMultiRegionSimpleFoam` solver into the user directory. Go to the solver, and change the name of the solver into `chtMultiRegionSimpleBoussinesqFoam`.
Adapt the name in the main file and also the path in the `Make/files`.

```
OF21x
run
cd ..
cp -r $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/chtMultiRegionSimpleFoam/ ./applications/solvers/
cd applications/solvers
```

```
mv chtMultiRegionSimpleFoam/ chtMultiRegionSimpleBoussinesqFoam
cd chtMultiRegionSimpleBoussinesqFoam/
wclean
mv chtMultiRegionSimpleFoam.C chtMultiRegionSimpleBoussinesqFoam.C
sed -i s/"chtMultiRegionSimpleFoam"/"chtMultiRegionSimpleBoussinesqFoam"/g chtMultiRegionSimpleBoussin
sed -i s/"chtMultiRegionSimpleFoam"/"chtMultiRegionSimpleBoussinesqFoam"/g Make/files
sed -i s/"$FOAM_APPBIN"/"$FOAM_USER_APPBIN"/g Make/files
sed -i s/"FOAM_APPBIN"/"FOAM_USER_APPBIN"/g Make/files
```

The solid part of the `chtMultiRegionSimpleFoam` solver will be untouched. We will only focus on the fluid part. Let us start by adapting the fluid and then changing the main file `chtMultiRegionSimpleBoussinesqFoam.C`.

```
cd fluid
vi solveFluid.H
```

We will compare the files in our new solver with the existing solver. Open therefore a new terminal:

```
OF21x
sol
cd heatTransfer/buoyantBoussinesqSimpleFoam$
vi buoyantBoussinesqSimpleFoam.C
vi TEqn.H
```

Change in `solveFluid.H` hEqn.H into TEqn.H and replace the content.

```
mv hEqn.H TEqn.H
gedit TEqn.H
```

Open UEqn.H and replace the content by

```
    tmp<fvVectorMatrix> UEqn
    (
        fvm::div(phi, U)
      + turbulence->divDevReff(U)
    );

    UEqn().relax();

        solve
        (
            UEqn()
            ==
            fvc::reconstruct
            (
                (
                  - ghf*fvc::snGrad(rhok)
                  - fvc::snGrad(p_rgh)
                )*mesh.magSf()
            )
        );
```

We deleted the if loop present in the original file as in the cht solver the simple loop is present in the main file.

We now have changed the UEqn.H, TEqn.H, solveFluid.H. To change pEqn.H we must compare the file with the pEqn.H file of buoyantSimpleFoam, to highlight the specific parts of the chtMultiRegionFoam, and then adapt this to the Boussinesq approach.
This gives the following `pEqn.H` for our new solver:

```
{
    volScalarField rAU("rAU", 1.0/UEqn().A());
    surfaceScalarField rAUf("(1|A(U))", fvc::interpolate(rAU));

    p_rgh.boundaryField().updateCoeffs();

    U = rAU*UEqn().H();
    UEqn.clear();

    phi = fvc::interpolate(U) & mesh.Sf();
    adjustPhi(phi, U, p_rgh);

    surfaceScalarField buoyancyPhi(rAUf*ghf*fvc::snGrad(rhok)*mesh.magSf());
    phi -= buoyancyPhi;

    // Solve pressure
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rAUf, p_rgh) == fvc::div(phi)
        );

        p_rghEqn.setReference(pRefCell, getRefCellValue(p_rgh, pRefCell));

        p_rghEqn.solve();

        if (nonOrth == nNonOrthCorr)
        {
            // Calculate the conservative fluxes
            phi -= p_rghEqn.flux();

            // Explicitly relax pressure for momentum corrector
            p_rgh.relax();

            // Correct the momentum source with the pressure gradient flux
            // calculated from the relaxed pressure
            U -= rAU*fvc::reconstruct((buoyancyPhi + p_rghEqn.flux())/rAUf);
            U.correctBoundaryConditions();
        }
    }

    #include "continuityErrs.H"

    p = p_rgh + rhok*gh;

    if (p_rgh.needReference())
    {
        p += dimensionedScalar
        (
            "p",
            p.dimensions(),
            pRefValue - getRefCellValue(p, pRefCell)
        );
        p_rgh = p - rhok*gh;
    }
}
```

Now the equations for our fluid part are set. We now must adapt the fields.
No changes were made in `createFluidMeshes.H` and `readFluidMultiRegionSIMPLEControls.H`. We now have to change the `createFluidFields.H` and `setRegionFields.H`. We compare the existing files with the `createField.H` file of the buoyantSimpleFoam solver, and try to adapt the files to the Boussinesq approach. This gives following files:

```
    // Initialise fluid field pointer lists

    PtrList<volScalarField> TFluid(fluidRegions.size());
    PtrList<volScalarField> p_rghFluid(fluidRegions.size());
    PtrList<volVectorField> UFluid(fluidRegions.size());
    PtrList<volScalarField> kappatFluid(fluidRegions.size());
    PtrList<volScalarField> rhokFluid(fluidRegions.size());
    PtrList<surfaceScalarField> phiFluid(fluidRegions.size());
    PtrList<uniformDimensionedVectorField> gFluid(fluidRegions.size());
    PtrList<incompressible::RASModel> turbulence(fluidRegions.size());
    PtrList<singlePhaseTransportModel> TransportModel(fluidRegions.size());
    PtrList<dimensionedScalar> beta(fluidRegions.size());
    PtrList<dimensionedScalar> TRef(fluidRegions.size());
    PtrList<dimensionedScalar> Pr(fluidRegions.size());
    PtrList<dimensionedScalar> Prt(fluidRegions.size());

    PtrList<volScalarField> ghFluid(fluidRegions.size());
    PtrList<surfaceScalarField> ghfFluid(fluidRegions.size());
    PtrList<radiation::radiationModel> radiation(fluidRegions.size());
    PtrList<volScalarField> pFluid(fluidRegions.size());

    List<label> pRefCellFluid(fluidRegions.size(),0);
    List<scalar> pRefValueFluid(fluidRegions.size(),0.0);


    // Populate fluid field pointer lists
    forAll(fluidRegions, i)
    {
        Info<< "*** Reading fluid mesh thermophysical properties for region "
            << fluidRegions[i].name() << nl << endl;


        Info<< "    Adding to TFluid\n" << endl;
        TFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                    "T",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                fluidRegions[i]
            )
       );

        Info<< "    Adding to p_rghFluid\n" << endl;
        p_rghFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                    "p_rgh",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                fluidRegions[i]
            )
        );

     // kinematic turbulent thermal conductivity m2/s
        Info<< "    Adding to kappatFluid\n" << endl;
        kappaFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                    "kappat",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                fluidRegions[i]
            )
        );

        Info<< "    Adding to UFluid\n" << endl;
        UFluid.set
        (
            i,
            new volVectorField
```

```
            (
                IOobject
                (
                    "U",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                fluidRegions[i]
            )
        );
//Instead of createPhi.H because of pointers.
        Info<< "    Adding to phiFluid\n" << endl;
        phiFluid.set
        (
            i,
            new surfaceScalarField
            (
                IOobject
                (
                    "phi",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                linearInterpolate(U) & mesh.Sf()

            )
        );

   //This is still a blur , this comes from readTransportProperties.H
   //and is now included in the region
   //here I have no clue if this is ok for the transport.
        Info<< " singlePhaseTransportModel\n" << endl;
        singlePhaseTransport.set
        (
            i,
            singlePhaseTransportModel laminarTransport
            (
                UFluid[i],
                phiFluid[i],
            ).ptr()
        );
//Here based on existing createFluidFields, looks ok

// Thermal expansion coefficient [1/K]
        beta.set
        (
            i,
            new dimensionedScalar
            (
                fluidRegions[i].laminarTransport.lookup
                (
                    "beta"
                )
            )
        );
// Reference temperature [K]
        TRef.set
        (
            i,
            new dimensionedScalar
            (
                fluidRegions[i].laminarTransport.lookup
                (
                    "TRef"
                )
            )
        );

    // Laminar Prandtl number
        Pr.set
        (
            i,
            new dimensionedScalar
            (
                fluidRegions[i].laminarTransport.lookup
                (
                    "Pr"
                )
            )
        );

    // Turbulent Prandtl number
         Prt.set
        (
            i,
            new dimensionedScalar
            (
                fluidRegions[i].laminarTransport.lookup
                (
                    "Prt"
                )
            )
        );

        Info<< "    Adding to turbulence\n" << endl;
        turbulence.set
        (
            i,
            incompressible::RASModel::New
            (
                UFluid[i],
                phiFluid[i],
                laminarTransport
            ).ptr()
        );
//in original file, not in buoyantSimpleFoam so I left it in.
        Info<< "    Adding to gFluid\n" << endl;
        gFluid.set
        (
            i,
            new uniformDimensionedVectorField
            (
                IOobject
                (
                    "g",
                    runTime.constant(),
                    fluidRegions[i],
                    IOobject::MUST_READ,
                    IOobject::NO_WRITE
                )
            )
        );

        Info<< "    Adding to ghFluid\n" << endl;
        ghFluid.set
        (
            i,
            new volScalarField("gh", gFluid[i] & fluidRegions[i].C())
        );

        Info<< "    Adding to ghfFluid\n" << endl;
        ghfFluid.set
        (
            i,
            new surfaceScalarField("ghf", gFluid[i] & fluidRegions[i].Cf())
        );

        pFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                    "p",
                    runTime.timeName(),
                    fluidRegions[i],
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
                ),
                p_rghFluid[i] + rhokFluid[i]*ghFluid[i]
            )
        );


        radiation.set
        (
            i,
            radiation::radiationModel::New(TFluid[i])
        );



        setRefCell
        (
            pFluid[i],
            p_rghFluid[i],
            fluidRegions[i].solutionDict().subDict("SIMPLE"),
            pRefCellFluid[i],
            pRefValueFluid[i]
        );

         if (p_rghFluid[i].needReference())
            {
                pFluid[i] += dimensionedScalar
                (
                    "p",
                    pFluid[i].dimensions(),
                    pRefValueFluid[i] - getRefCellValue(pFluid[i], pRefCellFluid[i])
                );
            }

    }
```

```
const fvMesh& mesh = fluidRegions[i];

    volScalarField& T = TFluid[i];
    volScalarField& kappat = kappatFluid[i];
    volScalarField& rhok = rhokFluid[i];
    volVectorField& U = UFluid[i];
    surfaceScalarField& phi = phiFluid[i];

    incompressible::RASModel& turb = turbulence[i];

    volScalarField& p = pFluid[i];

    radiation::radiationModel& rad = radiation[i];

    const label pRefCell = pRefCellFluid[i];
    const scalar pRefValue = pRefValueFluid[i];

    volScalarField& p_rgh = p_rghFluid[i];
    const volScalarField& gh = ghFluid[i];
    const surfaceScalarField& ghf = ghfFluid[i];
```

Once the fluid part has been adapted, we can change in the main file the existing headers. The thermophysical Model `basicRhoThermo` is no longer used, the turbulenceModel can be specified as RASModel, and a TransportModel can be added.
In the `Make/options` the libraries used in the Boussinesq approach should be added. The flow is now incompressible so change the file into:

```
EXE_INC = \
    -I.. \
    -Ifluid \
    -Isolid \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basicSolidThermo/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/radiationModels/lnInclude \
    -I$(LIB_SRC)/turbulenceModels \
    -I$(LIB_SRC)/turbulenceModels/incompressible/turbulenceModel/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/incompressible/RAS/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)//turbulenceModels/compressible/turbulenceModel/\
    derivedFvPatchFields/turbulentTemperatureCoupledBaffle

EXE_LIBS = \
    -lfiniteVolume \
    -lbasicThermophysicalModels \
    -lbasicSolidThermo \
    -lspecie \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleTransportModels \
    -lradiationModels
```

When trying to compile the new solver, a compiling error will be given. This is due to the fact that the compiler cannot find the header file `regionProperties.H` as this is part of derived patches for compressible flows.
In the definitions, we must find a way to couple regions without using compressible approach.

3.3 Conclusions

It still is not clear to me how to adapt the chtMultiRegionSimpleFoam to a chtMultiRegionBoussinesqSimpleFoam as the solver is built (especially the interface fluid-solid) assuming that the fluid is a compressible flow. All used classes are therefore based on compressible flow.
I now found a thread on the CFD-forum on "chtIcoMultiRegionFoam - Incompressible version of chtMultiRegionFoam". To come to a result, the approach used in this solver can be studied and may provide some answers. Suggestions are more than welcome!