Cite as: Venkatesh, B V.: Tutorial of convective heat transfer in a vertical slot. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H.,
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Tutorial of convective heat transfer in a vertical slot

Developed for OpenFOAM-4.x

Author:
Varun Venkatesh
vvarun@student.chalmers.se

Peer reviewed by:
Elias Siggeirsson
Håkan Nilsson

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

January 23, 2017

Contents

Learning outcomes

The main requirements of a tutorial is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- how to use the solvers in OpenFOAM to simulate convective heat transfer in a vertical slot.
- how to use viscosity models in OpenFOAM.
- how to use wall heat flux utility to calculate total boundary heat flux.

The theory of it:

- the theory of buoyantBoussinsqSimpleFoam and buoyantSimpleFoam solvers.
- how viscosity models, transport models and turbulent models are linked together in OpenFOAM.

How it is implemented:

- how to set up a case using the buoyantBoussinesqSimpleFoam and buoyantSimpleFoam solvers.
- how to implement the viscosity model into the case set up.

How to modify it:

- how to modify existing viscosity model to implement a new temperature based viscosity model.

Chapter 1

Introduction

1.1 Convective heat transfer in a vertical slot

Natural convection in a vertical enclosure with two walls at different temperatures involves complex interactions between the fluid and the walls which leads to formation of different flow patterns. Fig. 1.1 shows the 2D representation of the vertical slot. Let's consider two dimensional natural convection in a vertical slot with fixed temperatures at two vertical boundaries. In the slot, fluid rises along the hot wall, turns at the top, sinks down along the cold wall and turns again. This forms a uni-cellular motion of the fluid in the slot (represented by arrows in Fig. 1.1). The main parameters that define the flow are the Rayleigh number(Ra), the Prandtl number(Pr) and the aspect ratio (height/width) [1]. This tutorial involves 2D numerical analysis of natural convection in a vertical slot using OpenFOAM. The approach of this tutorial is not result driven, rather it focuses on the solvers in OpenFOAM and utilization of the libraries to best suit the case.

In the current tutorial, simulations will be carried out for a vertical slot of dimensions 300mm high (H) and 15mm wide (W) which leads to an aspect ratio of 20, See Fig. 1.1. The temperature of the hot wall is considered as 323K (T_1) and that of cold wall is considered as 273K (T_2) i.e. A temperature difference of 50K is maintained between the two vertical walls. The fluid in the enclosure is Silicone oil. The Prandtl number of Silicone oil is considered as 50.

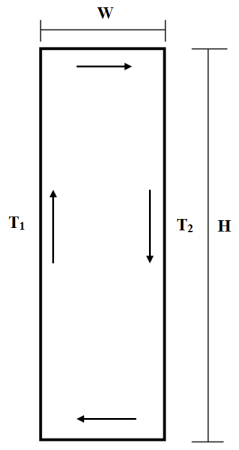

Figure 1.1: 2D representation of the vertical slot

1.2 Solvers in OpenFOAM for heat transfer

The solvers available in OpenFOAM for heat transfer problems are

- **buoyantBoussinesqSimpleFoam**: Steady state solver for buoyant and turbulent flow of incompressible fluids.
- **buoyantBoussinesqPimpleFoam**: Transient solver for buoyant and turbulent flow of incompressible fluids.
- **buoyantSimpleFoam**: Steady state solver for buoyant and turbulent flow of compressible fluids.
- **buoyantPimpleFoam**: Transient solver for buoyant and turbulent flow of incompressible fluids.
- **chtMultiRegionSimpleFoam**: Steady state solver for buoyant and turbulent flow and conjugate heat transfer between solids and fluids.
- **chtMultiRegionFoam**: Transient solver for buoyant and turbulent flow and conjugate heat transfer between solids and fluids.

Since the present case includes buoyant and turbulent flow, buoyantBoussinesqSimpleFoam and buoyantSimpleFoam solvers were considered. The transient solvers buoyantBoussinesqPimpleFoam and buoyantPimpleFoam were not considered for this tutorial due to the time restrictions.

1.3 Overview of the tutorial

The tutorial is mainly divided to two sections.

- Using the solver buoyantBoussinesqSimpleFoam and implementing a new temperature dependent viscosity model for the case of vertical slot.
- Using the solver buoyantSimpleFoam for the same case set up and calculating wall heat flux at the boundaries using the utility available in OpenFOAM.

Chapter 2

buoyantBoussinesqSimpleFoam

This chapter deals with the underlying theory and the governing equations that are solved by the solver buoyantBoussinesqSimpleFoam.

2.1 Solver description

The source code of the solver and governing equations solved are described in this section. The source code of the solver is located in

`$FOAM_SOLVERS/heatTransfer/buoyantBoussinesqSimpleFoam`

```
buoyantBoussinesqSimpleFoam
├── buoyantBoussinesqSimpleFoam.C
├── createFields.H
├── Make
│   ├── files
│   └── options
├── readTransportProperties.H
├── UEqn.H
├── TEqn.H
└── pEqn.H
```

The main source code of the solver is `buoyantBoussinesqSimpleFoam.C`. It contains several default .H files included in it.

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "radiationModel.H"
#include "fvOptions.H"
#include "simpleControl.H"


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

    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
```

```
#include "createControl.H"
#include "createFields.H"
#include "createFvOptions.H"
#include "initContinuityErrs.H"
```

The purpose of the .H files included are explained below briefly:

- `#include "fvCFD.H"` - This is the standard header file for Finite volume method in OpenFOAM. It in-turn includes a lot of header files of the classes that are used in the finite volume solvers.
- `#include "singlePhaseTransportModel.H"` - This class is for transport model based on the viscosity for single phase incompressible flows.
- `#include "turbulentTransportModel.H"` - This is the abstract base class for incompressible turbulence models.
- `#include "radiationModel.H"` - This Class is for modelling the radiation heat transfer.
- `#include "fvOptions.H"` - This is the Class for fvOptions in OpenFOAM such as run time selectable physics and more.
- `#include "simpleControl.H"` - SIMPLE control class to supply convergence information/checks for the SIMPLE loop.

The following .H files are included in the main loop. These are certain codes that are inserted to perform certain operations.

- `#include "postProcess.H"` - Executes applications `FunctionObjects` to postprocess the existing results.
- `#include "setRootCase.H"` - Checks the folder structure of the case.
- `#include "createTime.H"` - Checks the `runTime` according to `controlDict` and initiates time variables.
- `#include "createMesh.H"` - Creates the mesh for the runTime.
- `#include "createControl.H"` - Defines the solution control algorithm.
- `#include "createFields.H"` - Creates the fields for the domain, i.e U, p, T, DT, phi.
- `#include "createFvOptions.H"` - Defines the FV options.
- `#include "initContinuityErrs.H"` - Declares and initializes the continuity errors.

The next part of the code includes the solver loop which calculates the fields. The runTime field is set and the equations for velocity(UEqn), pressure(PEqn) and energy(TEqn) are loaded and solved.

```
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"
    }

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();

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

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

return 0;
```

The `createFields.H` file located in the directory creates the thermo-physical and transport properties as fields that will be used by solver. Thermo-physical properties include Temperature (T), velocity (U) and pressure (p_{rgh}). Total Pressure is expressed as $\rho.g + p_{rhg}$. Transport properties include density (ρ) and absolute viscosity (ν). Density is expressed using Boussinesq assumption, given as

$$\rho = 1 - \beta(T - T_{ref}) \tag{2.1}$$

Here β is thermal expansion co-efficient ($1/K$), T (K) and T_{ref} (K) are temperature and reference temperature respectively. The absolute viscosity is defined using the `singlePhaseTransportModel` which is a viscosity-based transport model for single phase flows.

The governing equation for velocity is solved in `UEqn.H`.

```
tmp<fvVectorMatrix> tUEqn
(
    fvm::div(phi, U)
  + MRF.DDt(U)
  + turbulence->divDevReff(U)
 ==
    fvOptions(U)
);
fvVectorMatrix& UEqn = tUEqn.ref();

UEqn.relax();

fvOptions.constrain(UEqn);

if (simple.momentumPredictor())
{
    solve
    (
        UEqn
```

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

      fvOptions.correct(U);
  }
```

The momentum equation is solved in UEqn.H, given as

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\rho \mathbf{u}\mathbf{u}) + \nabla.(\nu_{eff}\nabla \mathbf{u}) + \nabla.\Big(\nu_{eff}(\nabla \mathbf{u})^T - \nu_{eff}\frac{2}{3}tr(\nabla \mathbf{u})^T I\Big) = -(\nabla\rho)g.h.f - \nabla p_{rgh} \quad (2.2)$$

Here $\mathbf{u}$ represents the velocity vector, $(\nabla\rho)g.h.f$ term in the RHS represents the body force acting on the fluid element and $\nu_{eff} = \nu + \nu_t$.

Comparing the Eqn. 2.2 to the `UEqn.H`, it is clear that `fvm::div(phi, U)` represents the convective term and `MRF.DDt(U)` represents the time derivative of the velocity. The `turbulence->divDevReff(U)` function in the code refers to the viscous shear stress term in the momentum equation and given as

$$divDevReff(U) = \nabla.(\nu_{eff}\nabla \mathbf{u}) + \nabla.\Big(\nu_{eff}(\nabla \mathbf{u})^T - \nu_{eff}\frac{2}{3}tr(\nabla \mathbf{u})^T I\Big) \quad (2.3)$$

The terms `ghf*fvc::snGrad(rhok)` and `fvc::snGrad(p_rgh)` represents the body force and the pressure gradient terms respectively.

The energy equation is solved in `TEqn.H`

```
{
alphat = turbulence->nut()/Prt;
alphat.correctBoundaryConditions();

volScalarField alphaEff("alphaEff", turbulence->nu()/Pr + alphat);

fvScalarMatrix TEqn
(
    fvm::div(phi, T)
  - fvm::laplacian(alphaEff, T)
 ==
    radiation->ST(rhoCpRef, T)
  + fvOptions(T)
);

TEqn.relax();

fvOptions.constrain(TEqn);

TEqn.solve();

radiation->correct();
```

```
    fvOptions.correct(T);

    rhok = 1.0 - beta*(T - TRef);
}
```

The governing equation for temperature is solved, given as

$$\nabla.(\rho \mathbf{u} T) - \nabla.\alpha_{eff}\nabla T = S_{radiation} + S_T \tag{2.4}$$

Where, $\alpha_{eff} = \frac{\nu_t}{pr_t} + \frac{\nu}{pr}$, $S_{radiation}$ and S_T are source terms due to radiation and user defined source term respectively. The density is updated as $\rho = 1 - \beta(T - T_{ref})$

To calculate the pressure, SIMPLE algorithm is implemented for pressure velocity coupling. The equation for pressure is solved in `PEqn.H`.

```
    {
    volScalarField rAU("rAU", 1.0/UEqn.A());
    surfaceScalarField rAUf("rAUf", fvc::interpolate(rAU));
    volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p_rgh));

    tUEqn.clear();

    surfaceScalarField phig(-rAUf*ghf*fvc::snGrad(rhok)*mesh.magSf());

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

    MRF.makeRelative(phiHbyA);

    adjustPhi(phiHbyA, U, p_rgh);

    phiHbyA += phig;

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

    while (simple.correctNonOrthogonal())
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rAUf, p_rgh) == fvc::div(phiHbyA)
        );

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

        p_rghEqn.solve();

        if (simple.finalNonOrthogonalIter())
        {
            // 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())/rAUf);
            U.correctBoundaryConditions();
            fvOptions.correct(U);
        }
    }

    #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;
    }
}
```

In this code, the velocity at the face is obtained by interpolating the semi-discretized form of momentum equation. The semi discretized momentum equation does not include the pressure gradient term. The continuity equation along with semi-discretized momentum equation are used to solve the pressure (p_{rgh}). The equation for pressure is solved for prescribed number of non orthogonal corrector steps. Then the flux is corrected based on the solved pressure. The pressure is under-relaxed for momentum corrector and the velocity is corrected. Finally pressure is calculated as $p = p_{rgh} + \rho gh$.

Chapter 3

Temperature based viscosity model

This chapter gives a brief explanation about the viscosity models in OpenFOAM and also describes the procedure to implement a new temperature based viscosity model.

3.1 Viscosity models

The most important factors affecting the viscosity of a fluid are the temperature and the shear rate. For Newtonian fluids, viscosity does not vary with the shear rate, thus it is only a function of temperature. OpenFOAM has four viscosity models for non-Newtonian fluids and one for Newtonian fluids. All the five models do not consider temperature dependency of the viscosity.

The `viscosityModel` class in OpenFOAM is an abstract class. The five viscosity models are implemented as sub classes which inherit from the `viscosityModel` class. All the viscosity models return the corrected absolute viscosity(nu) when they are called. The five models available in OpenFOAM are:

- BirdCarreu
- CrossPowerLaw
- HerschelBulkley
- Newtonian
- powerLaw

The link between the turbulence model, the transport model and the viscosity model can be explained as follows. The `TransportModel` class is the base class for all the transport models used by the turbulence models. The `TransportModel` class has different derived classes which manage different transport properties based on the type of flow. For single phase flow, the `singlePhaseTransportModel` class reads the viscosity **nu**, corrects it and fetches it when it is called in the `turbulenceModel` class. The `singlePhaseTransportModel` class is a derived class which is based on the `viscosityModel` class. It has a pointer `viscosityModelPtr` which is a private member data.

```
class singlePhaseTransportModel
:
    public IOdictionary,
    public transportModel
{
    // Private Data
```

```
        autoPtr<viscosityModel> viscosityModelPtr_;
```

The member function **nu** returns the value of viscosity read from `viscosityModel` and the member function **correct()** corrects the viscosity.

```
Foam::tmp<Foam::volScalarField>
Foam::singlePhaseTransportModel::nu() const
{
    return viscosityModelPtr_->nu();
}
void Foam::singlePhaseTransportModel::correct()
{
    viscosityModelPtr_->correct();
}
```

The class `turbulenceModel` has a private member data `transportModel` which returns the value of viscosity (**nu**) as a volume scalar field.

3.2 Temperature based viscosity model

The viscosity model for Newtonian fluids in OpenFOAM considers viscosity as a constant value. A new viscosity model with viscosity as a function of temperature will be implemented. The new model is based on the power law model and Vogel's equation of viscosity. The power Law viscosity model available in OpenFOAM expresses viscosity as

$$\nu = k.(\dot{\gamma})^{(n-1)} \tag{3.1}$$

Vogel's equation for viscosity which is a temperature based model expresses viscosity as

$$log\nu = \Big(A + \frac{B}{T+C}\Big) \tag{3.2}$$

Here ν is in (mm^2/s). A,B and C are correlation parameters determined from viscosity measurements at three or more points[2].

The new temperature-based viscosity model expresses viscosity as

$$\nu = exp\Big(A + \frac{B}{T+C}\Big).(\dot{\gamma})^{(n-1)} \tag{3.3}$$

This law can be used both for Newtonian and non Newtonian fluids. With $n = 1$, it behaves as Vogel's equation of viscosity. The already available `powerLaw` will be used to implement the new model.

The complete implementation of the new viscosity model can be done by following the steps below. Also the new viscosity model is available in the accompanied files with the name `tempLaw`.

- **Copy the powerLaw to the run directory**
  Run the following commands in the terminal to copy existing powerLaw model.

  ```
  OF4x
  run
  cp -r $FOAM_SRC/transportModels/incompressible/viscosityModels/powerLaw tempLaw
  cd tempLaw/
  mkdir Make
  cd Make
  ```

```
cp $FOAM_SRC/transportModels/incompressible/Make/files files
cp $FOAM_SRC/transportModels/incompressible/Make/options options
```

The power law directory has `powerLaw.C`, `powerLaw.H` and `Make` directory in it.

- **In Make/files(replace)**

```
tempLaw.C
LIB = $(FOAM_USER_LIBBIN)/libusertempLaw
```

- **In Make/options(replace)**

```
EXE_INC = \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude/ \
    -I$(LIB_SRC)/finiteVolume/lnInclude

LIB_LIBS = \
    -lfiniteVolume
```

- **In powerLaw.C (Add/replace)**

Replace with the below part of code in the private member functions section.

```
// * * * * * * * * * * * * Private Member Functions  * * * * * * * * *  * * //
Foam::tmp<Foam::volScalarField>
Foam::viscosityModels::tempLaw::calcNu() const
{

const volScalarField& T= U_.mesh().lookupObject<volScalarField>("T");
    return max
    (
        nuMin_,
        min
        (
            nuMax_,
            (scalar(0.000001)*m_*Foam::exp(A_ +
            B_/(T*1.0/dimensionedScalar("one", dimTemperature, 1.0) + C_)))*pow
            (
                max
                (
                    dimensionedScalar("one", dimTime, 1.0)*strainRate(),
                    dimensionedScalar("VSMALL", dimless, VSMALL)
                ),
                n_.value() - scalar(1.0)
            )
        )
    );
}
```

Replace with the below code under the constructor section.

```
// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * *  * //
```

```
Foam::viscosityModels::tempLaw::tempLaw
(
    const word& name,
    const dictionary& viscosityProperties,
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    viscosityModel(name, viscosityProperties, U, phi),
    tempLawCoeffs_(viscosityProperties.subDict(typeName + "Coeffs")),
    m_("m", dimViscosity, tempLawCoeffs_),
    A_("A", dimless, tempLawCoeffs_),
    B_("B", dimless, tempLawCoeffs_),
    C_("C", dimless, tempLawCoeffs_),
    n_("n", dimless, tempLawCoeffs_),
    nuMin_("nuMin", dimViscosity, tempLawCoeffs_),
    nuMax_("nuMax", dimViscosity, tempLawCoeffs_),
    nu_
    (
        IOobject
        (
            name,
            U_.time().timeName(),
            U_.db(),
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        calcNu()
    )
{}
```

Replace with the below code in the member functions section.

```
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * ** * //

bool Foam::viscosityModels::tempLaw::read
(
    const dictionary& viscosityProperties
)
{
    viscosityModel::read(viscosityProperties);

    tempLawCoeffs_ = viscosityProperties.subDict(typeName + "Coeffs");

    tempLawCoeffs_.lookup("m") >> m_;
    tempLawCoeffs_.lookup("A") >> A_;
    tempLawCoeffs_.lookup("B") >> B_;
    tempLawCoeffs_.lookup("C") >> C_;
    tempLawCoeffs_.lookup("n") >> n_;
    tempLawCoeffs_.lookup("nuMin") >> nuMin_;
    tempLawCoeffs_.lookup("nuMax") >> nuMax_;
```

```
    return true;
}
```

- **In powerLaw.H(Add/replace)**
  Add/replace the below code in the private data section at the beginning.

```
// Private data

        dictionary tempLawCoeffs_;

        dimensionedScalar m_;
        dimensionedScalar A_;
        dimensionedScalar B_;
        dimensionedScalar C_;
        dimensionedScalar n_;
        dimensionedScalar nuMin_;
        dimensionedScalar nuMax_;

        volScalarField nu_;
```

- **Compilation**

  Run the following commands in the terminal to compile the new model.

```
cd ..
mv powerLaw.H tempLaw.H
mv powerLaw.C tempLaw.C
sed -i s/powerLaw/tempLaw/g tempLaw.C
sed -i s/powerLaw/tempLaw/g tempLaw.H
wmake libso
```

Chapter 4

Running the case

This chapter describes the procedure to run the simulation of heat transfer in a vertical slot using `buoyantBoussinesqSimpleFoam` and the new viscosity model.

4.1 Copy the tutorial

The tutorial available in OpenFOAM is copied and modified as per the necessary case set-up. The case-set up is available in the accompanied files with the name `verticalSlot`. Also the case can be set up by following the procedure described in the sections below. To copy the tutorial to the run directory, run the following commands in the terminal.

```
cp -r $FOAM_TUTORIALS/heatTransfer/buoyantBoussinesqSimpleFoam/hotRoom $FOAM_RUN
run
mv hotRoom verticalSlot
cd verticalSlot
```

4.2 Mesh generation

Replace the `system/blockMeshDict` with the following code

```
convertToMeters 0.001;
vertices
(
    ( 0     0  0)
    (15     0  0)
    (15  300  0)
    ( 0  300  0)
    ( 0     0   1)
    (15     0   1)
    (15  300   1)
    ( 0  300   1)
);
edges
(
);
blocks
(
    hex (0 1 2 3 4 5 6 7) (30 150 1) simpleGrading (1 1 1)
);
boundary
```

```
(
    frontAndBack
    {
        type empty;
        faces
        (
            (0 1 2 3)
            (4 7 6 5)
        );
    }
    topAndBottom
    {
        type wall;
        faces
        (
            (0 4 5 1)
            (3 2 6 7)
        );
    }
    hot
    {
        type wall;
        faces
        (
            (0 3 7 4)
        );
    }
    cold
    {
        type wall;
        faces
        (
            (6 2 1 5)
        );
    }
);
mergePatchPairs
(
);
```

Mesh can be created using `blockMesh` command.

4.3 Boundary conditions

The `0/` directory of the tutorial has eight files named `alphat, k, epsilon, nut, p, p_rgh, U and T.orig`. But to use the v2f turbulence model, two additional files v2 and f must be added. Initial guess of k, epsilon, v2 and f are calculated using empirical relations [4].

$$k = \frac{3}{2}(v.turbulentintensity), \epsilon = \frac{C_{\mu}^{0.75} * k}{l} \tag{4.1}$$

$$v2 = \frac{2}{3}k, f = zeroGradient \tag{4.2}$$

The name of the file `T.orig` has to be changed to `T`. All the files are modified as per the initial conditions of the case and the modifications are described in the Appendix.

4.4 Transport properties

Replace `constant/transportProperties` with the following code

```
transportModel tempLaw;
tempLawCoeffs
{
    m             m [0 2 -1 0 0 0 0] 1;
    A             A [0 0 0 0 0 0 0] -2.2;
    B             B [0 0 0 0 0 0 0] 812.9;
    C             C [0 0 0 0 0 0 0] -140;
    n             n [0 0 0 0 0 0 0] 1;
    nuMin         nuMin [0 2 -1 0 0 0 0] 5e-6;
    nuMax         nuMax [0 2 -1 0 0 0 0] 5.5e-5;
   }
// Thermal expansion coefficient
beta            [0 0 0 -1 0 0 0] 1.05e-03;
// Reference temperature
TRef            [0 0 0 1 0 0 0] 300;
// Laminar Prandtl number
Pr              [0 0 0 0 0 0 0] 10;
// Turbulent Prandtl number
Prt             [0 0 0 0 0 0 0] 0.65;
```

Replace the `constant/turbulenceProperties` with the following code

```
simulationType RAS;
RAS
{
    RASModel        v2f;

    turbulence      on;

    printCoeffs     on;
}
```

4.5 Solution control

The first thing to be done is to remove the `system/setFieldsDict` in the system directory of the tutorial since it is not necessary for the current case.

```
cd ../system
rm setFieldsDict
```

To use the new viscosity model, it has to be added in the `system/controlDict`. Add the following at the end of the `system/controlDict`

```
libs
(
"libusertempLaw.so"
);
```

The numerical schemes for solving v2 and f have to be added in the `system/fvSchemes`. Add the following under the `divschemes` section in `system/fvSchemes`.

```
div(phi,v2)      bounded Gauss upwind;
div(phi,f)       bounded Gauss upwind;
```

Solvers for v2 and f have to be specified. It can be done by replacing with the below code under the `solvers` section in `system/fvSolution`.

```
solvers
{
    p_rgh
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-05;
        relTol          0.01;
    }

    "(U|T|k|epsilon|v2)"
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }
    f
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-5;
        relTol          0.1;
    }
}
```

Also `"(k|epsilon|omega)"` should be replaced as `"(k|epsilon|v2|f)"` in the `SIMPLE` and the `relaxationFactors` sections of the `system/fvSolutions`.

4.6 Results

After all the case set-up, the simulation can be run by

```
cd ..
buoyantBoussinesqSimpleFoam
```

The results can be checked using `paraFoam`.

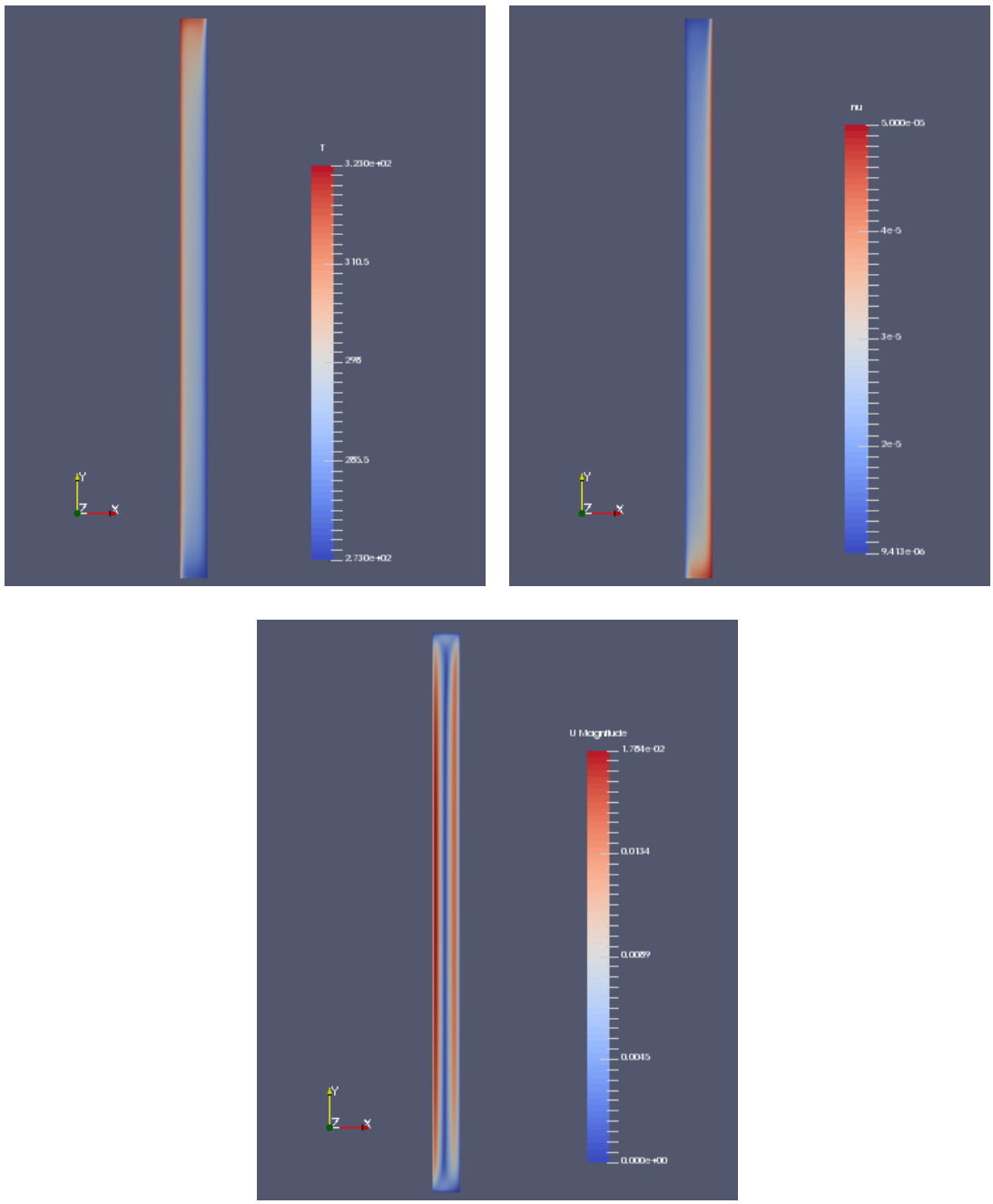

Figure 4.1: Plots of Temperature (top-left), Viscosity (Top-right) and Velocity (Bottom)

Figure. 4.1 shows the plots of temperature, viscosity and velocity contours in the slot. It can be observed from the velocity contour that the velocity near the two walls of the slot are of higher magnitude and the velocity is almost zero at the center of the slot. Also from the temperature contour it can be observed that at the top of the slot, the temperature is being distributed from the hot wall to the cold wall and it is the reverse at the bottom. This indicates that there is a unicellular motion of the fluid inside the slot.

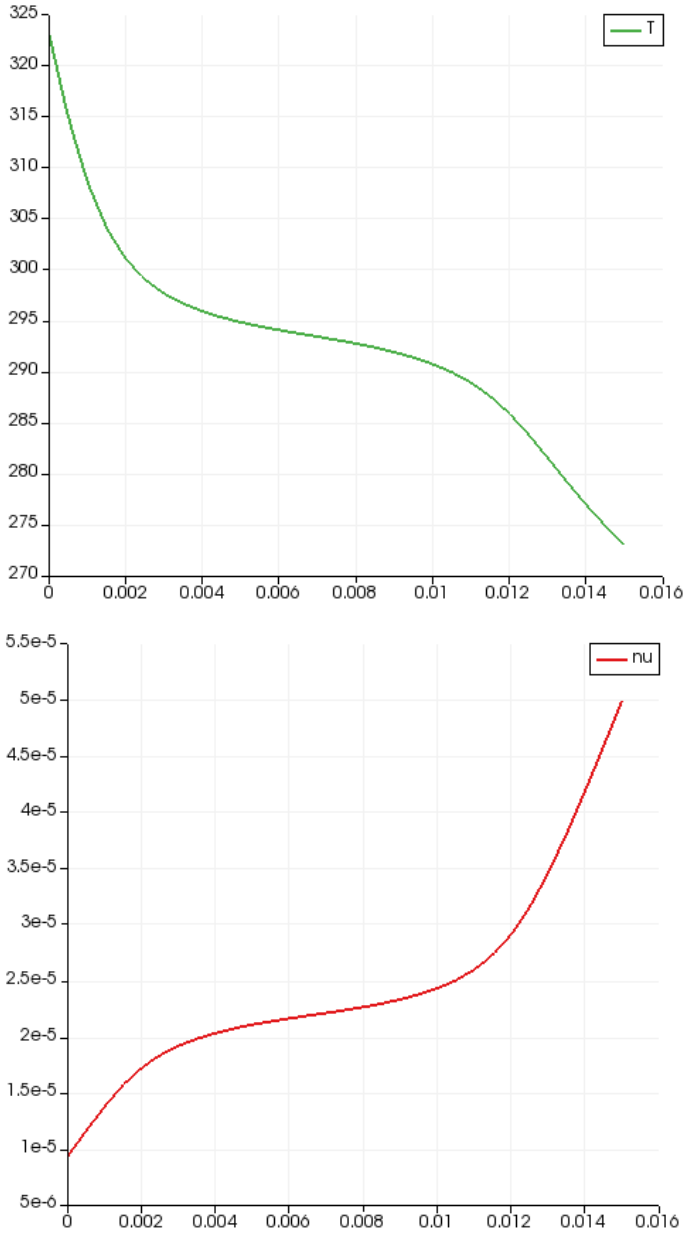

Figure 4.2: Temperature and viscosity along width of the slot

It can be observed from the viscosity profile in Fig. 4.1 that the viscosity is minimum where the temperature is maximum. This shows the viscosity model implemented accounts the influence of temperature on the viscosity. It is clearly evident from the X-Y plots of temperature and viscosity along width of the slot at a height of 150 mm , see Fig. 4.2

Chapter 5

buoyantSimpleFoam

This chapter deals with the underlying theory and the governing equations that are solved by the solver buoyantSimpleFoam. Since the folder organization, the solver structure and most of the include files are similar to that used in `buoyantBoussinesqSimpleFoam`, only the governing equations and few important differences between the two solvers are explained in this chapter.

5.1 Solver description

The source code of `buoyantSimpleFoam` is located in

`$FOAM_SOLVERS/heatTransfer/buoyantSimpleFoam`

```
buoyantSimpleFoam
├── buoyantSimpleFoam.C
├── createFieldRefs.H
├── createFields.H
├── Make
│   ├── files
│   └── options
├── UEqn.H
├── EEqn.H
└── pEqn.H
```

The structure of the solver code `buoyantSimpleFoam` is similar to that of `buoyantBoussinesqSimpleFoam`. The main solver code is `buoyantSimpleFoam.C`. It has the .H files included in it similar to the previous solver. Since this solver deals with compressible fluids, it has an additional .H file called `rhoThermo.H` included in its code which is a density based thermodynamic model.

In `createFields.H`, it can be observed that the density (rho) is defined using `thermo.rho()`. Also the temperature is not directly created as a field. Equation for enthalpy (h) or internal energy (e) is solved and the temperature is calculated based on the thermodynamic model defined. Thus the fields for enthalpy or internal energy is created using the `rhoThermo` model. The compressible face flux is initiated using `compressibleCreatePhi.H`. The governing equations for velocity (U), energy (h/e) and pressure (p_{rgh}) are solved in `UEqn.H, EEqn.H` and `PEqn.H` respectively.

The momentum equation is solved in `UEqn.H` and is given as

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla.(\rho \mathbf{u}\mathbf{u}) + \nabla.(\mu_{eff}\nabla \mathbf{u}) + \nabla.\Big(\mu_{eff}(\nabla \mathbf{u})^T - \mu_{eff}\frac{2}{3}tr(\nabla \mathbf{u})^T I\Big) = -(\nabla \rho)g.h.f - \nabla p_{rgh} \quad (5.1)$$

Here $\mu_{eff} = \mu + \mu_t$. μ represents laminar viscosity and μ_t represents turbulent viscosity.

The Energy equation is solved in `EEqn.H` and is given as:

$$\nabla.(\rho \mathbf{u} h_e) + \nabla.(\rho \mathbf{u}(0.5\sqrt{U} + \frac{p}{\rho})) - \nabla.\alpha_{eff}\nabla T = S_{radiation} + S_h \tag{5.2}$$

Here, $\alpha_{eff} = \alpha + \alpha_t$, α $[kg/(m.s)]$ is known as thermal conductivity and is calculated as

$$\alpha = \frac{k}{\rho c_p} \tag{5.3}$$

k $[W/(m.K)]$ is the thermal conductivity and c_p $[J/(kg.K)]$ is the specific heat capacity. $S_{radiation}$ and S_h are the radiation and user defined source terms respectively.

To calculate the pressure, SIMPLE algorithm is implemented and it is similar to that of `PEqn.H` in the previous solver. Here along with the velocity, density is interpolated at the faces to calculate the flux and then the pressure is calculated. The mass flux and velocity are corrected in a way similar to the previous solver.

Chapter 6

Running the case

The tutorial available in OpenFOAM is copied and modified as per the necessary case set-up. The case set-up is available in the accompanied files with the file name `verticalSlot_bsf`. Also the case can be set up by following the steps described in the below sections. To copy the tutorial to the run directory, run the following commands in the terminal.

```
cp -r $FOAM_TUTORIALS/heatTransfer/buoyantSimpleFoam/buoyantCavity $FOAM_RUN
mv buoyantCavity verticalSlot_bsf
cd verticalSlot_bsf
rm -r validation
```

The case directory organization is similar to the one that has been used in the previous section. It has `0/, constant/` and `system/` directories.

6.1 Mesh generation

The `blockMeshDict` in the system directory can be replaced with the one implemented in the incompressible (previous) case since the geometry and mesh will be the same. The below steps can be followed to copy the `blockMeshDict` file from the incompressible case.

```
cd system
rm blockMeshDict
cp $FOAM_RUN/verticalSlot/system/blockMeshDict blockMeshDict
```

The mesh can be generated by

```
cd ..
blockMesh
```

6.2 Boundary conditions

The boundary conditions are similar to that applied to the incompressible case. Hence the `0/` directory will be replaced with the default one but the boundary conditions for `alphat` and `p_rgh` is changed. The following steps can be followed to copy the 0/ directory to the current case directory.

```
rm -r 0
cp -r $FOAM_RUN/verticalSlot/0 0
cd 0/
```

The `alphat` directory must be modified since the compressible wall function has to be used in this case.

The alphat file has to be replaced with the code below.

```
dimensions      [1 -1 -1 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
    {
        type            compressible::alphatWallFunction;
        Prt             0.65;
        value           uniform 0;
    }
    hot
    {
        type            compressible::alphatWallFunction;
        Prt             0.65;
        value           uniform 0;
    }
    cold
    {
        type            compressible::alphatWallFunction;
        Prt             0.65;
        value           uniform 0;
    }
}
```

The boundary condition for `p_rgh` has to be replaced by the following code

```
dimensions      [1 -1 -2 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
    hot
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
    cold
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
```

```
}
```

Also the boundary condition for `p` has to be replaced by the following code

```
dimensions      [1 -1 -2 0 0 0 0];
internalField   uniform 101325;
boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
    {
        type            calculated;
        value           $internalField;
    }
    hot
    {
        type            calculated;
        value           $internalField;
    }
    cold
    {
        type            calculated;
        value           $internalField;
    }
}
```

6.3 Transport properties

The thermo-physical properties should be changed to add the properties of silicone oil.
The `constant/thermophysicalProperties` has to be replaced by the following code:

```
thermoType
{
    type            heRhoThermo;
    mixture         pureMixture;
    transport       const;
    thermo          hConst;
    equationOfState perfectGas;
    specie          specie;
    energy          sensibleEnthalpy;
}
mixture
{
    specie
    {
        nMoles          1;
        molWeight       162.38;
    }
    thermodynamics
    {
```

```
        Cp              1600;
        Hf              0;
    }
    transport
    {
        mu              5e-05;
        Pr              10;
    }
}
```

`buoyantSimpleFoam` solver uses `rhoThermo` to access the thermo-physical properties. It is a density based model. `mixture` specifies the mixture composition. `pureMixture` is used to specify a mixture without any reaction. `transport` specifies the transport model to be used. The transport model evaluates properties like dynamic viscosity, thermal conductivity and thermal diffusivity. `thermo` specifies the type of thermodynamic model used. `hthermo` assumes constant *cp*. The equations of state available in the thermo-physical library are rhoConst, perfectGas, incompressiblePerfectGas, perfectFluid etc. Density is calculated based on these equations. `perfectGas` is used in this case. `energy` specifies the form of energy used in the case. This defines which parameter has to be solved through the energy equation. sensibleEnthalpy and sensibleInternalEnergy are few examples. SensibleEnthalpy is used in this case.

The turbulence model used is similar to the one used in the incompressible case. Thus `constant/turbulenceProperties` file can be copied from the previous case by the following steps

```
cd ../constant
rm turbulenceProperties
cp $FOAM_RUN/verticalSlot/constant/turbulenceProperties turbulenceProperties
```

6.4 Solution control

The first thing to be done in the `system` directory is to remove the `sample` file. This can be done by

```
cd ../system
rm sample
```

The numerical schemes for solving v2 and f have to be added in `system/fvSchemes`. Add the following under the `divschemes` section in `system/fvSchemes`.

```
div(phi,v2) bounded Gauss limitedLinear 0.2;
div(phi,f) bounded Gauss limitedLinear 0.2;
```

Solvers for v2 and f have to be specified. It can be done by adding the below code under the solvers section in `system/fvSolution`.

```
solvers
{
    p_rgh
    {
        solver          GAMG;
        tolerance       1e-7;
        relTol          0.01;

        smoother        DICGaussSeidel;
```

```
    }

    "(U|h|k|epsilon|v2)"
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-8;
        relTol          0.1;
    }
    "(f)"
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-8;
        relTol          0.1;
    }
}
```

Also `"(k|epsilon|omega)"` should be replaced as `"(k|epsilon|v2|f)"` in the `SIMPLE` and `relaxationFactors` sections of `system/fvSolutions`.

6.5 Results

The case can be run by

```
cd ..
buoyantSimpleFoam
```

The results can be post-processed using `paraFoam`

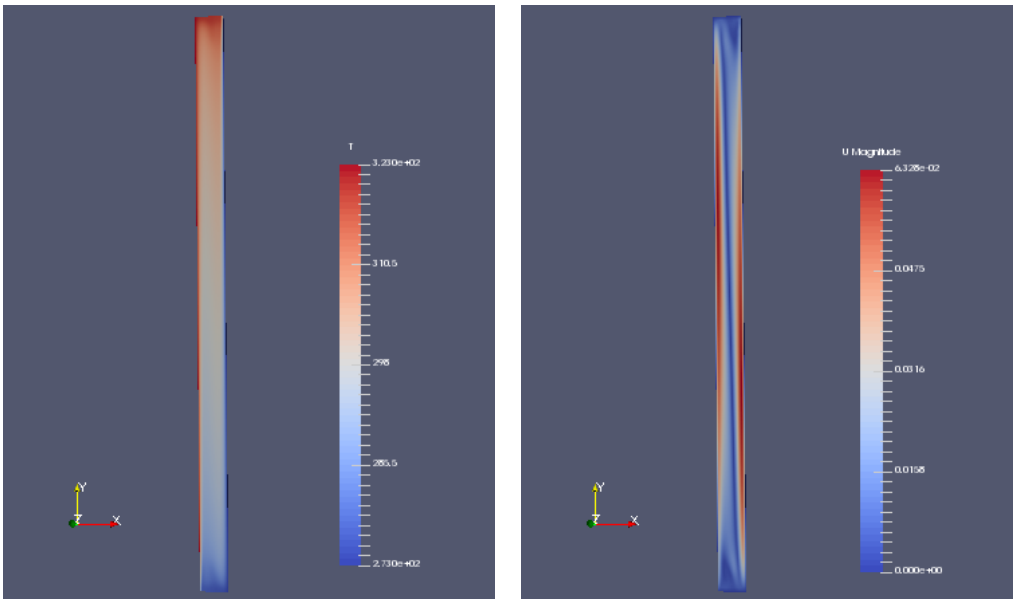

Figure 6.1: Plots of temperature (left) and velocity (right) contours

Figure. 6.1 shows the contours of temperature and velocity in the vertical slot simulated using `buoyantSimpleFoam` solver. The temperature distribution is approximately similar to the previous

case but the velocity profile does not clearly define the unicellular motion of fluid inside the slot. The magnitude of velocities are higher near the two walls but not along the entire height of the slot.

6.6 Wall heat flux

To check the global conservation of flux, the wall heat flux at the boundaries have to be calculated. This can be done using the `wallHeatFlux` utility in OpenFOAM. This utility will be used to calculate the sum of heat flux on all the boundaries.

6.6.1 The wallHeatFlux utility

The source code of the utility is located in:

`$FOAM_APP/utilities/postProcessing/toBeFunctionObjects/wallHeatFlux`

The directory has a Make directory, `createFields.H` and `wallHeatFlux.C`. The main source code can be found in file `wallHeatFlux.C`. The heat flux is calculated by

```
surfaceScalarField heatFlux
        (
            fvc::interpolate
            (
                (
                    turbulence.valid()
                  ? turbulence->alphaEff()()
                  : thermo->alpha()
                )
            )*fvc::snGrad(h)
        );
```

The total heat flux at all the boundaries is calculated as

```
forAll(patchHeatFlux, patchi)
        {
            if (isA<wallFvPatch>(mesh.boundary()[patchi]))
            {
                scalar convFlux = gSum(magSf[patchi]*patchHeatFlux[patchi]);
                scalar radFlux = -gSum(magSf[patchi]*patchRadHeatFlux[patchi]);

                Info<< mesh.boundary()[patchi].name() << endl
                    << "    convective: " << convFlux << endl
                    << "    radiative:  " << radFlux << endl
                    << "    total:      " << convFlux + radFlux << endl;
            }
        }
```

Here the total flux is considered as the sum of convective flux and the radiative flux.

6.6.2 Implementation

This section describes the procedure of using the utility for calculating wall heat flux at the boundaries as a post-processing step of the simulation performed using the `buoyantSimpleFoam` solver.

The wall heat flux utility can be copied to the run directory by running the commands below in the terminal

```
OF4x
run
cp -r $FOAM_APP/utilities/postProcessing/toBeFunctionObjects/wallHeatFlux $FOAM_RUN
```

The `Make/files` has to be modified so as to use it for the current case.

Replace with the following in the `Make/files`

```
wallHeatFlux.C
EXE = $(FOAM_USER_APPBIN)/wallHeatFlux
```

The utility can be compiled used `wmake`. The following commands can be run in the terminal to calculate the wall heat flux for our case.

```
cd verticalSlot_bsf
wallHeatFlux
```

The results will look like as shown below.

```
Time = 1000
Selecting thermodynamics package
{
    type            heRhoThermo;
    mixture         pureMixture;
    transport       const;
    thermo          hConst;
    equationOfState perfectGas;
    specie          specie;
    energy          sensibleEnthalpy;
}

Reading/calculating face flux field phi

Selecting turbulence model type RAS
Selecting RAS turbulence model v2f
bounding v2, min: 4.54768e-42 max: 2.65821e-06 average: 8.20716e-07
v2fCoeffs
{
    Cmu             0.22;
    CmuKEps         0.09;
    C1              1.4;
    C2              0.3;
    CL              0.23;
    Ceta            70;
    Ceps2           1.9;
    Ceps3           -0.33;
    sigmaK          1;
    sigmaEps        1.3;
}
Wall heat fluxes [W]
topAndBottom
    convective: 0
    radiative:  -0
    total:      0
hot
```

```
    convective: 0.237631
    radiative:  -0
    total:      0.237631
cold
    convective: -0.241561
    radiative:  -0
    total:      -0.241561

End
```

Chapter 7

Conclusion

By comparing the results of the simulation performed using the `buoyantBoussinesqSimpleFoam` and `buoyantSimpleFoam` solvers, it can be concluded that the results obtained from both the solvers matched the practical situation to certain degree of accuracy. However the results obtained from `buoyantBoussinesqSimpleFoam` solver seems more promising qualitatively. Also since there is no considerable density variation in the current case, `buoyantBoussinesqSimpleFoam` solver with the temperature-based viscosity model suits the case better. However, it could be better to deduce a conclusion by comparing the results obtained with experimental data if it was available.

Study questions

1. Where is the `singlePhaseTransport` model called in the `buoyantBoussinesqSimpleFoam` solver?
2. How does the class `turbulenceModel` access the viscosity nu?
3. What are the changes that have to be made in the case folder to implement a modified transport model?
4. Mention the important differences between the buoyantBoussinesqSimpleFoam and buoyantSimpleFoam solvers?

Bibliography

[1] Wakitani SS. Experiments on Convective Instability of Large Prandtl Number Fluids in a Vertical Slot. ASME. J. Heat Transfer. 1994;116(1):120-126. doi:10.1115/1.2910845.

[2] Yuan, W., Hansen, A.C., Zhang, Q. et al. J, Temperature-Dependent Kinematic Viscosity of Selected Biodiesel Fuels and Blends with Diesel Fuel. Amer Oil Chem Soc (2005) 82: 195. doi:10.1007/s11746-005-5172-6

[3] Naser Hamedi, Non-Newtonian Models in OpenFOAM - Implementation of a non-Newtonian model. `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2014/Naser_Hamedi/Documents/Report.pdf`

[4] Johan Magnusson. conjugateHeatFoam with explanational tutorial together with a buoyancy driven flow tutorial and a convective conductive tutorial. `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/johanMagnusson/johanMagnussonReport.pdf`

Appendix

Boundary conditions for the case set-up using the buoyantBoussinesqSimpleFoam solver

Replace with the following in the `alphat` file.

```
internalField   uniform 0;
boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
    {
        type            alphatJayatillekeWallFunction;
        Prt             0.65;
        value           uniform 0;
    }
    hot
    {
        type            alphatJayatillekeWallFunction;
        Prt             0.65;
        value           uniform 0;
    }
    cold
    {
        type            alphatJayatillekeWallFunction;
        Prt             0.65;
        value           uniform 0;
    }
}
```

Replace with the following in the `epsilon` file.

```
internalField   uniform 0.001;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }
```

```
    topAndBottom
    {
        type            epsilonLowReWallFunction;
        value           uniform 0.001;
    }
    hot
    {
        type            epsilonLowReWallFunction;
        value           uniform 0.001;
    }
    cold
    {
        type            epsilonLowReWallFunction;
        value           uniform 0.001;
    }
}
```

Replace with the following in the `k` file.

```
internalField   uniform 0.015;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
    {
        type            kLowReWallFunction;
        value           uniform 0.015;
    }
    hot
    {
        type            kLowReWallFunction;
        value           uniform 0.015;
    }
    cold
    {
        type            kLowReWallFunction;
        value           uniform 0.015;
    }
}
```

Replace with the following in the `nut` file.

```
internalField   uniform 0;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }
```

```
    topAndBottom
    {
        type            nutkWallFunction;
        value           uniform 0;
    }
    hot
    {
        type            nutkWallFunction;
        value           uniform 0;
    }
    cold
    {
        type            nutkWallFunction;
        value           uniform 0;
    }
}
```

Replace with the following in the `p` file

```
internalField   uniform 0;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }

    topAndBottom
    {
        type            calculated;
        value           $internalField;
    }

    hot
    {
        type            calculated;
        value           $internalField;
    }

    cold
    {
        type            calculated;
        value           $internalField;
    }
}
```

Replace with the following in the `p_rgh` file.

```
dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
```

```
{
    frontAndBack
    {
        type            empty;
    }

    topAndBottom
    {
        type            fixedFluxPressure;
        rho             rhok;
        value           uniform 0;
    }

    hot
    {
        type            fixedFluxPressure;
        rho             rhok;
        value           uniform 0;
    }

    cold
    {
        type            fixedFluxPressure;
        rho             rhok;
        value           uniform 0;
    }
}
```

Replace with the following in the `T` file.

```
internalField   uniform 298;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }

    topAndBottom
    {
        type            zeroGradient;
    }

    hot
    {
        type            fixedValue;
        value           uniform 323; // 50 degC
    }

    cold
    {
        type            fixedValue;
```

```
        value           uniform 273; // 0 degC
    }
}
```

Replace with following in the `U` file.

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    frontAndBack
    {
        type            empty;
    }

    topAndBottom
    {
        type            noSlip;
    }

    hot
    {
        type            noSlip;
    }

    cold
    {
        type            noSlip;
    }
}
```

Add the following to a new file called `v2`.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      v2;
}
dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0.01;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
```

```
    {
        type            v2WallFunction;
        value           uniform 0.01;
    }
    hot
    {
        type            v2WallFunction;
        value           uniform 0.01;
    }
    cold
    {
        type            v2WallFunction;
        value           uniform 0.01;
    }
}
```

Add the following to a new file called `f`.

```
\\Add to f file
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "0";
    object      f;
}
dimensions      [0 0 -1 0 0 0 0];

internalField   uniform 1e-10;

boundaryField
{
    frontAndBack
    {
        type            empty;
    }
    topAndBottom
    {
        type            fWallFunction;
        value           uniform 1e-10;
    }
    hot
    {
        type            fWallFunction;
        value           uniform 1e-10;
    }
    cold
    {
        type            fWallFunction;
        value           uniform 1e-10;
    }
}
```

Source code for the tempLaw

tempLaw.C

```
#include "tempLaw.H"
#include "addToRunTimeSelectionTable.H"
#include "surfaceFields.H"

// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //

namespace Foam
{
namespace viscosityModels
{
    defineTypeNameAndDebug(tempLaw, 0);

    addToRunTimeSelectionTable
    (
        viscosityModel,
        tempLaw,
        dictionary
    );
}
}


// * * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * * * * //

Foam::tmp<Foam::volScalarField>
Foam::viscosityModels::tempLaw::calcNu() const
{

const volScalarField& T= U_.mesh().lookupObject<volScalarField>("T");
    return max
    (
        nuMin_,
        min
        (
            nuMax_,
            (scalar(0.000001)*m_*Foam::exp(A_ +
            B_/(T*1.0/dimensionedScalar("one", dimTemperature, 1.0) + C_)))*pow
            (
                max
                (
                    dimensionedScalar("one", dimTime, 1.0)*strainRate(),
                    dimensionedScalar("VSMALL", dimless, VSMALL)
                ),
                n_.value() - scalar(1.0)
            )
        )
    );
}
```

```
// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //

Foam::viscosityModels::tempLaw::tempLaw
(
    const word& name,
    const dictionary& viscosityProperties,
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    viscosityModel(name, viscosityProperties, U, phi),
    tempLawCoeffs_(viscosityProperties.subDict(typeName + "Coeffs")),
    m_("m", dimViscosity, tempLawCoeffs_),
    A_("A", dimless, tempLawCoeffs_),
    B_("B", dimless, tempLawCoeffs_),
    C_("C", dimless, tempLawCoeffs_),
    n_("n", dimless, tempLawCoeffs_),
    nuMin_("nuMin", dimViscosity, tempLawCoeffs_),
    nuMax_("nuMax", dimViscosity, tempLawCoeffs_),
    nu_
    (
        IOobject
        (
            name,
            U_.time().timeName(),
            U_.db(),
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        calcNu()
    )
{}


// * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * * //

bool Foam::viscosityModels::tempLaw::read
(
    const dictionary& viscosityProperties
)
{
    viscosityModel::read(viscosityProperties);

    tempLawCoeffs_ = viscosityProperties.subDict(typeName + "Coeffs");

    tempLawCoeffs_.lookup("m") >> m_;
    tempLawCoeffs_.lookup("A") >> A_;
    tempLawCoeffs_.lookup("B") >> B_;
    tempLawCoeffs_.lookup("C") >> C_;
    tempLawCoeffs_.lookup("n") >> n_;
    tempLawCoeffs_.lookup("nuMin") >> nuMin_;
    tempLawCoeffs_.lookup("nuMax") >> nuMax_;
```

```
    return true;
}
```

tempLaw.H

```
#ifndef tempLaw_H
#define tempLaw_H

#include "viscosityModel.H"
#include "dimensionedScalar.H"
#include "volFields.H"

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

namespace Foam
{
namespace viscosityModels
{

/*---------------------------------------------------------------------------*\
                           Class tempLaw Declaration
\*---------------------------------------------------------------------------*/

class tempLaw
:
    public viscosityModel
{
    // Private data

        dictionary tempLawCoeffs_;

        dimensionedScalar m_;
        dimensionedScalar A_;
        dimensionedScalar B_;
        dimensionedScalar C_;
        dimensionedScalar n_;
        dimensionedScalar nuMin_;
        dimensionedScalar nuMax_;

        volScalarField nu_;


    // Private Member Functions

        //- Calculate and return the laminar viscosity
        tmp<volScalarField> calcNu() const;


public:

    //- Runtime type information
    TypeName("tempLaw");
```

```
    // Constructors

        //- Construct from components
        tempLaw
        (
            const word& name,
            const dictionary& viscosityProperties,
            const volVectorField& U,
            const surfaceScalarField& phi
        );


    //- Destructor
    ~tempLaw()
    {}


    // Member Functions

        //- Return the laminar viscosity
        tmp<volScalarField> nu() const
        {
            return nu_;
        }

        //- Return the laminar viscosity for patch
        tmp<scalarField> nu(const label patchi) const
        {
            return nu_.boundaryField()[patchi];
        }

        //- Correct the laminar viscosity
        void correct()
        {
            nu_ = calcNu();
        }

        //- Read transportProperties dictionary
        bool read(const dictionary& viscosityProperties);
};


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

} // End namespace viscosityModels
} // End namespace Foam

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

#endif
```