Lehrstuhl für THERMODYNAMIK

Hand Out of the Praktikum

Computational Thermo-Fluid Dynamics with Open Source Tools

W. Polifke et al.

Status: 7th April 2012

Lehrstuhl für Thermodynamik
Prof. Dr.-Ing. T. Sattelmayer
Prof. W. Polifke Ph.D. (CCNY)
www.td.mw.tum.de

It would have been impossible to create
this course without the contribution of:
Danilo Bruno, Sebastian Bomberg, Alejandro
Cardenas, Joao Carneiro, Frederic Collonval,
Patrick Dems, Florian Ettner, Tobias Holzinger,
Volker Seidel and Johannes Weinzierl.
We are grateful for the time they spend
to design the test cases and write this
hand out.

Contents

Objective of the Lab

Computational Fluid Dynamics (CFD) is the prediction and analysis of fluid flows by numerical simulation. Appropriate algorithms are used to determine approximate solutions to the equations describing the fluid motion. CFD is used for research of fundamental kind and for practical development of any kind of engines, where fluid flow plays a role. With increasing computational power and decreasing development cycles of industrial products CFD becomes more and more an important tool within the design process. It is used in very different fields to predict and analyze the flow dynamics in very early stages of the development and design process. Its advantages are the low cost insight in the fluid dynamics of the considered system compared to experimental setups. Even more the specific negligence or consideration of individual physical effects leads to more detailed understanding and analysis of the system. Although numberless advantageous and successful application of CFD in research and development, the accurate description of complex physical phenomena as combustion, multiphase flows or highly resolved turbulent flows with CFD is still a very ambitious challenge.

Beside the several commercial CFD software tools, OpenFOAM®, which is a C++ library based program package, becomes more and more popular. This is due to several reasons. First, OpenFOAM® is an open source as well as an open code software which is interesting for research and industrial application in two ways. On the one hand the open code allows to implement and modify the models, algorithms and solution processes in a nearly unlimited latitude which is very helpful and beneficial for research. On the other hand the free availability of the software is interesting for both, universities and industry. Basically for large concerns which need a lot of licenses or small companies which are not able to afford commercial CFD software is OpenFOAM® an interesting alternative. Also increasing application at the universities and growing experience with OpenFOAM® encourages the industrial partners to deal with this software.

Considering these developments and the fact, that more and more offers for Bachelor and Master theses contain the application of OpenFOAM®, the students demand for a professional introduction to this software in the scope of their academic studies is remarkable. Therefore the aim of this CFD course is to give a guided introduction to OpenFOAM®. Starting with the very first steps of computational fluid dynamics, this course should provide a fundamental overview and insight into the program structure, the solvers, the tools and also a brief introduction to the code itself and the implementation of new equations, models or applications. It is not objective to impart knowledge of basic fluid dynamics and thermodynamics, meshing tools, theoretical knowledge of finite-differences/elements/volume-methods or advanced C++ programming skills.

Script Structure

This script provides the description of the weekly problem including physical and numerical background, specific explanations concerning OpenFOAM® all information needed for the handling of the exercises and additional sources providing related information to the problem itself and its context. The following listing gives a more detailed description of structure of each chapter.

- **Todays Problem**
  To ensure a wide variety the course is divided into several individual and independent examples and simulation cases. Every week a different topic is presented, beginning with simple introduction examples and going further to more complex cases. They treat different aspects of CFD and OpenFOAM®, whereas the learned knowledge of one is not implicitly necessary for another one in every case. This section presents the background of the weekly topic. It describes the motivation for the simulation of the specific physical processes and its application in research or industry.

- **Physics of the Problem**
  This section is used for a brief derivation and introduction to the physics, which describe the problem. These are often simplified models which approximate the real problem. As the consideration of all physical effects at all scales in time and space is usually impossible, impractical or not required for solving a specific task, assumptions have to be made, which simplify the complexity of the physical processes. This leads to a reduced number of the governing equations. Furthermore several simplifications are necessary to reduce the complexity of the equation system itself. Sometimes even vague estimates must be made to close the problem, for example for unknown boundary conditions.

- **Numerics of the Problem**
  Here theoretical informations are provided about the methods used to solve the problem. This is at first an evaluation of the general approaches to solve the specific type of fluid flow discussing their advantages and shortcomings. This includes amongst others the requirements for the mesh, the boundary conditions, the solving process and the ratio of accuracy and computational effort. After that a detailed description of the methods used here is given. Also several hints to solve the numerical part of the exercises is provided.

- **OpenFOAM® specific stuff**
  To match the aim of this course numerous case-specific information about the solver structure, the used discretization schemes, related script settings, boundary conditions and data output options are listed with a detailed description of their function and correct specification. As programming is not a straightforward work, where problems can be solved in one way, it is also shown how the methods are realized in OpenFOAM® and how to implement new ones.

- **Extra Practice and Background Information**
  This section is used to provide further tutorials related to the topic and/or information about alternatives and advanced techniques whose disquisition would exceed the scope of this script and course.

Chapter

1 Introduction into Linux and OpenFOAM®

Bibliography

[1] Sevier E., Spainhour S., Figgins S. and Hekman J.P.: *Linux in a Nutshell* O'Reilly, 3rd edition

[2] http://en.opensuse.org/Tutorials *OpenSuse Tutorials*

[3] http://foam.sourceforge.net/doc/Guides-a4/UserGuide.pdf *OpenFOAM User Guide*, Version 1.6, 24th July 2009

[4] http://foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf *OpenFOAM Programmers Guide*, Version 1.6, 24th July 2009

[5] http://openfoamwiki.net/ *OpenFOAM Wiki*

The first sections of this chapter give a brief introduction into the necessary software tools that are not part of the CFD program OpenFOAM®. The second part offers an explanation of how to get a first OpenFOAM® solution without any detailed background.

1.1 Introduction into Linux

The basic software of a personal computer is a so-called operating system. Over the last decades three main tree's developed. Linux is a derivative of Unix and has the great advantage of being an open source software. The chair of thermodynamics currently uses the distribution of "OpenSUSE" , version 11.x.

As OpenFOAM® has no graphical user interface (GUI), we have to use some tools provided by the Linux operating system. The chosen software is only a possible choice and is adaptable by the users requirements.

1.1.1 xTerm - Konsole

In contrast to any GUI based software OpenFOAM® commands have to be started in a text-based window, which is called "Konsole" or "Terminal". The start button is placed at:

Applications → *System* → *Terminal* → *Terminal* .

This Konsole is our main operating window for using OpenFOAM® and provides a lot of Linux commands that are needed to work with the CFD software. The most relevant are listed in table 1.1.

A more detailed help for the different commands is available by typing *man $COMMAND*, *$COMMAND –help* or *$COMMAND -h*. Further information can be found in the Internet [2] or [1].

Table 1.2 shows some shortcuts and special characters that might be used in a Konsole.

ls	List all files
cd [$1]	Change to directory [$1]
mkdir [$1]	Create a new directory [$1]
cp [$1] [$2]	Copy file [$1] to [$2]
cat [$1]	Output text file [$1] to screen
less [$1]	Output text file [$1] to screen and enable scrolling
tail -n [$1] [$2]	Show the last [$1] lines of text file [$2]
head -n [$1] [$2]	Show the first [$1] lines of text file [$2]
top	show the processes running on the computer
bg, fg	set a process to back- or foreground
echo [$Variable]	show value of variable [$Variable]
rm [$1]	remove a file or directory (-r)

Table 1.1: A list of typical linux commands

$[Ctrl] + c$	Abort active process
$[Ctrl] + z$	Pause active process
$[Ctrl] + n$	Open a new Tab
$[Tab]$	Auto complete
$\sim$	The "home " path
$\&$	Run command and set it to background
$>$ [$log]	Forward output to [$log] file
./[$1]	Execute file [$1]

Table 1.2: Shortcuts and special characters for a linux *Terminal*

1.1.2 Kate - An Enhanced Text Editor

Typically all software programs are started by typing a special command in a terminal. As we need an editor for manipulating the text based files of OpenFOAM® we want to start Kate from the *Terminal* by typing:

```
kate
```

Kate is only one of a bunch of editors provided by SUSE. One of its advantages is the possibility to create sessions out of a collection of files. As every OpenFOAM® case consists of ten to hundreds of files this application allows us to structure our files and have them opened in only one working window.

Further common editors of Linux distributions are vi, vim (both terminal based), emacs and kwrite. Dolphin and Konqueror are helpful file management tools for opening, copying etc.

1.2 Introduction into OpenFOAM®

OpenFOAM® offers a large variety of solvers for a lot of different engineering problems, like heat transfer, mechanical engineering, fluid flow applications and even financial problems. The first kind of problems will be as well part of this student's lab as fluid flow phenomena.

1.2.1 Installation

Before using OpenFOAM® this section gives a brief description of how to install the software on a personal computer. The minimum recommendations for this installation is a 32 or 64 bit CPU with 2.0 GHz and 1 GB RAM. Additionally approximately 15 GB free disk space is necessary for the installation. We recommend to use a computer with a Debian-based distributions (like Ubuntu and Kubuntu) as the installation procedure is the easiest. But any other Linux distributions is fine for example at the chair you will use OpenSUSE.

The installation procedures are well described on `http://www.openfoam.com/download/`. The first one is only for Debian users and is the easiest. The second one requires to download the sources then to compile them. For that, the .tar balls have to be downloaded from `http://www.openfoam.com/download/source.php` and saved to `"$HOME/OpenFOAM/"`. Now all files must be uncompressed by typing

```
tar -xzvf $1,
```

starting with *OpenFOAM-2.0.1.gtgz* and followed by *ThirdParty-2.0.1.gtgz*. Then you will have to source the new data by typing in a terminal:

```
source $HOME/OpenFOAM/OpenFOAM-2.0.1/etc/bashrc
```

Before compiling the code, check your system by executing `foamSystemCheck`. Then if no errors occur, compile the code by typing the following commands:

```
cd $WM_PROJECT_DIR
./Allwmake
```

More advanced Linux users may follow the 'Git repository' installation for having an "up to date" version of the software.

Whatever version you have installed, you have to remember that during this course, you will use the version 2.0.1. And some exercises are not working with an earlier/latter version.
Remark: The latest version is maybe incompatible with some description in this manuscript. However on `http://www.openfoam.com/download`, there is a link to download old versions. You should find the right one there. Remark: The latest version is maybe incompatible with some description in this manuscript. However on `http://www.openfoam.com/download`, there is a link to download old versions. You should find the right one there.

When the installation is finished, the following commands have to be executed:

```
echo ". ~/OpenFOAM/OpenFOAM-2.0.1/etc/bashrc" >>~/.bashrc
mkdir -p $FOAM_RUN
bash
```

The success of the installation should now be tested by opening a new Terminal and typing: `foamInstallationTest > log.foamInstallationTest`

If the output of this file shows in the summary at the end of `log.foamInstallationText`; `Criticalsystems ok.`. The installation is successfully completed.

The OpenFOAM® version used during the lab is a server based 2.0.1 version. The whole installation process is already finished and all specifications should already exist. Therefore no installation to the local user profile is necessary.

1.2.2 Structure of a Case

This section gives a short overview of a typical OpenFOAM® problem. Case specific differences will be declared and explained later.

When the software is installed a first short test-case should be run. At first we copy a tutorial case into our user folder[1]:

```
mkdir -p $FOAM_RUN
mkdir  $FOAM_RUN/myFirstTestCase
cp -r $OFP/Chpt1/cavity/* $FOAM_RUN/myFirstTestCase/
cd $FOAM_RUN/myFirstTestCase
```

This folder should now consist of different files and folders that can be listed by
`~/OpenFOAM/user-2.0.1/run/myFirstTestCase > ls *`

This command should create the following output:

```
0:
p  U

constant:
polyMesh  transportProperties

system:
controlDict  fvSchemes  fvSolution
```

. The `system` folder contains the main control options of the solving process. The `constant` folder contains all settings that are time independent, like the meshing information, constant properties and turbulence settings. At first the `system` folder is considered.

The main control file is named `controlDict` and is, like all non-compiled OpenFOAM® files, written in C++ language. The `fvSchemes` file sets the different discretization schemes for all resolved differential operators. The last file, `fvSolution`, controls the internal solvers of the subsequently running equations to solve. Both files will be described in detail later.

[1] Normally you can refer to the tutorials folder of OpenFOAM® by using the environment variable `$FOAM_TUTORIALS`. But as some additional files have been added to the basic *cavity* case you will get access to the files in another place

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
application     icoFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         0.5;

deltaT          0.005;

writeControl    timeStep;

writeInterval   20;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;

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

Listing 1.1: The main control file: controlDict

The `controlDict` of the first test case should look like the one displayed in listing 1.1. The commented header is followed by the name of the OpenFOAM® solver that has to be used for the actual case.
The following lines contain switches for the most important simulation settings. They control the timing, outputs, beginning and ending of the case. Their values might be looked up in table 1.3 which can be found in the online documentation.

Time control	
startFrom	Controls the start time of the simulation.
- firstTime	Earliest time step from the set of time directories.
- startTime	Time specified by the *startTime* keyword entry.
- latestTime	Most recent time step from the set of time directories.
startTime	Start time for the simulation with *startFrom startTime*;
stopAt	Controls the end time of the simulation.
- endTime	Time specified by the *endTime* keyword entry.
- writeNow	Stops simulation on completion of current time step and writes data.
- noWriteNow	Stops simulation on completion of current time step and does not write out data.
- nextWrite	Stops simulation on completion of next scheduled write time, specified by *writeControl*.
endTime	End time for the simulation when *stopAt endTime*; is specified.
deltaT	Time step of the simulation.
adjustTimeStep	Some solvers adapt dynamically the time step. This boolean (value is `on` or `off`) allows the function or not.
maxCo	The criteria to adapt the time step is based on the Courant number, *Co*. This parameter sets the maximal value of it.
maxDeltaT	[optional] If the time step changes dynamically, this parameter sets the upper limit.
Data writing	
writeControl	Controls the timing of write output to file.
- timeStep†	Writes data every *writeInterval* time steps.
- runTime	Writes data every *writeInterval* seconds of simulated time.
- adjustableRunTime	Writes data every *writeInterval* seconds of simulated time, adjusting the time steps to coincide with the *writeInterval* if necessary used in cases with automatic time step adjustment.
- cpuTime	Writes data every *writeInterval* seconds of CPU time.
- clockTime	Writes data out every *writeInterval* seconds of real time.
writeInterval	Scalar used in conjunction with *writeControl* described above.

purgeWrite	Integer representing a limit on the number of time directories that are stored by overwriting time directories on a cyclic basis. Example of $t_0 = 5s$, $\Delta t = 1s$ and *purgeWrite* 2;: data written into 2 directories, 6 and 7, before returning to write the data at $8s$ in 6, data at $9s$ into 7, etc. To disable the time directory limit, specify *purgeWrite* 0;† For steady-state solutions, results from previous iterations can be continuously overwritten by specifying *purgeWrite* 1;
writeFormat	Specifies the format of the data files.
- ascii†	ASCII format, written to *writePrecision* significant figures.
- binary	Binary format.
writePrecision	Integer used in conjunction with *writeFormat* described above, 6† by default
writeCompression	Specifies the compression of the data files.
- uncompressed	No compression.†
- compressed	gzip compression.
timeFormat	Choice of format of the naming of the time directories.
- fixed	$\pm m.dddddd$ where the number of ds is set by *timePrecision*.
- scientific	$\pm m.dddddde \pm xx$ where the number of ds is set by *timePrecision*.
- general†	Specifies scientific format if the exponent is less than -4 or greater than or equal to that specified by *timePrecision*.
timePrecision	Integer used in conjunction with *timeFormat* described above, 6† by default
graphFormat	Format for graph data written by an application.
- raw†	Raw ASCII format in columns.
- gnuplot	Data in gnuplot format.
- xmgr	Data in Grace/xmgr format.
- jplot	Data in jPlot format.
Data reading	
runTimeModifiable	yes†/no switch for whether dictionaries, e.g.*controlDict*, are re-read by OpenFOAM® at the beginning of each time step.

Run-time loadable functionality	
libs	List of additional libraries (on `$LD_LIBRARY_PATH`) to be loaded at run-time, e.g.("libUser1.so" "libUser2.so")
functions	List of functions, e.g. probes to be loaded at run-time; see examples in 6.3.2 or `$FOAM_TUTORIALS`

Table 1.3: Settings for the *controlDict* file († denotes default entry if associated keyword is omitted).

The *constant* Folder

As mentioned above the *constant* folder contains time independant settings for the simulation. The *polyMesh* folder consists of different files describing the meshing of the considered problem. The most important entry of this subfolder is the *boundary* file.
The boundary file is a list of all defined boundaries and their basic boundary type. The basic boundary types are `wall`, `patch`, `wedge`, `cyclic`, `mapped` and `symmetryPlane`. All further boundary settings have to fit to these settings.

The folder itself further contains two additional files: *RASProperties* and *transportProperties*. The first sets the parameters for the computation of the turbulence model. The latter describes the basic properties of the material etc. Here the entry `nu` values the kinematic viscosity of the fluid. The appended field of seven values defines the powers of the basic SI units in the following order:

$$[\mathrm{kg} \quad \mathrm{m} \quad \mathrm{s} \quad \mathrm{K} \quad kgmol \quad \mathrm{A} \quad \mathrm{cd}] \ .$$

For example our parameter is declared by:

```
nu              nu [ 0 2 -1 0 0 0 0 ] 0.01;
```

. The kinematic viscosity ν of our fluid has the unit m^2/s. The last value gives the scalar value of the parameter. Here we have a value of 0.01.

The *0* Time Folder

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type            zeroGradient;
    }

    fixedWalls
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}

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

Listing 1.2: The pressure initial value file `p`

This folder provides the boundary and start conditions of all used parameter-fields. Exemplary we inspect the pressure p file in detail (see listing 1.2) The first lines equal the header of the *controlDict* file. The next settings provide informations about the current case and the files content and the variable name.

The `internalField` either consists of one `uniform` value for each scalar, vector or tensor entry or a non-uniform vector with as many values as cells defined by the meshing.

The last part of the file is the `boundaryField` structure. It defines the condition at each boundary of the meshed domain. `type` has to be consistent to the *boundary* file discussed above. Typical types are `fixedValue` (Dirichlet boundary condition), which, like in the internal field, provide a value for each surface patch, `fixedGradient` or `zeroGradient` which correspond to a Neumann boundary condition (**BC**) or `symmetryPlane`. For two dimensional problems the third direction entries are declared to be `empty`.

All fields are stored in similar files. Each time the output is written to a new folder and files of the current values are created.

1.2.3 Starting OpenFOAM®

Before running the case, the mesh has to be created. For that, type the following command in a terminal :

```
cd $FOAM_RUN/myFirstTestCase
blockMesh
```

After preparing or pre-processing the case the simulation can be started. We simply type the name of the solver to use, here:

```
icoFoam
```

and a lot of "prompt" is printed on the Terminal. When the simulation end is reached the cursor sign comes up again. Typing `ls` shows us a variety of new folders named by the software systematically.

After the simulation we need to convert the data for a post-processing software. Here we create files in 'paraview' format. This GUI based software is an open source tool and part of the OpenFOAM® third-parties package. It is the best program to get an first post-processed impression of the simulation results and it is shortly described in the next section.

As a lot of text output is created during a simulation, and the display is much to fast for the human eyes we prefer to forward the text to a log-file which can be observed independently. The standard linux way would be to extend the commands with, for example:

```
icoFoam > log.icoFoam &
```

. OpenFOAM® provides a further method, which is used in the `Allrun` script. Here all necessary commands are saved and starting it will execute the commands one after the other and create log files of the format `log.$APPLICATION` for each command. The `Allclean` script should restore the initial form of the case.

After running both scripts the folder should have the same content like after the first simulation run plus additional logs. Typing `less log.icoFoam` shows the simulation output. Here we can read the initial and final residuals for each solved equation for each simulation time step. Further the number of iterations needed by the internal solver to reach the residual is displayed at the end of the lines. Pressing *q* closes the display and the text mode is back.

1.2.4 Post-Processing

At first we have to convert the results into a data format that is compatible with *paraview*. For creating the files, we simply execute:

```
paraFoam -touch
```

and a `myFirstTestCase.OpenFOAM` construct is saved to the case directory.

Now we start the post-processing tool by executing `paraview&`. When the GUI is loaded we click *'File'* → *'Open'* and select `myFirstTestCase.OpenFOAM`.

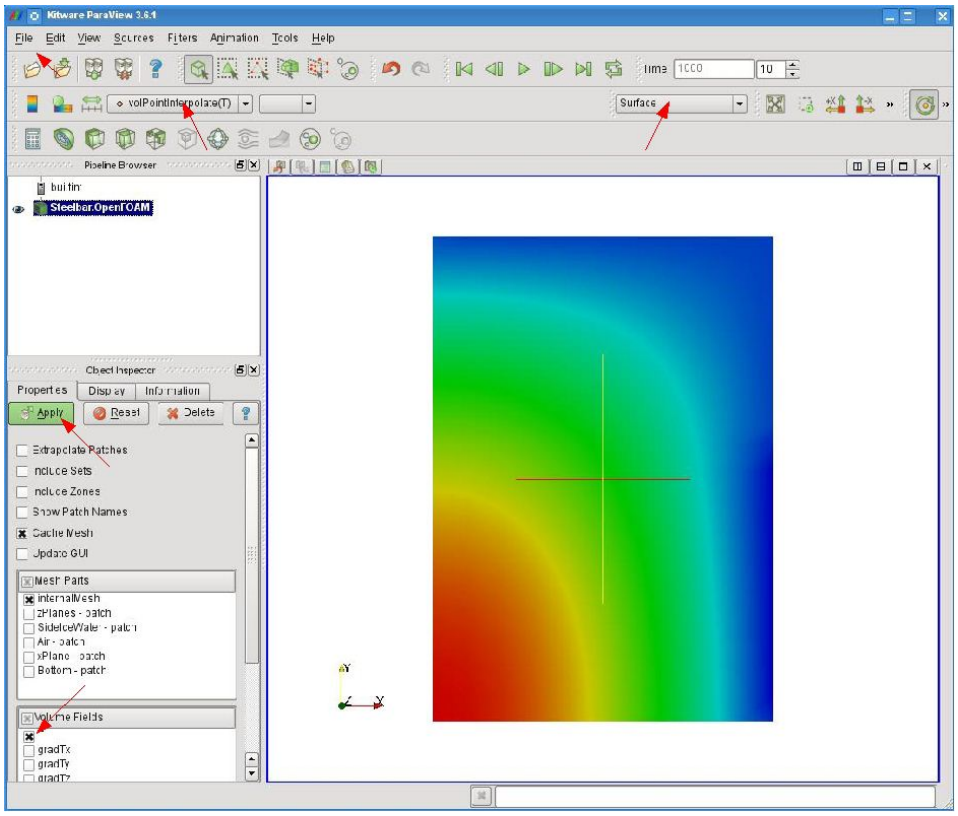


Figure 1.1: The Paraview GUI: The red arrows point on important buttons

A shorter way to do those actions is to type the command `paraFoam` without options.

After pressing '*Apply*' the mesh is loaded and displayed in the initial conditions. The drop down menu on the upper left allows us to select which computed field is displayed. A second button sets the type of view (Mesh, Points, Surface,...). Pushing the play button runs the results in a kind of film.

Exercises

1.1 Follow the instructions given in the text.

1.2 Manipulate the controlDict file to create a different output, timestep, etc.

1.3 Reverse the swirl by changing the boundary values in the velocity `U` file.

1.4 Try to get comfortable with the Paraview GUI. Cut the domain using the *Slice* filter to look at the pressure field in the mid-XY plan. In this plan, visualize the velocity field using the *Glyph* filter.

1.5 Run `Allclean` before logout. And check that your case is properly cleaned.

1.3 Extra Practice and Background Information

Extra Practice

Additionally to this first contact with OpenFOAM® and paraView, we advice you to do the first tutorial of the User Guide up to and including the *2.1.4 Post-processing* paragraph.

Background Information

1.3.1 Sources for documentation

The following section lists sites that can provide you more information about OpenFOAM® and the other free tools useful in the frame of Computational Thermo-Fluid Dynamics.

1. Official websites
   - openfoam.com: `http://www.openfoam.com/`
     The official website of OpenCFD®, the company that releases OpenFOAM®
   - User Guide: `http://www.openfoam.com/docs/user/`
     The official user guide
   - C++ source guide: `http://www.openfoam.com/docs/cpp`
     Link to the Doxygen documentation for the lastest version
2. Community websites
   - The OpenFOAM®-Extend Project: `http://www.extend-project.de/`
     The website of the biggest community project of OpenFOAM®. This website is supported actively by some of the first developers of OpenFOAM®. Their goal is to gather in the same place the contribution of the community. It will become (website launched mid-2010) the biggest unofficial source of information and ressources (tools, test cases, validation, bug tracking,...).
   - Unofficial OpenFOAM® wiki: `http://openfoamwiki.net/index.php/Main_Page`
     Everything is said in the title :)
   - CFD-online -> OpenFOAM®: `http://www.cfd-online.com/Forums/openfoam/`
     With the link you will have access to the forum of OpenFOAM®. You have to register to the website to post messages. But it is really worthy.
   - Foam CFD: Collaborative Open-Source CFD: `http://www.foamcfd.org/`
     Old site for the community but not updated since 2008.
3. Useful tools
   - paraView: `http://www.paraview.org/`
     The free post-processing software shipped by default with OpenFOAM®. Its wiki provides help and documentation: `http://paraview.org/Wiki/ParaView`.

- gnuplot: http://www.gnuplot.info/
  The command-line graphical tools.
- matplotlib: http://matplotlib.sourceforge.net/index.html
  matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats (Python base system).

4. Universities using OpenFOAM®

   - Lehrstuhl für Technische Thermodynamik der Universität Rostock: http://www.ltt-rostock.de/mediawiki/index.php/OpenFOAM
     This departement has a wiki on OpenFOAM® (in German). In addition, their tools are available through a svn repository. Unfortunately it looks that the wiki has some problems (or new restrictions?). So if you are interested, you can download the full content of the SVN repository typing in a terminal:

     ```
     svn co --username gast --password ""
     https://janus.fms.uni-rostock.de/svn/repository/OpenFOAM/
           trunk/LTTRostockExtensions
     ```

   - Department of Applied Mechanics at Chalmers University: http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/
     That departement offer a "PhD course in CFD with OpenSource software". The main advantage is that all ressources are available: teaching hand-out, presentations and reports of the attendees, additional links and information.
   - Faculty of Mechanical Engineering (UNIZAG FSB; http://www.fsb.hr/), University of Zagreb (http://www.unizg.hr/), hosts one of the original OpenFOAM® developer: Professor Hrvoje Jasak. He is still the leading developer of the community version OpenFOAM-extend. He organizes a summer school for advanced users of OpenFOAM® every September. For more information, here is the link to the Summer School 2011 website: http://www.fsb.hr/?OpenFOAM_Summer_School_2011.

Chapter

2 Heat Transfer in a Plate

2.1 Todays Problem

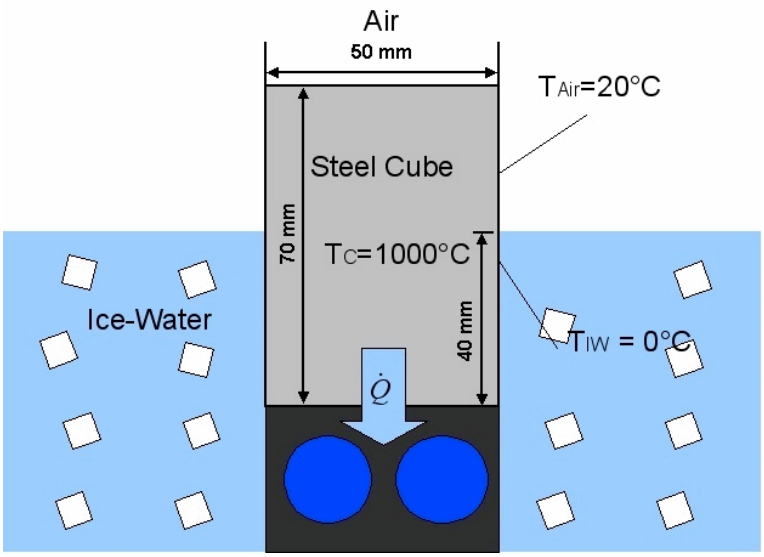


Figure 2.1: Heat transfer in the cross-section of a infinitely long steel profile

Consider a plate like the one in figure 2.1. It consists of steel, has an initial temperature of $1000°\mathrm{C}$ and is cooled by air of $20°\mathrm{C}$ in the upper part and a water-ice mixture $0°\mathrm{C}$ in the lower part. As it is a symmetrical problem, only the half right side of the plate will be modeled to reduce the computational cost. The bottom boundary is cooled with a constant heat flux[1] as $\dot{q}_{lin} = 200\mathrm{W/m^2}$.

Bibliography

[1] Wäermeübertragung, *Polifke and Koptiz, Pearson, 2nd edition, 2009*

[2] Fundamentals of Heat and Mass Transfer, Incropera and DeWitt, *John Wiley & Sons 1996*

[3] Script: Grundlagen thermo-fluiddynamischer Simulation, Polifke, Kopitz and Holzinger, TU München *2010*

[4] Script: Finite-Volumen-Methode in der Numerischen Thermofluiddynamik, Baumann et al.,TU Berlin *2006*

[1] So the global flux passing by the bottom boundary is equal to $200 * 0.05 = 10\mathrm{W}$ per meter unit in the infinite direction

2.2 Physics

2.2.1 Fourier's Law

Transient heat transfer problems are described by Fourier's differential equation, which will be derived for the 2D case in this chapter.

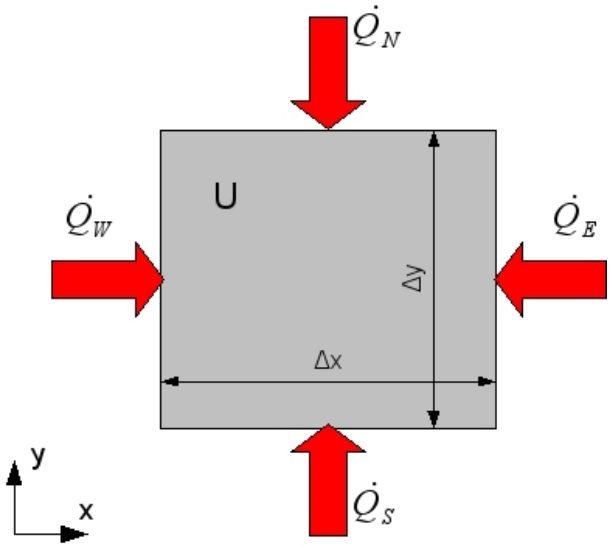


Figure 2.2: Heat transfer in an infinite small plate.

Consider the infinite small plate of the dimensions $\Delta x \times \Delta y$ shown in figure 2.2 with a depth of Δz. An energy balance leads to:

$$\frac{dU}{dt} = \sum \dot{Q}_i \tag{2.1}$$

The inner energy U of the solid material is defined by

$$U = mc_v T$$

, and its temporal derivative can be written by substituting the volume $V = \Delta x \Delta y \Delta z$ and density ρ as:

$$\frac{\partial U}{\partial t} = \rho \Delta x \Delta y \Delta z c_v \frac{\partial T}{\partial t} \tag{2.2}$$

. The heat fluxes $\dot{Q}_i$ are developed by $\dot{Q}_i = A_i \dot{q}_i$ and inserting Fourier's law

$$\dot{q}_i = -\lambda \frac{\partial T}{\partial x_i} \tag{2.3}$$

. Here x_i stands for x and y respectively. Inserting these expressions lead to

$$\rho \Delta x \Delta y \Delta z c_v \frac{\partial T}{\partial t} = \Delta y \Delta z \lambda \frac{\partial T}{\partial x} + \Delta x \Delta z \lambda \frac{\partial T}{\partial y}$$

Dividing this equation by the spacial variables and setting the limit yields

$$\rho c_v \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(\lambda \frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda \frac{\partial T}{\partial y}\right) \tag{2.4}$$

.

Introducing the thermal diffusivity $a = \frac{\lambda}{\rho c_v}$ this equation can be rewritten in tensor form :

$$\frac{\partial T}{\partial t} = \nabla \left(a \nabla T\right) \tag{2.5}$$

This equation is of the Laplacian type, which is valid for other physical problems, e.g. chemical diffusion as well. Generally the right hand side of this equation is called diffusive term.

2.3 Numerics

The continuous differential equation 2.5 is discretized by the Finite Volume Method (FVM) and has the following form

$$\begin{aligned} a_P T_P = \quad & \Psi \left(a_N T_N + a_E T_E + a_s T_S + a_W T_W\right) + \\ & (1-\Psi)\left[a_N T_N^0 + a_E T_E^0 + a_s T_S^0 + a_W T_W^0 + \right. \\ & \left. \left(a_P^0 - a_N - a_E - a_S - a_W\right) T_P^0\right] + b \end{aligned} \tag{2.6}$$

for a control volume in a two dimensional case. Here the subscripts mean:

P at this point, and in correspondence to
N the northern
W western
S southern
E eastern

cell. a_i is a factor depending on the discretization schemes. b stands for a constant term, which may come up with source terms. The superscript 0 identifies the current time step. Ψ depends on the time discretization method.

2.3.1 Temporal discretization schemes

All differential equations describing a transient problem consist of a time derivative term, generally of first order. A weighted integration from the actual time step to the next leads to the next point P value. There are four schemes (see [3, 1] for details) implemented in OpenFOAM® (see Table 2.1)

In OpenFOAM®, the time scheme doesn't determine whether the equations are solved implicitly or explicitly (See paragraph 2.4.2 for more explanation).

Time schemes	Description
`steadyState`	The term is not discretized and the corresponding sum terms of a_p, a_p^0 become 0.
`Euler`	The time derivation is computed from the current and the future time steps.
`backward`	The time derivative is computed from the previous, the current and the future time steps.
`CrankNicholson` Ψ	The current and future steps are blended, $\Psi = 0$ stands for an explicit formulation, $\Psi = 1$ for a fully implicit treatment and $\Psi = 0.5$ is the standard Crank Nicholson method.

Table 2.1: Time schemes in OpenFOAM®

2.3.2 Spatial discretization schemes

Laplacian Schemes

Laplacian operators are recomputed by a `Gauss`ian integration. This reduces the order of derivatives and creates a further term containing the fluxes over the surface, or , in other words, surface normal gradients. When `Gauss`ian integration is used, only the interpolation scheme for the approximation of the surface values has to be selected. Here we commonly choose a `linear` interpolation. Furthermore, if we select an `uncorrected` surface normal gradient method, this would correspond to a central difference scheme (CDS).

For completeness we introduce the two most common schemes for divergence and gradient terms.

Gradient Schemes

Gradient schemes, for example emerge with the pressure term of the momentum equation. Again, a `linear Gauss`ian integration is used for standard computations. Further possibilities are a `leastSquares` method of second or `fourth` order. All of them might be `limited` followed by a gradient scheme.

Divergence Schemes

Divergence Terms are very common in CFD. They especially describe the convective part of a differential equation. As the stability and exactness of the computation is strongly coupled to the selected scheme, care has to be taken. At first a stable low order method should be used before a more accurate scheme is selected.

The simplest method again is the CDS, which corresponds to a `linear Gauss`ian integration. However this method is not stable, and thus the `upwind` scheme should be used as standard method.

Considering the differential equation in Φ

$$\frac{d}{dx}(\rho u\Phi) = \frac{d}{dx}\left(\Gamma\frac{d\Phi}{dx}\right) \tag{2.7}$$

the CDS leads to

$$(\rho u\Phi)_E - (\rho u\Phi)_W = \left(\Gamma\frac{d\Phi}{dx}\right)_E - \left(\Gamma\frac{d\Phi}{dx}\right)_W \tag{2.8}$$

or by splitting into piecewise linear profile $\Phi_E = \frac{1}{2}(\Phi_E + \Phi_P)$, $\Phi_W = \frac{1}{2}(\Phi_P + \Phi_W)$ to

$$\frac{1}{2}(\rho u)_E(\Phi_E + \Phi_P) - \frac{1}{2}(\rho u)_W(\Phi_P + \Phi_W) = \frac{\Gamma_E}{\delta x_E}(\Phi_E - \Phi_P) - \frac{\Gamma_W}{\delta x_W}(\Phi_P - \Phi_W) \tag{2.9}$$

respectively. Reorganizing this equation in the standard manner it can be written as

$$a_P\Phi_P = a_W\Phi_W + a_E\Phi_E \tag{2.10}$$

, with $F = \rho u$ and $D = \frac{\Gamma}{\delta x}$ the parameters a_i become:

$$a_W = D_W - \frac{F_W}{2}, a_E = D_E - \frac{F_E}{2} \tag{2.11}$$

$$a_P = D_E + \frac{F_E}{2} + D_W + \frac{F_W}{2} = a_W + a_E + (F_E - F_W) \tag{2.12}$$

Here the possibility of a change in the signs of F_i and as a consequence of a_i, causes the instability.

When the upwind scheme is used, the convective term on the left-hand side of equation 2.7 is modeled differently. Depending on the sign of F_i the following formulation is used instead:

$$a_W = D_W - max[F_W, 0]\, a_E = D_E - max[F_E, 0] \tag{2.13}$$

$$a_P = a_W + a_E + (F_E - F_W) \tag{2.14}$$

This causes a fulfillment of the Scarborough (stability if $a_{N,E,W,S} > 0$) criterion and a reduction from second to first order in exactness of the discretization.

2.4 OpenFOAM®

2.4.1 Mesh Generation with blockMesh

OpenFOAM® has a mesh generation routine, which may be called by typing

```
blockMesh
```

in the case folder or with a `-case` option. This utility is text based and only sensible for very simple grids. The main construction file is located in the `constant/polyMesh` folder and called `blockMeshDict` (an example is shown in Listing 2.1). Again this file consists of an OpenFOAM® header and is followed by further substructures.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.6                                   |
|   \\  /    A nd           | Web:      http://www.OpenFOAM.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;   // coordinates are written in decimeter

vertices               // coordinates of the points needed to specify the geometry
(
    (0 0 0)            // point 0
    (5 0 0)            // point 1
    (5 5 0)
    (0 5 0)
    (0 0 0.1)
    (5 0 0.1)
    (5 5 0.1)
    (0 5 0.1)          // point 7
);

blocks
(
// definition of a mesh block
// type / list of points / number of cells in each direction / grading in the mesh
    hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (1 1 1)
);

edges
(
);
```

```
boundary              // list of boundary faces
(
    movingWall
    {
        type    wall;
        faces
        (
            (3 7 6 2)    // list of points defining the boundary
        );
    }
    fixedWalls
    {
        type wall;
        faces
            (
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
        );
    }
    frontAndBack  // faces with the normal in the 3rd dimension
    {
        type    empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);

mergePatchPairs
(
);

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

Listing 2.1: Example of a `blockMeshDict` for a plate of 50 cmx 50 cm

At first the scaling of the following geometric values is set. Setting

```
convertToMeters 0.001;
```

causes all coordinates to be converted from mm to m.

Next all points of interest have to be defined by inserting their coordinates in the `vertices` construct.

```
vertices
(
    ( 1000    0    0)
)
```

for example creates a vertex at $1\mathrm{m}$ in x-direction. In the case of a later reference, the n vertices are numbered from 0 to $n-1$.

The created points are now used to create `blocks` that get meshed later. The user can choose `hex` for hexaedral blocks, `wedge` for wedge shaped geometries in rotational symmetric 2D-simulations, `prism` for prisma shape or `tetra` for a tetrahedral form. However always eight vertices have to be named for each block. When

non-hexaedral blocks are created some vertices have to be called more than once to create a degenerated surface or line.

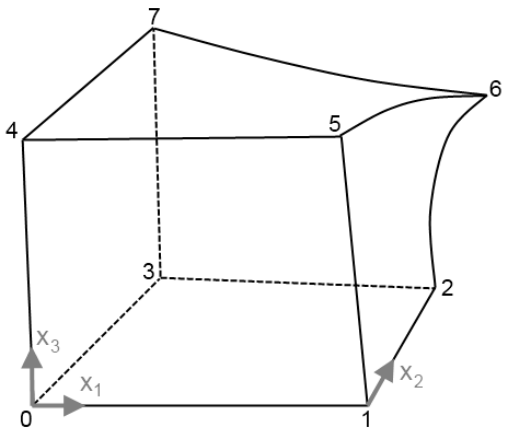


Figure 2.3: Vertex Numbering in a hexaedral block

The local coordinate System of the block has to be right hand sided. This is fulfilled if: The vector from the first vertex to the second vertex points into the positve x_1-direction , the second and third into x_2-direction. Those three vertices in combination with the fourth vertex create a plane perpendicular to the x_3-direction. The first and fifth vertex then point into x_3-direction. The last three vertices have to be found by moving the vertices two, three and four into the positive x_3-direction. An example of such a numbering might be found in figure 2.3.

The list of points is followed by the number of cells per direction and grading in their direction. This can either be done by `simpleGrading (1 1 1)` where the numbers define the ratio of the last to the first cell edge, or by `edgeGrading`, which is followed by the ratio for each block edge.

After defining the blocks special `edge` configurations might be set. By default all edges are assumed to be straight between two vertices. Further possibilities are `arc, simpleSpline, polyLine, polySpline` and `line`. The syntax can be looked up in the users manual.

The last step of the geometrical definition is the definition of the patches. Each patch consists of a generic patch `type` information and a characteristic name, like mentioned in chapter 1 and is followed by a list of block faces that belong to this patch. All faces are defined by four vertices, that are, observed from the inside, traversed in clockwise direction.

If multiple blocks are used, their connecting faces might be merged. This procedure is described in detail in the users manual.

2.4.2 Setup of the laplacianFoam Solver

```
#include "simpleControl.H"

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

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

    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"

    simpleControl simple(mesh);

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

    Info<< "\nCalculating␣temperature␣distribution\n" << endl;

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

        for (int nonOrth=0; nonOrth<=simple.nNonOrthCorr(); nonOrth++)
        {
            solve
            (
                fvm::ddt(T) - fvm::laplacian(DT, T)
            );
        }

        #include "write.H"

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

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

    return 0;
}
```

Listing 2.2: The laplacianFoam.C main file of the laplacianFoam solver without the commented header of the file. Usually the header consist of standard forms and a short description of the solver.

The laplacianFoam solver of OpenFOAM® solves the laplacian equation

$$\frac{\partial T}{\partial t} = \nabla \left(DT \nabla T \right) \tag{2.15}$$

, where the thermal diffusivity DT is specified in the `constant/transportProperties` file.

The solver is located in the applications folder of the OpenFOAM® installation directory. The main file `laplacianFoam.C` (listing 2.2) is written in `C++`. At first

some standard header files (`setRootCase.H, createTime.H, createMesh.H`) are loaded. Then, the first solver specific header, `createFields.H` (see Listing 2.3) is called.

```
Info<< "Reading␣field␣T\n" << endl;

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);


Info<< "Reading␣transportProperties\n" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);


Info<< "Reading␣diffusivity␣DT\n" << endl;

dimensionedScalar DT
(
    transportProperties.lookup("DT")
);
```

Listing 2.3: The createFields.H file of the laplacianFoam solver

This file defines the fields and parameters of the differential equations to solve.

The first part defines the temperature field, giving an entry for every cell. It is set to be a field of scalars by the `volScalarField` type, has to be read from the runtime dictionary, and is automatically written if a write step is reached.

The second part of this header passes the filename information of the physical parameters to the solver. Here the `transportProperties` file has to be read from the `constant` folder and is not overwritten.
After this dictionary is defined, the diffusivity DT is looked up from this file.

The runtime loop

After reading this file the runtime loop starts, and is executed until the condition defined in the `controlDict` file is fulfilled. Every time step the control switches in the SIMPLE subdirectory of the `systems/fvSolution` file are read. Afterwards equation 2.15 is solved.

Let us observe the equation construct in `laplacianFoam.C` (listing 2.2) a bit more detailed.

```
solve
(
    fvm::ddt(T) - fvm::laplacian(DT, T)
);
```

tells the solver to solve the discretized equation in the form of

$$\frac{\partial T}{\partial t} - \nabla \left(DT \nabla T \right) = 0$$

, but what means the `fvm:` part of this equation?

OpenFOAM® internally reformulates the partial differential equations (PDE) to a linear equation system

$$\mathbf{A} \overrightarrow{x} = \overrightarrow{b} \tag{2.16}$$

. The class `fvm` and its opposite `fvc` defines whether the derivative term is handled implicitly or explicitly. For reliability all time derivatives always have to be handled implicitly, but some spatial derivatives can be treated in both ways. Depending on the choice, the terms are part of the system matrix $\mathbf{A}$ or the right-hand side vector $\overrightarrow{b}$.

Here all terms are treated implicitly.

After solving the equation system the output is created in the `write.H` file.

```
    if (runTime.outputTime())
    {
        volVectorField gradT(fvc::grad(T));

        volScalarField gradTx
        (
            IOobject
            (
                "gradTx",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            gradT.component(vector::X)
        );

        volScalarField gradTy
        (
            IOobject
            (
                "gradTy",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            gradT.component(vector::Y)
        );

        volScalarField gradTz
        (
            IOobject
            (
                "gradTz",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            gradT.component(vector::Z)
        );


        runTime.write();
    }
```

Listing 2.4: The write.H file of the laplacianFoam solver

In this special solver the spatial derivatives are saved separately in addition to the scalar field of T if an output time step is reached. If so, first the gradient of the scalar field is computed. Afterwards all the three components of the gradient vector are saved to the time directory in a separate file named `gradTx`, `gradTy` and `gradTz`. The IO object T is already defined to be stored by the `AUTO_WRITE` option in the `createFields.H` file. The final task of this file is to write the output, which is done by the command `runTime.write();`.

2.4.3 The system Folder

As mentioned in the last chapter the main solver control settings are established in the `system/controlDict` file. The `system` folder also consists of two further files, which will be investigated in this chapter.

fvSchemes

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
    grad(T)         Gauss linear;
}

divSchemes
{
    default         none;
}

laplacianSchemes
{
    default         none;
    laplacian(DT,T) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    T               ;
}
// ************************************************************************* //
```

Listing 2.5: A typical fvSchemes file of the laplacianFoam solver

This file contains the definition of all used discretization schemes that are used, if they come up in the code. Starting with the first and second time derivative, the user can choose the settings from section 2.3.1. However for the second time derivative only `Euler` can be chosen.

As you can see in the `gradientSchemes` construct, not only a default scheme can be specified, but also special terms, of a differential equation.

The `divergenceSchemes` of section 2.3.2 are not the only one implemented. The list in table 2.2 completes the possible items.

Scheme	Numerical behaviour
`linear`	Second order, unbounded
`skewLinear`	Second order, (more) unbounded, skewness correction
`cubicCorrected`	Fourth order, unbounded
`upwind`	First order, bounded
`linearUpwind`	First/second order, bounded
`QUICK`	First/second order, bounded
TVD schemes	First/second order, bounded
`SFCD`	Second order, bounded
NVD schemes	First/second order, bounded

Table 2.2: Interpolation schemes for the divergence terms in OpenFOAM®

The syntax of the `laplacianSchemes` subdictionary has the following form:

```
laplacian(DT,T) <interpolationScheme> <snGradScheme>
```

. Possible interpolation schemes are listed in table 2.3

Centred schemes	
`linear`	Linear interpolation (central differencing)
`cubicCorrection`	Cubic scheme
`midPoint`	Linear interpolation with symmetric weighting
Upwinded convection schemes	
`upwind`	Upwind differencing
`linearUpwind`	Linear upwind differencing
`skewLinear`	Linear with skewness correction
`QUICK`	Quadratic upwind differencing
TVD schemes	

`limitedLinear`	limited linear differencing
`vanLeer`	van Leer limiter
`MUSCL`	MUSCL limiter
`limitedCubic`	Cubic limiter
NVD schemes	
`SFCD`	Self-filtered central differencing
`Gamma` Ψ	Gamma differencing

Table 2.3: Interpolation schemes in OpenFOAM®

If a strictly bounded scalar is computed the limitation might be set by adding the word `limited` in front of the scheme and the boundary values behind the scheme, e.g. `limitedVanLeer 1 2.1`.

Some interpolationSchemes are optimized for vector fields, and can be called by adding a `V`. The most common are `limitedLinearV`, `vanLeerV`, `GammaV`, `limitedCubicV` and `SFCDV`.

The surface normal gradients can be discretized by the schemes listed in table 2.4.

Scheme	Description
`corrected`	Explicit non-orthogonal correction
`uncorrected`	No non-orthogonal correction
`limited` Ψ	Limited non-orthogonal correction
`bounded`	Bounded correction for positive scalars
`fourth`	Fourth order

Table 2.4: Surface normal gradient schemes in OpenFOAM®

The default should be set to `corrected`.

After the laplacian interpolation and surface normal gradient schemes are set, those schemes for detached terms are selected.

Finally if the flux shall be computed in the solver after solving the equation the corresponding field has to be named in the `fluxRequired` field.

fvSolution

```
/*--------------------------------*- 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      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    T
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 2;
}


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

Listing 2.6: A typical `fvSolution` file for the laplacianFoam solver

The `fvSolution` file defines the solvers for the discrete differential equations, that the main solver should compute. This means the solution of equation 2.16 is approximated by different solvers for each PDE of the system.

The three possible solver are `PCG` (symmetric matrices) or `PBiCG` (asymmetric), `smoothSolver` and `GAMG` for generalized geometric-algebraic multi-grid solvers. The `smoothSolvers` can be looked up in the users manual or tutorial cases.

The preconditioning of the matrices might be done with the following keywords:

Preconditioner	Keyword
Diagonal incomplete-Cholesky (symmetric)	`DIC`
Faster diagonal incomplete-Cholesky (DIC with caching)	`FDIC`
Diagonal incomplete-LU (asymmetric)	`DILU`
Diagonal	`diagonal`
Geometric-algebraic multi-grid	`GAMG`
No preconditioning	`none`

After the preconditioning is set the absolute and relative residual tolerances are selected, meaning that either the approximation reaches a residual lower than the absolute value, or the ratio of actual to former residual falls below the `relTol` limit.

After setting all the solver controls, the algorithm settings are made. Depending on the solver either PISO/PIMPLE or SIMPLE (here) algorithms are used in OpenFOAM®. In the `SIMPLE` substructure the relevant parameters are set. They will be discussed later in chapter 4.

The sub-dictionary `relaxationFactors` controls under-relaxation, a technique used for improving stability of a computation, only when solving steady-state problems. Under-relaxation works by limiting the variation of a variable from one iteration to the next, either by modifying the solution matrix and source prior to solving for a field or by modifying the field directly. An under-relaxation factor α, $0 < \alpha < 1$ specifies the amount of under-relaxation, ranging from none at all for $\alpha = 1$ and increasing in strength as $\alpha \to 0$. The limiting case where $\alpha = 0$ represents a solution which does not change at all with successive iterations. An optimum choice of α is one that is small enough to ensure stable computation but large enough to move the iterative process forward quickly; values of α as high as 0.9 can ensure stability in some cases and anything much below 0.2 are prohibitively restrictive in slowing the iterative process. The user can specify the relaxation factor for a particular field by specifying first the word associated with the field, then the factor as shown below.

```
relaxationFactors
{
    T            0.7;
}
```

Exercises

2.1 Create a plate with the measures of figure 2.1.

2.2 Apply the correct boundary conditions to the plate and the initial condition of a constant temperature of $1000°C$. Set the right diffusivity [2].
Remark :

- Here is an example to set a boundary with a fixed value.

```
hotWall
{
        type                 fixedValue;
        value               uniform 400;
}
```

- Here is an example to set a boundary with a fixed gradient.

```
heatedWall
{
        type               fixedGradient;
        gradient              uniform 40;
}
```

2.3 At which time is the steel plate entirely cooled below $200°C$? Plot the evolution of the temperature at the point at the middle of the bottom face of the plate.

2.4 Apply a grading to the plate mesh. The cells have to be finer at the top and the right sides of the plate.

[2] The value of the physical properties for different materials can be found on `http://www.engineeringtoolbox.com/` (Look in this case after the *Carbon Steel*).

2.5 Extra Practice and Background Information

Extra Practice

Changing the boundary conditions

The steel plate stayed on the cooler during 6 minutes. But as nobody was there to move it to another place, it stays on the cooler 6 additional minutes. But during that time, the boundary conditions were different :

- The temperature of the air is $10°\mathrm{C}$
- The ice-water filling by fresh ice stays at $0°\mathrm{C}$.
- The bottom of the cooler due to the thermal inertia give now a flux of $100W/m$ to the steel plate (So the heat flux is inverted and now goes from the bottom to the plate).

What is the structure of the temperature field after 6 minutes of cooling ? And after the 6 additional minutes ? Plot the evolution of the temperature of the middle of the bottom face of the plate during the 12 minutes.

Changing the material

You can change the material from steel to aluminum. How many time is now needed to cool the plate below $200°\mathrm{C}$?

Background Information

2.5.1 Linear solvers and preconditioners

The resolution of a discretized partial differential equation required to solve a linear system $Ax = b$. OpenFOAM® has a number of linear solvers to do so, i.e. iterative solvers for linear sets of equations of the type $Ax = b$. They are listed below:

- diagonalSolver – diagonal solver. Remark : Cannot be selected usually.
- PBiCG – Preconditioned bi-Conjugate Gradient solver for asymmetric lduMatrices using a runtime selectable preconditioner.
- PCG – Preconditioned Conjugate Gradient (CG) solver for symmetric LDU matrices using a runtime selectable preconditiioner.
- smoothSolver – Iterative solver using smoother for symmetric and asymmetric matrices which uses a run-time selected smoother e.g. GaussSeidel to converge the solution to the required tolerance.
  To improve efficiency, the residual is evaluated after every nSweeps smoothing iterations.
  This algorithm is efficient for steady-state simulation.
- GAMG – Geometric Agglomerated Algebraic Multigrid solver.

A preconditioned iterative solver solves the system $M^{-1}Ax = M^{-1}b$, with M being the preconditioner[3]. The goal of the preconditioner is to make sure that convergence for the preconditioned system is much faster than for the original one. For that, M is typically an easily invertible approximation to A. The following types of preconditioners are implemented in OpenFOAM®:

- diagonal – Diagonal preconditioner for both symmetric and asymmetric matrices.
- DILU – Simplified diagonal-based incomplete LU preconditioner for asymmetric matrices. The reciprocal of the preconditioned diagonal is calculated and stored.
- DIC – Diagonal-based Incomplete Cholesky preconditioner for symmetric matrices (symmetric equivalent of DILU). The reciprocal of the preconditioned diagonal is calculated and stored.
- FDIC – Faster version of the DIC Preconditioner in which the reciprocal of the preconditioned diagonal and the upper coefficients divided by the diagonal are calculated and stored.
- GAMG – Geometric agglomerated algebraic multigrid preconditioner.

[3] The example shown is called "left" preconditioning, but also central and right preconditioning exists.

Chapter

3 Heat Transfer In a Complex Geometry

3.1 Todays Problem

The purposes of this second exercise are multiple. You will first modify an existing solver, *laplacianFoam*. Then an introduction to the tool to import a mesh from Fluent® to OpenFOAM® will be presented. And finally, a powerful tool to initialize the internal fields will be described. But let's start with the description of the todays problem.

The system studied is a passive cooler of a chip in aluminum[1]. The purpose of the cooler is to increase the exchange surface between the chip and the air. The dimensions of the cooler are defined by the dimensions of the chip, the maximum temperature viable for the chip and the Thermal Design Power (TDP) i.e. the maximum sustainable power to be dissipated by the chip.[2]

Figure 3.1: The Zalman cooler ZM-NB32K

The critical case simulated here will be the starting of the computer in a warm air and with a constant and maximal dissipation of energy from the chip.

1 For more information : Zalman Tech Co., ZM-NB32K. URL : `http://www.zalman.co.kr/ENG/product/Product_Read.asp?idx=132` (visited 06/03/2010)

2 Example of specifications used for this case : Intel® X38 Express Chipset Thermal and Mechanical Design Guidelines. URL `http://www.intel.com/products/desktop/chipsets/x38/x38-technicaldocuments.htm` (visited on 26/02/2010)

3.2 Physics

In this second chapter, *laplacianFoam* will be used again as the basis of your first solver. As you learnt in the first exercise, this application solves the Fourier-Kirchhoff equation :

$$\frac{\partial T}{\partial t} = a\nabla^2 T \tag{3.1}$$

In this equation a is the thermal diffusivity $[m^2/s]$ and it is a constant defined in `transportProperties`[3]. In order to improve the solver, we want to be able to set a varying value for the thermal diffusivity.

The exercise will be based on a cooling system for micro-chips. For that purpose a new type of boundary condition will be used : heat transfer between a wall and a fluid. This condition is modelled using the following equation :

$$q_s = -k\frac{\partial T}{\partial n} = h(T_{wall} - T_{fluid}) \tag{3.2}$$

where
q_s is the heat flux $[W/m^2]$
k the thermal conductivity of the solid $[W/mK]$
$\frac{\partial T}{\partial n}$ the gradient of temperature going out of the solid (<0 if the wall is hotter than the fluid) $[K/m]$
h the heat transfer coefficient $[W/m^2K]$
S the surface of the wall $[m^2]$

Consequently this is a Neumann condition, the gradient at the boundary is known.

The heat transfer coefficient depends on the physical properties of the fluid and the solid. Typical values are[4] :

- For the air : 10 - 100 $[W/m^2K]$
- For the water : 500 - 10 000 $[W/m^2K]$

3.3 Numerics

The structure of a code in C++ will be presented shortly in this paragraph.

There are two main categories of codes : the classes and the applications. They are compiled in libraries and in executables files respectively. The main difference is that a library cannot be run by itself. But it provides functionalities that could be linked into applications or other libraries. In the opposite an executable is runnable (e.g. a solver is an executable but the classes defining the boundary conditions are

[3] see Chapter 1.

[4] source : `http://www.engineeringtoolbox.com/convective-heat-transfer-d_430.html` (visited on 05-03-2010)

stored in libraries). In this chapter, we will focus on the applications. The class will be described in the chapter 8.

For an application like `laplacianFoam` (cf. listing 2.2), the source file contains a *main* function (entry point of the executable file). And some header files are added before the *main* function to define the classes used in the solver. Or they are added in the code to improve the readability of the code.

3.4 OpenFOAM®

3.4.1 The *funkySetFields* utility

funkySetFields[5] is an improved version of the official *setFields* tool. It allows to change the internal value of a field with conditions on the location of the point, with mathematical expression,.... It can be used as a command with arguments or by reading a dictionary `system/funkySetFieldsDict`. This second option will be used.
Here is an example of that dictionary. In this example all the optional parameters have the default values.

```
expressions
(
        pressure1 //Name of the action
        {
                field p; // field to be modified
        // expression to use to evaluate the field
                expression "10.*(0.1-pos().y)";
        /* [optional] subset of cells for which the action
         is valid */
                condition "";
        /* [optional] if true, keep the current boundary
         conditions. Otherwise the boundary condition
         is set to zeroGradient. */
                keepPatches false;
        /* [optional] list of patches where the boundary
         condition has to be set to fixedValue extrapolate
         from the internal field.*/
                valuePatches ();
        // [optional] is true if the field has to be created
                create false;
        /* [optional] if the field has to be created,
        you can specify the dimension of it.*/
                dimension ();
        }
);
```

Listing 3.1: Example of `funkySetFieldsDict`

Once the dictionary is set, you can execute the tool by typing the following command in a terminal :

```
funkySetFields -time 0
```

[5] The source code and the documentation of this tool is part of the toolbox `swak4Foam`. You can obtain more information and download the sources on `http://www.openfoamwiki.net/index.php/Contrib/swak4Foam`

3.4.2 Before writing some code

The following paragraph will give you some important advices to follow when you want to write/modify some code.
First, to avoid modifying the basic code of OpenFOAM®, you should work on a copy of the files and use a different name to avoid any conflict between your version and the one from OpenFOAM®. So we will start by copying the source code of `laplacianFoam` in your `$FOAM_RUN` folder. In a terminal, write the following commands[6] :

- creation of the folder structure
  `mkdir -p $WM_PROJECT_USER_DIR/applications/solvers/myLaplacianFoam`
- move to this new folder
  `cd` `$WM_PROJECT_USER_DIR/applications/solvers/myLaplacianFoam`
- copy the folder containing the parameters for the compiler
  `cp -r $FOAM_SOLVERS/basic/laplacianFoam/Make ./`
- copy the header files
  `cp $FOAM_SOLVERS/basic/laplacianFoam/*.H ./`
- copy the source code and rename the solver to `myLaplacianFoam`[7]
  `sed` `s/laplacianFoam/myLaplacianFoam/g`
  `$FOAM_SOLVERS/basic/laplacianFoam/laplacianFoam.C > myLaplacianFoam.C`

Before continuing, we can check if all the required files are copied in `$WM_PROJECT_USER_DIR/applications/solvers/myLaplacianFoam`. For that, type in the terminal

```
ls -R .
```

The command will list all the existing files in the current folder and its sub-folders.
You should have in the main folder : `createFields.H Make myLaplacianFoam.C write.H` and in the `Make` folder : `files options`.

3.4.3 Structure of a solver folder

OpenFOAM® is tool-box that contains lots of classes stored in different libraries. Consequently, writing a solver is like making a recipe from those ingredients.

Classically in a folder directory, you will find some source code files and a folder `Make` containing two files : `files` and `options`. In the following paragraphs, those elements will be described in more details, taking `myLaplacianFoam` as example.

Source code

There are two kinds of files containing the source code : the header files and one source file named *`nameSolver`*`.C`.

[6] To have more information about a Linux command, you can type *`command`* `-help` or `man` *`command`*. For example `mkdir -help`

[7] The command `sed s/`*`oldName`*`/`*`newName`*`/g `*`file`* replace in the *file* the *oldName* pattern by the *newName* pattern.

The header files (extension `.H`) : they contain pieces of code used in the solver. They are used with the `include` command. It is equivalent to copy the content of a header file instead of the command `#include "myheader.H"`. For example at the line 43 in `myLaplacianFoam.C`, you can replace `#include "createFields.H"` by the content of `createField.H` i.e. `Info« ... transportProperties.lookup("DT") );`.
There are two main reasons to use the header files like that. Firstly, it eases the comprehension of the code. For example `createFields.H` contains the code to create the fields used in the simulation as well as the code to read the parameters needed to do so. So if you want to change the equation solved by the solver, you know that is not in that part of the code. And inversely if you want to change the way a field is created, you have to look in that file and not in *`nameSolver`*`.C`. The second advantage of those header files comes when you want to reuse a part of the code. Indeed, for example, the creation of the geometry is a part common to all solvers. So this part is written only once in a file called `createMesh.H`. And you just have to include this file at the begin of the solver (e.g. line 42 in `myLaplacianFoam.C`).[8]

The source code file (extension `.C`) defines the solver itself. It starts by a series of header files that define the classes that could be used in the solver; e.g. `fvCFD.H` line 32 in `myLaplacianFoam.C` is included in all solvers because it contains the definition of all basic OpenFOAM® classes. Then the solver starts (= function `int main(int argc, char *argv[])`).
The structure of an OpenFOAM® solver is as follow :

1. Some header files that read the parameters, create the geometry and the fields,... In fact this part groups all the actions that have to be done before solving the equations.
2. The time loop (`while (runTime.loop())`). As OpenFOAM® is a transient solver, a time loop is an unavoidable item. During each time step the three following events are carried out :
   - Reading and setting of the solver parameters (in our case `readSIMPLEControls.H`)
   - Solution of the equation. In this case, only equation (3.1) has to be solved.
   - Writing of the data (after checking if the current time step corresponds to a writing time as defined in `system/controlDict`)

Make folder

This folder is used only by the compilation tools provided by OpenFOAM® to create the executable version of the solver. It always contains only two files named `files` and `options`.
The first one contains the list of the source code files that have to be compiled, the location of the executable file and the name of this last one. The content of `files` in our case is :

```
laplacianFoam.C

EXE = $(FOAM_APPBIN)/laplacianFoam
```

[8] For more information on the header files used at the beginning of solver, have a look to `$FOAM_SRC/OpenFOAM/include/`. But the comprehension of those files are not mandatory to understand the solver.

This is wrong. So change it to compile `myLaplacianFoam.C` and create an executable file called `myLaplacianFoam`. Another important change is the location of the executable. Currently the location is `$FOAM_APPBIN`, the main OpenFOAM® folder for applications. However, you should not use this directory. A good practice is to put your executable files in your working directory. To do so, change `$(FOAM_APPBIN)` to `$(FOAM_USER_APPBIN)`.

The second file, `options`, contains the location of the header files and the libraries[9] used by the solver. Those parameters are used during the compilation to check for error and to link the executable file with the libraries.

Compilation of the solver

OpenFOAM® provides powerful tools to compile : `wclean` and `wmake`. As an example is better than a long text, you will compile the new solver. For that make sure that the current directory is the one containing `myLaplacianFoam.C` and a `Make` folder. Then type the following command :

```
wclean; wmake
```

The following lines are then printed on the terminal :

```
Making dependency list for source file myLaplacianFoam.C
SOURCE=myLaplacianFoam.C ;
g++ -m64 -Dlinux64 -DWM_DP -Wall -Wextra -Wno-unused-parameter
    -Wold-style-cast -Wnon-virtual-dtor -O3  -DNoRepository
    -ftemplate-depth-100
    -I/.../OpenFOAM/OpenFOAM-2.0.1/src/finiteVolume/lnInclude
    -I../heatTransferFvPatchScalarField/lnInclude -IlnInclude -I.
    -I/.../OpenFOAM/OpenFOAM-2.0.1/src/OpenFOAM/lnInclude
    -I/.../OpenFOAM/OpenFOAM-2.0.1/src/OSspecific/POSIX/lnInclude
    -fPIC -c $SOURCE -o Make/linux64GccDPOpt/myLaplacianFoam.o
g++ -m64 -Dlinux64 -DWM_DP -Wall -Wextra -Wno-unused-parameter
    -Wold-style-cast -Wnon-virtual-dtor -O3 -DNoRepository
    -ftemplate-depth-100
    -I/.../OpenFOAM/OpenFOAM-2.0.1/src/finiteVolume/lnInclude
    -I../heatTransferFvPatchScalarField/lnInclude -IlnInclude -I.
    -I/.../OpenFOAM/OpenFOAM-2.0.1/src/OpenFOAM/lnInclude
    -I/.../OpenFOAM/OpenFOAM-2.0.1/src/OSspecific/POSIX/lnInclude
    -fPIC -Xlinker --add-needed Make/linux64GccDPOpt/myLaplacianFoam.o
    -L/.../OpenFOAM/OpenFOAM-2.0.1/platforms/linux64GccDPOpt/lib \
    -L/.../user-2.0.1/platforms/linux64GccDPOpt/lib -lfiniteVolume
    -lheatTransferBC -lOpenFOAM -ldl   -lm
    -o /.../user-2.0.1/platforms/linux64GccDPOpt/bin/myLaplacianFoam
```

The first line informs you about the header files needed by `myLaplacianFoam.C`. This list is stored in a file called `myLaplacianFoam.dep`. Then the source code is checked for warnings and errors. Here none are detected. So the first step of the compilation is done. In a second step the executable file called `myLaplacianFoam` is generated.

[9] A library is a binary file containing some compiled code that are not directly executable. For example, the functions defining the discretization of the equation (e.g. `fvm::laplacian()`) are contained in the `finiteVolume` library.

After the compilation, additional elements are present in the folder of the solver : a file `myLaplacianFoam.dep` and another folder in `Make`, `linux64GccDPOpt`. The first one contains a list of header files used by the solver and the folder in `Make` contains the intermediate objects created during the compilation process. When later you will run again the command `wclean; wmake`, the former will remove those two elements and then the compilation will take place again.

Congratulations, you have just compiled your first OpenFOAM® solver. But that was easy as you change only the name of the solver. Let's go further and change effectively the code. This is one of the objectives of the next section.

Exercise

Modification of the solver

As said previously, we want to change the thermal diffusivity from a constant scalar to a field varying in the domain. To do so, two steps are needed. First a new field has to be created at the start of the solver instead of a scalar. Then we need to modify the equation.

Addition of a field

The creation of the fields is defined in `createFields.H`. In this case, there is only one field created, the temperature T, from line 3 to 14. An analysis of those lines will allow you to understand how to create a field.

```
volScalarField T
(
        IOobject
        (
                "T",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
        ),
        mesh
);
```

In the first line, an object is created; its type is `volScalarField` and its name is `T`. The type `volScalarField` could be translated in a natural language by "field of a scalar parameter defined on the volume". Another important type is `volVectorField` (e.g. the velocity). It defines a field of a vector parameter defined for each point of the volume. In order to create such object, two parameters are required the so-called `IOobject` and the `mesh` variable.
`IOobject` is an Input/Ouput object managing the reading and writing of files. To construct it, 3 parameters are mandatory the name of the file to be read/written (here `"T"`), the folder where to read/write the data (here `runTime.timeName()` = *`caseDirectory/timeFolder`*) and the `mesh` object.
The two following are optional. The first option can take the values `MUST_READ`, `MUST_READ_IF_MODIFIED`, `READ_IF_ PRESENT` or `NO_READ` (default value). If

the first option is chosen, when the field is created, a file containing the values of it has to be present in the current time folder. The second option as the same effect than the first one, except during the execution of the solver. Indeed if you manually modify the file during the execution, the solver will be aware of your changes in the contrary with the first option. Usually fields like temperature or velocity are created with the option `MUST_READ`. However the dictionary like `transportProperties` are created with `MUST_READ_IF_MODIFIED`.
If the third option is chosen, the file with the values is read if present. And with the default option, this file with the values is ignored. If the option is not `MUST_READ` a default value has to be given to the `volScalarField` instead of the `mesh` object. The second optional parameter could be `AUTO_WRITE` or `NO_WRITE` (default value). With the first option, the field is written according the parameters specified in the `system/controlDict`. If the second option is specified, the field is not written.

For this case, a new field for the thermal diffusivity named *a* has to be created after the temperature field. The value of that new field will be stored in a file that has to be read. And the values of the field have to be written according the control dictionary.
As the variable `DT` is no more needed, you can comment the definition of it[10].

The last change to be made in `createFields.H` is to update the output information `Reading diffusivity DT\n`. Indeed you should at least update the name of the diffusivity variable.

Change the equation

In order to change the equation, find the line in `myLaplacianFoam.C` with the definition of the equation (3.1). Copy this line. Then put the original line in comment and modify the copy to use the new field *a* instead of *DT*.

You can now compile your modified solver.

The cooler case

Now that your new solver is compiled, it can be used to solve a problem, in particular the cooler test case of this chapter.

Preliminaries

But before running the case, some initial steps have to be carried out. First copy the archive file *cooler.tar.gz* into your OpenFOAM® working directory. Then extract the files by typing the following commands in a terminal :

```
cd $WM_PROJECT_USER_DIR
tar xf cooler.tar.gz
```

In addition with the test case, that file contains a new boundary condition. It has to be compiled. To do so execute the two following commands :

[10] You can also delete those lines but you should always think twice before deleting a piece of code that works.

```
cd $WM_PROJECT_USER_DIR/src/fvPatchFields/heatTransferFvPatchField
wclean libso; wmake libso
```

The new boundary condition is compiled in a library called *libheatTransferBC.so*. The difference between the compilation of an executable (like a solver) and a library is done by passing the argument `libso` to `wclean` and `wmake`. Now, all the code needed is compiled. The next step is the setup of the case.

Setup of the case

The archive file provides you with all the parameter files that you need. But the geometry, the initial state and the boundary conditions are incomplete. The mesh is defined using the Fluent® format (file `constant/polyMesh/mesh_cooler.msh`). In order to convert it in the OpenFOAM® format, the utility *fluentMeshToFoam* will be employed. First you can look at the options of that command. Write in a terminal :

```
fluentMeshToFoam -help
```

An interesting option is the *scale* option. In fact the unit used for the mesh is the millimeter. However OpenFOAM® uses SI units. Consequently you have to used the option `-scale` *`scale_factor`*.
The other options are not interesting in this case. So to transform the Fluent® mesh into an OpenFOAM® mesh, you will have to run the utility `fluentMeshToFoam`. The two elements that have to be specified are the mesh file and the scale factor.

The surfaces of the cooler are divided into three groups : PROCESSOR, EXCHANGE _SURFACE and COOLER. The first one is the large flat face in contact with the chip, the second groups all the surfaces that dissipate energy to the surrounding air and the last one is the surfaces negligible for the heat transfer with the air.
The list of boundary conditions can be checked by opening `constant/polyMesh/boundary`.

The last thing to be done before running the case is the definition of the initial and boundary conditions. If you look into the folder `0` (zero), there is only one file `a` but no file `T` to define the conditions on the temperature field. To do so, copy the file defining the thermal diffusivity and rename it as `T`. Then open it and change the following elements :

1. In the header, change the name of the field
2. Change the dimensions of the field from $[m/s^2]$ to $[K]$ (see pp.105-107 of the User Guide for more information[11].)
3. Set the internal field to the temperature of the air (here 35°C).
4. Set the boundary conditions :
   - PROCESSOR : the chip dissipates 26.5 W on a surface of 25 mm x 37 mm. Set that boundary condition using a `fixedGradient` condition. That condition requires the parameter `gradient`. For **example** :

```
wall1
{
        type            fixedGradient;
        gradient        uniform 20;
}
```

[11] And more specifically the paragraphs 4.2.6 and 4.2.8

- EXCHANGE_SURFACE : the cooler transfers the heat to the air following equation (3.2). The heat transfer coefficient is supposed to be constant and equal to 50 $W/m^2/K$. The air is at a temperature of 35°C. And the solid material is aluminium (its thermal conductivity is 250 $W/m/K$). An **example** of the definition of such boundary condition follows.

```
exchange
{
        type      heatTransfer;
        h         40;  // heat transfer coefficient
        k         120; // solid thermal conductivity
        Tfluid    293; // temperature of the air
        value     uniform 293; // temperature at t=0
}
```

- COOLER : the heat flux across that surface is zero.

Before running the solver, open the `a` file to have a look at it. As you can see, the internal field and the boundary conditions set the thermal diffusivity to the one of the aluminum : 8.418e-5 m^2/s.

Running and post-processing

The case is now ready to be solved. To run the new solver on the case, write the following command in a terminal where the current directory is the case directory (here `$FOAM_RUN/myLaplacianFoam/cooler`) :

```
myLaplacianFoam > log &
```

To have a look of the log file, type the following command :

```
tail -f log
```

And press *Ctrl+C* to stop the command.

Once the case is solved (the end time is 240 s), the data can be post-processed using Paraview®[12]. For that, type `paraFoam &` in the terminal. You can for example visualize the increase of temperature until it reaches the steady state.
If the design of the passive cooler is good, the heat flux at the end of the fins have to be close to zero. In the results, the heat flux is not computed directly. But the temperature gradient is available. And as shown by equation (3.2), it is linked to the heat flux.
The normal to the end face of the fin is z, the gradient of temperature in the z direction, *gradTz*, is consequently the one interesting in this case.
Have a look to the field *gradTz*.

- Is the evolution of the gradient along a fin physically correct ?
- What about the evolution in absolute value with z ?
- What about the sign ?

[12] For more information on how to use Paraview, have a look to the first tutorial in the User Guide of OpenFOAM®.

You can check a value of a field in a point in ParaView® thank to the filter *Probe Location*.
To look at the value at the points (0.0185, 0, 0.032) and (0.0185, 0.0175, 0.032) select the filter *Probe Location*. Then click on the *Split Vertical* button on the upper right of the frame view[13]. And choose *Spreadsheet View*.

- Is the difference between the gradient at those two points large ?
- Is the design of the cooler good (compare the value of the gradient at those two points with the gradient at the boundary condition `PROCESSOR`) ?

Another test is the comparison between the current results and the results given by the original solver `laplacianFoam`. Indeed as, in this case, the thermal diffusivity is constant, the results should match those obtained by `laplacianFoam`.
The results at the latest time step will be the only one to be compared.
Execute the following actions to do so :

1. Create a new directory `$FOAM_RUN/laplacianFoam/cooler2`
2. Copy in that directory the folders `0` (zero), `constant` and `system` from the cooler case directory
3. Modify, in the `system/controlDict`, the *application* parameter to *laplacianFoam* and *writeInterval* to 40
4. Run *laplacianFoam* on that new case[14]

Once the case is solved, launch `paraFoam`. And to be able to compare easily this case with the previous one, you can create two visualization windows. For that,

- Click on the icon *Split Horizontal*. It is the first button on the upper right corner of the frame view.
- Select *3D View*
- With a terminal, go to the directory of the previous case
- Type in the terminal :

```
touch cooler.OpenFOAM
```

- In paraView®, click on *Open file*[15]. And open the file `cooler.OpenFOAM` in the directory of the previous case.
- To change something in one of the frame, be sure that it is selected (The border of the frame is then blue) and then used the *Pipeline browser* and the *Object Inspector* panels as usual.

Compare the temperature field at the latest time step in two cutting plans, called *Slice* in ParaView® : the centre YZ plan and the centre XZ plan.

[13] Be sure that the filter *Probe Location* is selected before splitting the window.

[14] Reminder : the command is `laplacianFoam > log &`

[15] Before opening the file, you should check if the new window is the current selected window.s

The defective cooler case

The next step is to use the new feature introduced in *myLaplacianFoam* : the non-constant thermal diffusivity. To change the value of the field inside the domain, the tool *funkySetFields* will be used.

Description of the case

Unfortunately, during the manufacturing of a cooler, a piece of sand was captured in the aluminum. The thermal diffusivity of the sand is 1.12e-6 m^2/s and the one of the aluminum is 8.418e-5 m^2/s. That piece has roughly the form of an ellipsoid with its center at (0.015, 0.0045, 0.003) and its radii (0.003, 0.0007, 0.001).

Setup of the case

First create a new folder to store this case. Copy in that folder the folders `0` (zero), `constant` and `system` of the first case (directory `$FOAM_RUN/myLaplacianFoam/cooler`). Open the `funkySetFieldsDict` file. To achieve the goals described above, the following two actions will be set in `funkySetFieldsDict` :

1. First action, set the thermal diffusivity field to 8.418e-5 m^2/s. Set all the boundary conditions as `fixedValue` computed from the internal field.
2. Secondly, set the interior of the ellipsoid with the sand thermal diffusivity value. Keep the boundary conditions as defined previously.

Remark :

- The formula for the interior of an ellipsoid is :

$$\frac{(x-x_c)^2}{a^2}+\frac{(y-y_c)^2}{b^2}+\frac{(z-z_c)^2}{c^2}<1 \tag{3.3}$$

  where (x_c, y_c, z_c) are the centre coordinates and (a, b, c) the radii of the ellipsoid.
- To use a coordinate in *expression* or *condition*, use `pos().x` for x, `pos().y` for y or `pos().z` for z
- To take the square of a value *X*, use `sqr(X)`

Run *funkySetFields* on the time folder `0`.

Open now the *control dictionary* and change `writeInterval` to 20.

Running and post-processing

The case can be solved. For that run `myLaplacianFoam` on the current case[16]. Use ParaView® to compare the defective cooler case and the cooler case. And find the answer to the following questions :

[16] Reminder : the command is `myLaplacianFoam > log &`

- What is the maximum temperature of the chip in the cooler case after 240 s ? And in the defective cooler case ?
- Is the gradient at the end of the fin directly above the piece of sand different than in the cooler case ?
- How is the heat flux modified by the piece of sand ?

Hint : The vector of the temperature gradient allows you to visualize the heat flux lines. So in this case, you can for example slice the cooler and the hole. Then on the *Slice* filter in ParaView®, you can apply on the temperature field the filter *Gradient Of Unstructured Data Set*[17]. And finally apply on that last filter a *Glyph* filter.

[17] Location within the menu : *Filters->Alphabetical->Gradient Of Unstructured Data Set*

3.5 Extra Practice and Background Information

This section will introduce an additional tool for post-processing data in OpenFOAM®. And in the second part you will create your first boundary condition.

Reach the steady state

Until now, the data were analysed by presuming that the steady state were reached. In this paragraph, the tool *patchIntegrate* will help you to know if the system is in a steady state or not. The steady state of the system corresponds at the dynamical equilibrium between the power dissipated by the processor and the one exchanged with the surrounding air.

The utility *patchIntegrate*

This tool integrates given fields on specified patches. In this case, the heat flux is not directly accessible. However as written in (3.2), the heat flux is proportional to the gradient of temperature. Consequently by knowing the integral of this gradient, the output power and the input power could be compared.

The syntax for the `patchIntegrate` tool is :

```
patchIntegrate scalar_field patch [-latestTime]
[-time writing_time]
```

The time options are quite interesting in this case. Indeed, if no options are given, the integral is computed at each writing time. However only the latest time step is of interest in this case (or maybe the few last ones).

By default the command prints the data on the terminal in which it runs. In order to store them, you can redirect the flow to a file (like when you use the *log* file with a OpenFOAM® solver)[18].

A typical output of the function follows :

```
Time = 240
        Area vector of patch PROCESSOR[1] = (0 0 -0.000925635)
        Area magnitude of patch PROCESSOR[1] = 0.000925635
        Reading volScalarField gradTz
        Integral of gradTz over vector area of patch
                PROCESSOR[1] = (0 0 -0.0642255)
        Integral of gradTz over area magnitude of patch
               PROCESSOR[1] = 0.0642255
```

In order to correctly interpret the output above, you have to remember that in OpenFOAM® each face of a cell is characterized with a vector $\vec{S_f}$ parallel to the

[18] An example could be : `patchIntegrate p inlet -latestTime > integrationPInlet`

normal, pointing to the exterior of the cell and with a magnitude equal to the surface of the cell. So the four results correspond to the following equations :

$$\sum_{faces} \vec{S_f} \tag{3.4}$$

$$\sum_{faces} |\vec{S_f}| \tag{3.5}$$

$$\sum_{faces} f\vec{S_f} \tag{3.6}$$

$$\sum_{faces} f|\vec{S_f}| \tag{3.7}$$

where f is the scalar field put in argument of *patchIntegrate* (in the example is `gradTz`).

You should now be able to verify the following statement for the cooler case:

$$\iint_{PROCESSOR} \vec{\nabla T} \cdot \mathrm{d}\vec{S_f} \stackrel{?}{=} \iint_{EXCHANGE\_SURFACE} \vec{\nabla T} \cdot \mathrm{d}\vec{S_f} \tag{3.8}$$

My first boundary condition

The goals of this second optional exercise are to create a new boundary condition (BC) and to test it on the test case of the cooler.

Your first BC will be based on the *heatTransfer* BC used previously. To do so, you can duplicate[19] the header file and the source file `heatTransferFvPatchScalarField` in `$FOAM_RUN/src/fvPatchFields/heatTransferFvPatchScalarField`.
ATTENTION : do not forget to rename the files and change the class name inside the files according the new name you want to give to the new boundary condition[20]. You also have to choose the name of the boundary condition that will be used in the boundary condition file of the temperature field (e.g. : *heatTransfer* in the previous exercise). The name of the boundary condition is defined by the string *TypeName* at the line 92 of the header file.
Then you can add the new source file to the `Make/files` list.
Before going further try to compile the two boundary conditions to be sure that the renaming process carried out successfully. To compile a library the following command has to be used:

```
wclean libso; wmake libso
```

The boundary condition can now be changed. Until now, the condition is an exchange with the air using a constant heat transfer coefficient. However it is clear that in the reality, the heat transfer coefficient is higher on the external face of the

[19] Don't create a new folder containing your new boundary because you will have trouble to use it with a case. Or read the paragraph *3.2.6 Linking new user-defined libraries to existing applications* in the *User Guide*

[20] For that you can use advantageously the command `sed s/oldName/newName/g oldFile > newFile`

cooler and at the end of fins and lower at the center of the cooler. A definition of the heat transfer coefficient using an empirical equation could be implemented. But to stay focused on the computational aspect, the coefficient will be deduced only from the geometry in order to respect the following requirements :

- The coefficient is proportional to the cube of the distance, r, between the face and the centre of the cooler (18,5e-3; 0; 5e-3).
- The minimum value of the coefficient is 10 $W/m^2/K$ at r = 0 mm
- The maximum value is 60 $W/m^2/K$ at the maximal ray (R = 32,7 mm)
- The equation is $h = A((\frac{r}{R})^3 - 1) + B$

The gradient of temperature is defined in the function `updateCoeffs`. Try to modify the equation to use a variable heat transfer coefficient instead of the constant `h_`. To implement the new boundary condition, you need the coordinates of the face centre. They can be obtained as a `vectorField` by adding the following line in the function:

```
const vectorField& Cface = patch().Cf();
```

Then you can access the coordinates through the variable *Cface* by calling the function `component:Cface.component(vector::X),Cface.component(vector::Y)` and `Cface.component(vector::Z)`.
You will need also the function *pow*. For example to take the cube of *X*, write `pow(X, 3.0)`. After compiling the boundary condition, you can test it on the cooler test case. For that, only the name of the temperature boundary condition has to be changed. Once it's done, solve the case using *myLaplacianFoam* and answer the following questions.

- Is the steady state reached after 240 s (Hint : the utility *patchIntegrate* can help you) ? What's the difference between the heat flux provided by the processor and the one dissipated at 240 s ?
- Is the maximum temperature at the interface between the processor and the cooler still acceptable (Tc_{max} = 92°C)?
- Is the temperature field closer to the reality with the new boundary condition ? Compare, when the steady state is reached, the temperature profile along a fin in the centre of the cooler and in the extremity for this case and for the first case.

Chapter

4 Channel Pipe Flow

4.1 Physics

4.1.1 Laminar channel pipe flow

This tutorial case will describe a laminar, incompressible and isothermal flow in a pipe with circular cross section. Once the flow is fully developed $\partial u/\partial z = 0$ we expect a parabolic velocity profile $u(r)$. The velocity profile does not change along z which means $(1/\rho) \cdot (\partial p/\partial z) = const.$. We neglect any influence of gravity forces.

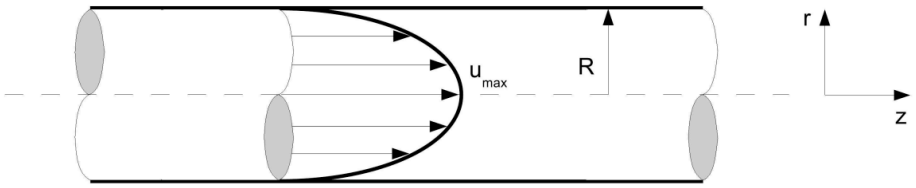


Figure 4.1: Poiseuille channel pipe flow

One can derive from the incompressible Navier-Stokes equations in cylindrical coordinates:

$$\frac{1}{\rho}\frac{\partial p}{\partial z} = \nu \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \cdot \frac{\partial u}{\partial r} \right) = const. \tag{4.1}$$

Thus we have a 2nd order differential equation with the boundary condition:

$$u(R) = 0 \tag{4.2}$$

and the side condition:

$$\left. \frac{\delta u}{\delta r} \right|_{r=0} = 0$$

Equation 4.1 can be solved for $u(r)$

$$u(r) = -\frac{R^2}{4\,\nu\,\rho}\,\frac{dp}{dz}\left(1 - \frac{r^2}{R^2}\right) \tag{4.3}$$

With the maximum velocity $u_{\text{max}} = -(R^2/(4\,\nu\,\rho)) \cdot (dp/dz)$ we get

$$u(r) = u_{\text{max}}\left(1 - \frac{r^2}{R^2}\right) \tag{4.4}$$

4.2 Numerics

In this section, the conservation equations will be expressed for the two-dimensional mesh presented in the figure 4.2. The extension to the 3D mesh is straightforward. The capital letters refer to the centroid of the cells and the small letters to the faces.

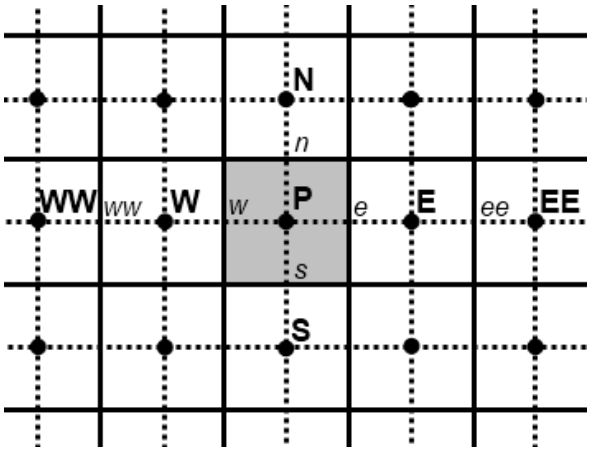


Figure 4.2: Pressure-velocity coupling: a 2D mesh example

The velocity U is decomposed in (u, v) in the x and y direction respectively.

4.2.1 Conservation equations

For incompressible and isothermal steady flows, the 2D governing equations are:

- continuity equation

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{4.5}$$

- momentum equations

$$\frac{\partial uu}{\partial x} + \frac{\partial uv}{\partial y} = \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{1}{\rho} \frac{\partial p}{\partial x} \tag{4.6}$$

$$\frac{\partial vu}{\partial x} + \frac{\partial vv}{\partial y} = \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{1}{\rho} \frac{\partial p}{\partial y} \tag{4.7}$$

The continuity equation is a single equation with a vector for unknowns: the velocity. But the momentum equations are a vector equation with a vector - the velocity - and a scalar - the pressure - for unknowns. As a result of this pressure-velocity coupling those two equations cannot solve in a segregated way. To solve it they have to be solve together or a special algorithm should be used. That second solution is used in the code.

The discretization for the cell centered on P (cf. figure 4.2) of the x-momentum equation using the finite volume framework is (with A being the face area)

$$\begin{aligned}&(uuA)_e - (uuA)_w + (uvA)_n - (uvA)_s = \\ &\left(\nu A\frac{\partial u}{\partial x}\right)_e - \left(\nu A\frac{\partial u}{\partial x}\right)_w + \left(\nu A\frac{\partial u}{\partial y}\right)_n - \left(\nu A\frac{\partial u}{\partial y}\right)_s \\ &-\frac{1}{\rho}((pA)_e - (pA)_w)\end{aligned} \tag{4.8}$$

The equation can still not be solved numerically. We need to interpolate the different variables at the cell faces as well as the gradient of u. This is the role of the numerical schemes. The latter will interpolate the fields on the cell faces using the known values in the neighboring cells of P (i.e. W, E, N, S,... grouped further under the abbreviation nb). The resulting equation is

$$a_P u_P = \sum_{nb} a_{nb} u_{nb} - \frac{A}{\rho}(p_e - p_w) \tag{4.9}$$

The values of the a_i are function of the geometry, the numerical schemes and the physical properties.

The Poisson's equation

Taking the divergence of the momentum equations and using the continuity equation ($\nabla U = 0$) gives

$$\frac{\partial^2(u_i u_j)}{\partial x_i \partial x_j} = -\frac{1}{\rho}\nabla^2 p \tag{4.10}$$

or

$$\nabla^2 p = -\rho\frac{\partial^2(u_i u_j)}{\partial x_i \partial x_j} \tag{4.11}$$

Hence, analytically, the combination of momentum and continuity gives a Poisson equation for pressure.

4.2.2 Collocated storage of variables

First of all, suppose that pressure and velocity are collocated (stored at the same positions) and that mass fluxes are evaluated by linear interpolation of the velocities to cell faces.

Momentum Equation

The net pressure force (in the x direction) depends on the difference of values p_w and p_e. These must be obtained by interpolation resulting in the elimination of p_P:

$$\begin{aligned}p_w - p_e &= \frac{1}{2}(p_W + p_P) - \frac{1}{2}(p_P + p_E) \\ &= \frac{1}{2}(p_W - p_E)\end{aligned} \tag{4.12}$$

Hence, the discretized momentum equation has the form:

$$u_P = \frac{1}{2} d_P (p_W - p_E) + ... \tag{4.13}$$

u_P depends on the difference in pressure at nodes separated by $2\Delta x$. It does not depend on p_P, which could, in fact, take any value without affecting the momentum balance of that cell.

Continuity Equation

If one applies continuity to a control volume centered on a pressure node, advective velocities u_e and u_w can be obtained on the cell faces by linear interpolation:

$$u_w = \frac{1}{2}(u_W + u_P), \qquad u_e = \frac{1}{2}(u_P + u_E) \tag{4.14}$$

Mass conservation implies

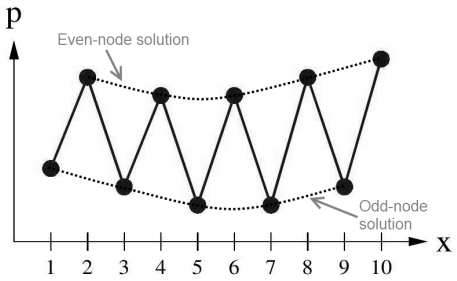
$$\begin{aligned} 0 = (Au)_e - (Au)_w + ... &= A\left\{\frac{1}{2}(u_P + u_E) - \frac{1}{2}(u_W + u_P)\right\} + ... \\ &= A\frac{1}{2}(u_E - u_W) + ... \end{aligned} \tag{4.15}$$

A pressure equation is generated by linking velocities to pressure differencing using 4.13,

$$\begin{aligned} 0 &= (Au)_e - (Au)_w + ... \\ &= A\frac{1}{2}(u_E - u_W) + ... \\ &= A\frac{1}{2}\left(\frac{1}{2} d_E (p_P - p_{EE}) - \frac{1}{2} d_W (p_{WW} - p_P)\right) + ... \end{aligned} \tag{4.16}$$

Thus, the continuity equation yields an equation for pressure. However, it only links pressure values at every second node.

The combination of collocated u, p and linear interpolation for advective velocities leads to a decoupling of odd nodal values $p_1, p_3, p_5, ...$ from even nodal values $p_2, p_4, p_6, ...$. This odd-even decoupling leads to indeterminate pressure oscillations in the pressure field.



There are two common remedies:

- Use the staggered grid where velocity and pressure are stored in different locations.
- Use a collocated grid but Rhie-Chow interpolation for the advective velocities.

Both provide a link between adjacent pressure nodes to prevent odd-even decoupling.

4.2.3 Staggered grid

In the staggered grid arrangement, velocity components are stored half-way between the pressure nodes that drive them. This leads to different sets of control volumes.

Other scalars are stored at the same position as pressure.

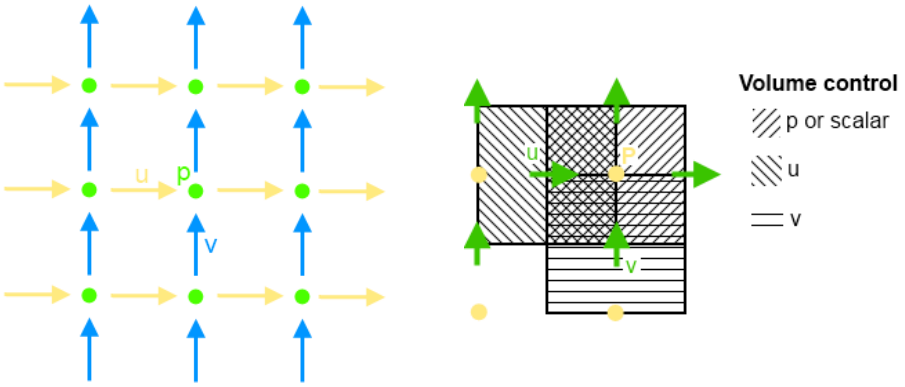


Figure 4.3: Staggered grid

On a Cartesian mesh: In the momentum equation, pressure is stored at precisely the points required to compute the pressure force. For example:

$$u_w = d_w \frac{1}{2}(p_W - p_P) + ... \tag{4.17}$$

In the continuity equation, pressure is stored at precisely the points required to compute the mass fluxes.

The net mass flux involves:

$$\begin{aligned} u_w - u_e + ... &= d_e \frac{1}{2}(p_P - p_E) - d_w \frac{1}{2}(p_W - p_P) + ... \\ &= -\frac{1}{2} d_w p_W + \frac{1}{2}\left(d_w + d_w\right) p_P - \frac{1}{2} d_e p_E \end{aligned} \tag{4.18}$$

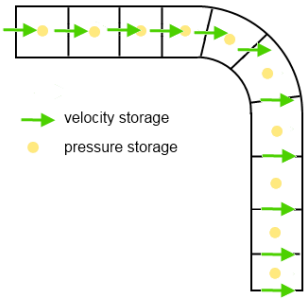
The contribution from p_P doesn't cancel out, so there is no odd-even decoupling. In both cases no interpolation is required for cell-face values and there is a strong linkage between successive, rather than alternate pressure nodes.

Advantages:

- No interpolation needed; variables are stored where they are needed.
- No problem of odd-even pressure decoupling

Disadvantages:

- Added geometry complexity from multiple sets of nodes and control volumes.
- If the mesh is not cartesian then the velocity nodes may cease to lie between the pressure nodes that drive them.
- A countermeasure is the non-orthogonal corrector in OpenFOAM® which takes into account the distortion of the mesh. It is available for both PISO and SIMPLE algorithms.



4.2.4 Rhie-Chow Velocity Interpolation

The alternative approach is to use collocated pressure and velocity nodes but employ a higher order interpolation for advective velocities (the cell-face velocities used to calculate mass fluxes).

The Rhie-Chow's interpolation practice is to obtain the cell face velocity u_e as

$$u_e = \bar{u} + \bar{d'}\nabla p - \overline{d'\nabla p} \tag{4.19}$$

in which the overline symbolizes the linear interpolation from the adjacent nodes. Consequently

$$\begin{aligned} u_e &= \frac{u_P + u_E}{2} + \frac{d_P + d_E}{2}(p_P - p_E) - \frac{d'_P \nabla p_P + d'_E \nabla p_E}{2} \\ &= \frac{u_P + u_E}{2} + \frac{d_P + d_E}{2}(p_P - p_E) - \frac{1}{4}\left(d_P(p_W - p_E) + d_E(p_P - p_{EE})\right) \end{aligned} \tag{4.20}$$

The face velocity is directly link to the driven pressure gradient due to the second term. The last term avoid the appearance of the troublesome checker-board pressure.

Assuming that the coefficient d are all identical, the face velocity u_e can be expressed as

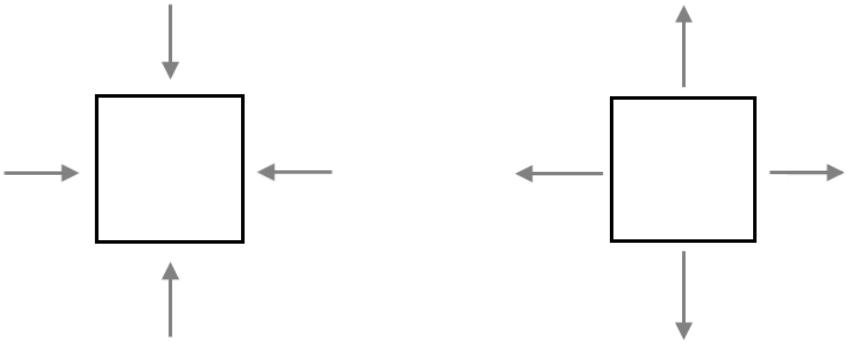
$$\begin{aligned} u_e &= \frac{u_P + u_E}{2} + \frac{d}{4}[4(p_P - p_E) - (p_W - p_E) - (p_P - p_{EE})] \\ &= \frac{u_P + u_E}{2} + \frac{d}{4}[p_{EE} - 3p_E + 3p_P - p_W] \\ &= \frac{u_P + u_E}{2} + \frac{d}{4}\left.\frac{\partial^3 p}{\partial x^3}\right|_e \Delta x^3 \end{aligned} \tag{4.21}$$

Using the central differencing, we can prove that the correction introduced is a third-order pressure gradient term. As the other parts of the method are at best second-order accurate, the correction doesn't reduced the global accuracy.

Pressure-velocity coupling is the dominant feature of the Navier-Stokes equations. Staggered grids are an effective way of handling it on Cartesian meshes. However, for non-Cartesian (or curvilinear) meshes, collocated grids are the norm and are employed in almost all general-purpose CFD codes.

4.2.5 Pressure-Correction Methods

Consider how changing pressure can be used to enforce mass conservation.



Net mass flux in;
→ increase cell pressure to drive mass out

Net mass flux out;
→ decrease cell pressure to suck mass in

Pressure-correction methods

- Iterative numerical schemes for pressure-linked equations
- Used to derive velocity and pressure fields which satisfy both mass and momentum equations
- Consist of alternating updates of velocity and pressure:
  - solve the momentum equation for velocity with the current pressure
  - observing the relationship between velocity and pressure changes imposed by the momentum equation, rephrase the continuity equation as a pressure correction equation and solve for the pressure correction p necessary to ”nudge “the velocity field towards mass conservation
- There are two common schemes: SIMPLE and PISO

The relationship between velocity and pressure corrections

The momentum equation connects velocity and pressure:

$$u = d\left(p_{-1/2} - p_{+1/2}\right) + ...$$

One must correct velocity to satisfy continuity:

$$u = u^* + u'$$

but simultaneously correct pressure so as to retain a solution of the momentum equation:

$$u' = d\left(p'_{-1/2} - p'_{+1/2}\right) + ...$$

The velocity-correction formula is, therefore,

$$u = u^* + d\left(p'_{-1/2} - p'_{+1/2}\right) + ...$$

4.2.6 SIMPLE - pressure velocity correction method

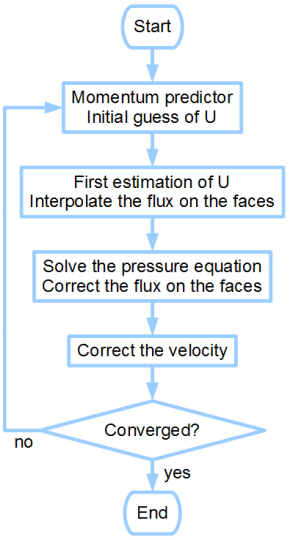
SIMPLE - Semi-Implicit Method for Pressure-Linked Equations (Patankar and Spalding, 1972)

For simplicity, only the steady-state situation will be considered since this tutorial's problem is not time dependent.

Stage 1. Solve momentum equations with current pressure.

$$a_P u_P - \sum_{nb} a_{nb} u_{nb} = A \underbrace{(p_w^* - p_e^*)}_{\text{pressure forces}} + \underbrace{b_P}_{\text{other sources}} \tag{4.22}$$

Resulting velocity generally won't be mass-consistent.



Stage 2. Formulate the pressure correction equation

1. Relate changes in u to changes in p

$$u'_P = \frac{\sum_{nb} a_{nb} u'_{nb}}{a_P} + d_P \left(p'_w - p'_e\right), \qquad d_P = \frac{A}{a_P}$$

2. Make the SIMPLE approximation: neglect $\sum_{nb} a_{nb} u'_{nb}$

$$u'_P \approx d_P \left(p'_w - p'_e\right), \qquad d_P = \frac{A}{a_P}$$

   This is legitimate since we are interested in the converged solution, where all corrections will be zero anyway.

3. Apply mass conservation to a control volume centered on the pressure node. The net mass flux results from current (u^*) plus correction (u') velocity fields:

$$\sum_{faces} \rho u_f^* A + \sum_{faces} \rho u'_f A = 0$$

   i.e.

$$(\rho u' A)_e - (\rho u' A)_w + \ldots = -\dot{m}^*$$

   (minus the current net mass flux) or, writing in terms of the pressure correction (staggered or non-staggered mesh):

$$(\rho u A)_e (p'_P - p'_E) - (\rho u A)_w (p'_W - p'_P) + \ldots = -\dot{m}^*$$

This results in a pressure-correction equation of the form:

$$a_P p'_P - \sum_{nb} a_{nb} p'_{nb} = -\dot{m}^* \tag{4.23}$$

Where $a_i = (\rho u A)_i$, etc. , and $a_P = \sum_{nb} a_{nb}$

Stage 3. Solve the pressure-correction equation
The important thing to notice is the discretized pressure-correction equation (4.23) is of precisely the same form as the discretized scalar equations, and hence the same algebraic solvers may be used.

Stage 4. Correct pressure and velocity

$$\begin{aligned} p_P &\rightarrow p^*_P + p'_P \\ u_P &\rightarrow u^*_P + d_P(p'_w - p'_e) \end{aligned} \tag{4.24}$$

(and similarly for other velocity components)

Hence the velocity field now satisfies momentum and continuity equations.

SIMPLE algorithm in OpenFOAM®

The SIMPLE loop is implemented in the solver code, e.g. `simpleFoam` used in this tutorial . The file *simpleFoam.c* can be found in:
`$FOAM_SOLVERS/applications/solvers/incompressible/simpleFoam`
The loop consists of the code fragment displayed in listing 4.1:

```
47    simpleControl simple(mesh);
48
49    Info<< "\nStarting␣time␣loop\n" << endl;
50
51    while (simple.loop())
52    {
53        Info<< "Time␣=␣" << runTime.timeName() << nl << endl;
54
55        p.storePrevIter();
56
57        // --- Pressure-velocity SIMPLE corrector
58        {
59            #include "UEqn.H"
60            #include "pEqn.H"
61        }
62
63        turbulence->correct();
64
65        runTime.write();
66
67        Info<< "ExecutionTime␣=␣" << runTime.elapsedCpuTime() << "␣s"
68            << "␣␣ClockTime␣=␣" << runTime.elapsedClockTime() << "␣s"
69            << nl << endl;
70    }
71
72    Info<< "End\n" << endl;
```

Listing 4.1: SIMPLE loop

The SIMPLE loop is implemented as a while-loop that ends when the determined run time is reached or when the desired convergence is reached.
Before the time loop the parameters to control the loop are read out when creating the object `simpleControl simple(mesh);`. These are for example the number of non-orthogonal correctors `nNonOrthogonalCorrectors` and the convergence parameters defined in `fvSolution`. They will be used to evaluate the residuals and determine the end of the run.

Before presenting the implementation, some notations used in the code will be introduced.

The governing equations are the momentum and continuity equations:

$$\begin{aligned} \nabla \cdot (U \otimes U) &= \nabla \cdot (\nu \nabla U) - \nabla \left(\frac{p}{\rho} \right) + S_U \\ \nabla \cdot U &= 0 \end{aligned}$$

p/ρ is the kinematic pressure and S_U is the source terms for the momentum equation. It is renamed simply p in OpenFOAM® for incompressible flow. You have seen that the discretized momentum equation results in

$$a_P U_P + \sum_{nb} a_{nb} U_{nb} = S_U - \nabla p$$

where p is the kinematic pressure.

By introducing the operator $H(U)$:

$$H(U) = S_U - \sum_{nb} a_{nb} U_{nb} \tag{4.25}$$

The momentum equation becomes:

$$a_P U_P = H(U) - \nabla p \tag{4.26}$$

$$U_P = (a_P)^{-1} \left(H(U) - \nabla p \right) \tag{4.27}$$

The pressure equation appears when substituting the expression of U_P (4.27) in the continuity equation $\nabla \cdot U = 0$

$$\nabla \cdot \left((a_P)^{-1} \nabla p \right) = \nabla \cdot \left((a_P)^{-1} H(U) \right) \tag{4.28}$$

So the pressure solution of that equation guarantees a divergence-free velocity field.

The convection term of the conservation equations requires the face mass flux called `phi` in OpenFOAM®. By looking at the discretized form of the continuity equation

$$\nabla \cdot U = \sum_{faces} S_{face} \cdot U = \sum_{faces} \texttt{phi}$$

Consequently using (4.27)

$$\begin{aligned} \texttt{phi} &= S_{face} \cdot U \\ &= S_{face} \cdot \left((a_P)^{-1} H(U) \right) - S_{face} \cdot \left((a_P)^{-1} \nabla p \right) \end{aligned} \tag{4.29}$$

Therefore the conservative face flux should be created from the pressure equation.

The pressure-velocity coupling is solved in the following five steps:

- **Stage 1**
  Guess the pressure field p^*
- **Stage 2**
  Now in `UEqn.H` the momentum equation is solved with the guess pressure p^* using the equation (4.27). The relaxation factor is applied and the initial and maximum residuals are calculated. This step is called *momentum predictor*.
  In `pEqn.H` (see Listing 4.2) the face flux is guessed by

  $$\texttt{phi} = \left(H(U)(a_P)^{-1}\right) \cdot S_f$$

- **Stage 3**
  The *pressure correction* step is carried out by solving (4.28).
- **Stage 4**
  Correct the conservative face flux: $\texttt{phi} -= (a_P)^{-1} S_f \cdot \nabla p$
- **Stage 5**
  Repeat to convergence

The corrected velocity is finally obtained by solving the momentum equation (4.27) with the new pressure.

Before the next loop starts the run and clock time are plotted and the convergence is checked.

```
{
    p.boundaryField().updateCoeffs();

    volScalarField rAU(1.0/UEqn().A());
    U = rAU*UEqn().H();
    UEqn.clear();
    phi = fvc::interpolate(U, "interpolate(HbyA)") & mesh.Sf();
    adjustPhi(phi, U, p);
    // Non-orthogonal pressure corrector loop
    for (int nonOrth=0; nonOrth<=simple.nNonOrthCorr(); nonOrth++)
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phi)
        );
        pEqn.setReference(pRefCell, pRefValue);
        pEqn.solve();
        if (nonOrth == simple.nNonOrthCorr())
        {
            phi -= pEqn.flux();
        }
    }
    #include "continuityErrs.H"
    // Explicitly relax pressure for momentum corrector
    p.relax();
    // Momentum corrector
    U -= rAU*fvc::grad(p);
    U.correctBoundaryConditions();
}
```

Listing 4.2: pEqn.H

Notes:

- **Staggered and unstaggered grids**. The distinction between staggered and unstaggered grids is subtle. In both cases, the expression for u'_e etc. at cell faces depends on the pressure corrections at adjacent nodes. However, for a (cartesian) staggered grid the relevant normal velocities are actually stored on the faces of the pressure control volumes where they are required to establish mass conservation.

- **Source term for the pressure-correction equation**. That the "source" for the pressure-correction equation should be minus the current net mass flux ($-\dot{m}^*$) is reasonable. If there is a net mass flow into a control volume then the pressure in that control volume must rise in order to "push" mass back out of the cell.

- **Under-relaxation**. In practice, substantial under-relaxation of the pressure update is needed to prevent divergence. On the correction step the pressure (but not the velocity) update is relaxed:

  $$p \rightarrow p^* + \alpha_P p' \tag{4.30}$$

  Typical values of α_P are in the range 0.1 - 0.3. Velocity is under-relaxed in the momentum equations, but the under-relaxation is generally less severe: $\alpha_U \approx 0.6 - 0.8$. A classical guideline states that $\alpha_P + \alpha_U \approx 1$.
- **Iterative process**. Since the equations are non-linear and coupled the matrix equations may change at each iteration. There is little to be gained by solving the matrix equations exactly at each stage, but only doing enough iterations of the matrix solver to reduce the residuals by a sufficient amount.
  Alternative strategies at each SIMPLE iteration are:

  m iterations of each u, v, w equation, followed by n iterations of the p' equation (typically $m = 1, n = 4$);

  or

  do enough iterations of each equation to reduce the residual error to a small fraction of the original (typically 10 %).

The SIMPLE scheme can be inefficient and requires considerable pressure under-relaxation. This is because the corrected fields are good for updating velocity (since the mass-consistent flow field is produced) but not pressure (because of the inaccuracy of the approximation connecting velocity and pressure corrections).
To remedy this, a number of variants of SIMPLE have been produced such as SIMPLER (SIMPLE Revised - Patankar 1980), SIMPLEC (Van Doormaal and Raithby, 1984) or SIMPLEX (Raithby and Schneider, 1988). For more information about this topic the reader is referred to the literature.

4.3 OpenFOAM®

4.3.1 simpleFoam

OpenFOAM® provides the solver `simpleFoam` to analyse incompressible and isothermal flows. It is formulated in steady-state mode which means it is iterative, so the solution does not change over time such as marching or propagation problems. SimpleFoam solves both laminar and turbulent problems.

4.3.2 fvSchemes

The *system* folder contains two important files *fvSolution* and *fvSchemes*. The latter defines the finite volume discretization schemes, it sets the numerical schemes for terms, such as derivatives in equations, that appear in applications being run. The terms range from derivatives e.g. the time marching scheme for first and second time derivatives $d/dt, d^2/dt^2$, gradient schemes ∇ interpolation schemes from point to point, convection divergence schemes $\nabla\bullet$ or the laplacian schemes ∇^2.
The problem we are analysing is steady-state so `steadyState` should be selected for the time derivatives in `timeScheme`. This essentially switches off the time derivative terms. Not all solvers in OpenFOAM® work both in transient and steady-state mode. In the finite volume method the discretization is based on the Gauss's theorem which needs to be selected for the `gradSchemes`.
For example the convection term in the momentum equation for incompressible problems $(\nabla \cdot (UU))$, denoted by the `div(phi,U)` keyword, uses `Gauss linear` which stands for the Central Difference Scheme. Since it is second order accurate both on structured and unstructured meshes it is the first choice. It should be noted that it tends to produce unphysical oscillations due to its unboundedness. If this is the case, Gauss upwind should be preferred.
The other terms are less problematic and commonly employed schemes may be selected so that the `fvSchemes` dictionary entries should be as written in the Listing 4.3.

4.3.3 fvSolution

When using the SIMPLE algorithm, a sub-dictionary `SIMPLE` has to be defined in `fvSolution`. The parameters are:

- `nNonOrthogonalCorrectors`: Number of correction loop to do to compensate the non-orthogonality of the mesh (0 is mesh is orthogonal and no more than 20 otherwise).
- `pRefCell` or `pRefPosition` [optional]: In closed incompressible system, the pressure is relative. Therefore a reference position and value as to be set. This parameter specifies the position at which the reference pressure is set.
- `pRefValue` [optional]: Reference pressure.
- `residualControl` [optional]: Specify convergence criteria. When the criteria are all fulfilled the simulation stops. For example to stop the simulation when the pressure and the velocity residuals are below 1e-05:

```
    SIMPLE
    {
            nNonOrthogonalCorrector 0;
            residualControl
            {
                    p    1e-05;
                    U    1e-05;
            }
    }
```

```
ddtSchemes
{
    default         steadyState;
}
gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
    grad(U)         Gauss linear;
}
divSchemes
{
    default         Gauss linear;
    div(phi,U)      Gauss linear;
    div((nuEff*dev(grad(U).T()))) Gauss linear;
}
laplacianSchemes
{
    default         none;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda) Gauss linear corrected;
    laplacian(1, p) Gauss linear corrected;
}
interpolationSchemes
{
    default         linear;
    interpolate(U)  linear;
}
snGradSchemes
{
    default         corrected;
}
fluxRequired
{
    default         no;
    p               ;
}
// ************************************************************************* //
```

Listing 4.3: fvSchemes

4.3.4 Sampling

The utility `sampling` can be used to sample field data at prescribed locations in the domain. The dictionary `sampleDict` is located in the *system* folder (cf. Listing 4.4 for an detailed example).
To interpolate data, three interpolation schemes are available: `cell`, `cellPoint` and `cellPointFace`. They determine the data by use of the cell-center values only (`cell`), the cell-center and vertex values (`cellPoint`) and cell-center, vertex and face values (`cellPointFace`). The user has the choice between different output formats for sets and surfaces. A common format is `raw` which exports data in text format `x y z value` that can be used for further post-processing. Finally in section `fields` it should be specified which fields to sample. To sample surface a base point and normal vector are required. It is also possible to extract values of prescribed boundary patches. The command to execute is simply `sample`.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.6                                   |
|   \\  /    A nd           | Web:      http://www.OpenFOAM.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version         2.0;
    format          ascii;
    class           dictionary;
    location        system;
    object          sampleDict;
}

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

// Set output format : choice of xmgr, jplot, gnuplot, raw
setFormat raw;

// Surface output format. Choice of
//      null        : suppress output
//      foamFile    : separate points, faces and values file
//      dx          : DX scalar or vector format
//      vtk         : VTK ascii format
//      raw         : x y z value format for use with e.g. gnuplot 'splot'.
// Note: other formats such as obj, stl, etc can also be written (by proxy)
// but without any values!
surfaceFormat vtk;

// interpolationScheme. choice of
//      cell          : use cell-centre value only; constant over cells (default)
//      cellPoint     : use cell-centre and vertex values
//      cellPointFace : use cell-centre, vertex and face values.
// 1] vertex values determined from neighbouring cell-centre values
// 2] face values determined using the current face interpolation scheme
//    for the field (linear, gamma, etc.)
interpolationScheme cellPoint;
```

```
// Fields to sample.
fields
(
    p
    U
);

// Set sampling definition: choice of
//      uniform             evenly distributed points on line
//      face                one point per face intersection
//      midPoint            one point per cell, inbetween two face intersections
//      midPointAndFace     combination of face and midPoint
//
//      curve               specified points, not nessecary on line, uses
//                          tracking
//      cloud               specified points, uses findCell
// axis: how to write point coordinate. Choice of
// - x/y/z: x/y/z coordinate only
// - xyz: three columns
//  (probably does not make sense for anything but raw)
// - distance: distance from start of sampling line (if uses line) or
//             distance from first specified sampling point
// type specific:
//      uniform, face, midPoint, midPointAndFace : start and end coordinate
//      uniform: extra number of sampling points
//      curve, cloud: list of coordinates
sets
(
    lineX1
    {
        type        uniform;
        axis        distance;
        start       (0 0 0.5);
        end         (0.1 0 0.5);
        nPoints     10;
    }
    lineX2
    {
        type        face;
        axis        x;
        start       (0 20  -20);
        end         (0 20   10);
    }
    somePoints
    {
        type    cloud;
        axis    xyz;
        points  ((0.049 0.049 0.005)(0.051 0.049 0.005));
    }

);
// Surface sampling definition: choice of
//      plane : values on plane defined by point, normal.
//      patch : values on patch.
//
// 1] patches are not triangulated by default
// 2] planes are always triangulated
// 3] iso-surfaces are always triangulated
```

```
surfaces
(
    constantPlane
    {
        type            plane;    // always triangulated
        basePoint       (0.0501 0.0501 0.005);
        normalVector    (0.1 0.1 1);
        //- Optional: restrict to a particular zone
        // zoneName        zone1;
    }
    interpolatedPlane
    {
        type            plane;    // always triangulated
        // make plane relative to the coordinateSystem (Cartesian)
        coordinateSystem
        {
            origin      (0.0501 0.0501 0.005);
        }
        basePoint       (0 0 0);
        normalVector    (0.1 0.1 1);
        interpolate     true;
    }
    movingWall_constant
    {
        type            patch;
        patchName       movingWall;
        // Optional: whether to leave as faces (=default) or triangulate
        // triangulate     false;
    }
    movingWall_interpolated
    {
        type            patch;
        patchName       movingWall;
        interpolate     true;
        // Optional: whether to leave as faces (=default) or triangulate
        // triangulate     false;
    }
    interpolatedIso
    {
        // Iso surface for interpolated values only
        type            isoSurface;    // always triangulated
        isoField        rho;
        isoValue        0.5;
        interpolate     true;
        //zone            ABC;          // Optional: zone only
        //exposedPatchName fixedWalls;  // Optional: zone only
        // regularise      false;    // Optional: do not simplify
    }
    constantIso
    {
        // Iso surface for constant values.
        // Triangles guaranteed not to cross cells.
        type            isoSurfaceCell;    // always triangulated
        isoField        rho;
        isoValue        0.5;
        interpolate     false;
        regularise      false;              // do not simplify
    }
```

```
    triangleCut
    {
        // Cutingplane using iso surface
        type            cuttingPlane;
        planeType       pointAndNormal;
        pointAndNormalDict
        {
            basePoint       (0.4 0 0.4);
            normalVector    (1 0.2 0.2);
        }
        interpolate     true;
        //zone            ABC;          // Optional: zone only
        //exposedPatchName fixedWalls;  // Optional: zone only
        // regularise      false;    // Optional: do not simplify
    }
);
// ************************************************************************* //
```

Listing 4.4: sampleDict

Exercises

4.1 2D mesh for channel pipe flow

The geometry we want to use for the case is shown in figure 4.4.

Figure 4.4: Geometry for channel pipe flow

We model this axisymmetric problem in 2D in order to save computational time. OpenFOAM always operates in 3 dimensional Cartesian coordinate system and all geometries are generated in 3D. The case is solved in 3 dimensions by default but can be instructed to be solved in 2 dimensions by specifying a 'special' `empty` boundary condition on boundaries normal to the (3^{rd}) dimension for which no solution is required. `Empty` corresponds to a 'symmetry' boundary condition. Furthermore we take advantage of the fact that the channel pipe is axisymmetric. For this purpose we only need to consider a slice of the tube and can use the element type `wedge`. It is suited for axisymmetric geometries such as tubes and ducts. It is important to model not more than a slice of $5°$ because we model only one cell in the 3^{rd} dimension. We neglect the bending of the wall, otherwise we could use `arc` in our `blockMeshDict`.

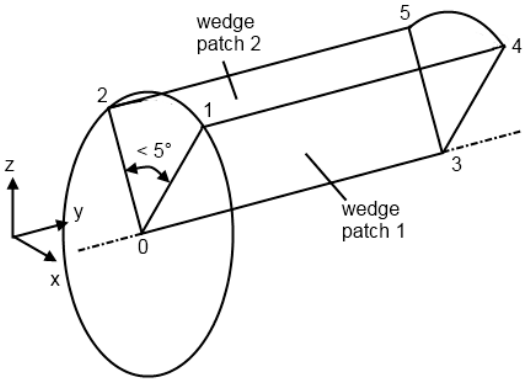


Figure 4.5: *Wedge* patch type used for axi-symmetric geometry

The boundary condition for the center axis will be *empty*. As in chapter 1 the mesh characteristics are defined in the *blockMeshDict* using blockMesh. A recommended initial mesh resolution is 30 points in radial and 100 points in flow direction. Since we want to study the effect of the wall on the flow we increase the mesh resolution close to the wall in relation to the mean flow in the center of the channel pipe by setting `simpleGrading` for example to 0.35 in x-coordinate.
Finally we define six different boundary conditions, two patches for `inlet` and `outlet`, `wall`, `empty` for the axis and `front` and `back` as `wedge` type.

4.2 Physical properties

We want to analyse a laminar flow in the channel pipe. So we need to make sure that the Reynolds number is in the appropriate range. The kinematic viscosity ν is represented by `nu` in the dictionary `transportProperties` in the `constant` folder. We use a gaseous media and set `nu` to $2e^{-5}kg/(m\,s)$. In order to reach a Reynolds number of approx. 50 we define constant inlet velocity of $|U| = 0.005m/s$ according to:

$$Re = \frac{d|U|}{\nu} \tag{4.31}$$

where d is the characteristic length, in our case the diameter of the channel pipe. For the outlet we specify the kinematic pressure as a fixed value of $0\,m^2/s^2$. Since we calculate an incompressible flow the indicated pressure is relative, not the absolute value.

As mentioned above, we define the discretization schemes in `fvSchemes`. The scheme used for convective terms is Central Difference indicated by `Gauss linear`. Since `simpleFoam` is a steady-state solver we set the time scheme to `steadyState` indicated by `ddtSchemes`. Linear solvers and algorithms are controlled in the `fvSolution` dictionary and we use for the velocity U the solver `smoothSolver` and for the pressure p `GAMG`.
simpleFoam is a solver designed for the analysis of turbulent flows. For our laminar case we need to modify the turbulence modeling method selectable in the `turbulenceProperties` dictionary in the folder `constant`. Here we have to switch off turbulence and set the `turbulenceModel` to `laminar`. For each turbulence model required coefficients called `Coeff` are stored below in `turbulenceProperties`.

4.3 Running the code

Finally the user should modify the `controlDict` to prepare the run. We set `startTime` to 0 and for this iterative solver we can define `deltaT` to 1. Set the appropriate `residualControl` such that the simulation stops when the pressure and the *radial* and *axial* components of the velocity residuals are below 1e-5.

4.4 Postprocessing and validation against analytical solution

After the calculation has finished we check the convergence of the solution by viewing the log-file. During the run, the user should check the initial residuals at each time step. The residuals should not change in the latest iterations, otherwise the solution takes more iterations to converge and needs additional time to run. Please right in the report the value of the initial residuals after the first iteration and at the latest one. Is the simulation converged?
The next step is to analyze the solution with *paraview* to get a first impression. Later on we use the `sample` command to extract a velocity profile at the end of the channel pipe flow where we consider the flow to be fully developed. This can be postprocessed for example with *Excel/openOffice*, *matlab* or *scilab* for visualization. Here we also implement the analytical solution to validate our numerical results.

4.4 Extra Practice and Background Information

Extra Practice

Increasing the Reynolds number

An increase of the Reynolds number requires an additional entry length to fully develop the flow. It has to be taken into account that our solver is not suited to solve turbulent flows with the settings made above. Therefore a laminar flow has to be guaranteed in the boundary conditions.
Increase the Reynolds Number and adapt the length of the domain. Is there a correlation between Reynolds Number and entry length[1]?

Development of the laminar profile from a block profile

Due to wall friction the flow will reach the laminar profile when a block profile is defined initially in the whole domain. Initialize the problem to start from a top-hat profile[2] and set the write interval to 20. When does the flow reach the steady state?

Reduction of spatial resolution

In order to save computational resources it is recommended to reduce the grid size, e.g. the amount of cells, as far as possible. This has a limit since we still want to compute the right solution. Reduce the spatial discretization successively until the solution does not match the analytical solution anymore. Which resolution is more important for this problem, the radial or the axial?

1 The entry length is the distance from the entry of the pipe needed by the flow to reach the parabolic profile i.e. the fully developed profile.

2 The velocity inside the domain is equal to the one defined at the inlet.

Background Information

4.4.1 Discretization best practice guidelines

One of the key step during the generation of a new case is the choice of discretization schemes. The following paragraph is an attempt to provide some guidelines for those choice[3]. As primary remark, the settings in the tutorials are appropriate but not recommended in general. In the opposite of commercial CFD softwares, OpenFOAM® doesn't come with default settings optimized on thousands of validation cases.

The accuracy of a simulation is strongly linked to the mesh quality and the numerical schemes. In particular, the sensible choice are the gradient scheme and the schemes of bounded scalars. If you want more information on the theory of the discretization schemes, have a look at the presentation of Prof. Jasak (available on our server */nfs/public/Numerik/OpenFOAM/HJ_Workshop2011*). But as this script focuses on OpenFOAM® usage only one scheme by configuration will be presented in the table 4.1.

Operator	**Restriction**	**Hex mesh**	**Tet mesh**
Gradient		Gauss linear (with or without limiter)	leastSquares (without limiter)
Convection	U	linearUpwind	reconCentral[1]
	scalar	any TVD/NVD schemes (e.g. Gamma)	
	turbulence	upwind[2]	
Laplacian	Non-orth < 60°	Gauss linear corrected	Gauss linear limited 0.5
	Non-orth > 70°	Gauss linear limited 0.5	
Time derivative	non-LES	Euler	
	LES	backward or CranckNicholson	
Interpolation		linear	reconCentral[1]

Table 4.1: Discretization best practice guidelines. [1] Scheme available only in OpenFOAM-extend. [2] Upwind is very stable. But then you could move to higher order for better accuracy.

[3] Those notes are inspired from a presentation done by Prof. Hrvoje Jasak, main developer of the OpenFOAM-extend who kindly agrees to shared his knowledge.

Chapter

5 Channel flows with Heat Transfer

5.1 Introduction

In this chapter we will handle channel flows with heat transfer to the walls. We will use a compressible solver to solve the flow in a complex geometry. For the meshing of this complex geometry, an automated mesh generation and refinement tool called `snappyHexMesh` will be presented. However, meshing of complex geometries involves a large number of cells and long simulation times. A second more favorable approach using source terms to model the effects of the complex geometry will be subsequently followed. This will be done by considering the heater as a porous media. A comparison of both methods should prove the interest of modeling in CFD as it allows drastic decrease of the computational cost at the expense of a reduced accuracy in the solution.

5.1.1 Todays problem

Figure (5.1) shows the test case considered in this chapter. Cold air flows through a rectangular channel and is heated up by a so called "stack". The stack consists of a copper solid with thin slots along the axial direction and occupies only a part of the channel. It is electrically heated with a maximum power of $\dot{Q}_{max} = 2.2W$. Due to the small dimensions, the slots behave like capillaries, and thus the flow is laminar. The stationary case will be studied.

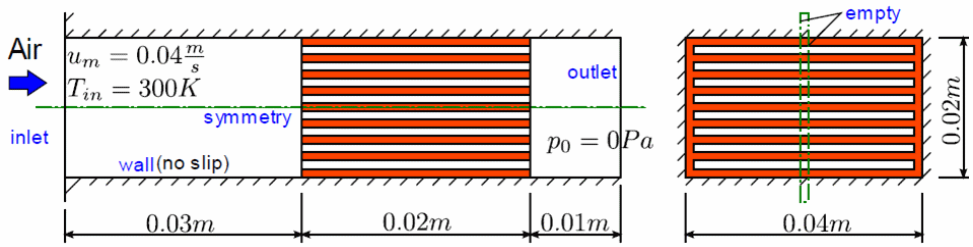


Figure 5.1: Channel flow, with a section heated up by a stack with small capillary slots.

Bibliography

[1] Wäermeübertragung, *Polifke and Koptiz, Pearson, 2nd edition, 2009*

[2] Analytical methods for heat transfer and fluid flow problems *Weigand, B. Springer, 2004*

[3] Computational methods for fluid dynamics *Ferziger, J.H and Peri, M. Springer Berlin, 1999*

[4] http://foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf *OpenFOAM® Programmers Guide*, Version 1.6, 24th July 2009

5.2 Physics

The resolution of the energy conservation equation forces us to use a compressible solver: `rhoSimpleFoam`. However the small variation in temperature allows the

hypothesis of constant thermophysical properties.

The continuity equation for an compressible fluid is:

$$\vec{\nabla} \cdot (\rho \vec{u}) = 0 \tag{5.1}$$

The momentum equation writes:

$$\frac{\partial \rho \vec{u}}{\partial t} + \vec{\nabla} \cdot (\rho \vec{u} \otimes \vec{u}) = -\vec{\nabla} p + \vec{\nabla}(\mu \vec{\nabla} \vec{u} - 2/3 \mu \vec{\nabla} \cdot \vec{u} I) + \vec{S}_M \tag{5.2}$$

with the terms named from left to right as: instationary, convection, pressure, diffusion (laplacian) and source term. The momentum source term is weighted by the density and thus $\vec{S}_M$ represents a volume force with the dimension $[N/m^3]$ acting on the fluid. It can be for example a gravitational force.

For the energy conservation, we start from an enthalpy h balance and a Fourier approach for the heat fluxes:

$$\frac{\partial \rho h}{\partial t} + \vec{\nabla} \cdot (\rho \vec{u} h) = \vec{\nabla} \cdot (\lambda \vec{\nabla} T) + \dot{S}_E \tag{5.3}$$

Introducing the thermal diffusivity $a = \frac{\lambda}{\rho c_p}$ the equation solved in OpenFOAM® appears:

$$\frac{\partial \rho h}{\partial t} + \vec{\nabla} \cdot (\rho \vec{u} h) = \vec{\nabla} \cdot (\rho a \vec{\nabla} h) + \dot{S}_E \tag{5.4}$$

In analogy to equation (5.2), the terms are again named from left to right as: instationary, convection, diffusion and source term. $\dot{S}_E$ represents a temporal energy volume source with dimensions $[W/m^3]$.

5.2.1 Laminar flow in a planar channel

In the previous chapter, the laminar solution for a pipe flow has been presented. The characteristic length is the channel height $2H$. Using Cartesian coordinates (notice the alignment of the axes in the figure), the hydrodynamically fully developed flow writes:

$$\frac{u}{u_m} = \frac{3}{2}\left(1 - \left(\frac{x - H}{H}\right)^2\right) \tag{5.5}$$

where u_m is the mean axial velocity and x is the distance to the bottom wall. It is as well a parabolic profile. The hydrodynamic entrance length can be approximated as follows:

$$l_e \approx 0.056 \mathrm{Re} \cdot H = 0.056 \frac{H^2 u_m}{\nu} \tag{5.6}$$

The solution procedure is very similar to the one in a pipe and can be taken from [2].

For the heat transfer problem, we will make use of the adiabatic bulk-temperature [1]:

$$T_m(z) = \frac{1}{\dot{M}} \int_A \rho T(z,x) \vec{u} \cdot d\vec{A} = \frac{1}{u_m H} \int_0^H T(z,x) u(x) dx \tag{5.7}$$

For an hydrodynamically developed flow, the axial distribution of T_m can be analytically determined for the Neumann boundary condition (constant wall heat flux $\dot{q}_w = const$) by solving equation (5.4) in an integral form. It is a linear profile of the form:

$$\frac{dT_m}{dz} = \frac{dT_w}{dz} = \frac{a\dot{q}_w}{\lambda u_m H} \tag{5.8}$$

For the Dirichlet boundary condition (constant wall Temperature $T_W = const$), an iterative solution has to be applied. Figure (5.2) shows qualitatively the temperature profile for both cases [1]:

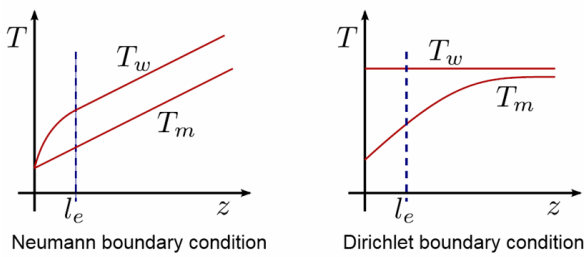


Figure 5.2: Temperature profile for constant wall heat flux (left) and constant wall temperature (right).

5.3 OpenFOAM®

5.3.1 Numerics in OpenFOAM®

The majority of physical problems concerning fluid dynamics can be described using the tensor calculus of up to rank 2, in other words, with operators dealing with scalars, vectors and second-rank tensors. Notice that in OpenFOAM® scalars are treated as tensors of rank 0 and vectors as tensors of rank 1. For the tensor representation, three basis classes have been created in OpenFOAM®: `scalarField`, `vectorField` and `tensorField`, which are essentially ordered lists of node values, however without any geometrical information. OpenFOAM® is build up using an object-oriented programming methodology in C++, and more complicated classes were subsequently created encapsulating and inheriting these basis classes. The next level of tensor classes are referred to as "geometric tensor fields" and, as

their name already suggests, they are enriched with positional information. This is achieved by a reference to a mesh class called `fvMesh` and encapsulation of the basic tensor classes. Again, the three ranks are present: `volScalarField`, `volVectorField` and `volTensorField` . Furthermore, these classes contain boundary information, previous time steps necessary for temporal discretization and dimensional information (SI).

Equation discretization

In the FVM, spatial differential equation for a certain problem are converted into a set of algebraic equations. The order and form of these equations depend on the discretization scheme, as explained in section (1.3). OpenFOAM® rearranges this set of equations into a matrix form:

$$A\vec{x} = \vec{b} \tag{5.9}$$

A is a square matrix of size $n \times n$, where n is the elements or nodes number, $\vec{x}$ is a column vector of the dependent variables and $\vec{b}$ is as well a column vector accounting the source and boundary condition terms. For a cell center mesh, the vector $\vec{x}$ contains the cell center values of searched quantities (T, $\vec{u}$, etc.). In other words, it is a list of values defined at locations in the geometry. Depending on the quantity, it can be a

- `volScalarField`
- `volVectorField`
- `volTensorField`

For the computation of the terms present in equation (5.9), two types of tensor derivative classes are implemented in OpenFOAM®: finite volume calculus (`fvc`) and finite volume method (`fvm`). The main difference between them is that `fvm` performs the evaluation in an implicit way returning a matrix. Implicit means in this context, that the matrix can be used to advance the dependent variables by a time step. Two classes are created to store the matrices: `fvMatrixScalar` and `fvMatrixVector`. On the other hand, `fvc` performs the evaluation in an explicit way returning a geometric tensor field (`vol<type>Field`). Explicit means that the field values are already known and the derivatives are only geometrical operations. Thus, `fvc` performs only a mapping from one tensor field to another.

These two classes are probably the most important ones in OpenFOAM®, since they inherit all the available discretization schemes of the code. Table (5.1) lists the main differential operators that are available in `fvm` and `fvc`.

Finite volume discretization of each term is formulated mostly by first integrating the term over a cell volume V, and then converting this volume integral into a surface integral using Gauss's theorem [3]:

$$\int_V \vec{\nabla} \star dV = \int_S d\vec{S} \star \phi \tag{5.10}$$

where $\star$ can be any tensor product and ϕ can represent any geometrical tensor field (`vol<type>Field`). Volume and surface integrals are linearized using appropriate interpolation schemes. Specially for the surface integrals and later for the boundary conditions too, a further set of classes is created: `surfaceScalarField`

Operator	Exp/Imp	Formula	OpenFOAM® syntax	Example
Gradient	Exp	$\vec{\nabla}\chi$	`grad(chi)`	`grad(p)`
		$\vec{\nabla}\phi$	`gGrad(phi)`	`gGrad(p)`
			`lsGrad(phi)`	`lsGrad(p)`
			`snGrad(phi)`	`snGrad(p)`
			`snGradCorrection(phi)`	`snGradCorrection(p)`
Divergence	Exp	$\vec{\nabla}\cdot\chi$	`div(chi)`	`div(U)`
Curl	Exp	$\vec{\nabla}\times\phi$	`curl(phi)`	`curl(U)`
Convection	Exp	$\vec{\nabla}\cdot\psi$	`div(phi)`	`div(U)`
	Imp / Exp	$\vec{\nabla}\cdot(\psi\phi)$	`div(psi,phi)`	`div(phi,U)`
Laplacian	Imp / Exp	$\vec{\nabla}^2\phi$	`laplacian(phi)`	`laplacian(T)`
		$\vec{\nabla}\cdot\Gamma\vec{\nabla}\phi$	`laplacian(Gamma,phi)`	`laplacian(a,T)`
Source	Imp	$\rho\phi$	`Sp(rho,phi)`	`Sp(CM,U)`
	Imp / Exp		`SuSp(rho,phi)`	`SuSp(CM,U)`
Time derivative	Imp / Exp	$\frac{\partial\phi}{\partial t}$	`ddt(phi)`	`ddt(U)`
		$\frac{\partial\rho\phi}{\partial t}$	`ddt(rho,phi)`	`ddt(rho, U)`
Second time derivative	Imp / Exp	$\frac{\partial}{\partial t}\left(\rho\frac{\partial\phi}{\partial t}\right)$	`d2dt2(phi)`	`d2dt2(U)`
			`d2dt2(rho,phi)`	`d2dt2(rho,U)`
Function arguments can be as follows: ϕ: `vol<type>Field` ψ: `surfaceSacalarField` χ: `surface<type>Field`, `vol<type>Field` Γ: `scalar`, `volScalarField`, `surfaceScalarField`, `volTensorField`, `surfaceTensorField` ρ: `scalar`, `volScalarField`				

Table 5.1: Differential operators offered by OpenFOAM® [4].

and `surfaceVectorField`, which are nothing else than fields defined on surface cell centers.

With the previously described classes the operators of the partial differential equations are evaluated. The matrices returned by the implicit operators are then summed up into the global matrix A, while the vectors returned by the explicit operators are summed up to the vector $\vec{b}$. For details on the discretization of differential operators see section (2.3), the programmers guide [4] or [3].

Since we will deal with source terms in this chapter, an explanation of its implementation will follow here. In OpenFOAM®, source terms can be specified in three ways:

Explicit: If the source term is not a function of the dependent variables, it can be incorporated into an equation simply as a field of variables (`vol<type>Field`). This field might be given directly by the user, or it could be as well the output of any `fvc` operation of a known field (previous time step, time independent

field,...). The source term will thus have only entries into the vector $\vec{b}$ in equation (5.9).

Implicit: If the source term is a function of the dependent variables, it has to be incorporated through the `fvm` class and it will produce entries into the matrix A of equation (5.9). For its evaluation, an implicit source term is integrated over a control volume weighted by the density and linearized by:

$$\int_V \rho \dot{S} dV = \rho_p V_p \dot{S}_p \tag{5.11}$$

where the subscript p denotes the cell average computed by an interpolation scheme.

Implicit/Explicit: The matrix A is a sparse matrix with diagonal dominance. The iterative solution of the matrix equation (5.9) might get instable if the diagonal dominance of the matrix is decreased. Since an implicit source term will change the coefficients of the diagonal of matrix A, it will affect the diagonal dominance of the matrix. Negative coefficients decrease diagonal dominance, positive coefficients increase diagonal dominance. To assure stability of the whole system, OpenFOAM® provides a mixed source term discretization procedure. It handles the source term explicit for the negative diagonal coefficients, and implicit for the positive diagonal coefficients.

Example

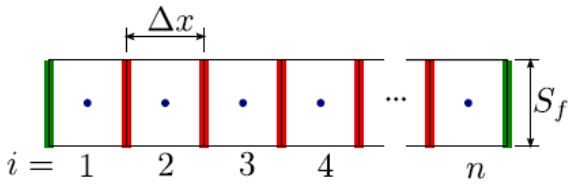


Figure 5.3: 1D-domain

To clarify the difference between the two classes `fvc` and `fvm`, lets handle a simple example. Consider a 1-dimensional domain discretized into a mesh of n elements shown in figure (5.3). Throughout the domain, values of a scalar field ψ are known and fixed. Suppose we want to calculate a new field ϕ which satisfies the following condition:

$$c\vec{\nabla}^2\phi = \vec{\nabla}\psi \tag{5.12}$$

where c is a constant. Expression (5.12) corresponds to our PDE. We can then arrange all the values of ϕ_i in the discretized domain into a vector $\vec{\phi}$. We do the same for the known values ψ_i. According to the OpenFOAM® notation described in the previous sections, the vector of dependent variables $\vec{\phi}$ and the vector of known values $\vec{\psi}$ would be objects of the class `volScalarField` of size n named `phi` and `psi` respectively. The PDE (5.12) can be then discretized using the `fvc` and `fvm` classes as following:

```
fvm::laplacian(c,phi) == fvc::grad(psi)
```

Since the field ψ is known, its gradient can be evaluated explicitly with the class `fvc`. The laplacian of the field ϕ has to be evaluated implicitly with the class `fvm`.

As explained in the previous subsection, the differential operators are integrated over each cell volume and transformed then to surface integrals through Gauss theorem. The surface integrals are then expressed as summation over cell surface values. For the laplacian operator this looks for one cell like:

$$\int_V \vec{\nabla} \cdot (c\vec{\nabla}\phi) dV = \int_S d\vec{S} \cdot (c\vec{\nabla}\phi) \approx \sum_{f=1}^{2} c\vec{S}_f \cdot (\vec{\nabla}\phi)_f \tag{5.13}$$

Since we are dealing with a 1D-problem, each element has only two relevant faces, the red colored in figure (5.3). To evaluate the gradient at the surfaces, for simplicity linear interpolation is chosen in this example, however higher schemes can been chosen:

$$\int_V \vec{\nabla} \cdot (c\vec{\nabla}\phi) dV \approx \sum_{f=1}^{2} c \left[S_f \frac{(\phi_N - \phi_p)}{\Delta x} \right] \tag{5.14}$$

where the subscript N denotes the neighboring cell and the subscript p the cell were the balance is being made. In a similar way, the discretization of the gradient operator look like:

$$\int_V \vec{\nabla}\psi dV = \int_S d\vec{S}\psi \approx \sum_{f=1}^{2} \vec{S}_f \psi_f = \sum_{f=1}^{2} \vec{S}_f \frac{\psi_N + \psi_p}{2} \tag{5.15}$$

For each of the domain cells, equation (5.12) can be discretized as:

$$\sum_{f=1}^{2} c \left[S_f \frac{(\phi_N - \phi_p)}{\Delta x} \right] = \sum_{f=1}^{2} \vec{S}_f \frac{\psi_N + \psi_p}{2} \tag{5.16}$$

This is a system of equations, which can be written in matrix form as:

$$S_f c \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \vdots & \\ 0 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_n \end{bmatrix} = S_f \begin{bmatrix} -BC_L + \frac{\psi_3}{2} + \psi_1 \\ \frac{\psi_1 + \psi_3}{2} + \psi_2 \\ \frac{\psi_2 + \psi_4}{2} + \psi_3 \\ \vdots \\ -BC_R + \frac{\psi_{n-1}}{2} + \psi_n \end{bmatrix} \tag{5.17}$$

where BC stands for the boundary conditions on the left and the right, which can be of Neumann (fixed gradient) or Dirichlet (fixed value) type. In this simple example, the surface cancels out. Now we can compare equation (5.17) with the OpenFOAM® expression presented above. The implicit evaluation called by the expression `fvm::laplacian(c,phi)` returns the matrix on the left hand side of equation (5.17), and the explicit evaluation `fvc::grad(psi)` returns the vector on the right hand side. The vector ϕ on the left hand side is the one declared in the `createFields` header file, and is of course mesh dependent. Important to notice is, that the matrix A is sparse, this means that it has few entries different from zero

on each line, but always one for the diagonal. Depending on the numbering of the mesh, the other non zero entries might be near to the diagonal or not. Strictly speaking, the boundary conditions terms are incorporated into vector $\vec{b}$ in a subsequently step further in the code. There is a function called `correctBC`, but it's explanation is beyond the scope of this chapter.

5.3.2 Mesh generation with the snappyHexMesh utility

The snappyHexMesh utility provided by OpenFOAM® is an automated mesh generation tool, that generates 3-dimensional meshes containing hexahedra (hex) and split-hexahedra (split-hex) from geometries defined by surfaces in a `STL` format. Additionally, it can refine existing meshes on volume regions. The specification of mesh refinement level is very flexible and the surface handling is robust with a pre-specified final mesh quality.

`STL` is a file format commonly used in the stereolithography for the representation of three dimensional objects. It describes the geometry only by a set of raw unstructured triangulated surfaces, which are themselves defined by their unit normal vectors and vertices. `STL` files exist both in ASCII and binary format.

In order to create the mesh, the user needs to provide the following to the utility:

- surface data files in `STL` format, either binary or ASCII, located in a `triSurface` sub-folder of the `constant` directory
- a background solely hex mesh usually generated using `blockMesh`, which defines the extent of the computational domain and a base level mesh density
- a `snappyHexMeshDict` dictionary, located in the `system` directory of the case

The utility creates the mesh in three main steps displayed in Figure (5.4) with an example geometry. It starts splitting the cells of the background mesh that are intersected by the STL-surfaces. This splitting already induces a mesh refinement in the vicinity of the surfaces, and the ratio can be specified in the dictionary. After that, cell removal starts. Only cells having 50% of their volume in the desired region will be kept. Thus, the user has to specify a point inside the desired region. That's why only closed surfaces make sense for this option. Additionally mesh refinement can be done on volume regions, e.q. boxes like in this example. The next step is the morphing of the resulting split-hex mesh to the surface. During the third step, the tool will optionally insert cell layers adjacent to chosen surface patches before the mesh quality provement is achieved.

Below, an example of the `snappyHexMeshDict` dictionary with its main structure is given. It begins at the top level with switches for the various stages of the meshing process (the `castellatedMesh` switch concerns to the mesh refinement and cell removal steps). Individual sub-dictionaries follow for each process. Below, the main structure of the dictionary is shown. In the `geometry` subdictionary, all the geometry used by `snappyHexMesh` is specified through either STL files or simple bounding geometry entities (boxes and spheres). A geometry in an STL file might have several patches (portions of a surface), which can be treated separately in the geometry subdictionary. In the other subdictionaries, several parameters have to be set. The template dictionary in OpenFOAM® provides a lot of comments that serve as a help reference. A copy of that file is given in the `/Exercises/Chpt4/snappyHexMesh/` folder, take a look at it.

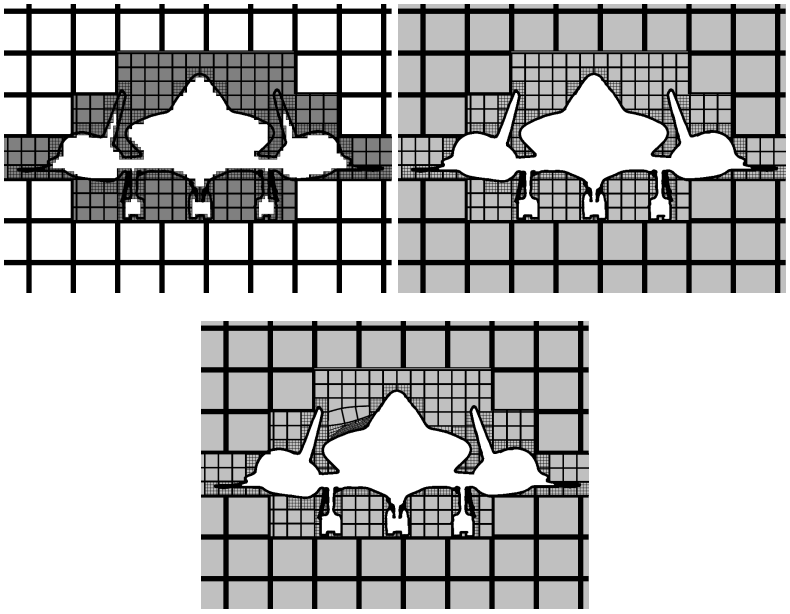


Figure 5.4: Three main steps of the meshing process when using snappyHexMesh. Left: cell refinement and removal. Rigth: cell morphing to the given surfaces. Bottom: Boundary layer addition.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.6                                   |
|   \\  /    A nd           | Web:      http://www.OpenFOAM.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      snappyHexMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
castellatedMesh true;
snap            true;
addLayers       true;

geometry { ... };                 // Geometry declaration
castellatedMeshControls  { ... }; // Settings for the refinement
snapControls { ... };             // Settings for the snapping
addLayersControls { ... };        // Settings for the layer addition
meshQualityControls { ... };      // Settings for the mesh quality

debug 0;                          // debug level
mergeTolerance 1E-6;              // tolerance
// ************************************************************************* //
```

Listing 5.1: Extract of `snappyHexMeshDict`.

5.3.3 Porous media and multiple reference frame (MRF) modeling

OpenFOAM® comes with a series of solver able to simulate porous media; e.g. `rhoPorousMRFSimpleFoam`. A porous media blocks the flow and can exchange energy with it. The blockage effect always reduces the pressure losses in the system. This is represented by a sink in the momentum equation. Two models for pressure losses are currently available:

- the power law with 2 coefficients (C_0 and C_1):

$$S_M = -\rho C_0 \left|U\right|^{(\frac{C_1-1}{2})} U \tag{5.18}$$

- the Darcy law with 2 coefficients (d and f):

$$S_M = -(\mu d + \frac{\rho|U|}{2})U \tag{5.19}$$

The heat transfer models available are:

- None: the porous media is adiabatic.
- fixed temperature: set the temperature of the flow to a given temperature.

Those models are of low interest. Therefore if you want something more advance, you will have to code a new `thermalModel` for porous media. For example, the exercise of this chapter uses a model with constant heat release.

All those parameters have to be specified in a dictionary `constant/porousZones` (see Listing 5.2).

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      porousZones;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

1   // Number of porousZone
(
    porosity // Name of the porous cell zone
    {
        // Settings for coordinate system are described in coordinateSystem.H
        coordinateSystem    system_10; // optional default origin (0 0 0) and cartesian
        printCoeffs     yes;        // optional: feedback for the user

        /*
        Darcy // coefficients
        {
            d   d [0 -2 0 0 0 0 0] (0 0 1e5);
            f   f [0 -1 0 0 0 0 0] (0 0 0);
        }
        */
```

```
        powerLaw // coefficients
        {
            C0  325;
            C1  1;
        }

        thermalModel
        {
            type        fixedTemperature; // or none

            // fixedTemperature coefficients
            T           500;
        }
    }
)

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

Listing 5.2: porousZones

A framework to simulate fan in a system (e.g. in air conditioning application) is also available in some solvers: e.g. `rhoPorousMRFSimpleFoam`. The model uses the so-called Multiple reference frame (MRF) to represent the fan in a different reference than the pipe. A fan creates Coriolis forces to be added in the momentum equation. The specification for the MRF are to be given in `constant/MRFZones` (see Listing 5.3).

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      MRFZones;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

1   // Number of MRFZones
(
    rotor // Name of the MRF cell zone
    {
        // Fixed patches (by default they 'move' with the MRF zone)
        nonRotatingPatches ();

        origin    origin [0 1 0 0 0 0 0]  (0 0 0); // Origin of the fan
        axis      axis   [0 0 0 0 0 0 0]  (0 0 1); // Axis of rotation
        omega     omega  [0 0 -1 0 0 0 0] 1047.2;  // Angular speed
    }
)

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

Listing 5.3: MRFZones

To specify a name for a zone in OpenFOAM®, there is a optional keyword that can be inserted in a `block` definition in `constant/blockMeshDict` between the vertex list and the number of cells:

```
blocks
(
    hex (4 5 6 7 8 9 10 11) porosity (10 1 20) simpleGrading (1 1 1)
);
```

5.3.4 Convergence to a steady state

In case the flow field is statistically steady or periodic over time it is important for a meaningful time average whether the simulation has reached this point of periodicity. Otherwise transient flow development processes from the numerical and probably unphysical start conditions to the developed flow field adulterate the quality and correctness of the time average. To determine whether the flow is developed or not, the values of different appropriate variables at certain points can be sampled. If the curves show periodic behavior the flow is developed. For this task the probe-function of OpenFOAM®. This method is also useful to determine whether a time average is converged, which means that the averaged values does not change anymore with further averaging. This can be seen if an averaged variable is probed at a position with the most transient flow behavior and does not change anymore from a certain point on.

A good way to evaluate a real steady state configuration is to look at the residuals of the fields. When the residuals are stable at a low value (roughly inferior at 10^{-6}), the system is in a steady state.

In OpenFOAM®, the residuals can be analyzed if the output information are stored in a log file. Then by using a tool to analyze that log file, the residuals can be plotted or stored in separated files.

There are 3 main tools :

- `gnuplot`
- `foamLog`
- `pyFoamPlotRunner.py`

The latest tool won't be described in this hand-out. But you can find more information about it on *http://www.openfoamwiki.net/index.php/Contrib_PyFoam*.

foamLog tool

`foamLog` is a standard tool shipped with OpenFOAM® to analyzed the log file.

The syntax of the tool is :

```
foamLog [-n][-s] <log>
    extracts xy files from log
foamLog -l <log>
    lists the variables but does not extract
foamLog -h
    for a help message
```

The default is to extract for all the 'Solved for' variables the initial residual, the final residual and the number of iterations.

The program will write a set of files, logs/<var>_<subIter>, for every <var> specified, for every occurrence inside a time step with the initial residual named <var>.

The files are a simple *xy* format with the first column *Time* (default) and the second the extracted values. Options :

- -n creates single column files with the extracted data only
- -s suppresses the default information and only prints the extracted variables

After you have run foamLog, you can plot the residuals by typing for example for the pressure residuals :

```
gnuplot
gnuplot> plot 'logs/p_0'
```

To visualize the time evolution, a bash script that runs foamLog with some period has to be used. Here is an example :

```
#!/bin/sh
#
# Run the foamLog tool periodically
# and then launch gnuplot to visualize the results
#
# This script accepts two optional parameters :
# $1 = name of the log file (default = "log.*Foam")
# $2 = save in a file the residuals - value : "on" or "off" (default = "off")
#
# author Frederic Collonval
# version 2011.02.24
# email collonval@td.mw.tum.de

SAVE="off"
LOG="log.*Foam"

if [ $# -ge 1 ]; then
    LOG=$1
fi
echo "The log file is $LOG"
if [ $# -ge 2 ]; then
    if [ $2 = "on" ]; then
      SAVE=$2
      echo "The residuals will be save in \"Residuals.png\""
    fi
fi



# Look if foamLog was not previously executed
if [ ! -e ./logs/foamLog.awk ]; then
    echo "Execute foamLog"
    foamLog -s $LOG

# Move each residual you want to plot to a file name ./logs/res_*
```

```
    for f in $( ls ./logs/?_0 ./logs/??_0 ./logs/ep*on_0 ); do
        name=`echo $f | cut -d'/' -f3`
        mv $f ./logs/res_$name
    done
fi

# create a gnuplot script to plot the residuals
if [ ! -e plot.gnu ]; then
    echo "Generate␣the␣gnuplot␣script"

    if [ $SAVE = "on" ]; then
        echo "set␣terminal␣png;␣set␣output␣\"Residuals.png\"" > plot.gnu
    else
        echo "␣" > plot.gnu
    fi

    echo -n "set␣logscale␣y;␣set␣ylabel␣\"Residuals\";␣set␣xlabel␣\"Time\";␣plot␣" >> plot.gnu
    K=0
    for f in $( ls ./logs/res_* ); do
        if [ $K -eq 0 ]; then
            echo -n "\"$f\"␣w␣l" >> plot.gnu
            K=1
        else
            echo -n ",␣\"$f\"␣w␣l" >> plot.gnu
        fi
    done
    # Introduce a break before closing the window
    echo -n ";␣pause␣3;" >> plot.gnu
    if [ ! $SAVE = "on" ]; then
         # Loop inside the gnuplot script except if output is a file
        echo -n "reread;" >> plot.gnu
    fi
fi

# Run in an external parallel process the script to extract the parameters
(
    J=0
    while [ $J -lt 1 ]; do
        echo "Execute␣the␣awk␣script␣on␣$LOG"
        awk -f ./logs/foamLog.awk $LOG

# Move each residual you want to plot to a file name ./logs/res_*
        for f in $( ls ./logs/?_0 ./logs/??_0 ./logs/ep*on_0 ); do
            name=`echo $f | cut -d'/' -f3`
            mv $f ./logs/res_$name
        done
        J=`tail $LOG | grep -c -e End -e Exit -e exiting`
    done
) &

echo "Subprocess␣launched"

# Plot the residuals (need to type Ctrl+C to stop gnuplot because of the command reread)
gnuplot plot.gnu

# remove the gnuplot script
rm plot.gnu
```

Listing 5.4: Example of bash script to run periodicaly `foamLog`.

Using gnuplot

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 example of the Listing 5.5 plots the initial residuals for *p*, *Ux*, *Uy* and *Uz*.

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

Listing 5.5: Script to visualize the residuals

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 9^{th} 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.

5.4 Exercises

Approach 1: fine geometry

In this first approach, you will try to model the problem with a relatively exact geometry. However, you will still approximate it to a quasi 2-dimensional domain marked with the green doted lines in figure (5.1). Furthermore, you will take advantage of the symmetry, and model only the bottom part of the channel. Copy the case `Approach1/fineStack`.

Meshing

You will mesh the geometry with the `snappyHexMesh` utility presented in section (5.3.2). The strategy is to use the mesh of the channel without stack as the background mesh, and to remove the cells occupied by the stack with the automated tool. Use the dictionary `Approach1/fineStack/constant/polyMesh/blockMeshDict` to create the background mesh with the `blockMesh` utility. The utility works better if the background mesh is uniform, and thus no mesh grading near the wall has been made.

A peculiarity has to be mentioned here. Even if the background mesh has only one cell in the z direction, it has been defined as 3-Dimensional. The boundaries on $y = 0.5mm$ and $y = -0.5mm$ have been attached to the symmetry patch. Thus they are not treated with the `empty` type of the previous sections. This is because the `snappyHexMesh` refines the mesh in all directions, and we will have more than 1 cell in the y direction after the refinement.

The geometry of the stack will be represented with a STL file created with ICEM[1]. Create a subfolder `triSurface` in the `constant` case folder, and copy the STL file from `Approach1/stack.stl`. It is in this case an ASCII file. Since all the surfaces of the stack will have later the same boundary conditions and mesh options, only a patch with the name `stack_WALL` has been declared (see the file with a text editor). You can view the mesh in `paraFoam` but, since we don't have yet initial fields, do not load the variable fields. `ParaFoam` can display STL geometries too. With the background mesh opened, load additionally the STL geometry with *file -> open -> ./constant/triSurface/stack.stl*.

You have provided the `snappyHexMeshDict` dictionary with the proper entries in the `Approach1/fineStack/system` directory. Take a look at it. The three options are switched on. In the `geometry` subdictionary, the STL file is loaded. Furthermore, two rectangular regions (`box1` and `box2`) with the type `searchableBox` have been declared too. They will help to refine the mesh near the channel wall. For simplicity, no comments have been printed in the dictionary.

You have now everything to create the fine mesh. In the case folder `fineStack` execute the command `snappyHexMesh`. As explained before, each step of the process shown in figure (5.4) will be separately stored in the folders `1`, `2` and `3` each of them with a `polyMesh` subdirectory. Take a look of all steps with `paraFoam`. You only need to execute `paraFoam` in the case folder, and the meshes will be treated

[1] Most CAD programs offer the option to export STL files. An open-source tool called *Blender* exists too. http://www.blender.org/

as different time steps. Use the next and back frame buttons to change between them. You will of course only use the third one. Create a temporary folder `TEMP` and move the three folders into it. In the `constant` folder, rename the `polyMesh` folder with the background mesh to `bak_polyMesh` and substitute it with a copy of the `polyMesh` subdirectory of the folder `TEMP/3/`.

You can check the created Mesh with the utility `checkMesh`. It displays several information about the Mesh, like cell number and cell type.

Meshing and setting up

You can now set the boundary conditions in the initial field files on the `\0` folder for pressure, velocity and temperature. Use `polyMesh\boundary` file and figure (5.1) to set the `p`,`U` and `T` initial files with the proper patch names and types. The channel wall is adiabatic. For the stack walls you will define a constant heat flux. Due to the good heat conduction of copper, the heating power of the stack can be homogeneously distributed over the whole surface:

$$\dot{q}_w = \frac{\dot{Q}_{max}}{A_s} = \lambda \frac{dT}{dy^*} \tag{5.20}$$

where y^* is here the local direction perpendicular to the stack walls.

The surface of the stack in contact with the fluid counts ca. $0.0197m^2$ and the maximum power of the stack is $2.2W$. Calculate and set the temperature gradient on the surface of the stack.

After setting the boundary conditions, you have to specify the thermodynamic properties. For that update the constants in `thermophysicalProperties` with the values given in the Table 5.2

Properties name	Keyword	value
Mean molecular weight	`molWeight`	$28.9kg/kmol$
Specific heat capacity	`Cp`	$1005\frac{J}{KgK}$
Enthalpy of formation	`Hf`	$2.544 \cdot 10^6 \frac{J}{kg}$
Dynamic viscosity	`mu`	$1.8 \cdot 10^{-5} Pa.s$
Prandtl number	`Pr`	0.7

Table 5.2: Averaged transport properties of air.

Finally adapt the `controlDict` and `fvSolution` files. The convergence criteria to be used is 1e-6 for all variable (i.e. pressure, enthalpy and velocity). The maximum number of iterations will be set to 500. The data will be stored the solution each 50 iterations. We will use the same discretization schemes as in the previous exercise.

You can now run the solver `rhoSimpleFoam`.

Questions

- Which value did you use for the heat flux boundary condition? How many cells had the mesh? How long took the solver to solve the problem?

- Analyze the axial temperature and pressure profiles. Use the plot over line option of `paraFoam` along one of the thin slots of the stack. Do they make sense compared with the classical patterns of laminar flow?
- What is the pressure drop Δp through the stack at this velocity?

Approach 2: modeling with porous media

In this second approach, we will use another strategy. The idea is to simulate the effects caused by the stack by adding source terms in the transport equations. The stack can be seen, due to its thin slots, as a porous medium. The walls of the stack induce friction forces through the viscosity on the fluid, which can be approximated as volume forces. Similarly, the wall heat fluxes can be homogenized into energy volume sources too.

$$\dot{S}_E = \frac{\dot{Q}_{max}}{V_s} \tag{5.21}$$

$$S_M = \frac{\Delta p A_\perp}{V_s} = \frac{\Delta p}{l_s} \approx f(\vec{u}) \tag{5.22}$$

The energy source term is more or less constant, without dependency of the flow field. The momentum source term is more complicated. Since the pressure drop is a function of velocity, the source term is generally flow field dependent too.

With the proper source terms, we will not need to mesh the complex geometry of the stack. Instead, a simple mesh created with `blockMesh` can be used. Figure (5.5) shows a sketch of the described domain divided into three blocks.

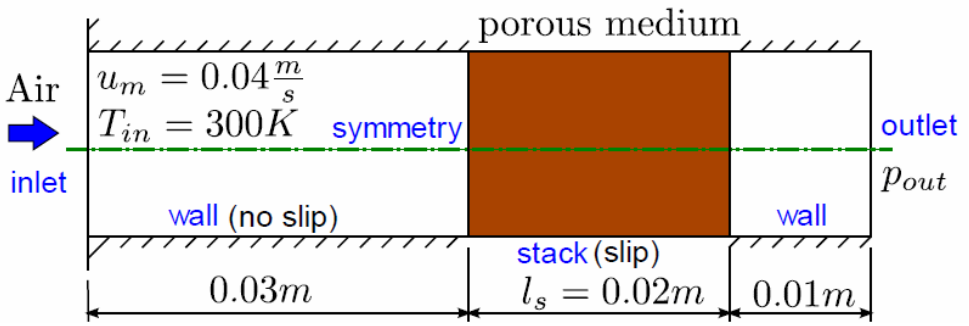


Figure 5.5: Simplified domain

Meshing and setting up

Copy the previous test case and rename the folder to `sourceStack`. Remove all time steps other than the initial one, the `constant/bak_polymesh` folder and the `TEMP` folder too. Change the `blockMeshDict` dictionary to create the simplified mesh with the block for the stack named `porosity`. The patch names correspond to the ones given in figure (5.5). The recommended mesh resolution is 5 to 10 cells

per millimeter with a grading towards the wall. Using expressions (5.21) and (5.22) and the pressure drop calculated in the previous approach, estimate the values of the coefficients S_E and the coefficients of the power law C_0 and C_1 (see equation 5.18). The `thermophysicalProperties` and `turbulenceProperties` files are already set correctly. You have now to add two dictionaries for the `porousZones` (see Listing 5.2) and the `MRFZones` (see Listing 5.3).
Remark: the `thermalModel` is a non-standard one, named `fixedHeatRelease` that requires one parameter `dQ`, the volumetric heat release. The code is provided in `Approach2/thermalModel`. To compile it, go in the directory and type `wmake libso`. Then to use it, add in the `controlDict` the following entry:

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

In the folder `sourceStack/0` modify the files for pressure, temperature and velocity applying the boundary conditions displayed in figure (5.5). The same applies for the solution accuracy and discretization schemes, so leave the `fvSchemes` and `fvSolution` as they are.

In the `controlDict` set the timestep to 1 and the maximum number of iterations to 500. Store the solution each 50 iterations and don't forget to specify the right solver. The convergence criteria will be the same as before (1e-6 for `p`, `h` and `U`). The case can now be solved by running `rhoPorousMRFSimpleFoam`.

Questions

- Check the independence of the results with the mesh used. For that plot the velocity profile before the porous zone for 2 meshes. If they are similar, you have reached mesh independence. If not refine the mesh up to the point, the profile is constant.
- For the channel wall section occupied by the stack, a slip condition has been chosen, why?
- How many cells composed the final case?
- Which values do you get for S_E, C_0 and C_1?
- Use `paraFoam` to analyse the results and make some plots for the temperature, velocity and pressure. Compare the plots with the ones of the previous approach. Do they look qualitatively similar? What about the pressure drop and temperature increase through the stack?
- Compare the time needed to solve the real geometry and the modeling one.

5.5 Extra Practice and Background Information

- You could trim the values for C_0, C_1 and S_E to get the same pressure drop and temperature increase as with the exact geometry.

Chapter

6 The Backward Step

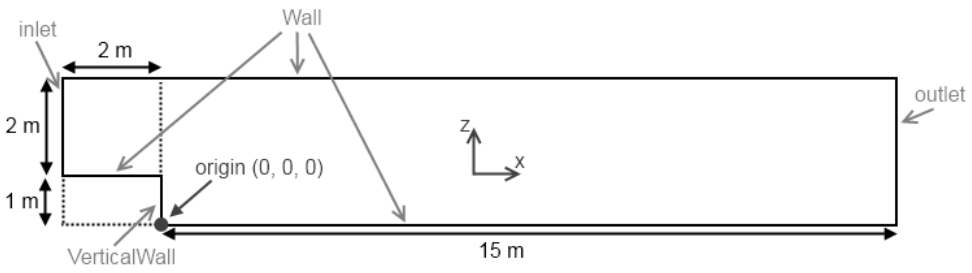


Figure 6.1: Todays problem - The mesh has to be cartesian and uniform with a size of the cells being 12.5 cmx 12.5 cm

6.1 Todays Problem

To test the capability of OpenFOAM® a flow problem with separation and reattachment is analyzed. The flow over a backward facing step is one of the simplest geometries that allows this kind of test even if, when the flow is turbulent, the flow structure is very complex. To ensure the quality of the simulation, a comparison with the experimental data of [1] is proposed.

Bibliography

[1] Three-dimensional particle-tracking velocimetry measurement of turbulence statistics and energy budget in a backward-facing step flow, *Kasagi, N. and Matsunaga A., Heat and Fluid Flow, Year 1995 vol 16 477-485*

[2] The numerical computation of turbulent flows *Launder, B.E. and Spalding, D.B. Comp. Meth. Appl. Mech. Eng. Year 1974 vol 3 269-289*

[3] Development of turbulence models for shear flows by a double expansion technique *Yakhot, V., Orszag, S.A., Thangam, S., Gatski, T.B. and Speziale, C.G. Physics of Fluids A, Year 1992 vol. 4, No. 7, pp1510-1520*

[4] Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications *Menter, F. R. AIAA Journal, Year 1994 vol. 32, pp. 269-289.*

[5] OpenFOAM® User Guide-1.6 *2009*

[6] 3-D PTV Database of Turbulent Flows, *Turbulence and Heat Transfer Laboratory - University of Tokyo*, 2004. URL : http://www.thtlab.t.u-tokyo.ac.jp/ (seen on 20-06-2010)

6.2 Physics

The todays problem concerns a fully developed flow reaching a step. When a 2D turbulent flow gets in a channel, the formation of the boundary layers can be observed. They modify the cross section of the channel and, therefore the velocity profile is changed. If x is the flow direction, the thickness of a boundary layer δ on a horizontal plate grows as:

$$\delta \propto x^{0.5}$$

Due to the interaction of the different boundary layers within the channel, it is not so easy to define this growing function. When this interaction reaches a steady state

the boundary layers do not grow anymore and the velocity profile does not change. This condition is the so called *fully developed state*. Its mathematical formulation is, being u and v, velocity respectively in x and y direction:

$$\begin{cases} v = 0 \\ \frac{\partial u}{\partial x} = 0 \end{cases}$$

The equations required for this numerical simulation are the equations of Navier-Stokes (for a viscous incompressible flow neglecting the gravity force)

Continuity

$$\vec{\nabla} \cdot \vec{u} = 0 \tag{6.1}$$

Momentum

$$\frac{D\vec{u}}{Dt} = \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \vec{\nabla} \otimes \vec{u} = -\frac{1}{\rho}\vec{\nabla} p + \nu \nabla^2 \vec{u} \tag{6.2}$$

and those of the turbulence model.

6.2.1 Turbulence Models

The aim of the turbulence modeling is to introduce the effect of the turbulence fluctuations in the equations for the mean quantities. Solving the turbulence is still a controversial issue, since more than one hundred years of experience have shown that neither simple analytic theory, nor solution can be achieved. The main issues for a turbulent flow are the three dimensionality, the unsteadiness, and the randomness of the velocity field, as well as the large range of time and length scales. Moreover, the largest turbulent motions are usually almost as large as the characteristic width of the flow, being directly affected by the boundary conditions and therefore not universal. The first approach to solve the turbulent flow is the Direct Numerical Simulation (DNS). In this case all length scales and time scales have to be resolved. Consequently the computational resources needed are enormous. DNS computation can be considered as a numerical experiment and it is not affordable for high Reynolds configurations.

The Reynolds Averaged Navier Stokes equations (RANS)[1] takes its origin from the decomposition of the instant value of velocity and pressure into a mean and a fluctuating part. The mean flow equations are the averaged equations:

$$u_i = \overline{U_i} + u_i'$$

$$\frac{\partial \overline{U_i}}{\partial t} + \overline{U_j}\frac{\partial \overline{U_i}}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 \overline{U_i}}{\partial x_j \partial x_j} - \frac{\partial \overline{u_i' u_j'}}{\partial x_j}$$

where $\bar{U}_i$ is the mean value of $\vec{u}$ in the i direction and u_i' is its fluctuation. The Reynolds stress tensor, $\rho\overline{u_i' u_j'}$, represents the mean flux of momentum due to the turbulent fluctuations, and analytically they appear as additional unknown in the Reynolds equations and should be determined by a turbulence model (called the

[1] Remark : OpenFOAM® uses the abbreviation *RAS* for the Reynolds Averaged Navier Stokes equations.

closure problem). The closure formulation for the RANS equations can be carried out by :

- a turbulent viscosity model: according the Boussinecq's hypothesis
  $-\rho\overline{u_i' u_j'} = \epsilon_{ij}(\frac{\partial \bar{U}_i}{\partial x_j} + \frac{\partial \bar{U}_j}{\partial x_i})$ where $\epsilon_{ij} = \nu_t$
  The constant ν_t can be computed from different kind of models.
- modeled Reynolds-stress transport equations

Model testing has a large number of possible error sources, and the accuracy of one model or another is only determined by a comparison between measured and calculated flow properties only if numerical errors, measurement errors, and discrepancies in boundary conditions are small.

An intermediate approach between DNS and RANS is the Large Eddy Simulation (LES), where the time dependence is taken into account and a spatial averaging is done. The objective of the LES is to explicitly compute the largest length scales (larger than the mesh size) of the phenomena, and try to model the small ones.

The turbulence models[2] used in this chapter are described below:

- The $k-\epsilon$ *models* provide a good compromise between robustness, computational cost and accuracy. They are generally well suited to industrial-type applications that contain complex recirculation, with or without heat transfer. A $k-\epsilon$ turbulence model is a two-equation model in which transport equations are solved for the turbulent kinetic energy k and its dissipation rate ϵ. The definition of the turbulent kinetic energy is, if $u_i = \bar{U}_i + u_i'$:

  $$k = \frac{1}{2}(u'^2 + v'^2 + w'^2)$$

  The standard $k-\epsilon$ model [2] requires the solution of the following equations[3] :

  $$\frac{\partial k}{\partial t} + (\bar{U}\cdot\nabla)k - \nabla\cdot\frac{\nu_t}{C_{mu}}\nabla k = P_k - \epsilon \tag{6.3}$$

  $$\frac{\partial \epsilon}{\partial t} + (\bar{U}\cdot\nabla)\epsilon - \nabla\cdot\frac{\nu_t}{\sigma_{eps}}\nabla\epsilon = C_1\frac{\epsilon}{k}P_k - C_2\frac{\epsilon^2}{k} \tag{6.4}$$

  with C_i being model coefficients and the production rate of the turbulent kinetic energy P_k is defined as:

  $$P_k = 2\nu_t(\frac{1}{2}(\vec{\nabla}\bar{U} + \vec{\nabla}\bar{U}^T))^2$$

  In this model only a turbulent length scale is taken into account. It means that the turbulent diffusion is the one that occurs only at this specific length. Taking into account the contribution of all the turbulent length scales, thanks to the modification of the production term, leads to a modified version of the model so called RNG-$k-\epsilon$. The mathematical formulation of this model can be found in [3].

[2] The equations solved in OpenFOAM® for a particular turbulent model can be found in the function *correct* of the model. For example for the $k-\epsilon$ model for incompressible flow, the equations are found in the file `$FOAM_SRC/turbulenceModels/incompressible/RAS/kEpsilon/kEpsilon.C`.

[3] The equation are written as implemented in OpenFOAM® in particular for the name of the coefficients.

- The $k-\omega$ *models* are similar to $k-\epsilon$ models, solving also two transport equations, but differ in the choice of the second transported turbulence variable : the specific dissipation of k. The performance differences are likely to be a result of the subtle differences in the models, rather than a higher degree of complexity in the physics being captured. The $SSTk-\omega$ model combines the insensitivity to free-stream conditions of the $k-\epsilon$ model in the far-field, with retains the advantages of the $k-\omega$ model near walls. As drawback it produces a bit too large turbulence levels in regions with large normal strain, like stagnation regions and regions with strong acceleration. The formulation of the model can be found in [4].

6.2.2 Law of the wall

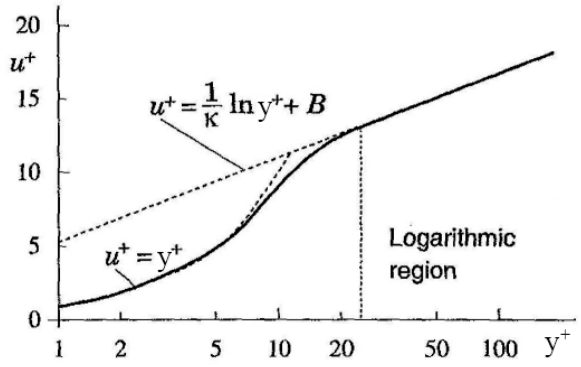


Figure 6.2: The three regions of the velocity boundary layer.

In wall attached boundary layers, the normal gradients become large as the wall distance reduces to zero, furthermore viscous effects become important. The difficult near-wall region is usually not explicitly resolved with the numerical model, but it is bridged using a so-called law of the wall[4].

The law of the wall are usually expressed using non-dimensional expressions of the velocity and the distance normal to the wall.

The non-dimensional velocity is $u^+ = \frac{u}{u_\tau}$ where u_τ, the wall friction velocity, is defined as:

$$u_\tau^2 = \frac{\tau_W}{\rho}$$

, with τ_W is the shear stress at the wall. And calling y the normal distance from the wall, the dimensionless wall distance y^+ is defined as:

$$y^+ = \frac{\rho y u_\tau}{\mu}$$

In case of flow re-attachment or point of impingement, due to the vanishing of τ_W, this definition of the dimensionless wall distance is not longer useful; therefore an alternative formulation, based on the turbulent kinetic energy, k, instead of the wall shear stress is performed :

$$y^+ = \frac{C_\mu^{0.25} k^{0.5} y}{\nu}$$

[4] The wall functions available for incompressible flow in OpenFOAM® are in the folder `$FOAM_SRC/turbulenceModels/incompressible/RAS/derivedFvPatchFields/wallFunctions`.

The boundary layer close to the wall is composed of three parts cf. Figure (6.2) :

- The viscous sublayer for $y^+ < 5$ in which $u^+ = y^+$
- The buffer sublayer for $5 < y^+ < 30$ in which no law exists
- The turbulence sublayer for $30 < y^+ < 300$ in which $u^+ = \frac{1}{\kappa}\ln(Ey^+)$

Unfortunately the wall functions do not free the user from the need to adequately resolve the turbulent portion of the boundary layer. A lower limit for y^+ ensures that the first point does not fall into the viscous sublayer, commonly the meshing should be arranged so that the values of y^+ at all the wall-adjacent integration points is between 20 and 30. Also an upper limit is present, due to the requirement of adequate boundary layer resolution, it requires at least 8-10 points in the layer. For moderate Re, y^+ should not be higher than 100.

In OpenFOAM® this can be easily checked after the simulation typing :

```
yPlusRAS
```

in the *case* directory. This function calculates the y^+ values and stores it in the solution files at each time step. N.B.: the flow has to be solved prior running that tool. So the value of y^+ can only be checked at posteriori.

Concretely, in OpenFOAM® the logarithmic law is used for $y^+ > 11.53$ with the default value of κ and E. And the linear law is used if $y^+ < 11.53$.

OpenFOAM® allows you to tune the value of the two constants in the law of the wall by given specific values in the `RASProperties` dictionary as shown in the Figure (6.4).

6.3 OpenFOAM®

To simulate an unsteady turbulent flow a good option is to use the `pisoFoam` for incompressible flows (as in this case) or the `rhoPimpleFoam` solver[5] for a compressible test. In this chapter the incompressible one is explained. It solves the equations (6.1) and (6.2) as written above inside a `PISO` loop.

6.3.1 pisoFoam

The PISO (Pressure Implicit with Splitting of Operators) algorithm is a *predictor corrector method* like the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations 4.2.6); both PISO and SIMPLE can be used for steady-state and transient problems, whereas SIMPLE was originally designed for steady-state flows and PISO for transient flows. PISO performs at every time step a "steady-state SIMPLE" algorithm in which the velocity-pressure coupling is solved.

The algorithm performs the following steps:

5 *PIMPLE* is abbreviation for a combined *PISO-SIMPLE* algorithm.

- An initial guess of the velocity is computed from the momentum equation : the *Momentum predictor* step[6].
- Enter the PISO loop
  - Estimate the velocity from the moment equation without the pressure gradient effect
  - Interpolate the mass flux on the cell faces (with a correction to insure the mass conservation)
  - Solving the pressure equation `nNonOrthCorr` times (the flux is updated at each iteration).
  - Correct the velocity (and correct the field the boundary conditions)
- The PISO loop will be solved `nCorr` times.
- Solve the turbulence model

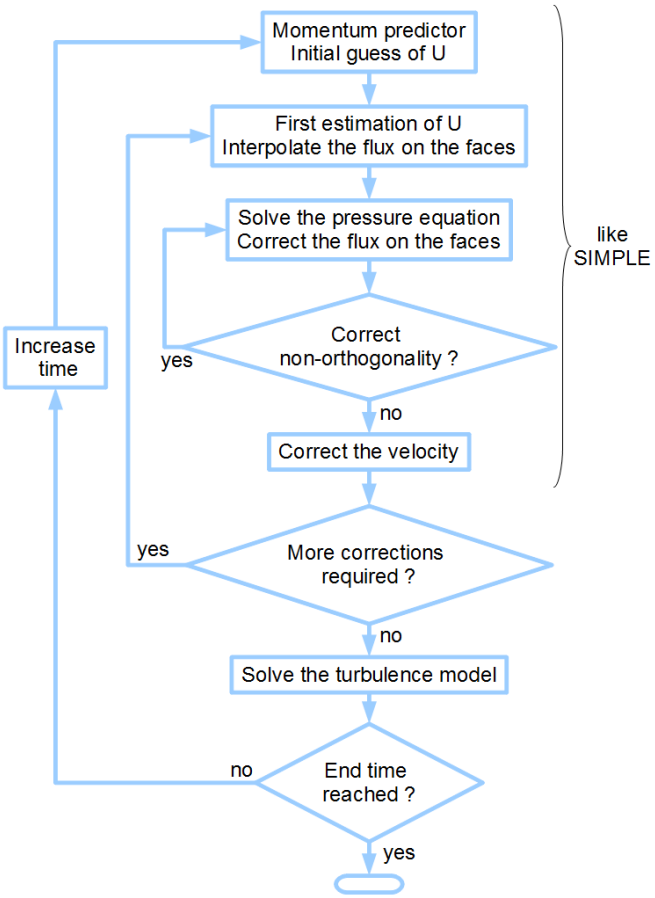


Figure 6.3: PISO algorithm

6 If the flag `momentumPredictor` in the block `PISO` of `fvSolution` is `false`, this step is not carried out. If this flag is not specified, it is set to `true`.

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

        #include "readPISOControls.H"
        #include "CourantNo.H"

        // Pressure-velocity PISO corrector
        {
            // Momentum predictor

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

            UEqn.relax();

            if (momentumPredictor)
            {
                solve(UEqn == -fvc::grad(p));
            }

            // --- PISO loop

            for (int corr=0; corr<nCorr; corr++)
            {
                volScalarField rAU(1.0/UEqn.A());

                U = rAU*UEqn.H();
                phi = (fvc::interpolate(U) & mesh.Sf())
                    + fvc::ddtPhiCorr(rAU, U, phi);

                adjustPhi(phi, U, p);

                // Non-orthogonal pressure corrector loop
                for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
                {
                    // Pressure corrector

                    fvScalarMatrix pEqn
                    (
                        fvm::laplacian(rAU, p) == fvc::div(phi)
                    );

                    pEqn.setReference(pRefCell, pRefValue);

                    if
                    (
                        corr == nCorr-1
                     && nonOrth == nNonOrthCorr
                    )
                    {
                        pEqn.solve(mesh.solver("pFinal"));
                    }
                    else
```

```
                {
                    pEqn.solve();
                }

                if (nonOrth == nNonOrthCorr)
                {
                    phi -= pEqn.flux();
                }
            }

            #include "continuityErrs.H"

            U -= rAU*fvc::grad(p);
            U.correctBoundaryConditions();
        }
    }

    turbulence->correct();

    runTime.write();

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

Listing 6.1: Time loop of `pisoFoam`

After describing the key part of the `pisoFoam` solver, the full structure of the solver will be presented.

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulenceModel.H"

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

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

    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"

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

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

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

        #include "readPISOControls.H"
        #include "CourantNo.H"

        // Pressure-velocity PISO corrector
        {
        ...
        }

        turbulence->correct();

        runTime.write();

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

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

    return 0;
}
```

Listing 6.2: `pisoFoam` source code. Transient solver for incompressible flow. Turbulence modelling is generic, i.e. laminar, RAS or LES may be selected.

After loading the standard header file `fvCFD.H`, `turbulenceModel.H` will be read. This header allows the definition in the *case directory* of the desired turbulence model. Consequently the choice of a *laminar, RAS* or *LES* model is done indepently of the solver.
Remark : all solvers don't get this flexibility. But nearly all of them are suitable to use a *RAS* or *LES* model.

In the `createFields.H` all the required fields are defined:

- `p`: a field of scalar representing the kinematic pressure p/ρ
- `U`: a vector field representing the velocity U
- `phi`: the velocity flux
- `pRefCell` and `pRefValue`: in a closed incompressible system the pressure is relative. The pressure difference and not its absolute value is important. For this reason the solver sets a reference level by `pRefValue` in the cell `pRefCell`. Changing this value become important in the compressible case.
- `laminarTransport`: is the transport model from which the laminar viscosity can be taken. In this case a single phase is transported, it means that the coexistence of a two phase (liquid-gas for example) is not allowed.
- `turbulence`: is the general reference to the turbulence model (incompressible in this case).

```
    Info<< "Reading␣field␣p\n" << endl;
    volScalarField p (...);

    Info<< "Reading␣field␣U\n" << endl;
    volVectorField U (...);

#   include "createPhi.H"

    label pRefCell = 0;
    scalar pRefValue = 0.0;
    setRefCell(p, mesh.solutionDict().subDict("PISO"), pRefCell, pRefValue);

    singlePhaseTransportModel laminarTransport(U, phi);

    autoPtr<incompressible::turbulenceModel> turbulence
    (
        incompressible::turbulenceModel::New(U, phi, laminarTransport)
    );
```

Listing 6.3: Sample of `createFields.H` for `pisoFoam`

After reading the controls for the PISO loop and computing the Courant number, the pressure-velocity PISO corrector is applied. And finally before outputting the data, the turbulence model is solved.

6.3.2 Set up a turbulent flow

The turbulence model

In the `constant` folder there is the file `turbulenceProperties` where the keyword `simulationType` is specified. The options for `simulationType` are `laminar`, `RASModel` and `LESModel`. With `RASModel` selected, the choice of RAS modelling is specified in a `RASProperties` file, also in the `constant` directory. The turbulence model is selected by the `RASModel` entry from a long list of available models, that are listed in [5]

- `RASModel`: is the name of the chosen model

- `turbulence`: it is a flag that can be `on` or `off`, i.e. if a turbulent simulation has to be done
- `printCoeffs`: it is a flag that print on the screen, at the beginning of the simulation, the coefficients of the model

The coefficients of the turbulence model are set by default in the source code of OpenFOAM®. But they can be overwritten by setting them in the `RASProperties` file, as can be seen in listing (6.4)

```
RASModel            RNGkEpsilon;
turbulence          on;
printCoeffs         on;

RNGkEpsilonCoeffs  // optional
{
    Cmu              0.0845;
    C1               1.42;
    C2               1.68;
    sigmak           1.39;
    sigmaEps         1.39;
    eta0             4.38;
    beta             0.012;
}

wallFunctionCoeffs // optional
{
    kappa            0.4187;
    E                9;
}
```

Listing 6.4: RASProperties

When you are using a turbulent model, additional fields have to be created : the turbulent kinetic energy, k, the other fields used by the model (e.g. ϵ or ω) and the turbulent kinematic viscosity, `nut`. The latter is computed from the fields used in the turbulent model. Consequently its boundary conditions have the type `calculated` except for the boundaries of type `wall`. Indeed for those boundaries, the law of the wall has to be used by using here the default law `nutkWallFunction`. An example of such boundary condition is:

```
Wall
{
    type     nutkWallFunction;
    Cmu      0.09; // [optional] default value 0.09
    kappa    0.41; // [optional] default value 0.41
    E        9.8;  // [optional] default value 9.8
    value    uniform 0;
}
```

For the turbulent fields, there are specific boundary conditions when the boundary is a `wall` :

Field	Name boundary condition	Mandatory parameters
`k`, `q` or `R`	`kqRWallFunction`	`value`
`epsilon`	`epsilonWallFunction`	`value`
`omega`	`omegaWallFunction`	`value`

The other difficult boundary condition for turbulent fields is at the inlet. Indeed, usually the profiles of k, ϵ, etc. are unknown. But the level of turbulence is specified by its intensity, I, a turbulence length scale, l or the turbulent viscosity ratio, $\frac{\mu_t}{\mu}$. The table below lists formula that could be used to set the turbulent fields.

Variable		Formula
Turbulence intensity, I	if Re_{D_H} is the Reynolds number based on the hydraulic diameter D_H	$I = \frac{u'}{u_{avg}}$ $I = 0.16(Re_{D_H})^{-1/8}$
Turbulence length scale, l	if L is the relevant dimension of the inlet D_H is a good guess	$l = 0.07L$
Turbulent kinematic viscosity, ν_t	a typical value of C_μ is 0.09	$\nu_t = C_\mu k^2/\epsilon$ $\nu_t = k/\omega$
Turbulent kinetic energy, k		$k = \frac{3}{2}(u_{avg}I)^2$
Turbulent dissipation rate, ϵ		$\epsilon = C_\mu^{0.75}\frac{k^{1.5}}{l}$ $\epsilon = C_\mu \frac{k^2}{\nu}\left(\frac{\mu_t}{\mu}\right)^{-1}$
Turbulent specific dissipation rate, ω		$\omega = \frac{k}{\nu}\left(\frac{\mu_t}{\mu}\right)^{-1}$

Remark : By definition, the hydraulic diameter, $D_H = \frac{4A}{P}$ where A is the area of the section and P is perimeter.

As you will use an unsteady turbulent solver, in the `fvSchemes` file the temporal derivative schemes and the `divSchemes` and `laplacianSchemes` concerning the turbulent quantities are required. An example of these terms can be seen in Listing 6.5.

```
ddtSchemes
{
    default                          backward;
}

divSchemes
{
    ...
    div(phi,k)                        Gauss upwind;
    div(phi,epsilon)                  Gauss upwind;
    div(R)                            Gauss linear;
    div((nuEff*dev(T(grad(U)))))     Gauss linear;
    div(nonlinearStress)              Gauss linear;
}

laplacianSchemes
{
    laplacian(nuEff,U)                Gauss linear corrected;
    laplacian(DkEff,k)                Gauss linear corrected;
    laplacian(DepsilonEff,epsilon)    Gauss linear corrected;
    laplacian(DREff,R)                Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda)    Gauss linear corrected;
}
```

Listing 6.5: fvSchemes

In `fvSolution`, the solvers for the transport equations concerning the turbulence model (for example k and ϵ or k and ω) as well as the solver for `pFinal` (the solver used to solve the pressure in the last iteration for each time step) have to be defined. For this kind of solver, the `PISO` options has to be chosen carefully. An example of all the parameters available can be seen in Listing 6.6. The user have to specify the number of correctors within a time step in the PISO dictionary by the `nCorrectors` keyword. The algorithm requires this number to be more than 1, but typically not more than 4. An additional correction to account for mesh non-orthogonality is available, the number of non-orthogonal correctors is specified by the `nNonOrthogonalCorrectors` keyword. The number of non-orthogonal correctors should correspond to the mesh for the case being solved, i.e. 0 for an orthogonal mesh and increasing number with the degree of non-orthogonality up to 20 for the most non-orthogonal meshes. If the user doesn't specify a value, the default is 0.

```
solvers
{
    ...
    pFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
    k
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    epsilon
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    R
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
    nuTilda
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
}

PISO
{
```

```
    // number of correction loop by time step (advice : 1<nCorrectors<=4)
    nCorrectors                2;
    // number of loop to correct for the non-orthogonality of the mesh
    // [optional] default : 0 (ok if cartesian mesh)
    nNonOrthogonalCorrectors   0;
    // flag to compute a first guess of the velocity from the momentum equation
    // [optional] default : true
    momentumpredictor          true;
    // not used in the pisoFoam solver
    // [optional] default : false
    transonic                  false;
    // not used in the pisoFoam solver
    // [optional] default : 1
    nOuterCcorrectors          1;
}
```

Listing 6.6: Sample of `fvSolution` for a PISO solver.

Set up the geometry and the boundary conditions

The skeleton of `blockMeshDict` is provided. The geometry is a 2D case with the following patches:

- `inlet`
- `outlet`
- `Wall`: all the horizontal walls
- `VerticalWall`: the step's wall
- `frontAndBack`

To set the `inlet`, the *setDiscreteFields* tool will used. This tool can set a scalar or vector field on patches and/or inside of the mesh from a 1D profile. You can find it on *http://openfoamwiki.net/index.php/Contrib_setDiscreteFields*. But it is also ship with the today exercise. To install it, copy the `setDiscreteFields` folder in `$WM_PROJECT_USER_DIR/applications/preprocessor`. Then execute `wmake` in the folder of the tool.

In the `system` folder there is the `setDiscreteFieldsDict`. An example of this file can be seen in the Listing 6.7, its entries are explained after this paragraph.

- `field` is the target field, i.e. the field for which the profile is specified
- `type` is the type of the target field, i.e. `scalar` or `vector`
- `direction` [optional] is the direction in which the interpolation to the mesh is done : `"x"`, `"y"`, or `"z"` (default `"x"`)
  For example, if you know the profile of the pressure in the y direction, the `direction` parameter has to be set to `y`.
- `internal` [optional] sets internal field or not : `true` or `false` (default `true`)
- `patchNames` [optional] is the the list of the patches where the field has to be set (default *empty*)
- `profile` is the list of the value to set

```
Fields
(
    keyword1
    {
        field     U;     //target field
        type      vector;//type "scalar" or "vector"
        direction "x";   //direction "x","y", or "z"
                         //in which the interpolation is done
        internal  true;  //set internal field or not, true or false
        patchNames
        (
            patch1
            patch2
        );               //patchNames to set value
        profile          //if type is vector (x y z Ux Uy Uz)
        (
            (0 0 0 1.0 0.0 0.0)
            (1 0 0 0.5 0.5 0.0)
            (2 0 0 0.0 1.0 0.0)
        );
    }

    keyword2
    {
        field     p;     //the target field is p
        type      scalar;
        direction "y";
        internal  true;
        patchNames
        (
            patch1
        );
        profile          //if type scalar (x y z p)
        (
            (0 0 0 -1.0 )
            (0 1 0 -0.5 )
            (0 2 0 0.0 )
        );
    }
);
```

Listing 6.7: setDiscreteFieldsDict

Typing `setDiscreteFields` in the *case directory* sets the fields.

For the todays problem, the inlet has to be set with a fully developed flow. So the velocity, the turbulent kinetic energy and the dissipation rate profiles at the inlet are given. You will have to use the relation described before to compute the profile at the inlet for the specific dissipation rate, ω. Then using `setDiscreteFields`, you will set the profile at the inlet.

The function objects

At the end of the `controlDict` file a block called `functions` can be specified. The elements defined in this block carried out a post-process on the case at the end of each time step. The non-exhaustive list of the `functions` is :

- `faceSource` (`cellSource`) compute a given operation on a set of faces (cells). The available operations are: none, sum, average, integrate, weighted average. And exclusively for the `faceSource`: min and max.
- `fieldAverage` calculates the field averages and variances.
- `fieldMinMax` calculates scalar minimim and maximum field values.
- `forces` calculates the forces and moments by integrating the pressure and skin-friction forces over a given list of patches.
- `forcesCoeffs` creates a specialisation to calculate lift and drag force coefficients.
- `probes` sets of locations to sample.
- `sets` sets of `sets` to sample.
- `surfaces` sets of `surfaces` to sample.
- `staticPressure` converts kinematic pressure to static pressure, from the name of the pressure field, and density, i.e. $p_{static} = density * p_{kinematic}$.
- `systemCall` executes system calls at every step, at every write step or when exiting the time-loop. The system calls have to be entered in the form of a string list.
- `writeRegisteredObject` takes over the writing of registered IO objects.

In this paragraph, four *functions* will be described : `fieldAverage`, `probe`, `writeRegisteredObject` and `sets`. But first the general definition of a *function* will be described.

```
functions
{
    name    // Name to differentiate the defined functions
    {
        type    < nameFunctions >;  // type of the function : e.g. fieldAverage
        // name of the library containing the compiled version of the function
        functionObjectLibs ("<␣libraryName␣>");
        enabled           true; // [optional] default = true

        // Control the output of the function depending on the
        // number of time steps calculated or at each write step
        outputControl     timeStep; //outputTime;
        // Needed only if the outputControl is timeStep. Frequency of output
        outputInterval    1;

        ... // The entries needed for the specified function
    }
}
```

Listing 6.8: General definition of the block `functions` in `controlDict`.

1. `fieldAverage`
   This function calculates the average (in time or in number of iteration) of a desired scalar/vector field and the variance around the mean. The parameters are :

   - `type` : `fieldAverage`
   - `functionObjectLibs` : `"libfieldFunctionObjects.so"`
   - `cleanRestart` : [optional] whether to perform a clean restart, or start from previous averaging info if available (default : `false`)
   - `resetOnOutput` : [optional] whether to reset the averaged fields after they have been written.Used to average over only the preceding write interval for transient cases (default : `false`)
   - `fields` : fields to be averaged (`runTime` modifiable). For each the following sub-entries have to be specified :
     - `mean` : it is a flag, if activated the mean value on the base is calculated
     - `prime2Mean` : it is a flag, if activated the value $(f - \bar{f})^2$ is calculated
     - `base`: it is the base on which the mean is calculated, it can be `time` or `iteration`

```
functions
{
    fieldAverage1
    {
        type                fieldAverage;
        functionObjectLibs  ( "libfieldFunctionObjects.so" );
        enabled             true;
        outputControl       outputTime;
        cleanRestart    false;
        resetOnOutput   false;
        fields
        (
            U
            {
                mean        on;
                prime2Mean  on;
                base        time;
            }
        );
    }
}
```

Listing 6.9: Example of use for the `fieldAverage` function.

2. `probe`
This function sets specific points where fields can be probed. The entries for this function are:

   - `type : probes`
   - `functionObjectLibs : "libsampling.so"`
   - `fields` : list of fields to be sampled at the probe locations
   - `probeLocations` : Locations to be probed. runTime modifiable. It is a list of the points with the x, y and z coordinates.

   Here is an example of the `probes` function.

```
functions
{
    probes
    {
        type                probes;
        functionObjectLibs  ( "libsampling.so" );
        outputControl       timeStep;
        outputInterval      10;

        probeLocations
        (
            ( x1 y1 z1 )
            ( x2 y2 z2 )
        );
        fields              ( p U );
    }
}
```

Listing 6.10: Example of use of the `probes` function.

3. `writeRegisteredObject`
This function allows to change the output parameters of fields used in the simulation (overwrites the parameters of `controlDict` for the specified fields). The entries for this function are :

   - `type : writeRegisteredObject`
   - `functionObjectLibs : "libIOFunctionObjects.so"`
   - `objectNames` : list of objects for which the output parameters have to be changed

   The listing below shows an example of use of this function.

```
functions
{
    intermediateSave
    {
        type                writeRegisteredObject;
        functionObjectLibs  ( "libIOFunctionObjects.so" );
        outputControl       timeStep;
        outputInterval      10;
        objectNames
        (
            U
        );
    }
}
```

Listing 6.11: Example of use of the function `writeRegisteredObject`.

4. `sets`
   This function is similar to run the block `sets` of the `sample`. The advantage of using it as a function is to output data for time steps that are not stored due to the output parameters of the `controlDict`. The entries of this functions are :

   - `type` : `sets`
   - `functionObjectLibs` : `"libsampling.so"`
   - `setFormat` : `xmgr`, `jplot`, `gnuplot` or `raw`
   - `interpolationSchemes` : `cell`, `cellPoint` or `cellPointFace` ([optional] default value `cell`)
   - `fields` : list of fields to be sampled
   - `sets` : list of sets to sample
     The available types are `uniform`, `face`, `midPoint`, `midPointAndFace`, `curve` and `cloud`.

   The listing below shows an example of use for this function.

```
functions
{
    setsTosample
    {
        type                sets;
        functionObjectLibs  ( "libsampling.so" );
        outputControl       timeStep;
        outputInterval      10;

        setFormat           raw;
        interpolationScheme cell;
        fields              ( p U );
        sets
        (
            line1
            {
                type        midPoint;
                axis        distance;
                start       (0.05 0.0 0.005);
                end         (0.05 0.1 0.005);
            }
        );
    }
}
```

Listing 6.12: Example of use for the function `sets`.

Exercises

6.1 Set up the case :

1. Write `blockMeshDict`[7] and generate the mesh
2. Fulfill the `controlDict`: set `endTime` to reach the steady state and `deltaT` to have a Courant number around 0.8
3. Set the boundary conditions for the velocity and the pressure (at the inlet fixed the value of the velocity to an arbitrary constant).
4. Set the different dictionaries in the `constant` folder. N.B. : `nu` has to be set to obtain $Re_H = U_c H/\nu = 5540$ where H is the height of the step and U_c, the maximal velocity at the inlet.
5. Use `setDiscreteFields` to set the velocity, `U`, the turbulent kinetic energy, `k` and its dissipation rate, `epsilon`.
6. Run the tool `setDiscreteFields`

6.2 Prepare the post-processing :

1. Calculate the average of the velocity and turbulent kinetic energy using a `function` defined in `controlDict`.
2. Write `sampleDict` to be able to compare the **average velocity** and the **average turbulent kinetic energy** of the simulation with the experimental data (files : *kasagiExpUMean.txt* and *kasagiExpURMS.txt*[8] [6]). The comparison has to be done for 3 axial positions where experimental data are available (see below for the axial positions). The experimental data are non-dimensional by using H for the lengths and U_c for the velocities.

6.3 Run the case with the three different turbulence models described in the text : $k-\epsilon$, $RNGk-\epsilon$ and $k-\omega$.

- The ω boundary conditions are not defined. To do so,
- Copy `0/epsilon` and change each `espilon` by `omega`; even in the name of the boundary conditions.
- To set the inlet boundary condition, first define it as a fixed value boundary with the value equal to zero.
- Add an entry to `setDiscreteFieldsDict` to set the profile of `omega` at the inlet. As no experimental data are provided for *omega* use the relation described previously to evaluate it.
- Set the parameters for solving `omega` in `fvSolution` as well as the numerical schemes in `fvSchemes`. *Hint*: take inspiration from the parameters used for `epsilon`.

6.4 For one model, check the position of the cells close to the wall with the tool `yPlusRAS`. This can't be done a priori because y^+ needs the value of the velocity and the turbulent variables to be computed.

6.5 How long does the simulation take to reach a steady state for k and for U? For that use the script `scriptsResiduals`.

6.6 Compare the mean velocity and turbulent kinetic energy profiles with the experimental data for 3 positions $x = 1m$, 3m and 5m. Remark: Using the URMS of the experiment you should be able to reconstruct the turbulent kinetic energy.

6.7 How does the recirculation zone change using the three different methods?

---

7 The geometry is shown in the figure (6.1)

8 URMS = U Root Mean Square = (u', v', w')

6.8 Which model achieves the best performance for this case? How could you improve the results

6.4 Extra Practice and Background Information

The vertical wall is now heated to a constant temperature of $800K$. To take into account the density variation due to the temperature, the compressible version of the solver is required. To run a compressible simulation a thermo-physical model has to be specified through the definition of a `thermophysicalProperties` file in the `constant` directory. A thermophysical model is constructed in OpenFOAM® as a pressure-temperature $p-T$ system, from which other properties are computed. There is one compulsory dictionary entry called `thermoType` which specifies the complete thermophysical model that is used in the simulation. The thermophysical modeling starts with a layer that defines the basic equation of state and then adds more layers of modeling that derive properties from the previous layers. The naming of the `thermoType` reflects these multiple layers of modeling as listed in [5] Table 7.1 . The basic thermophysical properties are specified for each species from input data. The data is specified using a compound entry with the following format for a specie accessed through the keyword mixture:
`mixture <specieCoeffs> <thermoCoeffs> <transportCoeffs>`
An example of entry for the `thermophysicalProperties` can be seen in Listing 6.13. All the possible entries are listed in [5], Chapter 7.

```
// ************************************************************************* //
thermoType    hPsiThermo<pureMixture<sutherlandTransport<specieThermo
                                                <hConstThermo<perfectGas>>>>>;

mixture
{
        specie
        {
                nMoles  1; // number of moles
                molWeight       28.9; // Molecular weight
        }
        thermodynamics
        {
                Cp              1007; // Heat capacity
                Hf              0;      // Heat formation
        }
        transport
        {       // Sutherland law coefficients
                As               1.4792e-06;
                Ts              116;
        }
// ************************************************************************* //
```

Listing 6.13: thermophysicalProperties

Mapping

The `mapFields` utility maps one or more fields related to a given geometry onto the corresponding fields for another geometry within the time directory specified by `startFrom/startTime` in the `controlDict` of the target case, reading the data from the equivalent time directory of the source case. If the geometry and the boundary conditions are identical the fields are `consistent`, and the mapping

operation is performed by typing in the target directory

```
mapFields <source dir> -consistent
```

When the fields are not consistent, a `mapFieldsDict` is required. The `mapFieldsDict` dictionary contain two lists that specify mapping of patch data (an example can be found in [5], Chapter 5):

- `patchMap` specifies mapping of data between pairs of source and target patches that are geometrically coincident;
- `cuttingPatches` contains names of target patches whose values are to be mapped from the source internal field through which the target patch cuts.

If one or both of the source and target cases are decomposed for running in parallel, additional options must be supplied when executing `mapFields`:

- `parallelSource`: if the source case is decomposed
- `parallelTarget`: if the target case is decomposed

Exercises

6.1 Use the final solution of the incompressible case as start solution for the compressible one;

6.2 Run the case with the three turbulence models;

6.3 How long did it take to reach a steady state for k and for U?

6.4 Compare the numerical solutions. Which turbulence model has the best performance?

6.5 How does the stagnation point change in presence of the heated wall?

6.6 Which model did achieve the best performance for this case?

Chapter

7 Combustion

7.1 Todays problem

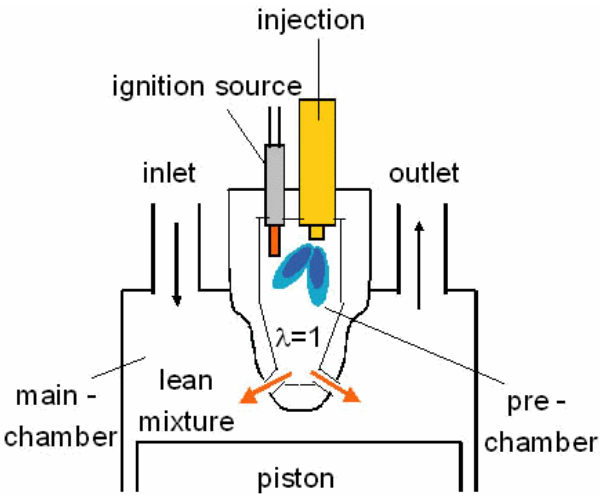


Figure 7.1: Sketch of an internal combustion engine with prechamber

Today we take a look at the processes taking place in an internal combustion engine. The engine under consideration is natural gas (largely composed of methane, CH_4) driven and used for medium-scale power production. In order to reduce NO_x emissions, the combustion chamber of such an engine is separated into a prechamber and a main chamber (see Fig. 7.1). The gas in the main chamber is very lean and thus burns with a minimum of emissions. However, to ensure a stable ignition, the mixture in the prechamber is richer. After ignition in the prechamber, flame streaks will shoot into the main chamber and ignite the gas there.

We want to simulate the combustion process inside prechamber and main chamber. In order to handle this problem with a minimum of numerical capacities, we will have to make several simplifications:

- We do the simulation in 2D, not 3D
- on a relatively coarse grid.
- We use a very simple combustion model (the "Schmid Model").
- We also ignore the different stoichiometry in prechamber and main chamber.
- We assume the geometry to be fixed, i.e. the piston is not moving.

All these simplifications are done because we want to focus on the usage of OpenFOAM®, not on the exact description of the physics happening in such a complex process.

Bibliography

[1] http://foam.sourceforge.net/doc/Guides-a4/UserGuide.pdf
OpenFOAM® Users Guide, Version 1.6, 24th July 2009
[2] http://foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf
OpenFOAM® Programmers Guide, Version 1.6, 24th July 2009

[3] S.R. Turns: *An Introduction to Combustion*, McGraw-Hill, 2nd edition, 2000
[4] A. Eder et al.: Investigation of the Transient Flame Development Using a Combination of Advanced Optical Measurement Techniques, *8th International Symposium on Flow Visualisation*, 1998
[5] H.-P. Schmid et al.: A Model for Calculating Heat Release in Premixed Turbulent Flames, *Combustion and Flame 113*, 1998

7.2 Physics

7.2.1 Combustion fundamentals

So far, the flow problems we solved were all dealing with inert flows. Now, we have to deal with a flow whose composition changes during the run, i.e. not only the fluid properties change but also chemical energy is converted to heat.

Consider the energy equation

$$\rho\frac{\partial h}{\partial t} + \rho\vec{v}\cdot\nabla h + \nabla\vec{q} = \frac{Dp}{Dt} \tag{7.1}$$

The specific enthalpy h of one species can be written as the sum of the enthalpy of formation h_f and the sensible enthalpy h_s. In perfect gases, enthalpy does not depend on pressure. h_f represents the enthalpy that is stored in the chemical bonds at some reference state (e.g. a temperature of $298\,\mathrm{K}$), h_s is the enthalpy due to the deviation from the reference state (e.g. due to a higher temperature):

$$h(T) = h_f + h_s(T) = h_f + \int_{T_{ref}}^{T} c_p(T')dT' \tag{7.2}$$

The specific enthalpy h of a gas mixture consisting of N different species is defined as

$$h(T) = \sum_{i=1}^{N} y_i \left(h_{f,i} + \int_{T_{ref}}^{T} c_{p,i}(T')dT' \right) \tag{7.3}$$

where y_i denotes the mass fraction of the respective species:

$$y_i = \frac{\text{mass of species i in the mixture}}{\text{mass of the mixture}} \quad \left[\frac{\mathrm{kg}}{\mathrm{kg}}\right] \tag{7.4}$$

The mass fractions of all species must always sum up to unity:

$$\sum_{i=1}^{N} y_i = 1 \tag{7.5}$$

The absolute value of the enthalpy of formation does not matter. What is important is the relative change in a chemical reaction. Consider for example the simple reaction

$$2\mathrm{H}_2 + \mathrm{O}_2 \rightarrow 2\mathrm{H}_2\mathrm{O} \tag{7.6}$$

For convenience, the enthalpy of formation of H_2 and O_2 can be set to zero. Then, the enthalpy of formation of $\mathrm{H}_2\mathrm{O}$ can be quantified to $-241\,845\,\mathrm{kJ/kmol}$. If we

assume that the reaction takes place under isobaric conditions in an adiabatic reactor (H=constant), we can compute the temperature after the reaction. We use molar quantities (indicated by a the subscript m) instead of specific quantities, so that we can derive the enthalpy balance directly from Eqn. 7.6. For simplicity, let us assume that the initial temperature equals the reference temperature and that $c_{p,m,H_2O} = 50\,\mathrm{kJ/kmolK}$=constant. This renders the enthalpy of the reactants to exactly zero. From the condition H=constant we get:

$$H_{\text{react}} = H_{\text{prod}} \tag{7.7}$$

$$1 \cdot h_{m,H_2} + \frac{1}{2} \cdot h_{m,O_2} = 1 \cdot h_{m,H_2O} \tag{7.8}$$

$$0 + 0 = h_{f,m,H_2O} + c_{p,m,H_2O} \cdot (T - T_{ref}) \tag{7.9}$$

$$T = T_{ref} - \frac{h_{f,m,H_2O}}{c_{p,m,H_2O}} = 298\,\mathrm{K} - \frac{-241\,845\,\mathrm{kJ/kmol}}{50\,\mathrm{kJ/kmol - K}} = 5135\,\mathrm{K} \tag{7.10}$$

We see that the change of the chemical composition of the mixture leads to a huge temperature rise. Of course, the temperature changes we usually encounter in technical systems are much lower because there is a lot of non-reacting gas to be heated as well (e.g. nitrogen or excess oxygen), heat losses to the surrounding have to be taken into account and the heat capacity is very sensible to the temperature for non-diatomic molecules.

Another convenient way to express the heat generated in a combustion process is the heating value h_u. From Eq. 7.6 and the knowledge of all enthalpies of formation we can conclude that the conversion of $1\,\mathrm{kmol}$ of hydrogen releases a chemical energy of $241\,845\,\mathrm{kJ}$:

$$h_{u,m,H2} = 241\,845\,\mathrm{kJ/kmol} \quad \text{or, based on mass} \tag{7.11}$$

$$h_{u,H2} = \frac{h_{u,m,H2}}{M_{H2}} = \frac{241\,845\,\mathrm{kJ/kmol}}{2{,}016\,\mathrm{kg/kmol}} = 119\,963\,\mathrm{kJ/kg} \tag{7.12}$$

However, the open question is: how fast does a gas burn? Actually, the combustion of hydrogen does not work the way described in Eqn. 7.6, but a variety of different elementary reactions occur and produce a chain reaction. We do not look at the elementary reactions here, but we know from experience that if the gas (fuel and oxidant) is perfectly premixed, this results in a constant burning velocity in laminar flow: the laminar flame speed s_L. Typical flame speeds for stoichiometric mixtures are listed in Table 7.1.

Methane	CH_4	$0{,}40\,\mathrm{m/s}$
Acetylene	C_2H_2	$1{,}36\,\mathrm{m/s}$
Hydrogen	H_2	$2{,}10\,\mathrm{m/s}$

Table 7.1: Laminar flame speeds in air; stoichiometric mixtures at standard pressure; taken from Turns[3]

7.2.2 Turbulent Combustion

Actually, the flame velocity observed in technical systems is usually much higher. The reason is the turbulence. In turbulent flow, the heat and mass exchange over a flame front can be many times higher than in laminar flow. Imagine a laminar flame front as a very thin, but finite layer, moving at a certain velocity and turning fuel+oxidizer into products. If this flame "sheet" is subject to a moderately turbulent flow field, the turbulent flow will wrinkle the flame and thereby increase its surface. If we assume, that in every point on the flame surface, the flame proceeds at laminar flame speed into the direction perpendicular to the flame surface, it is obvious that the net fuel consumption rate will increase, the more the surface increases. There is a direct dependence between turbulent intensity (quantified by the turbulent velocity fluctuation u') and turbulent flame speed s_T which is often modeled by the following approach:

$$s_T = s_L \cdot (1 + f(u')) \tag{7.13}$$

If turbulence continues to increase, the flame surface begins to break apart so that distributed reaction zones appear. Finally, in highly turbulent flows the flame can no longer be described as a surface, but rather as a perfectly stirred reactor, as turbulent eddies quickly level out all inhomogeneities in the flow.

A good way of classifying the nature of premixed-flames encountered in turbulent flow is the so-called *Borghi diagram*, depicted in Fig. 7.2. Based on the turbulent quantities u' and L and the chemical quantities s_L and δ_L the flame regime can be found in the diagram.

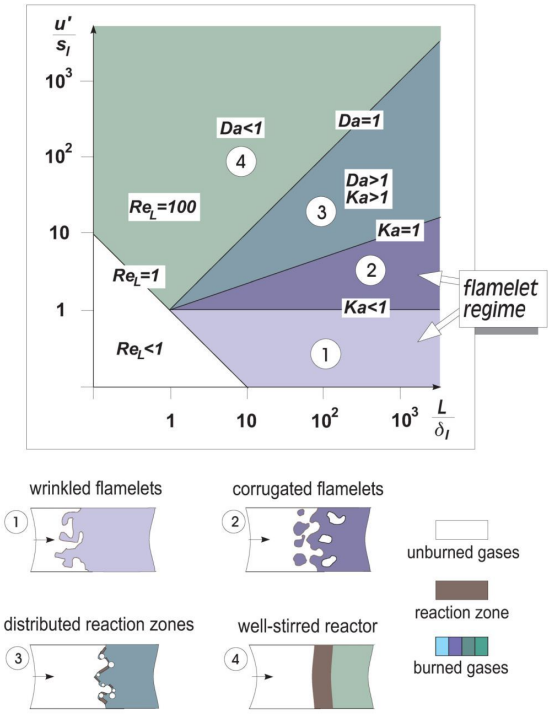


Figure 7.2: Borghi Diagram [4]

7.3 Numerics

Resolving the turbulent scales that are present in technical systems is out of question (and still will be for several decades, if computing power continues to increase at the pace it did in the past). Therefore, we have to do simulation with a RANS approach, using simple two-equations turbulence models like the $k-\varepsilon$ or $k-\omega-$SST model.

In order to model the interaction between turbulent flow and laminar chemistry, there are a nearly infinite number of combustion models, varying from very simple to very sophisticated models. Many of them are adapted to a specific flame regime (e.g. flamelets, well-stirred flames) or even to specific applications. A very simple turbulent combustion model was presented by Schmid et al. in 1998 [5] ("Schmid model"). The authors claim it to be valid for nearly the whole Borghi diagram.

In order to model a reacting flow (like a burner or an engine), in addition to the Reynolds-Averaged Navier-Stokes equations, an equation has to be solved for the mass fraction conservation of each species needed by the chemical reactions. If Y_k is the mass fraction of the material k, the transport equation is:

$$\frac{\partial(\rho Y_k)}{\partial t} + \nabla(\rho \vec{u} Y_k) + \nabla\left(\alpha_{eff} \nabla \rho Y_k\right) = \dot{\omega}_k$$

where $\dot{\omega}_k$ is a source term depending on the chemical reaction. For example, to solved the reaction (7.6), three additional equations have to be solved for Y_{H2}, Y_{O2} and Y_{H2O}.

The Schmid model has an interesting hypothesis: the advancement of the reaction (7.6) can be evaluated using a so-called reaction progress variable c with $0 \leq c \leq 1$, where $c = 1$ means completely burnt, $c = 0$ means completely unburnt gas. Consequently the temperature and the composition of the flow can then be derived from the local value of c. And only one more transport equation has to be solved: a transport equation for a reaction progress variable.

The link between the reaction progress variable, the composition and the temperature will be now explain. For premixed flames, the flow is composed of two mixtures : the reactants (e.g. H_2 and O_2) and the products (e.g. H_2O). The progress variable can be seen has the state of the real mixture between those two known states. Furthermore, remembering that $\sum_k Y_k = 1$:

$$Y_{reactants} = 1 - c \tag{7.14}$$

$$Y_{products} = c \tag{7.15}$$

Consequently all the thermodynamical properties, $x_{mixture}$ can be computed from the values of those properties for the reactants and the products:

$$x_{mixture} = \frac{(1-c)}{W_{reactant}} x_{reactants} + \frac{c}{W_{products}} x_{products} \tag{7.16}$$

where W_i is the molecular weight of the mixture i.

The change in temperature is a bit more subtle. Indeed the energy equation solved for reacting flow is the total enthalpy conservation; equation (7.1). In this equation,

there is no source term. So the increase of temperature is not due to that equation but due to the change in composition. If we consider the hydrogen combustion, the hydrogen and the oxygen burn to produce steam water that has a heat formation lower. Consequently as the total enthalpy is constant, the decrease in heat formation is compensated by the increase in sensible enthalpy, h_s. And so an growth in temperature; equation (7.2).

We end up with the following equations to solve for a compressible flow:

$$\text{mass:} \quad \frac{\partial \rho}{\partial t} + \nabla(\rho \cdot \vec{u}) = 0 \tag{7.17}$$

$$\text{momentum:} \quad \frac{\partial(\rho\vec{v})}{\partial t} + \nabla(\rho\vec{u}\vec{v}) - \nabla\left(\mu_{\text{eff}}\nabla\vec{v}\right) = -\nabla p \tag{7.18}$$

$$\text{energy (total enthalpy):} \quad \frac{\partial(\rho h)}{\partial t} + \nabla(\rho\vec{u}h) - \nabla\left(\alpha_{\text{eff}}\nabla T\right) = \frac{Dp}{Dt} \tag{7.19}$$

$$\text{equation of state:} \quad p = \rho r T \tag{7.20}$$

$$\text{reaction progress:} \quad \frac{\partial(\rho c)}{\partial t} + \nabla(\rho\vec{u}c) - \nabla\left(\alpha_{\text{eff}}\nabla c\right) = \dot{\omega}_c \tag{7.21}$$

$$\text{turbulence:} \quad \textit{e.g. k-}\varepsilon\textit{-model} \quad \textit{(2 equations)} \tag{7.22}$$

Schmid et al. formulate the source term for the reaction progress, $\dot{\omega}_c$, with the following relation:

$$\dot{\omega}_c = 4{,}96 \cdot \frac{\varepsilon}{k} \cdot \left(\frac{s_L}{\sqrt{2/3k}} + (1 + \mathrm{Da}^{-2})^{-1/4} \right)^2 \cdot c \cdot (1 - c) \cdot \rho_0 \cdot y_{fuel} \tag{7.23}$$

ρ_0 is the density of the unburnt gas and y_{fuel} is the mass fraction of fuel in the gas mixture. The dimensionless number Da (Damköhler number) indicates the combustion regime. It is defined as the ratio of the integral turbulent time scale t_T and the chemical time scale t_C. t_T depends only on the flow and t_C only on the gas mixture:

$$\mathrm{Da} = t_T / t_C \tag{7.24}$$

$$t_T = k/\varepsilon \tag{7.25}$$

$$t_C = \nu_0 / s_L^2 \tag{7.26}$$

A high Damköhler number corresponds to low turbulence, e.g. a mildly wrinkled flame, and a low turbulent flame speed. A low Damköhler number means high turbulence, consequently a high burning velocity. The Damköhler number can also be found in the Borghi diagram in Fig. 7.2. Eqn. 7.23 blends smoothly between high and low values of the source term $\dot{\omega}_C$ (that means between a high and low turbulent flame speed).

As mentioned previously, the heat release is due to the water formation. And in this case it can be explicitly computed thanks to:

$$\dot{q}_C = \dot{\omega}_C \cdot y_{fuel} \cdot h_u \tag{7.27}$$

The value of the heat release is interesting because it's a good way to visualize where is the flame (= where the heat is released).

7.4 OpenFOAM®

7.4.1 Preparing the solver files

We would like to design a solver to solve the problem described in section 7.1 using the Schmid model. It would be very laborious to start programming a solver from scratch. So we better look for a solver which comes very close to our desired solver so that we have to implement as few additional code as possible. The OpenFOAM® solver *rhoPimpleFoam* can serve as such a starting point. It solves Eqns. 7.17-7.19 and the turbulence equations for a compressible flow.

Before you start working on your own solver, you should have a thorough look at the source code of *rhoPimpleFoam*, i.e. at what it does. For example, locate the lines in the code where the equations just mentioned are solved. You might also want to run the tutorial located in
`$FOAM_TUTORIALS/compressible/rhoPimpleFoam/ras/cavity`
(*copy it to your local directory before you run it*).

To get started, copy the *rhoPimpleFoam* solver files to your local solver directory. Then rename the folder (e.g. to *SchmidFoam*) and the *C++* file (e.g. to *SchmidFoam.C*). Then change the `Make/files` file

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

Now you can compile the solver for the first time - it should work without problems, as the code was not changed from *rhoPimpleFoam*
You can now also run the *cavity* tutorial mentioned above with *SchmidFoam* which should make no difference to running it with *rhoPimpleFoam*.

7.4.2 Exploring the case setup

Property files

Before we start working on the solver, let's explore some of the files given in today's `engine` case (we recommend to work on a copy of the original files).

- `constant/turbulenceProperties: RASModel` is selected.
- `constant/RASProperties: kEpsilon`
- `constant/thermophysicalProperties:`
  `thermoType`
  `hPsiMixtureThermo<homogeneousMixture<sutherlandTransport`
  `<specieThermo<janafThermo<perfectGas>>>>>;`
  Let's analyze this starting from right:
  - `perfectGas`: The perfect gas equation $p = \rho r T$ is used as equation of state
  - `janafThermo`: c_p is changing with the temperature according a polynomial law[1].

[1] Two sets of coefficients are provided: one for the lower range of temperature and one for the high temperature.

- `specieThermo`: A class template that provides function for c_p, h_s etc.
- `sutherlandTransport`: The molecular transport properties (k, μ, D) are assumed are function of the temperature.
- `homogeneousMixture`: A class template if you want to derive material properties via a mixing law (7.16) from two species (`reactants` and `products`).
- `hPsiMixtureThermo`: T is derived from h (not the other way round) and the mixture is pressure based (ρ is computed from p and `psi`$= p/\rho = 1/(rT)$)

Hint: A easy way to check which properties are available you can misspell the thermoType and start the solver. OpenFOAM® will come up with an error message and a list of all possible settings.

```
products
{
    specie
    {
        nMol            1;
        molWeight       28.2176;
    }
    thermodynamics
    {
        Tlow            200.0;
        Thigh           5000.0;
        Tcommon         1000.0;
        highCpCoeffs    ( 3.0547E+00 1.6364E-03  -5.8903E-07 1.0131E-10
-6.6422E-15 -6.1931E+03 5.4434E+00 );
        lowCpCoeffs     ( 3.2481E+00 2.005E-03  -4.1765E-06 5.3968E-09
-2.2766E-12 -6.2449E+03 4.3344E+00 );
    }
    transport
    {
        As              1.6721e-6;
        Ts              170.672;
    }
}
```

We use a mixture that has two components. The typical structure for a material will be described using `products` as example. The first sub-dictionary specifies species parameters: `nMol`, the number of moles and `molWeight`, the molecular weight. Then come the thermodynamics properties. If the Janaf model is chosen, five entries must be given:

- `Tlow`: the low bound of the temperature
- `Thigh`: the high bound of the temperature
- `Tcommon`: the temperature threshold to switch between the two sets of coefficients
- `highCpCoeffs`: the 7 polynom coefficients for the high temperature range
- `lowCpCoeffs`: the 7 polynom coefficients for the low temperature range

Finally the parameters of the transport model are written. A_s and T_s, are used by the Sutherland law:

$$\mu = A_s \frac{\sqrt{T}}{1 + T_s/T}$$

Mesh definition

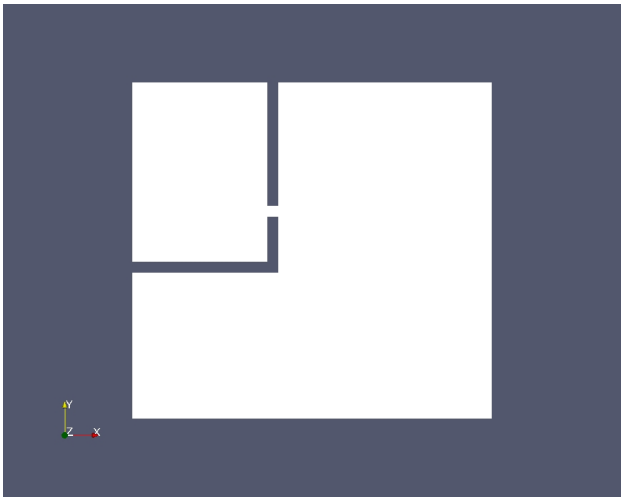


Figure 7.3: The geometry in paraFoam

In order to save computational time, we want to set up the 3D combustion problem in the engine as a 2D case with rotational symmetry. This means that we cannot model the small holes in the wall of the prechamber (see Fig.7.1). Instead, we define a slit running around the whole circumference of the prechamber. We define the geometry as a wedge with an opening angle of approximately 5 degrees. If we set it up in `blockMeshDict` as a block structured mesh we would have to define a large number of vertices and blocks. Take a look at the 2D view of the wedge geometry in Fig. 7.3 and try to draw the blocks in it - you should end up with at least 9 blocks. Fortunately, there is a much more convenient way to set up such a mesh in OpenFOAM® without using a graphical meshing tool. We set up the full wedge ignoring the walls of the prechamber and deactivate the cells located at the prechamber walls afterwards. In this case we have to define only 6 vertices and 1 block. (The definition of a wedge is also described in the OpenFOAM® User Guide [1], section 5.3.3.) The first entries in `constant/polyMesh/blockMeshDict` should be in the following way:

```
convertToMeters 0.001;

vertices
(
    (0  -30  0  )
    (32 -30 -1.5)
    (32   0 -1.5)
    (0    0  0  )
    (32 -30  1.5)
    (32   0  1.5)
);

blocks
(
    hex (0 1 2 3 0 4 5 3)     (64 60 1) simpleGrading (1 1 1)
);
```

```
edges
(
);
```

This results in a mesh grading of 0.5mm. The boundary patches should be defined the following way:

```
boundary
(
    wand
    {
        type    wall;
        faces
        (
            (1 2 5 4)
            (0 1 4 0)
            (3 5 2 3)
        );
    }

    wedge1
    {
        type    wedge;
        faces
        (
            (0 4 5 3)
        );
    }

    wedge2
    {
        type    wedge;
        faces
        (
            (1 0 3 2)
        )
    }
);
```

You can run `blockMesh` and get a wedge mesh. In order to cut out the cells that form the prechamber walls, we use the `setSet` and the `subSetMesh` utilities. In the file `prechamber.setSet` the respective cells are defined:

```
# an empty cellSet named "newWall" is defined
cellSet newWall new
# it is inverted. So now it contains all cells of the mesh
cellSet newWall invert

# the cells within the following boxes are deleted form the set:
cellSet newWall delete boxToCell (0.000 -0.017 -1) (0.013  -0.016 1)
cellSet newWall delete boxToCell (0.012 -0.017 -1) (0.013  -0.012 1)
cellSet newWall delete boxToCell (0.012 -0.011 -1) (0.013   0.000 1)
```

N.B. : A box is defined by the two corners of one box diagonal.

We can run `setSet` to create a set of cells that contains all cells but the ones that form the prechamber walls:

```
setSet -batch prechamber.setSet
```

Here the actions to be done are stored in a file. Consequently you have to use the option `batch` to specify the file. Another possibility is to type only `setSet` to enter in console mode.

We then reduce the mesh to this cell set by using the `subSetMesh` utility. All cell faces that are now turned into an exterior face (because they are now touching the prechamber walls) are added to the patch named `wand`.

```
subsetMesh newWall -patch wand -overwrite
```

Start *paraFoam*, choose *Display* → *Style/Representation* → *Surface With Edges*. Your mesh should look like in Fig. 7.4.

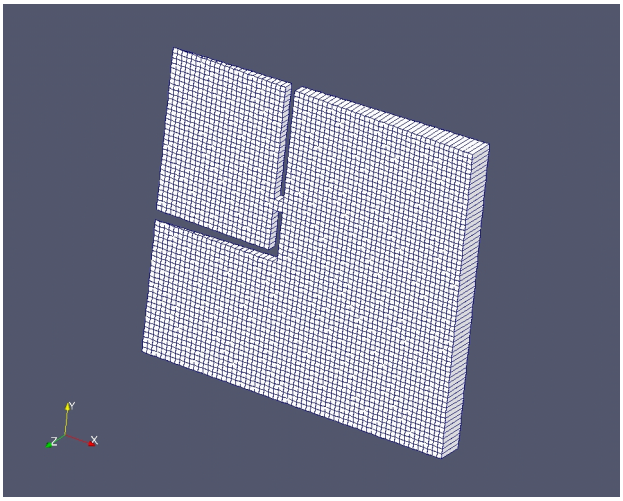


Figure 7.4: The mesh in paraFoam

7.4.3 Implementing the Schmid Model

Incorporating changes in the existing solver structure

Now that the geometry is set, the solver can be created. If we look at the solver in its current state we have to make 4 modifications:

1. We have to include the transport equation for reaction progress.
   That part is a little bit tricky in OpenFOAM® because it isn't the progress variable that have to be used but the regression variable, $b = 1 - c$
2. Therefore, we need to calculate the source term $\dot{\omega}_C$ in every time step (and we will computed also $\dot{q}_c$)
3. Therefore, we need to know some more fluid properties: h_u, s_L and y_{fuel} of the gas mixture (constant in the whole domain)
4. We have to create fields for new quantities that are not constant in the whole domain (b, $\dot{\omega}_C$, $\dot{q}_C$, Da)

In order to keep the solver well-readable, we do this by putting the additional code for the first 3 items into header files that are called in `SchmidFoam.C` (the 4^{th} task is done by adding some code to `createFields.H`). Let's copy in our `SchmidFoam` solver directory the 3 files: `readChemistryProperties.H`, `burn.H` and `bEqn.H`. Include them in `SchmidFoam.C` at the correct position (indicated by comments). And change the thermodynamical class from `basicPsiThermo.H` to `hCombustionThermo.H` because now reactions take place inside the flow:

```
#include "fvCFD.H"
//#include "basicPsiThermo.H"    // Comment this line
#include "hCombustionThermo.H"   // Add this line
#include "turbulenceModel.H"
#include "bound.H"
#include "pimpleControl.H"

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

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    // read values for hu, sL and yfuel
    #include "readChemistryProperties.H"    // HERE
    #include "createFields.H"
    #include "initContinuityErrs.H"

    pimpleControl pimple(mesh);

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

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

        runTime++;
```

```
        Info<< "Time = " << runTime.timeName() << nl << endl;
// compute source terms before solving the transport equations
        #include "burn.H"                    // HERE
        #include "rhoEqn.H"

        // --- Pressure-velocity PIMPLE corrector loop
        for (pimple.start(); pimple.loop(); pimple++)
        {
            if (pimple.nOuterCorr() != 1)
            {
                p.storePrevIter();
                rho.storePrevIter();
            }
// solve first the reaction progress equation
            #include "bEqn.H"                // HERE
            #include "UEqn.H"
            #include "hEqn.H"

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

            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }

        runTime.write();

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

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

    return 0;
```

Listing 7.1: fig:schmidFoamC

As a new class is used for the thermodynamics, new entries has to be specified in `Make/options` to let the compiler know where are the header file and the library.
So you need to add in the first section after the entry for `thermophysicalModels/basic`:

```
-I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \
-I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
```

And in the second section after `-lbasicThermophysicalModels`:

```
-lreactionThermophysicalModels \
```

We will start doing the 4 modifications mentioned above by defining the `readChemistryProperties.H` file.
Remark : each time there is a series of *?*, that implies that an element is missing.

readChemistryProperties.H: Reading the combustion properties from a dictionary

Instead of fixing the fluid properties h_u, s_L and y_{fuel} at solver level, we read them from a dictionary called `chemistryProperties` in the `constant` folder. This allows the user to modify them for any run without having to recompile the solver.

The skeleton `readChemistryProperties.H` is :

```
Info << "????????????" << nl << endl; // Change it

IOdictionary chemistryProperties
(
  IOobject
  (
    "????", // name of the dictionary
    runTime.constant(), // folder where to find the file
    mesh,
    ????,  // reading option : IOobject::MUST_READ or
           // IOobject::READ_IF_PRESENT or IOobject::NO_READ
    ????,  // writing option : IOobject::NO_WRITE or
           // IOobject::AUTO_WRITE
    false
  )
);

// fuel mass fraction, yFuel :
????? // constant scalar with dimension in chemistryProperties
// heating value, hu :
????? // constant scalar with dimension in chemistryProperties
// laminar burning velocity, sL :
????? // constant scalar with dimension in chemistryProperties
// convenient for switching off the reaction for debugging purposes
const Switch combustion(chemistryProperties.lookup("combustion"));

// Maximal heat released if all the fuel burnt
dimensionedScalar q=yFuel*hu;
// If the combustion is turn off the
if(! combustion) {q *= 0.0;}
```

The solver will look for a file *chemistryProperties* where the quantities can be read from. The heat added to the gas if it burns completely is not h_u but $q = y_{fuel} \cdot h_u$ as only the mass fraction y_{fuel} releases heat. We also add a switch with the name "`combustion`". Using this switch, the user can decide at run time if he wants to do a simulation with or without combustion. A switch in OpenFOAM® is comparable to a boolean variable and can have the values `on` or `off` where `on` corresponds to the boolean value `true`.

Go back to our case and check that there is a file called `constant/chemistryProperties` where the three parameters and the *combustion* switch are defined. The values there are for a lean methane/air mixture. The elevated laminar burning velocity (compared to table 7.1) is due to a higher initial pressure. Make sure that the combustion switch is turned on.

Next, we will create the additional fields.

createFields.H: Creating the required fields

Go back to the solver directory. We need fields for b, $\dot{\omega}_C$, $\dot{q}_C$ and Da - the quantities that are not constant inside the domain. b and Da are dimensionless, $\dot{\omega}_C$ has the unit $\mathrm{kg/m^3s}$, $\dot{q}_C$ has the unit $\mathrm{W/m^3}$.

But first the class to manage the thermodynamic has to be changed. Update the begin of `createFields.H` like that:

```
    /* Old code
    autoPtr<basicPsiThermo> pThermo
    (
        basicPsiThermo::New(mesh)
    );
    basicPsiThermo& thermo = pThermo();
    */

    autoPtr<hCombustionThermo> pThermo
    (
        hCombustionThermo::New(mesh)
    );
    hCombustionThermo& thermo = pThermo();
```

We add the following lines at the end of `createFields.H`:

```
// Get the regression variable field from the thermodynamics object
volScalarField& b = thermo.composition().Y("b");

volScalarField omegaC
(
  IOobject
  (
    ????? // This field isn't read and has to be written with the other fields.
  ),
  mesh,
// set the initial value and the dimension of the field:
  dimensionedScalar("zero", dimMass/dimVolume/dimTime, 0.0)
);
omegaC.write();

volScalarField dQ
(
  IOobject
  (
    ??????  // This field isn't read and has to be written with the other fields.
  ),
  mesh,
  dimensionedScalar("zero", dimEnergy/dimVolume/dimTime, 0.0)
);
dQ.write();

volScalarField Da
(
  IOobject
  (
    ??????  // This field isn't read and has to be written with the other fields.
  ),
  mesh,
```

```
  dimensionedScalar("zero", dimless, 0.0)
);
Da.write();
```

The `write()` command is used to save the values at the beginning of the run. They are still zero as they have not been computed yet. However, saving them is convenient for postprocessing routines - otherwise the fields will not be present in the `0` directory. Compile the solver to check for errors.

burn.H: Computing the source terms in each timestep

In each timestep we have to recompute the values for $\dot{\omega}_C$ and $\dot{q}_C$. This is done in the newly created file `burn.H`. In order to calculate $\dot{\omega}_C$ according to Eqn. 7.23 we first compute the Damköhler number according to Eqns. 7.24 to 7.26. Therefore, we need to know two more quantities: density ρ_0 and kinematic viscosity ν_0 of the unburnt mixture. They need to be computed only once, at the beginning of the run. Therefore, we can add the necessary lines (for example) at the end of `createFields.H`:

```
    dimensionedScalar rho0 = max(rho);
    dimensionedScalar nu0  = min(thermo.mu())/rho0;
```

The maximum and minimum values are used in case that the ignition area in the domain is patched with a higher temperature than the unburnt mixture.

With these values, we can compute the Damköhler number. For $\dot{\omega}_C$ we still need the turbulent quantities k and ε. They cannot be accessed directly, as they are private objects of the turbulence class. However, this class offers public member functions that return the respective fields. `turbulence` is defined as a pointer in `createFields.H`:

```
     autoPtr<compressible::turbulenceModel> turbulence
```

So we get k and ε by invoking the respective member function using the dereference operator `->` on the pointer. The file `burn.H` should get the following entries:

```
if(combustion)
{
  Da = ??????;
  omegaC = 4.96*turbulence->epsilon()/turbulence->k()
        *pow(sL/pow(2.0/3.0*turbulence->k(),0.5)
             +pow(1.0+pow(Da,-2),-0.25),2)
        *b*(1.0-b)*rho0*yFuel;
}
else
{
  omegaC *= 0.0;
}

dQ = ?????;
```

You have to specified the expression of the heat release `dQ` and Da:

$$\mathrm{Da} = \frac{s_L^2}{\nu_0}\frac{k}{\epsilon}$$

The combustion switch defined in `readChemistryProperties.H` can be used here: If the user sets it to "`false`", the source term will always be zero.

Now that we know $\dot{\omega}_C$, we still have to include the regression variable transport equation into the solver.

bEqn.H: Regression variable transport equation

We can use the enthalpy transport equation as a template for the reaction progress transport equation:

```
cp hEqn.H bEqn.H
```

Compare Eqns. 7.19 and 7.21 to make the appropriate changes to `bEqn.H`:

```
{
    solve
    (
      fvm::ddt(rho, b)
    + fvm::div(phi, b)
    - fvm::laplacian(turbulence->alphaEff(), b)
    ==
    ???????
    );
    b.max(0.0);
    b.min(1.0);
}
```

The latter two lines limit b to stay between 0 and 1. The `min` and `max` function might be a bit confusing. `b.max(0.0)` actually means: go through all values of b, and replace them with the maximum value of b and 0.0.

Now our solver is complete and we can compile it.

7.4.4 Running the case

You can go to the `engine` directory and run the case using the `SchmidFoam` solver. Before, we still need to model the ignition of the gas by patching a part of the domain with a value $b < 1$. For that the tool `setFields` will be used. It requires a dictionary called `system/setFieldsDict` that is provided. In this case the dictionary is presented in the listing below.

```
defaultFieldValues
(
    volScalarFieldValue b 1.0
);

regions
(
    boxToCell
    {
        box (0.0 -0.006 -1) (0.004 0.0 1);

        fieldValues
        (
```

```
                volScalarFieldValue b   0.99
        );
    }
);
```

Listing 7.2: Extract of `setFieldsDict`.

The `defaultFieldValues` sets the default value of the fields, i.e. the value the field takes unless specified otherwise in the `regions` sub-dictionary. That sub-dictionary contains a list of subdictionaries containing `fieldValues` that override the defaults in a specified region. The region is expressed in terms of a `topoSetSource` that creates a set of points, cells or faces based on some topological constraint. Here, `boxToCell` creates a bounding box within a vector minimum and maximum to define the set of cells in which the combustion has started. The regression variable b is defined as 0.99 in this region. The user should execute `setFields` as any other utility is executed.

Patch the field and then run the solver by typing

```
setFields
SchmidFoam
```

The computation will take several minutes. When it is finished, have a look at the case in `paraFoam`. Visualize the reaction progress (use the play button to see how it develops over time). You will realize that the reaction progress proceeds in a very unrealistic way. The gas burns fastest along the wall and at some spots starts to burn even before it is reached by the actual flame (cf. Fig. 7.5).

This is a known problem of the Schmid model. It overpredicts the reaction next to a wall. Have a look at Eq. 7.23. Next to a wall, k is necessarily very low while ε is very high. This leads to a very high reaction rate. We will try to remedy this problem in the Extra Practice section.

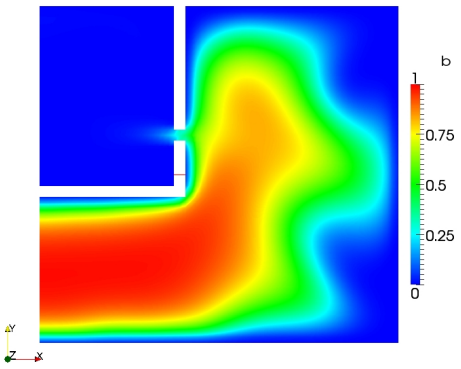


Figure 7.5: Reaction progress in the engine after 0.1 s.

7.5 Extra Practice and Background Information

Extra Practice

Reducing the source term next to walls

Hint: For frequent rerunning of the case as it will be required in the Extra Practice section there is a script called `Allrun` *in the case directory. You can use this script to redo all the things required to start the case again from the beginning.*

As noted above, we will try to find a way to remove the over-prediction of the reaction rate next to the engine walls. We realized that one part of the problem is the high value of ε/k next to walls. However, from Eq. 7.23 it should be clear that $\dot{\omega}_C$ can only be different from zero if c is different from zero.

The problem is that next to the walls a high value for ε/k can be sufficient to start the reaction. A simple "quick and dirty" solution to suppress this is to switch the reaction off in each cells close to the wall.

We can uncomment the following lines in `burn.H` (after the computation of $\dot{\omega}_C$):

```
forAll(mesh.boundary(), patchi) // loop over the patches
{
    // test if the patch is a wall
    if(Foam::isType<Foam::wallFvPatch>(mesh.boundary()[patchi]))
    {// loop on the faces of the patch
        forAll(mesh.boundary()[patchi], facei)
        {
            // get the index of the cell limited by the facei
            label faceCelli = mesh.boundary()[patchi].faceCells()[facei];

            omegaC[faceCelli] = 0.0;
        }
        // set the value of omegaC to zero on the wall
        omegaC.boundaryField()[patchi] = 0.0;
    }
}
```

And in `SchmidFoam.H`, a new header file has to be add before the function `main`:

```
#include "wallFvPatch.H"
```

This will result in a more realistic flame propagation, but not solve the problem completely. Compile the modified solver and run the case again.

Think of other possible ways of suppressing the excess source term next to the walls.

Refining the mesh

You might already have realized that the connecting hole between prechamber and main chamber is discretized with only two cells in y direction. This is not advisable for a good CFD simulation. Of course we could define a finer mesh grading in `blockMeshDict`. However, doubling the cell number in each direction leads to

a huge increase in computational time. It would be much nicer to refine the mesh only where necessary.

This can be done using the `refineMesh` utility. For this utility, we have to define the region in the mesh where refining should take place and the directions of refining (e.g. in our 2D case it is sufficient to refine only in the x and y directions). The `refineMesh` utility will then double the number of vertices in the respective directions within this mesh region. In the `system` directory of our case you will find a file called `cellSetDict`. It is used to define a cell set consisting of regular box surrounding the connection hole between prechamber and main chamber:

```
name            toBeRefined;

action          new;

topoSetSources  (
    boxToCell
    {
        box ( 0.010 -0.014 -0.1 ) ( 0.015 -0.009 0.1 );
    }
);
```

Listing 7.3: cellSetDict

The cell set is named `toBeRefined`. The second dictionary we need is `system/refineMeshDict`. It should look like this:

```
// Cells to refine; name of cell set
set toBeRefined;

// Type of coordinate system:
// - global : coordinate system same for every cell. Usually aligned with
// x,y,z axis. Specify in globalCoeffs section below.
// - patchLocal : coordinate system different for every cell. Specify in
// patchLocalCoeffs section below.
coordinateSystem global;
//coordinateSystem patchLocal;


// .. and its coefficients. x,y in this case. (normal direction is calculated
// as tan1^tan2)
globalCoeffs
{
    tan1 (1 0 0);
    tan2 (0 1 0);
}

patchLocalCoeffs
{
    patch outside; // Normal direction is facenormal of zero'th face of patch
    tan1 (1 0 0);
}

// List of directions to refine
directions
(
    tan1
    tan2
```

```
);

// Whether to use hex topology. This will
// - if patchLocal: all cells on selected patch should be hex
// - split all hexes in 2x2x2 through the middle of edges.
useHexTopology true;

// Cut purely geometric (will cut hexes through vertices) or take topology
// into account. Incompatible with useHexTopology
geometricCut false;

// Write meshes from intermediate steps
writeMesh false;
```

Listing 7.4: refineMeshDict

This dictionary tells the `refineMesh` utility to refine only the cells contained in the set `toBeRefined`. The `globalCoeffs` tell the utility in which direction to proceed along the cell edges when cutting. Our mesh is aligned with the principal axes, so we can use these. It is getting more complicated if you use other meshes. With the `directions` option we tell the utility to define only in the x and y direction.

To execute the utility we have to invoke two commands:

```
cellSet
refineMesh -dict -overwrite
```

The `dict` option tells the utility not to refine the whole mesh but to use the `refineMeshDict` dictionary. The `overwrite` option tells the utility to overwrite the current mesh instead of saving the new mesh to a new folder. As the number of cells in the mesh now changes, you have to do the mesh refinement before starting the `setFields` utility. You can also include the commands in the `Allrun` script.

1. Run the case with the refined mesh. What differences do you see? How can they be explained?
2. Why should the time step of the simulation be changed when the mesh is refined?
3. You might also start on an even coarser grid in the beginning and do multi-stage refinement. Write a script to do this.

Using a detailed chemistry

As an example of a full detailed chemistry solver, you can try the tutorial case: `$FOAM_TUTORIALS/combustion/reactingFoam/ras/counterFlowFlame2D`. To run the case, execute `blockMesh` and then `reactingFoam`. So you can quickly try the case, look at the results and at the files required to define the case. Have especially a careful look to the `constant/thermo..compressibleGas` and `constant/reactions` in which the thermodynamical properties and the chemical reactions are defined.

Background Information

7.5.1 Tools to handle the mesh

This paragraph will introduce a series of tools included with OpenFOAM® in order to manipulate a mesh.

In the following those tools will be shortly described:

- `transformPoints`: Transform the mesh (combination of scaling, translation and rotation of the mesh).
- `rotateMesh`: rotate the mesh and the vector and tensor fields.
- `setSet`: allow to select points, faces and/or cells.
- `subsetMesh`: select a sub part of the mesh.
- `refineMesh`: refine locally the mesh.
- `createPatch`: Create new patches from existing patches and/or set of faces.

The complete set of tools to manipulate a mesh is available by looking in the directory:

```
$FOAM_UTILITIES/mesh/manipulation
```

setSet

This tool allow you to select a group of points, of faces or of cells to treat them later using an other tool. For example you could create a vent in a wall by assigning a couple of faces to a new boundary condition.

You can use this tool interactively or by providing a script.
To run it interactively, just type `setSet` in a terminal positioned in the case directory. Then a short help is provided (as well as examples) by typing simply `help`. By typing `quit` you will quit the program. And `list` will print a list of all current sets. The syntax is always the same:

```
<sort of set> <name> <action> <source>
```

The `sort of set` available are `cellSet`, `faceSet` and `pointSet`. Then the name will characterized the selected elements. The third parameter indicates the action to carried out:

- `new <source>`: create a new set (empty if no source is specified)
- `add <source>`: add elements to an existing set
- `delete <source>`: remove elements from an existing set
- `subset  <source>`: combines current set with the source set
- `list`: prints the contents of the set
- `clear`: clears the set
- `invert`: inverts the set
- `remove`: remove the set

The `source` is a function (type `topoSetSource`) selecting from some parameters cells, faces or points. The full list is quite long and depend of the kind of elements you are dealing with. But the full list can be found in the source code documentation in `$FOAM_SRC/meshTools/sets`. You used one of them when calling the tool `setFields`: `boxToCell`. That function selects all the cells that have their center in the specified box.

subsetMesh

This tool selects a sub-part of the current mesh specified by a `cellSet` to use it at the next time step.

The command is:

```
subsetMesh <cellSet> [-overwrite] [-parallel] [-patch <patch name>]
    [-case <dir>]
```

List of the options:

- `overwrite`: overwrite the current mesh instead of creating the mesh for the next time step.
- `parallel`: if the case is run in parallel and so the mesh distributed.
- `patch <patch name>`: add the new external faces to the specified patch.
- `case <dir>`: path to the case directory.

refineMesh

This tool allow to refine partially or totally the mesh along 1, 2 or 3 directions.

The command is:

```
refineMesh [-dict] [-overwrite] [-parallel] [-case <dir>]
```

List of the options:

- `dict`: refine only the cell specified by the dictionary `system/refineMeshDict`.
- `overwrite`: overwrite the current mesh instead of creating the mesh for the next time step.
- `parallel`: if the case is run in parallel and so the mesh distributed.
- `case <dir>`: path to the case directory.

An example of dictionary is presented in the listing 7.4. The required keywords are:

- `set`: the subset of cells that have to be refined.
- `coordinateSystem`: `global` if the coordinate system is global otherwise `patchLocal`. Both need additional coefficients
  - `globalCoeffs`: list the 3 directions (name + vector)
  - `patchLocalCoeffs`: list the patch used to determine the normal and another direction.

- `directions`: list of directions to refine
- `useHexTopology`: boolean, `true` if the mesh is composed of hexaedra.
- `geometricCut`: boolean, if `true` cut purely the geometry
- `writeMesh`: boolean, if `true` write meshes from intermediate steps

transformPoints

The command is

```
transformPoints -translate "vector" -rotate "(vector␣vector)"
    -scale "vector" -rotateFields
```

The three available options are:

1. `translate`: Translate the mesh according the specified vector
2. `rotate`: Rotate the mesh according the rotation needed to turn the first vector in the direction of the second.
   Alternatively you can use `-yawPitchRoll (yawDegrees pitchDegree rollDegree)` or `-rollPitchYaw (rollDegrees pitchDegrees yawDegrees)` where yaw is the rotation around the z axis, pitch around y and roll around x.
   When rotating the mesh, you can make use of the option `-rotateFields` to rotate the vector and tensor fields as well as the mesh.
3. `scale`: Scale the mesh in the three coordinates direction using each component of the given vector.

The action are done in the order specified above (translation, then rotation and finally scaling). Consequently if you want to change the order you have to execute more than one time `transformPoints`.

rotateMesh

This tool rotate the mesh and the fields at all time steps. The rotation is defined by the rotation to bring the vector *n1* to *n2*.

The command is

```
rotateMesh <n1> <n2> [-latestTime] [-time <ranges>] [-parallel]
    [-noZero] [-case <dir>]
```

The list of options are:

- `latestTime`: Apply the rotation of the mesh at the latest time step only.
- `time <ranges>`: Apply the rotation of the mesh at the time steps specified by the *ranges*[2].
- `parallel`: if the case is run in parallel and so the mesh distributed.
- `noZero`: exclude the time folder 0 from the selected time steps.
- `case <dir>`: path to the case directory.

[2] More information about the specification of *time ranges* is available at `http://openfoamwiki.net/index.php/Tip_Advanced_Time_Selection_Options`

createPatch

Utility to create patches out of selected boundary faces. Faces come either from existing patches or from a faceSet.
More specifically it:

- creates new patches (from selected boundary faces). Synchronise faces on coupled patches.
- synchronises points on coupled boundaries
- remove patches with 0 faces in them

The action are specified in a dictionary, `system/createPatchDict`. The full description of it is shown in the listing below.

```
// This application/dictionary controls:
// - optional: create new patches from boundary faces (either given as
//   a set of patches or as a faceSet)
// - always: order faces on coupled patches such that they are opposite. This
//   is done for all coupled faces, not just for any patches created.
// - optional: synchronise points on coupled patches.

// 1. Create cyclic:
// - specify where the faces should come from
// - specify the type of cyclic. If a rotational specify the rotationAxis
//   and centre to make matching easier
// - always create both halves in one invocation with correct 'neighbourPatch'
//   setting.
// - optionally pointSync true to guarantee points to line up.

// 2. Correct incorrect cyclic:
// This will usually fail upon loading:
//  "face 0 area does not match neighbour 2 by 0.0100005%"
//  " -- possible face ordering problem."
// - in polyMesh/boundary file:
//      - loosen matchTolerance of all cyclics to get case to load
//      - or change patch type from 'cyclic' to 'patch'
//        and regenerate cyclic as above

// Do a synchronisation of coupled points after creation of any patches.
// Note: this does not work with points that are on multiple coupled patches
//       with transformations (i.e. cyclics).
pointSync false;

// Patches to create.
patches
(
    {
        // Name of new patch
        name cyc_half0;

        // Dictionary to construct new patch from
        patchInfo
        {
            type cyclic;
            neighbourPatch cyc_half1;

            // Optional: explicitly set transformation tensor.
```

```
            // Used when matching and synchronising points.
            transform rotational;
            rotationAxis (1 0 0);
            rotationCentre (0 0 0);
            // transform translational;
            // separationVector (1 0 0);

            // Optional non-default tolerance to be able to define cyclics
            // on bad meshes
            //matchTolerance 1E-2;
        }

        // How to construct: either from 'patches' or 'set'
        constructFrom patches;

        // If constructFrom = patches : names of patches. Wildcards allowed.
        patches (periodic1);

        // If constructFrom = set : name of faceSet
        set f0;
    }
    {
        // Name of new patch
        name cyc_half1;

        // Dictionary to construct new patch from
        patchInfo
        {
            type cyclic;
            neighbourPatch cyc_half0;

            // Optional: explicitly set transformation tensor.
            // Used when matching and synchronising points.
            transform rotational;
            rotationAxis    ( 0 0 1 );
            rotationCentre  ( 0.3 0 0 );
        }

        // How to construct: either from 'patches' or 'set'
        constructFrom patches;

        // If constructFrom = patches : names of patches. Wildcards allowed.
        patches (periodic2);

        // If constructFrom = set : name of faceSet
        set f0;
    }
);
```

Listing 7.5: createPatchDict

The command is

```
createPatch [-overwrite] [-parallel] [-case <dir>]
```

List of the options:

- `overwrite`: overwrite the current mesh instead of creating the mesh for the next time step.

- `parallel`: if the case is run in parallel and so the mesh distributed.
- `case <dir>`: path to the case directory.

Chapter

8 Multiphase Flow

8.1 Todays problem

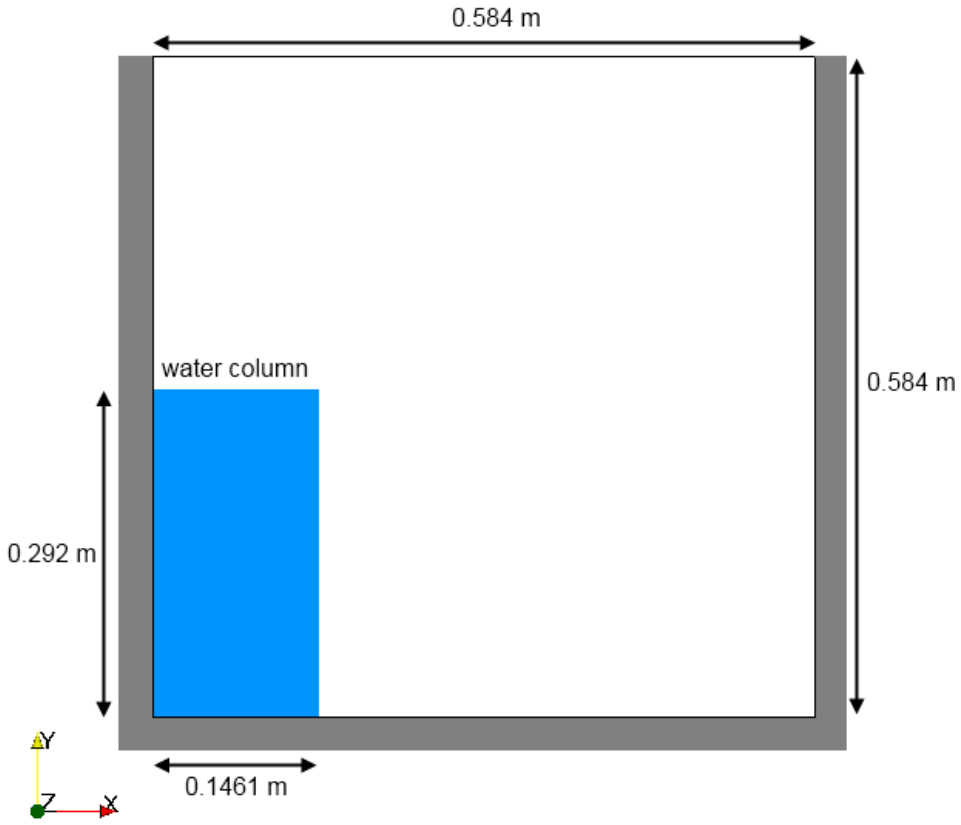


Figure 8.1: Geometry of the collapsing column test case.

This week incompressible air-water flow will be solved in two examples:

1. A column of water falling due to the gravitational acceleration (cf. Figure 8.1). This case will be an opportunity to run a case in parallel using OpenFOAM®.
2. A simplified collector of water droplets will be used to test a homemade boundary condition.

But before introducing the exercises, the physics of incompressible two-phase flow will be presented.

8.2 Physics

A large number of flows in nature and industry involve free surfaces. Their applications range from environmental sciences to numerous engineering problems. Several examples can be cited such as the interface between liquid water and air in the Earth's atmosphere or cargo slosh in ships and trucks transporting liquids.

Flows with free surfaces are a special class of flows with moving boundaries. The position of the boundary is known only at the initial time; its location at later times

Figure 8.2: Slosh phenomenon at laboratory scale.

has to be determined as a part of the solution. There are several methods dedicated to find the shape of the free surface. They can be classified into two major groups:

- **Surface Methods**
  These methods treat the free surface as a sharp interface whose motion is followed. Either boundary fitted grids are used and advanced each time step, which normally requires complex mesh motion and topology change schemes in order to track the position of the interface, or the interface is marked by interconnected massless particles and followed in a Lagrangian manner with the local flow velocities.
- **Volume Methods**
  Under this category are the methods which do not define the interface as a sharp boundary. The computation is performed on a fixed grid, which extends beyond the free surface. The shape of the free surface is determined by computing the (volume) fraction of each near-interface cell that is partially filled.

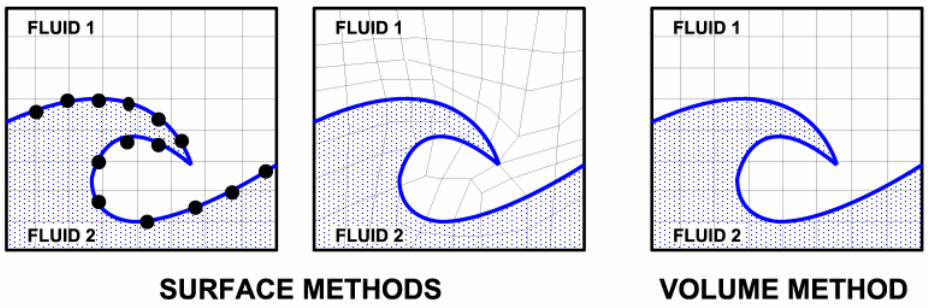


Figure 8.3: Surface vs. Volume schemes.

The Volume-Of-Fluid (VOF) approach is the most popular Volume-based method due to its simplicity. A set of conservation equations is constructed for the phase mixture, *not for the individual phases*, yet the fluid properties (air and water, for example) are blended smoothly and the position of the interface is identified with help of the phase volume fractions. This approach has been shown to be very successful for a wide range of applications, and simple examples will be used here to demonstrate the main ideas of VOF. Its implementation in OpenFOAM® will be presented next.

8.3 Numerics

8.3.1 Volume-Of-Fluid in OpenFOAM®

As mentioned above, the basic idea of the VOF approach is that the two-phase system can be represented as a mixture of the phases in which the volume fraction distribution includes sharp yet resolved transitions between the phases. The complete system of equations of the VOF approach for incompressible two-phase flow without phase change are presented below.

The volume fraction equation (continuity) is defined as

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\mathbf{U}\alpha) = 0, \tag{8.1}$$

which is solved for the volume fraction α. The momentum equation is given by

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla P + \nabla \cdot \tau + \mathbf{F}_\sigma, \tag{8.2}$$

which is solved for the velocity $\mathbf{U}$. The definition of the mixture density is

$$\rho = \alpha \rho_\alpha + \beta \rho_\beta \tag{8.3}$$

in which the constraint $\beta = 1 - \alpha$ is implicit, and the two phase densities are ρ_α and ρ_β. A model for the stress in the momentum equation is given as:

$$\tau = \mu(\nabla \mathbf{U} + \nabla \mathbf{U}^T - \frac{2}{3}\mathbf{I}\nabla \cdot \mathbf{U}), \tag{8.4}$$

with the following definition of the mixture viscosity for laminar flow:

$$\mu = \alpha \mu_\alpha + \beta \mu_\beta. \tag{8.5}$$

It may be augmented by the turbulent viscosity either from a RANS or LES model as appropriate.

The source term $\mathbf{F}_\sigma$ is given by

$$\mathbf{F}_\sigma = \frac{1}{V} \int_{S(t)} \sigma \kappa \mathbf{n} \, dS, \tag{8.6}$$

which represents surface tension effects due to the existence, locally, of two the phases simultaneously. However, since the interface position is not explicitly tracked within the VOF context, this integral can not be calculated explicitly. To overcome this problem, a continuum surface force model is used. This approach considers the surface tension force as a continuous volumetric source. The effective region of

the surface force is limited to the transition between the phases and can be approximated by the following expression:

$$\mathbf{F}_\sigma \simeq \sigma\kappa\nabla\alpha, \tag{8.7}$$

where κ denotes the interfacial curvature, which reads:

$$\kappa = \nabla \cdot \left(\frac{\nabla\alpha}{|\nabla\alpha|} \right). \tag{8.8}$$

In addition to these equations it is necessary to define an equation for the pressure which is present in the momentum equation. For incompressible flow it is traditional to construct a Poissons equation for the pressure based on the divergence-free constraint

$$\nabla \cdot \mathbf{U} = 0 \tag{8.9}$$

which is obtained from the continuity equation and is valid for single and two-phase flow as in this case.

8.3.2 Counter-Gradient transport

Given that the interface disperses due to numerical diffusion it might appear that a good approach would be to limit this effect by including a diffusion operator into the volume fraction equation with a negative diffusion coefficient. While this approach would be conservative it would also be unbounded and unstable; negative diffusion is always problematic. An alternative to negative diffusion which is also conservative is to apply some kind of additional convection-based term which compresses the interface, maintains boundedness and is reduced as the mesh is refined. OpenFOAM® makes use of a so called "counter-gradient" convective term in the continuity equation, such that:

$$\frac{\partial\alpha}{\partial t} + \nabla \cdot (\mathbf{U}\alpha) + \nabla \cdot (\mathbf{U_r}\alpha\beta) = 0. \tag{8.10}$$

Note that the product of the volume fractions guarantees that the compression term vanish in regions where only one phase is present. In order to ensure the compression term does not bias the solution in anyway it should only introduce flow of alpha normal to the interface, i.e. in the direction of the unit normal vector $\frac{\nabla\alpha}{|\nabla\alpha|}$. The compression rate should be set to ensure interface sharpness but more than that might introduce unnecessary numerical difficulties or expense. The worst possible interface dispersion speed is at the fluid velocity $|\mathbf{U}|$. These considerations suggest a model for $\mathbf{U_r}$ of the form:

$$\mathbf{U_r} = c_\alpha |\mathbf{U}| \frac{\nabla\alpha}{|\nabla\alpha|} \tag{8.11}$$

where the compression coefficient c_α should be of order 1.

Another alternative currently being tested is to use $\max(|\mathbf{U}|)$ as a limiter on Eq. (8.11) when c_α is chosen to be larger than 1 to enhance compression

$$\mathbf{U_r} = \min(c_\alpha |\mathbf{U}|, \max(|\mathbf{U}|)) \frac{\nabla\alpha}{|\nabla\alpha|}. \tag{8.12}$$

Note in this expression $\max(|\mathbf{U}|)$ returns the largest value of $|\mathbf{U}|$ anywhere in the domain whereas $\min(c_\alpha |\mathbf{U}|, \max(|\mathbf{U}|))$ returns the minimum value of the $c_\alpha |\mathbf{U}|$

field, limited by $\max(|\mathbf{U}|)$. In practice values of $1 \leq c_\alpha \leq 4$ give good behaviour although for some cases it might be useful to use $c_\alpha \geq 4$.

8.4 OpenFOAM®

8.4.1 Implementation of the volume of fluid model

Equations are to be implemented using a second-order finite-volume discretisation approach with special attention paid to conservation and boundedness of the volume fraction. The first two terms of Eq. (8.10) can be discretised in the same manner as the equivalent terms in other transport equations provided a bounded convection scheme is used, UDS (Upwind Differencing Scheme) or any NVD (Normalized Variable Diagram) based or TVD (Total Variation Diminishing) scheme available for general transport equations should suffice. Discretisation of the remaining term requires the phase compression flux:

$$\phi_{c_\alpha} = -\beta_{f(-\phi_c,\Lambda,)}\,\phi_r \tag{8.13}$$

where Λ represents any bounded interpolation scheme and

$$\phi_r = c_\alpha\,|\phi|\left(\frac{(\nabla\alpha)_f}{|(\nabla\alpha)_f|}\right)\cdot\frac{\mathbf{S}_f}{|\mathbf{S}_f|} \tag{8.14}$$

which is used in the volume fraction equation

$$\left[\!\left[\frac{\partial[\alpha]}{\partial t}\right]\!\right] + \left[\!\left[\nabla\cdot\left(\phi\,[\alpha]_{f(\phi,\Lambda,)}\right)\right]\!\right] + \left[\!\left[\nabla\cdot\left(\phi_{c_\alpha}\,[\alpha]_{f(\phi_{c_\alpha},\Lambda,)}\right)\right]\!\right] = 0, \tag{8.15}$$

OpenFOAM®'s VOF solver is called `interFoam`. The implementation of the volume fraction equation can be found in the `alphaEqn.H` file, which is shown below:

```
{
    word alphaScheme("div(phi,alpha)");
    word alpharScheme("div(phirb,alpha)");

    surfaceScalarField phic(mag(phi/mesh.magSf()));
    phic = min(interface.cAlpha()*phic, max(phic));
    surfaceScalarField phir(phic*interface.nHatf());

    for (int aCorr=0; aCorr<nAlphaCorr; aCorr++)
    {
        surfaceScalarField phiAlpha
        (
            fvc::flux
            (
                phi,
                alpha1,
                alphaScheme
            )
          + fvc::flux
            (
                -fvc::flux(-phir, scalar(1) - alpha1, alpharScheme),
                alpha1,
                alpharScheme
            )
        );
```

```
        MULES::explicitSolve(alpha1, phi, phiAlpha, 1, 0);

        rhoPhi = phiAlpha*(rho1 - rho2) + phi*rho2;
    }

    Info<< "Liquid␣phase␣volume␣fraction␣=␣"
        << alpha1.weightedAverage(mesh.V()).value()
        << "␣␣Min(alpha1)␣=␣" << min(alpha1).value()
        << "␣␣Max(alpha1)␣=␣" << max(alpha1).value()
        << endl;
}
```

Listing 8.1: alphaEqn.H

The first two entries found in this file, "`div(phi,alpha)`" and "`div(phirb,alpha)`" are assigned to the variables "`alphaScheme`" and "`alpharScheme`", respectively. They define the convection discretization schemes of the volume fraction equation in the file `fvSchemes`.

"`cAlpha`" is a scalar expression for limiting the artificial compression velocity. It can be found in the PIMPLE sub-dictionary of the file `fvSolution` and is defined in

`.../src/transportModels/interfaceProperties/interfaceProperties.C`

and

`.../src/transportModels/interfaceProperties/interfaceProperties.H`.

`phic` and `phir` are given by Eqs. (8.13) and (8.14). In the expression for `phir`, `nHatf` is the projection of the interface unit normal vector at the direction normal to the cell face: `nHatf = nHat & Sf` (defined in the `interfaceProperties` files).

`flux` is a member function of the class `fvc` and is called by `fvc::flux`. It will calculate explicit values of `phi` and `phir` defined above.

The solution of the volume fraction equation is performed by calling the Multi-dimensional Universal Limiter with Explicit Solution (MULES) solver. It is specially designed to limit the flux of the variables and guarantee a bounded solution. MULES requires 5 entries:

1. The actual variable which is being solved for, `alpha1` (the volume fraction)
2. The volumetric flux at the cell face (which is the usual convective flux), `phi`
3. The actual explicit flux of the variable, `phiAlpha`
4. The upper bound of the variable to solve for
5. The lower bound of the variable to solve for

After the equation is solved, the total mass-flux (`rhoPhi`) required by the momentum balance is computed such that the limited fluxes employed to solve the volume fraction equation are accounted for in the PIMPLE-Loop.

8.4.2 Structure of a class folder

This week you will have to create a new boundary condition. In OpenFOAM® a boundary condition is a special class. Consequently before focusing on the typical function of a boundary condition, the general structure of a class will be described.

Source code

The source code of class is contained in a folder having the same name as the class itself. And in this folder, you will find usually a header file in which the members and the functions of the class are defined as well as a source file in which the implementation of the functions are written. Those two files have the name of the class but a different extension: .H for the header file and .C for the source file. Each class possesses at least one function: the constructor (explains how to create an object of the class). That function has the same name as the class itself as shown in the listing 8.2.

```
class heatTransferFvPatchScalarField:
public fixedGradientFvPatchScalarField
{
    // Private data
        scalar  k_;
        scalar  h_;
        scalar  Tfluid_;
public:
    // Constructors

    //- Construct from patch, internal field and dictionary
        heatTransferFvPatchScalarField(
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

    // Member functions

    //- Update the coefficients associated with the patch field
        virtual void updateCoeffs();

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

Listing 8.2: Extract of an boundary condition header file.

The last element in the folder is a sub-folder `Make` used by the compilation tool of OpenFOAM®.

The Figure 8.4 shows an example of that structure (This example is the boundary condition used in the chapter 3).

Make folder

The `Make` folder contains two files `files` and `options`. The former contains the list of source files to be compiled in the library as well as the location and the

```
-- heatTransferFvPatchScalarField
    |- heatTransferFvPatchScalarField.H
    |- heatTransferFvPatchScalarField.C
    |- Make
        |- files
        |- options
```

Figure 8.4: Structure of the folder for the `heatTransferFvPatchScalarField` class

name of the library. For a user defined class the path in which the library is stored should look like that `LIB = $(FOAM_USER_LIBBIN)/libMyClasses` . In the latter, you will have to write the folders containing the header files needed and the libraries to be included/linked with the new library.

Compilation of a library

To compile a library, the following command has to be executed :

```
wmake libso
```

This command will compiled the class into a dynamic library (extension .so). And during this compilation, new elements will be generated :

- a file with the name of the class and the extension .dep. That file contains the paths to all header files needed by the current class.
- a folder `lnInclude` that contains a symbolic link to the header files of the classes contained in the compiled library
- a folder `Make\linux64GccDPOpt`[1] that contains the temporary files created during the compilation

It is recommended to clean those elements before compiling again the libraries. To do so, use the command :

```
wclean libso
```

8.4.3 Class for boundary conditions

This section will highlight the major functions defined for a boundary conditions.

As you can see in the listing 8.2, a boundary condition contains an important constructor using a dictionary (i.e. the parameters defined in the boundary conditions file for the current *patch* and the current field). This constructor should consequently interpret the values of the specified keywords in order to initiate the boundary condition. For example, for the *heatTransfer* BC the thermal conductivity k, the heat transfer coefficient h and the temperature of the surrounding fluid $Tfluid$ are read from the dictionary (see lines 12-14 in the listing 8.3).

[1] That name can change according to the specification of the computer

```
#include "heatTransferFvPatchScalarField.H"
#include "volFields.H"

//  * * * * * * * * Constructors  * * * * * * * //
heatTransferFvPatchScalarField::heatTransferFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
):
    fixedGradientFvPatchScalarField(p, iF),
    k_(readScalar(dict.lookup("k"))),
    h_(readScalar(dict.lookup("h"))),
    Tfluid_(readScalar(dict.lookup("Tfluid")))
{
    fvPatchField<scalar>::operator=(patchInternalField());
    gradient() = 0.0;
}

// * * * * * * * * Member Functions  * * * * * * //
void heatTransferFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvPatchField<scalar>& T =
        patch().lookupPatchField<volScalarField, scalar>("T");

    gradient() = h_/k_*(Tfluid_-T);

    fixedGradientFvPatchScalarField::updateCoeffs();
}

void heatTransferFvPatchScalarField::write(Ostream& os) const
{
      fixedGradientFvPatchScalarField::write(os);
          os.writeKeyword("h") << h_ << token::END_STATEMENT << nl;
          os.writeKeyword("k") << k_ << token::END_STATEMENT << nl;
          os.writeKeyword("Tfluid") << Tfluid_ << token::END_STATEMENT << nl;
          writeEntry("value", os);
}
```

Listing 8.3: Extract of a boundary-condition source code.

The second important function is `updateCoeffs`. It is read every time the boundary condition have to be applied. In the example, this function tests if the patch was already updated previously. Then if not the temperature on the patch is obtain by the command `lookupPatchField<Type1, Type2>`. In the third step the gradient of temperature is determined. And finally the called to the top function `fixedGradientFvPatchScalarField::updateCoeffs()` set the proper value to the boundary condition.

The last important function is `write`. This function defined what to write in the field file when an output is required. In the example, first a called to the *write* function of the mother class is done (consistency purpose only). Then the key pa-

rameters are written with first the keyword entry, then the value of it and finally the semi-column. The last thing written is the value of the field on the patch.

Other functions have to be implemented (especially other constructors). But in general their content is not as important as the three previous ones.

8.4.4 The groovyBC boundary condition

This boundary condition is an unofficial boundary condition part of the toolbox *swak4Foam* (for more information see `http://www.openfoamwiki.net/index.php/Contrib/swak4Foam`). Basically it allows the user to defined easily a new boundary condition without having to implement a new class. In addition it comes with most of the characteristics of the `funkySetFields` tool (cf. chapter 3): e.g. expressions could be used to compute values depending on fields solved in the simulation.

To used it with a case, you have to add the following entry in the `controlDict` of the case:

```
libs ( "libOpenFOAM.so" "libgroovyBC.so" );
```

To defined a boundary condition using that boundary condition, you have to specify `groovyBC` as `type`. The parameters that you can set are:

Parameter	Description
`valueExpression`	String with the value to be used if a Dirichlet-condition is needed. Defaults to zero
`value`	is used for the first timestep/iteration if `valueExpression` is specified or all the time if no `valueExpression` is given. Remark: If `valueExpression` is specified without setting "value", 0 is taken for the first timestep/iteration. (might cause an floating point exception)
`gradientExpression`	String with the gradient to be used if a Neumann conditon is needed. Defaults to zero
`fractionExpression`	Determines whether the face is Dirichlet (1) or Neumann (0). Defaults to 1
`variables`	List with temporary variables separated by a semicolon. May make the writing of expressions shorter. Defaults to empty. Names defined here "shadow" fields of the same name
`timelines`	List with subdictionaries that specify interpolation tables over time. See the original `timeVaryingUniform`-condition. Currently only scalars are allowed. The parameter `name` specifies the name under which this may be accessed. The name "shadows" fields of the same name.

Here are two examples:

```
    outlet
    {
        type            groovyBC;
        valueExpression    "vector(0,0,0)";
        gradientExpression    "vector(0,0,0)";
        fractionExpression    "(phi␣>␣0)␣?␣0␣:␣1";
        value            uniform (0 0 0);
   }

   forced
   {
      type      groovyBC;
      value     uniform (0 0 0);
      timelines (
         {
            name impulse;
            outOfBounds clamp;
            fileName "$FOAM_CASE/impulse.data";
         }
      );
      valueExpression      "-impulse*normal()";
      gradientExpression   "vector(0,0,0)";
      fractionExpression "(time()<5)␣?
␣␣␣␣␣␣␣␣␣␣␣␣((pos().x>0.45␣&&␣pos().x<0.55)␣?␣1␣:␣0)␣:␣0";
   }
```

Listing 8.4: Examples of groovyBC usage.

8.4.5 Running in parallel

The mesh and fields are decomposed using the `decomposePar` utility. They can be broken up according to a set of parameters specified in a dictionary named `decomposeParDict` that must be located in the `system` directory of the case of interest. The dictionary entries for the present exercise are reproduced below:

```
numberOfSubdomains 4;

method          simple;

simpleCoeffs
{
    n               ( 2 2 1 );
    delta           0.001;
}

hierarchicalCoeffs
{
    n               ( 1 1 1 );
    delta           0.001;
    order           xyz;
}

manualCoeffs
{
    dataFile        "";
```

```
}

distributed     no;

roots           ( );

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

Listing 8.5: Extract of `decomposeParDict`.

The user has a choice of four methods of decomposition, specified by the method keyword as described below:

`simple`

Simple geometric decomposition in which the domain is split into pieces by direction.

`hierarchical`

Hierarchical geometric decomposition which is the same as simple except the user specifies the order in which the directional split is done.

`scotch`

Scotch decomposition which requires no geometric input from the user and attempts to minimize the number of processor boundaries. The user can specify a weighting for the decomposition between processors, through an optional keyword called `processorWeights` which can be useful on machines with differing performance between processors. The decomposition strategy can be specified with the optional keyword entry `strategy`. The following source code file provides more information: `$FOAM_SRC/decompositionMethods/decompositionMethods/scotchDecomp/scotchDecomp.C`

`manual`

Manual decomposition, where the user directly specifies the allocation of each cell to a particular processor.

For each method there are a set of coefficients specified in a sub-dictionary of decompositionDict, named `<method>Coeffs` as shown in the dictionary listing. The full set of keyword entries are explained below.

The

Compulsory Entries

`numberOfSubdomains`	N
`method`	simple, hierarchical, scotch, metis, manual

`simpleCoeffs`

`n`	Number of subdomains in x,y,z	(n_x, n_y, n_z)
`delta`	Cell skew factor	Typically, 10^{-3}

hierarchicalCoeffs

n	Number of subdomains in x,y,z	(n_x, n_y, n_z)
delta	Cell skew factor	Typically, 10^{-3}
order	Order of decomposition	xyz, xzy, ...

scotchCoeffs

processorWeights	List of weighting factors for allocation of cells to processors; <wt1> is the weighting factor for processor 1, etc.; weights are normalised so can take any range of values	(<wt1><wtN>)
strategy	Decomposition strategy; defaults to "b"	

manualCoeffs

dataFile	Name of file containing data of allocation of cells to processors	"<fileName>"

The

Distributed Data Entries

distributed	Is the data distributed across several disks?	yes/no
roots	Root paths to case directories; <rt1> is the root path for node 1, etc	(<rt1><rtN>)

decomposePar utility is executed in the usual way with the command decomposePar. A set of subdirectories will be then created, one for each processor, in the case directory. The directories are named processorN, where $N = 0, 1...$ represents a processor number and contains a time directory, with the decomposed field descriptions, and a constant/polyMesh directory with the decomposed mesh description.

A decomposed OpenFOAM® case is run in parallel using the openMPI implementation of the standard Message Passing Interface (MPI). The execution line with the command mpirun is given by:

```
mpirun --hostfile <machines> -np <nProcs>
               <foamExec> <otherArgs> -parallel > log &
```

where <nProcs> is the number of processors; <foamExec> is the executable, e.g. interFoam; and, the output is redirected to a file named log. The <machines> file contains the information on where the case is actually going to be run.

Once the case has completed running, the decomposed fields and mesh must be reassembled for post-processing using the reconstructPar utility.

Parallel running in the computer lab

To run a case in parallel in the computer lab, you will use a script based on the following one:

```
1  #!/bin/sh
2  #
3  #This is an example script to execute a parallel job with OpenFOAM
4  # ! Be sure to have decompose using decomposePar before submitting this script
5  #
6  #These commands set up the Grid Environment for your job:
7  # Output folder for the journal file
8  #PBS -o /nfs/home/ofoam/OpenFOAM/ofoam-2.0.1/run/Chpt9/interFoam/experiment
9  #PBS -j oe
10 # Name of the job
11 #PBS -N experiment3procs
12 #PBS -q batch
13 # After ppn= write (2 * number of processors to use)
14 # Here 6 for using 3 processors (this is due to the hyperthreading technology)
15 #PBS -l nodes=1:ppn=6:buddies
16 # E-mail address at which notification will be sent
17 #PBS -M neo@matrixReloaded.de
18 #PBS -m abe
19
20 # Load OpenFOAM-2.0.1
21 source /nfs/etc/bashrc
22 OF201
23 # Change the current directory to the case directory
24 cd /nfs/home/ofoam1/OpenFOAM/ofoam-2.0.1/run/Chpt9/interFoam/experiment
25 # Execute interFoam on the current case directory
26 mpiexec interFoam -parallel > log.interFoam
```

Listing 8.6: Script to run in parallel an OpenFOAM® case.

This script will be sent to a queue in which it will wait until the required resource will be free to run it. To submit the script, `jobScript`, type the command:

```
qsub jobScript
```

This command will print a line containing the *job-ID* of your job. It is important to find your job in the queue. And eventually delete it.

If you want to a have look to the queue and the status of your job, type:

```
qstat -u <username>
```

If you want to delete a job, type:

```
qdel <job-ID>
```

In order to adapt it to your case the following lines have to be modified:

- At the line 8, you specify the path of the journal file of the job.
- At line 11, you can give a name to your job
- At line 15, you can change the number of processors you need. It is the latest number
- At line 24, you specify the path of the test case.
- At line 26, you specify the OpenFOAM® solver you want to run and the name of the log file.

8.5 Exercises

Collapse of a liquid column

A classical case used in the validation of mathematical modelling of free-surface flows is the collapse of a liquid column. Gravitational acceleration causes the water column in the left of the tank to seek the lowest possible potential energy level. Thus, the column will collapse and eventually come to rest at the bottom of the tank[2].

The tank is considered with a base length of 0.584 m. At $t = 0$ s, the water column has a base length of 0.146 m and a height of 0.292 m (cf. Figure 8.1).

A non-uniform initial condition for the volume fraction shall be specified here. This will be done by running the `setFields` utility. It requires a `setFieldsDict` dictionary, located in the `system` directory, whose entries for this case are shown below.

```
defaultFieldValues
(
    volScalarFieldValue alpha1 0
);

regions
(
    boxToCell
    {
        box (0 0 -1) (0.1461 0.292 1);
        fieldValues
        (
            volScalarFieldValue alpha1 1
        );
    }
);
```

Listing 8.7: Extract of `setFieldsDict`.

The top boundary is free to the atmosphere so needs to allow both outflow and inflow according to the internal flow. We therefore use a combination of boundary conditions for pressure and velocity that does this while maintaining stability. They are:

- `totalPressure`, which is a `fixedValue` condition calculated from specified the total pressure and the local velocity;
- `pressureInletOutletVelocity`, which applies `zeroGradient` on all components, except where there is inflow, in which case a `fixedValue` condition is applied to the tangential component;
- `inletOutlet`, which is a `zeroGradient` condition when flow outwards, `fixedValue` when flow is inwards. The keyword to specified the constant inwards value is `inletValue`.

2 The flow scales can be considered to be large when compared to the capillary length, so that surface tension effects are negligible

At all wall boundaries, the `buoyantPressure` boundary condition is applied to the pressure field, which calculates the normal gradient from the local density gradient. The no-slip condition is used for the velocity and `zeroGradient` is used for α. The `defaultFaces` patch representing the front and back planes of the 2D problem, is, as usual, an empty type.

Running of the code has been described in detail in previous tutorials.

The default mesh given for this exercise is fairly coarse. However, as outlined before, the nature of the VOF method means that an interface between the fluids is not explicitly computed, but rather emerges as a property of the volume fraction field. Since the volume fraction can have any value between 0 and 1, the interface is never sharply defined, but occupies a volume around the region where a sharp interface should exist. In order to achieve a resolution that resembles that of a real, sharp, interface, normally much finer mesh levels are required than the one used, increasing the costs of running in a single processor considerably. The method of parallel computing used by OpenFOAM® is known as domain decomposition, in which the geometry and associated fields are broken into pieces and allocated to separate processors for solution.

Questions

Preliminary remark: the execution time needed for this first exercise is quite high. So it's advised that you start the second exercise when waiting for the results.

- Run the collapsing column case for two successively finer grid levels (2x and 4x the **total** number of cells) and compare the solutions qualitatively, analysing the interface resolution. For grid level with 2 times more cells, run the case with 1, 2 and 3 processors. For the coarsest and finest meshes, use 3 processors. Compare the CPU time for each run. Is a linear speed-up observed in the parallel computations?
  Advice: Make advantageous use of a grading mesh within the `blockMeshDict` or use the `refineMesh` tool.
- Then compare qualitatively the results for the form of the interface with the experimental images available at time instants $t = 0$, 0.2, 0.4, 0.6, 0.8 and 1 s (see Fig. 8.5). In `paraview`, consider the interface as the geometrical entity connecting the regions with $\alpha = 0.5$.

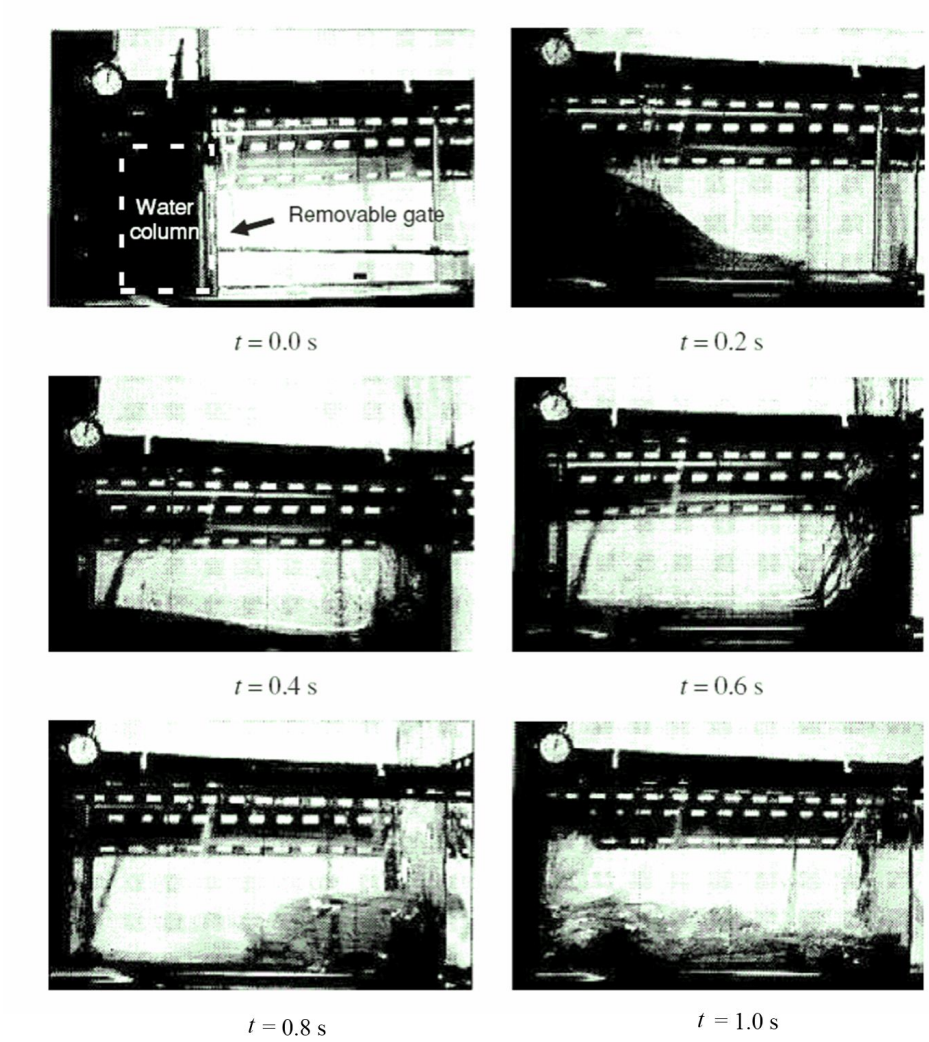


Figure 8.5: Experimetal observations of a collapsing water-column without obstacle (Ubbink, 1997).

Droplets collector

In this second exercise, a new boundary condition will be implemented to generate random droplet appearing at the bottom of the domain and falling in a opened tank below as described in the figure 8.6.

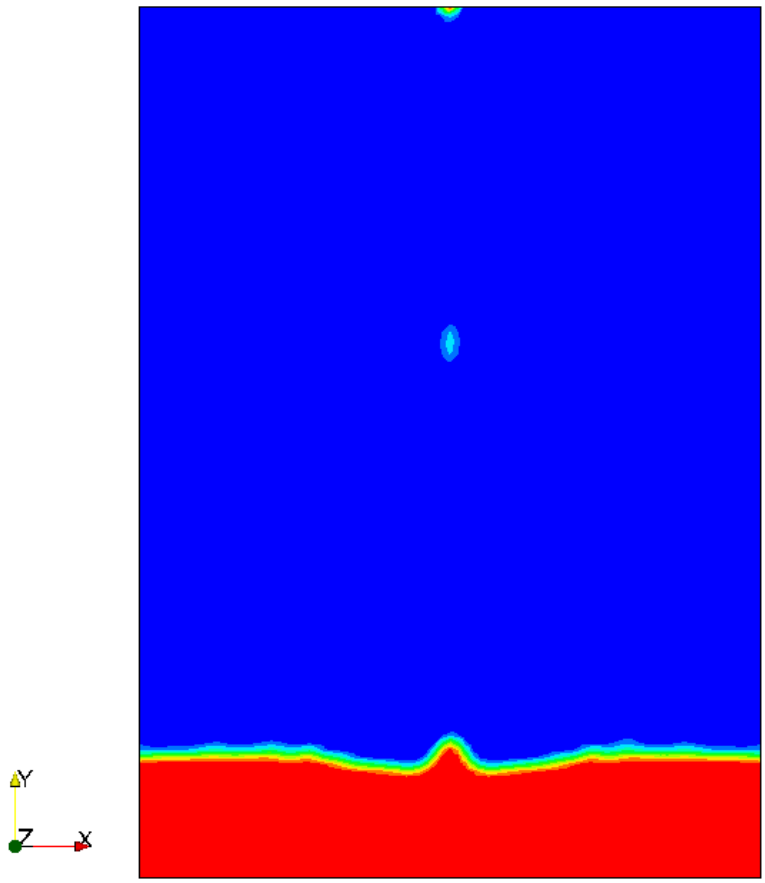


Figure 8.6: Droplet collector geometry.

To be able to simulate this case, a new boundary condition has to be implemented for the velocity: `rainDropletInletVelocity`. This function will reproduce the behavior of droplet falling in the domain. The switch between a status with and without droplet will be described using a Poisson process: i.e. the velocity value switches for the nth time at the time t when the following statement is true:

$$U_{[0,1]} > exp(-\lambda(t - T_{n-1})) \tag{8.16}$$

where $U_{[0,1]}$ is a random number uniformly distributed on $[0, 1]$, λ is the frequency of the process.
The value of the velocity, U, is computed from a given volume flow rate (as the flow is incompressible).
The skeleton of this function is provided. So some elements have to be added:

1. The members of the class: two scalars `frequency_` and `flowRate_` and a

Random variable `rnd_`[3].

2. The constructor base on a `dictionary` has to be fulfilled. For that you have to
   - Read `flowRate_` and `frequency_` from the dictionary
     The command to read a scalar value is `readScalar(dict.lookup("keyword"))`
   - Initialize the random variable with a integer (the one you want)
3. In the function `updateCoeffs`, the condition for which the value of the velocity switch between 0 and a certain value (cf. equation 8.16) (Hint : to know how to generate an uniform random variable have a look at the documentation of the `Random` class[4]).
4. The `write` function has to be completed to output `flowRate_`, `frequency_`, `lastSwitch_` and the value of the velocity on the patch.

Hint: As source of inspiration, you can have a look at the boundary condition *flowRateInletVelocity*. Location of the sources :
`$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/flowRateInletVelocity`

When the changes are made, you can try to compiled the boundary condition.

Before running the test case, the `droplet` boundary condition for `alpha` has to be given. As the droplet is there only if the velocity is greater than zero, the `groovyBC` will be used to set `alpha1` to one when $U > 0$ and zero otherwise.
Remark: You need to change the boundary condition in the file `alpha1.org`. Then run `Allclean` that will overwrite `alpha1` with `alpha1.org`.

You can now run the case using the script `Allrun`. Or if you want to parallelize it, run the commands prior to `runApplication $application` in `Allrun`. Then the cluster can be used to run the case in parallel.

[3] By convention in OpenFOAM® the member names end with an *underscore*

[4] See background information for more details about the source code documentation.

8.6 Extra Practice and Background Information

Extra Practice

Droplets collector: advanced level

If you have a look in the script `Allrun`, you will see that the geometry is generated using two blocks defined in `blockMeshDict`. Then using the tool `setSet` some faces of the top boundary are selected in a `faceSet`. Then using `createPatch` those faces are converted in a new boundary condition called `droplet`.

For this advanced exercise, you will have to create first two additional positions at which the droplets appeared inside of the domain. Then using `timeVarying FlowRateInletVelocity` instead of your customized boundary condition, you will create a time variation of the velocity to simulate droplet appearing at the top of the domain.

Remark:

- Here is an example of use of the time varying boundary condition.

```
inlet
{
    type            timeVaryingFlowRateInletVelocity;
    flowRate        0.2;  // Volumetric/mass flow rate [m3/s or kg/s]
    value           uniform (0 0 0);  // placeholder
    fileName        "$FOAM_CASE/time-series";
    outOfBounds     repeat;  // (error|warn|clamp|repeat)
}
```

  Note: The value is positive inwards
- The structure of the time series looks like:

```
// Time FlowRate
(
(0.0 0.12)
(0.1 0.14)
(0.2 0.0)
);
```

Slosh phenomena in a tank truck

As additional exercise, the slosh phenomena in a tank truck will be studied. The tank will be half full of water. And the slosh phenomena will be initiated by presuming that the truck brakes.

For that, only the tank will be modeled as a 2D geometry with the dimensions specified in the Figure (8.7). The cell size will be 10 cmx 10 cm.

The *k-epsilon* turbulence model will be used (not `laminar` as for the `damBreak` case). So the boundary conditions as to be set for the pressure, `p`, the velocity, `U`, the volume fraction, `alpha1`, the turbulent kinetic energy, `k`, its dissipation rate, `epsilon` and the dynamic viscosity, `mut`. As the system is closed, all the boundary

Figure 8.7: Tank truck.

conditions are similar and correspond to the wall condition.
For `mut`, the name of the wall boundary condition is `mutWallFunction`.
For `k`, don't forget to set the values to something small but non zero.

As the system is closed, you need to provide two additional entry in the PISO dictionary (in `system/fvSolution`):

1. `pRefValue`: reference for the pressure (here 0).
2. `pRefPoint`: the point where the reference pressure is set (here you can set the upper right corner).

The last parameters to set are the numerical schemes and the solver parameters for `k` and `epsilon`. For that, copy the parameters used in the tutorial case: `$FOAM_RUN/multiphase/interFoam/ras/damBreak`.

Use then `setFields` to set `alpha1` to 1 in the down half of the tank.

The scenario is a truck braking uniformly from 80 km/h to 56 km/h in 3 seconds. Consequently the contain of the tank will be move due to mass inertia. Physically this is translated in a source term in the momentum equation, S_{truck}:

$$S_{truck} = -ma$$

, where a is the vector acceleration of the truck.

In finite volume model this is equivalent to add a volume source term of $-\rho a$ like for the gravitation. So you don't have to modify the solver. Only the gravitation acceleration has to be changed to take this acceleration into account.
Remark: the acceleration of the truck is horizontal and the gravitation is vertical.

Run then this case on 4 processors. Be aware that it will take some time.
In a second step, the truck stabilizes its speed to 56 km/h ($a = 0m/s^2$). Starting for the state after 3 s, run the case for an additional 3 s.

Background Information

8.6.1 Source code documentation

The code source of OpenFOAM® contains formatted comments that are gathered by a software called *Doxygen*. The output is a documentation in html accessible from the official website of OpenCFD® (`http://www.openfoam.com/docs/cpp/`).

The main page looks like this:

Figure 8.8: C++ Source Guide: main page

If you are looking for a solver documentation, the best documentation is generally the code itself. To have a look to it, you can either navigate to `$FOAM_SOLVERS` or in the Doxygen documentation look in the tab *Directories*. The beginning is the list of the solvers.

To describe the structure of the documentation for a model or a class, we will look for the *k-epsilon* model for incompressible flow as an example. For that, type in the search box the key word *kEpsilon*.

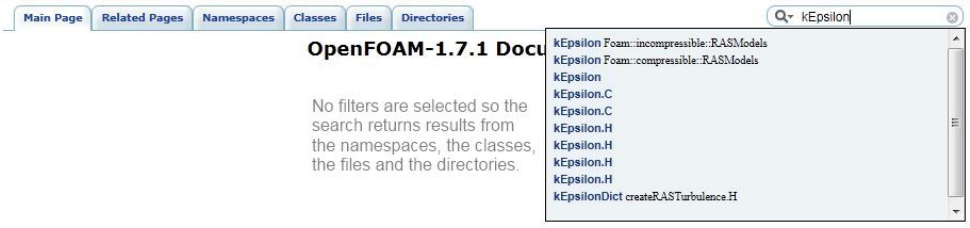


Figure 8.9: C++ Source Guide: search box

Click on the first link **kEpsilon** *Foam::incompressible::RASModels*.

You are now on the page describing the *Foam::incompressible::RASModels* class. The description of a class is divided into 6 sub-parts:

- The inheritance and the collaboration diagrams
- The public member functions list
- The detailed description of the class (the element usually missing in OpenFOAM®).
- The documentation for the constructor and the destructor
- The documentation for all the member functions
- The list of files corresponding to the documentation of the class

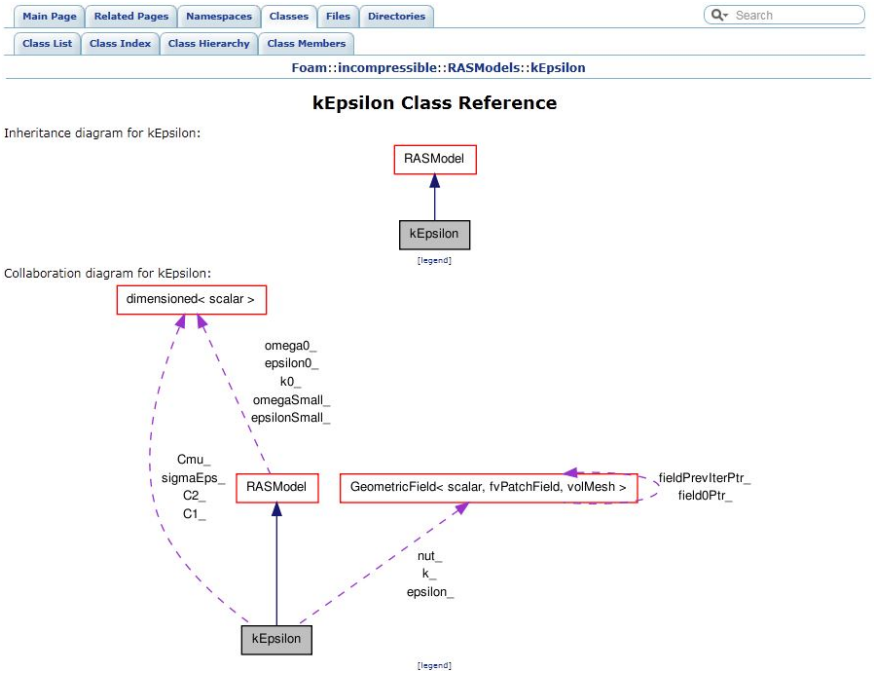
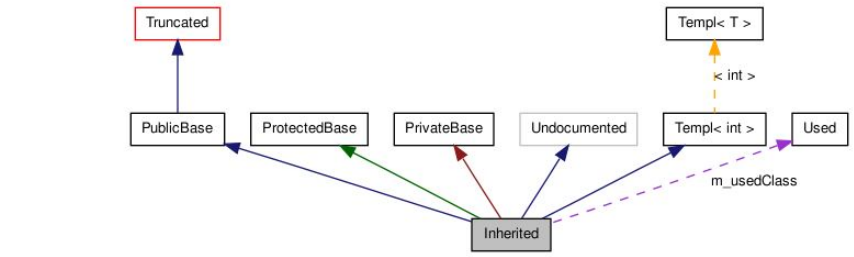


Figure 8.10: C++ Source Guide: inheritance and collaboration diagrams



The boxes in the above graph have the following meaning:

- A filled gray box represents the struct or class for which the graph is generated.
- A box with a black border denotes a documented struct or class.
- A box with a grey border denotes an undocumented struct or class.
- A box with a red border denotes a documented struct or class forwhich not all inheritance/containment relations are shown. A graph is truncated if it does not fit within the specified boundaries.

The arrows have the following meaning:

- A dark blue arrow is used to visualize a public inheritance relation between two classes.
- A dark green arrow is used for protected inheritance.
- A dark red arrow is used for private inheritance.
- A purple dashed arrow is used if a class is contained or used by another class. The arrow is labeled with the variable(s) through which the pointed class or struct is accessible.
- A yellow dashed arrow denotes a relation between a template instance and the template class it was instantiated from. The arrow is labeled with the template parameters of the instance.

Figure 8.11: C++ Source Guide: legend of the diagrams

The inheritance and collaboration diagrams

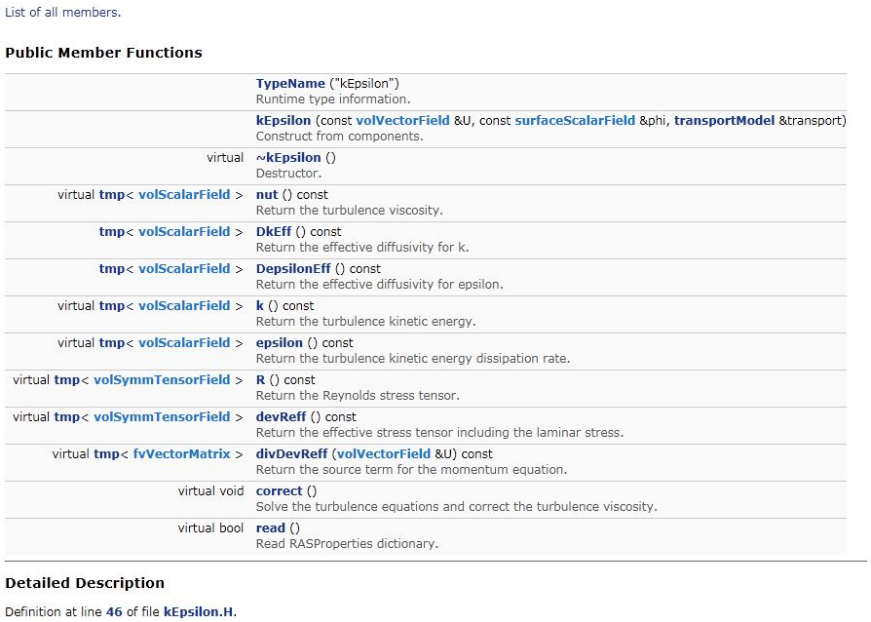
The inheritance diagram presents the mother classes of the current class (with the arrows starting from the class name) and the children (arrows ending on the class name). In this case the *kEpsilon* class inherits directly only from the *RASModel*.

The collaboration diagram highlights the relations between the classes due to inheritance or due to members.

The description of the symbols (arrow, color,...) used in those two diagrams is available by clicking on the link *legend* below each diagram. For easiness, it's reproduced in the figure (8.11).

The public member functions list

This is a short list of the public functions (meaning accessible outside of the class) implemented in the current class. This list is followed by a detailed description of the class. In this case, as many others, it says nothing... It is preceded by a link


List of all members.

Public Member Functions

	TypeName ("kEpsilon") Runtime type information.
	kEpsilon (const **volVectorField** &U, const **surfaceScalarField** &phi, **transportModel** &transport) Construct from components.
virtual	**~kEpsilon** () Destructor.
virtual **tmp**< **volScalarField** >	**nut** () const Return the turbulence viscosity.
tmp< **volScalarField** >	**DkEff** () const Return the effective diffusivity for k.
tmp< **volScalarField** >	**DepsilonEff** () const Return the effective diffusivity for epsilon.
virtual **tmp**< **volScalarField** >	**k** () const Return the turbulence kinetic energy.
virtual **tmp**< **volScalarField** >	**epsilon** () const Return the turbulence kinetic energy dissipation rate.
virtual **tmp**< **volSymmTensorField** >	**R** () const Return the Reynolds stress tensor.
virtual **tmp**< **volSymmTensorField** >	**devReff** () const Return the effective stress tensor including the laminar stress.
virtual **tmp**< **fvVectorMatrix** >	**divDevReff** (**volVectorField** &U) const Return the source term for the momentum equation.
virtual void	**correct** () Solve the turbulence equations and correct the turbulence viscosity.
virtual bool	**read** () Read RASProperties dictionary.

Detailed Description

Definition at line **46** of file **kEpsilon.H**.

Figure 8.12: C++ Source Guide: public member functions

List of all members to display the full list of the public member functions including the inherited public member functions. For example at the end of the figure (8.13), you can see that an inherited function of *kEpsilon* is the function *coeffsDict()* from *RASModel*.

Constructor and Destructor documentation

The only interesting additional element of this part of the documentation is a direct link to the code implementation of the function. For example here (see figure (8.14)), you can access directly at the line 40 of `kEpsilon.C` where the constructor is implemented.

Member functions documentation

As for the constructor and destructor documentation, the only interest here is the direct link to the implementation in the code of the function.

kEpsilon Member List

This is the complete list of members for **kEpsilon**, including all inherited members.

add(entry *, bool mergeEntry=false)	**dictionary**	
add(const entry &, bool mergeEntry=false)	**dictionary**	
add(const keyType &, const word &, bool overwrite=false)	**dictionary**	
add(const keyType &, const string &, bool overwrite=false)	**dictionary**	
add(const keyType &, const label, bool overwrite=false)	**dictionary**	
add(const keyType &, const scalar, bool overwrite=false)	**dictionary**	
add(const keyType &, const dictionary &, bool mergeEntry=false)	**dictionary**	
add(const keyType &, const T &, bool overwrite=false)	**dictionary**	[inline]
append(link *)	**DLListBase**	
AUTO_WRITE enum value	**IOobject**	
bad() const	**IOobject**	[inline]
BAD enum value	**IOobject**	
begin()	**DLListBase**	[inline]
begin() const	**DLListBase**	[inline]
caseName() const	**IOobject**	
cbegin() const	**DLListBase**	[inline]
cend() const	**DLListBase**	[inline]
changeKeyword(const keyType &oldKeyword, const keyType &newKeyword, bool forceOverwrite=false)	**dictionary**	
checkIn()	**regIOobject**	
checkOut()	**regIOobject**	
ClassName("dictionary")	**dictionary**	
clear()	**dictionary**	
Foam::clone() const	**IOobject**	[inline]
Foam::dictionary::clone() const	**dictionary**	
close()	**regIOobject**	
coeffDict() const	**RASModel**	[inline]
coeffDict_	**RASModel**	[protected]

Figure 8.13: C++ Source Guide: all public member functions

List of files

For information or to get access to the source code.

Constructor & Destructor Documentation

kEpsilon (**const volVectorField &** **U,**
const surfaceScalarField & **phi,**
transportModel & **transport**
)

Construct from components.

Definition at line **40** of file **kEpsilon.C**.

virtual ~kEpsilon ( ) [inline, virtual]

Destructor.

Definition at line **84** of file **kEpsilon.H**.

Member Function Documentation

● ● ● Some functions have been skipped

tmp< fvVectorMatrix > divDevReff (**volVectorField &** **U**) **const** [virtual]

Return the source term for the momentum equation.

Implements **RASModel**.

Definition at line **172** of file **kEpsilon.C**.

void correct ( ) [virtual]

Solve the turbulence equations and correct the turbulence viscosity.

Implements **RASModel**.

Definition at line **200** of file **kEpsilon.C**.

bool read ( ) [virtual]

Read RASProperties dictionary.

Implements **RASModel**.

Definition at line **182** of file **kEpsilon.C**.

The documentation for this class was generated from the following files:

- src/turbulenceModels/incompressible/RAS/kEpsilon/**kEpsilon.H**
- src/turbulenceModels/incompressible/RAS/kEpsilon/**kEpsilon.C**

Figure 8.14: C++ Source Guide: detailed documentation

Chapter

9 Lagrangian Particle Tracking

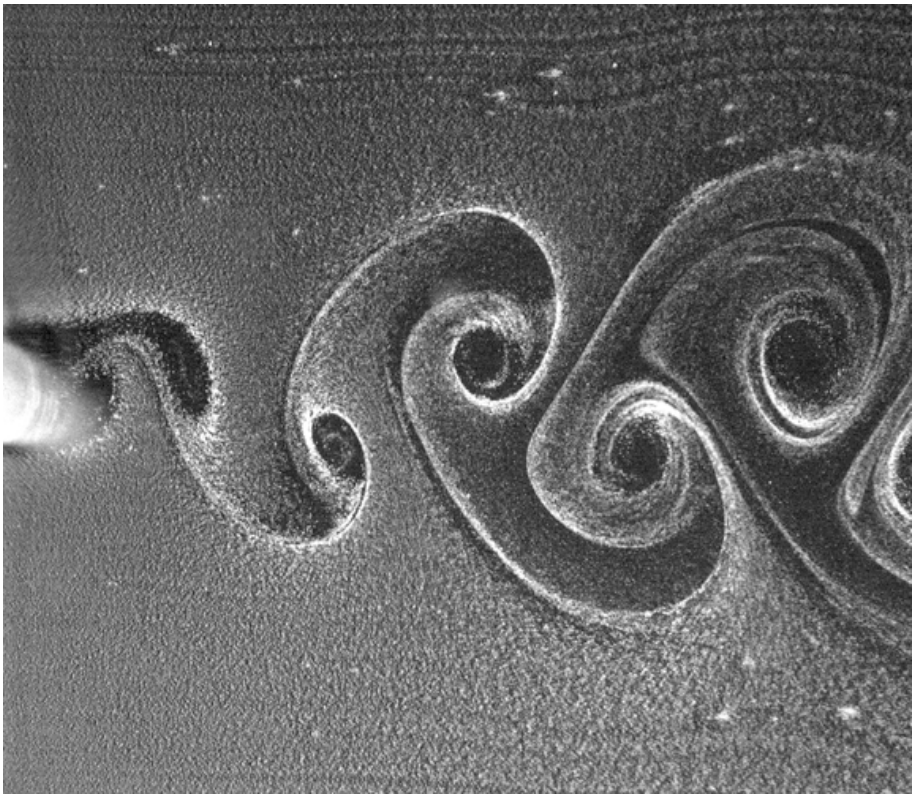


Figure 9.1: von Karman street pattern

9.1 Todays problem

The exercise of this chapter will be to highlight the von Karman street pattern appearing behind a cylinder when the Reynolds number of the flow is moderate. To make the pattern clearer small particles will be injected in the flow upward of the cylinder. Consequently a lagrangian solver is required to solve the movement of those particles.

Bibliography

[1] Euler/Lagrange simulation method for particle-laden flows, *Verlag 1996*

[2] Fluid dynamics and transport of droplets and sprays, W.A.Sirignano, *Cambridge University Press 1999*

9.2 Physics

Multi-fluid and multi-phase flows play a dominant role in a wide range of natural processes and industrial applications. Considering the field of particle-laden or bubbly flows, which means that particles with a low volume fraction are dispersed in a continuous fluid, examples can be categorized as (not complete):

particles/continuous phase

- solid/gaseous: sandstorm, snowfall, sand brush, coal combustion, centrifuges, etc.
- liquid/gaseous: rain, ocean surface, airbrush, drying processes, fuel sprays, etc.
- gaseous/liquid: ocean surface, cavitation, chemical processes, etc.

Assuming that the density of the surrounding fluid is much less than the particle material density and considering only the friction force between continuous phase and the particles (drag), and the gravitational acceleration, the lagrangian equation of particle motion reads:

$$\underbrace{\frac{d\vec{u_p}}{dt}}_{\text{inertia}} = \underbrace{-18\frac{\mu_g}{\rho_p D_p^2}\left(\vec{u_g} - \vec{u_p}\right)}_{\text{friction}} + \underbrace{\vec{g}}_{\text{gravity}} , \tag{9.1}$$

where subscript d stands for dispersed phase properties and subscript c for continuous phase properties. Rearranging the equation in terms of the drag coefficient, it becomes

$$\frac{d\vec{u_p}}{dt} = -\frac{3}{4}\frac{\rho_g}{\rho_p}\frac{c_D}{D_p}|\vec{u}_g - \vec{u}_p|(\vec{u}_g - \vec{u}_p) + \vec{g} \, . \tag{9.2}$$

c_D is given by $c_D = \frac{24}{\mathrm{Re_p}}$ for Stokes flow ($\mathrm{Re_p} \ll 1$) and for $\mathrm{Re_p} \gg 1$ the drag coefficient can be assumed following Schiller and Naumann as:

$$c_D = 24/\mathrm{Re_p}\left(1 + 0.15{\mathrm{Re_p}}^{0.687}\right) \tag{9.3}$$

Re_p is the particle Reynolds number built with the particle diameter and the relative velocity between fluid and particle.

To calculate the particle trajectory Eqn. 9.4 must be integrated.

$$\vec{x_p}(\vec{x}, t) = \int_t^{t+\Delta t} \vec{u_p}(\vec{x}, t)dt \, . \tag{9.4}$$

Besides the drag, other particle-fluid interaction forces are described below. Apart from special cases which gives optimal conditions to evoke these forces, they are negligible in most of situations. To keep simplicity all following described forces are neglected in our simulations.

- Lift or Magnus force: force which results from pressure gradient perpendicular to the motion, evoked by particle rotation or velocity gradients for example in boundary layers.
- Saffmann force: special case of the Magnus force for very small particles with creeping flow ($\mathrm{Re_p} \ll 1$)
- Virtual mass and Basset force: unsteady forces which occur during particle acceleration or deceleration

9.3 Numerics

In most of relevant cases which are object of numerical simulation, the particle diameter is much smaller than the typical cell size of the computational grid. This means that the flow scales in size order of the particle diameters can not be resolved. The particles are assumed therefore as mass points, where the flow around the particle is not resolved but modeled in terms of the drag coefficient for example. Since the particle-phase is not a continuous fluid the Navier-Stokes equations are not valid. Two mathematical approaches are common in describing the particle motion. The more intuitively method is the Euler/Lagrange method where the particles are represented by numerical particles for which the equation of single particle motion is solved. For efficiency reasons these numerical particles are used instead of real particles where each numerical particle or parcel (of particles) represents numerous real particles. Each numerical particle is created during the simulation and its trajectory calculated. Within every iteration of the continuous phase, several iterations for the particle equation are executed, because the particle trajectory has to be resolved in scales smaller than the grid scale. This method is very accurate in predicting polydispersity (diameter dependency) and particle trajectory crossing, however, drawbacks are the stability behavior, computational cost (time and storage) and parallelization. Following each individual particle one have to know where it is, within which cell, who are the neighbor particles, their locations, whether a collision is probable, etc. This lead to the further development of other techniques. The most important alternative approach is to treat the particles also as a continuous phase described by (parts of) the Navier-Stokes equations which interacts with the surrounding phase via exchange terms (mass, momentum, energy). This grid-based method is less expensive in computational cost and storage but also does not resolve the particle dynamics in the detailed resolution as the Euler/Lagrange methods. It is for example more difficult to describe polydisperse effects as particles having different velocities and flow directions within one cell, which includes phenomena as crossing particle trajectories and different inertia behavior of particles with different diameters. In this chapter we focus on the Euler/Lagrange method.

9.4 OpenFOAM®

9.4.1 Lagrangian Particle Tracking in OpenFOAM®

Several solvers are available in OpenFOAM® for particle-laden flows, especially for evaporating and reacting particles (`coalChemistryFoam`, `reactingParcelFoam`, etc.). Unfortunately the only solver to transport particles without interaction with

the gas flow are post-processing solver; meaning that they will read the velocity field at the latest time step and transport the cloud of particles using that velocity field. A cloud in OpenFOAM® is a "cloud of parcels or numerical particles", which means that every cloud is a *<particleType>-cloud* of *<particleType>-particles*. Since we are interested in solving neither the thermal properties of the particles nor the compressibility of the continuous fluid, we need to create a new solver. As a start point we use the solver `pisoFoam`, which provides the framework for solving an incompressible continuous fluid either laminar or with RANS/LES. The aim is now to insert the additional functions and definitions for solving the particle tracking and the momentum exchange between the two phases. Following steps are needed:

- Definition and initialization of the particle cloud
- Solving for the particle motion
- Introduction of source terms in the flow equations due to the particles (momentum, energy, species conservation,...)

The last step will be skipped for this exercise as the particles do not interact with the carrier flow. For that kind of particles, we can use the type `<basicKinematicCollidingCloud>` for cloud definition. The post-processing solver `icoUncoupledKinematicParcelFoam` is a good example to find the key functions required by such cloud class.

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

    Info<< "Evolving " << kinematicCloud.name() << endl;

    laminarTransport.correct();

    mu = nu*rhoInfValue;

    kinematicCloud.evolve();

    runTime.write();

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

Listing 9.1: Time loop of `icoUncoupledKinematicParcelFoam`

The function `evolve()` of `kinematicCloud` calculates the particle motion including the complete particle tracking solver. That function doesn't require any arguments. Therefore all the useful fields should be given to `kinematicCloud` at its creation as can be seen in Listing 9.2.

```
Info<< "Constructing kinematicCloud " << kinematicCloudName << endl;
basicKinematicCollidingCloud kinematicCloud
(
    kinematicCloudName,
    rhoInf,
    U,
    mu,
    g
);
```

Listing 9.2: Definition of a `basicKinematicCollidingCloud`

9.4.2 Postprocessing of Particle Trajectories

Using `foamToVTK`

The OpenFOAM® related version *paraFoam* of the software *paraView* can not post-process the lagrangian particle tracks directly. Therefore a trick has to be used when working with *paraView*. First the results have to be converted from the OpenFOAM® files into VTK files using the tool `foamToVTK`. A folder `VTK` will be created with several subfolders and a single `.vtk` file at the same folder level. This contains the variables as we usually use with *paraFoam*. They can be opened by loading this file in *paraView*. After that, these data can be handled in the same way as in *paraFoam*. Plotting the contours on a slice with $z = 0$ (3^{rd} dimension) instead on the "volume" ensures the visibility of the particles, because particles have z-values of $z = 0$ and the depth of the artificial cell in the z-direction is larger than the particle sizes, which are appropriate for visualization. Within the folder `lagrangian` the particle trajectories and related information are stored. Opening the corresponding `.vtk` file and creating a glyph object choosing *scalar=d, vector=U, type=sphere, scale mode=scalar* and *scale factor* of about 2 in our case will create spheres which represents the particles. By time animation of the files, the movement of the particles and the transient flow field contours can be made visible.

Using `paraFoam`

It is also possible to use the traditional tool *paraFoam*. However it has to be used in two times. First execute the command as usual and load only the geometry and the fields linked to the continuous phases (i.e. select nothing in the *Lagrangian fields* section neither some *lagrangian* geometry in the Mesh parts section.). And like usually finish by clicking on the *Apply*.

Then click on the *Open* button (or in the menu *File -> Open*) and select in the case folder the file with the extension *.OpenFOAM*. It will let you load again all the data of the simulation. So this time select only the geometry (in the *Mesh Parts* section) of type *lagrangian* and only fields in the *Lagrangian Fields* section. Finally click on the *Apply* button.

As the particle are very small, we advice you to used a slice of the domain on the plan $z = 0$. Then to visualize the particles apply the *Glyph* filter to the particles fields (cf. previous paragraph for more information).

9.4.3 Boundary conditions for external flows

Up to now, the flows simulated where contained in a channel or in a closed geometry. In this exercise, you will simulate another kind of flow known as external flow. In external flow an object in submerged in the atmosphere. The main goal is usually to know the forces acting on the object and the flow distortion introduced

by it. Typical examples are flows around wings, planes, cars,... .
For such kind of flows, the boundary conditions have to be computed from the *freestream* state. That means from the know velocity, pressure and turbulence characterizing the flow far from the submerged object[1].

To simplified the settings of the boundary conditions in such cases when the flow is subsonic, OpenFOAM® provides simple boundary types:

Field	Name boundary condition	Mandatory parameters
All except the pressure	`freestream`	`freestreamValue`
Pressure, `p`	`freestreamPressure`	*none*

The `freestream` corresponds to an `inletOutlet` boundary condition with the inward value specified by the keyword `freestreamValue`. That value has to be set with the value of the farfield velocity.
The `freestreamPressure` boundary conditions corresponds to a `zeroGradient` boundary condition for the pressure. And the mass flux through the patch faces, ϕ, is computed using:

$$\phi = \rho \vec{S}_f \cdot \vec{U}_{\texttt{freestreamValue}}$$

9.5 Exercises

A classical configuration is the flow around a cylinder, where the very interesting flow structures which can be made visible with so-called tracer particles. Tracer particles should follow the fluid flow perfectly and their impact on the continuous phase should be negligible to ensure accurate visualization of the single-phase flow. Injected at appropriate locations the tracer particles concentrate at the vortex cores of the cylinder wake and make the vortex shedding well visible. For simplification we use here the infinite long cylinder configuration which can be approximated accurately as a 2D flow around a circle. Dependent on the Reynolds number (built with the diameter of the cylinder) different types of the von Karman vortex street occur. For very low Reynolds numbers the recirculation downstream of the cylinder is axisymmetrical and becomes instable for increasing Re. At Reynolds numbers around $\mathrm{Re} = 3.e+05$ the flow changes from laminar to turbulent. We choose a moderate Reynolds number of

$$\mathrm{Re} = \frac{c\,d}{\nu} = \frac{0.13062\frac{m}{s} \cdot 0.01m}{1.7894e^{-06}} \frac{m^2}{s} = 730\,,$$

where the flow is laminar but with a nice and clearly periodic vortex shedding. In figure 9.2 a typical snapshot of this flow structure is shown.

With given velocity and cylinder diameter a vortex shedding frequency according to Strouhal (with $\mathrm{Sr} = 0.216$) can be calculated with

$$f = \frac{\mathrm{Sr}u}{d}\,.$$

[1] A tutorial using those boundary conditions is available there: `$FOAM_TUTORIALS/incompressible/simpleFoam/airFoil2D`

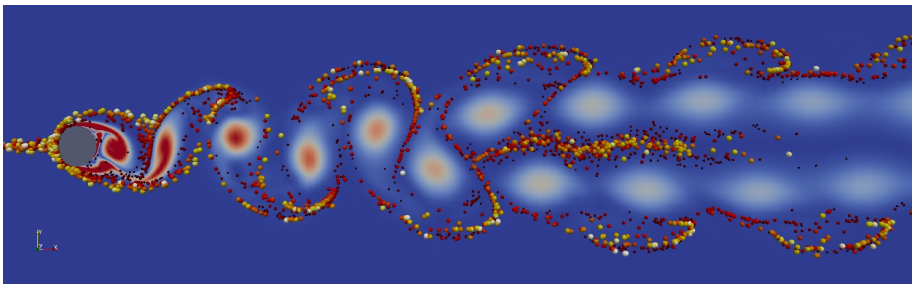


Figure 9.2: Particle-laden flow around a cylinder at low Reynolds numbers, vorticity magnitude and polydispersed particles

This is valid for the single-phase flow. Heavy particles may influence this frequency due to their acceleration or deceleration effect on the gas flow.

To ensure that the injected particles follow the continuous flow exactly we choose a Stokes number of $\mathrm{St} \ll 1$. The particles then behave like tracer particles to visualize the flow structure. Therefore we set their mean diameter to a low value, arbitrarily to 50 microns. With this characteristics the impact of the particles on the flow is negligible. We only have the momentum flux from the gas phase to the particles (one-way coupling).

For the solver `kinematicParcelFoam` as also for the other particle solvers the injection can be realized by different injection types. These are amongst others `ConeInjection`, `PatchInjection`, `ManualInjection`, etc. We use in a first step the `ConeInjection`, which is a multi-points source creating a spray cone, where the inner and outer cone angle can be specified. The relevant section of the `kinematicCloudProperties` is shown below.

```
solution
{
    // Boolean to active this cloud of particles
    active                    true;
    // If true, two-way coupling
    coupled                   false;
    // If true, the simulation is transient
    transient                 yes;
    cellValueSourceCorrection on;

    sourceTerms
    {   // Source terms schemes
        schemes
        {
        }
    }
    // Interpolation schemes used when solving the particle motion
    interpolationSchemes
    {
        rho                   cell;
        U                     cellPoint;
        mu                    cell;
    }
    // Time integration scheme for solving the particles motion
    integrationSchemes
    {
        U                     Euler;
```

```
    }
}

constantProperties
{   // Id of the particle type
    parcelTypeId              1;
    // Minimal density of particle material per cells
    rhoMin                    1e-15;
    // Minimal mass of a particle
    minParticleMass           1e-15;
    // Density of the particle material
    rho0                      964;
    // Young's modulus of the particle material
    youngsModulus             6e8;
    // Poisson's ratio of the particle material
    poissonsRatio             0.35;

    constantVolume            false;
}

subModels
{   // List of the forces acting on the particles
    particleForces
    {
        sphereDrag;
        gravity;
    }
    // Injection model
    injectionModel            coneInjection;
    // Dispersion model / modification of the particles velocity due to the turbulence
    dispersionModel           none;
    // Interaction with the patches model
    patchInteractionModel     standardWallInteraction;
    // Heat transfer model
    heatTransferModel         none;
    // Surface film model
    surfaceFilmModel          none;
    // Collision between particles model
    collisionModel            none;
    // Radiation model
    radiation                 off;

// Sub-dictionary with the coefficients for the chosen model
    standardWallInteractionCoeffs
    {
        type                  rebound;
    }
    // Properties of the injection model (<InjectionModel>Coeffs)
    coneInjectionCoeffs
    {
        massTotal             0.0002;
        parcelBasisType       mass;
        SOI                   10; // Start Of Injection
        duration              3;
        // List of position and direction of the conical injectors
        positionAxis
        (
            (( -0.02 0.0 0.0 ) ( 1 0 0 ))
```

```
        );
        parcelsPerInjector    3000;
        parcelsPerSecond      1000;
        flowRateProfile       constant 0.01;
        Umag                  constant 0.13026;
        thetaInner            constant 0;
        thetaOuter            constant 30;

        sizeDistribution
        {
            type              RosinRammler;
            RosinRammlerDistribution
            {
                minValue      2.5e-05;
                maxValue      7.5e-05;
                d             5e-05;
                n             0.5;
            }
        }
    }
}
// Functions for the cloud particles (similar to functions in controlDict for the flow)
cloudFunctions
{}
```

Listing 9.3: Section of `kinematicCloudProperties`

In the `constant` folder also a file called `transportProperties` is included, which provides the air properties needed for particle cloud creation. As OpenFOAM® can only create cloud of particles for compressible flow, the density of the carrier flow has to be specified in the transport dictionary.

Create the solver

The exercise will be to create the solver for the von Karman street case.

Exercises

9.1 Create a new folder `kinematicParcelFoam` in your own solver folder. Copy the `.C`, `.H` file and the `Make` folder of the solver `pisoFoam` into this folder and rename the files and the phrase `pisoFoam` to `kinematicParcelFoam` everywhere in the code. Do not forget to modify the entries in the `files` file. Compile it.

9.2 Create an object of type `basicKinematicCollidingCloud`. Name it `kinematicCloud`. Refer to the code of the solver `$FOAM_SOLVERS/lagrangian/icoUncoupledKinematicParcelFoam`. Do not forget the header files needed for cloud definition. The constructor of the cloud object requires the density `rho` not existing in `pisoFoam`. Use the trick of `icoUncoupled KinematicParcelFoam` to create a density field based on a constant value read in the dictionary `transportProperties`.

9.3 Include the call in the main file of the solver for the particle transport function.

9.4 Use the `options` file given with the case for the compilation.

9.5 Start running the code for verification. Use the tutorial's example. It should be possible to eliminate the occurring errors applying the knowledge learned in the chapters before. Make yourself familiar with the additional files in the `constant` folder needed for cloud properties and injection definitions.

9.6 Before running the complete simulation, define appropriate probe location(s) and variables in order to determine the vortex shedding frequency. Start the simulation using the single-phase solution after 10 s as start condition (! This simulation requires a long time.)

9.7 For postprocessing animate simultaneously the *vorticity*[2] contours on a slice with $z = 0$ and the particle motion.

9.8 Compare the vortex shedding frequency of the simulation with the theoretical one. Use $\mathrm{Sr} = 0.216$ ($d = 0.01\mathrm{m}$).

9.6 Extra Practice and Background Information

Extra Practice

In this tutorial we considered a laminar particle-laden flow around a cylinder. What changes with turbulence? Assuming that we use RANS or LES there is always an amount of turbulent scales, which is smaller than the cell size. These fluctuations we can not resolve and have to model their impact on the large scale flow. In two-phase flows we have to consider also the effect of these turbulent scales on the particle motion and vice versa. The so called turbulent dispersion can be explained as the amount of the physical drag force which results from the interaction of the turbulent subgrid-scale velocity fluctuations with the particles. For RANS simulations the model of Gosman and Ionnides is often used, because it is computational cheap and uses the kinetic energy and dissipation rate which are provided by most of the RANS turbulent stress models. The idea is that particles cross several subgrid-scale eddies, which modify their trajectory. The resulting deviation from the mean velocity is determined by the time which a particle need to cross an eddy or by the eddy lifetime.

Within this tutorial we neglected particle-particle (four-way-coupling) since we use particles with low Stokes numbers which follow the flow almost exactly, and because we have a very low loading which makes particle-particle collisions not probable. The map shown in Figure 9.3 gives the different regimes of gas-particle and particle-particle interaction, dependent on the volume fraction and on the relation between particle response times and characteristic turbulence times.

In the framework of the Euler/Lagrange solver in OpenFOAM®, four-way coupling can be considered. Details, however, can be found only directly in the code.

Play around with the cloud-properties dictionary

- For a more homogeneous distribution of the particles a line injection would be better than the point behavior of the `ConeInjection`. Therefore we want to use the `KinematicLookupTableInjection`.This injection model allows us

[2] To compute the vorticity of a flow, execute the post-processor `vorticity` on the case.

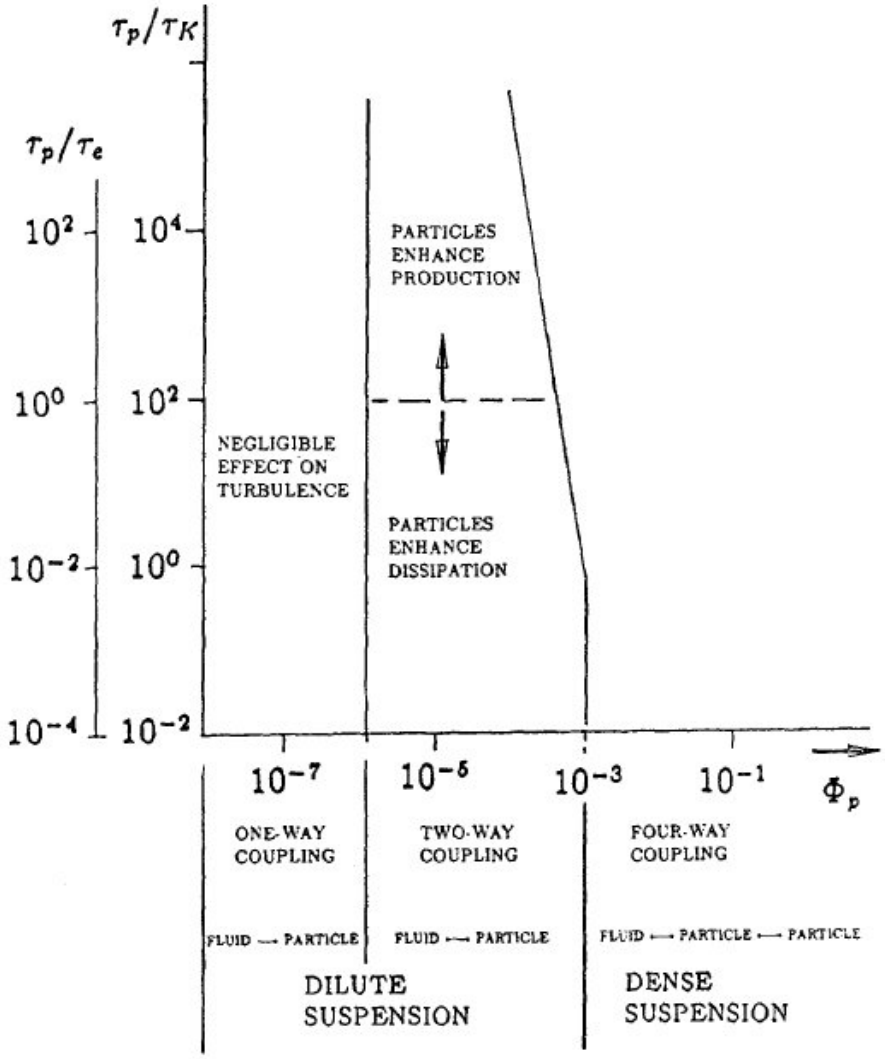


Figure 9.3: Conditions for the different types of particle-particle and particle-turbulence interaction.

to define more than one injection point and, in addition, each with individual velocity components and individual diameter. For application appropriate entries within the `kinematicCloudProperties` are necessary. Find out all entries needed and specify them by using the corresponding settings of the cone injection. Specify 5 injection points with x = -0.015 and y=-0.005, 0.0025, 0.0, 0.0025, 0.005 and u=0.13. Start the simulation again for a check of the settings. If you see only one injection stream, the flow rate of particles by time step is to low to add a particle at each injection point at each time step. To change that, switch `parcelBasisType` from `mass` to `number` and tune the parameter `parcelsPerSecond`.

Create the new injection model

The goal here is to create an injection model that has a certain number of injection points (specify in a file). And for each of them the particles introduced have a size distribution specified by a probability density function.
This injection model is close to `ManualInjection` model. So you will start from the code of that model.

Exercises

9.1 First copy the provided folder `MultipleInjection` in `$WM_PROJECT_USER_DIR/src/InjectionModel`. Then go with a terminal inside `MultipleInjection` and copy there `ManualInjection.H` and `ManualInjection.C` with the new name `MultipleInjection`. For that you will used advantageously the following command:

```
export KINEMATIC=$FOAM_SRC/lagrangian/intermediate/submodels/Kinematic
sed s/Manual/Multiple/g \
$KINEMATIC/InjectionModel/ManualInjection/ManualInjection.C \
> MultipleInjection.C
sed s/Manual/Multiple/g \
$KINEMATIC/InjectionModel/ManualInjection/ManualInjection.H  \
> MultipleInjection.H
```

Try to compile the class to check that the rename process is correctly done.

9.2 Remove the unused member `diameters_` from the header file and add the three following members:

```
//- Injection duration - common to all injection sources
const scalar duration_;

//- Number of parcels per injector - common to all injection sources
const label nParcelsPerSecond_;

//- Volume flow rate of parcels to introduce relative to SOI [m^3]
const autoPtr<DataEntry<scalar> > volumeFlowRate_;
```

Because a new class `DataEntry` is used, you have to *include* its definition `DataEntry.H` at the top of the header file and add before starting the definition of `template<class CloudType> class MultipleInjection` the following lines:

```
template<class Type>
class DataEntry;
```

9.3 Now go the source file and **comment the content** of the following functions: `parcelsToInject`, `volumeToInject`, the constructor, `timeEnd`, `setPositionAndCell` and `setProperties`. Then remove the initialization of `diameters_` in the constructor and add after the initialization of `sizeDistribution_` these lines to initialize the new members from the dictionary:

```
duration_(readScalar(this->coeffDict().lookup("duration"))),
nParcelsPerSecond_
(
    readScalar(this->coeffDict().lookup("parcelsPerSecond"))
),
injectorCells_(positions_.size()),
volumeFlowRate_
(
```

```
        DataEntry<scalar>::New
        (
            "volumeFlowRate",
            this->coeffDict()
        )
    )
```

Check the changes made up to now by compiling the class after having add the header file `DataEntry.H` at the list of included headers.

9.4 The constructor needs to be changed to initialize the total volume of particles injected. The implementation is:

```
// Determine volume of particles to inject
this->volumeTotal_ = 0.0;
this->volumeTotal_ = positions_.size()*volumeFlowRate_().integrate(0.0, duration_);
```

9.5 The next step is the new implementation of the functions. The action performed by each function is described here after. You can take inspiration from `KinematicLookupTableInjection` to figure out the implementation

- `parcelsToInject`: return the number of parcels to be injected between `time0` and `time1`.
- `volumeToInject`: return the volume of parcels to be injected between `time0` and `time1`. Hint: In the constructor, the total volume to be injected is computed.
- `timeEnd`: time at which the injection has to stop. Hint: `SOI_` is the Starting-Of-Injection time and `duration_` the total time during which the parcels are injected.
- `setPositionAndCell`: set the position and the cell in which the particle is injected. `injectorI`[3] is the index of the injector from which the particle enters the simulation, `position` the vector of initial position of the particle (and so the position of the injector), `cellOwner` the cell in which is the injector, `tetFaceI` the face of the injector tetrahedron face and `tetPtI` the injector tetrahedron point. Hint: the variable `nParcels` in `setPositionAndCell` is the second parameter appearing without name `const label,`. So to use it, write instead `const label nParcels,`.
- Finally the `setProperties` function has to be implemented. That function prescribes the velocity and the diameter of the injected particles. The former is given by the variable `U0` (as described in the code you commented). And as we want it to be random, the diameter `d` is calculated as:

  ```
  parcel.d() = sizeDistribution_().sample();
  ```

9.6 Change also the comment at the begin of the header file explaining the usage of the injection model. You can now compile your new injection model.

9.7 Test your new injection model on a copy of the previous case (reduce the duration of simulation to 1 s and not 3 s - you could also reduced the writing interval). The input file needed and the parameters are already available. So you only have to change the keyword `InjectionModel` in the dictionary `kinematicCloudProperties`.

---

[3] Have a look at `KinematicLookupTableInjection.C` to find how to compute this index.

Background Information

9.6.1 Add new models

Add to runtime

The configuration files of a test case are based on keyword and string entries. When you select a model, e.g. the k-epsilon model for the turbulence, OpenFOAM® checks if this model exists within a database such database are created for each top level model e.g. all turbulence model in RANS for incompressible flows. Consequently in addition to the implementation of a new model, you will have most of the time to add it to some database. This is done by calling some static function in the source code. The one creating a selection table (to store one type of model) is usually called `defineRunTimeSelectionTable(nameModel, dicitonary)`.
And the one adding a specific model to the table is usually
`addToRunTimeSelectionTable(nameModel, nameSpecificModel, dictionary)`.
For example to define a selection table for the RANS model, the function called is in `$FOAM_SRC/turbulenceModels/incompressible/RAS/RASModel/RASModel.C`:

```
defineRunTimeSelecetionTable(RASModel, dictionary)
```

And to add the sub-model *k-epsilon*, the following call is made in
`$FOAM_SRC/turbulenceModels/incompressible/RAS/kEpsilon/kEpsilon.C`

```
addToRunTimeSelectionTable(RASModel, kEpsilon, dictionary);
```

Add debug flag

As described in the User's Guide (3.2.5 Debug messaging and optimization switches), you can output more information or carry out more checks during the execution of your code by switching some flags in `$HOME/.OpenFOAM/$WM_PROJECT_VERSION/controlDict`[4]. Those flags turn on some parts of the code that usually print more information to the output and sometimes carry out more checks on the data. There are therefore very useful for debugging but not to run efficiently a test case.
The additional information are provided when the integer after the name of the class used is equal to 1 (0 turn off the output of the information).

When you implement a new model/class, you can easily add that ability to your new class. This is done by calling one static function usually called `defineTypeNameAndDebug(className, 0);`. The integer is the default value of the flag. So if zero, by default no additional information are output in the log file.
For example for the *k-epsilon* model:

```
defineTypeNameAndDebug(kEpsilon, 0);
```

Then to add some optional output in our code, you just have to surround them by

```
if(className::debug)
{
      Info << /*Optional output*/ << endl;
      // Additional tests
}
```

[4] To create a default list of debug flag, copy the file `$WM_PROJECT_DIR/etc/controlDict`

Some advanced examples

When you specify a model within a dictionary, OpenFOAM® checks if the name of the model is valid. Consequently when you develop a new model (e.g. a new injection model), the name of this new model has to be added to those lists. This is not done manually by editing some files. But directly by calling specific function within the libraries used for the current case (usually called `make`*`TypeOfModel`*). Without knowing it, you have already used a coupled of time this functionality; e.g. in the previous chapter you have created a new boundary condition that as to be added to list. For that the following lines are put at the end of `rainDropletInletVelocity`:

```
namespace Foam
{
   makePatchTypeField
   (
       fvPatchVectorField,
       rainDropletInletVelocityFvPatchVectorField
   );
}
```

Listing 9.4: Add a new boundary condition to the runTime

For a boundary condition, this is not too complicated. However due to that trick the compilation of a new model could be not so straightforward. As for the new injection model you can try to implement in the extra-practice: `MultipleInjection`.

```
MultipleInjection/makeBasicKinematicCollidingParcelSubmodels.C

LIB = $(FOAM_USER_LIBBIN)/libuserlagrangianIntermediate
```

Listing 9.5: `Make/files` to compile a new injection model.

If you look to `Make/files` (see listing 9.5), the compilation of the class, `MultipleInjection`, is not done directly. But it is a special function `makeBasicKinematicCollidingParcelSubmodels.C` that it is compiled. Then in the beginning of that file the header file `makeParcelInjectionModels.H` is included (see listing 9.6). In it the addition of the new injection model is carried out.

```
#include "basicKinematicCollidingCloud.H"

// Kinematic
#include "makeParcelInjectionModels.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
    makeParcelInjectionModels(basicKinematicCollidingCloud);
}
```

Listing 9.6: `makeBasicKinematicParcelSubmodels.C`

Indeed the new injection model will be compiled thanks to the inclusion at line 33 of `MultipleInjection.H` (see listing 9.7).

```
27 #ifndef makeParcelInjectionModels_H
28 #define makeParcelInjectionModels_H
29
```

```
30 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
31
32 #include "MultipleInjection.H"
33
34 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
35
36 #define makeParcelInjectionModels(CloudType)                                    \
37                                                                                  \
38     makeInjectionModelType
   \
39     (                                                                            \
40         MultipleInjection,                                                       \
41         CloudType                                                                \
42     );
43 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
44
45 #endif
```

Listing 9.7: `makeParcelInjectionModels.H`

To be able to figure out how to compile a specific type of model, you have to find the way is done for an existing model. For that, start by analyzing the `Make/files` file of the library containing the model to discover the compiled files. Then go upward in the hierarchy of the header files to find where your new model should appear.

Chapter

10 Moving Mesh

10.1 Todays problem

Up to now the computational domain was always fixed. This is sufficient for many basic investigations and for some real computations. But often a static domain is not sufficient. In case of an external influence on the domain that cause a geometric change of the area of interest (as in an internal combustion engine) the mesh has to be adapted to the actual geometry (the changing of the domain could also be caused by internal reasons like in an fluid flow engine, of course).

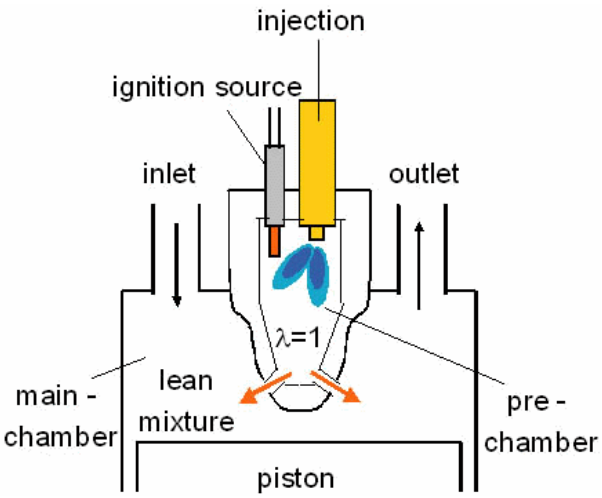


Figure 10.1: Sketch of an internal combustion engine with prechamber

For demonstration of a dynamically changing mesh we take the internal combustion engine discussed earlier 7.1. The simplifications done earlier are retained except the fixed geometry. So for repetition, we want to simulate the combustion in a prechamber internal combustion engine with the following simplifications:

- We do the simulation in 2D, not 3D
- on a relatively coarse grid.
- We use a very simple combustion model (the "Schmid Model")
- We also ignore the different stoichiometry in prechamber and main chamber.

Bibliography

[1] http://foam.sourceforge.net/doc/Guides-a4/UserGuide.pdf
OpenFOAM Programmers Guide, Version 1.6, 24th July 2009

[2] http://foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf
OpenFOAM Programmers Guide, Version 1.6, 24th July 2009

[3] H. Jasak: Dynamic Mesh Handling in OpenFOAM, AIAA-2009-341 *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, 2009

[4] J. H. Ferziger; M. Peric: *Computational methods for fluid dynamics*, Springer, 2end edition, 2001
[5] P. Moradnia: http://www.tfd.chalmers.se/h̃ani/kurser/OS_CFD_2007/ *A tutorial on how to use Dynamic Mesh solver IcoDyMFOAM* Spring 2008

10.2 Physics

The physics of the problem does not change compared to the ones told earlier. The usage of a mesh in any CFD-calculation is a simplification of the real object of interest. So a dynamically changing mesh, also called "moving mesh" is only a mater of simulation and has no physical background related to the physics of reality.

10.3 Numerics

Here we take a closer look on the effect of a dynamic mesh on the basic Equations of a CFD-simulation. Assuming that the coordinate system does not change during the simulation only the convective term will be affected by a relative velocity due to the mesh motion.
First, we consider a 1-D- continuity-equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho v)}{\partial x} = 0. \tag{10.1}$$

As we are working with the FVM-Method as most CFD-Codes does, we integrate this equation over a control volume with moving walls from $x_1(t)$ to $x_2(t)$:

$$\int_{x_1(t)}^{x_2(t)} \frac{\partial \rho}{\partial t} dx + \int_{x_1(t)}^{x_2(t)} \frac{\partial (\rho v)}{\partial x} dx = 0 \tag{10.2}$$

Considering the chain rule for the first term eq. 10.2 leads to:

$$\frac{d}{dt} \int_{x_1(t)}^{x_2(t)} \rho dx - \left[\rho_2 \frac{dx_2}{dt} - \rho_1 \frac{dx_1}{dt} \right] + \rho_2 v_2 - \rho_1 v_1 = 0 \tag{10.3}$$

Taking into account that $\frac{dx}{dt}$ represents the velocity of the moving mesh respectively the integration boarders, the term $\frac{dx}{dt}$ could be replaced by v_b, representing the boarder velocity. Therefor the term in brackets became similar to the last two terms and the eq: 10.3 yields:

$$\frac{d}{dt} \int_{x_1(t)}^{x_2(t)} \rho dx + \int_{x_1(t)}^{x_2(t)} \frac{\partial}{\partial x} \left[\rho (v - v_b) \right] dx = 0 \tag{10.4}$$

If the border velocity v_b is equal to the fluid velocity v the continuity equation reduce to the first term, that can be written in the Lagrange mode, $dm/dt = 0$
Writing the eq. 10.3 in 3D leads to

$$\frac{d}{dt} \int_V \rho dV - \int_S \rho \frac{d\mathbf{r}}{dt} \cdot \mathbf{n} \, dS + \int_S \rho \mathbf{v} \cdot \mathbf{n} \, dS = \mathbf{0} \tag{10.5}$$

respectively with the replacement of $\frac{dx}{dt}$ by v_b:

$$\frac{d}{dt} \int_V \rho dV + \int_S \rho (\mathbf{v} - \mathbf{v_b}) \cdot \mathbf{n} \, dS = 0 \tag{10.6}$$

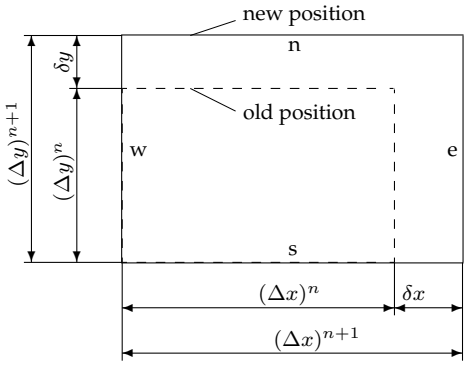


Figure 10.2: Control volume growing in time due to different mesh velocities in x- and y-direction

The same cane be done with the momentum equation. For example the integral mode of the i-th component of the momentum equations can be written as follows:

$$\frac{d}{dt}\int_V \rho u_i dV + \int_S \rho u_i(\mathbf{v}-\mathbf{v_b})\cdot\mathbf{n}\,dS = \int_S (\tau_{ij}\mathbf{i}_j - p\mathbf{i}_i)\cdot\mathbf{n}\,dS + \int_V b_i dV. \tag{10.7}$$

According to this example, the equation of each scalar value can be derived by replacing the velocity vector in the convective term by the relative velocity $\mathbf{v}-\mathbf{v}_b$.
So if the position of the mesh is known for every time as it is the case if the mesh movement is predefined, the solution of the Navier-Stokes-Equations constitute no new problems. The convective flux is simply calculated with a relative velocity on the borders of the control volume (CV). Therefore you need to know the Velocity of each side of the control Volume. And there you have to take care how to calculate them because the conservation of any conservative value e.g. mass is not necessarily satisfied by every kind of approximated mesh velocity. In the following an example will be shown, where mass conservation is not achieved and a solution of this problem will be presented.
Let us assume a rectangular control volume where the sides move with constant but different velocities as it is shown in picture 10.2. The fluid of interest should be incompressible and moves with a constant velocity. As an example, the continuity-equation is considered with an implicit Euler equation.

According to the implicit Euler integration over time the discretized continuity equation for the CV in fig. 10.2 yields

$$\begin{aligned}\frac{\rho[(\Delta V)^{n+1}-(\Delta V)^n]}{\Delta t} + \rho[(u-u_b)_e-(u-u_b)_w]^{n+1}(\Delta y)^{n+1} + \\ \rho[(v-v_b)_n-(v-v_b)_s]^{n+1}(\Delta x)^{n+1}=0,\end{aligned} \tag{10.8}$$

where u and v are velocity components of the fluid according to the cartesian system. Due to a constant fluid velocity like it was predefined, the fluid velocities crosses out. Therefore you get the following term:

$$\frac{\rho}{\Delta t}[(\Delta V)^{n+1}-(\Delta V)^n] - \rho(u_{b,e}-u_{b,w})(\Delta y)^{n+1} - \rho(v_{b,n}-v_{b,s})(\Delta x)^{n+1} = 0 \tag{10.9}$$

Because of a constant mesh velocity we can replace the difference of the velocity terms by the first derivation in time:

$$u_{b,e}-u_{b,w} = \frac{\delta x}{\Delta t},\ v_{b,n}-v_{b,s} = \frac{\delta y}{\Delta t} \tag{10.10}$$

Inserting this in Eq. 10.9 an considering that $(\Delta V)^{n+1} = (\Delta x \Delta y)^{n+1}$ and $(\Delta V)^n = [(\Delta x)^{n+1} - \delta x][(\Delta y)^{n+1} - \delta y]$ one recognize a mass source:

$$\delta \dot{m} = \frac{\rho \delta x \delta y}{\Delta t} = \rho(u_{b,e} - u_{b,w})(v_{b,n} - v_{b,s})\Delta t \tag{10.11}$$

If you analyze the right hand side, it is obvious that the mass production is zero if the mesh velocity in one direction is zero or the velocities on opposite sides are equal. In addition as the production rate is proportional to the time step, a small value of the latter will create an acceptable production rate. However the accumulation with the time can result in a problematic value.
The explicit Euler scheme faces the same problem, whereas the Crank-Nicholson discretization or the implicit discretization on three time levels would solve it correctly. But there are cases where even these methods produce artificial errors.
A possible solution for this problem is to add an equation called space conservation equation. It is close to a continuity equation with a density of one.

$$\frac{d}{dt}\int_V dV - \int_S \mathbf{v}_b \cdot \mathbf{n}\, dS = 0 \tag{10.12}$$

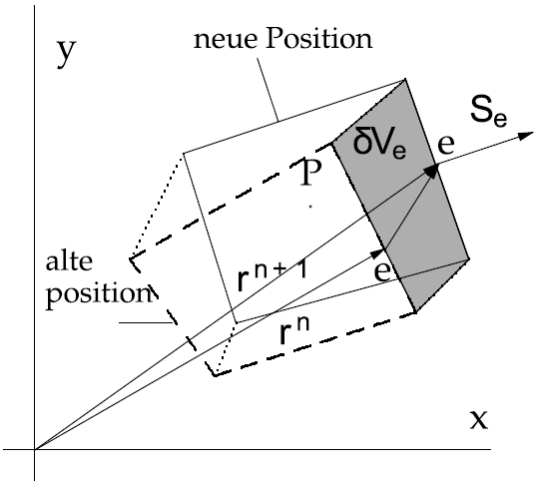


Figure 10.3: Figure of a CV with different velocities of the sides.

Starting again with the Euler method for time integration, the space conservation equation of the CV shown in Fig. 10.3 gets the following form:

$$\frac{(\Delta V)^{n+1} - (\Delta V)^n}{\Delta t} = \sum_c [(\mathbf{v}_b \cdot \mathbf{n})]_c^{n+1}\,,\ c = e, w, n, s \tag{10.13}$$

The difference in the control Volume over the time could also be described by sum of all partial changes in volume that occur for every side of the CV from timestep to timestep (see pic. 10.3).

$$\frac{(\Delta V)^{n+1} - (\Delta V)^n}{\Delta t} = \frac{\sum_c \delta V_c^{n+1}}{\Delta t} \tag{10.14}$$

Because the left hand side of both equations are equal also the right hand-side have to be equal. Using this the change in volume over time can be written as

$$\dot{V}_c^{n+1} = (\mathbf{v}_b \cdot \mathbf{n})_c S_c = \frac{\delta V_c^{n+1}}{\Delta t}. \tag{10.15}$$

Using this the mass flow through one side can be determined by the equation

$$\dot{m}_c^{n+1} = \int_{S_c^{n+1}} \rho(\mathbf{v} - \mathbf{v}_b) \cdot \mathbf{n} dS \approx [\rho(\mathbf{v} \cdot \mathbf{n})S]_c^{n+1} - (\rho \dot{V})_c^{n+1} \tag{10.16}$$

Going now back to the implicit Euler equation this discretization can be used for the connective term:

$$\frac{(\rho \Delta V)^{n+1} - (\rho \Delta V)^n}{\Delta t} + \sum_c \dot{m}_c^{n+1} = 0\,,\ c = e, w, n, s \tag{10.17}$$

For a sequential solver as the later used PISO-Solver, the mass fluxes in all conservation equations despite the continuity equation are presumed to be known. In such a case only the continuity equation has to be changed.

10.4 OpenFOAM®

10.4.1 Preparing the solver files

The solver described here is a modification of the solver developed in Chapter 7. As we have learned already its the easier way to start with an existing solver it is recommended to copy the solver *SchmidFoam* you have created earlier into your local solver directory (e.g. to *SchmidDyMFoam*). Then it is better to change the name of the C++ file into e.g. *SchmidDyMFoam* to avoid a mix up with the original file while editing. For that adapt `Make/files`. Then you must also change the `Make/options` file. For a dynamically changing mesh additional headers and libraries have to be used: `dynamicFvMesh` and `meshTools`. By executing the `wmake` command within the solver folder you can compile the solver for a first time. There should be no errors, only two warnings about not used variables which can be neglected.

10.4.2 Modifying the solver

Changes in the main file

Two major changes have to done to the solver: allow the mesh to move and the relative fluxes instead of the absolute ones. The new solver is presented in the listing (10.1). The first requirement is answered by including the header file `dynamicFvMesh.H` at the beginning that makes available the dynamic mesh utilities. Secondly we replace the line

```
#include "createMesh.H"
```

by

```
#include "createDynamicFvMesh.H"
```

to generate an non-static mesh.

Then the mesh is moved at each time step using the function:

```
mesh.update();
```

This function is called in front of every time step within the time loop.

The fluxes have to be absolute when setting the time step. So the function `makeAbsolute` is called. Then after some checks and before the equations are solved they are made relative using the function `makeRelative`.

```
#include "setRootCase.H"
#include "createTime.H"
// create a dynamic mesh and not a static one
//#include "createMesh.H"
#include "createDynamicFvMesh.H"                              // HERE
// Read values for hu, sL and yFuel
#include "readChemistryProperties.H"
#include "createFields.H"
#include "initContinuityErrs.H"

pimpleControl pimple(mesh);

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

```
while (runTime.run())
{
    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    // Make the fluxes absolute
    fvc::makeAbsolute(phi, rho, U);                          // HERE

    #include "setDeltaT.H"

    runTime++;

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

    // Update the mesh
    Info << "Update the mesh." << nl << endl;                // HERE
    bool meshChanged = mesh.update();                        // HERE

    if (meshChanged)                                         // HERE
    {                                                        // HERE
        thermo.correct();                                    // HERE
        #include "compressibleCourantNo.H"                   // HERE
    }                                                        // HERE

    // Make the fluxes relative to the mesh motion
    fvc::makeRelative(phi, rho, U);                          // HERE

    // Compute the source terms
    #include "burn.H"
    #include "rhoEqn.H"

    // --- Pressure-velocity PIMPLE corrector loop
    for (pimple.start(); pimple.loop(); pimple++)
    {
        if (pimple.nOuterCorr() != 1)
        {
            p.storePrevIter();
            rho.storePrevIter();
        }

        #include "UEqn.H"
        // Solve the transport of the regression variable
        #include "bEqn.H"
        #include "hEqn.H"

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

        if (pimple.turbCorr())
        {
            turbulence->correct();
        }
    }

    runTime.write();
```

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

Listing 10.1: `SchmidDyMFoam.C` file of `SchmidDyMFoam` solver.

Adapt the `pEqn.H` file

As the fluxes are computed also for the Poison equations we have to modify the `pEqn.H` file. The file should look like this (changes were made at the marked lines)

```
rho = thermo.rho();
rho = max(rho, rhoMin);
rho = min(rho, rhoMax);
rho.relax();

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

if (pimple.nCorr() <= 1)
{
    UEqn.clear();
}

if (pimple.transonic())
{
    surfaceScalarField phid
    (
        "phid",
        fvc::interpolate(psi)
       *(
            (fvc::interpolate(U) & mesh.Sf())
         // + fvc::ddtPhiCorr(rAU, rho, U, phi)                // HERE
        )
    );

          fvc::makeRelative(phid, psi, U);                          // HERE

    for (int nonOrth=0; nonOrth<=pimple.nNonOrthCorr(); nonOrth++)
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvm::div(phid, p)
          - fvm::laplacian(rho*rAU, p)
        );

        pEqn.solve
        (
            mesh.solver(p.select(pimple.finalInnerIter(corr, nonOrth)))
        );

        if (nonOrth == pimple.nNonOrthCorr())
        {
            phi == pEqn.flux();
        }
```

```
    }
}
else
{
    phi =
        fvc::interpolate(rho)*
        (
            (fvc::interpolate(U) & mesh.Sf())
          //+ fvc::ddtPhiCorr(rAU, rho, U, phi)                // HERE
        );

    fvc::makeRelative(phi, rho, U);                             // HERE

    for (int nonOrth=0; nonOrth<=pimple.nNonOrthCorr(); nonOrth++)
    {
        // Pressure corrector
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvc::div(phi)
          - fvm::laplacian(rho*rAU, p)
        );

        pEqn.solve
        (
            mesh.solver(p.select(pimple.finalInnerIter(corr, nonOrth)))
        );

        if (nonOrth == pimple.nNonOrthCorr())
        {
            phi += pEqn.flux();
        }
    }
}

#include "rhoEqn.H"
#include "compressibleContinuityErrs.H"

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

// Recalculate density from the relaxed pressure
thermo.correct();                                               // HERE
rho = thermo.rho();
rho = max(rho, rhoMin);
rho = min(rho, rhoMax);
rho.relax();
Info<< "rho␣max/min␣:␣" << max(rho).value()
    << "␣" << min(rho).value() << endl;

U -= rAU*fvc::grad(p);
U.correctBoundaryConditions();

DpDt = fvc::DDt(surfaceScalarField("phiU", phi/fvc::interpolate(rho)), p);
```

Listing 10.2: PISO loop for the moving mesh solver.

Now we can compile the solver as explained before.

Case setup

Now we start the case setup. The easiest way is again to copy the `SchmidFoam` case-directory to a new directory (e.g. `engineDyM`). The initial mesh geometry is the same for both cases but we have to put the lower boundary and the right boundary within separate wall patches respectively `piston` and `cylinderWall`. And for thermodynamically reasons the external wall of the cylinder becomes also a separate patch. To do this we change the `constant/polymesh/blockMeshDict` file.

The boundary values of the newly introduced patches must also be added at the initial conditions at the start time directory. They are the same as for the patch `wand` for almost all variables as they are no more than a split of the former wall.

The exception are the velocity and the temperature. The velocity of the lower patch (the `piston`) should still be $(0, 0, 0)$ relative to the reference of the piston. However the reference of the simulation is a viewer that see the piston moving. So for the simulation reference, the velocity on the wall is no more $(0, 0, 0)$ but equals to the wall velocity. To set that in OpenFOAM® you have to use the `movingWallVelocity` boundary condition :

```
piston
{
        type    movingWallVelocity;
        value   uniform (0 0 0);
}
```

For a better calculation the following start conditions will be used for the temperature:

```
    cylinderWall
    {
        type            wallHeatTransfer;
        alphaWall       uniform 100;
        Tinf            uniform 300;
        value           uniform 300;
    }
    wand
    {
        type            wallHeatTransfer;
        alphaWall       uniform 100;
        Tinf            uniform 300;
        value           uniform 300;
    }
    piston
    {
        type            wallHeatTransfer;
        alphaWall       uniform 100;
        Tinf            uniform 400;
        value           uniform 300;
    }
```

Listing 10.3: Temperature boundary conditions.

The procedure to create the mesh via using `subsetMesh` remains the same.

Definition of the mesh motion

The mesh motion is defined in a file called `dynamicMeshDict` in the `constant` folder. Here is an example of it:

```
FoamFile
{
    version     2.0;
    format      binary;
    class       dictionary;
    location    "constant";
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dynamicFvMesh   dynamicMotionSolverFvMesh;

motionSolverLibs ( "libfvMotionSolvers.so" );

solver          velocityComponentLaplacian y;

diffusivity     directional ( 1 1 0 );
```

Listing 10.4: `dynamicMeshDict` defining the piston motion.

The parameters of this dictionary will be now described.

dynamicFvMesh

There are two mesh manipulation approaches available which are defined in 2 different classes. The one uses for this case is the `dynamicFvMesh` model. This approach is useful for cases where the topology remains constant and the changes within the mesh stay small. The changes are done by squeezing or stretching the cells and by changing the node positions. Five different classes can be used with this approach:

- `staticFvMesh`:
  This class is identical to a solver without mesh motion. The `dynamicMeshDict` file does not need any further items.
- `dynamicMotionSolverFvMesh`:
  This class changes the mesh via squeezing or stretching of cells and movement of points. If you use this approach you will have to make sure, that the changes in the mesh does not corrupt the solution in an undesired way. In addition to the entry `dynamicFvMesh` you have to specify the `solver` and the `diffusivity` within the `dynamicMeshDict` file.
- `dynamicInkJetFvMesh`:
  Whereas the `dynamicMotionSolverFvMesh` is for a mesh deformation with a constant velocity along a specified direction the `dynamicInkJetFvMesh` class performs an oscillating movement along the X axis for all cells on the left side of `refPlaneX`[1]. Therefore also the `solver` and the `diffusivity` have to be specified as before. Additionally a subdictionary named `dynamicInkJetFvMeshCoeffs` has to be created in `dynamicMeshDict` with the coefficients

  ```
  amplitude (the amplitude of the motion)
  frequency (the frequency of the motion)
  refPlaneX (the x-Component of the reference plane)
  ```

[1] All cells on the right side will remain fixed

The x-coordinates of the points are scaled using the following relation: $X_{new} = X_{init}(1 + 0.5A\cos(2\pi ft))$

- `dynamicRefineFvMesh`:
  `dynamicRefineFvMesh` is similar to the `staticFvMesh` class. The only difference is that according to predefined field the mesh will be refined during the run according to the defined parameters. The `dynamicMeshDict` file should therefore look something like this:

```
dynamicRefineFvMeshCoeffs
{
    // How often to refine
    refineInterval  1;
    // Field to be refinement on
    field           alpha1;
    // Refine field inbetween lower..upper
    lowerRefineLevel 0.001;
    upperRefineLevel 0.999;
    // If value < unrefineLevel unrefine
    unrefineLevel   10;
    // Have slower than 2:1 refinement
    nBufferLayers   1;
    // Refine cells only up to maxRefinement levels
    maxRefinement   2;
    // Stop refinement if maxCells reached
    maxCells        200000;
    // Flux field and corresponding velocity field. Fluxes on changed
    // faces get recalculated by interpolating the velocity.
    correctFluxes
    (
        (phi    U)
    );
    // Write the refinement level as a volScalarField
    dumpLevel       true;
}
```

Listing 10.5: `dynamicMeshDict` to use with the class `dynamicRefineFvMesh`.

- `solidBodyMotionFvMesh`
  This class is used for complex calculation like Ship design Analysis (SDA) with a 3 degree of freedom (DoF) motion function or Sea Keeping Analysis (SKA) with a 6DoF motion function. This a special part of the area of moving mesh and is still a field in development.

The second approach of a moving mesh is done by the `topoChangerFvMesh`. Within this class the topology does not remain constant. This approach is used if the topology can not be kept at all or the simulation results would be affected in an unacceptable way. The class `polyTopoChanger` will search for the `meshModifiers` input file, an if it exists extract the necessary data out of it. Otherwise the data are read from the `dynamicMeshDict`. There are four sub-classes within this `topoChangerFvMesh` .which are listed below:

- `linearValveFvMesh` uses sliding meshes between the interfaces of two pieces of mesh in relative linear motion. For the parameters needed the dictionary `linearValveFvMeshCoeffs` is used to define the solver and the mesh handling utility. For more information see [5].

- linerValveLayersFvMesh is an extension of the previous mentioned one. In addition this class performs layer addition and removal. Therefore the extra dictionary layer is needed.
- mixerFvMesh allows the implementation of a case with a rotor/stator interaction. A number of coefficients like refinement values and solver have to specified in the dynamicMeshDict.
- movingConeTopoFvMesh is a utility for mesh manipulation like squeezing and stretching and also inserting and deleting cells. The coefficient for refining and inserting cells are declared in the dynamicMeshDict.

motionSolverLibs

The keyword motionSolverLibs specifies the library used by the manipulation approach. The keyword solver defines as the word already says the solver by which the motion equation is solved. In OpenFOAM® there are four solvers available.

- displacementLaplacian The equation of cell displacement are solved based on the Laplacian discretisation of the diffusivity and the cell displacement. Therefore the diffusivity model must be read from the dynamicMeshDict and the displacement from an extra file named pointDisplacement in the starting time folder. There you specify the final displacement of mesh components e.g moving walls and the displacement of the internal field, if this is possible. In general the file contains the following items:

```
 dimensions (determines the displacement dimensions)
 internalField (the displacement of the internal field
mesh)
 boundaryField (the displacement of the boundaries mesh)
```

  Additionally you can define a cellDisplacement file. The content is quiet the same but the types differ from pointVectorField within pointDisplacement to volVectorFiled within cellDisplacement. The cellDisplacement is not compulsory and can be neglected. By adding the vector name, e.g. y to the solver name we get displacementComponentLaplacian y an the equations are only solved in the y direction. The same has to be done with the pointDisplacement. So you get pointDisplacementy
- velocityLaplacian This solver is almost identical to the one mentioned previously. The only difference is in dealing with velocities instead of displacement. So a pointMotionU file is needed. Again there can be also be supplied a cellMotionU file although it is not necessary. The difference is again the type as told before. In the present case we use a component type of this solver velocityComponentLaplacian y (see above) together with a mointMotionUy file in the initial time directory, shown below:

```
 FoamFile
{
    version     2.0;
    format      ascii;
    class       pointScalarField;
    location    "0.000000";
    object      pointMotionUy;
}
// * * * * * * * * * * * * * * * * * * * * * * * * *  * * //
```

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

internalField   uniform 0;

boundaryField
{
    cylinderWall
    {
        type            slip;
    }
    wand
    {
        type            fixedValue;
        value           uniform 0;
    }
    piston
    {
        type            fixedValue;
        value           uniform 0.02;
    }
    wedge1
    {
        type            wedge;
    }
    wedge2
    {
        type            wedge;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

Listing 10.6: Boundary condition to define the motion of the boundaries.

As a motion velocity is specified without final position, you have to take care that your domain does not crash, due to a zero volume mesh.

- `LaplaceFaceDecomposition` This approach is used if the maximum order of the displacement is not known or is known to be very big. The mesh is rebuilt after a decomposition of all cells and faces and the Laplace smoothing equation is solved by the Finite Element Method. It is a very robust method but robustness is achieved by a high computational effort.
- `SBRStress` It is a displacement model, solving a diffusive equation and the cell displacement. It also considers the solid body rotation term in calculations. The `pointDisplacement` file must be supplied within the initial time directory.

As already mentioned the equations solving the mesh displacement are all based on a diffusion model. Therefore we have to specify how the motion spreads over the domain. These methods can be divided into quality-based and distance-dependent methods. Quality-based methods are

- `uniform`
- `directional ( , )`
- `motionDirectional ( , )`
- `inverseDistance`

The distance based methods can be split into

- `linear`
- `quadratic`
- `exponential ( )`

The brackets should symbolize the number of variables needed for this type of diffusivity model

For the `cellMotion` an discretization scheme has to be provided. It is specified in the `system/fvSchemes`file within the `laplacianSchemes` sub-dictionary.

```
laplacian(diffusivity,cellMotionU) Gauss linear uncorrected;
```

To solve the cell motion, the parameters of the solver have to be given in the `system/fvSolution` file by adding:

```
    cellMotionUy
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-08;
        relTol          0;
    }
```

Now the case can be run by executing `SchmidDyMFoam`.

10.5 Extra Practice and Background Information

Restricting the mesh movement

By looking at the mesh motion, you will see that the cells in the upper part of the cylinder skew. This behavior is not very good because the mesh at the prechamber opening is also skewed. This will disturbed the flame propagation. It would be better if the mesh movement is limited to the lower part of the domain. This can be done by changing the diffusivity in a appropriate way.

Piston movement

In a real engine the piston does not perform a movement with a constant velocity. The real piston movement can be better described by a oscillation function. Try create a case based on the previous one in which the piston is moving based on an oscillating function at 120 rpm. It is suggested that you use the `dynamicInkJetFvMesh` class for this case setup.

Chapter

11 Annexes

11.1 Paraview : some hints

11.1.1 Animation

In order to produce a movie of the simulation using Paraview, you have to do two steps:

1. First to click to *File -> Save animation*. A window will prompt you with a name e.g. `animation`. Write one and click *Ok*. Then Paraview will save a picture for each time step of the simulation with the name `animation_####.jpg`.
2. Then the pictures can be converted in a movie thank to the following command:

```
convert -monitor -quality 90 -delay 10 -antialias
                animation*jpg animation.mpg
```

The command `convert` is called with the following 4 options:

- `monitor`: print information about the actions done by the command
- `quality X`: compress the pictures using a quality *X* given in percent (between 1 and 100)
- `delay N`: add a delay of *N* ms between two pictures
- `antialias`: apply to the movie an anti-aliasing filter

11.1.2 Print nice picture

By default the color scheme of Paraview has a good contrast for a screen. However it is not adapted neither environment friendly to be printed. But that could be quickly change when you save a screenshot.

1. Click on *File->Save screenshot*
2. Change the *Palette* from *Current palette* to *Print*
3. Click on *Ok*.
4. Provide a name for the file and then click on *Ok*.

List of Figures

Listings

Index