OpenFOAM Course Final Assignment:
Tutorial for Natural Convection Boundary Layer

Abolfazl Shiri (751028-2051)

1 Introduction

The aim of this project is to simulate the natural convection boundary layer along a vertical cylinder with constant wall temperature which is hotter than the surrounding air. To eliminate the effect of surroundings, the cylinder in placed inside a long vertical tunnel. The size of the tunnel is bigger than the boundary layer thickness so we will have a zero velocity region far from the cylinder's surface. To ignore the effect of confinement and to avoid the air stratification inside the tunnel, there are inlets at the bottom of the tunnel and the air can be exhausted from top of the rig.
The application of this flow is in heat exchangers and reactors where we want to have a buoyancy driven flow over vertical cylinders and a controlled heat transfer to the surrounding fluid. There is a standard case studied in tutorial of OpenFOAM on heat transfer inside a 3-D cavity which we can use the solver and modify the geometry and boundary conditions to solve our case. The turorial case is called `hotRoom` and The steady state solver which is used in this tutorial is `buoyantSimpleFoam`.

2 Physics of the Flow

In this flow the effect of the forced convection on creating a bounday layer along the vertical cylinder is considered negligible. By this assumption, the only source of the flow generation is the temperature difference between cylinder's wall and the air inside the cavity. The temperature difference between the layers of air causes a density variation. Although the flow in this situation is incompressible, in order to consider this density variation we should use a compressible solver. We can use the Boussinesq approximation to solve the flow as an incompressible flow with density variation. The flow can be considered steady-state when the boundary conditions remain constant. The properties of air inside the tunnel is written in `constant/thermophysicalProperties` file and air is considered to be a perfect gas:

```
thermoType      hThermo<pureMixture<constTransport<specieThermo\
                         <hConstThermo<perfectGas>>>>>;
mixture         air 1 28.9 1000 0 1.8e-05 0.7;
pRef            1e5;
```

The `thermoType` which specifies the thermophysical model in this simulation, defines the equations of state. For this case a the keywords stand for:

- `perfectGas`: Perfect gas equation of state.
- `hConstThermo`: Constant specific heat C_p model with evaluation of enthalpy h and entropy s.
- `specieThermo`: Thermophysical properties of species, derived from C_p, h and s.

- `constTransport`: Constant transport properties.
- `pureMixture`: General thermophysical model calculation for passive gas mixtures.
- `hThermo`: General thermophysical model calculation based on enthalpy h.

The `mixture` reperesents the thermophysical property data for each species as below:

- $n_{moles} = 1$: Number of moles of the specie.
- $W = 28.9$: Molecular weight (kg/kmol).
- $C_p = 1000$: Heat capacity at constant pressure.
- $H_f = 0$: Heat of fusion (off).
- $\mu = 1.8e - 05$: dynamic viscosity.
- $Pr = 0.7$: Prandtl number.

`pRef` is the reference pressure and set to $100kpa$.

3 Governing Equations

To solve the Navier Stokes equation in a steady state buoyancy driven flow condition we chose the solver code `buoyantSimpleFoam`. Source code can be found in this address:
`$FOAM_APP/solvers/heatTransfer/buoyantSimpleFoam`
The source codes and information about the solver along with a description of each class and its functions can be found in Doxygen documentation at:
`$WM_PROJECT_DIR/doc/Doxygen/html/index.html`.

This solver is a steady-state solver for buoyant, turbulent flow. The `buoyantSimpleFoam.C` consists of a `runTime` loop which recalls the equations `UEqn.H`, `hEqn.H` and `pEqn.H` in every iteration and uses the pressure-velocity `SIMPLE` corrector.
In `createFields.H` the density field is read from `basicThermo` thermophysical properties. The velocity field is set from `startTime` directory and compressible `phi` is calculated using `compressibleCreatePhi.H`. A turbulence model is created by defining a new class from `compressible::RASModel` consisting of `rho, U, phi, thermo()`. The field `g.h` is created using the gravitational acceleration (`g`) which is defined in `environmentalProperties` and the mesh specifications. Next the dynamics pressure (`pd`) is set from `startTime` and the total pressure is defined as: `p = pd + rho*gh + pRef`
in which `pRef` is a reference pressure set in the initial conditions.

The general equations used in this solver are listed below:

- **Momentum Equations**
  The solver for the momentum equation is `UEqn.H` in the same directory.

```
00001     // Solve the Momentum equation
00002
00003     tmp<fvVectorMatrix> UEqn
00004     (
00005         fvm::div(phi, U)
00006       - fvm::Sp(fvc::div(phi), U)
```

```
00007         + turbulence->divDevRhoReff(U)
00008       );
00009
00010       UEqn().relax();
00011
00012       eqnResidual = solve
00013       (
00014           UEqn() == -fvc::grad(pd) - fvc::grad(rho)*gh
00015       ).initialResidual();
00016
00017       maxResidual = max(eqnResidual, maxResidual);
```

The `UEqn.H` code solves the differential equation below:

$$\nabla.(\Phi U) - (\nabla.\Phi)U - \nabla.\mu_{eff}\nabla U - \nabla.(\mu_{eff}(\nabla U)T) = -\nabla pd - (\nabla\rho)gh$$

The `divDevRhoReff()` is the source term for the momentum equation and is defined as below in the `compressible/RASModel.H`:

```
00209 tmp<fvVectorMatrix> kEpsilon::divDevRhoReff(volVectorField& U)\
const
00210 {
00211     return
00212     (
00213       - fvm::laplacian(muEff(), U)
00214       - fvc::div(muEff()*dev2(fvc::grad(U)().T()))
00215     );
00216 }
```

- **Energy Equations**
  The solver for this equation is `hEqn.H` in the same directory.

```
00001 {
00002     fvScalarMatrix hEqn
00003     (
00004         fvm::div(phi, h)
00005       - fvm::Sp(fvc::div(phi), h)
00006       - fvm::laplacian(turbulence->alphaEff(), h)
00007      ==
00008         fvc::div(phi/fvc::interpolate(rho)*fvc::interpolate(p))
00009       - p*fvc::div(phi/fvc::interpolate(rho))
00010     );
00011
00012     hEqn.relax();
00013
00014     eqnResidual = hEqn.solve().initialResidual();
00015     maxResidual = max(eqnResidual, maxResidual);
00016
00017     thermo->correct();
00018 }
```

The `hEqn.H` code solves the differential equation below:

$$\nabla.(\Phi h) - (\nabla.\Phi)h - \nabla.\alpha\nabla h = \nabla.\left(\frac{\Phi}{\rho p}\right) - p\nabla.\left(\frac{\Phi}{\rho}\right)$$

- **Dynamic Pressure Equations**
  The equation for Dynamic Pressure in steady-state compressible flow is solved in the file `pEqn.H` in the same directory.

```
00001 volScalarField rUA = 1.0/UEqn().A();
00002 U = rUA*UEqn().H();
00003 UEqn.clear();
00004 phi = fvc::interpolate(rho)*(fvc::interpolate(U) & mesh.Sf());
00005 bool closedVolume = adjustPhi(phi, U, p);
00006 phi -= fvc::interpolate(rho*gh*rUA)*fvc::snGrad(rho)*mesh.magSf()
00007
00008 for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
00009 {
00010     fvScalarMatrix pdEqn
00011     (
00012         fvm::laplacian(rho*rUA, pd) == fvc::div(phi)
00013     );
00014
00015     pdEqn.setReference(pdRefCell, pdRefValue);
00016     // retain the residual from the first iteration
00017     if (nonOrth == 0)
00018     {
00019         eqnResidual = pdEqn.solve().initialResidual();
00020         maxResidual = max(eqnResidual, maxResidual);
00021     }
00022     else
00023     {
00024         pdEqn.solve();
00025     }
00026
00027     if (nonOrth == nNonOrthCorr)
00028     {
00029         phi -= pdEqn.flux();
00030     }
00031 }
00032
00033 #include "continuityErrs.H"
00034
00035 // Explicitly relax pressure for momentum corrector
00036 pd.relax();
00037
00038 p = pd + rho*gh + pRef;
00039
00040 U -= rUA*(fvc::grad(pd) + fvc::grad(rho)*gh);
00041 U.correctBoundaryConditions();
00042
00043 // For closed-volume cases adjust the pressure and density levels
00044 // to obey overall mass continuity
00045 if (closedVolume)
00046 {
00047     p += (initialMass - fvc::domainIntegrate(thermo->psi()*p))
00048         /fvc::domainIntegrate(thermo->psi());
```

```
00049 }
00050
00051 rho = thermo->rho();
00052 rho.relax();
00053 Info<< "rho max/min : " << max(rho).value() << " " << \
min(rho).value() << endl;
```

In this code, the flux is and velocity is calculated on the cell boundaries and the equation for the flux and pressure is solved as below:

$$\nabla.\rho(\texttt{rUA})\nabla pd = \nabla.\Phi$$

The residuals are calculated and then the pressure is calculated as $p = pd + rho * gh + pRef$. Finally the corrected velocity is calculated as:

$$U = \texttt{rUA} * (\nabla.pd + \nabla\rho * gh)$$

where $\texttt{rUA} = 1.0/\texttt{UEqn}().A()$

- **Continuity Equation**
  The continuity equation for steady-state compressible flow is

  $$\nabla.(\rho U) = 0$$

  The solver checks the mass continuity by using calculationg the continuity error. file `initContinuityErrs.H` declares and initialises the cumulative continuity error. The convergence condition is controlled by the condition `maxResidual < convergenceCriterion` in the file `convergenceCheck.H`.

- **Equation of State**
  The basic thermodynamic properties based on perfect gas assumption ($p = \rho RT$) are calculated using this code:
  `$FOAM_SRC/thermophysicalModels/basic/basicThermos/basicThermo.H` This code defines a class in which properties like pressure(p), temperature(T), dynamic viscosity(μ) and thermal diffusivity(α) are calculated as `volScalarField` based on the model defined in `thermophysicalProperties` file.

4 Geometry

In order to simplify the mesh generation an axisymmetric field has been considered. This can simulate a two dimensional flow field within a three dimensional geometry. A 45 degree section of the cylinder has been considered as it shown in Figure 1. Because the geometry is axisymmetric, we can consider that the gradient of mean values in tangential direction is negligible. To simplify the calculations and decrease the number of cells, we can just solve for a section of the flow and use a `cyclic` boundary condition to simulate the entire field. Even though that the `wedge` boundary condition is usually used for an axi-symmetric flow, I had problem implementing this B.C. because the inner pipe has changed the geometry in this case. The blocks used to make a wedge boundary condition should be also a wedge but I had to use a hexahedron block to create the surface of the pipe.
The height of the tunnel is $1.5m$ and it has a $0.6m$ radius. the cylinder inside the tunnel has $0.15m$ radius. The top surface of the tunnel is outlet and there is a vertical opening in the bottom of the tunnel with the height of $0.12m$ as inlet.

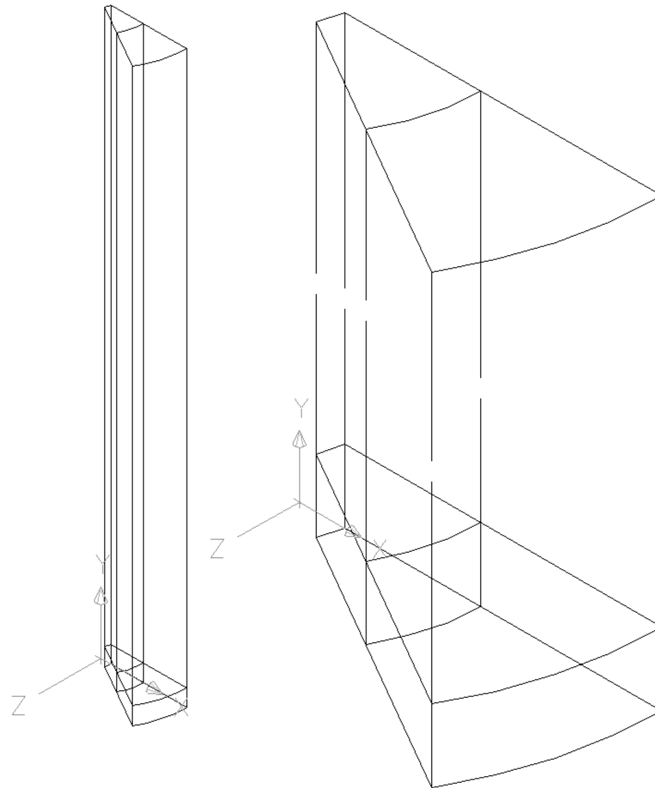

Figure 1: The computational domain and block topology.

5 Mesh Generation

The important part of the flow is the near wall regions. In order to have a graded mesh close to the walls we divided the field into four different blocks. We used `blockMesh` to mesh the field considering that the curved edges on the tunnel wall and the cylinder wall need to be defined. The `blockMeshDict` file is as below:

```
convertToMeters 0.001;
vertices
(
    (75 0 0)          // 0
    (262.5 0 0)       // 1
    (600 0 0)         // 2
    (75 120 0)        // 3
    (262.5 120 0)     // 4
    (600 120 0)       // 5
    (75 1500 0)       // 6
    (262.5 1500 0)    // 7
    (600 1500 0)      // 8
    (53.03301 0 53.03301)        // 9
    (185.61553 0 185.61553)      // 10
    (424.26407 0 424.26407)      // 11
    (53.03301 120 53.03301)      // 12
    (185.61553 120 185.61553)    // 13
    (424.26407 120 424.26407)    // 14
    (53.03301 1500 53.03301)     // 15
    (185.61553 1500 185.61553)   // 16
```

```
    (424.26407 1500 424.26407) // 17
);
blocks
(
    hex (0 9 10 1 3 12 13 4) (20 20 20) simpleGrading (1 3 2)
    hex (1 10 11 2 4 13 14 5) (20 20 20) simpleGrading (1 1 2)
    hex (3 12 13 4 6 15 16 7) (20 20 80) simpleGrading (1 3 1.5)
    hex (4 13 14 5 7 16 17 8) (20 20 80) simpleGrading (1 1 1.5)
);
edges
(
   arc 0 9 (69.29096 0 28.70126)
   arc 1 10 (242.51838 0 100.4544)
   arc 11 2 (554.32772 0 229.61006)
   arc 3 12 (69.29096 120 28.70126)
   arc 4 13 (242.51838 120 100.4544)
   arc 14 5 (554.32772 120 229.61006)
   arc 6 15 (69.29096 1500 28.70126)
   arc 7 16 (242.51838 1500 100.4544)
   arc 17 8 (554.32772 1500 229.61006)
);
patches
(
    wall pipeWall
    (
        (0 9 12 3)
(3 12 15 6)
    )
    wall tunnelWall
    (
        (5 8 17 14)
    )
    wall downWall
    (
        (0 1 10 9)
(1 2 11 10)
     )
    cyclic cyclicRightAndLeft
    (
        (0 3 4 1)
        (1 4 5 2)
        (3 6 7 4)
        (4 7 8 5)
     (9 10 13 12)
     (10 11 14 13)
     (12 13 16 15)
     (13 14 17 16)
    )
    patch inlet
    (
        (2 5 14 11)
```

```
        )
    patch outlet
    (
        (6 15 16 7)
        (7 16 17 8)
    )
);
mergePatchPairs
(
);
```

In the mesh there are four blocks with hexahedron meshes which are graded toward the wall surfaces to make a finer mesh along the walls.

6 Boundary Conditions

The temperature is set for the cylinder wall as $T = 360$ Kelvin and the rest of the walls have a constant $T = 290K$ ambient temperature. The inlet and outlet are open to the ambient air so the temperature and pressure for both are the ambient temperature and pressure.The turbulence properties(ϵ, k) are given a `zeroGradient` boundary condition at all boundaries.The velocities at walls are `fixedValue` with a zero value (`uniform (0 0 0)`). At the first attempt to run the case I used a `pressureInletVelocity` for the inlet velocity and `totalPressure` for the outlet velocity. But the code diverged with almost every inlet and outlet condition that I tried. So I decided to close the opennings and make the case as a cavity simulation. All the properties on the cyclic boundaries are give `cyclic` without any values.

7 Initialization

The temperature field initialization is a uniform temperature equal to the ambient temperature $T = 290K$. The velocity initialization is set to a uniform zero velocity field. The pressure is set to the reference pressure ($10^5 pa$). The turbulence properties are calculated based on the maximum fluctuating velocity expected in the field considering a turbulence intensity of 10% as below:

$$U_{max} = 0.5\ \frac{m}{s} \implies u' = 0.05\ \frac{m}{s} \implies k = \frac{3}{2}(0.05)^2 = 0.00375\ \frac{m^2}{s^2}$$

and if we estimate the turbulent length scale as the maximum thickness of boundary layer $\approx 0.05m$. we can estimate the dissipation as below:

$$\epsilon = \frac{C_\mu^{0.75} k^{1.5}}{l} = 0.00075\ \frac{m^2}{s^3}$$

After solving the flow the value for epsilon in most of the flow field was calculated as order of 10^{-5} and the maximum value for epsilon was in a very narrow region close to the pipe wall in order of 0.1.

8 Solver Settings

The discretization schemes are set in the `fvScheme` dictionary as follow: `ddtScheme` should be `steadyState`; the `gradSchemes` is set to `linear` scheme. `divSchemes` are set to `Gauss upwind`. and finally the `laplacianSchemes` are selected as `linear`.
The selected RAS turbulence model is k-Epsilon model with the coefficients as below

```
    Cmu              0.09;
    C1               1.44;
    C2               1.92;
    C3               0.85;
    alphah           1;
    alphak           1;
    alphaEps         0.76923;
```

The `fvSolvers` file set as below:

```
 solvers
{
    pd PCG
    {
        preconditioner   DIC;
        tolerance        1e-08;
        relTol           0;
    };
    U PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    h PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    k PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    epsilon PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    R PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
}
SIMPLE
{
    nNonOrthogonalCorrectors 0;
```

```
    pdRefCell        0;
    pdRefValue       0;
}
relaxationFactors
{
    rho             1.0;
    pd              0.3;
    U               0.7;
    h               0.7;
    k               0.7;
    epsilon         0.7;
    R               0.7;
}
```

As it can be seen the velocity-pressure coupling method is the SIMPLE method and the under-relaxation of the solution is required since the problem is steady.
In the controlDict dictionary, the time step deltaT should be set to 1 to act as a counter. The endTime is set to a big number to allow the solution to converge to $(10^{-}5)$. The convergence can be monitored by the pyFoamPlotWatcher.py. We run this in a separate terminal window:

```
$ touch log
$ pyFoamPlotWatcher.py log
```

9 Results

The solver that we use in this simulation is buoyantSimpleFoam:

```
$ buoyantSimpleFoam >&log
```

Unfortunately I couldn't make the solution converge with any of the boundary conditions for inlet and outlet like inletOutlet or pressureInletVelocity. So I decided to close the opening and solve the flow as a cavity with two different wall temperature. The only change in the boundary conditions are: the velocity at inlet and outlet are fixedValue equal to zero and the pressure boundary condition is zeroGradient.

The solution first has done in a coarse mesh with 12800 cells and then mapped to a finer mesh with 48000 cells. To do the mapping we changed the startTime in the controlDict file of the fine mesh to the last iteration on the coarse mesh.

```
$ mapFields ../NC_coarse -consistent
```

The results of the temperature and velocity field in a cross-section of the flow-field are shown in Figures 2 to 4. The result of the simulation even with the fine mesh could not resolve the boundary layer region in order to be compared with the experimental data or other simulations.

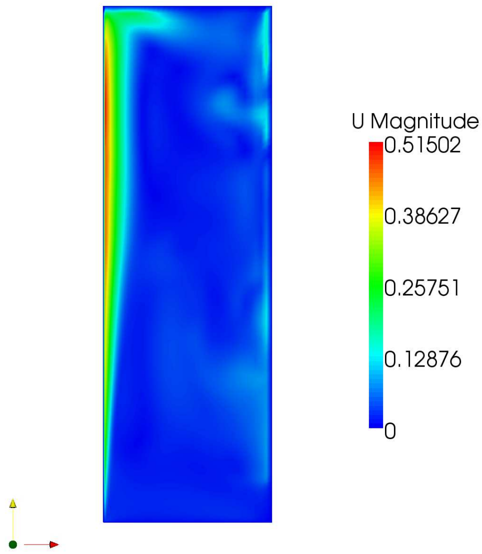

Figure 2: Velocity Field

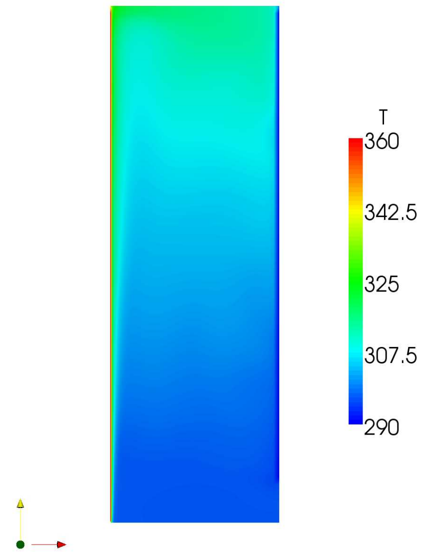

Figure 3: Temperature Field

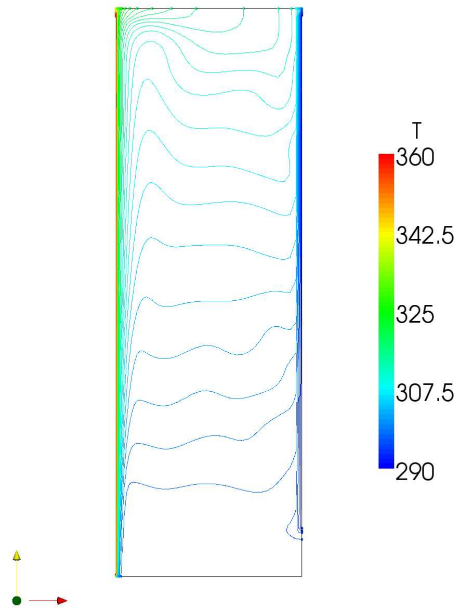

Figure 4: Temperature contour in a cross-section of the field.