CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Conjugate heat transfer in OpenFOAM

Developed for OpenFOAM-4.0 and FOAM-extend-4.0
Requires: python 3.0, numpy, matplotlib

Author:
Turo VÄLIKANGAS
Tampere University of Technology

Peer reviewed by:
MINGHAO LI
HÅKAN NILSSON

February 14, 2017

Learning outcomes

The reader will learn:

- How the main points of steady-state conjugate heat transfer (chtMultiRegionSimpleFoam and conjugateHeatSimpleFoam) solvers are written in OpenFOAM 4.0 and FOAM-extend-4.0.
- How a case setup for these solvers should be done.
- How a coupled conjugate heat transfer solver conjugateHeatSimpleFoam differs from the cht solvers.
- How the results of conjugateHeatSimpleFoam differs from the cht solvers.
- How to speed up the chtMultiRegion solvers solid region convergence.
- How to implement a uniformExternalWallHeatFlux boundary condition.

Contents

Chapter 1

Introduction and Theory

In this report we first go through the equations that covers the heat transfer in both the solid and fluid and some theoretical background about them. Then we study an open source CFD program called OpenFOAM and what different solvers are available for conjugate heat transfer problems and how they differ from each other. Then we do a little modification to the solver to speed up the convergence. In the end we also implement a new boundary condition that combines existing externalWallHeatFlux and uniformFixedValue boundary types.

1.1 Conduction and convection heat transfer

If we consider heat transfer without any source terms or radiation, what is left are conduction and convection. Steady-state conduction of heat in solid is defined by the Fourier's Heat equation which is defined as

$$k\frac{\partial^2 T}{\partial x_i^2} = 0 \tag{1.1}$$

This means that the heat conductivity k of the material is the only variables that affects the temperature field in the solid when the boundary conditions are fixed. On the other hand the equation that governs the transport of the heat in the flow field is known as the energy equation, which is defined as

$$\frac{\partial}{\partial x_i}(\rho u_i T) = \frac{\partial}{\partial x_i}(\frac{k}{C_p}\frac{\partial u_j}{\partial x_j}) \tag{1.2}$$

where T is the temperature, k is the conductivity, ρ the density and C_p the specific heat of the fluid [1]. The u_i is of course the velocity in each Cartesian directions $i, jandk$.

When both the conduction in the solid and the convection in the fluid are computed together so that the temperature field obeys the laws of thermodynamics, it is called a conjugate heat transfer problem. [2]

1.2 Conjugate heat transfer projects

Conjugate Heat Transfer solvers have been implemented in OpenFOAM at least since 2009 but a lot of development is still left to be done. The author has experienced that one of the main problems of the OpenFOAM solvers is very slow convergence speed when compared to commercial conjugate heat transfer solvers, so a lot of development still needs to be done. Previous work on this course has been done on this subject by Maaike van der Tempel 2012 [3]

`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2012/MaaikeVanDerTempel/`
`chtMultiRegionSimpleBoussinesqFoam7.pdf`

and Johan Magnusson 2010 [4]

`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/`
`johanMagnusson/johanMagnussonReport.pdf`

Maaike did a study on the chtMultiRegionSimpleFoam and tried to transform it into an incompressible solver with a Boussinesq approximation for the buoyancy term but she was not able to do it in her project. Johan Magnusson did a study on the conjugateHeatFoam which was an icoFoam based solver and was therefore restricted to laminar, incompressible flow of Newtonian fluids. Johan introduced the solver and how two different cases are constructed with this solver. This already gives you an idea that the solvers for conjugate heat transfer have been under development even after they have been implemented and the development is still continuing.

1.3 The tutorial case setup

The tutorial case that is used to compare different solvers is illustrated in figure 1.1.

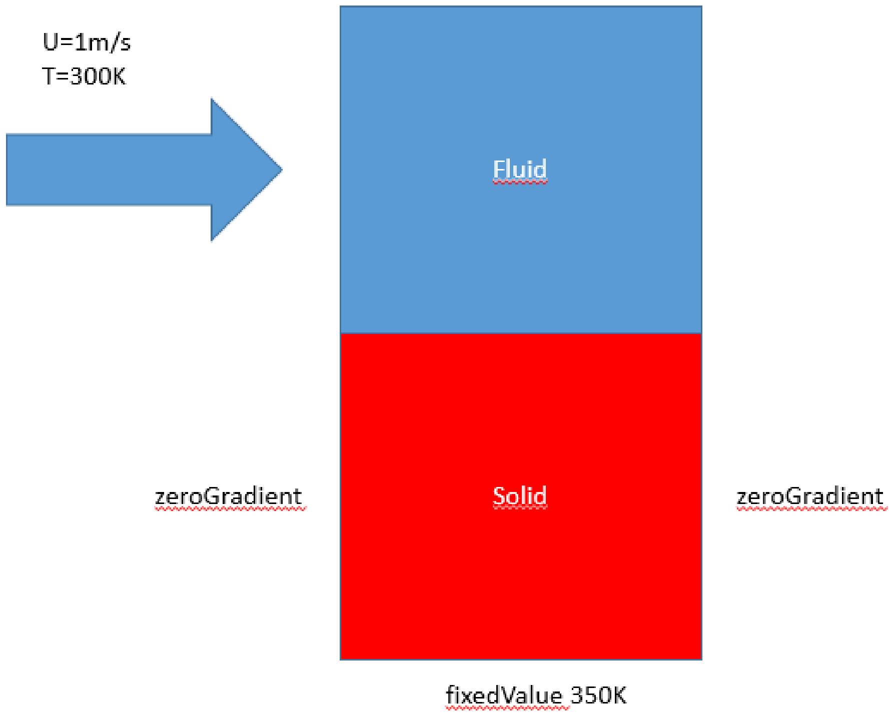

Figure 1.1: Illustration of the tutorial case

In the case, there is two blocks of computational mediums. The lower one is a solid block with a fixed temperature value set on the lower boundary. The upper part on the other hand is a fluid block that has a 350K flow from left to right with the speed of 1 meter per second. The upper boundary is set to as a periodic boundary condition, othervise all the other boundaries have a zero gradient value for all the fields.

The main difference in setting up the case for the chtMultiRegion in the OpenFOAM 4.0 foundation version is that you don't need to extract zones for the patches that connect the different regions. This saves you one work step compared to the workflow in FOAM-extend. Different regions are divided into their own directories and corresponding meshes for every region. The case setup with all the required files to run the cases are provided with this report in a separate zip file called **oneFluidOneSolid2D.tgz**.

Chapter 2

chtMultiRegionSimpleFoam in OpenFOAM 4.0

For the conjugate heat transfer problems in OpenFOAM, a conjugate heat transfer multi region simple foam solver is developed(chtMultiRegionSimpleFoam). It is a compressible solver that combines laplacian/heatConductionFoam and buoyantSimpleFoam for solving cases with multiple solid and fluid regions. It is a steady-state version of chtMultiRegionFoam. In this chapter we study the solver and examine the case structure of a tutorial case.

2.1 chtMultiRegionSimpleFoam.C

Open the solver's directory

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

The solver file **chtMultiRegionSimpleFoam.C** contains the following inclusion lines. Rest of the lines in the file are left out as irrelevant.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#include "fvCFD.H"
#include "rhoThermo.H"
#include "turbulentFluidThermoModel.H"
#include "fixedGradientFvPatchFields.H"
#include "regionProperties.H"
#include "solidThermo.H"
#include "radiationModel.H"
#include "fvOptions.H"
#include "coordinateSystem.H"

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

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

    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMeshes.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"
```

A list of the included .H files and their purposes is given below:
fvCFD.H - Standard file for finite volume method

```
$FOAM_SRC/finiteVolume/cfdTools/general/include/fvCFD.H
```

rhoThermo.H - Basic thermodynamic properties based on density

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

turbulentFluidThermoModel.H - Typedefs for turbulence, RAS and LES models for compressible flow based on the standard laminar transport package.

```
$FOAM_SRC//TurbulenceModels/compressible/turbulentFluidThermoModels/turbulentFluidThermoModel.H
```

fixedGradientFvPatchFields.H - Defines the fixed boundary condition

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

regionProperties.H - Simple class to hold region information for coupled region simulations.

```
$FOAM_SRC/regionModels/regionModel/regionProperties/regionProperties.H
```

solidThermo.H - Fundamental solid thermodynamic properties

```
$FOAM_SRC//thermophysicalModels/solidThermo/solidThermo/solidThermo.H
```

radiationModel.H - Namespace for radiation modelling

```
$FOAM_SRC/thermophysicalModels/radiation/radiationModels/radiationModel/radiationModel.H
```

fvOptions.H - Finite-volume options properties

```
$FOAM_SRC/finiteVolume/cfdTools/general/fvOptions/fvOptions.H
```

coordinateSystem.H - Base class for other coordinate system specifications.

```
$FOAM_SRC/meshTools/coordinateSystems/coordinateSystem.H
```

postProcess.H - List of function objects with start(), execute() and end() functions that is called for each object.

```
$FOAM_SRC/OpenFOAM/db/functionObjects/functionObjectList/postProcess.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
```

createMeshes.H - Defines a dummy default region mesh.

`$FOAM_SRC/OpenFOAM/include/createMeshes.H`

createFields.H - Create and populate all the required fields for all fluid and solid regions.

`$FOAM_APP/solvers/heatTransfer/chtMultiRegionFoam/createFields.H`

initContinuityErrs.H - Declare and initialise the cumulative continuity error.

`$FOAM_APP/solvers/heatTransfer/chtMultiRegionFoam/fluid/initContinuityErrs.H`

After all the .H files are included in chtMultiRegionSimpleFoam.C, the solver loop starts. In the beginning of every loop the time step is highlighted. After that, the solver first solves the fields for all the fluid regions and then proceeds to solve the required fields for all the subsequent solid regions. In the solver loops, the first line writes out what is the region that it is solving. Then in the **"setRegion(Fluid/Solid)Fields.H"** it populates all the regired fields, in the **"read(Fluid/Solid)MultiRegionSIMPLEControls.H"** file the non-orthogonal correction number is looked up from the SIMPLE settings in fvSolution. Then finally in **"solve(Fluid/Solid)"** the fields are solved based on the defined relations.

```
    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;
     }
```

2.1.1 solveFluid.H

In the **solveFluid.H**, the required fields are solved on the fluid side. In OpenFOAM 4.0 you have the option to only solve the energy equation by setting frozenFlow **true** in the fvSchemes file inside the Simple or Pimple settings. In the else loop the pressure and density are looked up from the previous iteration and then used to solve momentum equation UEqn, energy equation EEqn and pressure equation pEqn. After these the turbulence is calculated and turbulent viscosity is corrected.

```
//  Pressure-velocity SIMPLE corrector

 {
     if (frozenFlow)
     {
         #include "EEqn.H"
     }
     else
     {
         p_rgh.storePrevIter();
         rho.storePrevIter();

         #include "UEqn.H"
         #include "EEqn.H"
         #include "pEqn.H"

         turb.correct();
     }
 }
```

In the UEqn the momentum equation is calculated as shown below. The first line corrects the velocity field at the boundaries if the Moving Rotating Frame option is used.

```
// Solve the Momentum equation

MRF.correctBoundaryVelocity(U);

tmp<fvVectorMatrix> tUEqn
(
    fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turb.divDevRhoReff(U)
 ==
    fvOptions(rho, U)
);
fvVectorMatrix& UEqn = tUEqn.ref();

UEqn.relax();

fvOptions.constrain(UEqn);

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

fvOptions.correct(U);
```

The first part is to create the **tUEqn**, which is the pseudo transient momentum equation which is used with the MRF setup. The **fvm::div(phi, U)**, represents the convection with the mass phi, **MRF.DDt(rho, U)** updates the velocity according to Multiple Reference Frame setup. **turb.divDevRhoReff(U)** is used to update the turbulent viscosity. The following lines after that creates and solves the momentum equation in a steady-state manner. The energy equation is solved in the **EEqn.H**, and can be seen below. It can be seen from the laid out code that the solver uses enthalpy for the calculations.

```
{
    volScalarField& he = thermo.he();

    fvScalarMatrix EEqn
    (
        fvm::div(phi, he)
      + (
            he.name() == "e"
          ? fvc::div(phi, volScalarField("Ekp", 0.5*magSqr(U) + p/rho))
          : fvc::div(phi, volScalarField("K", 0.5*magSqr(U)))
        )
      - fvm::laplacian(turb.alphaEff(), he)
     ==
        rho*(U&g)
      + rad.Sh(thermo)
      + fvOptions(rho, he)
    );

    EEqn.relax();

    fvOptions.constrain(EEqn);

    EEqn.solve();

    fvOptions.correct(he);

    thermo.correct();
    rad.correct();

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

Then in the first part of the **pEqn.H**, density is corrected according to the MRF settings. Density is adjusted if the case is a closed volume, flux consistency over boundary condition is ensured and compressibility is calculated and checked if it should be considered.

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

    volScalarField rAU("rAU", 1.0/UEqn.A());
    surfaceScalarField rhorAUf("rhorAUf", fvc::interpolate(rho*rAU));
    volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p_rgh));
    tUEqn.clear();

    surfaceScalarField phig(-rhorAUf*ghf*fvc::snGrad(rho)*mesh.magSf());

    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        fvc::flux(rho*HbyA)
    );

    MRF.makeRelative(fvc::interpolate(rho), phiHbyA);

    bool closedVolume = adjustPhi(phiHbyA, U, p_rgh);

    phiHbyA += phig;

    // Update the pressure BCs to ensure flux consistency
    constrainPressure(p_rgh, rho, U, phiHbyA, rhorAUf, MRF);

    dimensionedScalar compressibility = fvc::domainIntegrate(psi);
    bool compressible = (compressibility.value() > SMALL);

     << max(rho).value() << endl;
}
```

The second part shown on the next page, solves the pressure field. First the **p_rghEqn** is created and then the reference value for the hydraulic pressure is calculated. This must be done because the solver calculates **p_rgh** and not the **p** field. This means that the solver calculates pressure without the hydrostatic pressure and is defined as

$$p\_rgh = p - rho * gh \tag{2.1}$$

where rho is the density, g is the free fall acceleration and h is the distance to the reference level. It calculates the fields repetitively if non-orthogonal correction loops are defined.

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

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

        p_rghEqn.solve();

        if (nonOrth == nNonOrthCorr)
        {
            // Calculate the conservative fluxes
            phi = phiHbyA - 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 = HbyA + rAU*fvc::reconstruct((phig - p_rghEqn.flux())/rhorAUf);
            U.correctBoundaryConditions();
            fvOptions.correct(U);
        }
    }

    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() << ' '
```

We can see that some parts of the solver is commented and is self explanatory. It can be seen that the pressure solver is based on compressibility ϕ and the density is calculated at the end of the solver.

2.1.2 solveSolid.H

In the solveSolid.H, only the laplacian equation for temperature in the solid is solved.

```
    {
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix hEqn
        (
            (
                thermo.isotropic()
              ? -fvm::laplacian(betav*thermo.alpha(), h, "laplacian(alpha,h)")
              : -fvm::laplacian(betav*taniAlpha(), h, "laplacian(alpha,h)")
            )
          ==
            fvOptions(rho, h)
        );

        hEqn.relax();

        fvOptions.constrain(hEqn);

        hEqn.solve();

        fvOptions.correct(h);
    }
}

thermo.correct();

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

When the case setup is done for this solver, also the pressure field must be initialised even though it is not used or solved at all. It is also notisable that the energy equation solves for enthalpy **h** using the thermal diffusivity $\alpha = \frac{k}{\rho C_p}$, where k and C_p are the conductivity and specific heat of the solid, respectively.

2.2 Boundary condition between the regions

The patch that is used to connect different regions in the OpenFOAM 4.0 is called **mappedWall**.

```
fluid_to_solid
{
    type            mappedWall;
    inGroups        1 ( wall );
    nFaces          100;
    startFace       16040;
    sampleMode      nearestPatchFace;
    sampleRegion    solid;
    samplePatch     solid_to_fluid;
}
```

The FOAM-extend has a boundary condition called **directMappedWall** which is identical to the one in the foundation version except it does not have the alternative copy constructor as shown below.

```
            //- Construct and return a clone, resetting the face list
        //  and boundary mesh
        virtual autoPtr<polyPatch> clone
        (
            const polyBoundaryMesh& bm,
            const label index,
            const labelUList& mapAddressing,
            const label newStart
        ) const
        {
            return autoPtr<polyPatch>
            (
                new mappedWallPolyPatch
                (
                    *this,
                    bm,
                    index,
                    mapAddressing,
                    newStart
                )
            );
        }
```

In the OpenFOAM Foundation version, thermophysical properties for the solid and the fluid are defined in the **thermophysicalProperties** file in the corresponding file in the constant folder. If the meshes are not conformal between the solid and the fluid region it is then suggested to use **nearestPatchFaceAMI** for the sampleMode options. This makes sure that the values are interpolated between the patches and no discontinuity problems occur.

2.3 Example of how to run the case with OF40

Now we go through the first example tutorial from the package that is provided with this report. Copy the tutorial oneFluidOneSolid2D.tgz to the run directory, unpack and go to the case.
tar xzf oneFluidOneSolid2D.tgz
cd oneFluidOneSolid2D/oneFluidOneSolid2DOF401Loop/

When typing **tree**, we can see the directory listing of the case directory.

tree

```
.
├── 0
│   ├── cellToRegion
│   ├── fluid
│   │   ├── p
│   │   ├── p_rgh
│   │   ├── T
│   │   └── U
│   ├── p
│   ├── p_rgh
│   ├── solid
│   │   ├── p
│   │   └── T
│   ├── T
│   └── U
├── Allclean
├── Allrun
├── constant
│   ├── fluid
│   │   ├── fvOptions
│   │   ├── g
│   │   ├── radiationProperties
│   │   ├── RASProperties
│   │   ├── thermophysicalProperties
│   │   └── turbulenceProperties
│   ├── regionProperties
│   └── solid
│       ├── fvOptions
│       └── thermophysicalProperties
└── system
    ├── controlDict
    ├── decomposeParDict
    ├── fluid
    │   ├── blockMeshDict
    │   ├── changeDictionaryDict
    │   ├── decomposeParDict
    │   ├── fvSchemes
    │   └── fvSolution
    ├── fvSchemes
    ├── fvSolution
    └── solid
        ├── blockMeshDict
        ├── changeDictionaryDict
        ├── decomposeParDict
        ├── fvSchemes
        └── fvSolution
```

We can see that both in constant and in the system folder the case has a set of files for two different regions **solid** and **fluid**. The 0 folder contains all the initial conditions for the simulation, the constant folder contains the mesh and other settings that stay constant during the simulation, and the system folder contains the schemes and other solver specific settings.

In the root of the case structure we also have **Allrun** and **Allclean** scripts. With Allclean the user can delete the meshes and all the data that was created. And by running the Allrun script, the mesh is created, the initial conditions are set and the solver is started. Type **./Allrun** >**& log &** to run the case and direct all the outputs to the **log** file.

Chapter 3

chtMultiRegionSimpleFoam in FOAM-extend-4.0

The chtMultiRegionSimpleFoam in FOAM-extend-4.0 has the same name as the equivalent in the OpenFOAM foundation, but when studied closer, it is different in both the implementation and the usage of the solver. FOAM-extend is another version of OpenFOAM that has departed from the original OpenFOAM. The idea is to provide community contributed extensions in the spirit of the OpenSource development for OpenFOAM. [5]

3.1 Solver break down

For the FOAM-extend 4.0 chtMultiRegion solver break down, we go through only the parts that differ from the OpenFOAM 4.0 foundation version.

Instead of rhoThermo.H, FOAM Extend uses:
basicPsiThermo.H - Basic thermodynamic properties based on compressibility

```
$FOAM_APP/thermophysicalModels/basic/psiThermo/basicPsiThermo/basicPsiThermo.H
```

Instead of turbulentFluidThermoModel.H, FOAM Extend includes:
turbulenceModel.H - Basic thermodynamic properties based on compressibility

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

compressibleCourantNo.H - Calculates and outputs the mean and maximum Courant Numbers.

```
$FOAM_SRC/finiteVolume/cfdTools/compressible/compressibleCourantNo.H
```

ChtMultiRegion in FOAM-extend also does not include the following files at this point: **solidThermo.H, radiationModel.H, fvOptions.H and coordinateSystem.H**.

In the solver loop everything else looks identical compared to OpenFOAM 4.0 but **initConvergenceCheck.H** is included before the solve(Fluid/Solid) part to initialize values for convergence parameters and then in **convergenceCheck.H** the convergence is checked.

3.1.1 solveFluid.H

As can be seen in **solveFluid.H** below in the FOAM-extend there is no option to solve the case as frozen flow at all. Otherwise it looks exactly like the one in foundation version. First the momentum equation is solved, then energy and in the end the pressure is computed.

```
//  Pressure-velocity SIMPLE corrector

    p.storePrevIter();
    rho.storePrevIter();
    {
        #include "UEqn.H"
        #include "hEqn.H"
        #include "pEqn.H"
    }

    turb.correct();
```

In UEqn the momentum equation is calculated. First the vector matrix system **UEqn** is created, then relaxed and finally solved. In the end the maximum residual is stored.

```
        // Solve the Momentum equation
    tmp<fvVectorMatrix> UEqn
    (
        fvm::div(phi, U)
      - fvm::Sp(fvc::div(phi), U)
      + turb.divDevRhoReff()
    );

    UEqn().relax();

    eqnResidual = solve
    (
        UEqn()
     ==
        fvc::reconstruct
        (
            fvc::interpolate(rho)*(g & mesh.Sf())
          - fvc::snGrad(p)*mesh.magSf()
        )
    ).initialResidual();

    maxResidual = max(eqnResidual, maxResidual);
```

Then in the **hEqn.H** the enthalpy field is computed. The structure follows the same guidelines as for the velocity field.

```
{
    fvScalarMatrix hEqn
    (
        fvm::div(phi, h)
      - fvm::Sp(fvc::div(phi), h)
      - fvm::laplacian(turb.alphaEff(), h)
     ==
        fvc::div(phi/fvc::interpolate(rho)*fvc::interpolate(p))
      - p*fvc::div(phi/fvc::interpolate(rho))
    );

    hEqn.relax();

    eqnResidual = hEqn.solve().initialResidual();
    maxResidual = max(eqnResidual, maxResidual);

    thermo.correct();

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

The first part of **pEqn.H** creates the density, velocity, mass fluxes and does some corrections for the fields if the case is a closeVolume case or has buoyancy.

```
{
    // From buoyantSimpleFoam

    rho = thermo.rho();

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

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

    phi = fvc::interpolate(rho)*(fvc::interpolate(U) & mesh.Sf());
    bool closedVolume = adjustPhi(phi, U, p);

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

```

The second part solves the pressure field. First the **pEqn** is created, then the reference value for the hydrostatic pressure is calculated and then the field is solved and the maximum residual is stored. Then if the non-orthogonal corrections is chosen the field is corrected accordingly.

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

        pEqn.setReference(pRefCell, pRefValue);

        // retain the residual from the first iteration
        if (nonOrth == 0)
        {
            eqnResidual = pEqn.solve().initialResidual();
            maxResidual = max(eqnResidual, maxResidual);
        }
        else
        {
            pEqn.solve();
        }

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

            // Calculate the conservative fluxes
            phi -= pEqn.flux();

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

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


    #include "continuityErrs.H"

    rho = thermo.rho();
    rho.relax();

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

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

3.1.2 solveSolid.H

As can be seen for the solid side the solver only solves for the laplacian equation for the temperature field.

```
{
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix tEqn
        (
            -fvm::laplacian(Kappa, T)
        );
        tEqn.relax();
        eqnResidual = tEqn.solve().initialResidual();
        maxResidual = max(eqnResidual, maxResidual);

    }

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

3.2 Boundary condition between the regions

For the extended version the boundary patch type used for connecting the solid and the fluid part is called the directMappedWall which for the air side looks like the example shown below. It can be found from the mesh boundary file $/polyMesh/boundary$.

```
    fluid_to_solid
    {
         type            directMappedWall;
    sampleMode      nearestPatchFace;
    sampleRegion    solid;
    samplePatch     solid_to_fluid;
    offset          ( 0 0 0 );
        nFaces          100;
        startFace       16040;

    }
```

For this boundary condition no interpolative sampleMode can not be chosen.

3.3 Example of how to run the case with fe40

Foam extend also requires one extra step in the workflow. The matching boundary patches need to be extracted as faceZones so that the solver can use them in the computation. This is why the **Allrun** script includes these following lines.

```
runApplication -l log.setSet.solid setSet -region solid -batch solid.setSet
runApplication -l log.setsToZones.solid setsToZones -region solid -noFlipMap

runApplication -l log.setSet.fluid setSet -region fluid -batch fluid.setSet
runApplication -l log.setsToZones.fluid setsToZones -region fluid -noFlipMap
```

The case can be ran and the case structure observed by typing the following to the terminal.

```
cd oneFluidOneSolid2DFE40/
./Allrun >& log &
tree
```

We can see that the thermophysical properties **cp** and **Kappa** for the solid must be defined in the 0/ folder as fields and not in the thermophysicalProperties file inside **constant**/.

```
.
├── 0
│   ├── cellToRegion
│   ├── fluid
│   │   ├── p
│   │   ├── T
│   │   └── U
│   ├── p
│   ├── solid
│   │   ├── cp
│   │   ├── Kappa
│   │   ├── rho
│   │   └── T
│   ├── T
│   └── U
├── Allclean
├── Allrun
├── constant
│   ├── fluid
│   │   ├── fvOptions
│   │   ├── g
│   │   ├── polyMesh
│   │   │   ├── boundary
│   │   │   ├── faces
│   │   │   ├── faceZones
│   │   │   ├── neighbour
│   │   │   ├── owner
│   │   │   ├── points
│   │   │   └── sets
│   │   │       └── fluid_to_solidZone
│   │   ├── radiationProperties
│   │   ├── RASProperties
│   │   ├── thermophysicalProperties
│   │   └── turbulenceProperties
│   ├── regionProperties
│   └── solid
│       ├── fvOptions
│       └── polyMesh
│           ├── boundary
│           ├── faces
│           ├── faceZones
│           ├── neighbour
│           ├── owner
│           ├── points
│           └── sets
│               └── solid_to_fluidZone
│
```

```
├── fluid.setSet
├── solid.setSet
└── system
    ├── controlDict
    ├── decomposeParDict
    ├── fluid
    │   ├── blockMeshDict
    │   ├── changeDictionaryDict
    │   ├── decomposeParDict
    │   ├── fvSchemes
    │   └── fvSolution
    ├── fvSchemes
    ├── fvSolution
    └── solid
        ├── blockMeshDict
        ├── changeDictionaryDict
        ├── decomposeParDict
        ├── fvSchemes
        └── fvSolution
```

Otherwise the case structure and all the files are the same as in the foundation version. We can see that the mesh folder polyMesh now has the sets for solid_to_fluidZone in the solid mesh and fluid_to_solidZone in the fluid mesh. Now the solver knows what are the patches that are coupling the regions together.

Chapter 4

conjugateHeatSimpleFoam

The advantage in the conjugateHeatSimpleFoam, is that the temperature equation in the solid part is embedded into the temperature equation of the fluid and solved together. This will increase the convergence speed of the problem since the boundary conditions on both regions will influence the full set of linear differential equation and the field starts to obey the conditions straight from the first iteration. In the following, we go through the solver and the tutorial case.

4.1 Study of the solver

The conjugateHeatSimpleFoam, is a steady-state solver for buoyancy-driven turbulent flow of incompressible Newtonian fluids with conjugate heat transfer, complex heat conduction and radiation. It can be found only in the FOAM-extend version of OpenFOAM. Because it is an incompressible solver it should only be used in flow situations with low velocity levels.

For the buoyancy, the solver uses the **Boussinesq approximation**, which means that the buoyancy can be calculated accurately only for flows with low temperature difference induced flows. The density is treated as constant in the momentum and pressure equation and is a variable only in the gravitational term. This means that density is then assumed to vary linearly with the temperature and it can be computed as

$$\Delta\rho = \rho_0 \beta \Delta T \tag{4.1}$$

where ρ_0 is the reference density, β is the coefficient of thermal expansion and ΔT is the temperature difference. [6]

Next we go through the important included **.H** files that are introduced in the **conjugateHeatSimpleFoam.C**.

coupledFvMatrices.H - Coupled Finite-Volume matrix is a collection of FV matrices solver as a block system.

```
$FOAM_SRC/coupledMatrix/coupledFvMatrices/coupledFvMatrices.H
```

regionCouplePolyPatch.H - Region couple patch. Used for multi-region conjugate simulations.

```
$FOAM_SRC/foam/meshes/polyMesh/polyPatches/constraint/regionCouple/regionCouplePolyPatch.H
```

thermalModel.H - Thermal material properties for solids.

```
$FOAM_SRC/solidModels/thermalModel/thermalModel.H
```

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/singlePhaseTransportModel/singlePhaseTransportModel.H`

RASModel.H - Abstract base class for incompressible turbulence models.

`$FOAM_SRC/turbulenceModels/incompressible/RAS/RASModel/RASModel.H`

simpleControl.H - SIMPLE control class to supply convergence information/checks for the SIMPLE loop.

`$FOAM_SRC/finiteVolume/cfdTools/general/solutionControl/simpleControl/simpleControl.H`

readGravitationalAcceleration.H - Read the gravitational acceleration from the **g**-file

`$FOAM_SRC/finiteVolume/cfdTools/general/include/readGravitationalAcceleration.H`

The first part of the solver includes all the required **.H** files.

```
#include "fvCFD.H"
#include "coupledFvMatrices.H"
#include "regionCouplePolyPatch.H"
#include "radiationModel.H"
#include "thermalModel.H"
#include "singlePhaseTransportModel.H"
#include "RASModel.H"
#include "simpleControl.H"

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

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

    simpleControl simple(mesh);

#   include "readGravitationalAcceleration.H"
#   include "createFields.H"
#   include "createSolidFields.H"
#   include "initContinuityErrs.H"

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

And then in the second part of the solver, the solver loop itself is laid out. We can see that in the solver, first the regions and patches are detached from each other. Then the pressure for previous iteration step is stored for convergence monitoring. Then in the **UEqn.H** and **pEqn.H** the momentum and the pressure equations are solved for the fluid field. Following the turbulence and radiation corrections. Then the conductivities for the fluid and solid regions are updated according to the temperature. Afterwords the regions are attached and then the conductivity in the boundary is corrected and finally the conductivity in the solid region is used at the interface and interpolated on the fluid part. Finally before solving the energy equation a thermal resistance is enforced between the regions if one is used.

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

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

        // Detach patches
#       include "detachPatches.H"

        p_rgh.storePrevIter();

#       include "UEqn.H"
#       include "pEqn.H"

        // Update turbulent quantities
        turbulence->correct();

        radiation->correct();

        // Update thermal conductivity in the fluid
        kappaEff = rho*Cp*(turbulence->nu()/Pr + turbulence->nut()/Prt);

        // Update thermal conductivity in the solid
        solidThermo.correct();
        kSolid = solidThermo.k();

        // Coupled patches
#       include "attachPatches.H"

        kappaEff.correctBoundaryConditions();
        kSolid.correctBoundaryConditions();

        // Interpolate to the faces and add thermal resistance
        surfaceScalarField kSolidf = fvc::interpolate(kSolid);
        solidThermo.modifyResistance(kSolidf);

#       include "solveEnergy.H"

        // Update density according to Boussinesq approximation
        rhok = 1.0 - beta*(T - TRef);
        rhok.correctBoundaryConditions();

        runTime.write();

        Info<< "ExecutionTime = "
            << runTime.elapsedCpuTime()
            << " s\n\n" << endl;
    }

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

    return(0);
}
```

4.1.1 solveEnergy.H

The solving of the energy equation in the **conjugateHeatSimpleFoam** is the most interesting part of the solver and this is why next we go through the **solveEnergy.H** part of the solver loop. First we can see that the simple solver loop controls are created for the solid part. In the while loop, first a **coupledFvScalarMatrix TEqns** is created for two equations. Then the fluid part of the energy equation is established and relaxed. Then the same is done for the solid part and afterwords they are embedded together in the **TEqns** and the matrix system is solved.

```
{
    // Solid side
    simpleControl simpleSolid(solidMesh);

    while (simpleSolid.correctNonOrthogonal())
    {
        coupledFvScalarMatrix TEqns(2);

        fvScalarMatrix* TFluidEqn = new fvScalarMatrix
        (
            rho*Cp*
            (
                fvm::div(phi, T)
              + fvm::SuSp(-fvc::div(phi), T)
            )
          - fvm::laplacian(kappaEff, T)
            ==
            radiation->Ru()
          - fvm::Sp(4.0*radiation->Rp()*pow3(T), T)
          + 3.0*radiation->Rp()*pow4(T)
        );

        TFluidEqn->relax();

        fvScalarMatrix* TSolidEqn = new fvScalarMatrix
        (
          - fvm::laplacian(kSolidf, Tsolid, "laplacian(k,T)")
          + fvm::SuSp(-solidThermo.S()/Tsolid, Tsolid)
        );

        TSolidEqn->relax();

        // Add fluid equation
        TEqns.set(0, TFluidEqn);

        // Add solid equation
        TEqns.set(1, TSolidEqn);

        TEqns.solve();
    }
}
```

4.2 Boundary condition between the regions

The coupling between the solid and the fluid region is of course very important. This is why a special boundary condition is needed for this type of solver. If we look at the boundary condition for the patch between the regions. For the coupled solver we use regionCouple. It can be found from the mesh boundary file */polyMesh/boundary*. This boundary condition is based on the general grid interface (ggi) object which means that it automatically interpolates between the coupling patches.

```
    fluid_to_solid
{
    type            regionCouple;
    nFaces          100;
    startFace       19900;
    shadowRegion    solid;
    shadowPatch     solid_to_fluid;
    zone            fluid_to_solidZone;
    attached        off;
    master          on;
    isWall          on;
    bridgeOverlap   false;
}
```

In the first option the user must define if the patches that are connecting the regions are already attached to each other(in the same mesh), or not. In the solver loop the solver always attaches the patches before solving the energy field and then detaches them for solving the fluid region. If the coupling region is a wall then the user should specify it in the next option. In the last option in the regionCouple boundary condition the user must select the **bridgeOverLap** option **true** if the patches are overlapping. This means that some part of the patch does not have a matching face in the shadow patch, or vice versa. The solver will then change the boundary condition to slip in these faces. The author observed that if the bridgeOverlap option is set to true when the case is then decomposed for a parallel simulation the **decomposePar** program does not retain the option but changes it to **false**. To fix this it is essential to run the following sed script to change all falses to true.

```
find -name "boundary" -exec sed -i 's/false/true/g' {} +
```

4.3 Example of how to run the case with fe40

To run the case with **conjugateHeatSimpleFoam** the user can copy paste the following lines to the terminal

```
cd ..
cd oneFluidOneSolid2DFE40Coupled/
./Allrun >& log &
tree
```

We can then observe the case structure.

```
.
├── 0
│   ├── kappaEff
│   ├── p
│   ├── p_rgh
│   ├── solid
│   │   ├── k
│   │   └── T
│   ├── T
│   └── U
├── Allclean
├── Allrun
├── constant
│   ├── g
│   ├── polyMesh
│   │   └── boundaryRegionAddressing
│   ├── radiationProperties
│   ├── RASProperties
│   ├── solid
│   │   ├── fvOptions
│   │   ├── polyMesh
│   │   └── thermalProperties
│   ├── transportProperties
│   └── turbulenceProperties
├── fluid.setSet
├── solid.setSet
└── system
    ├── blockMeshDict
    ├── changeDictionaryDict
    ├── controlDict
    ├── decomposeParDict
    ├── fvSchemes
    ├── fvSolution
    └── solid
        ├── blockMeshDict
        ├── changeDictionaryDict
        ├── decomposeParDict
        ├── fvSchemes
        └── fvSolution
```

It can be noticed that the conductivity is defined in the 0/ folder as well as in the thermalProperties folder. The one in the 0/ folder is used for the coupling of the patches to correct the conductivity in the boundary between the fluid and the solid. But the conductivity in the constant/ folder is used for the calculation of the temperature field.

Chapter 5

chtCustomMultiRegionSimpleFoam

5.1 How to edit the solver

In this chapter we do a quick tutorial of how to change the chtMultiRegionSimpleFoam solver so that the user can choose how many times the fluid and the solid region is solved in each iteration step. This is done to speed up the convergence of the solver.

First we create the essential files for our new solver and change the names accordingly. The copy pasted lines below can be found from the file copyPaste that was provided with the case files.

```
   cd $WM_PROJECT_USER_DIR
 mkdir -p applications/solvers/heatTransfer/
 cd applications/solvers/heatTransfer/
 cp -rf $FOAM_SOLVERS/heatTransfer/chtMultiRegionFoam/ ./
 mv chtMultiRegionFoam/ chtCustomMultiRegionFoam
 cd chtCustomMultiRegionFoam
 mv chtMultiRegionSimpleFoam/ chtCustomMultiRegionSimpleFoam
 cd chtCustomMultiRegionSimpleFoam
 mv chtMultiRegionSimpleFoam.C chtCustomMultiRegionSimpleFoam.C
 sed -i s/chtMultiRegionSimpleFoam/chtCustomMultiRegionSimpleFoam/g //
 chtCustomMultiRegionSimpleFoam.C
 sed -i s/chtMultiRegionSimpleFoam/chtCustomMultiRegionSimpleFoam/g Make/files
 sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
```

Then we want to create two new variables **numberOfFluidLoops** and **numberOfSolidLoops**. And then these are used as a counter and create a for loop around the actual solver loops. Copy and paster the following to the terminal.

```
sed -i '57i int numberOfFluidLoops=runTime.controlDict().lookupOrDefault<int>("numberOfFluidLoops", 1); \n' chtCustomMultiRegionSimpleFoam.C
sed -i '58i int numberOfSolidLoops =runTime.controlDict().lookupOrDefault<int>("numberOfSolidLoops", 1);\n' chtCustomMultiRegionSimpleFoam.C
sed -i '59i Info<< "The number of Fluid region loops per iteration step "<< numberOfFluidLoops<< endl;\n' chtCustomMultiRegionSimpleFoam.C
sed -i '60i Info<< "The number of Solid region loops per iteration step "<< numberOfSolidLoops << endl;\n' chtCustomMultiRegionSimpleFoam.C
sed -i '69i for(int iFluid =1 ; iFluid <= numberOfFluidLoops; iFluid++){' chtCustomMultiRegionSimpleFoam.C
sed -i '79i }\nfor(int iSolid =1 ; iSolid <= numberOfSolidLoops; iSolid++){ ' chtCustomMultiRegionSimpleFoam.C
sed -i '90i }' chtCustomMultiRegionSimpleFoam.C
```

After this the user needs to compile the solver, source the bashrc again and then use the solver for the two tutorial cases. The first one will do 2 loops for the solid and the second one will do 5 loops for the solid for every iteration step.

```
wmake
. $WM_PROJECT_DIR/etc/bashrc
run
cd oneFluidOneSolid2D/oneFluidOneSolid2DOF402Loop/
./Allrun >& log&
```

```
cd ..
cd oneFluidOneSolid2DOF405Loops/
./Allrun >& log&
cd ..
```

Then we wait for the solvers to finish and then we can compare the results

Chapter 6

Comparison of the results

In this chapter we compare and visualise the results from the provided tutorial case that is computed with all the different solvers.

6.1 Temperature profiles

A simple conjugate steady-state simulation case is used to compare the convergence speed of different solver options. The tutorial case and the results for the unaltered chtMultiRegion in OpenFOAM foundation 4.0 is illustrated in figure 6.1. The case has a solid body on the bottom part with a fixed value of 350K boundary condition on the lower side of the solid and over the solid we have a flow of air from left to right with the initial temperature of 300 Kelvins. For the boundary condition on top of the air side a symmetrical boundary condition is applied.

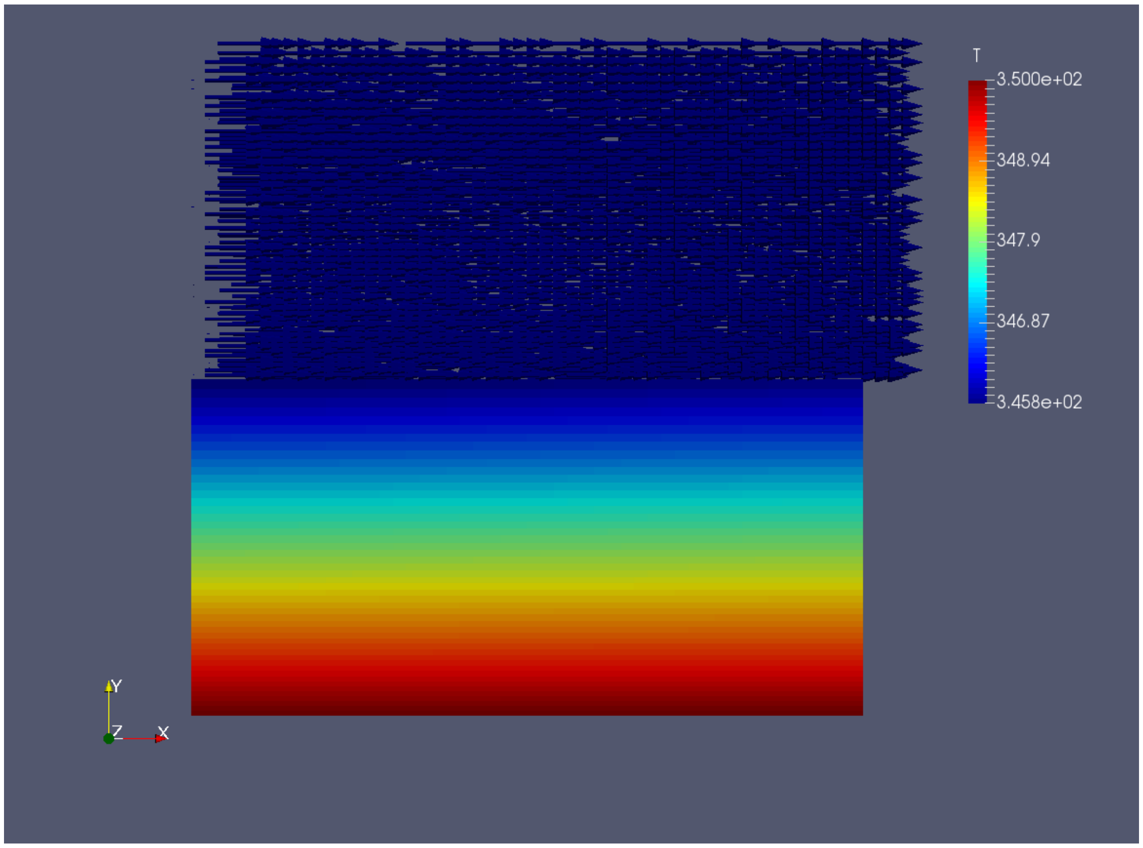

Figure 6.1: Illustration of the temperature field when calculated with the chtMultiRegion in the OpenFOAM Foundation 4.0 (case oneFluidOneSolid2DOF401Loop).

We see that in the OpenFOAM foundation version the temperature profile is almost constant in the x-direction. But when the same case is calculated with the FOAM-extend, we can see that this is not the case, as is illustrated in figure 6.2.

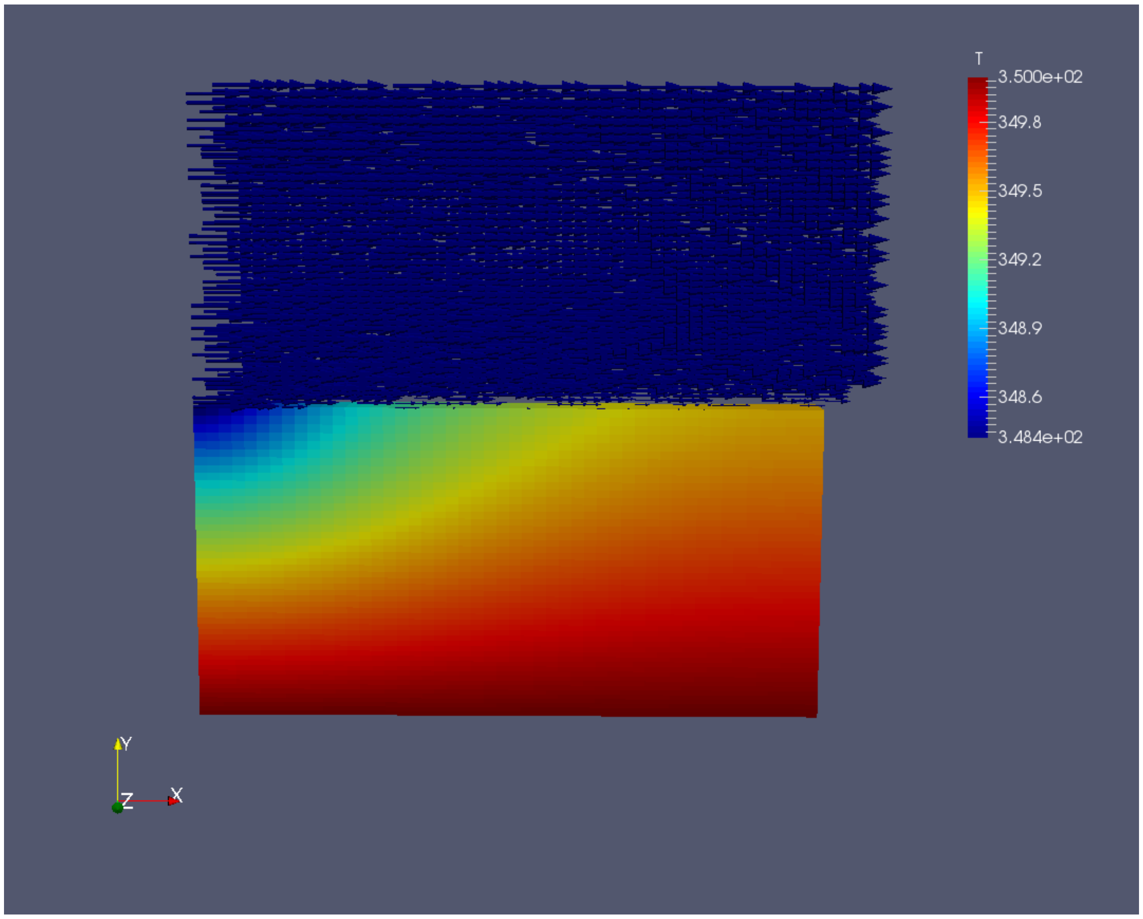

Figure 6.2: Illustration of the temperature field when calculated with the chtMultiRegion in FOAM-extend-4.0 (case oneFluidOneSolid2DFE40)

If we then study the results of the last solver conjugateHeatSimpleFoam, in the figure 6.3, we can see that the temperature fields for the FOAM-extend solvers looks the same but the minimum temperature of the solid is different. And as can be seen the temperature profile is not constant in the x-direction.

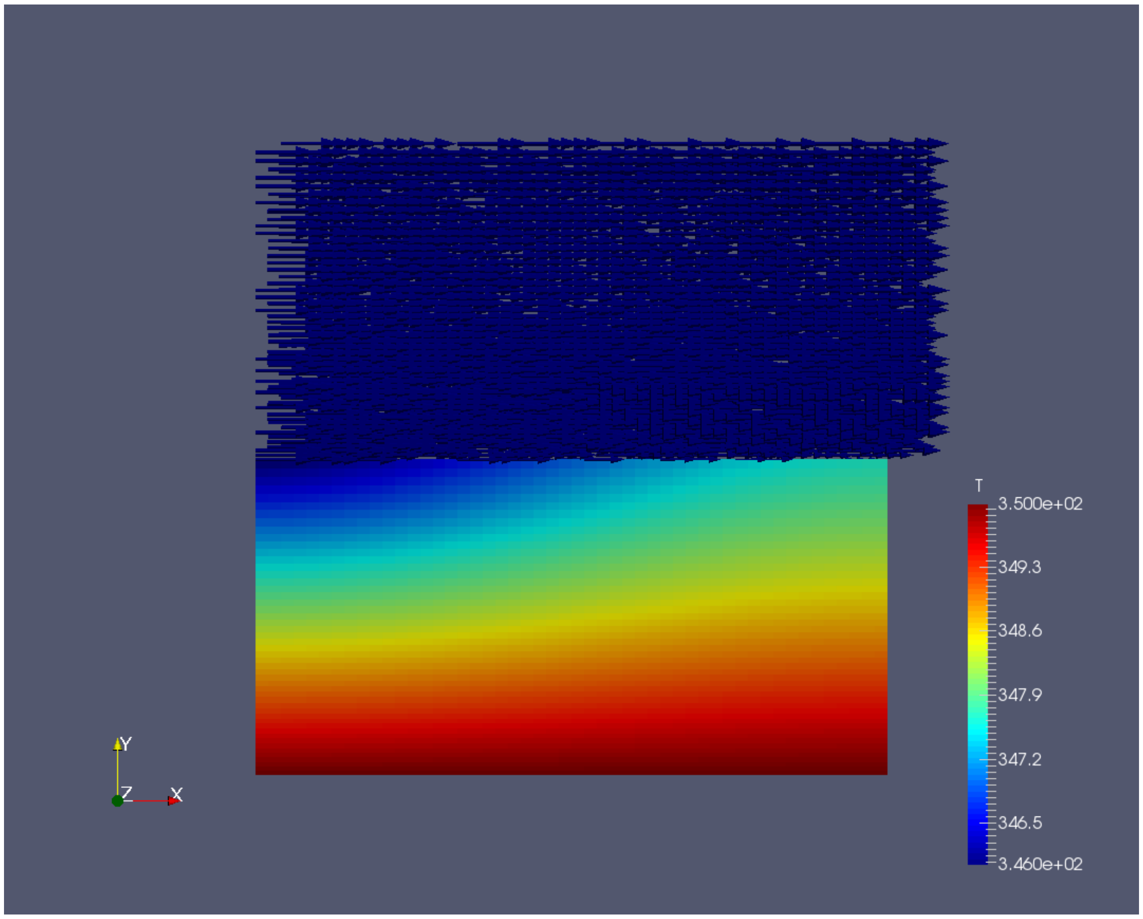

Figure 6.3: Illustration of the temperature field when calculated with the conjugateHeatSimpleFoam in FOAM-extend-4.0 (case oneFluidOneSolid2DFE40Coupled).

The author found no explanation why the temperature fields are so different between the FOAM-extend and OpenFOAM foundation version.

6.2 A convergence monitor

If we set the initial temperature of the solid part to 350 kelvins and then start the simulation, we can monitor the convergence of the conjugate simulation by plotting the minimum temperature of the solid part against the actual executional time that the computer took as shown in figure 6.4. The plots are made with the *minTempPlot_forOneFluid.py* file that is provided with the tutorial files and it can be executed with spyder for example. The libraries that are required are numpy, matplotlib and the python needs to be 3.0 or newer.

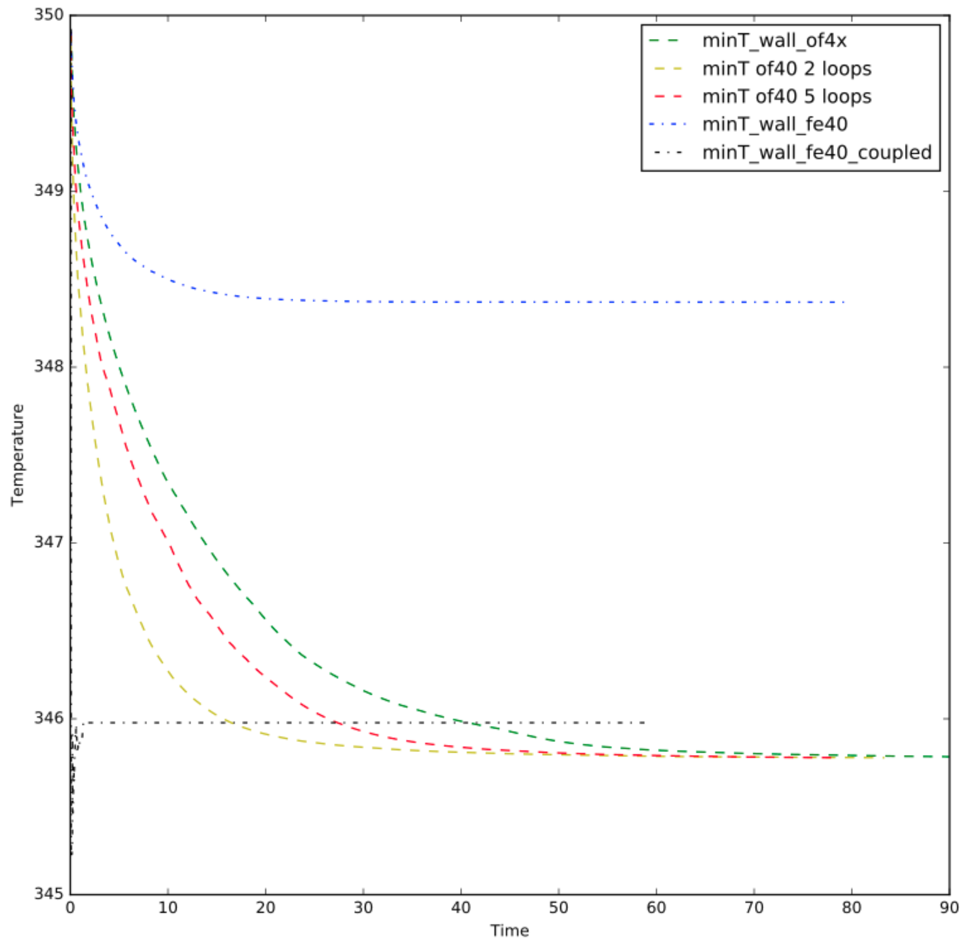

Figure 6.4: Minimum temperature in the solid part plotted against the execution time for all the solvers

The minimum temperature in the solid is plotted against the actual execution time that the simulation took. This means that we can have a realistic comparison of which solver converges the fastest. We can see that the **conjugateHeatSimpleFoam** is superior compared to all the other solvers in convergence speed as was expected. We can also observe that the FOAM-extend version of the **chtMultiRegionSimpleFoam** gives results that deviate quite much from all the other solvers. It is still the second fastest to converge, around 3-4 times faster than the OpenFOAM Foundation version.

If we then look at the results from **chtCustomMultiRegionSimpleFoam** we can see that the case version with 1 fluid loop and 2 solid loops per an iteration step converges fastest when compared to the one with 5 solid loops and the original **chtMultiRegionSimpleFoam** with no modification to the solver.

6.3 Tips for conjugateHeatTransfer problems in OpenFOAM

In this section we give some tips for running conjugate heat transfer cases with OpenFOAM.

limitTemperature in fvOptions (for example in *constant/solid/fvOptions*) is a good way to restrict the values of a specific field. This prevents the simulation from diverging and therefore crashing. An example can be seen below.

```
limitT
{
    type            limitTemperature;
    region fluid;
    active          yes;
    limitTemperatureCoeffs
    {
        selectionMode   all;
        Tmin             200;
        Tmax             400;
    }
}
```

Another tip is to always use **changeDictionary** to set up boundary conditions and initial values for conjugate heat transfer cases in OpenFOAM. It is a tool that requires a changeDictionaryDict in the system/ folder. In the file you can specify for the polyMesh/boundary file and the 0/ initial conditions for all different regions.

Chapter 7

uniformExternalWallHeatFlux

In this chapter we show how to edit the externalWallHeatFlux boundary condition in OpenFOAM foundation 4.0. This boundary condition is created for external boundary to specify the wall heat flux or the heat transfer coefficient on the external boundary patch. If the user chooses to use the heat transfer coefficient then the ambient external temperature needs to be defined. The basic boundary condition only supports a constant scalar value to be used in this purpose. Sometimes the user could want to use a time dependent ambient temperature for example in a standardised fire simulation where the temperature of the flame increases as the fire extends.

Lets first define all the required files and copy the externalWallHeatFluxTemperature boundary condition for a foundation.

```
cd $WM_PROJECT_USER_DIR
mkdir -p myBC/
cd myBC
cp -rf $FOAM_SRC/TurbulenceModels/compressible/turbulentFluidThermoModels/ //
derivedFvPatchFields/externalWallHeatFluxTemperature ./
mv externalWallHeatFluxTemperature ./uniformExternalWallHeatFluxTemperature
cd uniformExternalWallHeatFluxTemperature
mv externalWallHeatFluxTemperatureFvPatchScalarField.C //
uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C
mv externalWallHeatFluxTemperatureFvPatchScalarField.H //
uniformExternalWallHeatFluxTemperatureFvPatchScalarField.H
sed -i s/externalWallHeatFluxTemperatureFvPatchScalarField/uniformExternalWallHeatFluxTemperatureFvPatchScalarField/g //
uniformExternalWallHeatFluxTemperatureFvPatchScalarField.H
sed -i s/externalWallHeatFluxTemperatureFvPatchScalarField/uniformExternalWallHeatFluxTemperatureFvPatchScalarField/g //
uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C

sed -i s/externalWallHeatFluxTemperature/uniformExternalWallHeatFluxTemperature/g //
uniformExternalWallHeatFluxTemperatureFvPatchScalarField.H
```

Then we want to include the required libraries **options** and **files** in the Make folder. Copy and paster all the following to the terminal.

```
mkdir Make
touch ./Make/files
touch ./Make/options
echo "" >> Make/files
sed -i '$a uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C' Make/files
sed -i '$a LIB = $(FOAM_USER_LIBBIN)/libCustomBC' Make/files

echo "" >> Make/options
sed -i '$a EXE_INC = \\' Make/options
sed -i '$a -I../turbulenceModels/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/transportModels/compressible/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/thermophysicalModels/solidThermo/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/thermophysicalModels/solidSpecie/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/finiteVolume/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/meshTools/lnInclude \\' Make/options
sed -i '$a -I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude' Make/options
echo "" >> Make/options

sed -i '$a LIB_LIBS = \\' Make/options
sed -i '$a -lcompressibleTransportModels \\' Make/options
sed -i '$a -lfluidThermophysicalModels \\' Make/options
sed -i '$a -lsolidThermo \\' Make/options
sed -i '$a -lsolidSpecie \\' Make/options
sed -i '$a -lturbulenceModels \\' Make/options
sed -i '$a -lspecie \\' Make/options
sed -i '$a -lfiniteVolume \\' Make/options
sed -i '$a -lmeshTools' Make/options
```

Then copy paste the following lines to the **uniformExternalWallHeatFluxTemperatureFv-PatchScalarField.H** file, which creates the new variable that contains the time dependent temperature values. First we include the **Function1** class that is a more advanced version of the old DataEntry class. Then we create a pointer to the **Function1** class and define that it can only take scalar inputs. Then the next part is for all the different constructors so that also our new variable **uniformAmbientTemperature** gets constructed and then finally to write out the data in the write member function of the created class.

```
sed -i '97i #include "Function1.H" \n' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.H
sed -i '130i autoPtr<Function1<scalar> > uniformAmbientTemperature_; \n' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.H
sed -i '77i uniformAmbientTemperature_(),' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C
sed -i '104i uniformAmbientTemperature_(ptf.uniformAmbientTemperature_),' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C
sed -i '128i uniformAmbientTemperature_(),' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C
sed -i '198i uniformAmbientTemperature_(tppsf.uniformAmbientTemperature_),' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C
sed -i '363i uniformAmbientTemperature_->writeData(os);' uniformExternalWallHeatFluxTemperatureFvPatchScalarField.C
```

Then we have to modify the part of the code where the ambient temperature is used. But now in this version we want to make it time dependent so that the value will be fetched from uniformAmbientTemperature object based on the time t. Then copy paster the following to the terminal.

```
sed -i '301i Ta_ = uniformAmbientTemperature_->value(t);' uniformExternalWallHeatFluxTemperatureFvPatchScalarFiel
sed -i '301i const scalar t = this->db().time().timeOutputValue();' uniformExternalWallHeatFluxTemperatureFvPatchS
```

After this you can see that the part where the heat transfer coefficient is calculated should now look like the following.

```
void Foam::uniformExternalWallHeatFluxTemperatureFvPatchScalarField::updateCoeffs()
{
            hp = 1.0/(1.0/h_ + totalSolidRes);

            Qr /= Tp;
            refGrad() = 0.0;
          //Modified part starts
        const scalar t = this->db().time().timeOutputValue();
            Ta_ = uniformAmbientTemperature_->value(t);
        //Modified ends
            refValue() = hp*Ta_/(hp - Qr);
            valueFraction() =
                (hp - Qr)/((hp - Qr) + kappa(Tp)*patch().deltaCoeffs());
```

Now the last thing we need to edit is the write-member function

```
void Foam::uniformExternalWallHeatFluxTemperatureFvPatchScalarField::write
case fixedHeatTransferCoeff:
        {
            h_.writeEntry("h", os);
            Ta_.writeEntry("Ta", os);
        uniformAmbientTemperature_->writeData(os);
            thicknessLayers_.writeEntry("thicknessLayers", os);
            kappaLayers_.writeEntry("kappaLayers", os);
            break;
        }
```

Then just write **wmake libso** in the directory where the Make file is to compile the boundary condition. Then for example the boundary condition can be used as illustrated below.

```
wallCold
    {
   type uniformExternalWallHeatFluxTemperature;
kappaMethod solidThermo;
kappaName default;
value uniform 293.15; // initial temp.
    Ta              uniform 500;     // ambient temperature /[K]
    h               uniform 25.0;      // heat transfer coeff /[W/K/m2]
    thicknessLayers (0); // thickness of layer [m]
    kappaLayers     (0);          // thermal conductivity of // layer [W/m/K]
    value           uniform 500;     // initial temperature / [K]
uniformAmbientTemperature table
((0 243)
(20 437.82)
(40 519.71)
(60 572.36));
    }
```

The ambientTemperature is defined with a table in this example. When the time value is between the defined discrete values in the table the boundary condition knows how to interpolate the right value. The ambient temperature can be defined with any of the options that are constructed in Function1 class

`$FOAM_SRC/OpenFOAM/primitives/functions/Function1`

Chapter 8

Study questions

- What is a conjugate heat transfer problem?
- Which conjugate heat transfer solver studied in this report has an option to solve the case as a frozen flow field(momentum and pressure are not updated)?
- What type of sampleMode for the boundary type mappedWall in OpenFOAM Foundation should be used if the mesh regions are not conformal between the regions?
- Does the chtMultiRegionSimpleFoam in OpenFOAM foundation solve for the temperature or the enthalpy in the energy equation?
- Does the chtMultiRegionSimpleFoam in FOAM-extend solve for the temperature or the enthalpy in the energy equation?
- Does the conjugateHeatSimpleFoam in FOAM-extend solve for the temperature or the enthalpy in the energy equation?
- What does the conjugateHeatSimpleFoam do with the energy equations of the solid and the fluid and how does it effect the computation?
- What is the object type in OpenFOAM that is used to store the information about the time dependent ambient temperature?

Bibliography

[1] Venkanna B,K, (2010) Fundamentals of heat and mass transfer

[2] Frank M, White (2003) Fluid Mechanics

[3] Maaike van der Tempel (2012) A chtMultiRegionSimpleFoam tutorial

[4] Johan Magnusson (2010) conjugateHeatFoam with explanational tutorial together with a buoyancy driven flow tutorial and a convective conductive tutorial

[5] FOAM-extend, (2017) http://www.extend-project.de/the-extend-project

[6] H.K. Versteeg and W. Malalasekera (2007) An introduction to Computational Fluid Dynamics