Frédéric Collonval
3A Energetics and Propulsion

Computational Fluid Dynamics : Modeling of wall fires

Department of Fire Protection Engineering

Advisors :

Arnaud Trouvé – UMD, dept. of Fire Protection Engineering
Gilles Grondin – ISAE, dept. of Aerodynamics, Energetics & Propulsion

UNIVERSITY OF MARYLAND

2008-2009

Acknowledgment

This project could not be realized without Dr. Arnaud Trouvé who offers me this great opportunity to work on an original subject for me that is the fire modeling. His support was really helpful to have a better point of view to criticize my results despite the geographical constraint between us at the end of this internship.

This study would be more painful without the support and the availability of Dr. Yi Wang from FM Global. He helped me a lot by his suggestions and his answers to my numerous questions.

I cannot be complete without thanking the graduate students of the fire protection engineering with a special emphasis for Vivien Lecoustre, Meghan McKeever and Keenan Dotson. They were always ready to help for technical or linguistic troubles as well as to share some coffee to survive all working days and more.

Thank you to my parents who support me in all my choices, especially those one that bring me here in United States or earlier in France.

Table of Contents

1. Introduction

The following report presents my internship in the Department of Fire Protection Engineering (DFPE) of the University of Maryland. This work ends my education as engineer in Propulsion System at the “Institut Superieur de l’Aeronautique et de l’Espace” (Toulouse, France) as well as a Master of Science in Energy and Heat Transfer.

In collaboration with the insurance company, FM Global, the DFPE is developing a software, called fireFOAM, to model fire in buildings. This open-source software will be a new tool to improve the fire protection in industrial infrastructure. And for insurance companies, this tool will allow to evaluate with more accuracy the risk of fire in housing and so the terms of a fire insurance contract.

The OpenFOAM® (Open Field Operation And Manipulation) CFD Toolbox, developed by OpenCFD Ltd., is used as basis for fireFOAM. In particular, we focused on the boundary layer model. That is an important element in fire applications. Indeed, in the opposite of an engine where the reacting flow is mainly in the core of the combustion chamber, during a fire, the flame stays near the wall, the source of combustible material. That case is well known as a wall fire problem. So the most interesting part of the flow is compound of a boundary layer and a flame.

Currently, OpenFOAM includes a law of the wall to evaluate the shear stress and the heat flux. But they are not suitable for a temperature-driven flow. Consequently, a part of this work was to test a new law of the wall for the energy equation.

In addition with that specific objective, the DFPE wants to increase its knowledge of the powerful toolbox OpenFOAM. Indeed, that tool would be more interesting than a software like FLUENT as it is open-source. So you can modify it or add some functions. And also, you don't have to write a solver from scratch for each new research. You can re-use the “basic” functions already developed by OpenCFD or others sources.

This report is divided into five parts : an explanation of the environment of the internship, the different approaches to model a fire, a review of the tools used and created, the boundary theories and the test cases and their results.

2. The main contributors

In this first chapter, the major contributors bound with this internship will be introduced.

2.1. The University of Maryland [1]

In 1859, the Maryland Agriculture College opened its doors. That first year 34 students were enrolled and the only option was Agriculture.

But in its first years, the college had financial problems and is bankrupt during the secession war. It finally reopened in 1867 and increased the number of student to about 100.

The importance of the college began to be developed at the begin of the 20th century thanks to a series of state laws that gave college many powers (control of farm disease, state geological survey, board of forestry,etc.).

In 1916 the Maryland state took full control of the college and changed its name to Maryland State College. The next year, the president, Albert F. Woods divided the college into seven schools : Agriculture, Engineering, Arts and Sciences, Chemistry, Education, Home Economics and Graduate School. Three years later, in 1920, to reinforce the links between the College Park (near Washington DC) and the Baltimore campus, the University of Maryland was created. Since that time, the number of students increased from 2,000 students in 1935 to more than 38,000 in 1985.

Today, the University of Maryland is, in some figures [2] :

- 5 campuses (the most important is the 500-hectares College Park campus)
- 37,000 students (in fall 2008)
- Nearly 19,000 employees
- 13 colleges and school, 127 undergraduate majors and 112 graduate degrees
- An annual budget of $ 1.5 billion
- 37th in the World's Top 100 Universities Ranking of 2008 (according the Institute of Higher Education – Jiao Tong University, Shanghai)

2.2. The Department of Fire Protection Engineering [3]

The Department of Fire Protection Engineering (DFPE) was created in 1956 as a part of the School of Engineering. And the first program opened in Fall 1957. This department appeared to answer the needs expressed by companies and public administration. Consequently from the beginning, there were strong links between the DFPE and professional associations or private firms (like FM Global).

Currently there are about 110 students in fire protection programs (80 undergraduate and 30 graduate students) and about 10 faculty members (professors, Ph. D. students,...). They all work on the different aspects of fire from the materiel testing practices to the modeling techniques to predict fire growth. All these studies are supported by laboratory facilities providing experience with "standardized" ASTM test procedures and all modern equipment to analyze reacting flows.

Private companies are involved in the educational program by two means. First they decide with the academic authority the content of the engineering formation. And, as it's common in the United States, they signed research contract with the department and help to develop the infrastructure.

FM Global is one of main partner of the DFPE. This company is specialized in insurance for industrial property [4]. But its job is more than insurance, they invest a lot of money in research to improve the understanding of the effects of natural hazards (wind, hail, flood, fire, earthquake,...). With this information, they can better assess risk and help industries to prevent and/or control property loss. And, as lots of their research are shared with the public, they contribute to advance the science.

Figure 1: Department of Fire Protection Engineering [3]

Among all the natural hazards, fire have a particular place in FM Global research. Indeed it exists a specific program called "The FM Global Fire Prevention Grant Program". That program allows fire departments and organizations to receive support to develop fire prevention and educational program.

As you understand, they use engineering analysis to support their customers and to better assess the risks. That policy is a big success as showed by the name of their customers (eg : Mattel, Inc. or Reckitt Benckiser Group plc – owner of Calgon®, Air Wick®, Strepsils®,...) [5].

After this introduction on the collaborators involved in this internship, it is time to go deeper on the subject : fire. Especially the different approaches to model fire in building will be explained in the following chapter.

3. Modeling fire [6][7]

A large number of softwares exist which model fires ranging from the modeling of forest fires to the evaluation of the time needing to evacuate a building. This section will focus on the modeling of fires in compartments/buildings thanks to the determinist theory. Indeed, probabilistic theories could also be used to model fires. The main difference with the determinist theories is that each scenarios of fire evolution is weighted with a probability of occurrence.

Basically there are two main ways to model fires in buildings : zone models or field models.

The first one evaluates the effects of a fire in an enclosed volume by dividing it into two uniform zones. On each zone, the proper conservation equations are applied (mass, momentum, energy,...). And the interactions between the zones are modeled using some source terms in those equations. For example, the classic case is a fire in the middle of a room. To solve this case, two zones are created : the upper zone contains the hot smoke and the lower zone contains the fresh air, as shown in Figure 2. The plume acts as a pump of mass between the cold lower layer and the hot upper layer. This model is not restricted to a single room. But you can use multiple rooms with windows, doors and ventilation to create a more realistic building.

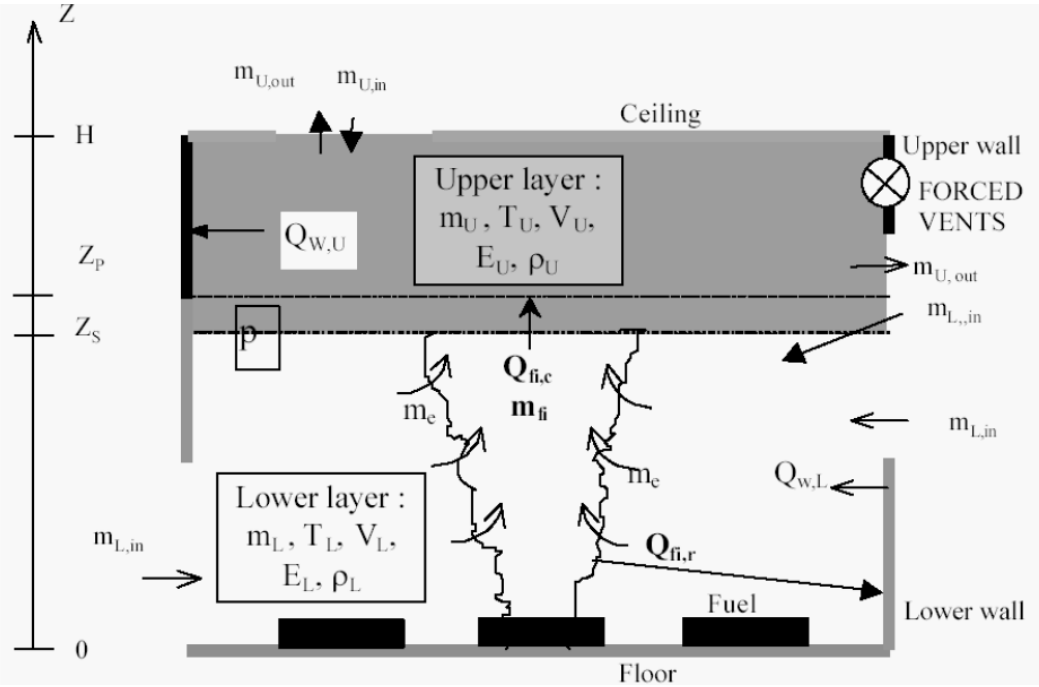

Figure 2: Division of a room in two zones [7]

It is obvious this theory is simple and doesn't take into account some phenomena like the interface between zones or the fact the plume can create non uniformity inside a zone. But despite this, zone models offer reasonable approximations for a large number of configurations. This model allows you to estimate an important amount of information like the CO or CO_2 concentration, the mass flux at the interfaces between rooms or the power of the fire.

A great number of software using zone models have been developed. One of the famous among them is CFAST. This free software was developed by the National Institute of Standards and Technology (NIST) of the United States Department of Commerce [8].

The Consolidated Model of Fire and Smoke Transport, CFAST, is a two-zone fire model used to calculate the evolving distribution of smoke, fire gases and temperature throughout compartments of a building during a fire. Engineers, architects or officials could use it to evaluate the impact of fires and smoke in buildings.

The other approach, the field model, divides the area of interest into smaller volumes. The number of subdivisions are huge as this theory uses the conservation equations using the finite volume model. That implies a more accurate solution. However, this model requires more work (properly defining the properties and the boundary conditions) and resources (memory, time CPU,...). Thanks to the continuous improvement of computer science, the use of this model has increased. The possibility to analyze more complex geometries is also a reason of its growing success.

Many field models exist today and most of them recently appeared due to the power of recent computers. In this category, two kinds of software are available. The first one are general CFD softwares like CFX, FLUENT or OpenFOAM or you can use dedicated solutions like FDS or SOFIE.

Among all those possibilities, FDS is the most employed in fire protection engineering. It's a free software developed by NIST in cooperation with VTT Technical Research Center of Finland. Its computational fluid dynamics model is dedicated to the fire-driven flows. So the Navier-Stokes equations solved are written for low-speed and thermally-driven flow and the smoke and heat transport are carefully modeled to fit the fire configurations.

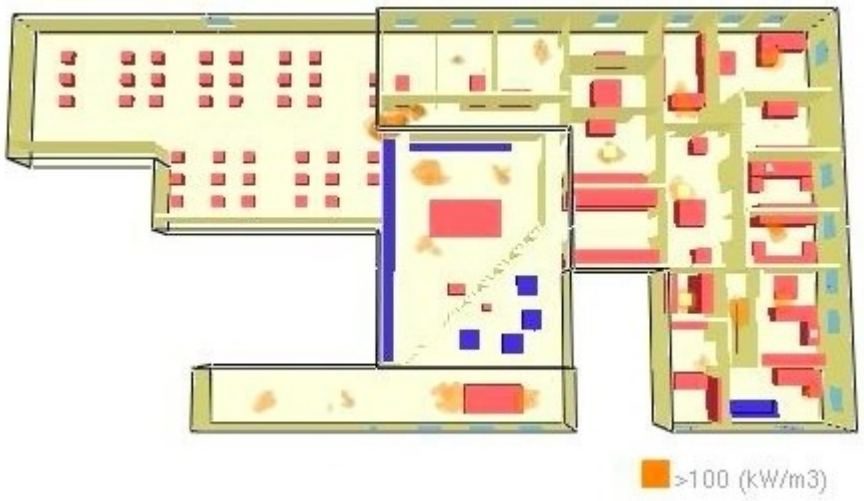

Figure 3: Simulation of a fire in the DFPE using FDS [9]

Two other types of software are also linked with fire applications. The first is focused on the detector response models and estimates the time of activation for devices like sprinklers and the response of the devices. Finally some solutions, using egress models, predict the time people have to evacuate a building. Most of them used zones models to determine the onset of untenable conditions in the rooms.

As you can see, the number of existing softwares is equal to the number of physical problems that occur in a fire. And the complexity of the solution come from a simple program written by a student for his thesis to a big commercial software like FLUENT. So, before describing the relevant parts of OpenFOAM, here is a short discussion about why this toolbox was chosen among the others possibilities.

First, OpenFOAM is an open-source toolbox. Consequently, it is free. You have access to the code and you can modify it. For an academic use, this is a great advantage in comparison with a solution like FLUENT that acts as a black box. Another advantage is the availability of a huge number of basic functions to handle the data, solve equations or create visualizations. Indeed, the cost of developing a new solution is reduced if you don't have to reinvent the wheel.

But, more than that, OpenFOAM is not an open-source solution maintain by some anonymous people on a website. It is developed and maintained by a company, OpenCFD Limited. This is an important point because you can obtain support and training from the developers and by creating partnership with OpenCFD Ltd, you can request specific developments for the code. Finally, as the core of the system is made by professional people in computer science, you can be confident on the performance of the solvers. This is really important for a CFD software.

In the future, as the number of users will increase, the number of new solvers, pre-/post-processors and utilities will grow rapidly. So to create a particular solution, you will have just to put all the bricks you want together, like a Lego. And your job will be focused on your problem and not too much on the tools.

4. The tools

After a small introduction on the main solution to model fires, this section will focus on the open-source toolbox OpenFOAM, its add-on fireFoam and the tools developed during this internship. As explain previously, the solution chosen here to model fire is a field model. But OpenFOAM is a general CFD software with few abilities to simulate low-speed, temperature-driven fluid flows. Consequently FM Global decided to develop an add-on called fireFoam dedicated to fire modeling.

Preliminary remark : All the code shown in this report referred to OpenFOAM version 1.6. At the beginning of this internship the version 1.5 was used. An upgrade to the development version was performed and the latest version 1.6 was introduced last month. The purpose of those upgrades was to keep a coherence between FM Global and the university.

4.1. OpenFOAM

As described on the website of OpenCFD [10], OpenFOAM is a powerful toolbox written in C++ to solve general CFD problems by using finite volume numerics. It allows to solve 3D unstructured mesh with an implicit, pressure-velocity, iterative process. In addition, you can easily use parallel computation because the decomposition models are taken into account in the very basic functions of the libraries.

As an open toolbox, you can modify and create whatever you want from new utilities to handle data to financial solvers. Since you can use this solution for a lot of problems, I would like to present a bit of the main structure of the code for a compressible LES solver using in this project[1].

4.1.1. Typical case

Before presenting the solver, the structure of the typical test case for this project will be introduced.

To create a case, you have to define a series of files called "*dictionaries*". In those dictionaries, you specify the mesh, the numerical schemes, the boundary conditions, the environment properties and all the other parameters needed to solve the case. The structure of the folder have to respect the template presented in Figure 4.

Let's now have a more detailed look at each files[2].

```
<case>
      |- system
      |     |- controlDict
      |     |- fvSchemes
      |     |- fvSolution
      |     |- …
      |- constant
      |     |- fireFoamProperties
      |     |- LESProperties
      |     |- thermophysicalProperties
      |     |- …
      |     |- polyMesh
      |           |- blockMeshDict
      |           |- points
      |           |- neighbour
      |           |- owner
      |           |- faces
      |           |- boundary
      |- <time step folders>
```

Figure 4: Structure of a case

1 Most of the information presented in the following paragraph come from the User Guide of OpenFOAM [11].
2 The file needed for a complete case using fireFOAM-1.6 with OpenFOAM-1.6 is described in the Appendix D.

4.1.1.1. system

The system folder contains the definition of all the parameters needed to tweak the solver and resolve the case :

- *controlDict* :
  - You can find the solver used (for us, fireFoam), the time parameters (start time, end time, time interval, writing interval, flag to adjust the time step or not, constraints on the Courant number and on the maximal time step,...).
- *fvSchemes* :
  - This file regroups the numerical schemes used to discretize the equations. Here is a piece of such file :

```
divSchemes
{
      default     none;
      div(phi,U)  Gauss filteredLinear 0.2 0.05;
      div(phi,h)  Gauss limitedLinear 1.0;
}
laplacianSchemes
{
      default     Gauss linear corrected;
      laplacian(muEff,U)      Gauss linear corrected;
}
```

  - This example show you how the syntax of the code is clear. And as the toolbox contains already lots of schemes, you change the scheme just by changing the name in the corresponding line.
- *fvSolution* :
  - The commands written in this file define the solvers and their parameters used to solve the different equations. Here is a short example of the code :

```
solvers
{
      rho
      {
            solver            PCG;
            preconditioner    DIC;
            tolerance         0;
            relTol            0;
      }
      U
      {
            solver            smoothSolver;
            smoother          GaussSeidel;
            tolerance         1e-8;
            relTol            0.0;
            nSweeps           1;
      }
}
```

In the folder "*system*", you could find other dictionaries like "*decomposeParDict*" or "*sampleDict*". Those are linked to utilities (for example, "*decomposeParDict*" define how to divide the volume to solve it in parallel).

4.1.1.2. constant

This second folder regroups all the properties of the flow and also the mesh. In the dictionary like "*fireFoamProperties*", you will find for example the definition of the reference pressure; in "*LESProperties*", the LES model used and its coefficients; in "*thermophysicalProperties*", the model used (Sutherland model, perfect gas hypothesis,...) and the associated coefficient (for example the Janaf coefficient) for all the species present in your case.

In this folder, another important component is included : the mesh. As all advanced CFD softwares, there are two possibilities to create a mesh. The first solution is to import it from a powerful mesher. The other solution is to use the more simple tool "*blockMesh*" provided with OpenFOAM, which was used during this study. Consequently, this utility will be described more in the following paragraph.

As for all applications of OpenFOAM, you customize your geometry thanks to a dictionary, in this case called "*blockMeshDict*". Here is the typical structure :

```
convertToMeters 0.1;
vertices
(
      (0 -1 -1)
      (3 -1 -1)
      …
);

blocks
(
      hex (0 1 2 3 4 5 6 7) (120 40 40) simpleGrading (2 1 1)
);

patches
(
      wall wallHot
      (
            (0 4 7 3)
      )
      patch up
      (
            (4 5 6 7)
      )
      cyclic side
      (
            (0 1 5 4)
            (7 6 2 3)
      )
      …
);
```

The first parameter *convertToMeters* defines a global scaling factor. This factor will be applied to all the coordinates in the list of *vertices* defined just after. The first step is the definition of the *blocks*. For that purpose, you have to give the shape of the volume (e.g. : *hex* for hexahedron), the index[3] of the shape corners and the number of cells along each coordinate direction. You can also specify a grading in the mesh (e.g. : *simpleGrading* allows you to create a geometric progression in the cell size for the given coordinate direction). In the final step the boundaries of the system will be defined. For that you write first the type of the boundary (*patch* for general boundary, *wall* for wall boundary, *cyclic* to bound two faces with periodic boundary conditions,...). Then a name is given to each boundaries. Finally, you define the faces that constitute the boundary by given the index of its corners.

As soon as the dictionary is written, you can run the "*blockMesh*" application. The results will be five files. Here is a short description of those files :

• *points* : contains the coordinates of all points.

• *faces* : contains a list of faces. Each face is represented by its number of summits and the index of them.

• *owner* : contains a list of owner cell labels. That means that the first entry provides the owner cell label of the face 0. The second entry is the owner cell label of the face 1, etc.

• *neighbour* : contains a list of neighbour cell labels. That means that the first entry is the neighbour cell label of face 0. The second entry refers to the face 1, etc.

• *boundary* : contains a list of patches described thanks to a dictionary entry.

Now, the solver, the properties and the mesh are determined. But, one important part is still missing : the initial conditions and boundary conditions.

4.1.1.3. Time step folders

In OpenFOAM, the value of all computed fields are stored at each written interval in a folder named according to the current time. So for example, if you run a case of 0.5 s with a written interval of 0.1s, the case directory contains 6 *"time step folders"* (0, 0.1, 0.2, 0.3, 0.4 and 0.5). Consequently, to create your initial conditions and boundary conditions, you have to define all computed fields in the folder *0*.

Typically, in that folder, you will find a file for each field you need/want to compute(a file "*U*" for the velocity, "*p*" for the pressure,etc.). The content of each file corresponds to a dictionary. For example, here is an extract from a "*U*" file.

3 The index is determined by the order in the list of vertices. For our example, the point 0 is (0 -1 -1). And the point 1 is (3 -1 -1).

```
FoamFile
{
      …
      class       volVectorField;
      object      U;
}

dimensions        [0 1 -1 0 0 0 0];
internalField     uniform (0 0 0);

boundaryField
{
      wallHot
      {
            type        fixedValue;
            value       uniform (0 0 0);
      }
      up
      {
            type        pressureInletOutletVelocity;
            phi         phi;
            value       uniform (0 0 0);
      }
      side
      {
            type        cyclic;
      }
      …
}
```

At the begin of the file, the type of the field is specified as well as the name of it (here *volVectorField* for a volumic field of vectors called *U*). Then the dimensions of the field is given. The representation used for the dimensions is an array containing the exponent of each fundamental unit[4] (e.g.: [0 1 -1 0 0 0 0] for kg^0 m^1 s^{-1} K^0 $kgmol^0$ A^0 cd^0 or m/s). The next line defines the value of the field in the internal part of the mesh. Finally the list of boundary conditions are given. For each boundary, the name of the patch comes first (e.g.: *wallHot*), then the type of the boundary conditions (e.g.: *fixedValue*) and finally the parameters needed by the chosen type of boundary.

4 The index of the array refers respectively to the mass, the length, the time, the temperature, the quantity, the current and the luminous intensity.

4.1.2. Utilities

To help you treat the data, OpenFOAM comes with utilities to pre-/post-process them. In this section, the major ones will be described.

4.1.2.1. ParaFoam

When you have finished the dictionaries defining your case, you can move to the resolution of it. However, OpenFOAM works exclusively with text files. So if you want to visualize the results, you have to use a third party software. But to help people, OpenFOAM comes with a tool, called "*paraFoam*". This application properly formats the data before sending them to ParaView®[5][12]. Thanks to that open-source application, analyzing and visualizing data are simplified.

4.1.2.2. FoamLog

When you run a case, by default, information are written on the terminal or in a file if you redirect the data flux, commonly called the *log* file. Among those data are the residuals, the number of iterations, the Courant number or some personal control parameters. But most of the time, you would like to monitor the evolution of those parameters and not just the values. For that, you have two choices, you can run a linux script using the "*gnuplot*" application[6]. Or you can use "*foamLog*". This last solution, extract from the log file all the possible variables. Then each variables are written in a file with the first column for the time and the second for the variable[7].

4.1.2.3. Parallel resolution

Today, all new computers have more than one core. As a matter of fact, the computer mainly used during this internship had an Intel® Core™ 2 Duo CPU. But if you don't specify that to the solver, it will carry out the simulation on a single core. Hopefully, running a case on multiprocessor's computer is easy with OpenFOAM. For that, you have to follow those steps :

1. Define the decomposition of the case in a dictionary "*system\decomposeParDict*". The easiest way is to use the *simple* method. For that method, you have to precise the number of subdomains and how you divide the global volume. For example, in my case, 2 processors are used and the domain is cut in the z-direction, the dictionary looks like :

```
numberOfSubdomains 2;
method      simple;
simpleCoeffs
{
   n      (1 1 2);
   delta 0.001;
}
distributed       no;
roots();
```

2. Then you run the application "*decomposePar*" in the case directory.

5 ParaView® is developed by Kitware, Inc. in association with Sandia National Laboratories, Advanced Simulation and Computing (ASC), Los Alamos National Lab and U.S. Army Research Laboratory.

6 An example is shown in the Appendix C.

7 For more information on the command *foamLog*, type `foamLog -h` in a terminal.

3. Finally the application is running in parallel thanks to the following command :

   ```
   mpirun -np 2 fireFoam -parallel > log
   ```

   This command will run the application *fireFoam* on the case contained in the current directory using two processors. The option *parallel* write the output data in the distinct directory *processorN*. And the output log is written on a file called *log* instead of the terminal.

To analyze the global domain, you could use the application *reconstructPar*. That tool reads the content of the *processorN* directories and reconstruct the fields for the global domain.

4.1.2.4. Function objects

Another interesting tool is the function object. That process run various utilities after each time step. For example, you could monitor the evolution of the average of a field thanks to the function object : *fieldAverage*. For that you have to add the following lines to in the dictionary "*controlDict*".

```
functions
(
   fieldAverage1
   {
       // Type of functionObject
       type fieldAverage;

       // Where to load it from (if not already in solver)
       functionObjectLibs ("libfieldFunctionObjects.so");

       // Optional parameters
       enabled true; // true or false – enabled the function or not
       log     true; // true or false - print or not information in the log file

       // Output control : outputTime or timeStep
       outputControl       outputTime;
       outputInterval      1;

       // Fields to be probed. runTime modifiable!
       fields
       (
           U
           {
               mean            on;
               prime2Mean      on;
               base            time;
           }
       );
   }
);
```

Others functions are available (cf. `$FOAM_SRC/postProcessing/functionObjects`) within OpenFOAM. And some specific functions were developed during this project (cf. paragraph 4.3.2. p.22).

4.2. FireFoam

As mentioned previously, OpenFoam comes with a lot of applications. However, none of them are perfectly suitable for FM Global purpose. So they decided to create their own solution, *fireFoam*. Below this paragraph, the structure of that solver will be presented.

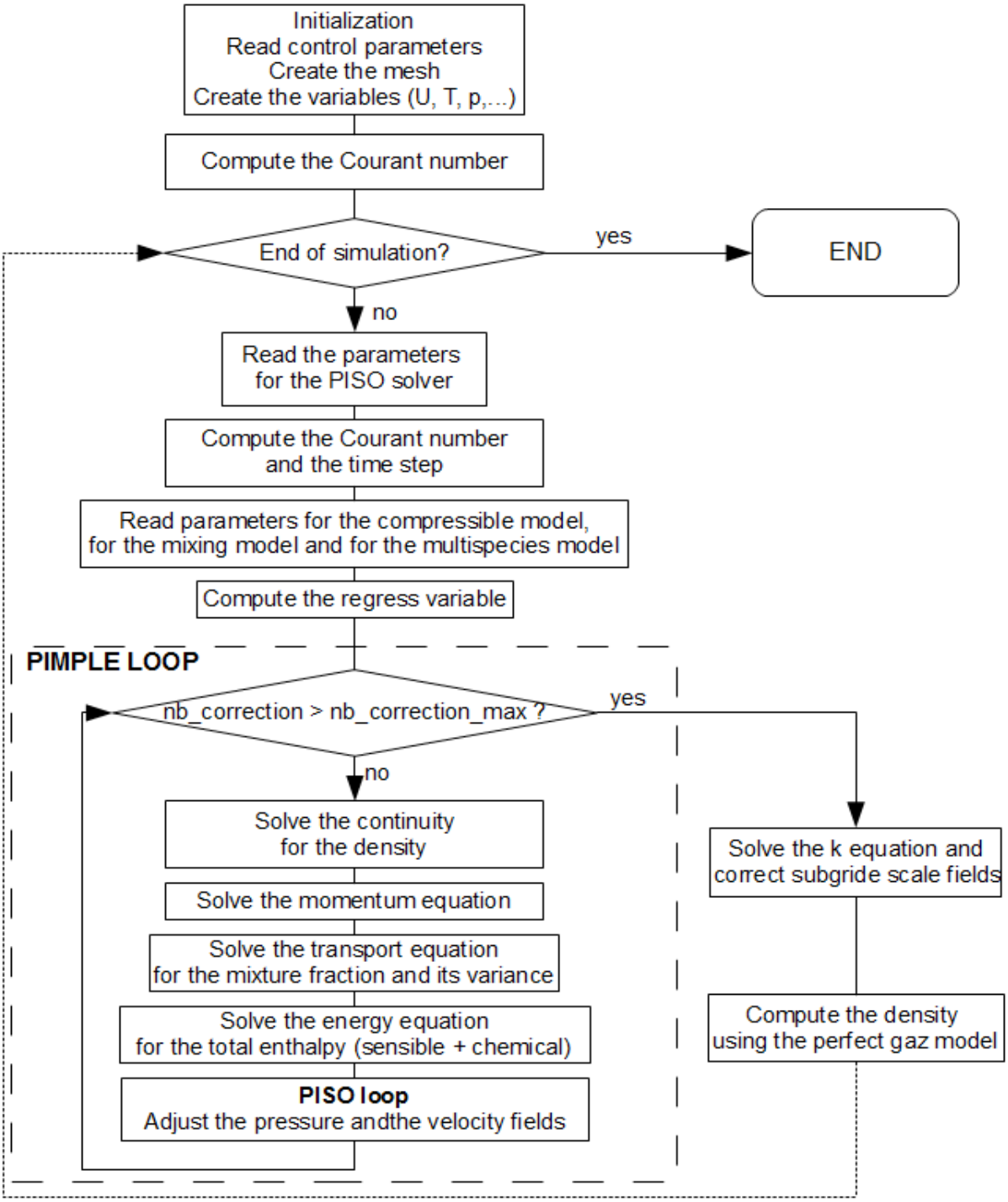

Figure 5: Structure of fireFoam-1.6 release 08/05/09

4.2.1. The solver[8]

Figure 5 describes the main structure of the application fireFoam. First the program read the geometry and all the physical properties. Then all the fields useful for the simulation are created. The last initialization step is the computation of the Courant number.

The second step is the time loop that could be divided into three parts. Foremost the time step is computed as well as the parameters needed by the solvers. A final computation is made before the main resolution. It is the computation of the regress variable. This variable is used in premixed combustion changing from 1 to 0. One represents the non reacting flow and zero the fully combustion. FireFoam uses that variable to create a ramp from a non-reacting case to a reacting case. Then the second part, the core of the application, is the resolution loop using the PIMPLE algorithm[9]. In that loop all the conservative equations are solved by beginning with the continuity equation. A first estimation of the velocity field follows before the fraction-mixture equation, and then the energy equation is solved. The final step of the resolution is the PISO loop that correct the velocity and pressure fields.

The algorithm is an iterative procedures for solving equations for velocity and pressure. PISO is used for transient problems and SIMPLE for steady-state. In fireFoam, a combination of them are employed to solve the two configurations. Both begin by evaluating some initial solutions and correct them. Typically PISO requires more than 1 correction but less than 5 when SIMPLE only makes 1 correction. Therefore in the tests case during this project, the *nb_correction_max* was equal to one, but 2 corrections were made for the PISO loop.

To set a case with fireFoam 11 fields have to be defined (the velocity, the temperature, the pressure, the dynamic pressure, the default conditions for the species, the turbulent kinetic energy, the mixture fraction and its variance, the regress variable, the dynamic viscosity and the thermal diffusivity for the subgride scale). But more fields than that are solved. Consequently some others fields are stored at each writing interval. Those additional fields are the mass fraction for the fuel, the oxygen and the burnt product, the convective heat flux, the density, the mass flux, the heat release rate and a field based on the mixture fraction estimating the flame height.

4.2.2. The additional content

In addition with the solver itself, FM Global develops other tools and models for OpenFOAM.

In fact, there is only a combustion model for premixed flame in OpenFOAM currently. However, the fire flames are diffusion flames and not premixed flames. Indeed, the “fuel” comes for the pyrolysis of the wall and the environment provides the oxygen. Consequently, a first simple diffusion model is implemented in fireFoam. This model uses the mixture fraction and its variance to evaluate the composition of the mixture thank to a probability density function. But an evolution to an eddy break-up model is planned.

Different utilities are developed to compute the integral of a field on a surface (for example to check the fuel mass flow). New boundary conditions for the enthalpy or for the fuel mass fraction come also with fireFoam.

8 A description of the code source is available in the Appendix A.

9 PIMPLE is a merge between PISO (Pressure-Implicit Split-Operator) and SIMPLE (Semi-Implicit Method for Pressure-Linked Equations).

4.3. Post-processing tools

When you test a software, you have to extract the pertinent information. That often implies to create small utilities to transform the output data. It is also a good way to improve your knowledge on the software. Indeed OpenFOAM comes with a huge number of functions. And, as the heritage and the template classes are used intensively, find the available functions and understand the code is a hard job. But by using an integrated development environment (IDE) such as Eclipse or NetBeans, the navigation inside the code is easier. So a tutorial to take advantages of those IDE's is presented in the Appendix C.

4.3.1. Post processors

Two post-processors were created. The first one averaged the value of the scalar field in a specified direction. Like in our case, the field can be averaged in the homogeneous direction along the wall to increase the accuracy of values like the heat flux. The second post-processor is linked to the evolution of the flow generated by hot vertical wall. In fact, this configuration goes to a steady-state. The maximum of the vertical component of the velocity or the integral of the turbulent kinetic energy can be use to monitor the convergence to steady-state. So a tool was designed to monitor those parameters.

4.3.1.1. Average in a homogeneous direction

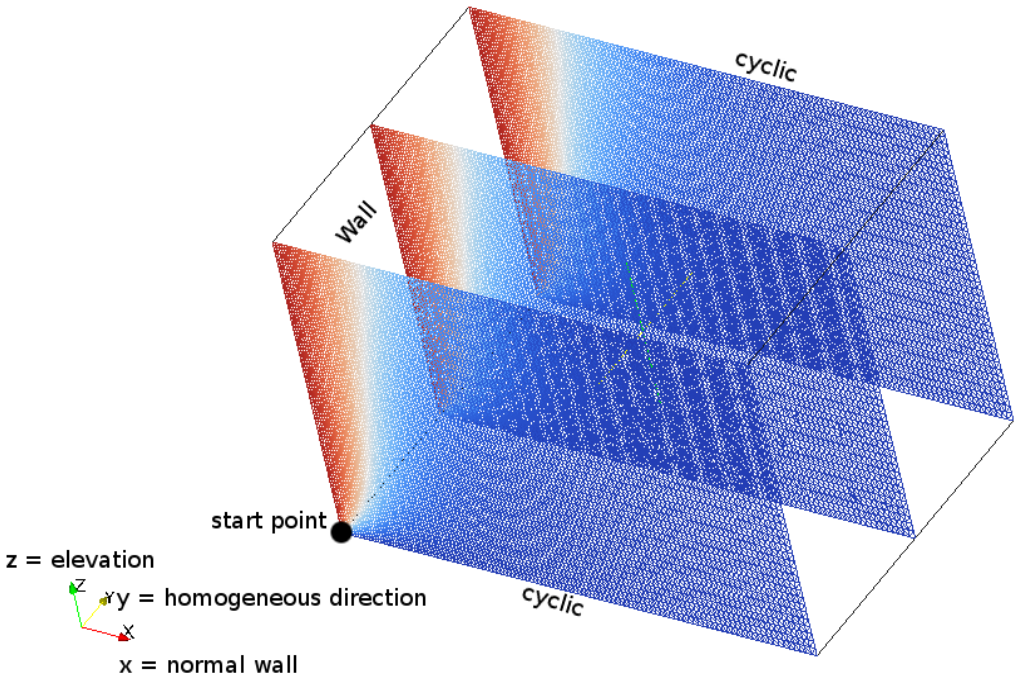

Figure 6: Average in a homogeneous direction - explanation

This tool is composed of three files[10] and a dictionary :

- *profileWall.C* contains the main function
- *wallIndex.H* is the header file of the class *wallIndex* used to compute the average field
- *wallIndex.C* contains the functions of the class *wallIndex*
- *profileWallDict* is the dictionary containing the parameters for this tool. This file have to be present in the directory *constant* of the present case

Figure 6 shows the typical case in which this tool is applied. In this example, a vertical hot wall is model with a 3D domain. The objective is to compute the mean profile of temperature along the x-direction at different heights z_i.

In order to achieve this, here is the procedure that had to be followed :

1. Define the dictionary "*profileWallDict*" in the directory "*constant*" of the current case
2. Run the function by typing in a terminal (the parameter case is optional) :
   ```
   profileWall [-case <directory of the studied case>]
   ```
3. The results are written in the directory :
   ```
   <lastTime step>\profiles
   ```

So the key point is the definition of the dictionary *"profileWallDict"*. Here is a description of all parameters followed by an example.

Parameter	Description
`Nx`	Number of cells in the x direction
`Ny`	Number of cells in the y direction
`Nz`	Number of cells in the z direction
`startPoint`	First point near the wall that defines the first studied elevation and the first value of the field near the wall
`normalWall`	(optional) define the direction normal to the wall (0 = x, 1 = y (default), 2 = z)
`elevation`	(optional) define the direction normal to the studied profile (0 = x, 1 = y, 2 = z (default))
`fields`	List of studied fields
`heights`	Heights where you want the profiles (put in the ascendant order !!!)

Table 1: Description of the dictionary profileWallDict

10 The source code of those three files is available in the Appendix C.

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.0                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
      version     2.0;
      format      ascii;
      root        "";
      case        "";
      instance    "";
      local       "";
      class       dictionary;
      object      profileWallDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Nx    40;
Ny    50;
Nz    30;
startPoint  (0    0     0);
//normalWall      1;
//elevation       2;
fields
(
      Ux
      T
);
heights
(
      -0.1
      0.0
);
// ************************************************************************* //
```

The results of this dictionary will be the following :

```
<lastTime step>/profiles
                        |-    -0.1
                        |     |- Uz.xy
                        |     |- T.xy
                        |-    0.0
                              |- Uz.xy
                              |- T.xy
```

A folder for each height will be created. And each folder will contain a file for each field. In those file, the first column represents the coordinate normal to the wall (x in the example). And the second one is the value of the field.

Some remarks have to be made on this tool :

- It was developed for the OpenFOAM version 1.6.
- It works only on a structured grid with regular hexadron cells
- The tool read the value of the field at the latest time step only.
- The field has to be a scalar. But for this last point, you can take advantage of the *foamCalc* tool that is able to decompose a non-scalar field in its different components (e.g. to extract the vertical component of the velocity, the command is `foamCalc components U`).

Another solution more powerful was created later. It is described in the paragraph 4.3.3. p.24.

4.3.1.2. Control parameters

The second utility developed is called "*diagnostic*"[11]. It reads the velocity and the turbulent kinetic energy fields at each output interval. From the first field, the maximum of the vertical component of the velocity is extracted. And with the second one, the integral of the turbulent kinetic energy on the domain is computed. In the final step, two files are created : *UMax.xy* and *kTot.xy*. They contained the temporal evolution of the two control parameters; respectively the maximum of the vertical component of the velocity[12] and the integral of the turbulent kinetic energy.

To run this tool, write the following command in a terminal :

```
diagnostic [-case <path to the case to analyze>]
```

4.3.2. Function objects

The main disadvantage of the post-processors described above is they use only the field values at the output interval. Consequently a lot of information is missing especially for the evolution of the control parameters. In fact, it is like using a temporal filter before applying the utility. So the high-frequency variations are not captured.

To avoid this, OpenFOAM allows for customized functions that are executed at each time step to be created. Those functions are called "*functionObjects*" and are defined in the "*system\controlDict*" dictionary.

4.3.2.1. Control parameters

By definition a control parameter follows the evolution of a simulation. It is clear that the function described in the paragraph 4.3.1.2. would be better as a *functionObject* than a post-processor[13].

To execute this function, you have to add the following lines in the dictionary "*controlDict*".

11 Its source code is available in the Appendix C.
12 Here vertical means the *Z* component of the velocity vector.
13 Its source code is available in the Appendix C.

```
functions
(
      diagnostic1 // name of the function
      {
            // general options
            type              diagnostic;       // type of the function
            functionObjectLibs      ("libuserFunctionObjects.so"); // library
            log               true;             // true or false
            enabled           true;             // true or false
            outputControl     timeStep;         // timeStep or outputTime
            outputInterval    1;

            // specific options
            // none for this function
      }
);
```

If the flag *log* is true, additional information is printed in the log file (or on the output screen if no log file are defined). In this case, the current value of the parameters are written.

This function creates a file *diagnostic.dat* in the folder :

```
<case root path>\<name of the function object>\<first time step>\
```

This file contains 3 columns, the first is the time step, the second is the maximum of the vertical velocity[14] and the third is the integral of the turbulent kinetic energy.

That function works on the version OpenFOAM-1.6 but not for parallel running.

4.3.2.2. Heat flux and Nusselt number

In natural convection problem, one of the key parameter to compare simulations, theories and experimental data is the heat flux or the Nusselt number. The other function created during this project focuses on the computation of those parameters[15].

The heat flux is computed using the following equation :

$$q_w = c_{p\,w}\,\rho_w\,a_{eff\,w}\left.\frac{\partial T}{\partial n}\right|_w \tag{1}$$

Where c_{pw} is the specific heat capacity at the wall [J/kg/K], ρ_w, the density at the wall [kg/m^3], $a_{eff\,w}$, the effective thermal diffusivity at the wall [m^2/s] and $\left.\frac{\partial T}{\partial n}\right|_w$, the normal gradient of the temperature [K/m].

The Nusselt number is given by :

$$Nu = \frac{Z\,\frac{a_{eff\,w}}{a_w}\left.\frac{\partial T}{\partial n}\right|_w}{(T_w - T_{ref})} \tag{2}$$

14 Here the vertical component is the *Z* component.
15 The source code is available in the Appendix C.

Where Z is the vertical distance between the leading edge of the hot wall and the current point [m], a_w, the thermal diffusivity at the wall [m^2/s], T_w, the wall temperature and T_{ref} is the temperature far from the wall[16].

In order to use this function, here are the lines you have to add in the *controlDict* dictionary.

```
functions
(
      heatFluxNusselt1 // name of the function
      {
            // general options
            type              heatFluxNusselt;
            functionObjectLibs ("libuserFunctionObjects.so"); // library
            log               true;       // true or false
            enabled           true;       // true or false
            outputControl     outputTime; // timeStep or outputTime
            outputInterval    1;
            // specific options
            patchName         wallHot;    // name of the analyzed patch
            vertical          Z;          // vertical direction X, Y or Z
            Ta                293;        // Temperature far away from the wall
      }
);
```

If the flag *log* is true, a simple sentence in the log file will be written to indicate which step of the function object is running. The option T_a corresponds to T_{ref} in the equation(2).

The function creates a folder for each time step (`<case root path>\<function object name>\<time step>`). In each folder, there are three files :

- *heatFluxNusselt.dat* : this file contains the current time step. Then three columns follow : the first one is the vertical coordinate of the point on the wall, the second is the heat flux at this point and the third one the Nusselt number at this point.
- *qwMean.xy* : the mean profile of the heat flux is stored in this file. The first column is the vertical coordinate and the second is the heat flux.
- *NuMean.xy* : the mean profile of the Nusselt number is stored in this file. The first column is the vertical coordinate and the second is the heat flux.

That function works properly with the version OpenFOAM-1.6 but not for parallel running.

4.3.3. Average in a homogeneous direction - sampledAveragePlane

A new approach is used to computed the average in a homogeneous direction[17]. This time the tool is based on the existing utilities : *sampledPlane* and *uniformSet*. The algorithm creates a plane normal to the homogeneous direction that will contain the mean values. For that the plane and its normal have to be specified by the user. Then for each face of the plane, a line parallel to the homogeneous is extracted. That line begins at the plane and ends at the coordinate defined by the user. The average value is computed from the number of points contained in the line.

16 The thermal diffusivity is used to correct the numeric thermal gradient.
17 The source code is available in the appendix C. It was developed under OpenFOAM-1.6

This function could be used as a post-processor with the utility "*sample*" or as a function object. To use it as a post-processor, here is an example of the "*system/sampleDict*".

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.6                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      sampleDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
interpolationScheme cell;//cell, cellPoint or cellPointFace
surfaceFormat   raw;//null (no output), vtk, raw or foamFile
sets
();
surfaces
(
    testPlane
    {
        //type to compute the average
        type averagePlane;
        //name for the plane - put what you want
        name meanPlane;

        //Here the plane is defined by a point and the normal to the plane.
        basePoint ( 0 0.0 0 );
        //the normal to the plane HAVE to be parallel to a coordinate axis
        normalVector ( 0 1 0 );

        //interpolate HAVE to be true else error occurs
        interpolate    true;
        //end of the domain (e.g. here the domain analyzed start from y=0.0 to
        //y=0.04 where y is the homogeneous direction)
        endOfDomain 0.04;
      //number of points used to compute the average. From the tests this number
      //have to be important to suppress boundary effect in the interpolation
      //of the values. If this number is too large a floating point error
      //could occur. A reduction of the number could solve that problem.
        nPoints     40;
    }
);
fields  //list of the analyzed field
(
    T
);
// ************************************************************************* //
```

Once the dictionary is defined, you can execute the command "*sample*". The output of the command will be a folder *surface* in the root case folder containing a folder for each writing interval. And the plane values are stored in those folders (one file per field).

To use it as a function object, you have to add the following lines in the "*system/controlDict*".

```
functions
(
    averageSampling     // name of the function
    {
        type surfaces;
        functionObjectLibs ("libsampling.so");
        enabled true;
        outputControl outputTime;
        outputInterval 1;

        surfaceFormat raw;//null (no output), vtk, raw or foamFile
        interpolationScheme cell;//cell, cellPoint or cellPointFace
        //list of the analyzed field
        fields
        (
            T
        );
        surfaces
        (
            plateMean
            {
                type          averagePlane;
                name        plateMean;
                //interpolate HAVE to be true else error occurs
                interpolate   true;
             //Here the plane is defined by a point and the normal to the plane.
                basePoint   (0 0 0);
             //the normal to the plane HAVE to be parallel to a coordinate axis
                normalVector    (0 1 0);
             //number of points used to compute the average. From the tests this
             //number have to be important to suppress boundary effect in the
             //interpolation of the values. If this number is too large a
             //floating point error could occur. A reduction of the number could
             //solve that problem.
                nPoints       40;
             //end of the domain (e.g. here the domain analyzed start from y=0.0
             //to y=0.04 where y is the homogeneous direction)
                    endOfDomain 0.04;
            }
        );
    }
);
```

The output will be a folder with the name of the function (here *averageSampling*) in the root case folder. That folder will contain a folder for each writing interval that contains a file for each field averaging.

5. The boundary conditions

The most important part of fireFoam for this project is not the solver, but instead the boundary conditions. Indeed, the goal is to create a good solver to study fire in a building or in a room, but the time and the resource needed to realize this have to be reasonable. Hence you can't use a very fine grid that could capture the boundary effects. The special behavior of the flow near the walls have to be modeled. As the available solutions in OpenFOAM focus on forced convection, they are not suitable for a temperature-driven flow.

First, let's remember that the purpose of a law of the wall (a model for the boundary layer) is to make a correction on the gradient of the velocity and the temperature. Indeed, when a coarse grid is used, the fast variation of the velocity and the temperature near the wall are not captured. Therefore if the numerical gradient is used to compute the heat flux or the shear stress at the wall, the value doesn't match the experimental data (or the exact numerical solution).

5.1. The goal

The final objective is to implement in fireFoam a thermal boundary condition for a buoyancy-driven boundary layer flow along a vertical solid surface. A. Trouvé made the first development. His model takes into account a possible presence of combustion and blowing (created by the wall pyrolysis).

The hypothesis is as followed :

- Stationary bidimensional flow along a vertical wall
- Large Grashof number
- Pressure effect neglected
- Fluids properties are constant except for the density (Boussinecq hypothesis)

The conservation equation for the thermal enthalpy becomes :

$$\dot{m}_f'' \frac{\partial \tilde{h}}{\partial y} = \frac{\partial}{\partial y}\left(\bar{\rho}\left(\frac{\nu}{Pr} + \frac{\nu_t}{Pr_t}\right)\frac{\partial \tilde{h}}{\partial y}\right) + \dot{q}_{comb}''' - \dot{q}_{rad}''' \qquad (3)$$

Where, $\dot{m}_f''$ is the fuel mass loss rate across the vertical wall [kg/s/m^2]

$\tilde{h}$, the mean fluid thermal enthalpy[18] [J/kg] ($\tilde{h} = \int_{T_0}^{\tilde{T}} c_p dT$)

y, the normal distance from the wall [m]

$\bar{\rho}$, the fluid mass density [kg/m^3]

$\nu\,(\nu_t)$, the molecular (turbulent) kinematic viscosity [m^2/s]

Pr (Pr_t), the molecular (turbulent) Prandtl number [-]

$\dot{q}_{comb}'''$, the heat release rate [W/m^3]

$\dot{q}_{rad}'''$, the radiation cooling rate [W/m^3]

18 The average uses here is the Fabre's average. For example the average enthalpy is $\tilde{h} = \overline{\rho h} / \bar{\rho}$

This equation could be integrated twice from the wall ($y = 0$) to the first node in the flow at y_1. Hence :

$$\tilde{h}_1 - h_w = \int_0^{y_1} \frac{-\dot{q}''_{w,c} + \dot{m}''_f(\tilde{h} - h_w) - \int_0^y (\dot{q}'''_{comb} - \dot{q}'''_{rad})\,dy}{\bar{\rho}\left(\frac{\nu}{Pr} + \frac{\nu_t}{Pr_t}\right)} dy \tag{4}$$

If some models are chosen for the turbulent kinematic viscosity and for the heat production from the combustion and from the radiation, the unknown is the heat flux or the temperature at the wall depending on the boundary conditions. So if the values of the enthalpy at y_1 and the heat flux (respectively the temperature) at the wall are known, the temperature (respectively the heat flux) at the wall could be deduced.

Unfortunately, this model is not suitable for our purpose because it is costly (spatial integration required in addition with a convergent iterative scheme) and it doesn't work with large fuel mass loss rate values. In addition to this, some works are required to model the heat release rate from the combustion and from the radiation because, until now, the assumption of uniformity is made for the term $(\dot{q}'''_{comb} - \dot{q}'''_{rad})$.

Consequently the first new model implemented in fireFoam is not this one but a model proposed by M. Hölling and H. Herwig [13]. This model is validated for High-Grashof-number turbulent natural convection in the vicinity of vertical walls. The theory is based on an asymptotic analysis of the equation in that configuration. The results show that as for a turbulent forced convection flow the boundary layer could be divided into two regions : a viscous inner layer and a fully turbulent outer layer.

The hypothesis is as followed :

- Stationary bidimensional flow along a vertical wall
- High Grashof number
- Pressure effect neglected
- Fluids properties are constant except for the density (Boussinecq approximation)

Hence the near-wall region is described by the following equations :

$$0 = \frac{\partial}{\partial y}\left(\nu \frac{\partial u}{\partial y} - \overline{u'v'}\right) + g\beta(T - T_0) \tag{5}$$

$$0 = \frac{\partial}{\partial y}\left(a \frac{\partial T}{\partial y} - \overline{v'T'}\right) \tag{6}$$

Where u, the time-averaged velocity parallel to the wall [m/s],

T, the time-averaged temperature [K],

T_0, a reference temperature [K],

$-\overline{u'v'}$, the Reynolds stress [m^2/s^2],

$-\overline{v'T'}$, the turbulent heat flux [mK/s],

a, the thermal diffusivity [m^2/s],

β, the thermal expansion coefficient [K^{-1}] and

g, the gravitational acceleration [m/s^2]

To stay general, the equations (5) and (6) are transformed to use non-dimensional variables. For that, a simple physic approach is made. The flow is characterized by $a\left.\frac{\partial T}{\partial y}\right|_w$ because it is the heat transfer with the wall that generates the flow. So this parameter has to appear in the expression of the characteristic temperature that is :

$$T_c = \left(\frac{a^2}{g\beta} \left| \frac{\partial T}{\partial y} \right|_w^3 \right)^{1/4} = \left(\left[\frac{q_w}{\rho c_p} \right]^3 \frac{1}{a\, g\, \beta} \right)^{1/4} \tag{7}$$

The characteristic distance is :

$$\delta = \frac{T_c}{|\partial T / \partial y|_w} \tag{8}$$

And the characteristic velocity is :

$$u_c = \frac{g\beta T_c^3}{\nu} \left| \frac{\partial T}{\partial y} \right|_w^{-2} \tag{9}$$

You need to resolve first the energy equation as the velocity is influenced by the temperature. And in order to solve the equation, you have to distinguish two cases : the viscous inner layer and the turbulent outer layer. In the first one the turbulent term of the equation (6) could be neglected and for the turbulent layer the turbulent term is dominant.

The results found by Hölling and Herwig are for the inner layer :

$$\Theta^{\times} = \frac{T_w - T}{T_c} = \frac{y}{\delta} = y^{\times} \tag{10}$$

And for the overlap-layer (Hölling defines this layer from the end of the inner layer to the beginning of the turbulent layer):

$$\Theta^{\times} = C \ln(y^{\times}) + D \tag{11}$$

Where $C = 0.427$ and $D = 1.93$ for a Rayleigh number $\rightarrow \infty$

Then the velocity profile could be found if the classical model for the Reynolds stress are use :

$$-\overline{u'v'} = \nu_t \frac{\partial u}{\partial y} \tag{12}$$

And the results are for the inner layer :

$$U^\times = \frac{u}{u_c} = \frac{1}{6} y^{\times 3} - \frac{1}{2} \Theta_0^\times y^{\times 2} + \left. \frac{\partial U^\times}{\partial y^\times} \right|_w y^\times \tag{13}$$

Where $\Theta_0^\times$ is the non-dimensional temperature at the reference temperature T_0 [-].

And for the overlap layer :

$$U^\times = C \frac{Pr}{Pr_t} y^\times \left(C \left[\ln(y^\times) - 2 \right] + D - \Theta_0^\times \right) + \left(0.49 \left. \frac{\partial U^\times}{\partial y^\times} \right|_w - 2.27 \right) \ln(y^\times) + 1.28 \left. \frac{\partial U^\times}{\partial y^\times} \right|_w - 1.28 \tag{14}$$

The temperature profile is relatively simple with a linear evolution in the viscous sublayer followed by a logarithmic law. But the velocity profile is far more complex especially for the overlap layer.

In order to implement that solution in a CFD software, another information is important : the range of validity for the different equation. The comparisons shown in [13] allow to make some estimation. The equations (10) and (13) are correct until $y^\times$ equal 2. And for values from 4 to 150, the equations (11) and (14) are better.

5.2. Available solutions

Before presenting how the previous equations are implemented, the boundary conditions available in OpenFoam will be presented[19].

In order to correct the shear stress, τ, (the heat flux, q_w) at a wall for a compressible LES solver, the molecular dynamic viscosity (the thermal diffusivity) of the subgride scale, μ_{Sgs} (a_{Sgs}), is modified. And to precise which model is chosen for a given surface, a dictionary for those parameters has to be present in the time directory (cf. paragraph 6.2.1. p.47 for a detailed example). And as for other fields, the model for each boundary is specified. In fact in OpenFOAM, the thermal diffusivity are not solved directly. The field solved is :

$$\alpha = a \rho \tag{15}$$

19 The source code is located in the folder "src\turbulenceModels\compressible\LES\derivedFvPatchFields\wallFunctions\alphaSgsWallFunctions" for alphaSgs and in "src\turbulenceModels\compressible\LES\derivedFvPatchFields\wallFunctions\muSgsWallFunctions" for muSgs

For the kinematic viscosity a unified law of the wall is the only model existing. The model is :

$$y^+ = U^+ + \exp(-\kappa C)\left(\exp(\kappa U^+) - 1 - \kappa U^+ - \frac{(\kappa U^+)^2}{2!} - \frac{(\kappa U^+)^3}{3!}\right) \tag{16}$$

Where, $y^+ = y u_\tau / \nu$ is the non-dimensional distance normal to the wall [-],

$u_\tau = \sqrt{\tau/\rho}$, the characteristic velocity [m/s],

$U^+ = \frac{u}{u_\tau}$, the non-dimensional velocity [-] and

κ and C are two constants[20].

This unified law is found by combining the two well-known laws :

$$\begin{cases} U^+ = y^+ \\ U^+ = \frac{1}{\kappa}\ln(y^+) + C \end{cases} \tag{17}$$

The first equation is valid for the viscous sublayer and the second for the turbulent sublayer. The unified law comes from a matching in the overlap layer. By using a unified law instead of two laws, you remove a test on the value of y^+ to know which law is correct. Or if you decide, like in our case, to implement only one law, you create constrains on the distance between the wall and the first node of the grid.

The algorithm implemented is :

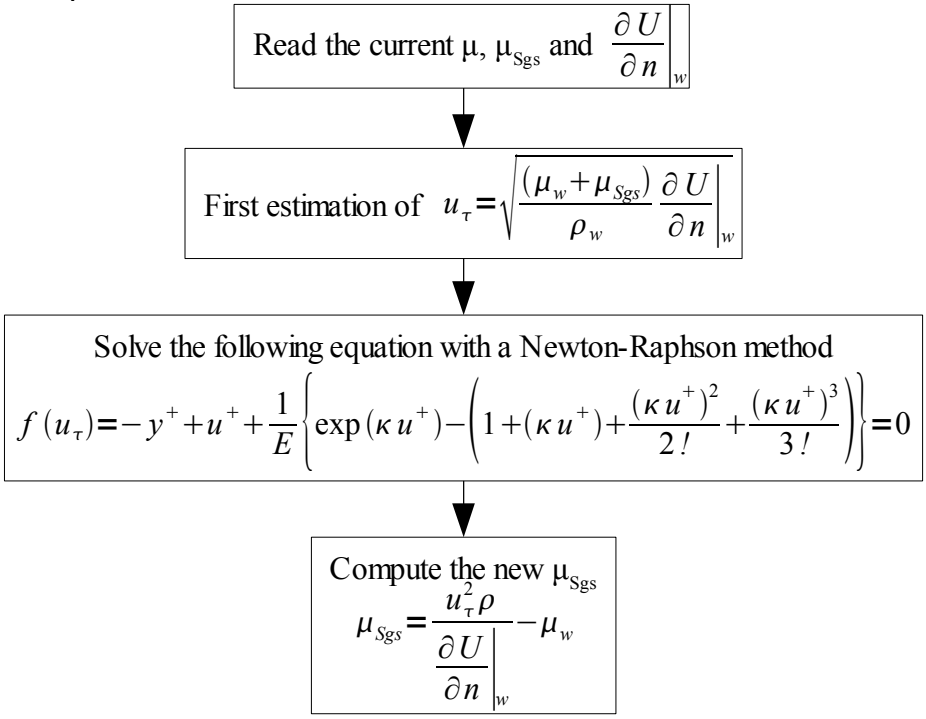

Figure 7: Law of the wall in OpenFOAM to correct the shear stress

20 κ is known as the von Karman constant and is typically equal to 0.41. C is typically equal to 5.3.

For the thermal diffusivity, two models are available. The first one is a backward compatibility with the OpenFOAM version 1.5 and is based on the turbulent Prandlt number[21] :

$$\alpha_{Sgs} = \frac{\mu_{Sgs}}{Pr_t} \tag{18}$$

Since the version 1.6, a model based on the analogy between the kinetic viscosity and the thermal diffusivity with a correction function is implemented. The correction function is the Jayatilleke "P-function".

The expression used in that case depends on the value of y_T^+ and y^+.

$$T^+(a_{eff}) = \begin{cases} Pr\, y^+ + \dfrac{\rho\, Pr\, u_\tau\, u^2}{2\,\dot{q}} & y^+ < y_T^+ \\ Pr_t \left\{ \dfrac{1}{\kappa} \ln(E\, y^+) + P \right\} + \dfrac{\rho\, u_\tau}{2\,\dot{q}} \{ Pr_t\, u^2 + (Pr - Pr_t)\, u_c^2 \} & y^+ > y_T^+ \end{cases} \tag{19}$$

Where, $T^+ = \dfrac{(T_w - T_1)\,\rho\, k_t\, u_\tau}{\dot{q}\, a_{eff}}$ is the non-dimensional temperature [-],

a_{eff}, the effective thermal diffusivity [m^2/s] ($a_{eff} = a_{Sgs} + a$)

k_t, the thermal conductivity of fluid [W/m/K]

$\dot{q}$, the heat flux at the wall [W/m^2],

y_T^+ , the y^+ value at which the linear and the logarithmic laws (17) intersects [-],

$u_c = \dfrac{u_\tau}{\kappa} \ln(E\, y_T^+)$, the mean velocity at y_T^+ [m/s] and

E, a constant typically equal to 9.8.

The Jayatilleke "P-function" is defined as :

$$P = 9.24 \left[\left(\frac{Pr}{Pr_t} \right)^{3/4} - 1 \right] (1 + 0.28\, e^{-0.007\, Pr/Pr_t}) \tag{20}$$

The algorithm computes first the characteristic velocity u_τ thanks to the logic presented in the Figure 8. Then the y_T^+ is determined in order to choose the correct relation in (19). With the right correlation, the new thermal diffusivity, a, is computed (this parameter is hidden in the non-dimensional temperature). And finally the value of α_{Sgs} is given by

$$\alpha_{Sgs} = \rho\, a_{eff} - \alpha_w \tag{21}$$

21 If the user doesn't define a value for the turbulent Prandlt number, the default value is 0.85.

Unfortunately, those models are not suitable for a temperature-driven flow as in a fire. Consequently, the model using equations (10, 11, 13 & 14 p.30) was implemented in the code. And one goal of this project is to validate this choice.

5.3. Implementation

This paragraph will describe the implementation of the Hölling model presented in the paragraph 5.1. p.28. As the problem is driven by the temperature, the model for α_{Sgs} will be introduced first followed by the correction of μ_{Sgs}.[22]

In this first approach, the solution for the overlap region is chosen. That implies a restriction on the minimal distance between the wall and the first node in the flow : the non-dimensional coordinate normal to the wall, $y_1^{\times}$ has to be greater than 4 and lower than 150.

For the heat flux correction, the algorithm is similar to the one presented at Figure 7 as shown is Figure 8.

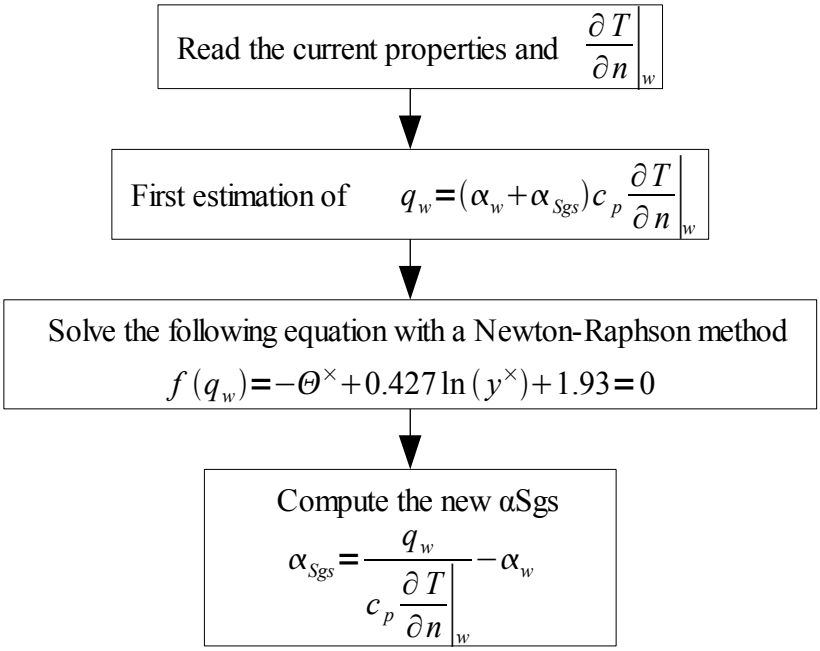

Figure 8: Implementation of the Hölling model

The correction on the velocity gradient is a bit different and doesn't need an iterative process like a Newton-Raphson method. Indeed, in the equation (14), the velocity gradient appears explicitly two times and it's possible to transformed the equation as

$$\left.\frac{\partial U^{\times}}{\partial y^{\times}}\right|_w=\frac{\alpha}{Pr\,\delta^2}\frac{Pr\frac{\delta}{\alpha}U-0.427\frac{Pr}{Pr_t}y^{\times}\left(0.427\left(\ln(y^{\times})-2\right)+1.93-\Theta_0^{\times}\right)+2.27\ln(y^{\times})-1.28}{\left(0.49\ln(y^{\times})+1.28\right)} \qquad (22)$$

22 The code itself is described in the Appendix B.

Consequently, as the second term of (22) is known, the real velocity gradient could be computed and the kinematic viscosity for the subgrid scale is given by

$$\mu_{Sgs} = \frac{\left.\frac{\partial u}{\partial y}\right|_{model} \mu_w}{\left.\frac{\partial u}{\partial y}\right|_{numeric}} - \mu_w \tag{23}$$

It is important to notice that the correction on μ_{Sgs} is realised directly after the correction on α_{Sgs}. Indeed, in the equation (22), the characteristic temperature, T_c, and the characteristic distance, δ, appear. But they depend on the temperature gradient at the wall that have to be computed using

$$\left.\frac{\partial T}{\partial n}\right|_{corrected} = \frac{\alpha_{eff}}{\alpha_w} \left.\frac{\partial T}{\partial n}\right|_{numeric} \tag{24}$$

In the OpenFOAM model to correct the velocity gradient, a unified law is used. Thanks to that the user doesn't have restrictions on the distance between the wall and the first node. This unified law is created using an matching between a linear law in the viscous inner layer and a logarithmic law in the outer layer. That is exactly the configuration obtains by Hölling for thermal boundary layer. Consequently, a unified law can be determine in that case too.

First of all, an analysis of the order between the different variables of the equation (6) have to be done using the equations (10) and (13) and the continuity equation. By assuming to be in the proximity of the wall :

$$u \propto y^3 + O(y^4) \quad => \quad v \propto y^4 + O(y^5)$$

$$\text{and} \qquad \Delta T \propto y \qquad \text{hence} \quad -\overline{v'T'} \propto \Delta T^5 + O(\Delta T^6)$$

If we model $-\overline{v'T'}$ using an analogy between the kinematic viscosity and the thermal diffusivity,

$$-\overline{v'T'} = a\,K \left[\frac{\Delta T^5}{T_c^5} + O\left(\frac{\Delta T^6}{T_c^6} \right) \right] \frac{\partial T}{\partial y} \tag{25}$$

Where K is an unknown constant. Hence the equation (6) becomes,

$$a \left[1 + K \frac{\Delta T^5}{T_c^5} + O\left(\frac{\Delta T^6}{T_c^6} \right) \right] \frac{\partial T}{\partial y} = a \left.\frac{\partial T}{\partial y}\right|_w \tag{26}$$

Or in non-dimensional form,

$$\left[1 + K\,\Theta^{\times 5} + O(\Theta^{\times 6}) \right] \partial \Theta^{\times} = \partial y^{\times} \tag{27}$$

So by integration the previous equation, we obtain

$$y^{\times} = \Theta^{\times} + K\,\Theta^{\times 6} + O\left(\Theta^{\times 7}\right) \tag{28}$$

The unified law is the result of the matching between the equations (11) and (28). But the equation (11) have to be written in another way.

$$y^{\times}=\exp\left(\frac{-D}{C}\right)\exp\left(\frac{\Theta^{\times}}{C}\right)=\exp\left(\frac{-D}{C}\right)\left[1+\frac{\Theta^{\times}}{C}+\frac{1}{2!}\left(\frac{\Theta^{\times}}{C}\right)^{2}+\frac{1}{3!}\left(\frac{\Theta^{\times}}{C}\right)^{3}+\ldots\right] \tag{29}$$

The matching is then easy. And this one gives the value of the constant K :

$$y^{\times}=\Theta^{\times}+\exp\left(\frac{-D}{C}\right)\left\{\exp\left(\frac{\Theta^{\times}}{C}\right)-\left[1+\frac{\Theta^{\times}}{C}+\frac{1}{2!}\left(\frac{\Theta^{\times}}{C}\right)^{2}+\frac{1}{3!}\left(\frac{\Theta^{\times}}{C}\right)^{3}+\frac{1}{4!}\left(\frac{\Theta^{\times}}{C}\right)^{4}+\frac{1}{5!}\left(\frac{\Theta^{\times}}{C}\right)^{5}\right]\right\} \tag{30}$$

Where $C = 0.427$ and $D = 1.93$ according Hölling.

As shown in Figure 9, the unified law matches correctly the data with high Rayleigh number for the values for C and D found by Hölling in the asymptotic analysis.

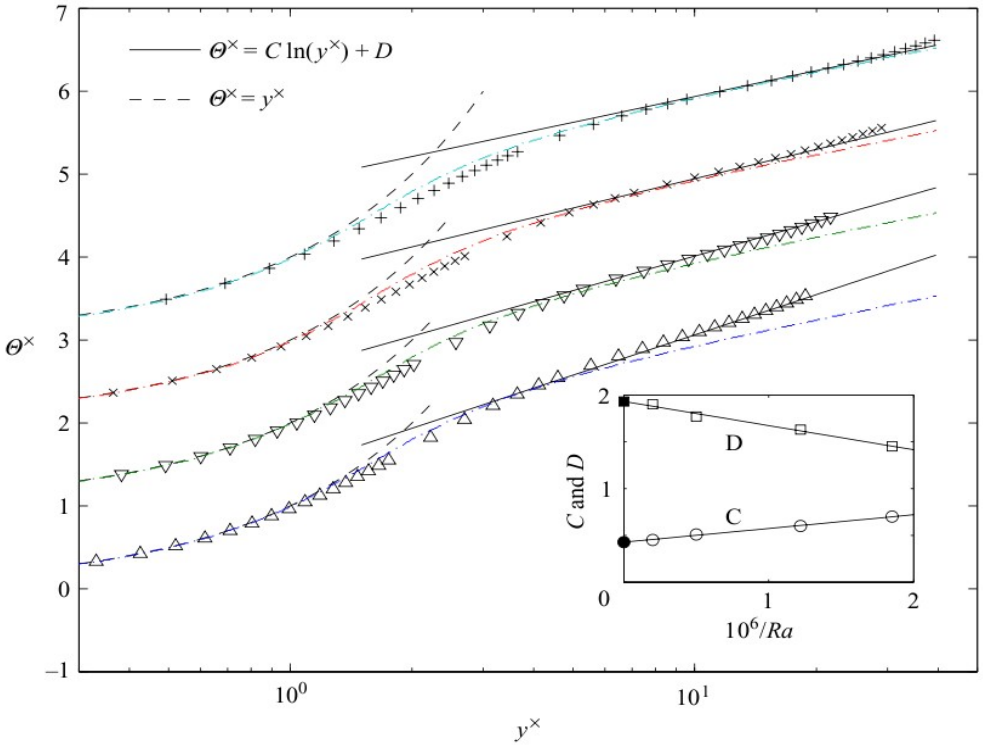

Figure 9: DNS temperature data from Versteegh & Nieuwstadt (1999) for the infinite channel:Ra = 5.4 × 10^5 (△), Ra = 8.2 × 10^5 (▽, shifted by 1), Ra = 2.0 × 10^6 (×, shifted by 2) and Ra = 5.0 × 10^6 (+, shifted by 3). Also the profiles in the viscous sublayer $\Theta^{\times} = y^{\times}$ (broken line) and in the overlap layer $\Theta^{\times} = C\ ln(y^{\times}) + D$ (solid line) are plotted. Due to small Rayleigh numbers each curve has its own value of C and D. The dash-dot lines represent the unified law in the four cases with the value of C_{∞}= 0.427 and D_{∞}=1.93 [13]

That unified law will be the next evolution of the boundary condition for α_{Sgs} coming with fireFoam. But before that the first implementation have to be tested. And that is precisely the content of the next chapter.

6. The test cases

6.1. Laminar natural convection

The first case is a vertical hot wall at 310 K surrounded by an environment at 293 K. That small difference in temperature generates a laminar natural convection structure in the fluid. This is interesting as an exact solution of the Navier-Stokes equation exists for such case.

This test was realized with OpenFOAM version 1.5 and with the development version.

6.1.1. The case

The first case[23] is composed of a wall of 20 cm height at 310 K. The size of the domain is 10x20x20 cm. And the names of the boundaries are shown in Figure 10.

The mesh chosen was (40 40 40) with a grading of (2 1 1). So the domain is divided into 40 cells in each direction. And for the *x* direction the last cell is twice the size of the first one[24].

To create a case, three parts have to be defined after the geometry : the physical properties, the solver and the initial conditions and boundary conditions.

Here is a list of all the physical properties needed for this case.

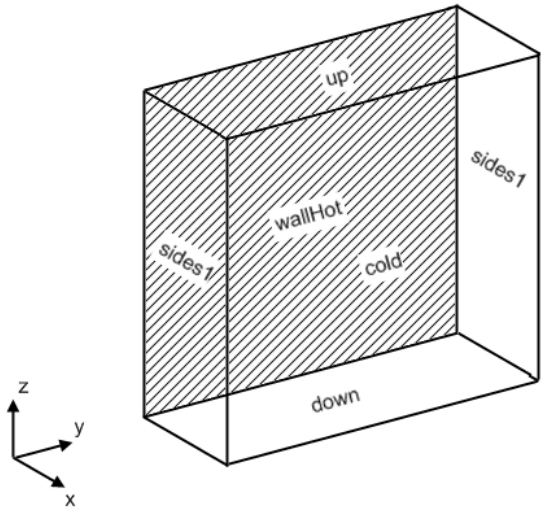

Figure 10: First case – name of the boundaries

- The gravity acceleration : *g* (0 0 -9.81)
- The coefficients for computing the variance of the mixture fraction, *ftVar* : *cftVar* = 0.09 and *cftVar2*=5
- The turbulent Prandlt number : *pr_t* = 0.3
- A flag to indicate if there is combustion : *ignited* = no
- The ramp time for the ignition : *ign_rampTime* = 0.01
- A flag to switch between a low Mach expression of the equation or a fully compressible version: *lowMach* = 1
- A flag to specified the model used to compute *ftVar* : *iftVar* = 1

23 This first case was made with OpenFOAM version 1.5 and fireFoam version 1.16 release 03/04/09. The values used in the tutorial CoFlowFlame2D that comes with fireFoam are the basis for the values specified in this first case.

24 In the future, the mesh will be described by (n_x n_y n_z) grading (S_x S_y S_z)

- A flag specifying if the flow contains different species : *multiVariate* = yes
- A flag specifying if the *MULES*[25] solver is used : *isMULES* = yes
- The reference density : *rhoRef* = 1.18426
- The reference pressure : *pRef* = 100000
- The thermodynamics model : Sutherland transport model, Janaf model for the thermodynamic and the model of perfect gas.
- The coefficients for all the previous models and for each species (here fuel, oxidant and burnt products)
- The method used to linked the mixture fraction with the species.
- The LES model uses is the one equation eddy viscosity model to compute the turbulent kinetic energy.

To solve this problem with fireFoam (release 03/04/09), 10 fields have to be defined :

- *ft* : the mixture fraction
- *ftVar* : the variance of the mixture fraction
- *b* : the regress variable. It should always be equal to 0 as the current combustion model is infinite fast chemistry.
- *Ydefault* : the mass fraction. This variable allows to define the boundary conditions for each species. That avoid to create a file for each species present in the flow.
- *T* : the temperature
- *p* : the pressure. If the low Mach hypothesis is made, *p* is really the pressure : $p = p_{ref} + \rho g z$ But if the fully compressible model is used, *p* represents the total pressure : $p = p_{ref} + \rho g z + p_d$
- *pd* : the dynamic pressure
- *U* : the velocity
- *muSgs* : the dynamic viscosity for the subscale grid
- *k* : the turbulent kinetic energy

Table 2 presents the discretization schemes used for this case. The discretization requires to know the value of fields on center faces. However, with the finite volume model, it is the value at the center cell that is known. So an interpolation scheme computes the value on a face using the values of the center cells connected by that face. For the laplacian, the surface normal gradient is needed. In this case, the normal gradient is evaluated from the value of the 2 cells connected by the surface with a non-orthogonal correction.

25 MULES is the acronym for Multidimensional Universal Limiter for Explicit Solution.

Derivation	Scheme		Interpolation
Time	Euler implicit	$\frac{\partial}{\partial t}\int_V \rho\phi dV = \frac{(\rho_P\phi_P V)^n - (\rho_P\phi_P V)^0}{\Delta t}$	
Gradient	Gaussian integration	$\int_V \nabla\phi dV = \int_S dS\,\phi = \sum_f S_f\phi_f$	Linear
Divergence	Gaussian integration	$\int_V \nabla\cdot\phi\, dV = \int_S dS\cdot\phi = \sum_f S_f\cdot\phi_f$	Total variation diminishing
Laplacian	Gaussian integration	$\int_V \nabla\cdot(\Gamma\nabla\phi)dV = \int_S dS\cdot(\Gamma\nabla\phi) = \sum_f \Gamma_f S_f\cdot(\nabla\phi)_f$	Linear / explicit non-orthogonal correction

Table 2: Discretization schemes - first case

The following table presents the initial conditions and the boundary conditions.

Field	internal	sides1	wallHot	up	down	cold
ft	0	cyclic	0			
ftVar	0	cyclic	0			
b	0	cyclic	Zero gradient			
Ydefault	0	cyclic	Zero gradient			
T	293	cyclic	310	Zero gradient		
p	100000	cyclic	Calculated (default value 100000)			
pd	0	cyclic	Zero gradient	Total pressure ($p_0 = 0$)		
U	(0 0 0)	cyclic	(0 0 0)	pressureInletOuletVelocity		
muSgs	0	cyclic	Zero gradient			
k	10^{-5}	cyclic	10^{-5}	inletOutlet (value 10^{-5})		

Table 3: First case - boundary conditions

The boundary conditions for the velocity, the pressure and the dynamic pressure are linked. Here are some additional explanations for the boundaries chosen :

- The total pressure condition implies that the field is equal to $p_0 + \frac{1}{2}\rho|U|^2$
- The *pressureInletOutletVelocity* condition is the combination of a condition *inletOutlet* (switch *U* and *p* between a fixed value and a zero gradient depending on the direction of *U*) and a condition *pressureInletVelocity* (when *p* is known at the inlet, *U* is evaluated from the flux normal to the patch).
- The total pressure is calculated by $p = p_d + p_{ref} + \rho g h$
- The condition cyclic means that the value on both sides, called sides1 in this case, are equal. That simulates a homogeneous direction.

To monitor the convergence of the case, the control parameters maximum of U_z and the integral of k are followed. Figure 11 shows the evolution of the integral of k.

The last step before having a look to the results is to execute the solver fireFoam. Then paraView could be use to visualize the solution.

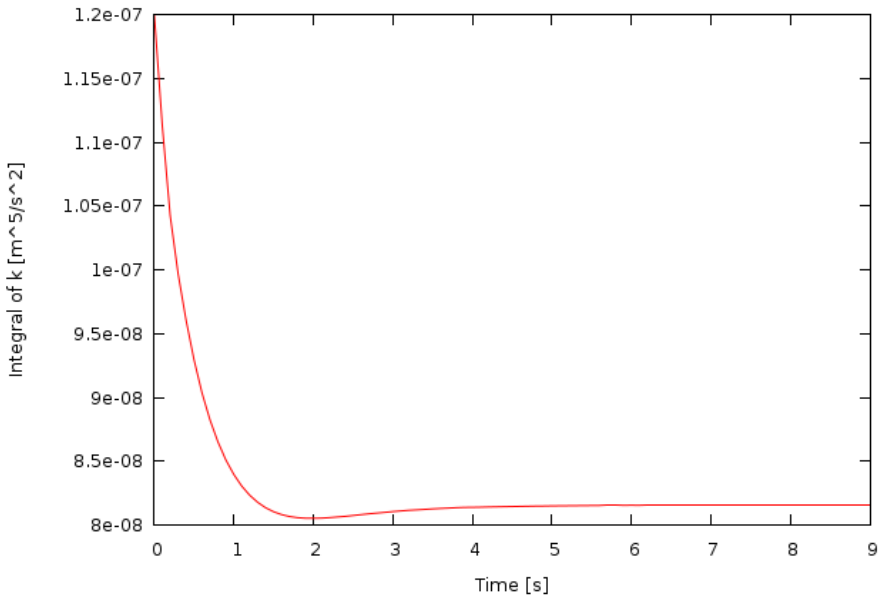

Figure 11: Evolution of the integral of k

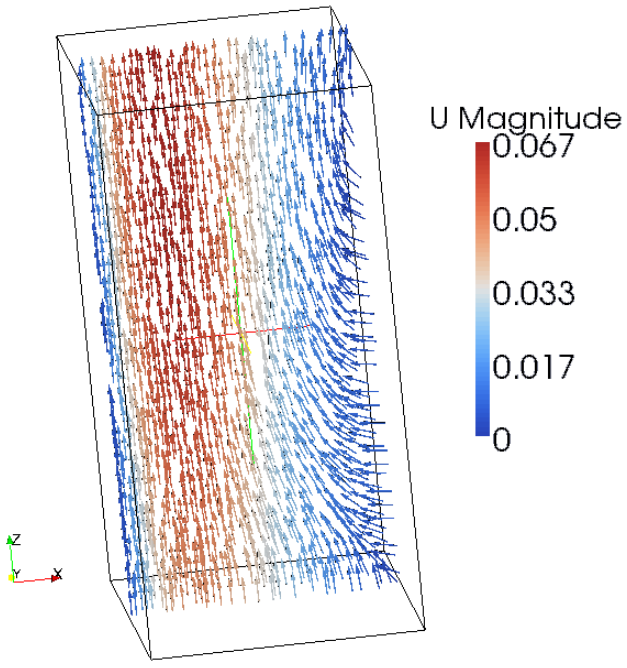

Figure 12: First case : velocity field [m/s]

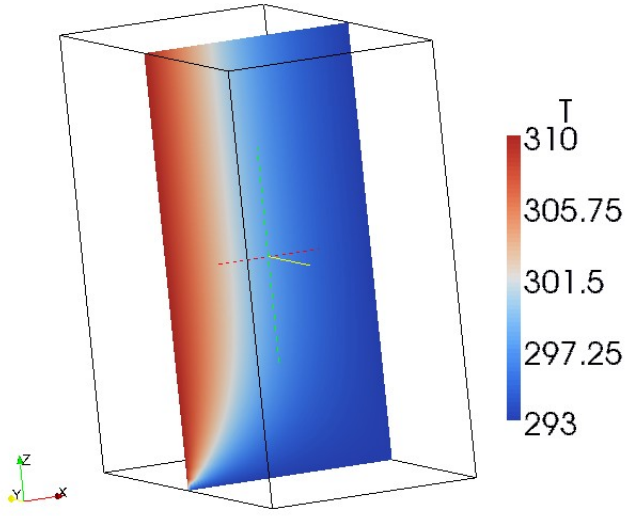

Figure 13: First case : temperature field [K]

The temperature field (Figure 13) looks correct. But the velocity field (Figure 12) is wrong because the boundary layer is mainly fed by the flow coming from the south border and not the east border. Consequently the boundary conditions chosen are incorrect. And they have to be changed. The modifications made followed the suggestion from Dr. Yi Wang, our FM Global contact who has a good experience on OpenFOAM. That new boundary conditions are described in Table 4 in which the modifications are in bold.

Field	**internal**	**sides1**	**wallHot**	**up**	**down**	**cold**
ft	0	cyclic	0			
ftVar	0	cyclic	0			
b	0	cyclic	Zero gradient			
Ydefault	0	cyclic	Zero gradient			
T	293	cyclic	310	**inletOutlet (value 293)**		
p	100000	cyclic	Calculated (default value 100000)			
pd	0	cyclic	**fixedFluxBuoyantPressure**	**Zero gradient**	Total pressure = 0	
U	(0 0 0)	cyclic	(0 0 0)	**inletOutlet**	pressureInletOuletVelocity	
muSgs	0	cyclic	Zero gradient			
k	10^{-5}	cyclic	10^{-5}	inletOutlet (value 10^{-5})		

Table 4: Correction of the boundary conditions

The *fixedFluxBuoyantPressure* defined the field gradient at the boundary as : $\frac{\partial f}{\partial n} = -g z \frac{\partial \rho}{\partial n}$.

Some errors in the choice of parameters were also corrected (isMULES yes → no; lowMach 1 → 0). The first command stops using the solver MULES. And the second change the equation solved from a low Mach formulation to a fully compressible equation.

The last change was an increase in the x direction to use a volume of 30x20x20 cm. Consequently the mesh was adapted to (120 40 40) grading (2 1 1).

The velocity field (Figure 14) keeps a structure with the boundary layer mainly fed by the south border. So the boundary condition for the velocity at the south border was change to slip that fix the normal component to zero and fixed the gradient of the tangential component to zero. Like that (Figure 15), the flow comes effectively from the east border. But unfortunately, a vortex at the north border implies a negative vertical component of the velocity in a part of the flow. It's unexpected, as the experimental data show a vertical component that goes asymptotically to zero far away from the wall.

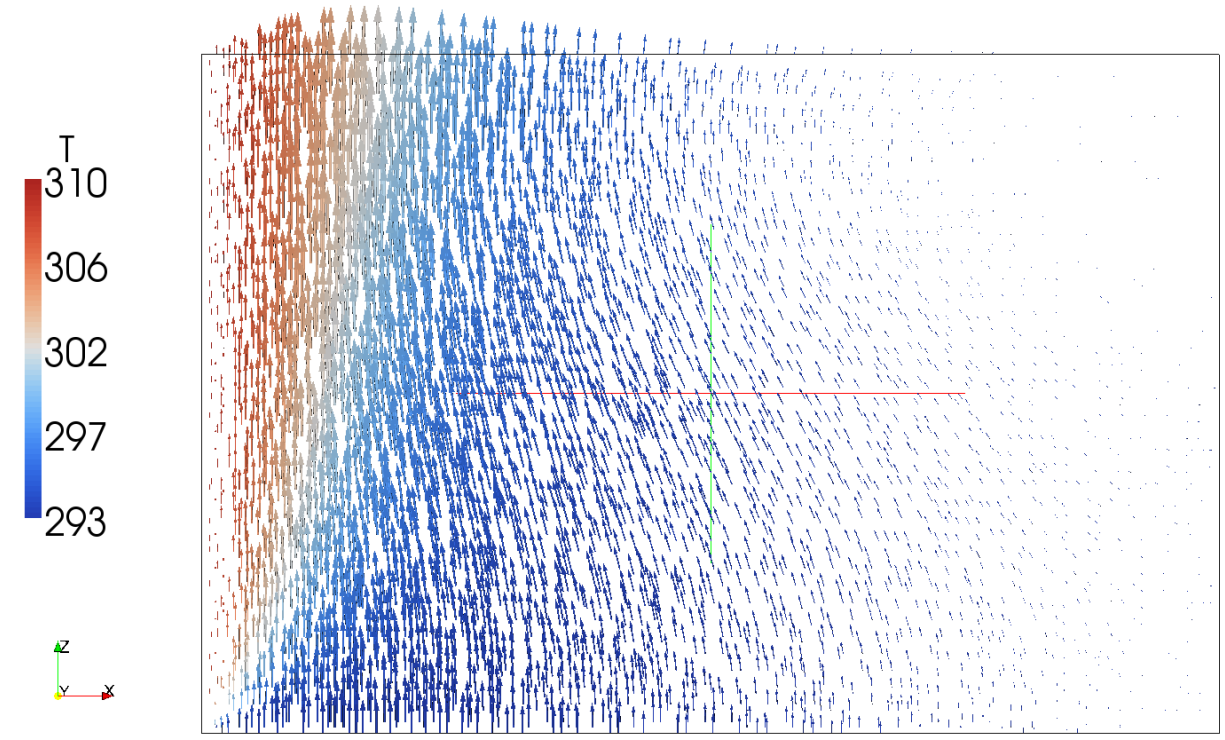

Figure 14: Velocity colored by the temperature - 1st case with first correction of the boundary conditions

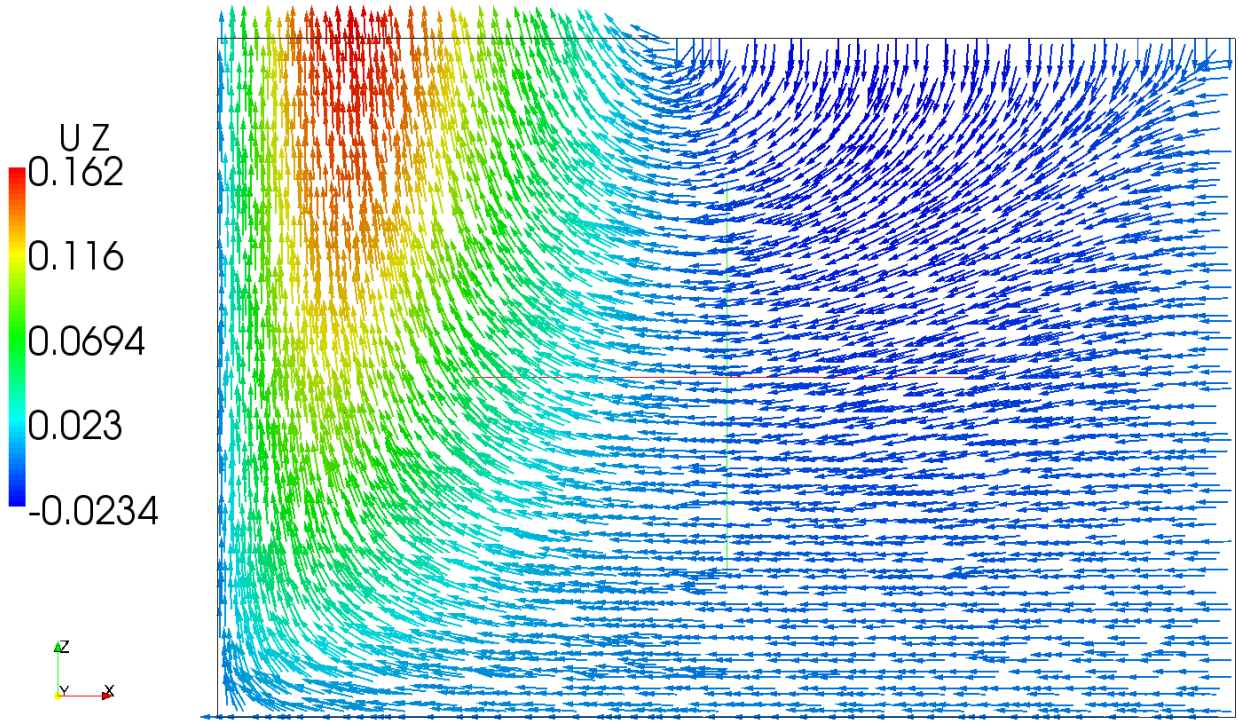

Figure 15: Direction of the velocity color by Uz - 1st case with a slip BC at the south border

An additional validation was carried out with the exact solution of Ostrach [14]. This validation was a comparison between the temperature and velocity profiles at Z = 10 and 20 cm. However, as shown in Figure 16 and Figure 17 (curve "*Without cold wall*"), the only similarity is the value of the peak in the vertical velocity. So two types of solutions were studied : the mesh size and the south boundary conditions.

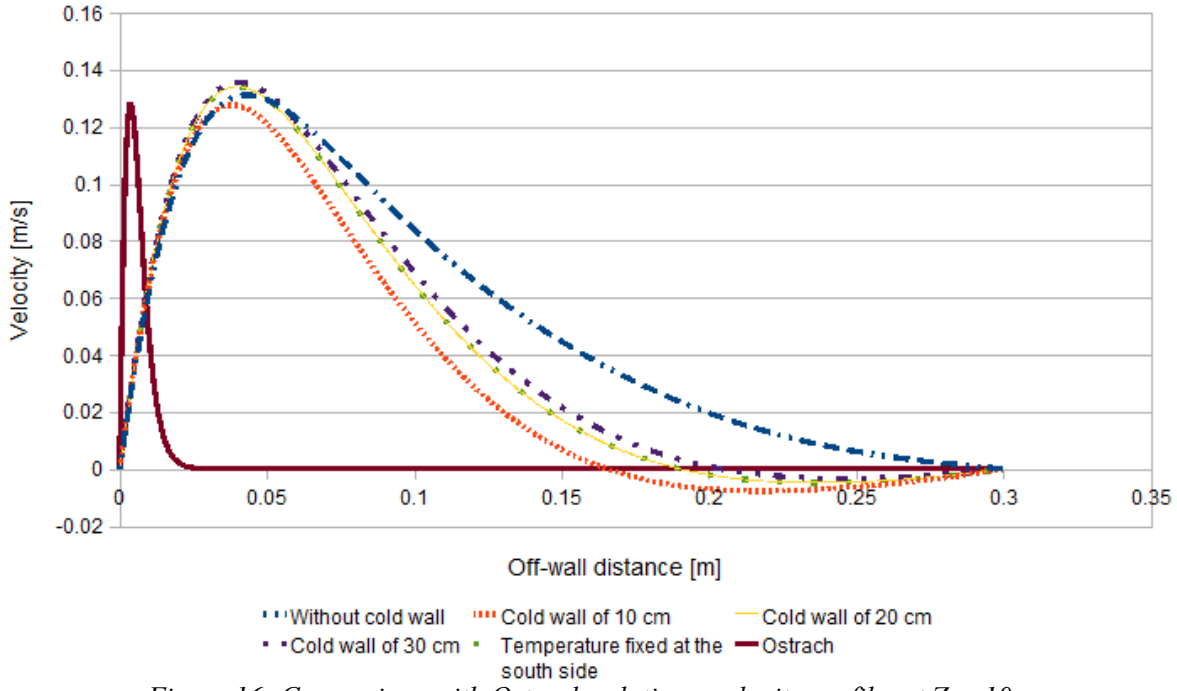

Figure 16: Comparison with Ostrach solution - velocity profile at Z = 10 cm

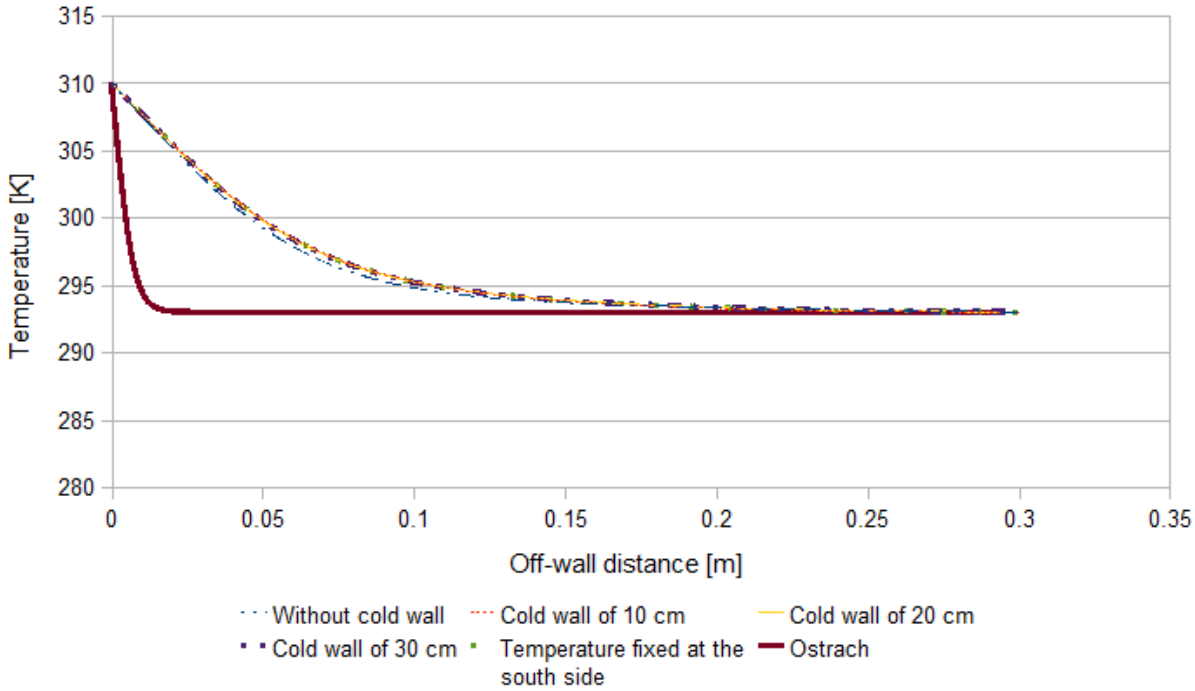

Figure 17: Comparison with Ostrach solution - temperature profile at Z = 10 cm

To try to fix the trouble with the south boundary conditions, the case was change by adding a cold wall (same temperature than the environment – 293K) below the hot wall. The goal was to reproduce a correct boundary condition at the leading edge of the hot wall. And in order to assure the independence of the solution with the size of the cold wall, three cold wall were tried with a height of 10, 20 and 30 cm. Because of this addition the boundary condition at the south boundary were modified for the velocity and the temperature. Here are the different combinations tested : fixed value of (0 0 0) or slip condition for the velocity and fixed value at 293 K or zero gradient for the temperature.

In Figure 16 and Figure 17 above, the curves for the three size of the wall with the boundary conditions of (0 0 0) for the velocity and a zero gradient for temperature are drawn. The case with a fixed temperature correspond at a cold wall of 20 cm and the velocity fixed at (0 0 0). But in all cases the modifications are small and the solutions stay far from the exact solution of Ostrach.

So a new series of tests were carried out by using a finer mesh and/or by increasing the grading in the x direction to catch the peak velocity in the proximity of the wall. But those new series, in addition with a huge increase of needed time, brought a new problem. The solution was unstable as shown in Figure 18.

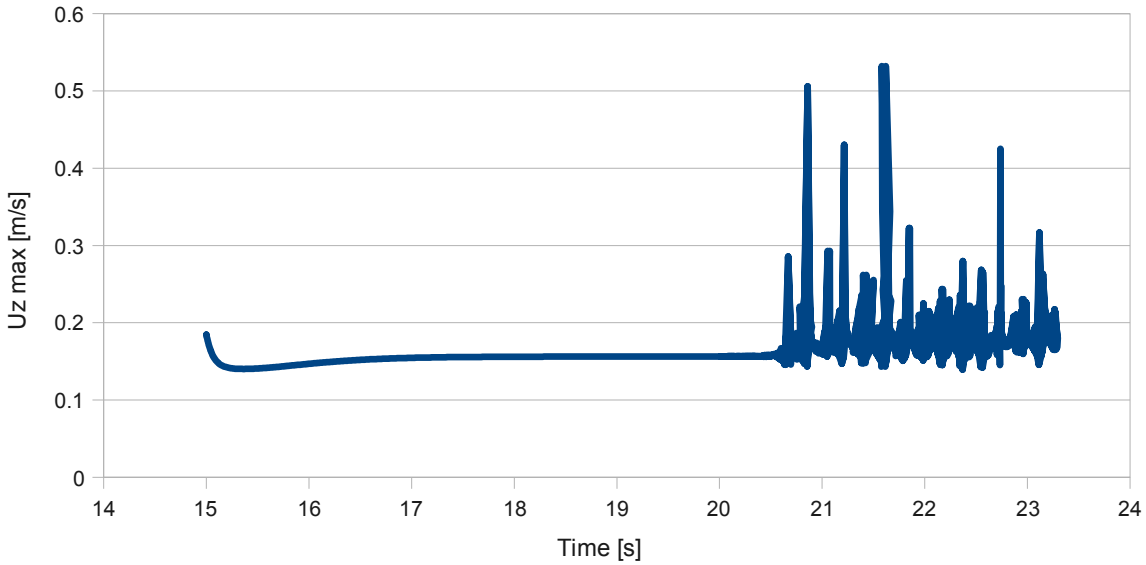

Figure 18: Evolution of the maximum of Uz for a finer mesh - (150 50 50) grading (2.5 1 1)

The reason of that instability is unclear. It could come from the finesse of the mesh. The resolution could be captured some acoustic modes of the volume. And if the boundary conditions are inadequate, some resonances could appear and generate the instability.

After a new discussion with Dr. Wang, we move on a fresh case defined on the new version of OpenFOAM/fireFoam (OpenFOAM-dev with fireFoam-dev release 06/18/09). The configuration is similar to the one presented at Figure 10. The only difference is the height of the wall : 25 cm. Four meshes were tested to insure the grid independence of the solution : (40 10 50) grading (3 1 1), (60 10 75) grading (3 1 1), (80 10 100) grading (3 1 1), and (120 15 150) grading (3 1 1).

In general there are few modifications of the parameters. Except the LES model, the model uses now is the Smagorinsky model and no more the one equation eddy viscosity model. The other important change is to turn off the multivariate solver.

For the boundary conditions there are really similar to those presented in Table 4. A new field appears : *alphaSgs*, the thermal diffusivity for the subgride scale multiply by the density. It is a consequence of using the new version of OpenFOAM.

Field	**internal**	**sides1**	**wallHot**	**up**	**down**	**cold**
ft	0	cyclic	Zero gradient	**InletOutlet (value 0)**		
ftVar	0	cyclic	0			
b	0	cyclic	Zero gradient			
Ydefault	0	cyclic	Zero gradient			
T	293	cyclic	310	inletOutlet (value 293)		
p	101325	cyclic	Calculated (default value 101325)			
pd	0	cyclic	fixedFluxBuoyantPressure	Zero gradient	Total pressure ($p_0 = 0$)	
U	(0 0 0)	cyclic	(0 0 0)	inletOutlet	pressureInletOuletVelocity	
muSgs	0	cyclic	Zero gradient			
alphaSgs	**0**	**cyclic**	**Zero gradient**			
k	10^{-5}	cyclic	10^{-5}	inletOutlet (value 10^{-5})		

Table 5: Boundary conditions for the natural convection case

That new configuration works great as the detail analysis of the next paragraph will show you.

6.1.2. Results

To validate the results, they are compared to the exact solution using the equation developed by Ostrach [14]. That solution is correct for a laminar natural convection case and at a small distance from the leading edge of the hot wall[26]. Consequently another simulation with a 50-centimeter hot wall was carried out too.

The validation is made on two aspects : the heat flux at the wall and the temperature and velocity profiles.

26 That distance has to be large as compared to the boundary layer thickness.

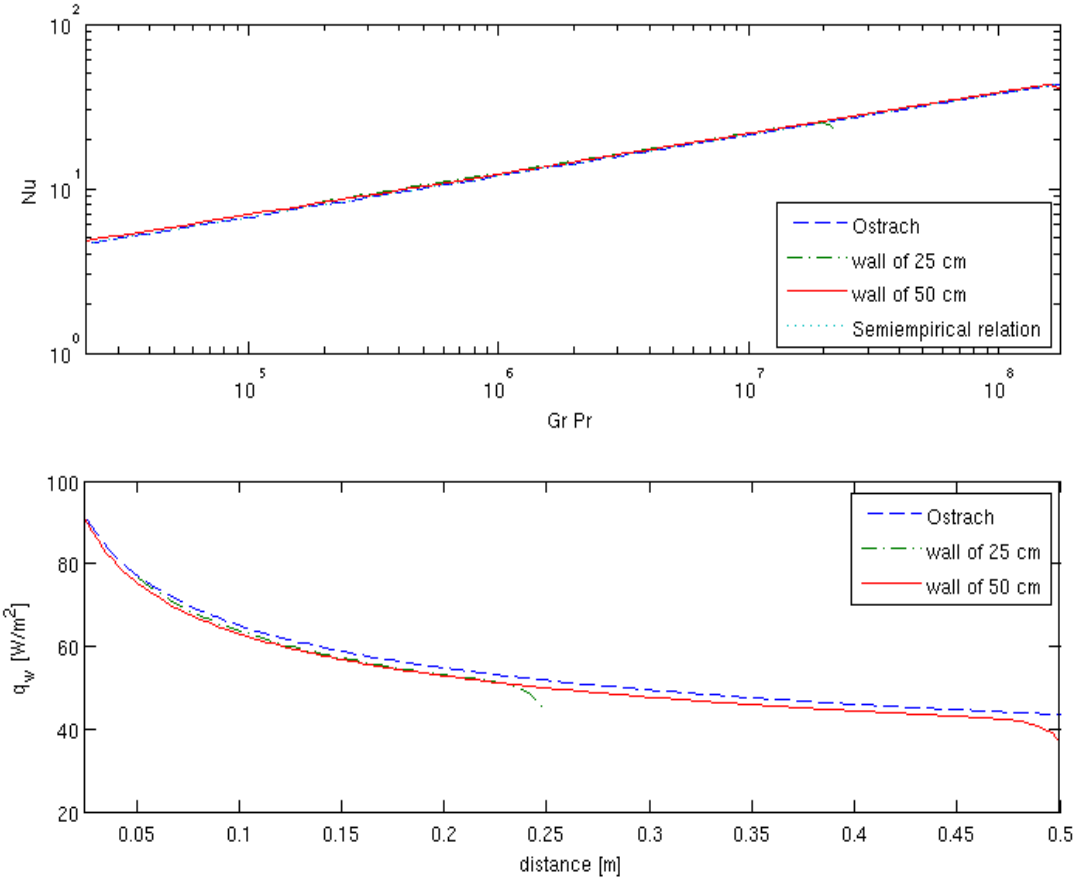

Figure 19: Comparison Ostrach and results from OpenFOAM

The heat flux and the Nusselt number are very well computed except at the end of the hot wall where some boundary effects appear. For the Nusselt number, the data are also compared to a semi-empirical relation with the coefficient found by Tsuji [15].

$$Nu = 0.387 (Gr_z Pr)^{1/4} \tag{31}$$

As shown in Figure 20 and Figure 21, the temperature and velocity profiles fit also very well the Ostrach's solution.

6.1.3. Comments

After lots of troubles, the software was validated for a laminar natural convection case. However the main goal of this project is to test a model for the boundary conditions in a turbulent regime. Consequently, a new case that copy the experiment of Tsuji [15] was tested. But, as this case requires lots of resources and time, a debugging phase of the boundary model was made in parallel using data from the previous natural convection test. The goal of this debugging part was to check the resolution of the model under OpenFOAM and also the values of the different parameters read by the model. This small study finished to validate the boundary model as no error was found.

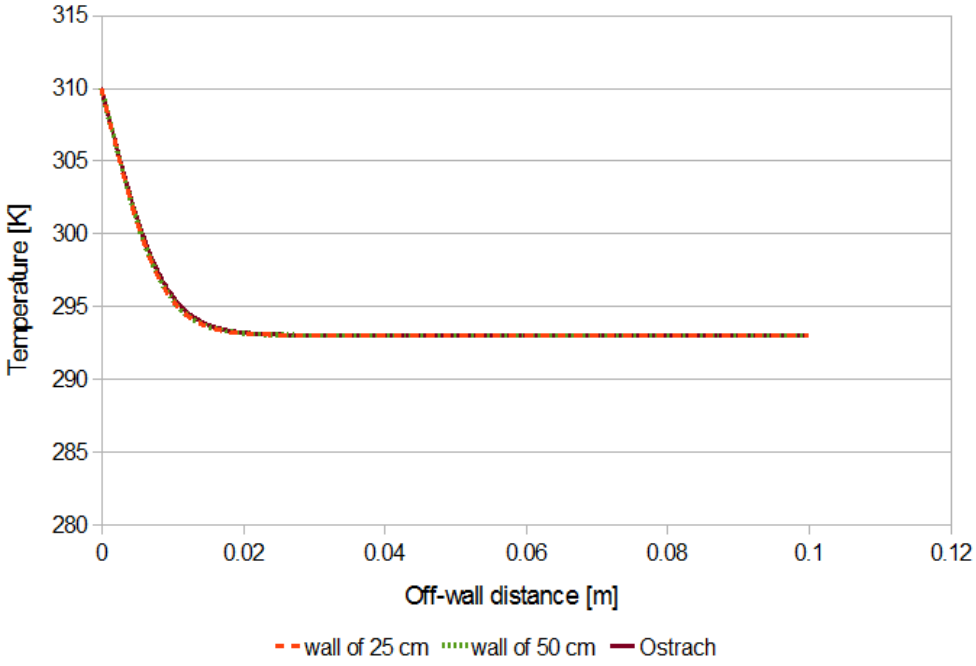

Figure 20: Comparison with Ostrach solution - temperature profile at Z = 20 cm

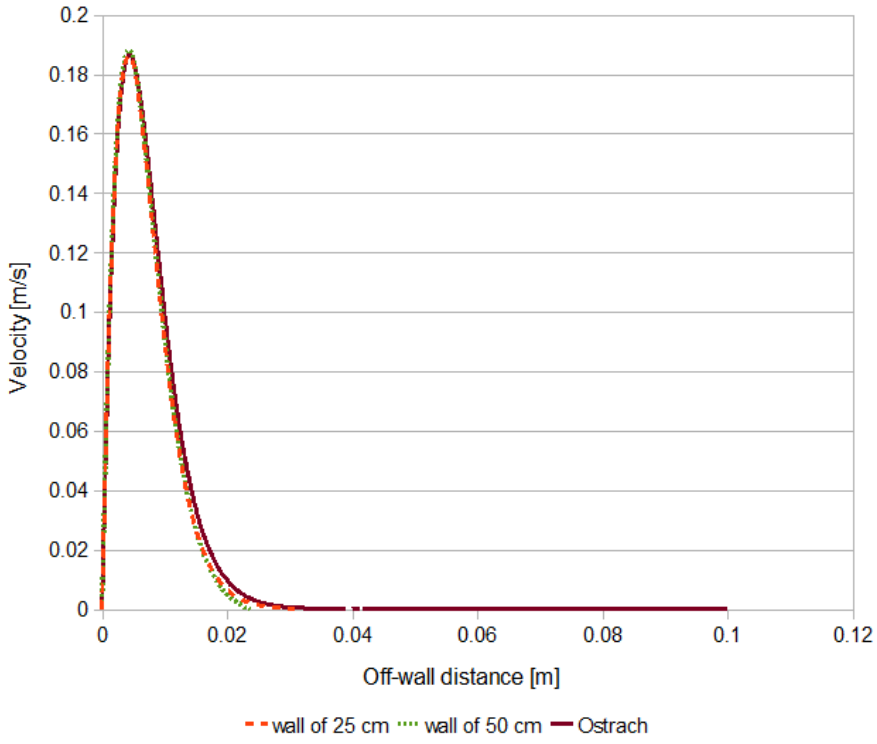

Figure 21: Comparison with Ostrach solution - Uz profile at Z = 20 cm

6.2. Turbulent natural convection

After a first case to look at the validation of the solver on a laminar natural convection case, a validation of the new model is wanted. For that a natural convection case is still chosen but with a part of the flow fully turbulent. The test case is a copy of the experiment realized by Tsuji [15]. As the experimental data are available in the ERCOFTAC database [16], it will be easy to compare the results from the experiment, those from the resolved simulation and the values provides by the model.

6.2.1. The case[27]

The domain and the names of the boundaries are similar to those presented in the Figure 10 p.36. This time the hot wall is 4 meters height and heat at 333 K in the configuration chosen. From the experimental data, it is expected to have a laminar flow until 1 meter and a fully turbulent flow from 1.2 meter until the top.

Two configurations should be tested : a mesh resolved solution and a coarser mesh using the new model for the boundary conditions. But by lack of time, those simulations are still running or required some changes in the finesse of the mesh. So in this section, the results shown are the ones available so far.

The parameters for the solvers and the physical properties are similar to those used in the last laminar convection case. In the case of the resolved mesh, for the initial conditions and boundary conditions are identical (cf. Table 5 p.44 for more details)[28]. But in the case of the coarser mesh, the boundary conditions are different for the fields *muSgs* and *alphaSgs* along the hot wall. For *muSgs* the boundary condition used is *muSgsBuoyantWallFunction* and for *alphaSgs* is *alphaSgsBuoyantWallFunction*. Those boundaries are the implementation of the Hölling model.

The geometry for the resolved mesh and the coarser mesh are different. The resolved mesh is divided into three parts for the laminar, the transition and the fully turbulent flow. The finesse for the laminar part is inspired from the one used to solve the laminar case. For the turbulent flow, suggestions from Dr. Wang were taken into account. And finally the transition was design to have a smoothly evolution between the two parts. So the first laminar domain is 0.3 m x 0.15 m x 1 m with a mesh (120 30 160) simpleGrading (4.5 1 1). The fully turbulent domain is 0.3 m x 0.15 m x 2.8 m with a mesh (120 30 100) simpleGrading (4.5 1 1). And the transition is 0.3 m x 0.15 m x 0.2 m with a mesh of (120 30 20) simpleGrading (4.5 1 7). The 30 cm in the direction normal to the wall is determined using the velocity profiles measured by Tsuji to know the minimum distance influenced by the wall (here at z=3.244, U_z is non-zero until x = 22.7 cm). As this simulation requires lots of resources, the cluster of the university was used to resolve it.

As the model doesn't work with laminar flow, only the fully turbulent part is modeled from 1.438m to 4m. The other reason is the short available time to run the case. The choice of z=1.438 m is due to the experimental data. Indeed the first turbulent profile measured was at that height. Then the boundary conditions for the velocity and the temperature were switched to a fixed-value condition

27 The appendix D provides a complete view of the files needed to simulate the case of the resolved mesh.

28 If you look at the appendix D, you could have the impression that the boundary condition for *pd* along the hot wall is different. But it is not. The name of the *fixedFluxBuoyantPressure* was changed to *buoyantPressure* in OpenFOAM version 1.6.

where the values are interpolated from the experimental data. Consequently, the domain is 0.5m x 0.5m x 2.562m with a mesh (50 50 100) and a uniform grading. For the model the 50 cm off-wall is the value advised in the ERCOFTAC database [16]. And the division in the normal direction is determined in order to put the first node in the overlap layer of the boundary layer where the model implemented is correct. From Tsuji's data the heat flux in the turbulent zone is about 210 W/m2. Consequently the first node have to be at 4.6 mm off the wall. Here 1 cm is chosen.

6.2.2. Results

As for the laminar case, two aspects are analyzed : the heat flux at the wall and the profiles of the temperature and the vertical velocity.

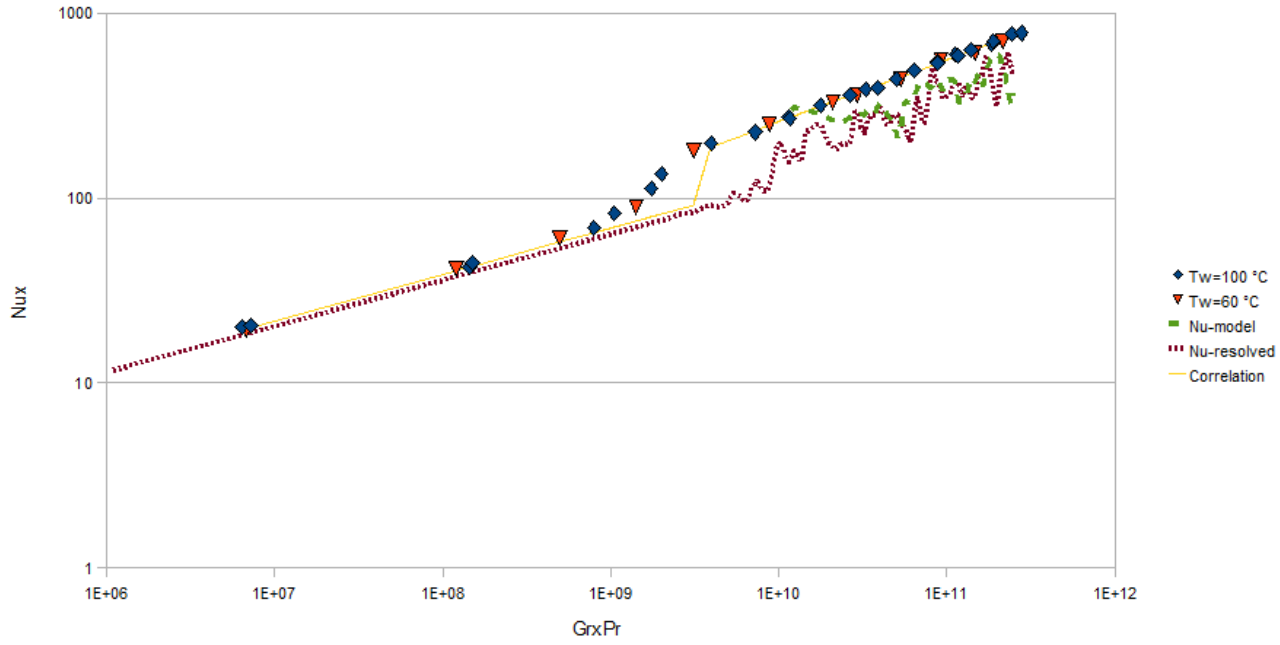

Figure 22: Tsuji experiment - Nusselt number

Tsuji realized an experiment for a hot wall at 333 K and at 373 K. And he found correlations between the Rayleigh number and the Nusselt number for the laminar and the turbulent parts (respectively (31) and $Nu_x = 0.120 (Gr_x Pr)^{1/3}$). The laminar part is well simulated by the resolved mesh. But the turbulent part is far from a steady-state so no precise conclusion can be made here. For the resolved mesh, the transition seems to occur a bit later than in the experiment.

And example of the profiles of temperature and vertical velocity for z = 3.244 m[29] is shown in Figure 23 and Figure 24. It seems that the global gradient of the temperature and the velocity at the wall is a bit underestimated by the model. But the biggest difference is the peak velocity value when the model is used. Each time it is overstimated.

29 It is the highest point for which experimental data are available. So the model has to be the most accurate there as the Grashof number is the largest.

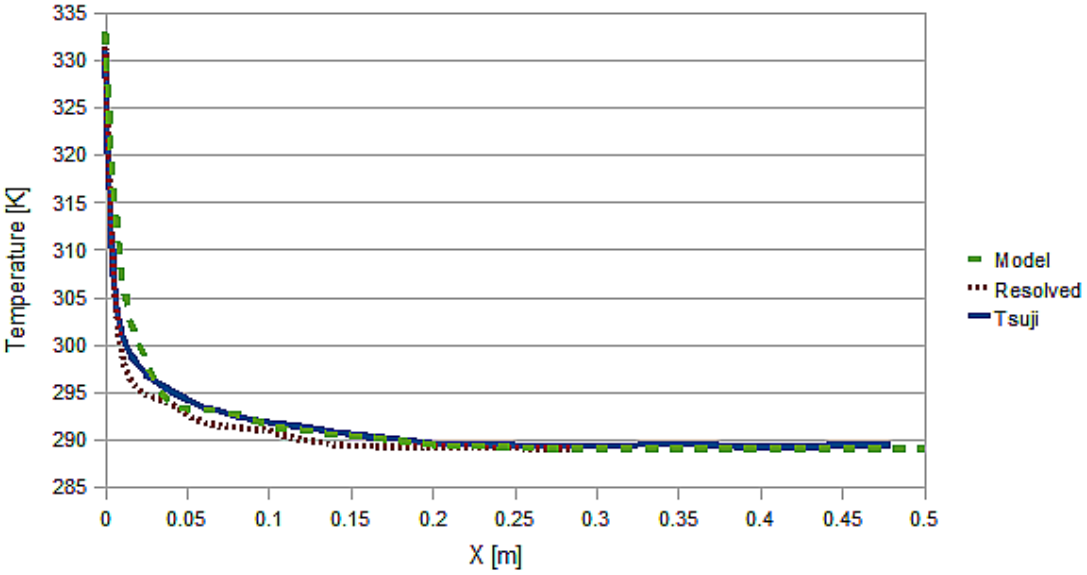

Figure 23: Temperature profiles at Z = 3.244 m

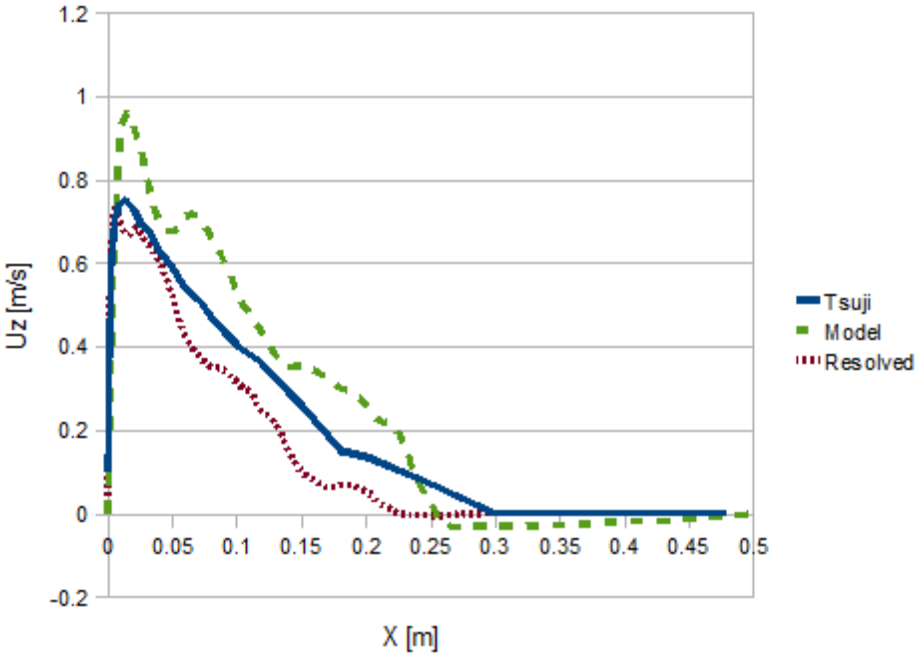

Figure 24: Vertical velocity profiles at Z = 3.244 m

6.2.3. Comments

Unfortunately, the mesh creation and the computational time needed to complete this second case took more time than expected. So it's complicated to conclude on the validity of the new boundary conditions. In particular on the difference for the velocity peak value.

7. Conclusion

By having a look on the other side of a computational fluid dynamics software, the complexity and the level of programming required to obtain a competitive, accurate and robust solution appears obvious. Consequently the addition of a new function even as simple as a post-processor to analyze the output data could take a lot of time to understand correctly the relation between the classes in the code. Besides to validate a new content, a rigorous analyze have to be performed on pertinent test cases. Indeed the validation requires comparison with previous solutions from other simulations or from experiments. That comparison is very important especially for CFD solutions that give in nearly all cases a result.

The validity of that result is strongly linked with the chosen parameters for the solvers or the boundary conditions. Unfortunately I realized that the choice of such variables is not easy and even really hard without some experience in CFD. A wrong choice could lead to a big loss of time. Especially when the source of the error is not common or obvious, like the instability encountered during the laminar test case.

Although the new model is not validated at the end of this project, the way to do it is much shorter now. And the troubles encountered were an opportunity to increase my knowledge on fluid dynamics and on computer science. So I had the possibility to develop my skills in C++ and in parallel computation. I also increased the necessary communication skills in English.

If the new boundary model is validated on the Tsuji case, different further steps could be imagined. Another test case could be done for two vertical walls differentially heated. And closer of a fire case, an experiment done by FM Global of a wall fire could be simulated. That last test could give an idea of the accuracy of the model in a typical fire situation where lots of other physical phenomena occur like pyrolysis, losses by radiation or combustion.

In order to improve the model, instead of developing a totally new one, the parameters of the law (mainly C and D (11)) could be parametrized in function of the Rayleigh number at least as shown in the Figure 9 p.35. That should improve the accuracy for low-Grashof-number flow commonly encountered in fire application. But before that the first simple betterment will be to implement the unified law and the model for the velocity inner layer.

8. Bibliography

[1] Office of University Communications (UMD), *University timeline*. (Seen on 07/10/2009) URL : http://www.urhome.umd.edu/timeline/

[2] Office of University Communications (UMD), *UMD Facts & Figures*. (Seen on 07/10/2009) URL : http://www.newsdesk.umd.edu/facts/quickfacts.cfm

[3] University of Maryland, *About the Department*, 2008. (Seen on 07/10/2009) URL : http://www.fpe.umd.edu/about/index.html

[4] FM Global, *Welcome to FM Global*, 2009. (Seen on 07/12/2009) URL : http://www.fmglobal.com/default.aspx

[5] FM Global, *2008 Annual Report*, FM Global 2009, USA.

[6] S. OLENICK and D. CARPENTER, *An Updated International Survey of Computer Models for Fire and Smoke*, SFPE Journal of Fire Protection Engineering 2003. 13 (2) 87-110

[7] CAPRON T., *Développement d'un logiciel de simulation de la propagation des fumées d'incendie dans un bâtiment*, Faculté Polytechnique de Mons 2008. (Master thesis)

[8] NIST, *Fire Growth and Smoke Transport Modeling with CFAST*, 2009. (Seen on 07/04/2009) URL : http://cfast.nist.gov/index.htm

[9] BENFER M., CAREY A., KOUTSAVLIS E. and SALYERS B., *ENFP 416 Final Report - Group 4*, University of Maryland 2009, College Park. (Student project)

[10] OpenCFD Ltd, *OpenFOAM®: The Open Source CFD Toolbox*. (Seen on 07/20/2009) URL : http://www.opencfd.co.uk/openfoam/index.html#openfoam

[11] OpenCFD Limited, *OpenFOAM The OpenSource CFD Toolbox - User Guide*, 2008.

[12] Kitware, Inc., *ParaView*, 2008-09. (Seen on 07/24/2009) URL : http://www.paraview.org/

[13] HOLLING M. and HERWIG H., *Asymptotic analysis of the near-wall region of turbulent natural convection flows*, J. Fluid Mechanics 2005, Cambridge. 541 383-397

[14] OSTRACH S., *An analysis of laminar free-convection flow and heat transfer about a flat plate parallel to the direction of the generating body force*, NACA Rep. 111 1951.

[15] TSUJI, T. and NAGANO, Y., *Characteristics of a turbulent natural convection boundary layer along a vertical flat plate*, Intl J. Heat Mass Transfer 1988. p. 1723-1735

[16] University of Manchester, *ECOFTAC "Classical Collection" Database*. (Seen on 08/15/2009) URL : http://cfd.mace.manchester.ac.uk/ercoftac/index.html

[17] SCHLICHTING, H. and GERSTEN, K., *Boundary-layer Theory*, Springer 2003, Berlin.

[18] POINSOT T. and VEYNANTE D., *Theoretical and Numerical Combustion Second Edition*, RT Edwards 2005.

[19] COUSSEMENT G., *Mécanique des fluides*, Faculté Polytechnique de Mons 2008, Mons. (Class notes)

[20] COUSTEIX J. & MAUSS J., *Asymptotic Analysis And Boundary Layers*, Springer 2007.

[21] CHASSAING P., *Turbulence et mélange*, INP ENSEEIHT 2008, Toulouse. (Class notes)

[22] LOUDON K., *C++ précis et concis*, O'Reilly 2003.

[23] DELANNOY C., *Exercices en langage C++*, Eyrolles 2007, Paris.

Appendix A : fireFoam solver

Table of Contents

This appendix will describe with more details the fireFoam solver. An emphasis will be done on the code created by FM Global.

fireFoam.C

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


    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    // * * * * * * * * * * * * * * * * * * * * * * * * * * *//

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

    while (runTime.run())
    {
        #include "readPISOControls.H"
        #include "readTimeControls.H"
        #include "compressibleCourantNo.H"
        #include "setDeltaT.H"

        #include "readLowMachControls.H"

        #include "readMixingControls.H"
        #include "readMultivarMULEControls.H"

        runTime++;
        Info<< "Time = " << runTime.timeName() << nl <<
endl;
```

Main function

Format the input arguments and set the root case
Create the run time
Create the mesh
Read the gravity acceleration value and direction
Declare and the continuity errors
Create the fields used by the solver (enthalpy, composition (species & ignition parameter), pressure, density, velocity,...) and read the properties described in the fireFoamPropertiesDict
Configure the time step (deltaT max, Courant number max,...)
Compute the Courant number

Time loop

Read the parameter for the PISO solver (fvSolution file)
Configure the time step (deltaT max, Courant number max,...)
Compute the Courant number
Compute the time step

Read the flag to choose between low Mach or fully compressible model (fvSolution file)
Read the parameters for the mixing model (fvSolution file)
Read the flags in fvSolution file to determine if the MULES algorithm and/or the multivariate algorithm have to be used
Increment the run time by the deltaT
Write the new run time in the log

```
        #include "computeB.H"
/*********** Content of computeB.H ******************\
   if (ignited)
   {
     b=b-(1.0/ignRampTime)*runTime.deltaT().value();
     b.max(0);
   }
\************************************************/
        // Pressure-velocity PIMPLE corrector loop
        for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
        {
            #include "rhoEqn.H"
   /*********** Content of rhoEqn.H *****************\
   solve(fvm::ddt(rho) + fvc::div(phi));
\************************************************/
            #include "UEqn.H"
/*********** Content of UEqn.H ***********************\
       // Solve the Momentum equation
       fvVectorMatrix UEqn
       (
           fvm::ddt(rho, U)
         + fvm::div(phi, U)
         + turbulence->divDevRhoReff(U)
       );

       UEqn.relax();

       if (oCorr == nOuterCorr - 1)
       {
           solve(UEqn == -fvc::grad(pd) - fvc::grad(rho)*gh,
   mesh.solver("UFinal"));
       }
       else
       {
           solve(UEqn == -fvc::grad(pd) -
   fvc::grad(rho)*gh);
       }
```

If combustion is present in the flow

Decrease the regress variable following the ignition ramp time

Set the negative value of the regress variable to zero (because that variable has to be in the interval [0, 1])

Execute the PIMPLE corrector nOuterCorr times. OpenFOAM recommends a value of one for that parameter.

Solve the continuity equation $\frac{\partial \rho}{\partial t}+\nabla\cdot(\rho U)=0$

Create a part of the momentum equation

$\frac{\partial \rho U}{\partial t}+\nabla\cdot(\rho U*U)+\nabla dev(\rho R_{eff})$ where devRhoReff is the effective stress tensor (laminar and Reynolds stress)

Relax the first part of the momentum equation

Check if the last iteration is reached

If true the resolution of the momentum equation is realized with the solver specified for the final iteration (fvSolution file : Ufinal solver)

$\frac{\partial \rho U}{\partial t}+\nabla\cdot(\rho U*U)+\nabla dev(\rho R_{eff})=-\nabla(p_d)-\nabla(\rho)\,g\,h$

If false the classical solver is used to solve the momentum equation (fvSolution file : U solver)

Solve the equation for the mixture fraction and its variance (more

```
\*****************************************************/
            #include "ftEqn.H"
            #include "hEqn.H"


            // --- PISO loop
            for (int corr=0; corr<nCorr; corr++)
            {
                #include "pEqn.H"
            }
        }

        turbulence->correct();

        rho = thermo.rho();

        runTime.write();

        #include "infoOutput.H"

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

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

    return 0;
}
```

details on this resolution later in this appendix)

Solve the energy equation (total enthalpy equation) (more details are presented later in this appendix)

pressure-implicit split-operator loop

Run the pressure equation nCorr times. OpenFOAM recommends a value superior to 1 but not more than 4. (Here 2)

Correct the pressure and the velocity fields (more details later in this appendix)

Solve the k equation and correct the subgride-scale fields

Compute the density using the thermodynamic model (here perfect gas)

Write the fields if required by the output

Write information on the log file

End of the solver

ftEqnSubCycle.H

```
tmp<fv::convectionScheme<scalar> > mvConvection
(
    fv::convectionScheme<scalar>::New
    (
        mesh,
        fields,
        phi,
        mesh.divScheme("div(phi,ft_b_h)")
    )
);
```

Create the convection term for the mixture fraction

```
if (composition.contains("ft"))
{
    if (isMULES)
    {
        MULES::explicitSolve
        (
            rho,
            ft,
            phi,
            fvc::flux
            (
                phi,
                ft
            )(),
            zeroField(),
            zeroField(),
            1, 0
        );
        solve
        (
            fvm::ddt(rho, ft) - fvc::ddt(rho, ft)
          - fvm::laplacian(turbulence->alphaEff(), ft)
        );
```

If the mixture fraction is one of the field

If the MULES algorithm is chosen

Resolve the convective part of the mixture fraction equation

Correct the mixture fraction for the diffusive part

$$\left.\frac{\partial \rho f_t}{\partial t}\right|_{n+1} = \left.\frac{\partial \rho f_t}{\partial t}\right|_{n} + \nabla^2(\rho a_{eff} f_t) \quad \text{(Hyp : Lewis = 1)}$$

```
    }
    else
    {
        if (multiVariate)
        {
            solve
            (
                fvm::ddt(rho, ft)
              + mvConvection->fvmDiv(phi, ft)
              - fvm::laplacian(turbulence->alphaEff(), ft)

            );
        }
        else
        {
            solve
            (
                fvm::ddt(rho, ft)
              + fvm::div(phi, ft)
              - fvm::laplacian(turbulence->alphaEff(), ft)

            );
        }
    }

    Info<< "max(ft) = " << max(ft).value() << endl;
    Info<< "min(ft) = " << min(ft).value() << endl;

}
forAll(ft01,celli)
{
    if (ft[celli] >= ftSt)
    {
        ft01[celli] = 1;
    }
    else
    {
```

If the MULES algorithm is not chosen

If the multivariate algorithm is chosen

Solve the mixture fraction equation

$$\frac{\partial \rho f_t}{\partial t} + \nabla \cdot (\rho U f_t) - \nabla^2 (\rho a_{eff} f_t) = 0$$ (Hyp : Lewis = 1)

If the MULES and the multivariate algorithms are not chosen

Solve the mixture fraction equation

$$\frac{\partial \rho f_t}{\partial t} + \nabla \cdot (\rho U f_t) - \nabla^2 (\rho a_{eff} f_t) = 0$$ (Hyp : Lewis = 1)

Write the max value of the mixture fraction in the log
Write the min value of the mixture fraction in the log

Create the value for the ft01 field

If the mixture fraction is equal or superior to the stoichiometric mixture fraction
The field ft01 is equal to 1

```
        ft01[celli] = 0;
    }
}
#       include "ftVarEqn.H"
/************** content of ftVarEqn.H ****************\
if (composition.contains("ftVar"))
{
    volScalarField& ftVar = composition.Y("ftVar");

    if (ftVarModel == "LESsimilarity")
    {
        ftVar = LESsimilarityC*turbulence-
>delta()*turbulence->delta()
        *magSqr(fvc::grad(ft));
    }
    else if (ftVarModel == "lengthScale")
    {
        ftVar = 1.0/lengthScaleC*turbulence->k()/turbulence-
>epsilon()
        *turbulence->alphaEff()/rho*magSqr(fvc::grad(ft));
    }
    else if (ftVarModel == "transport")
    {
        if (multiVariate&&!isMULES)
        {
            solve
            (
                fvm::ddt(rho, ftVar)
              + mvConvection->fvmDiv(phi, ftVar)
              - fvm::laplacian(turbulence->alphaEff(),
ftVar)
              ==
                2.*turbulence->alphaEff()
*magSqr(fvc::grad(ft))- lengthScaleC
*fvm::SuSp(2.0*rho*turbulence->epsilon()/turbulence->k()
,ftVar)
            );
```

Else ft01 is equal to 0

/************** Solve the mixture fraction variance field ************\

If the mixture fraction variance field is present

Create a link to the mixture fraction variance field

If the model chosen is LESsimilarity

$$\sigma_{ft} = constant\,(characteristic\,distance)^2\,\|\nabla f_t\|^2$$

If the model chosen is lengthScale

$$\sigma_{ft} = k_{Sgs} / constant\,\epsilon_{Sgs}\|\nabla f_t\|^2$$

If the model chosen is transport

If the multivariate algorithm is chosen but not the MULES one

solve the equation for the mixture fraction variance

$$\frac{\partial \rho \sigma_{ft}}{\partial t} + \nabla\cdot(\rho U \sigma_{ft}) - \nabla^2(\rho a_{eff}\sigma_{ft}) = 2\rho a_{eff}\|\nabla f_t\|^2 - constant\,SuSp(2\rho\frac{\epsilon}{k})$$

```
        }
        else
        {
            solve
            (
                fvm::ddt(rho, ftVar)
              + fvm::div(phi, ftVar)
              - fvm::laplacian(turbulence->alphaEff(),
ftVar)
              ==
                2.*turbulence-
>alphaEff()*magSqr(fvc::grad(ft))
              - lengthScaleC*fvm::SuSp(2.0*rho*turbulence-
>epsilon()/
                turbulence->k(),ftVar)
            );
        }
    }
    else
    {
        FatalError
            << args.executable() << " : Unknown ftVar model
" << ftVarModel
            << abort(FatalError);
    }

    Info<< "max(ftVar) = " << max(ftVar).value() << endl;
    Info<< "min(ftVar) = " << min(ftVar).value() << endl;
}

\*****************************************************/
```

Else

Solve the equation for the mixture fraction variance

$$\frac{\partial \rho \sigma_{ft}}{\partial t} + \nabla \cdot (\rho U \sigma_{ft}) - \nabla^2 (\rho a_{eff} \sigma_{ft}) = 2 \rho a_{eff} \| \nabla f_t \|^2 - constant\ SuSp (2 \rho \frac{\epsilon}{k})$$

If another model is specified, an error occurs

Write the maximum of the mixture fraction variance in the log
Write the minimum of the mixture fraction variance in the log

hEqn.H

```
{
  if (multiVariate&&!isMULES)
  {

    fvScalarMatrix hEqn
    (
        fvm::ddt(rho, h)
      + mvConvection->fvmDiv(phi,h)
      - fvm::laplacian(turbulence->alphaEff(), h)
     ==
        dpdt
      +
fvc::div(phi/fvc::interpolate(rho)*fvc::interpolate(p))
      - p*fvc::div(phi/fvc::interpolate(rho))
      + radiation->Sh(thermo)
//      - qrflamelet      // PC Added this line
    );

    hEqn.relax();
    hEqn.solve();
  }
  else
  {
    fvScalarMatrix hEqn
    (
        fvm::ddt(rho, h)
      + fvm::div(phi, h)
      - fvm::laplacian(turbulence->alphaEff(), h)
//alphaEff defined in ftEqn.H
     ==
        dpdt
      +
fvc::div(phi/fvc::interpolate(rho)*fvc::interpolate(p))
      - p*fvc::div(phi/fvc::interpolate(rho))
      + radiation->Sh(thermo)
```

/********** Solution for the energy equation – total enthalpy form ******\

If the multivariate algorithm is selected but not the MULES one

Create the total enthalpy equation

$$\frac{\partial \rho h}{\partial t}+\nabla\cdot(\rho U h)-\nabla^2(\rho a_{eff} h) = \frac{\partial p}{\partial t}+\nabla\cdot(U p)-p\nabla\cdot(U)+S_{radiation}$$

Add the radiation term

Relax the equation
Solve the equation

Else

Create the total enthalpy equation

$$\frac{\partial \rho h}{\partial t}+\nabla\cdot(\rho U h)-\nabla^2(\rho a_{eff} h) = \frac{\partial p}{\partial t}+\nabla\cdot(U p)-p\nabla\cdot(U)+S_{radiation}$$

Radiation term

```
      // - qrflamelet      // PC Added this line
    );
    hEqn.relax();                                          Relax the equation
    hEqn.solve();                                          Solve the equation
  }
# include "computeHRR.H"                                   Compute the heat release rate
/************ Content of computeHRR.H *********************\
  hc = thermo.hc();                                        Get the chemical enthalpy
  hs = thermo.hs();                                        Get the sensible enthalpy
  if (multiVariate&&!isMULES)                              If the algorithm is multivariate without MULES
  {
      HRR =                                                Compute the heat release rate
          -  fvc::ddt(rho, hc)
          - mvConvection->fvcDiv(phi, hc)
        + fvc::laplacian(turbulence->alphaEff(), hc);
      hrr=HRR;
      outHRR <<  runTime.value() << "  "                   Write the integral of the heat release rate in the log
             <<  fvc::domainIntegrate(HRR).value() << "  "
             <<  endl;
  }
  else                                                     Else
  {
      HRR =                                                Compute the heat release rate
          -  fvc::ddt(rho, hc)
          - fvc::div(phi, hc)
          + fvc::laplacian(turbulence->alphaEff(), hc);

      outHRR <<  runTime.value() << "  "                   Write the integral of the heat release rate in the log
             <<  fvc::domainIntegrate(HRR).value() << "  "
             <<  endl;
  }
\*********************************************************/
  thermo.correct();                                        Correct the temperature, the enthalpy fields but also the physical properties like the viscosity.

  radiation->correct();                                    Correct the radiation effect
}
```

$$HRR = \frac{-\partial \rho h_c}{\partial t} - \nabla \cdot (\rho U h_c) + \nabla^2 (\rho a_{eff} h_c)$$

$$HRR = \frac{-\partial \rho h_c}{\partial t} - \nabla \cdot (\rho U h_c) + \nabla^2 (\rho a_{eff} h_c)$$

pEqn.H

```
bool closedVolume = false;
```
Set the flag closedVolume to false

```
rho = thermo.rho();
```
Compute the density using the thermodynamic model (for example the perfect gas model)

```
volScalarField rUA = 1.0/UEqn.A();
```
Extract the central coefficients from the discretization of the momentum equation

```
U = rUA*UEqn.H();
```
Compute the velocity field

```
phi =
    fvc::interpolate(rho)
   *(
       (fvc::interpolate(U) & mesh.Sf())
     + fvc::ddtPhiCorr(rUA, rho, U, phi)
    );
```
Compute the mass flux in the mesh

```
phi -=
fvc::interpolate(rUA*rho*gh)*fvc::snGrad(rho)*mesh.magSf();

for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
{
```
Correct the dynamic pressure for the non orthogonality of the faces

```
    fvScalarMatrix pdEqn
    (
        fvc::ddt(psi)*pRef
      + fvc::ddt(psi, rho)*gh
      + fvc::div(phi)
      - fvm::laplacian(rho*rUA, pd)
    );
```
Create the dynamic pressure equation

```
    if (!lowMach)
    {
        pdEqn += fvm::ddt(psi,pd);
    }
```
Complete the dynamic pressure equation depending on the compressible model used

```
    closedVolume = pd.needReference();

    if (corr == nCorr-1 && nonOrth == nNonOrthCorr)
```
Choose the solver depending on the current iteration.

```
    {
        pdEqn.solve(mesh.solver(pd.name() + "Final"));
    }
    else
    {
        pdEqn.solve(mesh.solver(pd.name()));
    }
    if (nonOrth == nNonOrthCorr)
    {
        phi += pdEqn.flux();
    }
}
if (lowMach)
{
  p = rho*gh + pRef;
}
else
{
  p = pd + rho*gh + pRef;
}
dpdt = fvc::ddt(p);

#include "rhoEqn.H"
#include "compressibleContinuityErrs.H"
U -= rUA*(fvc::grad(pd) + fvc::grad(rho)*gh);

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

Solver for the final iteration

Solver for the others iteration

Correction on the mass flux at the last iteration

Compute the pressure field depending on the compressible model

p = the pressure if the lowMach model is chosen

p = the total pressure if the fully compressible model is chosen

compute the temporal derivation of the pressure

Solve the continuity equation to find rho

Compute the continuity errors

Correct the velocity field with the new values of the dynamic pressure and the density

Correct the velocity field to respect the boundary conditions

Correct the pressure field

Compute the density using the thermodynamic model

Appendix B : New model for the boundary conditions

Table of Contents

This appendix describes the boundary conditions that implement in fireFoam : the model proposed by Hölling.

In addition to that in the file alphaSgsBuoyantWallFunctionFvPatchScalarField.C the unified law is described in comment.

To install those boundary conditions, refer you to the installation of fireFoam.

The appendix D provide an example of the use of those boundary conditions.

N.B. : alpha will be called thermal diffusivity in the comments. But you have to keep in mind that in fact it is the thermal diffusivity multiplied by the density.

alphaSgs

alphaSgsBuoyantWallFunctionFvPatchScalarField.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    alphaSgsBuoyantWallFunctionFvPatchScalarField

Description
    Buoyant Thermal wall function for turbulent thermal diffusivity based on
    "Asymptotic analysis of the near-wall region of turbulence natural
    convection flows"
    Holling and Herwig. J. Fluid Mech (2005), vol 541, pp 383-397

SourceFiles
    alphaSgsJayatillekeWallFunctionFvPatchScalarField.C

\*---------------------------------------------------------------------------*/

#ifndef alphaSgsBuoyantWallFunctionFvPatchScalarField_H
#define alphaSgsBuoyantWallFunctionFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace compressible
{
```

```
namespace LESModels
{
/*---------------------------------------------------------------------------*\
     Class alphaSgsBuoyantWallFunctionFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/
class alphaSgsBuoyantWallFunctionFvPatchScalarField
:
      // This boundary condition is a particular case of a fixedValue condition
    public fixedValueFvPatchScalarField
{
    // Private data
        static scalar maxExp_;
        static scalar tolerance_;
        static label maxIters_;

        //- Thermal expansion coefficient
        scalar beta_;

        //- gravity magnitude
        scalar magG_;

    // Private member functions
        //- Check the type of the patch
        void checkType();

        //- Read
        //void read();
public:
    //- Runtime type information
    TypeName("alphaSgsBuoyantWallFunction");

    // Constructors
        //- Construct from patch and internal field
        alphaSgsBuoyantWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );
        //- Construct from patch, internal field and dictionary
        alphaSgsBuoyantWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );
        //- Construct by mapping given an
        //  alphaSgsBuoyantWallFunctionFvPatchScalarField
        //  onto a new patch
        alphaSgsBuoyantWallFunctionFvPatchScalarField
        (
            const alphaSgsBuoyantWallFunctionFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
```

```
            const fvPatchFieldMapper&
        );
        //- Construct as copy
        alphaSgsBuoyantWallFunctionFvPatchScalarField
        (
            const alphaSgsBuoyantWallFunctionFvPatchScalarField&
        );
        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new alphaSgsBuoyantWallFunctionFvPatchScalarField(*this)
            );
        }
        //- Construct as copy setting internal field reference
        alphaSgsBuoyantWallFunctionFvPatchScalarField
        (
            const alphaSgsBuoyantWallFunctionFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );
        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new alphaSgsBuoyantWallFunctionFvPatchScalarField
                (
                    *this,
                    iF
                )
            );
        }
    // Member functions
        // Evaluation functions

            //- Evaluate the patchField (it is the key function)
            virtual void evaluate
            (
                const Pstream::commsTypes commsType=Pstream::Pstream::blocking
            );
};
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace LESModels
} // End namespace compressible
} // End namespace Foam
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif
// ************************************************************************* //
```

alphaSgsBuoyantWallFunctionFvPatchScalarField.C

```
#include "alphaSgsBuoyantWallFunctionFvPatchScalarField.H"
#include "LESModel.H"
#include "basicThermo.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "addToRunTimeSelectionTable.H"
#include "wallFvPatch.H"
#include "muSgsBuoyantWallFunctionFvPatchScalarField.H"

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

namespace Foam
{
namespace compressible
{
namespace LESModels
{

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

scalar alphaSgsBuoyantWallFunctionFvPatchScalarField::maxExp_ = 50.0;
scalar alphaSgsBuoyantWallFunctionFvPatchScalarField::tolerance_ = 0.01;
label alphaSgsBuoyantWallFunctionFvPatchScalarField::maxIters_ = 10;

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

void alphaSgsBuoyantWallFunctionFvPatchScalarField::checkType()
{
    if (!isA<wallFvPatch>(patch()))
    {
        FatalErrorIn
        (
            "alphaSgsBuoyantWallFunctionFvPatchScalarField::checkType()"
        )
            << "Patch type for patch " << patch().name() << " must be wall\n"
            << "Current patch type is " << patch().type() << nl
            << exit(FatalError);
    }
}


// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * //


alphaSgsBuoyantWallFunctionFvPatchScalarField::
alphaSgsBuoyantWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
```

```
:
    fixedValueFvPatchScalarField(p, iF),
    beta_(1.0/293.0),
    magG_(9.80665)
{
    checkType();
    //read();
}


alphaSgsBuoyantWallFunctionFvPatchScalarField::
alphaSgsBuoyantWallFunctionFvPatchScalarField
(
    const alphaSgsBuoyantWallFunctionFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),
    beta_(ptf.beta_),
    magG_(ptf.magG_)
{}


alphaSgsBuoyantWallFunctionFvPatchScalarField::
alphaSgsBuoyantWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict),
    beta_(dict.lookupOrDefault<scalar>("beta", 1.0/293.0)),
    magG_(dict.lookupOrDefault<scalar>("magG", 9.80665))
{
    checkType();
}


alphaSgsBuoyantWallFunctionFvPatchScalarField::
alphaSgsBuoyantWallFunctionFvPatchScalarField
(
    const alphaSgsBuoyantWallFunctionFvPatchScalarField& tppsf
)
:
    fixedValueFvPatchScalarField(tppsf),
    beta_(tppsf.beta_),
    magG_(tppsf.magG_)
{
    checkType();
}
```

```
alphaSgsBuoyantWallFunctionFvPatchScalarField::
alphaSgsBuoyantWallFunctionFvPatchScalarField
(
    const alphaSgsBuoyantWallFunctionFvPatchScalarField& tppsf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(tppsf, iF),
    beta_(tppsf.beta_),
    magG_(tppsf.magG_)
{
    checkType();
}

// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
// Here is the key function that compute the new values at the boundary.
void alphaSgsBuoyantWallFunctionFvPatchScalarField::evaluate
(
    const Pstream::commsTypes
)
{
    // Get info from the SGS model (the db() variable is the structure that have
    // access to all the parameters used in the simulation)
    const LESModel& sgs = db().lookupObject<LESModel>("LESProperties");
    // Get the thermophysical properties
    const basicThermo& thermo = db().lookupObject<basicThermo>
    (
        "thermophysicalProperties"
    );

// Field data
    // Get the index of the boundary among the patch
    const label patchI = patch().index();

    // Get the “thermal diffusivity” on the boundary
    const scalarField& alphaw = sgs.alpha().boundaryField()[patchI];
    // Get the “thermal diffusivity” of the subgride scale on the boundary
    scalarField& alphaSgsw = *this;
    // Get the density on the boundary
    const scalarField& rhow = sgs.rho().boundaryField()[patchI];
    // Get the temperature on the boundary
    const fvPatchScalarField& Tw = thermo.T().boundaryField()[patchI];
    // Get the specific heat capacity on the boundary
    const scalarField Cpw = thermo.Cp(Tw, patchI);
    // Get the value of the temperature at the first node off the wall
    const scalarField T = Tw.patchInternalField();
    // Computed the magnitude of the temperature gradient at the boundary. And
    // avoid it to be zero (to avoid trouble later during division)
    const scalarField magGradTw = max(mag(Tw.snGrad()), VSMALL);
    // Get this inverse of the distance between the first node off wall and the
    // wall.
    const scalarField& ry = patch().deltaCoeffs();
```

```
    // Populate boundary values
    forAll(alphaSgsw, faceI)
    {
        // Initial value from the previous iteration
        scalar qw =  (alphaw[faceI]+alphaSgsw[faceI])*Cpw[faceI]*magGradTw[faceI];

        scalar gBeta = pow(magG_*beta_,0.25);
        scalar aCoef = (Tw[faceI]-T[faceI])
                     * gBeta
                     * pow(alphaw[faceI]*sqr(rhow[faceI])*pow(Cpw[faceI],3),0.25);
        scalar bCoef = 1.0 / ry[faceI]
                     * gBeta
                     * sqrt(rhow[faceI])
                     / pow(Cpw[faceI]*pow(alphaw[faceI],3),0.25);

        label iter = 0;
        scalar err = GREAT;

      // Use the model to compute the real heat flux using a Newton-Raphson method
        do
        {
              scalar f =
                      aCoef*pow(qw, -0.75)
                    - 0.427*log(bCoef*pow(qw,0.25))
                    - 1.93;

              scalar df =
                    - 0.75*aCoef*pow(qw,-1.75)
                    - 0.427/4.0/qw;

/*          // Implementation of the unified law
            scalar aCQw = aCoef/0.427*pow(qw,-0.75);
            scalar faCQw = exp(aCQw)-1.0-aCQw-sqr(aCQw)/2.0-pow(aCQw,3.0)/6.0
-pow(aCQw,4.0)/24.0;

            scalar f = aCoef*pow(qw, -0.75) + exp(-1.93/0.427) * (faCQw -
pow(aCQw,5.0)/120.0) - bCoef * pow(qw, 0.25);

            scalar df = -0.75 * aCoef * pow(qw,-1.75) -0.75 *aCQw/qw * exp(-
1.93/0.427) * faCQw - 0.25 * bCoef * pow(qw, -0.75);
*/

            scalar qwNew = qw - f/df;
            err = mag((qw - qwNew)/qw);
            qw = qwNew;

      //Stop the iteration if the tolerance is reached or the maximum of iteration
      //or if the flux is very small.
        } while (qw > VSMALL && err > tolerance_ && ++iter < maxIters_);

      //Compute the effective thermal diffusivity
        scalar alphaEff = qw/Cpw[faceI]/magGradTw[faceI];
```

```
      //Compute the subgride-scale thermal diffusivity needed to correct the
      //temperature gradient. Avoid the value to be negative.
        alphaSgsw[faceI] = max(0.0, alphaEff - alphaw[faceI]);

        if (debug)
        {
            Info<< "    alphaEff       = " << alphaEff << nl
                << "    alphaw         = " << alphaw[faceI] << nl
                << "    alphaSgsw      = " << alphaSgsw[faceI] << nl
                << "    Tw             = " << Tw[faceI] << nl
                << "    T              = "  << T[faceI] << nl
                << "    ry             = "  << ry[faceI] << nl
                << "    magGradTw      = "  << magGradTw[faceI] << nl
                << "    qw             = "  << qw << nl
                << endl;
        }

    }

//  Grab the muSgs patch field using generic/base type
    const fvPatchScalarField& muSgsPatchField =
       patch().lookupPatchField<volScalarField, scalar>("muSgs");

    // Perform the type checking
    if (!isA<muSgsBuoyantWallFunctionFvPatchScalarField>(muSgsPatchField))
    {
        FatalErrorIn("alphaSgsBuoyantWallFunctionFvPatchScalarField::evaluate()")
            << "Invalid boundary condition for muSgs" << nl
            << "use muSgsBuoyantWallFunction" << nl
            << endl << abort(FatalError);
    }

    const muSgsBuoyantWallFunctionFvPatchScalarField& muSgsw =
       refCast<const muSgsBuoyantWallFunctionFvPatchScalarField>(muSgsPatchField);
    muSgsBuoyantWallFunctionFvPatchScalarField& muSgs =
       const_cast<muSgsBuoyantWallFunctionFvPatchScalarField&>(muSgsw);

// Compute the correction on muSgs just after the correction on alphaSgs because
// the corrected gradient is needed for that
    muSgs.evaluateInAlphaSgs();
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
makePatchTypeField
(
    fvPatchScalarField,
    alphaSgsBuoyantWallFunctionFvPatchScalarField
);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace LESModels
} // End namespace compressible
} // End namespace Foam
// ************************************************************************* //
```

muSgs

muSgsBuoyantWallFunctionFvPatchScalarField.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    muSgsBuoyantWallFunctionFvPatchScalarField

Description
    Buoyant Thermal wall function for turbulent diffusivity based on
    "Asymptotic analysis of the near-wall region of turbulence natural
    convection flows"

    Holling and Herwig. J. Fluid Mech (2005), vol 541, pp 383-397

SourceFiles
    muSgsBuoyantWallFunctionFvPatchScalarField.C
\*---------------------------------------------------------------------------*/
#ifndef muSgsBuoyantWallFunctionFvPatchScalarField_H
#define muSgsBuoyantWallFunctionFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace compressible
{
namespace LESModels
```

```
{
/*---------------------------------------------------------------------------*\
     Class muSgsBuoyantWallFunctionFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/
class muSgsBuoyantWallFunctionFvPatchScalarField
:
// This boundary condition is a particular case of a fixed-value boundary condition
    public fixedValueFvPatchScalarField
{
    // Private data
        //- Thermal expansion coefficient
        scalar beta_;
        //- gravity magnitude
        scalar magG_;
        //- reference Temperature
        scalar Tref_;
        //- Turbulent Prandtl number
        scalar Prt_;

    // Private member functions
        //- Check the type of the patch
        void checkType();
public:
    //- Runtime type information
    TypeName("muSgsBuoyantWallFunction");
    // Constructors
        //- Construct from patch and internal field
        muSgsBuoyantWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );
        //- Construct from patch, internal field and dictionary
        muSgsBuoyantWallFunctionFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );
        //- Construct by mapping given an
        //  muSgsBuoyantWallFunctionFvPatchScalarField
        //  onto a new patch
        muSgsBuoyantWallFunctionFvPatchScalarField
        (
            const muSgsBuoyantWallFunctionFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );
        //- Construct as copy
        muSgsBuoyantWallFunctionFvPatchScalarField
        (
            const muSgsBuoyantWallFunctionFvPatchScalarField&
```

```
        );
        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new muSgsBuoyantWallFunctionFvPatchScalarField(*this)
            );
        }
        //- Construct as copy setting internal field reference
        muSgsBuoyantWallFunctionFvPatchScalarField
        (
            const muSgsBuoyantWallFunctionFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );
        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new muSgsBuoyantWallFunctionFvPatchScalarField
                (
                    *this,
                    iF
                )
            );
        }
    // Member functions
        // Evaluation functions
            //- Evaluate the patchField (dummy one)
      // as the correction on the temperature gradient have to be computed first
      // this function does exceptionnally nothing.
            virtual void evaluate
            (
                const Pstream::commsTypes commsType=Pstream::Pstream::blocking
            );
            //- Evaluate the patchField - function in which the Holling model is
            // implemented
            void evaluateInAlphaSgs
            (
                const Pstream::commsTypes commsType=Pstream::Pstream::blocking
            );
};
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace LESModels
} // End namespace compressible
} // End namespace Foam
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif
// ************************************************************************* //
```

muSgsBuoyantWallFunctionFvPatchScalarField.C

```
#include "muSgsBuoyantWallFunctionFvPatchScalarField.H"
#include "LESModel.H"
#include "basicThermo.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "addToRunTimeSelectionTable.H"
#include "wallFvPatch.H"

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

namespace Foam
{
namespace compressible
{
namespace LESModels
{

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


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

void muSgsBuoyantWallFunctionFvPatchScalarField::checkType()
{
    if (!isA<wallFvPatch>(patch()))
    {
        FatalErrorIn
        (
            "muSgsBuoyantWallFunctionFvPatchScalarField::checkType()"
        )
            << "Patch type for patch " << patch().name() << " must be wall\n"
            << "Current patch type is " << patch().type() << nl
            << exit(FatalError);
    }
}

// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * //

muSgsBuoyantWallFunctionFvPatchScalarField::
muSgsBuoyantWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(p, iF),
    beta_(1.0/293.0),
    magG_(9.80665),
    Tref_(293.0),
    Prt_(0.9)
```

```
{
    checkType();
}


muSgsBuoyantWallFunctionFvPatchScalarField::
muSgsBuoyantWallFunctionFvPatchScalarField
(
    const muSgsBuoyantWallFunctionFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),
    beta_(ptf.beta_),
    magG_(ptf.magG_),
    Tref_(ptf.Tref_),
    Prt_(ptf.Prt_)
{}


muSgsBuoyantWallFunctionFvPatchScalarField::
muSgsBuoyantWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict),
    beta_(dict.lookupOrDefault<scalar>("beta", 1.0/293.0)),
    magG_(dict.lookupOrDefault<scalar>("magG", 9.80665)),
    Tref_(dict.lookupOrDefault<scalar>("Tref", 293.0)),
    Prt_(dict.lookupOrDefault<scalar>("Prt", 0.9))
{
    checkType();
}

muSgsBuoyantWallFunctionFvPatchScalarField::
muSgsBuoyantWallFunctionFvPatchScalarField
(
    const muSgsBuoyantWallFunctionFvPatchScalarField& tppsf
)
:
    fixedValueFvPatchScalarField(tppsf),
    beta_(tppsf.beta_),
    magG_(tppsf.magG_),
    Tref_(tppsf.Tref_),
    Prt_(tppsf.Prt_)
{
    checkType();
}
```

```
muSgsBuoyantWallFunctionFvPatchScalarField::
muSgsBuoyantWallFunctionFvPatchScalarField
(
    const muSgsBuoyantWallFunctionFvPatchScalarField& tppsf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(tppsf, iF),
    beta_(tppsf.beta_),
    magG_(tppsf.magG_),
    Tref_(tppsf.Tref_),
    Prt_(tppsf.Prt_)
{
    checkType();
}


// * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
void muSgsBuoyantWallFunctionFvPatchScalarField::evaluate //dummpy one
(
    const Pstream::commsTypes
)
{ // this function is empty as the correction is explicitly called after the
  //computation of the correction on alphaSgs
}

// The key function
void muSgsBuoyantWallFunctionFvPatchScalarField::evaluateInAlphaSgs
(
    const Pstream::commsTypes
)
{
    // Get info from the SGS model
    const LESModel& sgs = db().lookupObject<LESModel>("LESProperties");
    // Get the thermophysical properties
    const basicThermo& thermo = db().lookupObject<basicThermo>
    (
        "thermophysicalProperties"
    );
    // Wall function constants

    // Field data
    // Get the index of the boundary among all the patches
    const label patchI = patch().index();
    // Get the effective “thermal diffusivity” on the boundary
    const scalarField alphaEffw = sgs.alphaEff()().boundaryField()[patchI];
    // Get the subgride-scale “thermal diffusivity” on the boundary
    const scalarField alphaSgsw = sgs.alphaSgs()().boundaryField()[patchI];
    // Get the physical “thermal diffusivity” on the boundary
    const scalarField alphaw = sgs.alpha().boundaryField()[patchI];
    // Get the physical dynamic viscosity on the boundary
    const scalarField muw = sgs.mu().boundaryField()[patchI];
```

```
// link to the subgride-scale dynamic viscosity on the boundary
scalarField& muSgsw = *this;
// Get the density on the boundary
const scalarField& rhow = sgs.rho().boundaryField()[patchI];
// Get the velocity on the boundary
const fvPatchVectorField& Uw = sgs.U().boundaryField()[patchI];
// Get the velocity at the first node off the wall
const vectorField U  = Uw.patchInternalField();
// Compute the difference between the two previous velocity. And avoid to get
// zero as result (could be used in division). This difference allows to manage
// the case of a moving wall because in most of the case Uw will be (0 0 0).
const scalarField magUp = max(mag(U - Uw), VSMALL);
// Compute the velocity gradient normal to the boundary. And avoid it to be
// equal to zero.
const scalarField magFaceGradU = max(mag(Uw.snGrad()),SMALL);
// Get the temperature of the boundary
const fvPatchScalarField& Tw = thermo.T().boundaryField()[patchI];
// Get the temperature at the first off-wall node
const scalarField T = Tw.patchInternalField();
// Get the temperature gradient normal to the boundary. And avoid it to be zero
const scalarField magGradTw = max(mag(Tw.snGrad()),SMALL);
// Compute the characteristic temperature (cf. Holling model)
// N.B. : the gradient of the temperature is always multiplied by alphaEffw to
// used the corrected value computed in the alphaSgs boundary condition. It's
// the reason why sometimes there is a multiplication by alphaEff and a
// division by alpha. Indeed magGradTw_corrected = alphaEffw/alphaw magGradTw
const scalarField Tc =
    pow
    (
        rhow/alphaw/magG_/beta_ *
        pow
        (    alphaEffw * magGradTw / rhow,
             3
        ),
        0.25
    );
// Compute the characteristic dimension (cf. Holling model)
const scalarField delta =
    pow
    (
        pow(alphaw,3)/magG_/beta_/alphaEffw/magGradTw,
        0.25
    ) / sqrt(rhow);
// Get the inverse of the distance between the wall and the first off-wall node
const scalarField& ry = patch().deltaCoeffs();
// Compute the non-dimensional distance of the first off-wall node
const scalarField yPlus = (1.0/ry)/delta;
// Compute the non-dimensional at the reference temperature
const scalarField thetaO = (Tw - Tref_)/Tc;

// Populate boundary values
forAll(muSgsw, faceI)
{
```

```
        // Compute the Prandtl number
        scalar Pr = muw[faceI]/alphaw[faceI];

        scalar Prat = Pr/Prt_;

        scalar logYPlus = log(yPlus[faceI]);
        // Compute the thermal diffusivity
        scalar alpha = alphaw[faceI] / rhow[faceI];
        // Compute the velocity gradient normal to the boundary using the Holling
        // model
        scalar DuDy = 1.0/Pr*alpha/sqr(delta[faceI])*
               (
                  Pr*delta[faceI]/alpha*magUp[faceI]
                - 0.427*Prat*yPlus[faceI]
                * (  0.427*(logYPlus - 2.0)
                   + 1.93 - thetaO[faceI]
                  ) + 2.27*logYPlus - 1.28
                )/( 0.49*logYPlus+1.28);
        // Compute the effective dynamic viscosity
        scalar muEff = DuDy * muw[faceI] / magFaceGradU[faceI];
        // Compute the subgride-scale dynamic viscosity. Avoid the value to be
        // negative
        muSgsw[faceI] = max(0.0, muEff - muw[faceI]);

        if (debug)
        {
            Info<< "    muEff           = " << muEff << nl
                << "    muw             = " << muw[faceI] << nl
                << "    muSgsw          = " << muSgsw[faceI] << nl
                << "    yPlus           = " << yPlus[faceI] << nl
                << "    Tw              = " << Tw[faceI] << nl
                << "    T               = " << T[faceI] << nl
                << "    Tc              = " << Tc[faceI] << nl
                << "    delta           = " << delta[faceI] << nl
                << "    ry              = " << ry[faceI] << nl
                << "    magUp           = " << magUp[faceI] << nl
                << "    thetaO          = " << thetaO[faceI] << nl
                << endl;
        }
    }
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
makePatchTypeField
(
    fvPatchScalarField,
    muSgsBuoyantWallFunctionFvPatchScalarField
);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace LESModels
} // End namespace compressible
} // End namespace Foam
// ************************************************************************* //
```

Appendix C : Utilities - Source code

Table of Contents

This appendix describes the source code of all the utilities created during this intership.

Analyze log file

The following script could be used with *gnuplot* to draw the evolution of different parameters contained in the log file. To launch the script, write the following in the folder containing the script and the log file :

```
gnuplot <name of the script file>
```

The following example plots the initial residuals for *rho*, *Ux*, *Uy*, *Uz* and *h*.

`set logscale y`	Set the scale of y in logarithmic scale
`set title "Residuals"`	Set the title
`set ylabel 'Residual'`	Set the y label
`set xlabel 'Iteration'`	Set the x label
`plot "< cat log | grep 'Solving for rho'| cut -d' ' -f9 | tr -d ','" title 'rho' with lines, \`	Read the log file and plot the *rho* residuals
`"< cat log | grep 'Solving for Ux'| cut -d' ' -f9 | tr -d ','" title 'Ux' with lines, \`	Read the log file and plot the *Ux* residuals
`"< cat log | grep 'Solving for Uy'| cut -d' ' -f9 | tr -d ','" title 'Uy' with lines, \`	Read the log file and plot the *Uy* residuals
`"< cat log | grep 'Solving for Uz'| cut -d' ' -f9 | tr -d ','" title 'Uz' with lines, \`	Read the log file and plot the *Uz* residuals
`"< cat log | grep 'Solving for h'| cut -d' ' -f9 | tr -d ','" title 'h' with lines`	Read the log file and plot the *h* residuals
`pause 1`	Break the execution during 1s
`reread`	Start again this script

Here is the description of the reading process :

- *cat* reads the file log
- *grep* filters the file to keep only the lines in which 'Solving for …' appears
- *cut* keep only the 9th element of those lines. The separator is specified with the option d. Here it is the blank character.
- *tr* suppresses the character specified by the option d, here the coma.
- The *title* option defines the legend for the data
- *with lines*, it is the command to draw a line between the data instead of plotting a symbol for each value.

Using an IDE software with OpenFOAM

Here are two howto's to use Eclipse and Netbeans with OpenFOAM version 1.6. But first, here is an important warning on the use of an IDE software. By using an IDE software, you can easily navigate in the source code OpenFOAM. Unfortunately, that implies a high risk to modify the basic source code. To avoid that, you can change the permission of the directory `$FOAM_SRC` and all its subdirectories to "read-only". If you plan to work on some parts linked with a solver or a utility, it could be safer to do the same with `$FOAM_SOLVERS` and `$FOAM_UTILITIES`.

Eclipse

This howto describes the parameters for Eclipse Ganymede SR2 with OpenFOAM version 1.6 on Ubuntu LTS 8.04 edition 32 bits.

Initial remark : this howto uses the postprocessor "*ptot*" as example[30].

1. Install Eclipse IDE for C/C++ Developers
   a) Download the binaries on http://www.eclipse.org/downloads/
   b) Extract the files where you want (e.g. in `$HOME`)
2. Launch Eclipse from a terminal :
   ```
   $HOME/eclipse/eclipse &
   ```
3. At the first prompt, Eclipse ask you where is the default workspace. (By default `$HOME/workspace` but you can create another folder).
4. Create a C++ Project
   a) File->New->C++ Project
   b) Choose a name for your project (e.g. *pTotTest*)
   c) If you want to create an executable, choose Executable->Empty Project
   d) If you want to create a library, choose Shared Library or Static Library (not tested)
   e) Click on the button 'Finish'
5. Precise the libraries from OpenFOAM
   a) Open the properties of the project Project->Properties
   b) Go to C/C++ Build -> Settings -> Tool Settings
   c) Change 'Configuration' from Debug to [All configurations]
   d) Add the preprocessor instructions
      - GCC C++ Compiler -> Preprocessor -> Defined symbols (-D)
      - Click on the button 'Add...'
      - Write '`linux`'
      - Click on the button 'Ok'
      - Add also a symbol '`WM_DP`' (choose the double precision in OpenFOAM) and '`NoRepository`'
      - A list of symbols is created :
        ```
        linux
        WM_DP
        NoRepository
        ```

30 This tool is located in the following folder "OpenFOAM-1.6\applications\utilities\postProcessing\miscellaneous\ptot"

e) Add the include directories
   - GCC C++ Compiler -> Directories -> Include paths (-I)
   - Click on the button 'Add...'
   - Click on the button 'File system'
   - Find the directory `$FOAM_SRC/OSspecific/POSIX/lnInclude`
   - Do the same for the directories
     `$FOAM_SRC/OpenFOAM/lnInclude`
     `$FOAM_SRC/finiteVolume/lnInclude`
     ... the other directories you need
   - A list of directories is created with the ABSOLUTE path (don't use the enviromment variables to specify the directories)

f) Add the libraries search path
   - GCC C++ Linker -> Libraries -> Library search path (-L)
   - Click on the button 'Add...'
   - Click on the button 'File system'
   - Find the directory `$FOAM_LIB/linuxGccDPOpt`
   - Do the same for the directories
     ... the other libraries you need
   - A list of directories is created with the ABSOLUTE path (don't use the enviromment variables to specify the directories)

g) Add the libraries
   - GCC C++ Linker -> Libraries -> Libraries (-l)
   - Click on the button 'Add...'
   - Write 'OpenFOAM'
   - Add the other libraries ('m', 'dl','finiteVolume' and any other librairies you need)

h) Click on the button 'Ok' of the properties panel

6. Import your basic code in Eclipse
   a) Copy the utility ptot

```
cp -r $FOAM_UTILITIES/postProcessing/postProcessing/miscellaneous/ptot \$HOME
```

   b) Import the file in your project
      - File->Import...
      - General->File System
      - Click 'Next'
      - 'Browse...' to the `$HOME/ptot` directory
      - Check the case in front of the '*ptot.C*' file
      - Click 'Finish'

7. Assuming that the cavity case is solved, let execute `ptot` on that case
   a) Run->Debug configurations...
   b) On the right panel, in the 'Arguments' subpanel, write

```
-case <write here the absolute path to the cavity directory>
```

   c) Click 'Debug'
   d) Then your are inside the code

Final remark : You have to launch Eclipse from the terminal in which the settings of OpenFoam version 1.6 are sourced[31]. In the other hand the Debug Tool is not able to find general libraries.

NetBeans

This howto describes the parameters for NetBeans 6.7 with OpenFOAM version 1.6 on Ubuntu LTS 8.04 edition 32 bits.

Initial remark : this howto uses the postprocessor "*ptot*" as example.

1. Install NetBeans 6.7 C/C++
   a) Download the binaries on http://www.netbeans.org/downloads/index.html
   b) Run the script '*netbeans-6.7-ml-cpp-linux.sh*' in a terminal
2. Launch NetBeans from a terminal :
   ```
   $HOME/netbeans-6.5.1/bin/netbeans &
   ```
3. Create a C++ Project
   a) File->New Project
   b) C/C++ :
      If you want to create an executable, choose C/C++ Application
      If you want to create an library, choose C/C++ Dynamic Library or Static Library (not tested)
   c) Click on 'Next'
   d) Choose a name for your project (e.g. *ptotTest*)
   e) Click on the button 'Finish'
4. Precise the libraries from OpenFOAM
   a) Open the properties of the project
      Right click on the name of the project ->Properties
   b) Go to Build -> C++ Compiler
   c) Change 'Configuration' from Debug (active) to <All configurations>
   d) Add the preprocessor instructions
      - General -> Preprocessor Definitions
      - Click on the button '…'
      - Click on the button 'Add'

31 Do that by writing the following line in the terminal if it is not parametrized in the `$HOME/.bashrc` file :
```
. $HOME/OpenFOAM/OpenFOAM-1.6/etc/bashrc
```

- Write '`linux`'
- Click on the button 'Ok'
- Add also a symbol '`WM_DP`' (choose the double precision in OpenFOAM) and '`NoRepository`'
- A list of symbols is created :

```
linux
WM_DP
NoRepository
```

- Click on the button 'OK'

e) Add the include directories

- General -> Include Directories
- Click on the button '…'
- Click on the button 'Add'
- Find the directory `$FOAM_SRC/OSspecific/POSIX/lnInclude`
- Do the same for the directories

```
$FOAM_SRC/OpenFOAM/lnInclude
$FOAM_SRC/finiteVolume/lnInclude
```

... the other directories you need

- A list of directories is created with the ABSOLUTE path (don't use the environment variables to specify the directories)
- Click on the button 'OK'

f) Add the libraries search path

- Build -> Linker
- General -> Additional Library Directories
- Click on the button '…'
- Click on the button 'Add'
- Find the directory `$FOAM_LIB/linuxGccDPOpt`
- Do the same for the directories

... the other libraries you need

- A list of directories is created with the ABSOLUTE path (don't use the enviromment variables to specify the directories)
- Click on the button 'OK'

g) Add the libraries

- Libraries -> Libraries
- Click on the button '…'
- Click on the button 'Add Library...'
- Go in the directory `$FOAM_LIB/linuxGccDPOpt`
- Change 'Files of Type' from 'Static Library (.a)' to 'Dynamic Library (.so)'
- Select '*libOpenFOAM.so*'
- Add the other libraries ('*finiteVolume*' and any other libraries you need)
- Click on 'Add Standard Library...'
- Choose 'Dynamic Linking'
- Click on the button 'OK'
- Do the same for the standard libraries 'Mathematics'
- Click on the button 'OK'
- Click on the button 'OK' of the properties panel

5. Import your basic code in NetBeans

a) Copy the utility ptot in your project directory

```
cp -r $FOAM_UTILITIES/postProcessing/postProcessing/miscellaneous/ptot \
$HOME/NetBeansProjects/ptotTest
```

b) Import the file in your project

- On the top left panel, make a rigth click on the folder 'Source Files'
- Choose 'Add existing item...'
- Select the '*ptot.C*' file

6. Assuming that the cavity case is solved, let execute ptot on that case

a) Open the panel of the project properties

b) In the 'Run' section, modify the 'General -> Arguments' by writing

-case <write here the absolute path to the cavity directory>

c) Click 'OK'

d) Right click on the project name and choose 'Step into'

e) Then your are inside the code

Final remark : You have to launch NetBeans from the terminal, in the other hand the Debug Tool is not able to find general libraries.

Post-processors

Average in a homogeneous direction

First solution

This first solution extract the profile of the average field at specified heights. The second solution is better than this one.

profileWall.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2008 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    postWall

Description
    Post-processes data from wall burner case.

    Use the mean field to compute the mean profile in the Xi direction (homogenous
    direction for the analyzed case).

    The process compute the average of the
    field along the x direction for each specified heights from the start point
    (the specified heights have to be parameter in the ascendant order in the
    'profileWallDict')

    !! This tool accepts only scalar fields. So if you want to compute the
    average of a velocity component, use first the tool 'foamCalc components U'.
```

```
    @author Frederic Collonval
    @email  fcollonv@umd.edu
    @version    2009_07_17
\*---------------------------------------------------------------------------*/

#include <string>
#include "fvCFD.H"
#include "wallIndex.H"
#include "makeGraph.H"

#include "OSspecific.H"


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
//  Main program:

int main(int argc, char *argv[])
{
    argList::noParallel();    // Forbid the parallel option (-parallel)
#   include "addTimeOptions.H" // Read the time option (-time or -latestTime)
#   include "setRootCase.H"   // Set the root case (option -case)

#   include "createTime.H"    // Get the time line

    // Get times list
    instantList Times = runTime.times();

    // set startTime and endTime depending on -time and -latestTime options
#   include "checkTimeOptions.H"
    // set the current time to startTime
    runTime.setTime(Times[startTime], startTime);

#   include "createMesh.H"
    // Get the graph format define in the controlDict
    const word& gFormat = runTime.graphFormat();

    // Setup wall indexing for averaging over wall down to a line
    IOdictionary dict_
    (
        IOobject
        (
            "profileWallDict",
            mesh.time().constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );
    // Initialize the class used to compute the average field
    wallIndex wallIndexing(mesh);
    wordList *fieldsList_;
```

```
    if(dict_.found("fields"))
      {
            fieldsList_ = new wordList(dict_.lookup("fields"));
      }
      else
      {
            Info<< nl << "No field specified" << nl;
            FatalError.exit();
      }
    // Read the list of the heights specified by the user [optional]
    scalarList *heightsList_;

    if(dict_.found("heights"))
    {
        heightsList_ = new scalarList(dict_.lookup("heights"));
    }
    // If no heights are specified, the utilities will output a fill for each
    // available heights
    else
    {
        // create a empty list of zero element;
        heightsList_ = new scalarList(0);
    }

    // As the only interest is on the mean field, this function analyse only the
    // last time step.
    // endTime is the number of time steps. But the first index of the array is 0
    //so the last is endTime - 1;
    endTime--;
    runTime.setTime(Times[endTime], endTime);

    forAll((*fieldsList_), fieldi)
    {
        Info<< "Collapsing field " << (*fieldsList_)[fieldi]  << endl;

        // Read fields
      IOobject fieldHeader
      (
                  (*fieldsList_)[fieldi],
                  runTime.times()[endTime].name(),
                  mesh,
                  IOobject::MUST_READ
            );

            if (!fieldHeader.headerOk())
            {
                  Info<< nl << "No field " << (*fieldsList_)[fieldi] << " found" <<
nl;
                  FatalError.exit();
            }

            volScalarField field
            (
```

```
                fieldHeader,
                mesh
            );

          // compute the profile
            List<scalarField>& fieldValues = wallIndexing.collapse
            (
                fieldHeader,
                field
            );

            // read the distance off the wall
            scalarField n = wallIndexing.xi(wallIndexing.normalWall());
            scalarField& h = wallIndexing.heights();

        List<scalarField> *fieldValues_;
        // Extrapolate the profile to the specified heights
        if(dict_.found("heights"))
        {
            label index = 0;
            fieldValues_ = new List<scalarField>(heightsList_->size());

            forAll(*heightsList_, hi)
            {
                do
                {
                    if(h[index]>=(*heightsList_)[hi])
                    {
                        break;
                    }
                    index++;
                }while(index < h.size());

                scalar dh = 0.0;
                scalar deltaH = 0.0;
                scalarField *sf = new scalarField(n.size());

                if(index == 0)
                {
                    // extrapole from the two first points
                    dh = h[index+1]-h[index];
                    deltaH = (*heightsList_)[hi]-h[index];
                    forAll((*sf), j)
                    {
                        (*sf)[j] = ((fieldValues[index+1])[j]-(fieldValues[index])
[j])/dh*deltaH+(fieldValues[index])[j];
                    }
                }
                else
                {
                    if(index == h.size())
                    {
                        // extrapole from the two last points
                        dh = h[index-1]-h[index-2];
```

```
                        deltaH = (*heightsList_)[hi]-h[index-1];

                        forAll((*sf), j)
                        {
                            (*sf)[j] = ((fieldValues[index-1])[j]-
(fieldValues[index-2])[j])/dh*deltaH+(fieldValues[index-1])[j];
                        }
                    }
                    else
                    {
                        // interpolate the value
                        dh = h[index]-h[index-1];
                        deltaH = (*heightsList_)[hi]-h[index];

                        forAll((*sf), j)
                        {
                            (*sf)[j] = ((fieldValues[index])[j]-(fieldValues[index-
1])[j])/dh*deltaH+(fieldValues[index])[j];
                        }
                    }
                }

                (*fieldValues_)[hi]=(*sf);
            }

            h = *heightsList_;
            fieldValues = *fieldValues_;
        }

            // write a folder for each elevation and a file for each field
            fileName file = fieldHeader.path()/"profiles";

            forAll(fieldValues, hi)
            {
                  // transform the heigh from scalar to string
                  std::ostringstream o;
                  o << h[hi];
                  fileName f(file/o.str());

                  if(!exists(f))
                  {
                        Foam::mkDir(f);
                  }

                  // write the data
                  makeGraph(n, fieldValues[hi], field.name(), f, gFormat);
            }

    }
    Info<< "end" << endl;
    return 0;
}
// ************************************************************************* //
```

wallIndex.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2008 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    Foam::wallIndex

Description
   This class allow to compute average profile of a specified field at each height.

SourceFiles
    wallIndex.C

    @author Frederic Collonval
    @email  fcollonv@umd.edu
    @version    2009_07_17
\*---------------------------------------------------------------------------*/

#ifndef wallIndex_H
#define wallIndex_H

#include "fvCFD.H"


/*---------------------------------------------------------------------------*\
                          Class wallIndex Declaration
\*---------------------------------------------------------------------------*/

class wallIndex
{
```

```
    // Private data
        IOdictionary indexingDict_;

        // number of cell in the xi direction
        const label nx_;
        const label ny_;
        const label nz_;

        // start point of the process. The process compute the average of the
        // field along the x direction for each elevation z from the start point
        const point startPoint_;

        // direction normal to the wall
        label normalWall_;
        // vertical direction
        label elevation_;

        scalarField *heights_;

        mutable bool computed_;

        const fvMesh& mesh_;

    // Private Member Functions
        //- Disallow default bitwise copy construct and assignment
        wallIndex(const wallIndex&);
        void operator=(const wallIndex&);

    // Enumeration
        enum direction {X, Y, Z};

public:
    // Constructors
        wallIndex(const fvMesh& m);
    // Destructor
        ~wallIndex();
    // Member Functions
        // Access
            //- number of cells in X direction
            label nx() const
            {
                return nx_;
            }
            //- number of cells in Y direction
            label ny() const
            {
                return ny_;
            }
            //- number of cells in Z direction
            label nz() const
            {
                return nz_;
            }
```

```
            point startPoint() const
            {
                return startPoint_;
            }

            label normalWall() const
            {
                return normalWall_;
            }

            label elevation() const
            {
                return elevation_;
            }
            // Get the available heights
            scalarField& heights() const
            {
                if(!computed_)
                {
                    Info<< nl << "You have to call the function 'collapse' before
asking the 'heights'" << nl;
                    FatalError.exit();
                }

                return *heights_;
            }

            /*- extract the cells along a line that contains the 'inPoint' in the
             *  direction 'direction'
             *  @param point inPoint point included in the line
             *  @param label direction  direction of the line (value
wallIndex::direction)
             *  @param bool  positive   if true, the cell are only extracted in
             *          the positive direction from the inPoint (default
value=false)
             *  @return labelList&      list of the cells extracted
             */
            labelList& extractLineCell
            (
                point   inPoint,
                label   direction,
                bool    positive = false
            )const;
            /*- extract the cells along a line that contains the 'inPoint' in the
             *  direction 'direction'
             *  @param label inCell     cell included in the line
             *  @param label direction  direction of the line (value
wallIndex::direction)
             *  @param bool  positive   if true, the cell are only extracted in
             *          the positive direction from the inCell (default
value=false)
             *  @return labelList&      list of the cells extracted
             */
```

```
            labelList& extractLineCell
            (
                label   inCell,
                label   direction,
                bool    positive = false
            )const;
            //- compute the profile of a scalar field at each available elevation
            //  in the direction normal to the wall.
            // @param const volScalarField& vsf the analyzed scalar field
            // @param label normalWall          the direction normal to the wall
            // @param label elevation           direction of the height of the wall
            // @return  List<scalarField>&      List of the profile
            List<scalarField>& collapse
            (
                const volScalarField& vsf,
                label normalWall,
                label elevation
            ) const;
            //- compute the profile of a scalar field at each available elevation
            //  in the direction normal to the wall.
            // @param const volScalarField& vsf the analyzed scalar field
            // @return  List<scalarField>&      List of the profile
            List<scalarField>& collapse
            (
                const volScalarField& vsf
            ) const;
            //- collapse a field to a line for a specific plane normal to the wall
            //  and the z direction
            // @param const volScalarField& vsf value of the analyzed field
            // @param const label inCell    label of the cell that is included in
the line
            // @param label direction       direction of the line
            // @return  scalar  return the mean value of the line
            scalar collapseLine
            (
                const volScalarField& vsf,
                const label inCell,
                label direction
            ) const;

            //- return the field of xi locations from the cells
            // @param label xi  the direction analyzed (0 = x; 1 = y; 2 = z)
            // @return scalarField list of the locations
            //
            // utility : obtain the evolution of the coordinate normal to the wall
            scalarField xi
            (
                    label   xi
            )const;
};
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif
// ************************************************************************* //
```

wallIndex.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2008 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

    @author Frederic Collonval
    @email  fcollonv@umd.edu
    @version    2009_07_17
\*---------------------------------------------------------------------------*/

#include "wallIndex.H"
#include "UList.H"

#define TOL 1e-12

// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * * //

wallIndex::wallIndex(const fvMesh& m)
:
    indexingDict_
    (
        IOobject
        (
            "profileWallDict",
            m.time().constant(),
            m,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    ),
    nx_(readLabel(indexingDict_.lookup("Nx"))),
```

```
    ny_(readLabel(indexingDict_.lookup("Ny"))),
    nz_(readLabel(indexingDict_.lookup("Nz"))),
    startPoint_(indexingDict_.lookup("startPoint")),
    mesh_(m)
{
      computed_ = false;

    if(indexingDict_.found("normalWall"))
    {
        normalWall_ = readLabel(indexingDict_.lookup("normalWall"));
    }
    else
    {
        normalWall_ = wallIndex::Y;
    }

    if(indexingDict_.found("elevation"))
    {
        elevation_ = readLabel(indexingDict_.lookup("elevation"));
    }
    else
    {
        elevation_ = wallIndex::Z;
    }

    switch(elevation_)
    {
        case wallIndex::X :
            heights_ = new scalarField(nx_);
            break;

        case wallIndex::Y :
            heights_ = new scalarField(ny_);
            break;

        case wallIndex::Z :
            heights_ = new scalarField(nz_);
            break;

        default :
            heights_ = new scalarField(nx_);
            Info<< nl << "The elevation is specified by 0, 1 or 2 for x, y or z" <<
nl;
            FatalError.exit();
    }
}


// * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * * * //

wallIndex::~wallIndex()
{}
```

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

List<scalarField>& wallIndex::collapse
(
    const volScalarField& vsf
) const
{
    return collapse(vsf, normalWall_, elevation_);
}

List<scalarField>& wallIndex::collapse
(
    const volScalarField& vsf,
    label normalWall,
    label elevation
) const
{
    List<scalarField> *profiles;

    switch(elevation)
    {
        case wallIndex::X :
            profiles = new List<scalarField>(nx_);
            break;

        case wallIndex::Y :
            profiles = new List<scalarField>(ny_);
            break;

        case wallIndex::Z :
            profiles = new List<scalarField>(nz_);
            break;

        default :
            profiles = new List<scalarField>(nx_);
            Info<< nl << "The elevation is specified by 0, 1 or 2 for x, y or z" <<
nl;
            FatalError.exit();
    }

    // extract the first cell of each profile next to the wall
    labelList& cellsAlongWall = extractLineCell(startPoint_, elevation, true);
    // Get the cell centers from the mesh
    const volVectorField& cellsCentre = mesh_.C();
    scalarField *sf;
    forAll(cellsAlongWall, celli)
    {
        switch(normalWall)
        {
            case wallIndex::X :
                sf = new scalarField(nx_);
                break;
```

```
            case wallIndex::Y :
                sf = new scalarField(ny_);
                break;

            case wallIndex:: Z :
                sf = new scalarField(nz_);
                break;

            default :
                sf = new scalarField(nx_);
                Info<< nl << "The normalWall is specified by 0, 1 or 2 for x, y or
z" << nl;
                FatalError.exit();
        }
        Info << nl << "Extraction starting from " <<
cellsCentre[cellsAlongWall[celli]] << nl ;

        // extract the cells along the normal of the wall
        labelList& distanceWall = extractLineCell(cellsAlongWall[celli],
normalWall, true);

        forAll(distanceWall, valuei)
        {
            // put the value of the mean field in the scalarField
            (*sf)[valuei] = collapseLine(vsf, distanceWall[valuei], 3-
(elevation+normalWall));
        }

        delete &distanceWall;
        // Store the height of the current profile
        switch(elevation)
        {
                  case wallIndex::X :
                        (*heights_)[celli] =
cellsCentre[cellsAlongWall[celli]].x();
                        break;

                  case wallIndex::Y :
                        (*heights_)[celli] =
cellsCentre[cellsAlongWall[celli]].y();
                        break;


                  case wallIndex::Z :
                        (*heights_)[celli] =
cellsCentre[cellsAlongWall[celli]].z();
                        break;
        }
        (*profiles)[celli] = (*sf);
    }
    computed_ = true;
    return *profiles;
}
```

```
scalar wallIndex::collapseLine
(
    const volScalarField& vsf,
    const label inCell,
    label direction
) const
{
    scalar mean = 0.0;
    // Get the list of cells in the homogeneous direction
    const labelList& cellsList = extractLineCell(inCell, direction);
    // Compute the average in the homogeneous direction
    forAll(cellsList, celli)
    {
        mean += vsf[cellsList[celli]];
    }

    return (mean/cellsList.size());
}

labelList& wallIndex::extractLineCell
(
    point inPoint,
    label direction,
    bool    positive
) const
{
    return extractLineCell(mesh_.findCell(inPoint), direction, positive);
}


labelList& wallIndex::extractLineCell
(
    label   inCell,
    label   direction,
    bool    positive
)const
{
    labelList *cellsLine;

    switch(direction)
    {
        case wallIndex::X :
            cellsLine = new labelList(nx_);
            break;

        case wallIndex::Y :
            cellsLine = new labelList(ny_);
            break;

        case wallIndex::Z :
            cellsLine = new labelList(nz_);
            break;
```

```
        default :
            cellsLine = new labelList(nx_);
            Info<< nl << "The direction is specified by wallIndex::X, wallIndex::Y
or wallIndex::Z" << nl;
            FatalError.exit();
    }
    // Get the list of neighbor cells for each cell of the mesh
    const labelListList& cellsNei = mesh_.cellCells();
    // Get the cell centers from the mesh
    const volVectorField& cellsCentre = mesh_.C();
    label celli = inCell;
    (*cellsLine)[0] = celli;
    scalar flag = 1.0;
    // Find the neighbor cell of the current inCell in the direction. If positive,
    // is true only the positive direction is studied. The next cell is the cell
    // for which the distance in the specified direction is the shortest when the
    // the distances for the other directions are equal to zero.
    for(label i=1; i < (*cellsLine).size(); i++)
    {
        vector owner = cellsCentre[celli];
        scalar dist = Foam::GREAT;

        forAll(cellsNei[celli], nei)
        {
            vector neighbour = cellsCentre[cellsNei[celli][nei]];
            scalar dx = flag*(neighbour.x()-owner.x());
            scalar dy = flag*(neighbour.y()-owner.y());
            scalar dz = flag*(neighbour.z()-owner.z());

            switch(direction)
            {
                case wallIndex::X :
                    if((-1.0*TOL < dy) && (dy < TOL) && (-1.0*TOL < dz) && (dz <
TOL) && (dx > 0.0) && (dx < dist))
                    {
                        dist = dx;
                        celli = cellsNei[celli][nei];
                    }
                    break;

                case wallIndex::Y :
                    if((-1.0*TOL < dx) && (dx < TOL) && (-1.0*TOL < dz) && (dz <
TOL) && (dy > 0.0) && (dy < dist))
                    {
                        dist = dy;
                        celli = cellsNei[celli][nei];
                    }
                    break;

                case wallIndex::Z :
                    if((-1.0*TOL < dy) && (dy < TOL) && (-1.0*TOL < dx) && (dx <
TOL) && (dz > 0.0) && (dz < dist))
                    {
```

```
                        dist = dz;
                        celli = cellsNei[celli][nei];
                    }
                }
            }
            // If there is no more cell in the specified direction
            if((*cellsLine)[i-1] == celli)
            {
                if(positive)
                { // if positive is true, stop here
                    break;
                }
                else
          // if positive is false, begin from the start cell but in the other direction
                {
                    celli = inCell;
                    flag = -1.0;
                }
            }
            else
            {
                // Add the cell to the list
                (*cellsLine)[i] = celli;
            }
        }
        return *cellsLine;
};

scalarField wallIndex::xi
(    label xi ) const
{
    // extract the list of cells in the direction xi form the startPoint
    const labelList& cellsList = extractLineCell(startPoint_, xi);
    const volVectorField& cellsCentre = mesh_.C();
    scalarField Xi(cellsList.size());
    // list the evolution of the coordinate in the xi direction
    forAll(Xi, i)
    {
        switch(xi)
        {
            case wallIndex::X :
                Xi[i]=cellsCentre[cellsList[i]].x();
                break;
            case wallIndex::Y :
                Xi[i]=cellsCentre[cellsList[i]].y();
                break;
            case wallIndex::Z :
                Xi[i]=cellsCentre[cellsList[i]].z();
        }
    }
    return Xi;
}
// ************************************************************************* //
```

Second solution

The key function is defined in the `sampledAveragePlaneTemplates.C` file. The other files are nearly identical at the *sampledPlane* source code.

To install it, the three files have to be copied in the `$FOAM_SRC/sampling/sampledSurface/sampledAveragePlane` folder. Then you have to add the `sampledAveragePlane.C` file to the list of `$FOAM_SRC/sampling/Make/files`. And finally you have to compile the sampling library (`wclean libso; wmake libso` in the `$FOAM_SRC/sampling` folder)

sampledAveragePlane.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    Foam::sampledAveragePlane

Description
    A sampledSurface defined by a cuttingPlane. Always triangulated.
    This cutting plane is the basic plane on which the average field will
    be projected. So this plane have to be normal to the homogeneous
    direction.
    Two additional parameters are required the endOfDomain and the nPoints.
    The first one is the maximal value of the coordinate normal to the plane.
    The second is the number of points used to compute the average in the
    homogeneous direction.

    @author Frederic Collonval
    @email fcollonv@umd.edu
```

```
    @version 08172009

SourceFiles
    sampledAveragePlane.C

\*---------------------------------------------------------------------------*/

#ifndef sampledAveragePlane_H
#define sampledAveragePlane_H

#include "sampledSurface.H"
#include "cuttingPlane.H"

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

namespace Foam
{

/*---------------------------------------------------------------------------*\
                        Class sampledAveragePlane Declaration
\*---------------------------------------------------------------------------*/

class sampledAveragePlane
:
    public sampledSurface,
    public cuttingPlane
{
    // Private data

        //- zone name (if restricted to zones)
        word zoneName_;

        //- Track if the surface needs an update
        mutable bool needsUpdate_;

    // Private Member Functions

        //- sample field on faces
        template <class Type>
        tmp<Field<Type> > sampleField
        (
            const GeometricField<Type, fvPatchField, volMesh>& vField
        ) const;


        template <class Type>
        tmp<Field<Type> >
        interpolateField(const interpolation<Type>&) const;


      //- End of domain from the plane
      scalar end_;
```

```
      //- Direction normal to the plane
      word axis_;

      //- Number of points used to compute the average
      label nPoints_;

public:

    //- Runtime type information
    TypeName("sampledAveragePlane");


    // Constructors

        //- Construct from components
        sampledAveragePlane
        (
            const word& name,
            const polyMesh& mesh,
            const plane& planeDesc,
                  scalar endOfDomain,
                  label nPoints,
            const word& zoneName = word::null
        );

        //- Construct from dictionary
        sampledAveragePlane
        (
            const word& name,
            const polyMesh& mesh,
            const dictionary& dict
        );


    // Destructor

        virtual ~sampledAveragePlane();


    // Member Functions

        //- Does the surface need an update?
        virtual bool needsUpdate() const;

        //- Mark the surface as needing an update.
        //  May also free up unneeded data.
        //  Return false if surface was already marked as expired.
        virtual bool expire();

        //- Update the surface as required.
        //  Do nothing (and return false) if no update was needed
        virtual bool update();
```

```
//- Points of surface
virtual const pointField& points() const
{
    return cuttingPlane::points();
}

//- Faces of surface
virtual const faceList& faces() const
{
    return cuttingPlane::faces();
}

//- For every face original cell in mesh
const labelList& meshCells() const
{
    return cuttingPlane::cutCells();
}

//- sample field on surface
virtual tmp<scalarField> sample
(
    const volScalarField&
) const;


//- sample field on surface
virtual tmp<vectorField> sample
(
    const volVectorField&
) const;

//- sample field on surface
virtual tmp<sphericalTensorField> sample
(
    const volSphericalTensorField&
) const;

//- sample field on surface
virtual tmp<symmTensorField> sample
(
    const volSymmTensorField&
) const;

//- sample field on surface
virtual tmp<tensorField> sample
(
    const volTensorField&
) const;
```

```
        //- interpolate field on surface
        virtual tmp<scalarField> interpolate
        (
            const interpolation<scalar>&
        ) const;


        //- interpolate field on surface
        virtual tmp<vectorField> interpolate
        (
            const interpolation<vector>&
        ) const;

        //- interpolate field on surface
        virtual tmp<sphericalTensorField> interpolate
        (
            const interpolation<sphericalTensor>&
        ) const;

        //- interpolate field on surface
        virtual tmp<symmTensorField> interpolate
        (
            const interpolation<symmTensor>&
        ) const;

        //- interpolate field on surface
        virtual tmp<tensorField> interpolate
        (
            const interpolation<tensor>&
        ) const;

        //- Write
        virtual void print(Ostream&) const;
};


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

} // End namespace Foam

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

#ifdef NoRepository
#   include "sampledAveragePlaneTemplates.C"
#endif

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

#endif
// ************************************************************************* //
```

sampledAveragePlane.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*---------------------------------------------------------------------------*/
#include "sampledAveragePlane.H"
#include "dictionary.H"
#include "polyMesh.H"
#include "volFields.H"

#include "addToRunTimeSelectionTable.H"

#define TOL 1e-6

// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(sampledAveragePlane, 0);
    addNamedToRunTimeSelectionTable(sampledSurface, sampledAveragePlane, word,
averagePlane);
}
// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //
Foam::sampledAveragePlane::sampledAveragePlane
(
    const word& name,
    const polyMesh& mesh,
    const plane& planeDesc,
    scalar endOfDomain,
    label nPoints,
    const word& zoneName
```

```
)
:
    sampledSurface(name, mesh),
    cuttingPlane(planeDesc),
    zoneName_(zoneName),
    needsUpdate_(true)
{
    end_ = endOfDomain;
    nPoints_ = nPoints;

    if (debug && zoneName_.size())
    {
        if (mesh.cellZones().findZoneID(zoneName_) < 0)
        {
            Info<< "cellZone \"" << zoneName_
                << "\" not found - using entire mesh" << endl;
        }
    }
    // verify that the normal is one of the coordinate axis
    const vector& normal = planeDesc.normal();
    if (abs(abs(normal.x())+abs(normal.y())+abs(normal.z())-1.0) > TOL)
    {
        FatalErrorIn
          (
               "Foam::sampledAveragePlane::sampledAveragePlane"
          )   << "The plane normal is not one of the coordinate axis"
              << exit(FatalError);
    }
    // Determine which direction is the normal
    if(abs(abs(normal.x())-1.0) < TOL)
    {
        axis_ = "X";
    }
    else if(abs(abs(normal.y())-1.0) < TOL)
    {
        axis_ = "Y";
    }
    else if(abs(abs(normal.z())-1.0) < TOL)
    {
        axis_ = "Z";
    }
}

Foam::sampledAveragePlane::sampledAveragePlane
(
    const word& name,
    const polyMesh& mesh,
    const dictionary& dict
)
:
    sampledSurface(name, mesh, dict),
    cuttingPlane(plane(dict.lookup("basePoint"), dict.lookup("normalVector"))),
    zoneName_(word::null),
```

```
    needsUpdate_(true)
{
    // make plane relative to the coordinateSystem (Cartesian)
    // allow lookup from global coordinate systems
    if (dict.found("coordinateSystem"))
    {
        coordinateSystem cs(dict, mesh);

        point  base = cs.globalPosition(planeDesc().refPoint());
        vector norm = cs.globalVector(planeDesc().normal());

        // assign the plane description
        static_cast<plane&>(*this) = plane(base, norm);
    }
    dict.readIfPresent("zone", zoneName_);
    if (debug && zoneName_.size())
    {
        if (mesh.cellZones().findZoneID(zoneName_) < 0)
        {
            Info<< "cellZone \"" << zoneName_
                << "\" not found - using entire mesh" << endl;
        }
    }
    // verify that the normal is one of the coordinate axis
   const vector& normal = planeDesc().normal();
    if (abs(abs(normal.x())+abs(normal.y())+abs(normal.z())-1.0) > TOL)
    {
        FatalErrorIn
          (
               "Foam::sampledAveragePlane::sampledAveragePlane"
          )   << "The plane normal is not one of the coordinate axis"
              << exit(FatalError);
    }
    // Determine which direction is the normal
    if(abs(abs(normal.x())-1.0) < TOL)
    {
        axis_ = "x";
    }
    else if(abs(abs(normal.y())-1.0) < TOL)
    {
        axis_ = "y";
    }
    else if(abs(abs(normal.z())-1.0) < TOL)
    {
        axis_ = "z";
    }
    // read the value for the parameters in the sample dictionary
    dict.lookup("endOfDomain") >> end_;
    dict.lookup("nPoints") >> nPoints_;
}
// * * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * * * //

Foam::sampledAveragePlane::~sampledAveragePlane()
```

```
{}
// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //

bool Foam::sampledAveragePlane::needsUpdate() const
{
    return needsUpdate_;
}

bool Foam::sampledAveragePlane::expire()
{
    // already marked as expired
    if (needsUpdate_)
    {
        return false;
    }

    sampledSurface::clearGeom();

    needsUpdate_ = true;
    return true;
}

bool Foam::sampledAveragePlane::update()
{
    if (!needsUpdate_)
    {
        return false;
    }

    sampledSurface::clearGeom();

    label zoneId = -1;
    if (zoneName_.size())
    {
        zoneId = mesh().cellZones().findZoneID(zoneName_);
    }

    if (zoneId < 0)
    {
        reCut(mesh());
    }
    else
    {
        reCut(mesh(), mesh().cellZones()[zoneId]);
    }

    if (debug)
    {
        print(Pout);
        Pout << endl;
    }
    needsUpdate_ = false;
    return true;
```

```
}
Foam::tmp<Foam::scalarField>
Foam::sampledAveragePlane::sample
(
    const volScalarField& vField
) const
{
    return sampleField(vField);
}

Foam::tmp<Foam::vectorField>
Foam::sampledAveragePlane::sample
(
    const volVectorField& vField
) const
{
    return sampleField(vField);
}

Foam::tmp<Foam::sphericalTensorField>
Foam::sampledAveragePlane::sample
(
    const volSphericalTensorField& vField
) const
{
    return sampleField(vField);
}

Foam::tmp<Foam::symmTensorField>
Foam::sampledAveragePlane::sample
(
    const volSymmTensorField& vField
) const
{
    return sampleField(vField);
}

Foam::tmp<Foam::tensorField>
Foam::sampledAveragePlane::sample
(
    const volTensorField& vField
) const
{
    return sampleField(vField);
}

Foam::tmp<Foam::scalarField>
Foam::sampledAveragePlane::interpolate
(
    const interpolation<scalar>& interpolator
) const
{
    return interpolateField(interpolator);
```

```
}
Foam::tmp<Foam::vectorField>
Foam::sampledAveragePlane::interpolate
(
    const interpolation<vector>& interpolator
) const
{
    return interpolateField(interpolator);
}

Foam::tmp<Foam::sphericalTensorField>
Foam::sampledAveragePlane::interpolate
(
    const interpolation<sphericalTensor>& interpolator
) const
{
    return interpolateField(interpolator);
}


Foam::tmp<Foam::symmTensorField>
Foam::sampledAveragePlane::interpolate
(
    const interpolation<symmTensor>& interpolator
) const
{
    return interpolateField(interpolator);
}


Foam::tmp<Foam::tensorField>
Foam::sampledAveragePlane::interpolate
(
    const interpolation<tensor>& interpolator
) const
{
    return interpolateField(interpolator);
}


void Foam::sampledAveragePlane::print(Ostream& os) const
{
    os  << "sampledAveragePlane: " << name() << " :"
        << "  base:" << refPoint()
        << "  normal:" << normal()
        << "  faces:" << faces().size()
        << "  points:" << points().size();
}


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

sampledAveragePlaneTemplates.C

This function contains the new algorithm to compute the average in the homogeneous direction.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*---------------------------------------------------------------------------*/

#include "sampledAveragePlane.H"
#include "uniformSet.H"
#include "meshSearch.H"
// * * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * //

template <class Type>
Foam::tmp<Foam::Field<Type> >
Foam::sampledAveragePlane::sampleField
(
    const GeometricField<Type, fvPatchField, volMesh>& vField
) const
{
    //You have to specify an interpolation scheme to compute the average. So the
    // important functio is the next one “interpolateField”
    FatalErrorIn
       (
           "Foam::averagePlan::sampleField(const GeometricField<Type, fvPatchField,
volMesh>& vField)"
       )   << "No interpolation scheme specified"
           << exit(FatalError);

    return tmp<Field<Type> >(new Field<Type>(vField, meshCells()));
}
```

```
template <class Type>
Foam::tmp<Foam::Field<Type> >
Foam::sampledAveragePlane::interpolateField
(
    const interpolation<Type>& interpolator
) const
{
    // One value per point
    tmp<Field<Type> > tvalues(new Field<Type>(points().size()));
    Field<Type>& values = tvalues();

      //- Mesh search engine
      meshSearch searchEngine(mesh(), true);
    // list of boolean to register if a point has already been analyzed or not
    boolList pointDone(points().size(), false);

    forAll(faces(), cutFaceI)
    {
        const face& f = faces()[cutFaceI];

        forAll(f, faceVertI)
        {
            label pointI = f[faceVertI];

            if (!pointDone[pointI]) //If the point has not already been analyzed
            {
              //- Creation of the homogeneous line that start from points()[pointI]
              // on the plane to endPt
                point endPt = points()[pointI];

                if (axis_ == "x")
                {
                        endPt.x() = end_;
                }
                else if (axis_ == "y")
                {
                        endPt.y() = end_;
                }
                else if (axis_ == "z")
                {
                        endPt.z() = end_;
                }

            if(debug)
            {
                  Info << "Start point : " << (points()[pointI]).x() << " " <<
(points()[pointI]).y() << " " <<  (points()[pointI]).z() << nl;
                  Info << "End point : " << endPt.x() << " " << endPt.y() << " " <<
endPt.z() << nl;
                  Info << "Normal : " << axis_ << nl;
                  Info << "End : " << end_ << endl;
            }
```

```
                tmp<Field<Type> > tlinevalues(new Field<Type>(nPoints_));
                Field<Type>& linevalues = tlinevalues();

                  // Extract the line in the homogeneous direction using the
                  // uniformSet class
                uniformSet line("homogeneousLine", mesh(), searchEngine, axis_,
points()[pointI], endPt, nPoints_);

              //- Interpolate the values of the fields in each point of the line
                forAll(line, lineI)
                {
                        linevalues[lineI] = interpolator.interpolate
                        (
                              line[lineI],
                              line.cells()[lineI],
                              line.faces()[lineI]
                        );
                }
                //- Compute the average of the line
                values[pointI] = Foam::average(tlinevalues);

                  if(debug)
                  {
                        Info << "Average value : " << values[pointI] << endl;
                  }
                //- Register that the point was analyzed
                pointDone[pointI] = true;
            }
        }
    }

    return tvalues;
}


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

Control parameters

diagnostic.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2008 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    diagnostic

Description
    Compute the maximum speed and the total turbulent kinetic energy at each
    time step.

    @author Frederic Collonval
    @email fcollonv@umd.edu
    @version 2009_07_01

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "makeGraph.H"

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

int main(int argc, char **argv)
{

#   include "addTimeOptions.H"     //Read the time option (-time or -latestTime)
#   include "setRootCase.H"        //Set the root case (option -case)
#   include "createTime.H"    // create the time line
```

```
    // Get times list
    instantList Times = runTime.times();

    // set startTime and endTime depending on -time and -latestTime options
#   include "checkTimeOptions.H"
    // set the run time to startTime
    runTime.setTime(Times[startTime], startTime);

#   include "createMesh.H"    // create the mesh
    // Get the graph format defined in the controlDict
    const word& gFormat = runTime.graphFormat();
    // initialize the arrays that will contain the control parameters and the time
    scalarField kTot(endTime-startTime);
    scalarField UMax(endTime-startTime);
    scalarField time(endTime-startTime);
    label index=0;
    // read each write interval
    for (label i=startTime; i<endTime; i++)
    {
        // Set the run time to the current index
        runTime.setTime(Times[i], i);
        // Print information on the log
        Info<< "Time = " << runTime.timeName() << endl;
        // Prepare the reading of the turbulent kinetic energy field
        IOobject kheader
        (
            "k",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ
        );
        // Prepare the reading of the velocity field
        IOobject Uheader
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ
        );


        // Check k and U exist
        if (kheader.headerOk() && Uheader.headerOk())
        {
            mesh.readUpdate(); // update the field on the mesh

            Info<< "    Reading k & computing k total" << endl;
            volScalarField k(kheader, mesh); // Read the field

            kTot[index] = (fvc::domainIntegrate(k)).value; // compute the integral

            Info<< "    Reading U & computing Uz max" << endl;
```

```
            volVectorField U(Uheader, mesh);    // read the velocity field
            // compute the maximum of the Z component
            UMax[index] = ((Foam::dimensionedScalar)
(Foam::max(U.component(vector::Z)))).value();

        }
        else
        {
            Info<< "    No k or U" << endl;
        }
            // Get the value of the current time
            time[index] = Times[i].value();

            index++;
    }
    // create the files in the runTime.path() that contains two columns : x = time
    // y = control parameter
    makeGraph(time, UMax, "UMax", runTime.path(), gFormat);
    makeGraph(time, kTot, "kTot", runTime.path(), gFormat);

    Info<< endl;

    return(0);
}


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

Function objects

Control parameters

diagnostic.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    Foam::diagnostic

Description
      This function computes the total turbulent kinetic energy and the maximum
      of the vertical component of the velocity.

      line to include in the controlDict
functions
(
      diagnostic1 // name of the function
      {
            type              diagnostic;
            functionObjectLibs      ("libuserFunctionObjects.so");
            log               true;
            enabled                 true;
            outputControl           timeStep;//outputTime;
            outputInterval          1;
      }
);
      author : Frederic Collonval
```

```
      email :     fcollonv@umd.edu
      last modified : 08082009

SourceFiles
    diagnostic.C
    IOdiagnostic.H

\*---------------------------------------------------------------------------*/

#ifndef diagnostic_H
#define diagnostic_H

#include "primitiveFieldsFwd.H"
#include "volFieldsFwd.H"
#include "HashSet.H"
#include "OFstream.H"
#include "Switch.H"
#include "pointFieldFwd.H"

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

namespace Foam
{

// Forward declaration of classes
class objectRegistry;
class dictionary;
class mapPolyMesh;

/*---------------------------------------------------------------------------*\
                        Class diagnostic Declaration
\*---------------------------------------------------------------------------*/

class diagnostic
{
protected:

    // Protected data

        //- Name of this set of field min/max.
        //  Also used as the name of the output directory.
        word name_;

        const objectRegistry& obr_;

        //- on/off switch
        bool active_;

        //- Switch to send output to Info as well as to file
        Switch log_;

        //- Min/max file ptr
        autoPtr<OFstream> diagnosticFilePtr_;
```

```
    // Private Member Functions

        //- If the output file has not been created create it
        void makeFile();

        //- Disallow default bitwise copy construct
        diagnostic(const diagnostic&);

        //- Disallow default bitwise assignment
        void operator=(const diagnostic&);

        //- Output file header information
        virtual void writeFileHeader();


public:

    //- Runtime type information
    TypeName("diagnostic");


    // Constructors

        //- Construct for given objectRegistry and dictionary.
        //  Allow the possibility to load fields from files
        diagnostic
        (
            const word& name,
            const objectRegistry&,
            const dictionary&,
            const bool loadFromFiles = false
        );


    // Destructor

        virtual ~diagnostic();


    // Member Functions

        //- Return name of the set of field min/max
        virtual const word& name() const
        {
            return name_;
        }

        //- Read the field min/max data
        virtual void read(const dictionary&);

        //- Execute, currently does nothing
        virtual void execute();
```

```
        //- Execute at the final time-loop, currently does nothing
        virtual void end();

        //- Write the diagnostic
        virtual void write();

        //- Update for changes of mesh
        virtual void updateMesh(const mapPolyMesh&)
        {}

        //- Update for changes of mesh
        virtual void movePoints(const pointField&)
        {}
};


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

} // End namespace Foam

#endif
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

diagnostic.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*---------------------------------------------------------------------------*/

#include "diagnostic.H"
```

```
#include "volFields.H"
#include "dictionary.H"
#include "Time.H"
#include "fvCFD.H"

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

namespace Foam
{
    defineTypeNameAndDebug(diagnostic, 0);
}

// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * //

Foam::diagnostic::diagnostic
(
    const word& name,
    const objectRegistry& obr,
    const dictionary& dict,
    const bool loadFromFiles
)
:
    name_(name),
    obr_(obr),
    active_(true),
    log_(false)
{
    // Check if the available mesh is an fvMesh otherise deactivate
    if (!isA<fvMesh>(obr_))
    {
        active_ = false;
        WarningIn
        (
            "diagnostic::diagnostic"
            "(const objectRegistry& obr, const dictionary& dict)"
        )   << "No fvMesh available, deactivating."
            << endl;
    }

    read(dict);
}


// * * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * * * //

Foam::diagnostic::~diagnostic()
{}




// * * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
void Foam::diagnostic::read(const dictionary& dict)
```

```
{
    if (active_)
    {
            log_ = dict.lookupOrDefault<Switch>("log", false);
    }
}


void Foam::diagnostic::makeFile()
{
    // Create the diagnostic file if not already created
    if (diagnosticFilePtr_.empty())
    {
        if (debug)
        {
            Info<< "Creating diagnostic file." << endl;
        }

        // File update
        if (Pstream::master())
        {
            fileName diagnosticDir;
            if (Pstream::parRun())
            {
                // Put in undecomposed case (Note: gives problems for
                // distributed data running)
                diagnosticDir =
                    obr_.time().path()/".."/name_/obr_.time().timeName();
            }
            else
            {
                diagnosticDir =
                    obr_.time().path()/name_/obr_.time().timeName();
            }

            // Create directory if does not exist.
            mkDir(diagnosticDir);

            // Open new file at start up
            diagnosticFilePtr_.reset
            (
                new OFstream(diagnosticDir/(type() + ".dat"))
            );

            // Add headers to output data
            writeFileHeader();
        }
    }
}


void Foam::diagnostic::writeFileHeader()
{
```

```
    if (diagnosticFilePtr_.valid())
    {
      // Write the header on the file
        diagnosticFilePtr_()
            << "# Time" << tab << "UzMax" << tab << "kTot"
            << endl;
    }
}


void Foam::diagnostic::execute()
{
    // Do nothing - only valid on write
}


void Foam::diagnostic::end()
{
    // Do nothing - only valid on write
}


void Foam::diagnostic::write() // the key function
{
    if (active_)
    {
            makeFile();
            if(Pstream::master())
            {     // Get the velocity and turbulent energy fields
                  const volVectorField& U = obr_.lookupObject<volVectorField>("U");
                  const volScalarField& k = obr_.lookupObject<volScalarField>("k");
                  // Write the time, the maximum of the Z component of the velocity
                  // and the integral of the turbulent kinetic field in the file
                  diagnosticFilePtr_() << obr_.time().value() << tab <<
((dimensionedScalar)(max(U.component(vector::Z)))).value() << tab <<
(fvc::domainIntegrate(k)).value() << endl;

                  if(log_)
                  {     // Write the information on the log
                        Info << "Control parameters" << endl;
                        Info <<"UzMax = "<< ((dimensionedScalar)
(max(U.component(vector::Z)))).value() <<endl;
                        Info <<"kTot = " <<(fvc::domainIntegrate(k)).value()<<endl;
                  }
            }
    }
}

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

Heat flux and Nusselt number

heatFluxNusselt.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    Foam::heatFluxNusselt

Description
      This function computes the total turbulent kinetic energy and the maximum
      of the vertical component of the velocity.

      Lines to include in the controlDict
functions
(
      heatFluxNusselt1 // name of the function
      {
            type              heatFluxNusselt;
            functionObjectLibs ("libuserFunctionObjects.so");
            log               true;
            enabled                 true;
            outputControl           outputTime;
            outputInterval          1;
            patchName         wallHot; // name of the analyzed patch
            vertical          Z; // vertical direction
            Ta                293; // Temperature far away from the wall
      }
);

      author : Frederic Collonval
```

```
      email :     fcollonv@umd.edu
      last modified : 08082009

SourceFiles
    heatFluxNusselt.C
    IOheatFluxNusselt.H

\*---------------------------------------------------------------------------*/

#ifndef heatFluxNusselt_H
#define heatFluxNusselt_H

#include "primitiveFieldsFwd.H"
#include "volFieldsFwd.H"
#include "HashSet.H"
#include "OFstream.H"
#include "Switch.H"
#include "pointFieldFwd.H"

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

namespace Foam
{

// Forward declaration of classes
class objectRegistry;
class dictionary;
class mapPolyMesh;

/*---------------------------------------------------------------------------*\
                        Class heatFluxNusselt Declaration
\*---------------------------------------------------------------------------*/

class heatFluxNusselt
{
protected:
    // Protected data
        //- Name of this set of field min/max.
        //  Also used as the name of the output directory.
        word name_;

        const objectRegistry& obr_;

        //- on/off switch
        bool active_;

        //- Switch to send output to Info as well as to file
        Switch log_;

        //- Min/max file ptr
        autoPtr<OFstream> heatFluxNusseltFilePtr_;

            //- Vertical direction
```

```
            word vertical_;

            //- Patch name
            word patchName_;

            //- Temperature ambient
            scalar Ta_;

    // Private Member Functions
        //- If the output file has not been created create it
        void makeFile();

        //- Disallow default bitwise copy construct
        heatFluxNusselt(const heatFluxNusselt&);

        //- Disallow default bitwise assignment
        void operator=(const heatFluxNusselt&);

        //- Output file header information
        virtual void writeFileHeader();

public:

    //- Runtime type information
    TypeName("heatFluxNusselt");

    // Constructors
        //- Construct for given objectRegistry and dictionary.
        //  Allow the possibility to load fields from files
        heatFluxNusselt
        (
            const word& name,
            const objectRegistry&,
            const dictionary&,
            const bool loadFromFiles = false
        );

    // Destructor
        virtual ~heatFluxNusselt();

    // Member Functions
        //- Return name of the set of field min/max
        virtual const word& name() const
        {
            return name_;
        }
        //- Read the field min/max data
        virtual void read(const dictionary&);
        //- Execute, currently does nothing
        virtual void execute();
        //- Execute at the final time-loop, currently does nothing
        virtual void end();
```

```
        //- Write the heatFluxNusselt
        virtual void write();

        //- Update for changes of mesh
        virtual void updateMesh(const mapPolyMesh&)
        {}

        //- Update for changes of mesh
        virtual void movePoints(const pointField&)
        {}
};
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace Foam
#endif
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

heatFluxNusselt.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*---------------------------------------------------------------------------*/
#include "heatFluxNusselt.H"
#include "volFields.H"
#include "dictionary.H"
#include "Time.H"
#include "LESModel.H"
#include <cmath>
#include "makeGraph.H"

// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
```

```
{
    defineTypeNameAndDebug(heatFluxNusselt, 0);
}
// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //
Foam::heatFluxNusselt::heatFluxNusselt
(
    const word& name,
    const objectRegistry& obr,
    const dictionary& dict,
    const bool loadFromFiles
)
:
    name_(name),
    obr_(obr),
    active_(true),
    log_(false)
{
    // Check if the available mesh is an fvMesh otherise deactivate
    if (!isA<fvMesh>(obr_))
    {
        active_ = false;
        WarningIn
        (
            "heatFluxNusselt::heatFluxNusselt"
            "(const objectRegistry& obr, const dictionary& dict)"
        )   << "No fvMesh available, deactivating."
            << endl;
    }
    read(dict);
}

// * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //
Foam::heatFluxNusselt::~heatFluxNusselt()
{}

// * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
void Foam::heatFluxNusselt::read(const dictionary& dict)
{
    if (active_)
    {       // read the parameters in the dictionary
            log_ = dict.lookupOrDefault<Switch>("log", false);
            dict.lookup("patchName") >> patchName_;
            dict.lookup("vertical") >> vertical_;
            dict.lookup("Ta") >> Ta_;
    }
}


void Foam::heatFluxNusselt::makeFile()
{
    // Create the heatFluxNusselt file if not already created
//    if (heatFluxNusseltFilePtr_.empty())
//    {
```

```
        if (debug)
        {
            Info<< "Creating heatFluxNusselt file." << endl;
        }

        // File update
        if (Pstream::master())
        {
            fileName heatFluxNusseltDir;
            if (Pstream::parRun())
            {
                // Put in undecomposed case (Note: gives problems for
                // distributed data running)
                heatFluxNusseltDir =
                    obr_.time().path()/".."/name_/obr_.time().timeName();
            }
            else
            {
                heatFluxNusseltDir =
                    obr_.time().path()/name_/obr_.time().timeName();
            }

            // Create directory if does not exist.
            mkDir(heatFluxNusseltDir);

            // Open new file at start up
            heatFluxNusseltFilePtr_.reset
            (
                new OFstream(heatFluxNusseltDir/(type() + ".dat"))
            );

            // Add headers to output data
            writeFileHeader();
        }
//    }
}
void Foam::heatFluxNusselt::writeFileHeader()
{
    if (heatFluxNusseltFilePtr_.valid())
    { // Write the header in the file
        heatFluxNusseltFilePtr_()
            << "# Time" << tab << obr_.time().value() << endl;
            heatFluxNusseltFilePtr_()
                  << vertical_ << tab << "qw" << tab << "Nu"
            << endl;
    }
}
void Foam::heatFluxNusselt::execute()
{
    // Do nothing - only valid on write
}
```

```
void Foam::heatFluxNusselt::end()
{
    // Do nothing - only valid on write
}

void Foam::heatFluxNusselt::write() // the key function
{
    if (active_)
    {
            makeFile();
            if(Pstream::master())
            {     // Get the LES properties
                  const Foam::compressible::LESModel & sgs =
obr_.lookupObject<compressible::LESModel>("LESProperties");
                  // Get the thermophysical properties
                  const Foam::basicThermo& thermo =
obr_.lookupObject<basicThermo>("thermophysicalProperties");
                  // Get the temperature field
                  const volScalarField & Ttmp =
obr_.lookupObject<volScalarField>("T");
                  // Get the mesh
                  const fvMesh & mesh = Ttmp.mesh();
                  // Get the index of the boundary in the patch list
                  label patchI = mesh.boundaryMesh().findPatchID(patchName_);

                  direction d_;
                  if(vertical_ == "X")
                  {
                        d_ = Foam::point::X;
                  }
                  if(vertical_ == "Y")
                  {
                        d_ = Foam::point::Y;
                  }
                  if(vertical_ == "Z")
                  {
                        d_ = Foam::point::Z;
                  }
                  // Get the temperature on the boundary
                  const fvPatchScalarField& Tw = thermo.T().boundaryField()
[patchI];
                  // Get the temperature gradient normal to the wall. Avoid to be
                  // equal to zero.
                  const scalarField magGradTw = max(mag(Tw.snGrad()), SMALL);
                  // Get the effective thermal diffusivity
                  const scalarField alphaEffw = sgs.alphaEff()().boundaryField()
[patchI];

                  // Get the physical thermal diffusivity
                  const scalarField alphaw = sgs.alpha().boundaryField()[patchI];
                  // Get the specific heat capacity
                  const scalarField Cpw = thermo.Cp(Tw, patchI);
```

```
                scalarField alphaEffwGradTw = alphaEffw*magGradTw;

                if(log_)
                {     //write information in the log
                      Info << "Compute the heat flux" << endl;
                }

                //- compute the heat flux for the patch
                scalarField qw = Cpw*alphaEffwGradTw;

                //- read the vertical component of the face center for each face
                // of the patch
                const scalarField Xv = mesh.boundaryMesh()
[patchI].faceCentres().component(d_);

                if(log_)
                {     // Write information in the log
                      Info << "Compute the Nusselt number" << endl;
                }

                //- compute the Nusselt number for the patch
                scalarField Nu = Xv*(alphaEffwGradTw/alphaw)/(Tw - Ta_);

                label length = Xv.size();

                for(label i=0; i<length; i++)
                {     // Write the evolution of the heat flux with the vertical
                      // coordinate in the file
                      heatFluxNusseltFilePtr_()
                            << Xv[i] << tab << qw[i] << tab << Nu[i]
                            << endl;
                }

                //- Compute the mean profiles
                if(log_)
                {     // Write information in the log
                      Info << "Compute the mean profiles" << endl;
                }

                //- Look for the number of vertical point
                scalarField Xv_tmp(Xv);
                Foam::sort(Xv_tmp); // sort the vertical point
                label nbVerticalPoint = 1;
                scalarField Xv_mean(length);
                Xv_mean[0] = Xv_tmp[0];
                // create the mean evolution of the vertical coordinate
                for(label i=1; i<length; i++)
                {
                      if((Xv_tmp[i] - Xv_mean[nbVerticalPoint-1]) > 1e-6)
                      {
                            Xv_mean[nbVerticalPoint++] = Xv_tmp[i];
                      }
                }
```

```
                    //- Compute the mean profile of qw and Nu
                    scalarField qwMean(nbVerticalPoint);
                    scalarField NuMean(nbVerticalPoint);
                    for(label j=0; j<nbVerticalPoint; j++)
                    {
                          qwMean[j] = 0.0;
                          NuMean[j] = 0.0;
                    }

                    for(label i=0; i<length; i++)
                    {
                          label index = -1;
                          for(label j = 0; j< nbVerticalPoint; j++)
                          {
                                if(((Xv_tmp[i] - Xv_mean[j]) < 1e-6) && ((Xv_tmp[i] -
Xv_mean[j]) > -1e-6))
                                {
                                      index = j;
                                      break;
                                }
                          }

                          qwMean[index] += qw[i];
                          NuMean[index] += Nu[i];
                    }

                    for(label j=0; j<nbVerticalPoint; j++)
                    {
                          qwMean[j] /= ceil(length/nbVerticalPoint);
                          NuMean[j] /= ceil(length/nbVerticalPoint);
                    }
                    // Generate the files containing the evolution of the mean heat
                    // flux and the mean Nusselt number along the vertical
                    SubField<scalar> XvSub(Xv_mean, nbVerticalPoint);
                    makeGraph(XvSub, qwMean, "qwMean",
obr_.time().path()/name_/obr_.time().timeName(), obr_.time().graphFormat());
                    makeGraph(XvSub, NuMean,
"NuMean",obr_.time().path()/name_/obr_.time().timeName(),
obr_.time().graphFormat());
               }
     }
}

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

Appendix D : Turbulent natural convection case

Table of Contents

Only the uncomment lines are shown in the following appendix as well as the parameters useful (for example in the original file LESProperties, the parameters for other models than the Smagorinski's one are present but there were suppressed in the extract presented here).

Additional comments on the code are present only if they are not described in the User Guide of OpenFOAM.

System folder

controlDict

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           dictionary;
    object          controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     fireFoam;
startFrom       startTime;
startTime       0.0;
stopAt          endTime;
endTime         40.0;
deltaT          1e-4;
writeControl    adjustableRunTime;
writeInterval   0.05;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression uncompressed;
timeFormat      general;
timePrecision   6;
graphFormat     raw;
runTimeModifiable yes;
adjustTimeStep  yes;
maxCo           0.5;
maxDeltaT 0.01;
/*
functions
(
      diagnostic1 // name of the function object
      {     //This function computes the two control parameters at each time step (
            //the maximum of the vertical velocity and the integral of the
            //turbulent kinetic energy
            type                    diagnostic; //type of the function object
            // library that contains the code of the function
```

```
        functionObjectLibs      ("libuserFunctionObjects.so");
        log                     true; //print or not something on the log file
        enabled                 true; //enabled or not
        //specify if the function is executed at each time step or only at the
        //write interval
        outputControl           timeStep;//outputTime;
        outputInterval          1; //specify the interval between each output
  }

  heatFluxNusselt1
  {
        //This function computes the heat flux and the Nusselt number on the
        //specified patch.
        type                    heatFluxNusselt;
        functionObjectLibs      ("libuserFunctionObjects.so");
        log                     true;
        enabled                 true;
        outputControl           outputTime;
        outputInterval          1;
        patchName               wallHot;    //name of the patch
        vertical                Z;          //vertical direction (X, Y or Z)
        Ta                      293;        //reference of temperature
  }

  patchSampling
{
    //This function extracts the convective heat flux and the temperature on
    //the specified patch using the fields created during the execution of
    //fireFoam
    type surfaces;
    functionObjectLibs ("libsampling.so");
    enabled true;
    outputControl timeStep;
    outputInterval 2;
    surfaceFormat raw; // format of the output (raw, vtk,...)
    fields             // fields to ouput
    (
        T
        convectiveHeatFlux
    );
    surfaces
    (
        wallHot //name of the output surface
        {
            type          patch; // type of the surface
            // name of the patch as specified in the geometry
            patchName     wallHot;
  // interpolation or not, if yes the interpolation scheme have to be specified
            interpolate   false;
            //[optional] specified the locations where the data are extracted
            locations
            (
                (0 0 0.1)
```

```
                    (0 0 0.2)
                );
            }
        );
    }

);
*/
// ************************************************************************* //
```

fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4.1                                 |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "/home/bauwensc/FireStuff";
    case            "Plume";
    instance        "system";
    local           "";
    class           dictionary;
    object          fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
ddtSchemes
{
    // default scheme for the temporal discretization - here Euler implicit method
    default         Euler;
}
gradSchemes
{
    // default discretization scheme for the gradient - Gaussian integration with
    // a linear interpolation to compute the face value from the neighboring cell
    // values
    default         Gauss linear;
}
divSchemes
{
    // default discretization scheme for the divergence - none (not recommended to
    // specified a default one
    default         none;
    // Gaussian integration with a filtered linear interpolation scheme to compute
    // the divergence of the mass flux and the velocity
    div(phi,U)      Gauss filteredLinear 0.2 0.05;
    // For all the following, Gaussian integration with a limited linear
    // interpolation scheme (Total variation diminishing scheme). The coefficient
```

```
    // 1.0 corresponds to a total variation diminishing conformance giving best
    // convergence
    div(phi,k)      Gauss limitedLinear 1.0;
    div(phi,h)      Gauss limitedLinear 1.0;
    div(phi,hc)      Gauss limitedLinear 1.0;
    div(phi,hs)      Gauss limitedLinear 1.0;
    div(phi,ft)     Gauss limitedLinear01 1;
    div(phi,fu)     Gauss limitedLinear 1;
    div(phi,ftVar)     Gauss limitedLinear 1;
    flux(phi,ft)     Gauss limitedLinear01 1;
    div(phi,ft_b_h) Gauss multivariateSelection
    {
        ft              limitedLinear01 1;
        h               limitedLinear 1;
    };
    // Gaussian integration with a linear interpolation scheme
    div((muEff*dev2(grad(U).T()))) Gauss linear;
}
laplacianSchemes
{
    // Gaussian integration with linear interpolation scheme and the gradient
    // interpolation is the kind of explicit with non-orthogonal correction
    //laplacian   integration method     interpolation scheme    gradient scheme
    default         Gauss                linear                  corrected;
    laplacian(muEff,U) Gauss linear corrected;
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DBEff,B) Gauss linear corrected;
    laplacian((rho*(1|A(U))),pd) Gauss linear corrected;
    laplacian(muEff,b) Gauss linear corrected;
    laplacian(muEff,ft) Gauss linear corrected;
    laplacian(alphaEff,h) Gauss linear corrected;
    laplacian(alphaEff,hu) Gauss linear corrected;
}
interpolationSchemes
{
// default interpolation scheme to compute the value on the face from the values
// at the cell centers
    default         linear;
}
snGradSchemes
{
// default surface normal gradient scheme, used the value at the cell centers with
// a correction if the normal to the common face is not parallel to the line that
// linked both cell center
    default         corrected;
}
fluxRequired
{
    default         no;
    pd;
}
```

fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4.1                                 |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "/home/bauwensc/FireStuff";
    case            "Plume";
    instance        "system";
    local           "";
    class           dictionary;
    object          fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
solvers
{
    rho // name of the field
    {
        // name of the solver
        solver           PCG;
        preconditioner   DIC;
        tolerance        0;
        relTol           0;
    };

    pd
    {
        solver           GAMG
        tolerance        1e-8;
        relTol           0.0;

        smoother         GaussSeidel;
        nPreSweeps       0;
        nPostSweeps      2;
        nFinestSweeps    2;
        directSolveCoarsest true;

        cacheAgglomeration true;

        nCellsInCoarsestLevel 10;

        agglomerator     faceAreaPair;
        mergeLevels      1;
    };
```

```
pdFinal       // solver for the final computation of the dynamic pressure
{
    solver           GAMG;
    tolerance        1e-8;
    relTol           0;

    smoother         GaussSeidel;
    nPreSweeps       0;
    nPostSweeps      2;
    nFinestSweeps    2;
    directSolveCoarsest true;

    cacheAgglomeration true;

    nCellsInCoarsestLevel 10;

    agglomerator     faceAreaPair;
    mergeLevels      1;
};


ftVar
{
    solver           smoothSolver;
    smoother         GaussSeidel;
    tolerance        1e-8;
    relTol           0;
    nSweeps          1;
};

U
{
    solver           smoothSolver;
    smoother         GaussSeidel;
    tolerance        1e-8;
    relTol           0.0;
    nSweeps          1;
};

UFinal
{
    solver           smoothSolver;
    smoother         GaussSeidel;
    tolerance        1e-8;
    relTol           0;
    nSweeps          1;
};

k
{
    solver           smoothSolver;
```

```
        smoother         GaussSeidel;
        tolerance        1e-8;
        relTol           0;
        nSweeps          1;
    };
    h
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-8;
        relTol           0;
    };

    ft
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-10;
        relTol           0;
    };
}

PISO // parameters for the pressure-implicit split-operator
{
    momentumPredictor yes;
    nOuterCorrectors  1;
    nCorrectors       2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}

multiVariate
{
    // specified if the model for multivariate species have to be used. As the
    // the natural convection is the phenomena studied is value is no
    bMultiVariate    no;// yes
}

MULES
{
    // specified if the multidimensional universal limiter for explicit solution is
    // used or not
    bMULES     no;
}

compressibleVersion
{
    // specified if the low-Mach model (yes) is used or the fully compressible one
    // (no). As in natural convection the movement is generated by the density
    // gradient, the fully compressible model have to be used.
    bLowMach     no;
}
```

```
mixing
{
    // mixing model to compute the variance of the mixture fraction
    ftVarModel       "LESsimilarity";//"LESsimilarity", "lenghtScale",
"transport"
    // the parameter used in the simulation depends on the model chosen
    LESsimilarityC   0.0;
    lengthScaleC     5;
}

betaIntegration
{
    LookUpTable "LookUpTable";
    pdfMethod        0;
/*
            // 0- delta
            // 1- lookUpTable
            // 2- recursive
*/
}

relaxationFactors
{
//    U       1.0;
//    h       1.0;
//    k       1.0;
}
// ************************************************************************* //
```

constant folder

polyMesh\blockMeshDict

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           dictionary;
```

```
    object          blockMeshDict;
}

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

convertToMeters 0.01; //scale factor applied to the coordinates

vertices // definition of all vertices used to define the geometry
(
      (0 0 0)
      (15 0 0)
      (15 4 0)
      (0 4 0)
      (0 0 100)
      (15 0 100)
      (15 4 100)
      (0 4 100)

      (0 0 120)
      (15 0 120)
      (15 4 120)
      (0 4 120)

      (0 0 400)
      (15 0 400)
      (15 4 400)
      (0 4 400)
);

blocks
(
      // Here the geometry is divided in three main blocks : the laminar, the
      // transition and the turbulent part.
//type of cells   summits           number of division           grading
      hex (0 1 2 3 4 5 6 7)         (120 10 300)      simpleGrading (3 1 1)
      hex (4 5 6 7 8 9 10 11)       (120 10 20)       simpleGrading (3 1 7)
      hex (8 9 10 11 12 13 14 15)   (120 10 125)      simpleGrading (3 1 1)
);

edges
(
);

patches // definition of the borders and the type of them
(
      // definition of the hot wall
    wall wallHot
    (       // list of surfaces that compose the wall
            (0 4 7 3)
            (4 8 11 7)
            (8 12 15 11)
    )
```

```
    patch cold // surface opposite to the hot wall
    (
            (1 2 6 5)
            (5 6 10 9)
            (9 10 14 13)
    )

    cyclic sides1 // boundaries in the spanwise direction of the flow
    (
            (0 1 5 4)
            (4 5 9 8)
            (8 9 13 12)
            (7 6 2 3)
            (11 10 6 7)
            (15 14 10 11)
    )

    patch up // boundary at the top of the domain
    (
       (12 13 14 15)
    )

    patch down // boundary at the bottom of the domain
    (
       (0 3 2 1)
    )
);

mergePatchPairs
(
);

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

fireFoamProperties

This dictionary contains all the properties specific at the fireFoam solver.

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4                                   |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;

    root            "";
```

```
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          environmentalProperties;
}

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

ignited   no; // is the flow ignited? yes or no

ignRampTime  0.2; // ignition ramp time - manage the speed of the reaction
// currently that parameter has no effect as the combustion model is an infinity
// fast chemistry model

pRef          pRef [1 -1 -2 0 0 0 0] 101325.0; // reference pressure

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

g

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  dev                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       uniformDimensionedVectorField;
    location    "constant";
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
// value and dimensions of the gravity acceleration
dimensions      [0 1 -2 0 0 0 0];
value           (0 0 -9.80665);

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

LESProperties

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4                                   |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
```

```
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           dictionary;
    object          LESProperties;
}

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

LESModel       Smagorinsky; // LES Model used
// model used to computed the characteristic length of a cell
delta           cubeRootVol;
// parameters used by the chosen models
SmagorinskyCoeffs
{
    ck               0.0;
    ce               0.202;
}

cubeRootVolCoeffs
{
    deltaCoeff      1;
}

kappa            0.4187; //value of the von Karman constant

wallFunctionCoeffs
{
    E                9; // value used is the default logaritmic law of the wall for
                        // the velocity profile is chosen to correct muSgs.
}
// ************************************************************************* //
```

thermophysicalProperties

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4                                   |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
```

```
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           dictionary;
    object          thermophysicalProperties;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
// models used to simulate the species
thermoType
hPsiMixtureThermo<fireMixture<sutherlandTransport<specieThermo<janafThermo<perfectG
as>>>>>;
// coefficient needed by the models chosen for each species
// for pure methane (100%CH4), air combustion
stoichiometricAirFuelMassRatio stoichiometricAirFuelMassRatio [0 0 0 0 0 0 0]
17.1271;

fuel            fuel 1 16.0428 200 6000 1000 1.63543 0.0100844 -3.36924e-06
5.34973e-10 -3.15528e-14 -10005.6 9.9937 5.14988 -0.013671 4.91801e-05 -4.84744e-08
1.66694e-11 -10246.6 -4.64132  1.67212e-06 170.672;

oxidant         oxidant 1 28.8504 200 6000 1000 3.10131 0.00124137 -4.18816e-07
6.64158e-11 -3.91274e-15 -985.266 5.35597 3.58378 -0.000727005 1.67057e-06
-1.09203e-10 -4.31765e-13 -1050.53 3.11239 1.67212e-06 170.672;

burntProducts   burntProducts 1 27.6334 200 6000 1000 3.0602 0.00182422 -5.93878e-
07 8.93807e-11 -4.97595e-15 -10998.7 5.32209 3.54628 0.000378279 2.02797e-07
9.31602e-10 -6.84016e-13 -11102.1 2.90098 1.67212e-06 170.672;

// value of the turbulent Prandtl number
pr_t    1.0;//0.3;


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

radiationProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.5                                   |
|   \\  /    A nd           | Web:      http://www.OpenFOAM.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
```

```
    object      environmentalProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
// Specify the radiation model - inactive here
radiation      on;

radiationModel none;

noRadiationCoeffs
{
}

absorptionEmissionModel constantAbsorptionEmission;

constantAbsorptionEmissionCoeffs
{
    a              a      [ 0 -1  0  0  0  0  0] 0.5;
    e              e      [ 0 -1  0  0  0  0  0] 0.5;
    E              E      [ 1 -1 -3  0  0  0  0] 0.0;
}

scatterModel constantScatter;

constantScatterCoeffs
{
    sigma          sigma  [ 0 -1  0  0  0  0  0] 0.0;
    C              C      [ 0  0  0  0  0  0  0] 0.0;
}

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

LookUpTable

```
// Look up table that allow to linked the mixture fraction and its variance to
// the fuel fraction
fields
2
(

{
    name            ft;
    min             0;
    max             1;
    N               100;
}


{
    name            ftVar;
    min             0.005;
    max             0.02;
    N               30;
}
```

```
)
;
output
1
(

{
    name            fu;
}

)
;
values
3000
(
3(0.01 0.0055 0)
… //2999 other points that define the table
);
```

BetaPdfCoef.txt

This file contains the coefficients for the probability density function beta. This function is used in the combustion model to compute the average of the chemical term.

```
0.00000000000019895196601282805204391479492187500000
-0.24288321744006680091843008995056152343750000000000
61.14201819407753646373748779296875000000000000000000
-4241.69690935453400015830993652343750000000000000000000
129337.37509399838745594024658203125000000000000000000000
-2179548.46037824451923370361328125000000000000000000000000
23492398.03040879964828491210937500000000000000000000000000
-175505677.41673016548156738281250000000000000000000000000000
953930579.98154354095458984375000000000000000000000000000000
-3891440231.40016174316406250000000000000000000000000000000000
12158339491.07389831542968750000000000000000000000000000000000
-29464720257.31177139282226562500000000000000000000000000000000
55752312853.44132995605468750000000000000000000000000000000000
-82462931013.09133911132812500000000000000000000000000000000000
94919655415.77449035644531250000000000000000000000000000000000
-84090423855.75593566894531250000000000000000000000000000000000
56201350151.57497406005859375000000000000000000000000000000000
-27397039520.52818679809570312500000000000000000000000000000000
9189197416.58379173278808593750000000000000000000000000000000
-1895337744.22678232192993164062500000000000000000000000000000
181174386.15354004502296447753906250000000000000000000000000
```

0 folder

In this folder, the initial conditions and boundary conditions for all the fields that need it are defined.

U - velocity

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volVectorField;
    object          U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 1 -1 0 0 0 0]; // dimensions here m/s

internalField   uniform (0 0 0); // initial value inside the domain

boundaryField
{
    wallHot       // boundary condition for the hot wall
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    up            // boundary condition for the surface at the top of the domain
    {
        type            inletOutlet;
      inletValue  uniform (0 0 0);
      value       uniform (0 0 0);
    }
    down
    {
      type        pressureInletOutletVelocity;
      phi         phi;
      value       uniform     (0 0 0);
    }
    cold          // boundary condition for the surface at the bottom of the domain
    {
      type        pressureInletOutletVelocity;
```

```
      phi         phi;
      value       uniform (0 0 0);
    }
    sides1        // boundary condition in the homogeneous direction
    {
      type        cyclic;
      value       uniform (0 0 0);
    }
}
// ************************************************************************* //
```

T - temperature

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 0 0 1 0 0 0];
internalField   uniform 289;
boundaryField
{
    wallHot
    {
        type            fixedValue;
        value           uniform 333;
    }
    up
    {
      type        inletOutlet;
      inletValue  uniform 289;
      value       uniform 289;
    }
    down
    {
      type        inletOutlet;
      inletValue  uniform 289;
      value       uniform 289;
    }
```

```
    cold
    {
      type        inletOutlet;
      inletValue  uniform 289;
      value       uniform 289;
    }
    sides1
    {
      type            cyclic;
      value       uniform 289;
    }
}
```

k – turbulent kinetic energy

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          k;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 1e-05;
boundaryField
{
    wallHot
    {
        type            fixedValue;
        value           uniform 1e-05;
    }
    up
    {
        type            inletOutlet;
        inletValue      uniform 1e-05;
        value           uniform 1e-05;
    }
    down
    {
        type            inletOutlet;
        inletValue           uniform 1e-05;
```

```
        value           uniform 1e-05;
    }
    cold
    {
        type            inletOutlet;
        inletValue      uniform 1e-05;
        value           uniform 1e-05;
    }
    sides1
    {
        type            cyclic;
    }
}
```

p – total pressure

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 101325;

boundaryField
{
    wallHot
    {
        type            calculated;
        value           uniform 101325;
    }
    up
    {
        type            calculated;
        value           uniform 101325;
    }
    down
```

```
    {
        type            calculated;
        value           uniform 101325;
    }
    cold
    {
        type            calculated;
        value           uniform 101325;
    }
    sides1
    {
        type            cyclic;
    }
}
```

pd – dynamic pressure

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    wallHot
    {
      type              fixedFluxBuoyantPressure;
      gradient          uniform 0;
      value             uniform 0;
    }

    up
    {
```

```
      type              zeroGradient;
    }

    down
    {
      type              totalPressure;
      U                 U;
      phi               phi;
      rho               rho;
      psi               none;
      gamma             0;//1.4;
      p0                uniform 0;
      value             uniform 0;
    }


    cold
    {
        type            totalPressure;
        U               U;
        phi             phi;
        rho             rho;
        psi             none;
        gamma           0;//1.4;
        p0              uniform 0;
        value           uniform 0;
    }
    sides1
    {
        type            cyclic;
      value       uniform 0;
    }
}
// ************************************************************************* //
```

muSgs – subgride-scale dynamic viscosity

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4                                   |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
```

```
    local           "";
    class           volScalarField;
    object          muSgs;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform 0;

boundaryField
{
    wallHot
    {
        type            zeroGradient;
/*
      // Definition of the Holling model
        type            muSgsBuoyantWallFunction;
        beta            1.0/333.0;
        magG            9.80665;
        Tref            293.0;
        Prt             0.9;
        value           uniform 0.0;
*/
    }
    up
    {
        type            zeroGradient;
    }
    down
    {
        type            zeroGradient;
    }
    cold
    {
        type            zeroGradient;
    }
    sides1
      {
        type            cyclic;
    }
}
// ************************************************************************* //
```

alphaSgs – subgride-scale thermal diffusivity

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          alphaSgs;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [1 -1 -1 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    wallHot
    {
        type            zeroGradient;
/*
      // definition of the Holling model
        type            alphaSgsBuoyantWallFunction;
        beta            1.0/293.0;
        magG            9.80665;
        value           uniform 0.0;
*/
    }
    up
    {
        type            zeroGradient;
    }
    down
    {
        type            zeroGradient;
    }
    cold
    {
        type            zeroGradient;
    }
    sides1
    {
        type            cyclic;
    }
}
```

Ydefault – default boundary conditions for the different species

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          Ydefault;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    wallHot
    {
        type            zeroGradient;
    }


    up
    {
        type            zeroGradient;
    }
    down
    {
        type            zeroGradient;
    }
    cold
    {
        type            zeroGradient;
    }
    sides1
    {
        type            cyclic;
    }
}
// ************************************************************************* //
```

b – regress variable

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4.1                                 |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version       2.0;
    format        ascii;
    root          "";
    case          "";
    instance      "";
    local         "";
    class volScalarField;
    object b;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 1;

boundaryField
{
    wallHot
    {
        type            zeroGradient;
    }

    up
    {
        type            zeroGradient;
    }
    down
    {
        type            zeroGradient;
    }
    cold
    {
        type            zeroGradient;
    }
    sides1
    {
        type            cyclic;
    }
}
// ************************************************************************* //
```

ft – mixture fraction

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          ft;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 0 0 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    wallHot
    {
        type            zeroGradient;
    }
    up
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
    down
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
    cold
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
    sides1
    {
        type            cyclic;
    }
}
// ************************************************************************* //
```

ftVar – variance of the mixture fraction

```
/*--------------------------------------------------------------------------*\
| =========                 |                                                |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
|  \\    /   O peration     | Version:  1.4                                  |
|   \\  /    A nd           | Web:      http://www.openfoam.org              |
|    \\/     M anipulation  |                                                |
\*--------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    root            "";
    case            "";
    instance        "";
    local           "";
    class           volScalarField;
    object          ftVar;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    wallHot
    {
        type            fixedValue;
        value                uniform 0;
    }
    up
    {
        type            fixedValue;
        value           uniform 0;
    }
    down
    {
        type            fixedValue;
        value           uniform 0;
    }
    cold
    {
        type            fixedValue;
        value           uniform 0;
    }
    sides1
      {
        type            cyclic;
    }
}
// ************************************************************************* //
```