OpenFOAM
A little User-Manual

Gerhard Holzinger
CD-Laboratory - Particulate Flow Modelling
Johannes Kepler University, Linz, Austria
`http://www.jku.at/pfm/`

16th July 2014

Abstract

This document is a collection of my own experience on learing and using OpenFOAM. Herein knowledge and background information is assembled that may be useful when learning to use OpenFOAM.

WARNING:

During the assembly of this manual OpenFOAM and other tools, e.g. *pyFoam*, have been continuously updated. This manual was started with OpenFOAM-2.0.x installed and at the time being the author works with OpenFOAM-2.2.x. Consequently it is possible that some facts or listings my be outdated by the time you read this. Furthermore, functionalities may have been extended or modified. Nevertheless, this manual is intended to cast some light on the inner workings of OpenFOAM and explain the usage in a rather practical way.

All informations contained in this manual can be found in the internet (`http://www.openfoam.org`, `http://www.cfd-online.com/Forums/openfoam/`) or they were gathered by trials and error (*What happens if ...?*).

Contents

1 Getting help

Apart from this manual, there are lots of resources on the internet to find help on OpenFOAM.

- The OpenFOAM User Guide
  `http://www.openfoam.org/docs/user/`
- The CFD Online Forum
  `http://www.cfd-online.com/Forums/openfoam/`
- The OpenFOAM Wiki
  `http://openfoamwiki.net/index.php/Main_Page`
  The OpenFOAM Wiki is maintained by a community of developers behind the OpenFOAM-extend project. This wiki covers not only the OpenFOAM but also tools that developed for OpenFOAM, e.g. *pyFoam* or *swak4foam*.
- The CoCoons Project
  `http://www.cocoons-project.org/`
  This is a community driven effort to create a documentation on solvers, utilities and modelling.
- The materials of the course CFD with open source software of Chalmers University
  `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/`
- The CAELinux Wiki
  `http://caelinux.org/wiki/index.php/Doc:OpenFOAM`
  CAELinux is a collection of open source CAE software including several CFD codes (OpenFOAM, Code_Saturne, Gerris, Elmer).

Part I
Installation

2 Install OpenFOAM

2.1 Prerequistes

OpenFOAM is easily installed by following the instructions from this website: `http://www.openfoam.org/download/git.php`.

First of all, you need to make sure all required packages are installed on your system. This is easily done via the package management software. OpenFOAM is a software made primarily for Linux systems. It can also be installed on Mac or Windows plattforms. However, the authors uses a Ubuntu-Linux system, therefore this manual will be based on the assumption that a Linux system is used.

```
sudo apt-get install git-core
sudo apt-get install build-essential flex bison cmake zlib1g-dev qt4-dev-tools libqt4-dev
    gnuplot libreadline-dev libxt-dev
sudo apt-get install libscotch-dev libopenmpi-dev
```

Listing 1: Installation of required packages

If OpenFOAM is to be used by a single user, then the User Manual suggests to install OpenFOAM in the `$HOME/OpenFOAM` directory.

2.2 Download the sources

First of all the source files need to be downloaded. This is done with the version control software *Git*. Afterwards we change into the new directory and check for updates. All steps to perform the described operations are listed in Listing 2.

```
cd $HOME
mkdir OpenFOAM
cd OpenFOAM
git clone git://github.com/OpenFOAM/OpenFOAM-2.1.x.git
cd OpenFOAM-2.1.x
git pull
```

Listing 2: Installation von *openFOAM*

Prior to compiling the sources some environment variables have to be defined. In order to do that a line (see Listing 3) has to added to the file `$HOME/.bashrc`.

```
source $HOME/OpenFOAM/OpenFOAM-2.1.x/etc/bashrc
```

Listing 3: Addition to *.bashrc*

When the command `source $HOME/.bashrc` is issued or when a new Terminal is opened this change is effective. Now with the defined environment variables OpenFOAM can be installed on the system. Before compiling a system check can be made by running *foamSystemCheck*.

```
user@host:~/OpenFOAM/OpenFOAM-2.1.x$ foamSystemCheck
Checking basic system... ------------------------------------
Shell:          /bin/bash
Host:           host
OS:             Linux version 2.6.32-39-generic
User:           user


System check: PASS
==================
Continue OpenFOAM installation.
```

Listing 4: *foamSystemCheck*

2.3 Compile the sources

If the system check produced to error messages then OpenFOAM can be compiled. This is done by executing `./Allwmake`. This is an installation script that takes care of all required operations. Compiling OpenFOAM can be done by using more than one processor to save time. In order to do this, an environment variable needs to be set before invoking `./Allwmake`. Listing 5 shows how to compile OpenFOAM using 4 processors.

```
export WM_NCOMPPROCS=4
./Allwmake
```

Listing 5: Parallel kompilieren

For working with OpenFOAM a user directory needs to be created. The name of this directory consists of the username and the version number of OpenFOAM. With version 2.1.x this folder needs to be named like this: `user-2.1.x`

2.4 Install paraView

paraView is a post processing tool, see `http://www.paraview.org/`. The OpenFOAM Foundation distributes *paraView* from its homepage and recommends to use this version. The source code can be downloaded from `http://www.openfoam.org/` in an archive, e.g. `ThirdParty-2.1.0.tgz`. This archive has to be unpacked into a folder named correspondingly to the OpenFOAM directory, e.g. `ThirdParty-2.1.x` when `OpenFOAM-2.1.x` is used. This naming scheme is mandatory because there is an environment variable that points to the location of *paraView*. As there is no development of *paraView* by the OpenFOAM developers, there is no repository release of third-party tools.

Subsequently *paraView* can be compiled by the use of an installation script. Afterwards some *plug-ins* for *paraView* need to be compiled.

```
cd $WM_THIRD_PARTY_DIR
./makeParaView

cd $FOAM_UTILITIES/postProcessing/graphics/PV3Readers
wmSET
./Allwclean
./Allwmake
```

Listing 6: Installation of *paraView*

2.5 Remove OpenFOAM

If OpenFOAM is to be removed from the system, then a few simple operations do the job[1], provided the installation was done following the installation guidelines of OpenFOAM[2].

Listing 7 shows how OpenFOAM can be removed from the system. We assume, we want to remove an installation of OpenFOAM-2.0.1. The first line changes the working directory to the installation directory of OpenFOAM. This folder contains all files of the OpenFOAM installation. Listing 8 shows the content of the `~/OpenFOAM`. In this example, two versions of OpenFOAM are installed.

The second line removes all files of OpenFOAM and the third line removes the files of the user related to OpenFOAM. The last line of Listing 7 removes a hidden folder. If there are several versions of OpenFOAM installed, then this folder should not be removed.

```
cd ~/OpenFOAM
rm -rf OpenFOAM-2.0.1
rm -rf user-2.0.1
cd
rm -rf ~/.OpenFOAM
```

Listing 7: Removing *OpenFOAM*

```
cd ~/OpenFOAM
ls -1
user-2.0.x
user-2.1.x
OpenFOAM-2.0.x
OpenFOAM-2.1.x
ThirdParty-2.0.x
ThirdParty-2.1.x
```

Listing 8: Content of `~/OpenFOAM`

Another thing to remove is the entry in the `.bashrc` file in the home directory. Delete the line shown in Listing 3.

2.6 Install several versions of OpenFOAM

It is possible to install several versions of OpenFOAM on the same machine. However due to the fact that OpenFOAM relies on some environment variables some precaution is needed. See `http://www.cfd-online.com/Forums/blogs/wyldckat/931-advanced-tips-working-openfoam-shell-environment.html` for detailed information about OpenFOAM and the Linux shell.

The most important fact about installing several versions of OpenFOAM is to keep the seperated.

3 Updating the repository release of OpenFOAM

3.1 Version management

OpenFOAM is distributed in two different ways. There is the *repository release* that can be downloaded using the *Git repository*. The version number of the repository release is marked by the appended x, e.g. OpenFOAM

[1] `http://www.cfd-online.com/Forums/openfoam-installation/57512-completely-remove-openfoam-start-fresh.html`
[2] `http://www.openfoam.org/download/git.php`

2.1.x. This release is updated regularly and is in some ways a development release. Changes and updates are released quickly, however, there is a larger possibility of bugs in this release. Because this release is updated frequently an OpenFOAM installation of version 2.1.x on one system may or will be different to another installation of version 2.1.x on an other system. Therefore, each installation has an additional information to mark different builds of OpenFOAM. The version number is accompanied by a hash code to uniquely identify the various builds of the repository release, see Listing 9. Whenever OpenFOAM is updated and compiled anew, this hash code gets changed. Two OpenFOAM installations are on an equal level, if the build is equal.

```
Build   : 2.1.x-9d344f6ac6af
```

Listing 9: Complete version identification of *repository releases*

Apart from the repository release there are also *pack releases*. These are upadated periodically in longer intervals than the repository release. The version number of a pack release contains no x, e.g. OpenFOAM 2.1.1. In contrast to the repository release all installations of the same version number are equal. Due to the longer release cycle the pack release is regarded to be less prone to software bugs.

There are several types of those releases. The are precompiled packages for widely used Linux distributions (Ubuntu, SuSE and Fedora) and also a source pack. The source pack can be installed on any system on which the source codes compile (usually all kinds of Linux running computers, e.g. high performance computing clusters, or even computers running other operation systems, e.g. Mac OSX[3] or even Windows[4]).

3.2 Check for updates

If OpenFOAM was installed from the repository release, updating is rather simple. To update OpenFOAM simply use *Git* to check if there are newer source files available. Change in the Terminal to the root directory of the OpenFOAM installation and execute `git pull`.

If there are newer files in the repository *Git* will download them and display a summary of the changed files.

```
user@host:~$ cd $FOAM_INST_DIR
user@host:~/OpenFOAM$ cd OpenFOAM-2.1.x
user@host:~/OpenFOAM/OpenFOAM-2.1.x$ git pull
remote: Counting objects: 67, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 44 (delta 32), reused 43 (delta 31)
Unpacking objects: 100% (44/44), done.
From git://github.com/OpenFOAM/OpenFOAM-2.1.x
  72f00f7..21ed37f  master     -> origin/master
Updating 72f00f7..21ed37f
Fast-forward
.../extrude/extrudeToRegionMesh/createShellMesh.C  |  10 +-
.../extrude/extrudeToRegionMesh/createShellMesh.H  |   7 +-
.../extrudeToRegionMesh/extrudeToRegionMesh.C      | 157 ++++++++-----
.../Templates/KinematicCloud/KinematicCloud.H      |   6 +-
.../Templates/KinematicCloud/KinematicCloudI.H     |   7 +
.../baseClasses/kinematicCloud/kinematicCloud.H    |  47 ++++++-
6 files changed, 193 insertions(+), 41 deletions(-)
```

Listing 10: There are updates available

If OpenFOAM is up to date, then *Git* will output a corresponding message.

```
user@host:~/OpenFOAM/OpenFOAM-2.1.x$ git pull
Already up-to-date.
```

Listing 11: OpenFOAM is up to date

[3]See `http://openfoamwiki.net/index.php/Howto_install_OpenFOAM_v21_Mac`
[4]See `http://openfoamwiki.net/index.php/Tip_Cross_Compiling_OpenFOAM_in_Linux_For_Windows_with_MinGW`

3.3 Check for updates only

If you want to check for updates only, without actually making an update, *Git* can be invoked using a special option (see Listings 12 and 13). In this case *Git* only checks the repository and displays its findings without actually making any changes. The option responsible for this is `--dry-run`. Notice, that `git fetch` is called instead of `git pull` [5].

```
user@host:~$ cd OpenFOAM/OpenFOAM-2.0.x/
user@host:~/OpenFOAM/OpenFOAM-2.0.x$ git fetch --dry-run -v
remote: Counting objects: 189, done.
remote: Compressing objects: 100% (57/57), done.
remote: Total 120 (delta 89), reused 93 (delta 62)
Receiving objects: 100% (120/120), 17.05 KiB, done.
Resolving deltas: 100% (89/89), completed with 56 local objects.
From git://github.com/OpenFOAM/OpenFOAM-2.0.x
   5ae2802..97cf67d  master     -> origin/master
user@host:~/OpenFOAM/OpenFOAM-2.0.x$
```

Listing 12: Check for updates only – updates available

```
user@host:~$ cd OpenFOAM/OpenFOAM-2.1.x/
user@host:~/OpenFOAM/OpenFOAM-2.1.x$ git fetch --dry-run -v
From git://github.com/OpenFOAM/OpenFOAM-2.1.x
 = [up to date]      master     -> origin/master
user@host:~/OpenFOAM/OpenFOAM-2.1.x$
```

Listing 13: Check for updates only – up to date

3.4 Install updates

After updates have been downloaded by `git pull` the changed source files need to be compiled in order to update the executables. This is done the same way as is it done when installing OpenFOAM. Simply call `./Allwmake` to compile. This script recognises changes, so unchanged files will not be compiled again. So, compiling after an update takes less time than compiling when installing OpenFOAM.

3.4.1 Workflow

Listing 14 shows the necessary commands to update an existing OpenFOAM installation. However this applies only for repository releases (e.g. OpenFOAM-2.1.x). The point releases (every version of OpenFOAM without an x in the version number) are not updated in the same sense as the repository releases. For simplicity an update of a point release (OpenFOAM-2.1.0 → OpenFOAM-2.1.1) can be treated like a complete new installation, see Section 2.6.

The first two commands in Listing 14 change to the directory of the OpenFOAM installation. Then the latest source files are downloaded by invoking `git pull`.

The statement in red can be omitted. However if the compilation ends with some errors, this command usually does the trick, see Section 3.5.2. The last statement causes the source files to be compiled. If `wclean all` was not called before, then only the files that did change are compiled. If `wclean all` was invoked then everything is compiled. This may or will take much longer.

If there is enough time for the update (e.g. overnight), then `wclean all` should be called before compiling. This will in most cases make sure that compilation of the updated sources succeeds.

```
cd $FOAM_INST_DIR
cd OpenFOAM-2.1.x
git pull
wclean all
./Allwmake
```

Listing 14: Update an existing OpenFOAM installation. The complete workflow

[5] `git pull` calls `git fetch` to download the remote files and then calls `git merge` to merge the retrieved files with the local files. So checking for updates is actually done by `git fetch`.

3.4.2 Trouble-shooting

If compilation reports some errors it is helpful to call `./Allwmake` again. This reduces the output of the successful operations considerably and the actual error messages of the compiler are easier to find.

3.5 Problems with updates

3.5.1 Missing packages

If there has been an upgrade of the operating system[6] it can happen, that some relevant packages have been removed in the course of the update (e.g. if these packages are only needed to compile OpenFOAM and the OS 'thinks' that these packages aren't in use). Consequently, if recompiling OpenFOAM fails after an OS upgrade, missing packages can be the cause.

3.5.2 Updated Libraries

When libraries have been updated, they have to be recompiled. Otherwise solvers would call functions that are not (yet) implemented. In order to avoid this problem the corresponding library has to be recompiled.

```
wclean all
```

Listing 15: Prepare recompilation with *wclean*

The brute force variant would be, to recompile OpenFOAM as a whole, instead of recompiling a updated library.

3.5.3 Updated sources fail to compile

In some cases, e.g. when there were changes in the organisation of the source files, the sources fail to compile right away. Or, if there is any other reason the sources won't compile and the cause is not found, then a complete recompilation of OpenFOAM may be the solution of choice. Although compiling OpenFOAM takes its time, this may take less time than tracking down all errors.

To recompile OpenFOAM the sources need to be reset. Instead of deleting OpenFOAM and installing it again, there is a simple command that takes care of this.

```
git clean -dfx
```

Listing 16: Reset the sources using *git*

The command listed in Listing 16 causes *git* to erase all files *git* does not track. That means all files that are not part of the *git*-repository are deleted. In this case, this is the official *git*-repository of OpenFOAM. *git clean* removes all files that are not under version control recursively starting from the current directory. The option `-d` means that also untracked folders are removed.

After the command from Listing 16 is executed, the sources have to be compiled as described in Section 2.3.

4 Install third-party software

The software presented in this section is optional. Without this software OpenFOAM is complete and perfectly useable. However, the software mentioned in this section can be very useful for specific tasks.

4.1 Install *pyFoam*

See `http://openfoamwiki.net/index.php/Contrib_PyFoam#Installation` for the instructions on the installation of *pyFoam*.

4.2 Install swak4foam

See `http://openfoamwiki.net/index.php/Contrib/swak4Foam` for instructions on installing *swak4foam*.

[6] An *upgrade* of an OS is indicated by a higher version number of the same (Ubuntu 11.04 → Ubuntu 11.10). An *update* leaves the version number unchanged.

4.3 Compile external libraries

There is the possibility to extend the functionality of OpenFOAM with additional external libraries, i.e. libraries for OpenFOAM from other sources than the developers of OpenFOAM. One example of such an external library is a large eddy turbulence model from `https://github.com/AlbertoPa/dynamicSmagorinsky`. The source code is stored in `OpenFOAM/AlbertoPa/`.

Such a library is compiled with `wmake libso`. This is also the case when libraries of OpenFOAM have been modified. The reason why typing `wmake libso` is sufficient is because all information *wmake* requieres is stored in the files `Make/files` and `Make/options`. These files tell *wmake* – and therefore also the compiler – where to find necessary libraries and where to put the executable. A more detailed description of this two files can be found in Sections 36.1.3 and 36.1.4.

To use an external library the solver needs to be told so. See Section 7.2.3.

```
cd OpenFOAM/AlbertoPa/dynamicSmagorinsky
wmake libso
```

Listing 17: Compilation of a library

Part II
General Remarks about OpenFOAM

5 Units and dimensions

Basically, OpenFOAM uses the International System of Units, short: SI units. Nevertheless, also other units can be used. In that case it is important to remember, that some physical constant, e.g. the universal gas constant, are stored in SI units. Consequently the values need to be adapted if other units that SI should be used.

5.1 Unit inspection

OpenFOAM performs in addition to its calculations also a inspection of the physical units of all involved variables and constants. For fields, like the velocity, or constants, like viscosity, the unit has to be specified. The unit is defined in the *dimension set*. Units in the International System of Units are defined as products of powers of the SI base units.

$$[Q] = \mathrm{kg}^{\alpha}\mathrm{m}^{\beta}\mathrm{s}^{\gamma}\mathrm{K}^{\delta}\mathrm{mol}^{\epsilon}\mathrm{A}^{\zeta}\mathrm{cd}^{\eta} \tag{1}$$

A dimension set contains the exponents of (1) that define the desired unit. With the dimension set OpenFOAM is able to perform unit checks.

```
dimensions        [0 1 -2 0 0 0 0];
```

Listing 18: False *dimensions* for U

```
--> FOAM FATAL ERROR:
incompatible dimensions for operation
    [U[0 1 -3 0 0 0 0] ] + [U[0 1 -4 0 0 0 0] ]

    From function checkMethod(const fvMatrix<Type>&, const fvMatrix<Type>&)
    in file /home/user/OpenFOAM/OpenFOAM-2.1.x/src/finiteVolume/lnInclude/fvMatrix.C at line
    1316.

FOAM aborting
```

Listing 19: *incompatible dimensions*

Listing 18 shows an incorrect definition of the dimension of the velocity, e.g. in the file `0/U`. m/s^2 has been defined instead of m/s. OpenFOAM recognises this false definition, because mathematical operations do not work out anymore. Listing 19 shows a corresponding error message produced by two summands having different units. Therefore, OpenFOAM aborts and displays an error message.

5.1.1 An important note on the base units

The order in which the base units are specified differs between OpenFOAM and many publications dealing with SI units, compare (2) and (3). The order of the base units as it is used by OpenFOAM swaps the first two base units. As the list of base units in [4, 3] starts with the metre followed by the kilogram, OpenFOAM reverses this order and begins with the kilogram followed by the metre. Also the fourth, fifth and sixth base units appear in a different position.

$$[Q]_{\mathrm{OpenFOAM}} = \mathrm{kg}^{\alpha}\mathrm{m}^{\beta}\mathrm{s}^{\gamma}\mathrm{K}^{\delta}\mathrm{mol}^{\epsilon}\mathrm{A}^{\zeta}\mathrm{cd}^{\eta} \tag{2}$$

$$[Q]_{\mathrm{SI}} = \mathrm{m}^{\alpha}\mathrm{kg}^{\beta}\mathrm{s}^{\gamma}\mathrm{A}^{\delta}\mathrm{K}^{\epsilon}\mathrm{mol}^{\zeta}\mathrm{cd}^{\eta} \tag{3}$$

Eq. (2) is based on the source code of OpenFOAM, see Listing 20. Eq. (3) is based on [4, 3].

```
//- Define an enumeration for the names of the dimension exponents
enum dimensionType
{
    MASS,                   // kilogram   kg
    LENGTH,                 // metre      m
    TIME,                   // second     s
    TEMPERATURE,            // Kelvin     K
    MOLES,                  // mole       mol
    CURRENT,                // Ampere     A
    LUMINOUS_INTENSITY      // Candela    Cd
};
```

Listing 20: The definition of the order of the base units in the file `dimensionSet.H`

The reason for changing the order of the base units may be motivated from a CFD based point of view. For fluid dynamics involving compressible flows as well as reactive flows and combustion the first five units of OpenFOAM's set of base units suffice.

5.1.2 Input syntax of units

Listing 21 shows the definition of a phase in a two-phase problem. Notice the difference between the first two definitions and the third one. The unit of `d` is defined by the full set of seven exponents, whereas the other two units (`rho` and `nu`) are defined only by five exponents. Apparently it is allowed to omit the last two exponents (defining candela and ampere).

Defining units with five entries (for kilogram, metre, second, kelvin and mol) seems to be perfectly appropiate. Whether the OpenFOAM User Guide [14] or the OpenFOAM Programmer's Guide [13] mention this behaviour. Defining a unit with an other number of values than five or seven leads to an error (see Listing 22).

```
phaseb
{
  rho            rho [ 1 -3 0 0 0 ] 1000;
  nu             nu [ 0 2 -1 0 0 ] 1e-06;
  d              d [ 0 1 0 0 0 0 0 ] 0.00048;
}
```

Listing 21: Definition of the unit

```
--> FOAM FATAL IO ERROR:
wrong token type - expected Scalar, found on line 22 the punctuation token ']'

file: /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bed/constant/transportProperties::
    phaseb::nu at line 22.

    From function operator>>(Istream&, Scalar&)
    in file lnInclude/Scalar.C at line 91.

FOAM exiting
```

Listing 22: Erroneous definition of units

5.2 Dimensionens

Fields in fluid mechanics can be scalars, vectors or tensors. There are in OpenFOAM different data types to distinguish between quantities of different dimension.

volScalarField A scalar field throughout the whole computaional domain, e.g. pressure.
volScalarField p

volVectorField A vector field throughout the whole domain, e.g. velocity.
volVectorField U

volTensorField A tensor field throughtout the whole domain, e.g. Reynolds stresses.
volTensorField Rca

surfaceScalarField A scalar field, defined on surfaces (surfaces of the finiten volumes), e.g. flux.
surfaceScalarField phi

dimensionedScalar A scalara constant throughout the whole domain (i.e. no field quantity).
dimensionedScalar nu

5.2.1 Dimension check

The data type defines also, as described before, the dimension of a quantity. The dimension of a quantity defines the syntax how quantities have to be entered.

Listing 24 shows the error message OpenFOAM displays when the value of a scalar quantity is entered as a vector (Listing 23).

```
dimensions          [ 0 0 0 0 0 0 0 ];
internalField       uniform ( 0 0 0 );
boundaryField
{
    inlet
    {
       type            fixedValue;
        value           uniform 0;
    }
```

Listing 23: Erroneous definition of α

```
--> FOAM FATAL IO ERROR:
wrong token type - expected Scalar, found on line 19 the punctuation token '('

file: /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bed/0/alpha::internalField at line
    19.

    From function operator>>(Istream&, Scalar&)
    in file lnInclude/Scalar.C at line 91.

FOAM exiting
```

Listing 24: Error message caused by invalid dimension

5.3 Kinematic viscosity vs. dynamic viscosity

To determine if OpenFOAM uses the kinematic viscosity [Ns/m^2 = Pas] or the dynamic viscosity [m^2/s] one has simply to take a look on the dimension.

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

Listing 25: *dimensions* of the viscosity

The type of viscosity is primarily determined by the used solver, e.g. compressible or incompressible.

5.4 Pitfall: pressure vs. pressure

The definition of pressure in OpenFOAM differs between the compressible and incompressible solvers. Compressible solvers work with the pressure itself. Incompressible solvers use a modified pressure. The reason for this is, because of $\rho = const$ the incompressible equations are divided by the density and to eliminate density entirely the modified pressure is introduced into the pressure term.

$$\hat{p} = \frac{p}{\rho} \tag{4}$$

For this reason the entries in the `0/p` files differ depending on the solver in use. This is visible by the unit of pressure.

5.4.1 Incompressible

The unit of the pressure in an incompressible solver is defined by (4)

$$[\hat{p}] = \frac{\mathrm{N}}{\mathrm{m}^2} \cdot \frac{\mathrm{m}^3}{\mathrm{kg}} = \mathrm{N}\frac{\mathrm{m}}{\mathrm{kg}} = \frac{\mathrm{kgm}}{\mathrm{s}^2} \cdot \frac{\mathrm{m}}{\mathrm{kg}} = \frac{\mathrm{m}^2}{\mathrm{s}^2} \tag{5}$$

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

Listing 26: Unit of pressure - incompressible

5.4.2 Compressible

The unit of the pressure in a compressible solver is the physical unit of pressure.

$$[p] = \frac{\mathrm{N}}{\mathrm{m}^2} = \frac{\frac{\mathrm{kgm}}{\mathrm{s}^2}}{\mathrm{m}^2} = \frac{\mathrm{kg}}{\mathrm{ms}^2} \tag{6}$$

```
dimensions      [ 1 -1 -2 0 0 0 0 ];
```

Listing 27: Unit of pressure - compressible

5.4.3 Pitfall: Pressure in incompressible multi-phase problems

When solving a multi-phase problem in an Eulerian-Eulerian fashion, for each phase a momentum equation is solved. In most cases it is assumed that the pressure is equal in all phases. For this reason the incompressible equations can not be divided by the density, because each phase has a different density and therefore, the modified pressure would be differnt for each phase. To avoid this issue, incompressible Euler-Euler solvers, like *bubbleFoam*, *twoPhaseEulerFoam* or *multiPhaseEulerFoam*, use the physical pressure like compressible solvers do.

6 Files and directories

OpenFOAM saves its data not in a single file, like Fluent does, it uses several different files. Depending on its purpose a specific file is located in one of several folders.

6.1 Required directories

An OpenFOAM case has a minimal set of files and directories. The directory that contains those folders is called the root directory of the case or case directory. Listing 28 shows the output of the commands `pwd` and `ls` when they are invoked from a case directory. The first command returns the absolute path of the current working directory. The second command prints the contents of the current folder. When `ls` is invoked without any options it returns the names of all non-hidden files and folders. In this case there are three subdirectories (*0*, *constant* and *system*). The fact that these three items are directories and not files is indicated by a different color. If `ls` is called with the option `-l` are more detailed list is printed. This detailed list indicates if an entry is a file or a directory.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ pwd
/home/user/OpenFOAM/user-2.1.x/run/icoFoam/cavity
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls
0  constant  system
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls -l
insgesamt 12
drwxrwxr-x 2 user group 4096 Okt  2 14:53 0
drwxrwxr-x 3 user group 4096 Okt  2 14:53 constant
drwxrwxr-x 2 user group 4096 Okt  2 14:53 system
```

Listing 28: Case directory

0 This is the first of the time-directories. It contains the initial and boundary conditions of all variable quantities. A case does not have to start at time $t = 0$. However, if there is no specific reason for a case to start at another time that $t = 0$, a case will always begin at time $t = 0$. The name of a time-directory is simply the number of elapsed seconds.

constant This folder contains all files dealing with constant quantities as well as the mesh.

polymesh This is a subdirectory of *constant*. In this folder all files defining the mesh reside.

system In this folder all files that control the solver or other tools are located

In the course of computing the case two kinds of folders are created. First of all, at defined times all information is written two the harddisk. A new time-directory is created with the number of elapsed seconds in its name. In this folder all kinds of files are saved. The number of files is equal or larger than in the *0*-directory containing the initial conditions.

The second category of directory subsumes all kinds of folders created for all kind of reasons or by all kind of tools, see Section 6.2 for a brief introduction to some of the more common of them.

6.2 Supplemental directories

Directories described in this Section may be created in the course of a computation.

6.2.1 *processor**

If a case is solved in parallel, i.e. the case is computed using more than one processor at the time. In this case the computational domain has to be decomposed into several parts, to divide the problem between the involved parallel processes. The tool that is used to decompose the case created the *processor**-directories. The * stands for a consecutive number starting with 0. So, if a case is to be solved using 4 parallel processes, then the domain has to be split into 4 parts. Therefore, the folders *processor0* to *processor3* are created.

Every one of the *parallel**-directories contains a *0*- and also a *constant*-directory containing only the mesh. The *system*-directory remains in the case folder. See Section 8.4 for more information about conducting parallel calculations.

6.2.2 functions

functions or functionObjects perform all kind of operations during the computation. Each function creates a folder of the same name to save its data in. See Section 24 for more information about functions.

6.2.3 *sets*

If the tool *sample* has been used, then all data generated by *sample* is stored in a folder named *sets*. See Section 25 for more information about *sample*.

6.3 Files in *system*

In the directory named *system* there are three files for controlling the solver. This files are necessary to run a simulation. Besides them there may also be additional files controlling other tools.

6.3.1 The main files

This files have to be present in the system folder to be able to run a calculation

controlDict This file contains the controls related to time steps, output interval, etc.

fvSchemes In this file the finite volume discretisation schemes are defined

fvSolution This files contains controls related to the mathematical solver, solver algorithms and tolerances.

6.3.2 Additional files

This list contains a selection of the most common files to be found in the system-directory.

probesDict Alternative to the use of the file *probesDict*, *probes* can also be defined in the file *controlDict*.

decomposeParDict Used by *decomposePar*. In this file the number of subdomains and the method of decomposition are defined.

setFieldsDict Necessary for the tool *setFields* to initialise field quantities.

sampleDict Definitions for the post-processing tool *sample*.

7 Control

Most of the controls of OpenFOAM are set in so called *dictionaries*. An important *dictionary* is the file *controlDict*.

7.1 Syntax

The dictionaries need to comply a certain format. The OpenFOAM User Guide states, that the dictionaries follow a syntax similar to the C++ syntax.

The file format follows some general principles of C++ source code.

The most basic format to enter data in a dictionary is the key-value pair. The value of a key-value pair can be any sort of data, e.g. a number, a list or a dictionary.

7.1.1 Keywords - the banana test

As OpenFOAM offers no graphical menus, in some cases allowed entries are not visible at a glance. If a key expects a value of a finite set of data, then the user can enter a value that is definitely not applicable, e.g. banana. Then OpenFOAM produces an error message with a list of allowed entries.

```
--> FOAM FATAL IO ERROR:
expected startTime, firstTime or latestTime found 'banana'
```

Listing 29: Wrong keyword, or the banana test

Listing 29 shows the error message that is displayed when the value *banana* is assigned to the key *startFrom* that controls at which time a simulation should start. The error message contains a note that is formated in this way: *expected X, Y or Z found ABC.*

If in a dictionary several key-value pairs are erroneous, only the first one produces an error, as OpenFOAM aborts all further operations.

Pitfall: assumptions & default values

In some cases the banana test behaves differently than expected. Listing 30 shows the warning message OpenFOAM returns, when the banana test is used with the control *compression* of *controlDict*. See Section 7.2.2 for a description of this control. In this case, OpenFOAM does not abort but continues to run the case. Instead of returning an error message and exiting, OpenFOAM simply assumes a value in place of the invalid entry.

```
--> FOAM Warning :
  From function IOstream::compressionEnum(const word&)
  in file db/IOstreams/IOstreams/IOstream.C at line 80
  bad compression specifier 'banana', using 'uncompressed'
```

Listing 30: Failed banana test

7.1.2 Mandatory and optional settings

Some settings are expected by the solver to be made. If they are not present, OpenFOAM will return an error message. Other settings have a default value, which is used if the user does not specify a value. In this sense, settings can be divided into mandatory and optional ones.

As mandatory settings causes an error if they are not set, a simulation can be run only if all mandatory settings were made.

About errors

- There will be an error when mandatory settings were not made.
- There is no error message if an optional setting (that is necessary) was omitted. All optional controls have a default value and will be in place.
- There is no error message if a setting was made and that setting is not needed. The solver simply ignores it. Consequently the definition of a variable time step in *controlDict* does not necessarily mean, that the simulation is performed with variable time steps, e.g. if *icoFoam* (a fixed time step solver) is used.
- Sometimes an error message points to the setting of a keyword that is actually not faulty. See Section 7.1.3.

See Section 34.3 for a detailed discussion – including a thorough look at some source code – about reading keywords from dictionaries.

7.1.3 Pitfall: semicolon (;)

Similar to C++, lines are terminated by a semicolon. Listing 31 shows the content of the file *U1* in the *0*-directory. The line defining the boundary condition (BC) for the outlet was not terminated properly. Listing 32 shows the provoked error message. This error message does not mention *outlet*, but rather *walls – keyword walls is undefined*. The definiton of the boundary condition for the walls comes after the outlet definition. One reason for this may be, that OpenFOAM terminates reading the file after the missing semicolon causes a syntax error, and therefore the boundary condition for the walls remain undefined.

This example demonstrates that the error messages are sometimes not very meaningful if they are taken literally. The error was made at the definiton of the BC for the outlet. If only the definition. of the BC of the walls is examined, the cause for the error message will remain unclear, because the BC definition of the walls is perfectly correct.

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

internalField     uniform (0 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (0 0 0.03704);
    }

    outlet
    {
        type            zeroGradient
    }

    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
}
```

Listing 31: Missing semicolon in the definition of the BC

```
--> FOAM FATAL IO ERROR:
keyword walls is undefined in dictionary "/home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam
    /case/0/U1::boundaryField"

file: /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/case/0/U1::boundaryField from line
    25 to line 47.

From function dictionary::subDict(const word& keyword) const
in file db/dictionary/dictionary.C at line 461.

FOAM exiting
```

Listing 32: Error message caused by missing semicolon

7.1.4 Switches

Besides key-value pairs there are switches. These enable or disable a function or a feature. Consequently, they only can have a logical value.

Allowed values are: *on/off, true/false* or *yes/no.* See Section 34.4.1 for a detailed discussion about valid entries.

7.2 *controlDict*

In this *dictionary* controls regarding time step, simulation time or writing data to hard disk are located.

The settings in the `controlDict` are not only read by the solvers but also by all kinds of utilities. E.g. some mesh modification utilities obey the settings of the keywords `startFrom` and `startTime`. This has to be kept in mind when using a number of utilities for pre-processing.

7.2.1 Time control

In this Section the most important controls with respect to time step and simulation time are listed. This list makes no claim of completeness.

startFrom controls the start time of the simulation. There are three possible options for this keyword.

- ***firstTime*** the simulation starts from the earliest time step from the set of time directories.
- ***startTime*** the simulation starts from the time specified by the `startTime` keyword entry.
- ***latestTime*** the simulation starts from the latest time step from the set of time directories.

startTime start time from which the simulation starts. Only relevant if `startFrom   startTime` has been specified. Otherwise this entry is completely ignored[7].

stopAt controls the end of the simulation. Possible values are *{endTime, nextWrite, noWriteNow, writeNow}*.

- **endTime** the simulation stops when a specified time is reached.

endTime end time for the simulation

deltaT time step of the simulation if the simulation uses fixed time steps. In a variable time step simulation this value defines the initial time step.

adjustTimeStep controls whether time steps are of fixed or variable length.[8] If this keyword is omitted, a fixed time step is assumed by default.

runTimeModifiable controls whether or not OpenFOAM should read certain dictionaries (e.g. *controlDict*) at the beginning of each time step. If this option is enabled, a simulation can be stopped by using setting `stopAt` to one of these values *{nextWrite, noWriteNow, writeNow}*.

[7] If the simulation is set to start from *firstTime* or *latestTime*, this keyword can be omitted or the value of this keyword can be anything – `startTime banana` does not lead to an error, what would be the case if the simulation started from a specific start time.

[8] This keyword is important only for solvers featuring variable time stepping. A fixed time step solver simply ignores this control without displaying any warning or error message.

7.2.2 Data writing

In *controlDict* the controls regarding data writing can be found. Often, it is not necessary to save every time step of a simulation. OpenFOAM offers several ways to define how and when the data is to be written to the hard disk.

writeControl controls the timing of writing data to file. Allowed values are *{adjustableRunTime, clockTime, cpuTime, runTime, timeStep}*.

- **runTime** when this option is chosen, then every `writeInterval` seconds the data is written.
- **adjustableRunTime** this option allows the solver to adjust the time step, so that every `writeInterval` seconds the data can be written. Otherwise the times at which data is written does not exactly match the entry in `writeInterval`. I.e. for a 1 s interval the data is written at $t = 1.0012, 2.0005, \ldots$ s.
- **timeStep** the data is written every `writeInterval` time steps.

writeInterval scalar that controls the interval of data writing. This value gets its meaning from the value assigned to *writeControl*.

writeFormat controls how the data is written to hard disk. It is possible to write text files or binary files. Consequently, the options are *{ascii, binary}*.

writePrecision controls the precision of the values written to the hard disk.

writeCompression controls whether to compress the written files or not. By default compression is disabled. When it is activated, all written files are compressed using *gzip*.

timeFormat controls the format that is used to write the time step folders.

timePrecision specifies the number of digits after the decimal point. The default value is 6.

7.2.3 Loading additional Libraries

Additional libraries can be loaded with an instruction in *controlDict*. Listing 33 shows how an external library (in this case a turbulence model that is not included in OpenFOAM) is included. This model can be found at`https://github.com/AlbertoPa/dynamicSmagorinsky/`.

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

Listing 33: Load additional libraries; *controlDict* entry

7.2.4 *functions*

functions, or *functionObjects* as they are called in OpenFOAM, offer a wide variety of extra functionality, e.g. probing values or run-time post-processing. See Section 24.

7.2.5 Outsourcing a *dictionary*

Some definitions can be outsourced in a seperate *dictionary*, e.g. the definition of a *probe-functionObject*.

All inclusive

In this case the *probe* is defined completely in *controlDict*.

```
functions
{
  probes1
  {
    type probes;
    functionObjectLibs ("libsampling.so");

    fields
```

```
        (
          p
          U
        );
        outputControl    outputTime;
        outputInterval   0.01;

        probeLocations
        (
          (0.5  0.5  0.05)
        );
    }
}
```

Listing 34: Definition of a *probe* in *controlDict*

Seperate *probesDict*

In this case the definition of the *probe* is done in a seperate file – the *probesDict*. In *controlDict* the name of this dictionary is assigned to the keyword *dictionary*. This dictionary has be located in the *system-directory* of the case. It is not possible to assign the path of this dictionary to this keyword.

```
functions
{
    probes1
    {
        type probes;
        functionObjectLibs ("libsampling.so");

        dictionary probesDict;
    }
}
```

Listing 35: External definition of *probes*; Entry in *controlDict*

```
fields
(
    p
    U
);

outputControl    outputTime;
outputInterval   0.01;

probeLocations
(
    (20.5  0.5  0.05)
);
```

Listing 36: Definition of *probes* in the file *probesDict*

Everything external

There is also the possibility to move the whole definition of a functionObject into a seperate file. In this case the macro `#include` is used. This macro is similar to the pre-processor macro if C++.

```
functions
{
    #include "cuttingPlane"
}
```

Listing 37: Completely external definition of a *functionObject*; Entry in `controlDict`

```
cuttingPlane
{
  type              surfaces;
  functionObjectLibs ("libsampling.so");
  outputControl     outputTime;

  surfaceFormat     raw;
  fields            ( alpha1 );

  interpolationScheme cellPoint;

  surfaces
  (
    yNormal
    {
      type              cuttingPlane;
      planeType         pointAndNormal;
      pointAndNormalDict
      {
        basePoint         (0 0.1 0);
        normalVector      (0 1 0);
      }

      interpolate       true;
    }
  );
}
```

Listing 38: Definition of a *cuttingPlane functionObject* in a seperate file named `cuttingPlane`

7.3 Run-time modifcations

If the switch *runTimeModifiable* is set *true*, *on* or *yes*; certain files (e.g. *controlDict* or *fvSolution*) are read anew, if a file has changed. In this way, e.g. the write interval can be changed during the simulation. If OpenFOAM detects a run-time modification it issues a message on the Terminal.

```
regIOobject::readIfModified() :
  Re-reading object controlDict from file "/home/user/OpenFOAM/user-2.1.x/run/
    multiphaseEulerFoam/bubbleColumn/system/controlDict"
```

Listing 39: Detected modifaction of *controlDict* at run-time of the solver

7.4 *fvSolution*

The file `fvSolution` contains all settings controlling the solvers and the solution algorithm. This file must contain two dictionaries. The first controls the solvers and the second controls the solution algorithm.

7.4.1 Solver control

The `solvers` dictionary contains settings that determine the work of the solvers (e.g. solution methods, tolerances, etc.).

7.4.2 Solution algorithm control

The dictionary controlling the solution algorithm is named after the solution algorithm itself. I.e. the name of the dictionary controlling the PIMPLE algorithm is `PIMPLE`. Note, that the name of this dictionary is in upper case letters unlike most other dictionaries.

Listing 40 shows an example of a `PIMPLE` dictionary. See Section 19.2 for a detailed discussion on the PIMPLE algorithm.

```
PIMPLE
{
    nOuterCorrectors  1;
    nCorrectors       2;
    nNonOrthogonalCorrectors  0;
    pRefCell          0;
    pRefValue         0;
}
```

Listing 40: The PIMPLE dictionary

7.5 Pitfalls

7.5.1 *timePrecision*

If the time precision is not sufficient, then OpenFOAM issues a warning message and increases the time precision without aborting a running simulation.

Listing 41 shows such a warning message. The simulation time exceeded 100 s and OpenFOAM figured that the time precision was not sufficient anymore.

```
--> FOAM Warning :
    From function Time::operator++()
    in file db/Time/Time.C at line 1024
    Increased the timePrecision from 6 to 13 to distinguish between timeNames at time 100.001
```

Listing 41: Warning message: automatic increase of time precision

A side effect of this increase in time precision was a slight offset in simulation time. The time step of this simulation was 0.001 s and the time steps were written every 0.5 s. As it is clearly visible in Listing 42, the names of the time step folders indicate this offset. This effect on the time step folder names was the reason, the automatic increase of time precision was noticed by the author.

However, automatic increase of time precision has no negative effect on a simulation. This purpose of this section is to explain the cause for this effect.

```
101.5000000002
101.0000000002
100.5000000002
100
99.5
99
98.5
```

Listing 42: Time step folders after increase of time precision

8 Usage of OpenFOAM

8.1 Use OpenFOAM

In the most simple case, Listing 43 represents a complete simulation-run.

```
blockMesh
checkMesh
icoFoam
paraFoam
```

Listing 43: Compute a simple simulation case

The first command, *blockMesh*, creates the mesh. The geometry has to be defined in *blockMeshDict*. *checkMesh* performs, as the name suggests, checks on the mesh. The third command is also the name of the solver. All solvers of OpenFOAM are invoked simply by their name. The last command opens the post-processing tool ParaView.

There are additional tasks that extend the sequence of commands shown in Listing 43. These can be

- Convert a mesh created by an other meshing tool, e.g. import a Fluent mesh

- Initialise fields
- Set up an parallel simulation; see Section 8.4

8.1.1 Redirect output and save time

The solver output can be printed to the Terminal or redirected to a file. Listing 44 shows how the solver output is redirected to a file named *foamRun.log*.

```
mpirun -np N icoFoam -parallel > foamRun.log
```

Listing 44: Redirect output to a file

Redirecting the solver output does not only create a log file, it also save the time that is needed to print the output to the Terminal. In some cases this can reduce simulation time drastically. However, writing to hard disk also takes its time.

Time steps	Cells	Print to Terminal		Redirect to file	
		executionTime	clockTime	executionTime	clockTime
5000	400	6,36	9	4,6	6
10000	400	12,71	18	9,22	10
12500	400	15,8	23	11,54	12
25000	400	32,33	47	22,99	23
5000	1600	9,74	11	9,3	10
5000	6400	282,19	283	282,83	283

Table 1: Run-time *cavity* test case

executionTime is the time the processor takes to calculate the solution of the case. *clockTime* is the time that elapses between start and end of the simulation, this is the time the wall clock indicates. The value of the *clockTime* is always larger than the value of the *executionTime*, because computing the solution is not the only task the processor of the system performs. Consequently, the value of the *ClockTime* depends on external factors, e.g. the system load.

Redirect output to nowhere

If the output of a program is of no interest it can be redirected to virtually nowhere to prevent it from being displayed on the Terminal. Listing 45 shows haw this is done. `/dev/null` is a special file on unix-like systems that discards all data written to it.

```
mpirun -np N icoFoam -parallel > /dev/null
```

Listing 45: Redirect output to nowhere

8.1.2 Run OpenFOAM in the background, redirect output and read log

In Section 8.1.1 the redirection of the solver output was explained. To monitor the progress of running calculation the end of the log can be read with the *tail* command.

Listing 46 shows how a simlation with *icoFoam* is started and the solver output is redirected. The `&` at the end of the line causes the invoked command to be executed in the background. The Terminal remains therefore available. Otherwise the Terminal would be waiting for *icoFoam* to finish before executing any further commands.

The second command invoked in Listing 46 prints the last 5 lines of the log file to the Terminal. *tail* returns the last lines of a text file. Without the parameter `-n` *tail* returns by default the last 10 lines.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ icoFoam > foamRun.log &
[1] 10416
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ tail foamRun.log -n 5
ExecutionTime = 0.74 s  ClockTime = 1 s

Time = 1.12

Courant Number mean: 0.444329 max: 1.70427
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$
```

Listing 46: Read redirected output from log file while the solver is running

8.1.3 Save hard disk space

OpenFOAM saves the data of the solution in intervals in time directories. The name of a time directory represents the time of the simulation. Listing 47 shows the content of a case directory after the simulation has finished. Besides the three folders that define the case (*0*, *constant* and *system*) there are more time directories and a *probes1*-folder present.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls
0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1  constant  probes1  system
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$
```

Listing 47: List folder contents

The *probes1*-directory contains the data generated by the functionObject named probes1. The time-directories contain the solution data of the whole computational domain. Listing 48 shows the contents of the *0*- and the *0.1*-directory. Typically, time-directories generated in the course of the computation contain more data than the *0*-directory defining the initial conditions.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavityBinary$ ls 0
p  U
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavityBinary$ ls 0.1
p  phi  U  uniform
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavityBinary$
```

Listing 48: List folder contents

Using binary files or compressing files

In general the time-directories use the majority of the hard disk space a completed case takes. If the time-directories are saved in binary instead of ascii format, these use generally a little less space. Another advantage of storing time step data in binary format, the time step data has full precision.

OpenFOAM also offers the possibility to compress all files in the time step directories. For compression OpenFOAM uses *gzip*, this is indicated by the files names in the time step directories, i.e. `alpha1.gz` instead `alpha1`.

Table 2 shows a comparison of hard disk use. The most reduction is achieved by compressing ascii data files. However, storing the time step data in ascii has the disadvantage that the numerical precision is limited to the number of digits stated with the `writePrecision` keyword in the `controlDict`. In this case `writePrecision` was set to 6, i.e. numbers have up to 6 significant digits. Compressing the binary files shows less effect than compressing the ascii files, which indicates that the binary files contain less redundant bytes.

Write settings	Used space	reduction	
`ascii`	45.5 MB		
`ascii`, `compressed`	16.7 MB	28.8 MB	-63.3 %
`binary`	33.8 MB	11.7 MB	-25.7 %
`binary`, `compressed`	28.8 MB	16.7 MB	-36.7 %

Table 2: Comparison of hard disk space consumption

Make sure to avoid unnecessary output

Disk space can easily be wasted by writing everything to disk. Not only writing too many time steps to disk can waste space, *functionObjects* can be the culprit too. See 24.4.3.

8.2 Abort an OpenFOAM simulation

8.2.1 Terminate a process in the foreground

If a command is executed in the Terminal without any additional parameters the process runs in the foreground. The Terminal is therefore busy and can not be used until the process is finished. When a process is running in the foreground it can easily terminated by pressing CTRL+C. Listing 49 features the GNU command `sleep`. The only function of this command is to pause for a specified amount of time. With this command the permature termination of a process can be tried.

```
user@host:~$ sleep 3
user@host:~$
```

Listing 49: Keep the Terminal busy

8.2.2 Terminate a background process

If a process runs in the background, the Terminal is free to be used for further tasks while the process is running. In this case, the background process can not be terminated by pressing CTRL+C because the Operating System can not tell which background process the user wants to terminate.

Identify the process

On UNIX based systems every process is identified by a unique number. This is the PID, the **p**rocess **id**entifier. The PID is equivalent to a licence plate for a car. During run-time this number is unique. However, after a process has finished the PID of this process is available for other, later processes.

To find out which processes are currently running, invoke the command *ps*. This lists all running processes. Without any further parameters only the processes that were executed from the current Terminal are listed. Listing 50 shows the result if a new Terminal is opened and *ps* is called. The first entry – `bash` – is the Terminal itself. The second entry – ps – is the only other process active at the time *ps* looks for all running processes. The PID is listed in the first column of Listing 50. Depending on the parameters passed to *ps* the output can be formatted differently.

```
user@host:~$ ps
  PID TTY          TIME CMD
13490 pts/1    00:00:00 bash
13714 pts/1    00:00:00 ps
user@host:~$
```

Listing 50: List processes in a fresh Terminal

The output of 50 is rather dull. However, there are lots of parameters telling *ps* what to do. The option `-e` makes *ps* list all systemwide running processes. The output of such a call can be quite long, because *ps* lists all processes started by the users as well as all system processes[9].

The option `-F` controls the output format of *ps*. In this case `-F` stands for *extra full*. This means the output contains a lot of information. Another option to display much information is `-l`. This option truncates the names of the processes to 15 characters, whereas `-F` displays not only the full name of the process, it also displays the parameters with which the processes were called.

```
ps -eF
```

Listing 51: List all running processes of the system

ps displays much information about a process. For terminating a process only the PID is necessary.

[9]System processes are processes run by the Operating System itself.

Search in the list of processes

The output of ps is a list which can be quite long. To terminate a certain process its PID has to be known. Searching a number in a list of numbers can be quite painful and errorprone. Therefore it would be handy to search in the list *ps* has returned for the desired process.

Before all else, *grep* does the trick. And now for something more detailed. *grep* is a program that searches the lines of its input for a certain pattern. *grep* can use a file or the standard input as its input. As it is unpractical to redirect the output of *ps* into a file only for *grep* to read it, we directly redirect the output of *ps* to the input of *grep*. This is achieved by the use of a pipe.

Listing 52 shows how this is done. The first part of the command invoked – `ps -eF` – calls ps to list all processes currently running in great detail. The option `-F` is used to make sure long process names can be distinguished, e.g. to tell ***buoyantBoussinesq****PimpleFoam* apart from ***buoyantBoussinesq****SimpleFoam*. Both are standard solvers of OpenFOAM. The bold part are the first 15 characters of the solver's name. If the option `-F` was omitted and both solvers were running, the results of *ps* would be ambiguous.

The second part of the command invoked in Listing 52 shows the call of *grep*. *grep* can be called with one or two arguments. If only one argument is passed to *grep*, *grep* uses the standard input as input. If *grep* is called with two parameters, the second argument has to specify the file from which *grep* has to read. As *grep* is called with only one argument, it reads from the standard input.

Because it would be even more boring to type the list returned by ps we redirect the output of ps to the standard input of grep. This is done by the pipe. The character `|` marks the connection of two processes in the Terminal. The command left of the `|` passes its output directly to the command specified right of the `|`.

Now we can read and interpret Listing 52. It shows the output of the search for all running processes containing the pattern `Foam`. In this case a parallel computation is going on. The first line of the result is *mpirun*. This process controls the parallel running solvers. The next four lines are the four instances of the solver. How parallel simulation works is explained in Section 8.4. The second last entry of the result is *grep* waiting for input[10]. The last line of the result is the pdf viewer which displays this document at that time. This example shows that is important to choose the pattern wisely, the search may return unexpected results.

```
user@host:~$ ps -ef | grep Foam
user    11005  5117  0 17:11 pts/2     00:00:05 mpirun -np 4 twoPhaseEulerFoam -parallel
user    11006 11005 99 17:11 pts/2     00:40:27 twoPhaseEulerFoam -parallel
user    11007 11005 99 17:11 pts/2     00:40:28 twoPhaseEulerFoam -parallel
user    11008 11005 99 17:11 pts/2     00:40:27 twoPhaseEulerFoam -parallel
user    11009 11005 99 17:11 pts/2     00:40:26 twoPhaseEulerFoam -parallel
user    11673 11116  0 17:52 pts/12    00:00:00 grep --color=auto Foam
user    32041     1  0 Aug01 ?         00:00:31 evince /tmp/lyx_tmpdir.J18462/lyx_tmpbuf0/open
    FoamUserManual_CDLv2.pdf
user@host:~$
```

Listing 52: Search for processes

List only specified processes

You can tell *ps* directly in which processes you are interested. The option `-C` of *ps* makes *ps* list only those processes that stem from a certain command. Listing 53 shows the output when `ps -C twoPhaseEulerFoam` is typed into the Terminal. In this case also there are four parallel processes running. Notice, that only the processes directly related to the solvers are shown. No other results are displayed unlike in Listing 52.

One has to bear in mind, that `ps -C` does not search for patterns. If the command name passed to *ps* as an argument is misspelled, *ps* will not display the desired result. Listing 54 shows the effect of typos in this case. The truncation of the process name in the list does not affect the search if the passed command name is equal or longer than the truncated process name. The first two commands issued in Listing 54 result in a list of all running instances of the solver. If the passed argument is shorter than the truncated process name – the third command – *ps* does not output any results. Also if there is a typo in the passed argument, *ps* does not find anything.

```
user@host:~$ ps -C twoPhaseEulerFoam
  PID TTY          TIME CMD
```

[10]On most Unix-like systems processes connected by a pipe are started at the same time. For this reason *grep* is already running while *ps* is listing all running processes.

```
11006 pts/2    00:47:44 twoPhaseEulerFo
11007 pts/2    00:47:44 twoPhaseEulerFo
11008 pts/2    00:47:44 twoPhaseEulerFo
11009 pts/2    00:47:43 twoPhaseEulerFo
user@host:~$
```

Listing 53: List all instances of *twoPhaseEulerFoam*

```
user@host:~$ ps -C twoPhaseEulerFoa
  PID TTY          TIME CMD
12741 pts/0    00:00:34 twoPhaseEulerFo
12742 pts/0    00:00:34 twoPhaseEulerFo
12743 pts/0    00:00:34 twoPhaseEulerFo
12744 pts/0    00:00:34 twoPhaseEulerFo
user@host:~$ ps -C twoPhaseEulerFo
  PID TTY          TIME CMD
12741 pts/0    00:00:36 twoPhaseEulerFo
12742 pts/0    00:00:36 twoPhaseEulerFo
12743 pts/0    00:00:36 twoPhaseEulerFo
12744 pts/0    00:00:36 twoPhaseEulerFo
user@host:~$ ps -C twoPhaseEulerF
  PID TTY          TIME CMD
user@host:~$ ps -C twPhaseEulerFoa
  PID TTY          TIME CMD
```

Listing 54: List all instances of *twoPhaseEulerFoam* – the effect of typos

Terminate

The operating system interacts with running processes using signals. The user can also send signals to processes using the command *kill*. *kill* sends by default the termination signal. To identify the process to which the signal is to be sent, the PID of this process has to be passed as an argument.

Listing 55 shows how the programm sleep is executed, all running processes are listed, the running instance of sleep is terminated and the running processes are listed again. When *ps* was executed the second time, a message is displayed stating the process has been terminated[11]. If the process would not have been terminated the message at the “natural” end of the process would be like in Listing 56[12].

```
user@host:~$ sleep 20 &
[1] 13063
user@host:~$ ps
  PID TTY          TIME CMD
12372 pts/0    00:00:00 bash
13063 pts/0    00:00:00 sleep
13064 pts/0    00:00:00 ps
user@host:~$ kill 13063
user@host:~$ ps
  PID TTY          TIME CMD
12372 pts/0    00:00:00 bash
13065 pts/0    00:00:00 ps
[1]+  Beendet                 sleep 20
user@host:~$
```

Listing 55: Terminate a process using *kill*

```
user@host:~$ sleep 1 &
[1] 13126
user@host:~$ ps
  PID TTY          TIME CMD
12372 pts/0    00:00:00 bash
13127 pts/0    00:00:00 ps
[1]+  Fertig                  sleep 1
user@host:~$
```

[11] On other systems this message is displayed immediately – see Listing 57. In this case the procedure was tried on the local computing cluster.

[12] A system with English language setting the message would read `Terminated` if the process would have been terminated and `Done` if the process would have been allowed to finish.

Listing 56: The natural end of a process

```
user@cluster user> sleep 10 &
[1] 31406
user@cluster user> kill 31406
user@cluster user>
[1]    Terminated                sleep 10
user@cluster user>
```

Listing 57: Terminate a process using *kill* on a different machine

8.3 Continue a simulation

If a simulation has ended at the end time or if it has been aborted there may be the need to continue the simulation. The most important setting to enable a simulation to be continued has to be made in the file `controlDict`. There, the keyword `startFrom` controls from which time the simulation will be started.

The easiest way to continue a simulation is to set the `startFrom` parameter to `latestTime`. Then, if necessary, the value of `endTime` needs to be adjusted. After this changes, the simulation can be continued by simply invoking the solver in the Terminal.

8.4 Do parallel simulations with OpenFOAM

OpenFOAM is able to do parallel simulations. There is no great difference between calculating a case with one single process or using many parallel processes. The only obvious additional task is to split the computation domain into several pieces. This step is called *domain decomposition*. After the domain is decomposed several instances of the solver are running the case on a subdomain each. Additionally, the invokation of the solver differs from the single process case.

8.4.1 Starting a parallel simulation

To enable a simulation using several parallel instances of a solver, OpenFOAM uses the MPI standard in the implementation of OpenMPI. OpenMPI ensures that all parallel instances of the solver run synchronously. Otherwise the simulation would generate no meaningful results. In order to be able to manage all parallel processes the simulation has to started using the command *mpirun*.

Listing 58 shows how a parallel simulation using 4 parallel processes is started. The solver outputs are redirected into a file called `> foamRun.log` and the simulation runs in the background of the Terminal. So the same Terminal can be used to monitor the progress of the calculation. See Section 8.1.2 for a discussion about running a process in the background.

The output message in the Listing shows the PID of the running instance of *mpirun*. This PID can be used to terminate the parallel calculation, like it is explained in Section 8.2.2.

```
user@host:~$ mpirun -np 4 icoFoam -parallel > foamRun.log &
[1] 11099
user@host:~$
```

Listing 58: Run OpenFOAM with 4 processes

The number of processes, in this case 4, has to be equal the number of *processor** folders. These folders are created by *decomposePar* and their number is defined in *decomposeParDict*. See Section 8.4.2 for information about domain decomposition.

If this numbers – the number of *processor** folders and the number of parallel processes with which mpirun is invoked – are not equal OpenFOAM issues an error message similar to Listing 59. In this case the domain was decomposed into 4 subdomains and it was tried to start the parallel simulation with 2 processes. If the parallel simulation is called with too many processes, OpenFOAM issues an error message like in Listing 60. The first example shows, that OpenFOAM reacts differently whether the parallel job was started with loo little or too many processes.

```
[0] --> FOAM FATAL ERROR:
[0] "/home/user/OpenFOAM/user-2.1.x/run/icoFoam/cavity/system/decomposeParDict" specifies 4
    processors but job was started with 2 processors.
```

Listing 59: Run OpenFOAM with too little parallel processes

```
[0] --> FOAM FATAL ERROR:
[0] number of processor directories = 4 is not equal to the number of processors = 8
```

Listing 60: Run OpenFOAM with too many parallel processes

Pitfall: `-parallel`

The parameter `-parallel` is important. If this parameter is omitted, the solver will be executed n times. Listing 61 shows the output of the command *ls* when it is run with *mpirun* with two processes. In this case *ls* is simply run twice.

If the parameter `-parallel` is missing, the same happens as in the case of *ls*. The simulation is run by n processes at roughly the same time. Listing 62 shows the first lines of output of a situation where the `-parallel` parameter was omitted. All solvers start the calculation of the whole case and write their output to the Terminal. The output appears on the Terminal in the order as it is generated by the solvers – in other words, the output on the Terminal is completely disarranged. If the `-parallel` parameter is missing, there is also no check if the *processor** folders are present.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ mpirun -np 2 ls
0  constant  system
0  constant  system
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$
```

Listing 61: Run *ls* using 2 processes

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ mpirun -np 4 icoFoam
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 2.1.x-6e89ba0bcd15
Exec   : icoFoam
Date   : Jan 29 2013
Time   : 10:51:12
Host   : "host"
PID    : 25622
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 2.1.x-6e89ba0bcd15
Exec   : icoFoam
```

Listing 62: Run *icoFoam* without the `-parallel` parameter

Pitfall: domain decomposition

If there was no domain decompositin prior to starting a parallel simulation, OpenFOAM will issue an corresponding error message.

```
[0] --> FOAM FATAL ERROR:
[0] twoPhaseEulerFoam: cannot open case directory "/home/user/OpenFOAM/user-2.1.x/run/
    twoPhaseEulerFoam/testColumn/processor0"
[0]
[0]  FOAM parallel run exiting
```

Listing 63: Missing *domain decomposition*

Pitfall: domain resonstruction

After a parallel simulation has ended, all data is residing in the *processor** folders. If *paraView* is started – without prior domain reconstruction – *paraView* will only find the data of the 0 directory.

8.4.2 Domain decomposition

Before a parallel simulation can be started the domain has to be decomposed into the correct number of subdomains – one for each parallel process. The parallel processes calculate on their own subdomain and exchange data of the border regions at the end of each time step. This is also the reason why the parallel processes have to be synchonous. Otherwise, processes with a lower computational load would overtake other processes and they would exchange data from different times.

Just before starting the simulation the domain has to be decomposed. The tool *decompsePar* is used for this purpose. Other operations, e.g. initialising fields using *setFields* have to take place before the domain decomposition. *decomposePar* reads from *decomposeParDict* in the *system* directory. This file has to contain al least the number of subdomains and the decomposition method.

decomposePar creates the *processor** directories in the case directory. Inside the *processor** folders a *0* and a *constant* folder are created. The *0* folder contains the initial and boundary conditions of the subdomain and the *constant* folder contains a *polyMesh* folder containing the mesh of the subdomain.

All parallel processes read from the same *system* directory, as the information stored there is not affected by the domain decomposition. Also the files in the *constant* directory are not altered.

Pitfall: Existing decomposition

If the domain has already been decomposed and *decomposePar* is called again, e.g. because the number of subdomains has been changed or some fields have been reinitialised, OpenFOAM issues an error message. Listing 64 shows an example. In this case the domain has already been decomposed into 2 subdomains and the attempt is made to decompose it again. OpenFOAM always issues an error message, whether the number of subdomains has changes or not.

The resulting error message proposes two possible solutions. The first is to invoke *decomposePar* with the `-force` option to make *decomposePar* remove the *processor** folders before doing its job. The second proposed solution is to manually remove the *processor** folders. In this case the error message contains the proper command to do so. The user can retype the command or copy and paste it into the Terminal.

```
--> FOAM FATAL ERROR:  Case is already decomposed with 2 domains, use the -force option or
    manually
remove processor directories before decomposing. e.g.,
    rm -rf /home/user/OpenFOAM/user-2.1.x/run/icoFoam/cavity/processor*
```

Listing 64: Already decomposed domain

Time management with *decomposePar*

In the course of an update of OpenFOAM decompose gained the option `-time`. This enhancement took place between the release of OpenFOAM 2.1.0 and OpenFOAM 2.1.1. Such enhancements typically first appear in the respository release OpenFOAM 2.1.x. So, it may be, that some installations of OpenFOAM 2.1.x contain this feature and some not depending on the time of installation or the time of the last update.

The option time lets the user specify a time from which or a time range in which the domain is to be decomposed. Listing 65 shows some examples of how this option works.

The option `-latestTime` makes *decomposePar* use the latest time step as starting time step for the subdomains.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls
0  0.1  0.2  constant  probes1  processor0  processor1  processor2  system
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ decomposePar -time 0.1:0.2 -force > /dev/
    null
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls processor0
0.1  0.2  constant
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ decomposePar -time 0.2 -force > /dev/null
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls processor0
0.2  constant
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$
```

Listing 65: Time management with *decomposePar*

8.4.3 Domain reconstruction

To be able to look at the results the data has to be reassembled again. This job is done by *reconstructPar*. This tool collects all data of the *processor** folders and reconstructs the original domain using all the generated time step data. After *reconstructPar* has finished the data of the whole domain resides in the case directory and the data of the subdomains resides in the *processor** folders.

Listing 66 shows the content of the case directory after a parallel simulation has finished. The first command is a simple call of ls to display the contents of the case directory. This is not different from the situation before the parallel simulation was started with the exception of the log file. However, this log file could be from a previous run. So, listing the contents after a parallel simulation has finished carries no real information.

The second command lists the contents of the *processor0* directory. In this directory – as well as in all other *processor** folders – there is time step data. The third command reconstructs the domain. After this tool has finished, the case directory also contains time step data. The last command lists the contents of the *processor0* folder again. This data has not been removed. So, a finished parallel case stores its time step data twice and therefore uses a lot of space.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls
0  constant  foamRun.log  probes1  processor0  processor1  processor2  processor3  system
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls processor0
0  0.1  0.2  0.3  0.4  0.5  constant
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ reconstructPar > foamReconstruct.log &
[1] 26269
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls
0  0.1  0.2  0.3  0.4  0.5  constant  foamReconstruct.log  foamRun.log  probes1  processor0
    processor1  processor2  processor3  system
[1]+  Fertig                    reconstructPar > foamReconstruct.log
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$ ls processor0
0  0.1  0.2  0.3  0.4  0.5  constant
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/cavity$
```

Listing 66: A finished parallel simulation

Time management

If a simulation has been startet from $t = t_1$ the domain has to be reconstructed for times $t > t_1$. Calling *reconstructPar* without any options regarding time, the program starts reconstructing the domain at the earliest time. To prevent the tool from reconstructing already reconstructed time steps the `-time` option can be used. Listing 67 shows how simulation results are reconstructed for $t \leq 60\,\mathrm{s}$.

```
reconstructPar -time 60:
```

Listing 67: Zeitparameter für *reconstructPar*

Another option to reconstruct only the new time steps is the command line option `-newTimes`. By using this option the proper time span to reconstruct is automatically determined.

8.4.4 Run large studies on computing clusters

Simulating parallel on a machine brings some advantages and enables the user to run even large simulations on a workstation. However, if the cases is very large, or parametric studies are to be conducted, using the workstation can be counter productive. Therefore, simulating on a computing cluster is the method of choice for large scale calculations. The user can follow a two step method.

1. Set up the case and run some test simulations, e.g. for a small number of time steps, on the workstation to ensure the simulation runs
2. Do the actual simulation on the cluster

The fact, that OpenFOAM runs on a great number of platforms enables the user to do simulations on the workstation as well as on a big cluster with tens or hundreds of processors.

Run OpenFOAM using a script

Section 42.5 explaines how to set up a script that runs multiple cases.

8.5 Using tools

OpenFOAM consists besides of solvers of a great collection of tools. These tools are used for all kind of operations.

All solvers and tools of OpenFOAM[13] assume that they are called from the case directory. If an executable is to be called from another directory the path to the case diretory has to be specified. Then the option `-case` has to be used to specify this path.

Listing 68 shows the error message displayed by the tool *fluentMeshToFoam* as it was executed from the *polyMesh* directory. The tool added the relative path `system/controlDict` to the currect working directory. This resulted in an invalid path to *controlDict* as the error message tells the user. Actually, the error message states that the file could not be found. This does not solely imply an invalid path. The file could simply be missing.

```
--> FOAM FATAL IO ERROR:
cannot find file

file: /home/user/OpenFOAM/user-2.1.x/run/icoFoam/testCase/constant/polyMesh/system/controlDict
     at line 0.

    From function regIOobject::readStream()
    in file db/regIOobject/regIOobjectRead.C at line 73.

FOAM exiting
```

Listing 68: Wrong path

The correct usage of the `-case` option is shown in Listung 69. There the correct path to the case directory – two levels upwards – is specified using `../..`[14]

```
user@host:~/OpenFOAM/user-2.1.x/run/icoFoam/testCase/constant/polyMesh$ fluent3DMeshToFoam -
    case ../.. caseMesh.msh
```

Listing 69: Specify the correct path to the case

[13]No exeption known to the author.

[14]On most Linux or Unix systems `.` refers to the current directory and `..` refers to the directory above the current one. To change in the Terminal one directory upwards on Linux `cd ..` does the job and on MS-DOS or Windows `cd..` is the proper command.
Also, on Linux systems the tilda refers to the home directory of the current user.

Part III

Pre-processing

9 Geometry creation & other pre-processing software

There are many ways to create a geometry. There is a great number of CAD software, there is a number of CFD pre-processors capable of creating geometries and there is the good old *blockMeshDict*.

This section is about the different ways to generate a geometry for a subsequent CFD simulation.

9.1 *blockMesh*

blockMesh is OpenFOAMs own pre-processing tool. It is able to create the domain geometry and the corresponding mesh. See Section 11 for a discussion on *blockMesh*. For the reason of simplicity all aspects of *blockMesh* – geometry creation as well as meshing – are covered in Section 11.

9.2 CAD software

There is a great number of CAD software around. Each CAD program usually uses its own file format. However most CAD programs support exporting the geometry in different formats, e.g. STL, IGES, SAT. If CAD software is used to create the geometry the data has to be exported to be used by a meshing program. A common file format for this purpose is the STL format. *snappyHexMesh* can be used with STL[15] geometry definitions.

9.2.1 OpenSCAD

OpenSCAD [`http://www.openscad.org/`] is an open source CAD tool for creating solid 3D CAD models. A CAD model is created by using primitve shapes (cubes, cylinders, etc.) or by extruding 2D paths. Models are not created interactively like in other CAD software. The user writes an input script which is interpreted by OpenSCAD. This makes it easy to create parametric models.

For further information on usage see the documentation `http://en.wikibooks.org/wiki/OpenSCAD_User_Manual`.

Pitfall: STL mesh quality

OpenSCAD is a tool to create CAD models. Therefore the requirements on the produced STL mesh are completely different than on a mesh for CFD simulations. OpenSCAD produces STL meshes that define the geometry correctly but the mesh is of a bad quality from a CFD point of view.

Figure 1 shows the STL mesh of a circular area. All triangles defining the circular area share one vertex. This vertex is probably the base point for the mesh creation of OpenSCAD. From a CFD point of view the triangular face elements are highly distorted and have a bad aspect ratio. However from a CAD point of view these triangles are prefectly sufficient to represent the circular area.

If a finite volume mesh is to be derived from the STL surface mesh (e.g. with GMSH) problems may arise. If the only purpose of the STL mesh is to represent some geometry – like it is the case with *snappyHexMesh* – then this quality issues can be ignored.

[15] STL is infact a surface mesh enclosing the geometry. Therefore the term STL mesh or STL surface mesh is also valid.

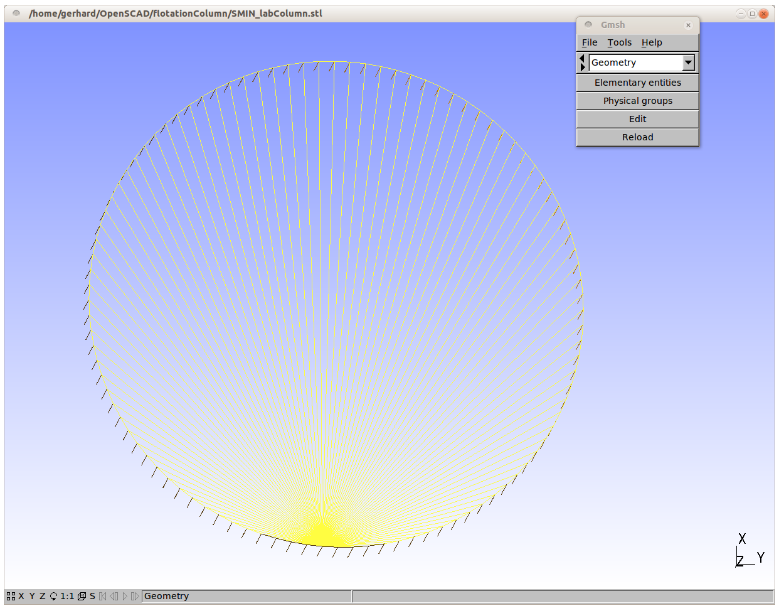


Figure 1: The STL mesh of a circular area generated by OpenSCAD

9.3 Salome

Salome [`http://www.salome-platform.org/`] is a powerful open source pre-processing software developed by EDF. Salome can be used to create a geometry interactively or by interpreting a python script[16]. Salome comes with a number of internal and external meshing utilities. Salome has also a post-processing module.

Salome is a part of a collection of open source software developed by EDF. Salome serves as the pre- and post-processor for Code_Aster (structural analysis) and Code_Saturne (CFD).

When Salome is used to create a mesh, this mesh needs to be exported by Salome in the UNV format. Then the mesh can be converted by the *ideasUnvToFoam* utility of OpenFOAM.

See `http://caelinux.org/wiki/index.php/Doc:Salome` for documentation and usage examples of Salome.

9.4 GMSH

GMSH is a meshing tool with some pre- and post-processing capabilities [`http://www.geuz.org/gmsh/`].

10 Meshing & OpenFOAMs meshing tools

OpenFOAM brings its own meshing utilities: *blockMesh* and *snappyHexMesh*. Alternatively there is a number of other meshers that can be used. Then, some conversion utilities (listed in Section 10.2) have to be used. *checkMesh* is a utility to investigate the mesh quality regardless of how the mesh was created.

blockMesh is able to also create the geometry of the simulation domain. *snappyHexMesh* is, in contrast to *blockMesh*, a meshing tool that uses an external geometry definition – in the form of an STL file.

10.1 Basics of the mesh

10.1.1 Files

A mesh is defined by OpenFOAM using several files. All of these files reside in `constant/polyMesh/`. The names of these files are rather self explanatory, the rest is explained in the OpenFOAM User Guide [14].

boundary contains a list of all faces forming the boundary patches

faces contains the definition of all faces. A face is defined by the points that form the face.

neighbour contains a list of the neighbouring cells of the faces

[16]Salome can be controlled completely by Python. Thus parametric geometry or mesh creation is possible.

owner contains a list of the owning cells of the faces

points contains a list of the coordinates of all points

The description of a mesh is based on the faces. The geometry is discretised into finite volumes – the cells. Each cell is delimited by a number of faces, e.g. a hexahedron has 6 faces. The faces can be divided into two groups. Boundary faces border only one cell. These faces make up the boundary patches. All other faces can be seen as the connection between two cells and are called internal faces. A face bordering more than two cells is not possible. An internal face is, by definition, owned by one cell and neighboured by the other one. So, the two cells connected by a face can be destincted.

This five files are absolutely necessary to describe a mesh regardless of how the mesh was created in the first place. However, some ways of creating a mesh produce additional files. Listing 70 shows a list of all files created with Gambit and converted by *fluentMeshToFoam*.

```
user@host:~/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/columnCase$ ls constant/polyMesh/
boundary  cellZones  faces  faceZones  neighbour  owner  points  pointZones
```

Listing 70: Content of `constant/polyMesh`

10.1.2 Definitions

Face

A face is defined by the vertices or points that are part of the face. The points need to be stated in an order which is defined by the face normal vector pointing to the outside of the cell or the block. The way faces are defined is the same for cells of the mesh or for blocks of the geometry.

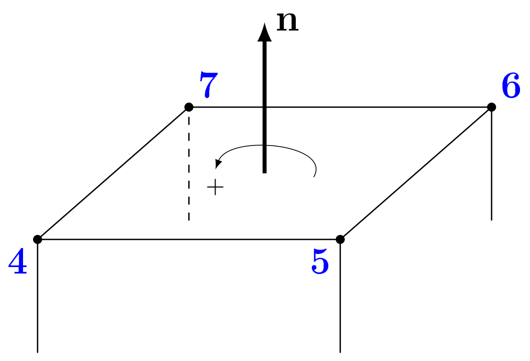


Figure 2: The top face of the generic block of Figure 3

To elaborate this further we look at the top face of the generic block of Figure 3 in Figure 2. The vertices with the numbers 4, 5, 6 and 7 are part of the face. The face normal vector – denoted by **n** in Figure 2 – that points outwards of the block is parallel to the local z axis. Therefore we need to specify the vertices defining the face in counter-clockwise circular order, when we look at the block from the top. The direction of rotation is marked in Figure 2 with the + sign. The starting vertex is arbitrary but it must not appear twice in the list.

Correct definitions			
`(4 5 6 7)`	`(7 4 5 6)`	`(6 7 4 5)`	`(5 6 7 4)`
Wrong direction of rotation			
`(7 6 5 4)`	`(4 7 6 5)`	`(5 4 7 6)`	`(6 5 4 7)`
Non-circular		Starting point repeated	
(7 5 6 4)		(4 5 6 7 4)	

Table 3: Valid and invalid face definitions

10.2 Converters

To use meshes created by programs other than *blockMesh* there is a number of converters. The User Guide [14] lists the following converters:

- *fluentMeshToFoam*
- *starToFoam*
- *gambitToFoam*
- *ideasToFoam*
- *cfx4ToFoam*

The names of the converters are pretty self explanatory.

10.2.1 *fluentMeshToFoam* and *fluent3DMeshToFoam*

fluentMeshToFoam converts meshes stored in the `*.msh` file format into the format of OpenFOAM. To be more specific, *fluentMeshToFoam* converts only 2D meshes, whereas 3D meshes can be converted using *fluent3DMeshToFoam*.

The converter expects the path to the `*.msh` file as an argument. The converter saves the mesh in the format of OpenFOAM in the `constant/polymesh` directory.

If converter is invoked from a directory other than the case directory, then the path to the case directory has to be specified via an additional argument. See Section 8.5.

If the mesh was created using an other dimension than in metres, the command line parameter `-scale` can be used to correct the scaling. OpenFOAM expects the mehs data to be expressed in metres.

All other possible option can be displayed with this command line parameter `fluentMeshToFoam -help`.

10.3 Mesh manipulation

10.3.1 *transformPoints*

The tool *transformPoints* can be used to scale, translate or rotate the points a mesh. Section 15.3.4 contains a case in which this tool can be useful.

11 *blockMesh*

blockMesh is used to create a mesh. The geometry is defined in *blockMeshDict*. This file also contains all necessary parameters needed to create the mesh, e.g. the number of cells. Therefore, *blockMesh* is a combined tool to define and mesh a geometry in contrast to other meshers that use CAD files to import a geometry created by some other software.

11.1 The block

The geometry created by *blockMesh* is based on the generic block. Figure 3 shows a generic block.

The blue numbers are the local vertex numbers of the block. The vertices are numbered counter-clockwise[17] in the local $x-y$ plane starting at the origin of the local coordinates[18]. Then the vertices above the local $x-y$ plane are counter-clockwise numbered starting with the vertex on the local z axis.

The local vertex numbers are important when defining the block. The first part of the `blockMeshDict` is generally a list of vertices. From this vertices the blocks are constructed. A block is defined by a list of 8 vertices which have to be ordered in a way to match the local vertices. Therefore the first entry in the list of vertices is the local 0 vertex, then the local 1 vertex follows. The local vertex numbers define the order in which the vertices have to passed when constructing a block.

[17]In mathematics the positive direction of rotation is generally determined with the right-hand or cork-screw rule. Let the thumb of your right hand point in the positive direction of the rotation axis, then the fingers of the right hand point in the positive direction of revolution.

[18]If we number all vertices in the $x-y$ plane then the local z axis is the axis of revolution. Thus the counter-clockwise direction is the mathematically positive direction of revolution.

The coordinate system originating from vertex 0 are the local coordinates. The local coordinates are important when specifying the number of cells or mesh grading (see *simpleGrading* in Section 11.4). The local coordinate axes do not need to be parallel or to coincide with the global coordinate axes.

The edges are also numbered and have a direction. Starting with the edge parallel to the local x axis the edges are numbered counter-clockwise starting with the edge emanating from the origin of the local coordinates. Next the edges parallel to the local y axis are numbered and finally the edges parallel to the local z axis. The edge number is important when specifying a grading for each edge individually (see *edgeGrading* in Section 11.4).

As it is indicated on Figure 3, the edges do not need to be parallel or straight. See Section 11.2.4 on how to define curved edges.

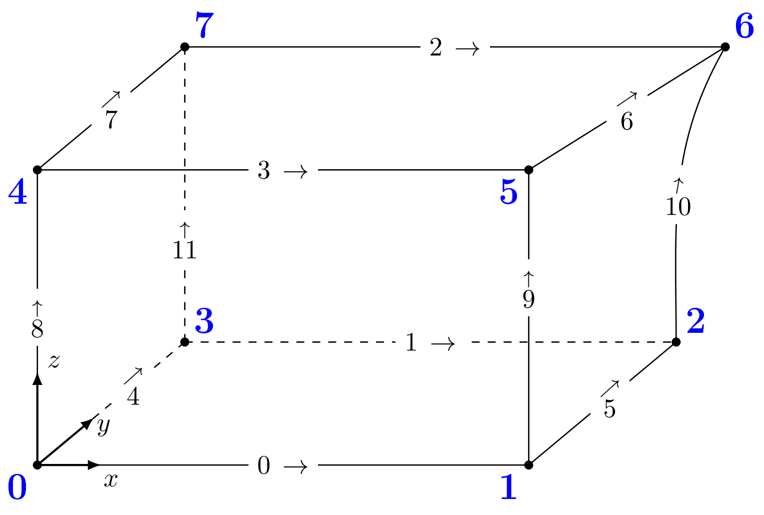


Figure 3: The generic block

11.2 The `blockMeshDict`

The file `blockMeshDict` defines the geometry and controls the meshing process of *blockMesh*. Listing 71 shows a reduced example of the `blockMeshDict`. This file was taken from the *cavity* tutorial case.

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;

vertices
(
    (0 0 0)   // 0
    (0 0 0.1) // 1
    ...
);
```

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    movingWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    ...
);

mergePatchPairs
(
);

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

Listing 71: A minimal `blockMeshDict`

11.2.1 `convertToMeters`

`convertToMeters` is a scaling factor to convert the vertex coordinates of `blockMeshDict` into meters. If the vertex coordinates are entered in an other unit than meters, this value has to be chosen accordingly. Listing 72 shows how to set this factor if the vertex coordinates are entered in millimeters.

```
convertToMeters 0.001;
```

Listing 72: *convertToMeters*

If the keyword `convertToMeters` is missing in the `blockMeshDict`, then no scaling is used, i.e. the default value of 1 is assumed.

To make sure if a scaling factor has been used, the output of *blockMesh* can be checked. Listing 73 shows the message issued by *blockMesh* regarding the scaling factor defined with `convertToMeters`.

```
Creating points with scale 0.1
```

Listing 73: Output of *blockMesh* when `convertToMeters` is set to 0.1

`convertToMeters` is a uniform scaling factor. Non-uniform scaling or other operations can be performed with another tool. See Section 10.3.1 and 15.3.4.

11.2.2 `vertices`

The `vertices` sub-dictionary contains a list of vertices. Each vertexs is defines by its coordinates in the global coordinate system. By default OpenFOAM treats these coordinates as in metres. However, with the help of the keyword `convertToMeters`, the vertices can be specified in other units.

The index of a vertex in this list is also the global number of this vertex, which is needed when constructing blocks from the vertices. Remember, counting starts from zero. Thus the first vertex is the list of vertices can be addressed by its index 0. A way to keep oneself aware of this fact is to add comments[19] to the vertex list as in Listing 71.

[19]As OpenFOAM treats its dictionaries much in the same way as C/C++ source files are treated by the C/C++ compiler. Therefore comments work the same way as they do in C or C++.

11.2.3 blocks

The only valid entry in the `blocks` sub-dictionary is the `hex` keyword. The `blocks` section of the `blockMeshDict` contains a list of `hex` commands. Listing 74 shows an example of a block definition with the `hex` keyword.

After the word `hex` a list of eight numbers defining the eight vertices of the block follows. The order of the entries in this list is the same order as the local vertex numbers of the block in Figure 3.

Then a list of three positive integer numbers follows. These numbers tell *blockMesh* how many cells need to be created in the direction of the local coordinate axes. Thus, the first number is the number of cells in the local x direction.

The next entry is a word stating the grading of the edges. This entry is in fact redundant. In OpenFOAM-2.1.x only the last entry, the list of expansion ratio, controls the grading. The third entry could even be omitted. However, maybe future versions of OpenFOAM make use of this entry. So the author does not advocate to omit this parameter.

The last entry of the block definition is a list of either three or twelve positive numbers. This numbers define the expansion ratio of the grading. In the case of three numbers, *simpleGrading* is applied. If twelve numbers are stated, then *edgeGrading* is performed.

If the list contains only one entry, then all edges share the same expansion ratio. Any other number of entries in this list leads to an error.

```
hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (2 4 1)
```

Listing 74: The `hex` command in `blockMeshDict`

Creating a block with 6 faces

The `hex` instruction can also be used to create a prism with a triangular cross-section. Such blocks are needed for simulations that make use of axi-symmetry. See the User Manual [14] for instructions on this topic.

11.2.4 edges

The `edges` sub-dictionary contains pairs of vertices that define an edge. By default edges are straight, by explicitely specifying the shape of the edge, curved edges can be created. This sub-dictionary can be omitted. Listing 75 shows the message issued by *blockMesh* when `edges` is omitted.

```
No non-linear edges defined
```

Listing 75: Output of *blockMesh* when `edges` is omitted

Otherwise, *blockMesh* issues a message as in Listing 76 regardless whether curved edges are actually created or only an empty `edges` sub-dictionary is present.

```
Creating curved edges
```

Listing 76: Output of *blockMesh* when `edges` is present

Creating arcs

With the keyword `arc` a circular arc between two vertices can be created. Listing 77 shows the definition of a circular arc between the vertices 0 and 3. In order to define a circular arc three points are necessary. Therefore the third point follows the indizes of the two vertices defining the edge.

```
edges
(
  arc 0 3 (0 0.5 0.05)
);
```

Listing 77: Definition of a circular edges in the `edges` sub-dictionary

The keyword `arc` can not be used to define a straight edge. If the two vertices and the additional interpolation point are co-linear, *blockMesh* will abort issuing an error message as in Listing 78.

```
--> FOAM FATAL ERROR:
Invalid arc definition - are the points co-linear?  Denom =0

    From function cylindricalCS arcEdge::calcAngle()
    in file curvedEdges/arcEdge.C at line 55.

FOAM aborting
```

Listing 78: Output of *blockMesh* when the three points defining an arc are co-linear

Creating splines

The keyword `spline` defines a spline. After the two vertices defining the edge a list of interpolation points has to follow.

```
edges
(
  spline 0 3 ((0 0.25 0.05) (0 0.75 0.05))
);
```

Listing 79: Definition of a spline in the `edges` sub-dictionary

Creating a poly-line

Other than a spline, a poly-line connects several points with straight lines.

```
edges
(
  polyLine 0 3 ((0 0.25 0.05) (0 0.75 0.05))
);
```

Listing 80: Definition of a poly-line in the `edges` sub-dictionary

Creating a straight line

For the sake of completeness there is the keyword `line`. This keyword takes the two vertices defining the edge as arguments. Straight lines are created by *blockMesh* by default. So there is no need for the user to specify straight lines.

```
edges
(
  line 0 3
);
```

Listing 81: Definition of a line in the `edges` sub-dictionary

11.2.5 `boundary`

The `boundary` list contains a dictionary per patch. This dictionary contains the type of the patch and the list of faces composing the patch. Listing 82 shows an example of how a patch consisting of one face is defined.

```
boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 3 2 1)
        );
    }

    ...
);
```

Listing 82: The `boundary` list of `blockMeshDict`

Pitfall: `patches`

In older versions of OpenFOAM, there was a `patches` sub-dictionary instead of the boundary sub-dictionary, see `http://www.openfoam.org/version2.0.0/meshing.php`. In some tutorial cases the old `patches` sub-dictionary can be found. However, it is recommended to use the `boundary` sub-dictionary because in some cases the use of the `patches` sub-dictionary results in errors.

To find out if there are still tutorial cases present that use the `patches` sub-dictionary the command of Listing 83 searches all files with the name `blockMeshDict` in the tutorials for the word `patches`.

```
find $FOAM_TUTORIALS -name blockMeshDict | xargs grep patches
```

Listing 83: Find cases that still use the `patches` sub-dictionary in the `blockMeshDict` to define the boundaries

11.2.6 `mergePatchPairs`

The `mergePatchPairs` list contains pairs of patches that need to be connected by the mesher.

Nothing to merge

This entry can be omitted. Listing 84 shows the message issued by *blockMesh* when `mergePatchPairs` is omitted.

```

There are no merge patch pairs edges
```

Listing 84: Output of *blockMesh* when `mergePatchPairs` is omitted

Patches to merge

When two patches need to be merged, then the patch pair needs to be stated in the `mergePatchPairs` list. The first patch of the pair is considered the master patch the second is the slave patch. The reason and consequences of this are described in the official User Manual [14].

```
mergePatchPairs
(
  (master slave)
);
```

Listing 85: The `mergePatchPairs` list in the `blockMeshDict`

If the patches that are part of the merging operation contain faces which are unaffected by the merging, the merge operation will fail. When the blocks of Figure 7 are to be connected, then the patch pair consists only of the face `(1 2 6 5)` and `(12 15 11 8)`. If one of the two patches contains an additional face, *blockMesh* will crash with an error. Thus the patches need to be defined as in Listing 86.

```
boundary
(
    master
    {
        type patch;
        faces
```

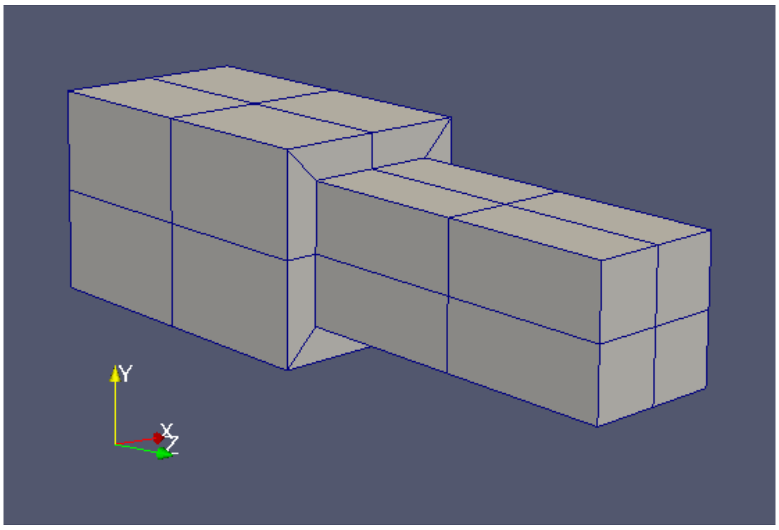


Figure 4: The mesh of two merged blocks

```
            (
                (1 2 6 5)
            );
        }
        slave
        {
            type patch;
            faces
            (
                (12 15 11 8)
            );
        }
        ...
);
```

Listing 86: The patch definitions needed to connect the blocks of Figure 7 with `mergePatchPairs` in the `boundary` sub-dictionary

blockMesh creates hanging nodes in order to connect the mesh of the blocks. Figure 4 shows the mesh of two merged blocks. Figure 5 shows the larger of the two blocks. The diagonal lines – one of them is marked with a red square in Figure 5 – are artefacts of the depiction of ParaView. The diagonal line that divides the L-shaped area is not present in the mesh. The right image in Figure 5 was edited with an image manipulation program to reflect the actual situation of the mesh. During the merging operation the face touching the second block is divided to match the second block. Thus, a quadrangular cell face is divided to two faces. The face denoted with the red 1 consists of 6 nodes and the face with the red 2 constists of four nodes.

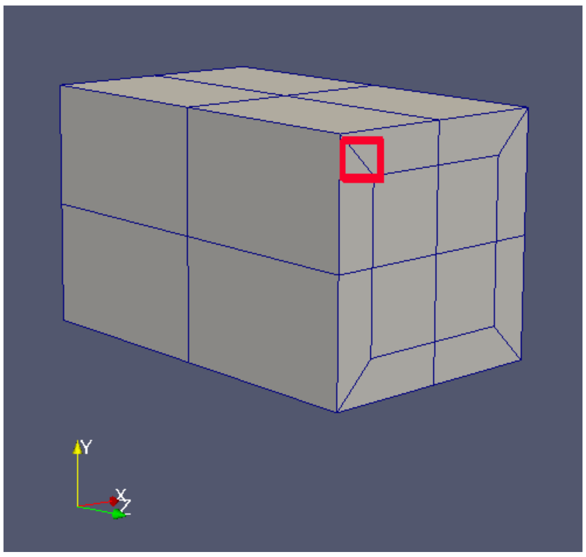
11.3 Create multiple blocks

A single block is almost never sufficient to model the geometry of a CFD problem. *blockMesh* offers the possibility to create an arbitrary number of blocks which can be connected. If blocks are constructed in a fashion that they share vertices, then they are connected by *blockMesh* by default.

11.3.1 Connected blocks

Figure 6 shows two connected blocks. These blocks share vertices. Therefore, the blocks are connected automatically.

Listing 87 shows the `blocks` sub-dictionary to create two connected blocks as they are depicted in Figure 6. The global vertex numbering is arbitrary. However, the order in which the vertex numbers are listed after



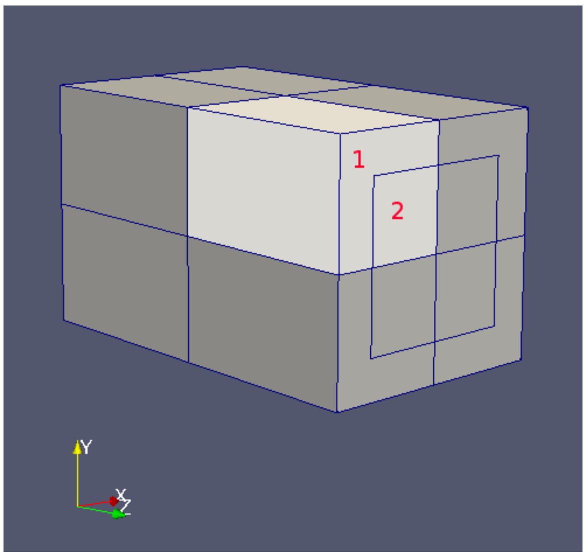


Figure 5: The mesh of two merged blocks. Left: screenshot of ParaView. Right: edited image to depict the actual faces.

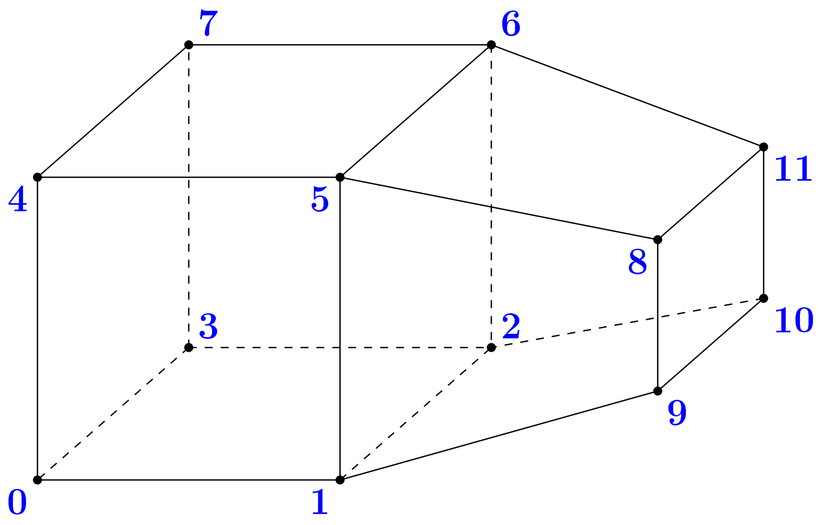


Figure 6: Two connected blocks

the `hex` keyword corresponds with the local vertex numbering of the generic block in Figure 3.

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (10 10 10) simpleGrading (1 1 1)
    hex (1 9 10 2 5 8 11 6) (10 10 10) simpleGrading (1 1 1)
);
```

Listing 87: The `blocks` entries in `blockMeshDict` to create the connected blocks of Figure 6

11.3.2 Unconnected blocks

Figure 7 shows a situation in which two blocks were created that share no vertices. Creating multiple blocks is done simply by adding a further entry in the `blocks` list. The blocks are connected by the statements in the `mergePatchPairs` section of the `blockMeshDict`.

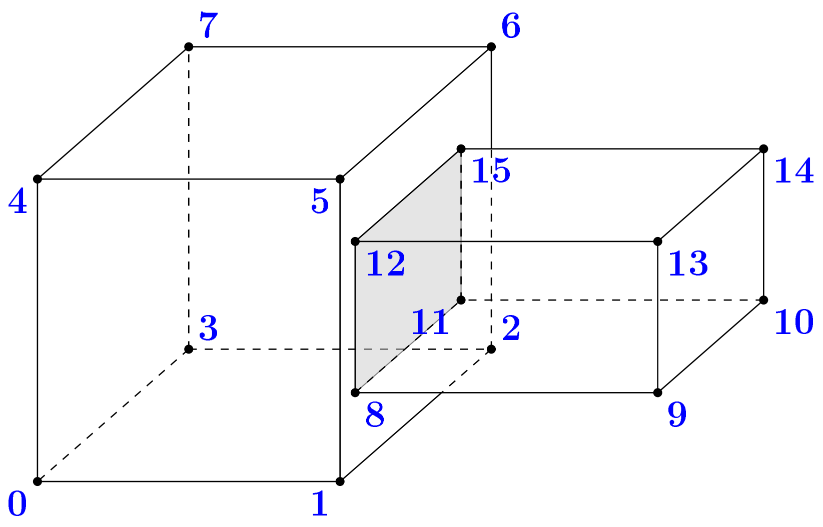


Figure 7: Two unconnected blocks

Listing 88 shows the `blocks` sub-dictionary to create two unconnected blocks as they are depicted in Figure 7.

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (10 10 10) simpleGrading (1 1 1)
    hex (8 9 10 11 12 13 14 15) (10 10 10) simpleGrading (1 1 1)
);
```

Listing 88: The `blocks` entries in `blockMeshDict` to create the unconnected blocks of Figure 7

In order to generate a connected mesh of the two blocks, the ??mergePatchPairs+ section of the `blockMeshDict` has to be provided with the two touching patches.

11.4 *Grading*

In the file `blockMeshDict` the grading can be defined globally for the edges of the block or for all edges individually. The grading is specified by the expansion ratio. This is the ratio of the widths of the first and the last cell along an edge. The direction of an edge is defined in the general definition of a block (see OpenFOAM Users Manual [14]).

simpleGrading

The global grading is defined for all edges parallel to the local x, y and z direction of the block. In Listing 89 the grading of all edges parallel to the local x axis oy the block is one, the grading of all edges parallel to the local y axis is two and the grading of all edges parallel to the local z axis is three.

```
simpleGrading (1 2 3)
```

Listing 89: *simpleGrading*

edgeGrading

With the keyword `edgeGrading` the grading of each edge of the block is specified individually. Therefore, the value of this keyword is a list with 12 numbers. The numbering of the edges – the list index corresponds to the edge number – is defined in the general definition of a block (see OpenFOAM Users Manual [14]). Listing 90 has the same effect as Listing 89.

```
edgeGrading (1 1 1 1 2 2 2 2 3 3 3 3)
```

Listing 90: *edgeGrading*

Pitfall: inconsistent grading

When a mesh consists of more than one block, then the grading of coincident edges must be consistent, i.e. these edges must have the same grading. In Listing 91 the grading of the last block is erroneous – the grading is set to 2 instead of 3. The error message caused by this fault is shown in Listing 92. The message mentions the blocks 5 and 8. This is correct, because OpenFOAM counts – like C, C++ and many more programming languages – from 0. Therefore, block 8 is the ninth block.

```
blocks
(
  hex (0 16 20 4 1 17 21 5) (30 5 10) simpleGrading (1 0.5 0.33)  // 1
  hex (1 17 21 5 2 18 22 6) (30 5 2) simpleGrading (1 0.5 1)  // 2
  hex (2 18 22 6 3 19 23 7) (30 5 15) simpleGrading (1 0.5 3) // 3

  hex (4 20 24 8 5 21 25 9) (30 2 10) simpleGrading (1 1 0.33)  // 4
  hex (5 21 25 9 6 22 26 10) (30 2 2) simpleGrading (1 1 1) // 5
  hex (6 22 26 10 7 23 27 11) (30 2 15) simpleGrading (1 1 3) // 6

  hex (8 24 28 12 9 25 29 13) (30 5 10) simpleGrading (1 2 0.33)  // 7
  hex (9 25 29 13 10 26 30 14) (30 5 2) simpleGrading (1 2 1) // 8
  hex (10 26 30 14 11 27 31 15) (30 5 15) simpleGrading (1 2 2)  // 9
);
```

Listing 91: Inconsistent grading

```
--> FOAM FATAL ERROR:
Inconsistent point locations between block pair 5 and 8
  probably due to inconsistent grading.

  From function blockMesh::calcMergeInfo()
  in file blockMesh/blockMeshMerge.C at line 294.

FOAM exiting
```

Listing 92: Error message caused by inconsistent grading

Pitfall: inconsistent discretisation

When a mesh consists of more than one block, then the number of cells of neighbouring blocks must be consistent, i.e. the blocks must have the same number of cells along coincident axes. In Listing 93 the number of cells of the first block is erroneous – the number is set to 44 instead of 45 along the local z direction. The error message caused by this faulty definition is shown in Listing 94. The message mentions the blocks 0 and 1. This error message indicates more clearly – other than Listing 92 – that OpenFOAM counts from 0.

```
blocks
(
    hex (0 1 5 4  8  9 13 12 ) (9 1 44) simpleGrading (1 1 1) // 1
    hex (1 2 6 5  9 10 14 13 ) (2 1 45) simpleGrading (1 1 1) // 2
    hex (2 3 7 6 10 11 15 14 ) (9 1 45) simpleGrading (1 1 1) // 3
);
```

Listing 93: Inconsistent discretisation

```
--> FOAM FATAL ERROR:
Inconsistent number of faces between block pair 0 and 1

    From function blockMesh::calcMergeInfo()
    in file blockMesh/blockMeshMerge.C at line 221.

FOAM exiting
```

Listing 94: Error message caused by inconsistent discretisation

Interesting observation

The source code also allows to state a list with only one entry. This is not documented in the official User Manual [14].

Listing 95 prooves this observation in the form of the responsible source code. The first command reads a scalar list from the input stream `is`. Then the three valid cases – one, three or twelve entries – are handled If none of the three branches of the `if-else` branching is entered an error is reported.

This code listing is a beautiful example of deducting the behaviour of a program from its source code. Unfortunately not all parts of OpenFOAMs source code are that easy to read and understand.

```
scalarList expRatios(is)

if (expRatios.size() == 1)
{
  // identical in x/y/z-directions
  expand_ = expRatios[0];
}
else if (expRatios.size() == 3)
{
  // x-direction
  expand_[0]  = expRatios[0];
  expand_[1]  = expRatios[0];
  expand_[2]  = expRatios[0];
  expand_[3]  = expRatios[0];

  // y-direction
  expand_[4]  = expRatios[1];
  expand_[5]  = expRatios[1];
  expand_[6]  = expRatios[1];
  expand_[7]  = expRatios[1];

  // z-direction
  expand_[8]  = expRatios[2];
  expand_[9]  = expRatios[2];
  expand_[10] = expRatios[2];
  expand_[11] = expRatios[2];
}
else if (expRatios.size() == 12)
{
  expand_ = expRatios;
}
else
{
  FatalErrorIn
  (
    "blockDescriptor::blockDescriptor"
    "(const pointField&, const curvedEdgeList&, Istream&)"
  )  << "Unknown definition of expansion ratios: " << expRatios
```

```
    << exit(FatalError);
}
```

Listing 95: Some content of `blockDescriptor.C`

11.5 Parametric meshes by the help of *m4* and *blockMesh*

In `blockMeshDict` only plain text is allowed, i.e. no symbols can be used. Also, no calculations can be made by *blockMesh* with the exception of the keyword `convertToMeters`.

11.5.1 The `blockMeshDict` prototype

If the user wants to create parametrised meshes, i.e. properties of the mesh are calculated from certain parameters, an additional working step is necessary. In order to create a parametric mesh a prototype of the file `blockMeshDict` is needed. This prototype contains symbols. Listing 96 shows the block definition of such a prototype. This block definition is not fully parametric, only the number of cells is calculated. Note, that in local y direction only one cell is used for discretisation. This indicates a 2D problem.

```
blocks
(
    hex (0 1 5 4   8  9 13 12 ) (N1x 1 N1z) simpleGrading (1 1 1)  // 1
    hex (1 2 6 5   9 10 14 13 ) (N2x 1 N1z) simpleGrading (1 1 1)  // 2
    hex (2 3 7 6  10 11 15 14 ) (N1x 1 N1z) simpleGrading (1 1 1)  // 3
);
```

Listing 96: Block definition of the prototype

11.5.2 The macro programming language *m4*

In order to replace the symbols of the prototype with meaningful numbers, the prototype has to be processed by a macro programming language interpreter. In this case the programming language *m4*[20] is used. The interpreter of this language scans the prototype for valid expressions (macros) and replaces them with their result.

To replace a symbol of the prototype with a meaningful number, a macro has to be defined. Listing 97 shows the definition of the symbols used in Listing 96. In the first line a general variable `h` is defined. The second and the third instruction calculate the number of cells in the local x direction based on the variable `h`. The last instruction calculates the number of cells in the local z direction.

```
define(h,2)

define(N1x,`eval(9*h)')
define(N2x,`eval(2*h)')

define(N1z, `eval(45*h)')
```

Listing 97: Block definition of the prototype

This kind of parametrisation allows to specify a multiplier for the number of cells. The discretisation length can not be refined gradually this way. Specifying the discretisation length requires more complex math than integer operations.

Complex math - first shot

The builtin mathematic macros of *m4* are restricted to integer operations only. As *m4* supports system calls, floating point calculations can be done by an external program. Consequently, the symbol is replaced by the result of the system call.

In Listing 98 some variables are defined. In line 13 a macro is defined that passes its arguments to the operating system via a system call. The argument of the command `esyscmd` gets executed in the command

[20] m4 is part of the GNU project. See `http://www.gnu.org/software/m4/manual/index.html`

line. This is the reason for the rather complicated argument of `esyscmd`. The output of the command echo is the input of the command `bc`[21]. Note the use of the pipe.

The input of the command `echo` is composed of three successive operations that need to be performed by the calculator. The first instruction says that two digits after the decimal point should be used. The second instruction calculates the difference between the first two arguments and the last instruction divides this difference by the third argument. These operations compute first the length of the block that needs to be descretised. Then by dividing this length by the discretisation length the number of cells is calculated.

The output is then formatted by the macro `format`. Note the formatting string `%.0f`. This causes the result to loose its digits after the decimal point. This step is absolutely necessary, because only integers are allowed to define the number of cells.

```
// # enter discretization length
define(dx,0.005)
define(dz,0.005)

// # enter x coordinates
define(x1,0.0555)
define(x2,0.0945)

// # enter heights (z coordinates)
define(H1, 0.20)

// # relDiff: ($1 - $2) / $3   # decimal places truncated (done by format %.0f)
define(relDiff,`format(`%.0f', esyscmd(echo "scale=2; a=$1-$2; a/$3" | bc))')

define(N1x,`relDiff(x1,0,dx)')
define(N2x,`relDiff(x2,x1,dx)')

define(N1z,`relDiff(H1,0,dz)')
```

Listing 98: Block definition of the prototype

Listing 98 allows to calculate the number of cells from a specified discretisation length. Due to rounding operations the specified discretisation length is not exactly met. Listing 99 shows the result after the macros from Listings 96 and 98 have been processed.

```
blocks
(
    hex (0 1 5 4  8  9 13 12 ) (11 1 40) simpleGrading (1 1 1)  // 1
    hex (1 2 6 5  9 10 14 13 ) (7  1 40) simpleGrading (1 1 1)  // 2
    hex (2 3 7 6 10 11 15 14 ) (11 1 40) simpleGrading (1 1 1)  // 3
);
```

Listing 99: Resulting parametric block definition

Complex math - the better solution

The above described way to do mathematical operations is not very elegant. At this place a more elaborate solution is presented.

Listing 100 shows some examples taken from a *m4* script found in the tutorials. The first statement changes the delimiter for comments. By changing the delimiter to `//`, comments have the same delimiter as C or C++. Remember, OpenFOAM dictionaries follow the C++ syntax, therefore, anything following a `//` is treated as a comment. Now, commented lines are always treated as comments by *m4* as well as OpenFOAM. See the first line of Listing 98. There, the `//` starts a comment for OpenFOAM and the `#` starts a comment for *m4*. Setting the delimiter for comments to be the same as in C++ removes an ambiguity and a possible source for errors.

The second line of Listing 100 redefines the quote delimiter. Changing this delimiters from the standard to the brackets is probably done to improve readability.

In line 4 of Listing 100 a macro named `calc` is defined. This macro also uses a system call to outsource the actual math. In this case the interpreter of the script programming language Perl[22] is called. This interpreter receives a command line argument and an instruction. The command line argument `-e` tells the interpreter

[21] *bc* is a calculator program. It is part of the GNU project.
[22] See `http://www.perl.org/`

that only one line of code will follow. The interpreter will interpret this single line and exit. The instruction `print ($1)` is a function that prints its argument on the standard output. The argument of the `print` function is the argument of the `calc` macro. Therefore, the mathematical operation can be written directly in the code. See line 9 for an example. There, the symbols `rb` and `Rb` are replaced my *m4* by their definition. The argument of the `calc` macro is passed via the system call to the Perl interpreter. As Perl is able to do mathematical operations, the interpreter computes the result of the expression and executes the function `print`. The macro `esyscmd` returns the standard output of the command it executed.

Line 12 of Listing 100 shows that even more complex math – e.g. using trigonometric functions – is possible.

```
1  changecom(//)
2  changequote([,])
3
4  define(calc, [esyscmd(perl -e 'print ($1)')])
5
6  define(rb, 0.5)
7  define(Rb, 0.7)
8
9  define(ri, calc(0.5*(rb + Rb)))
10
11 define(pi, 3.14159265)
12 define(ca0, calc(cos((pi/180)*a0)))
```

Listing 100: Doing complex math with *m4*

11.6 Trouble-shooting

11.6.1 Viewing the blocks with *ParaView*

A mesh created by *blockMesh* consists of blocks. Listing 101 shows how *ParaView* can be used to visualise the blocks.

```
paraFoam -block
```

Listing 101: Visualising the blocks

This way, only the blocks are displayed. ParaView only reads the file `blockMeshDict`. Figure 8 shows the blocks of a parametric mesh. It consists of nine blocks. The image shows also the numbers of the vertices.

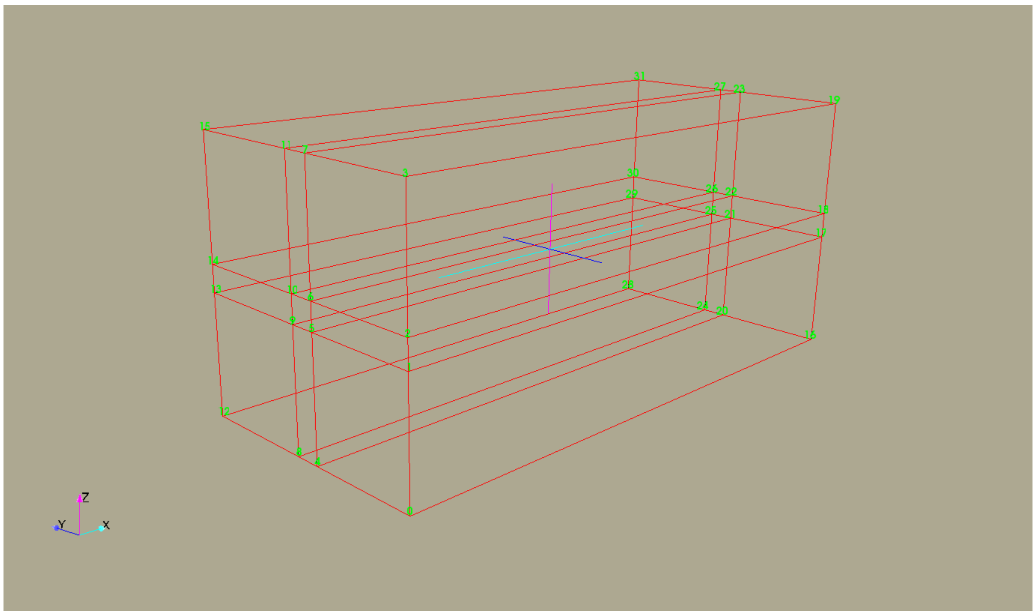


Figure 8: The blocks of a parametric mesh consisting of nine blocks.

11.6.2 Viewing the blocks with *pyFoam*

Troubleshooting can be difficult when *blockMesh* doesn't create a mesh and displays some error messages instead.
See Section 11.6.1 for the discussion of a tool which is able to display the blocks as they are defined in `blockMeshDict`. This tool even works, when *blockMesh* fails due to an errorneous definition in `blockMeshDict`.

12 *snappyHexMesh*

snappyHexMesh is a meshing tool that is able to mesh the space around an arbitrary object. This is generally the case in external aerodynamics. *snappyHexMesh* can only be used in conjunction with *blockMesh*. Even though, *snappyHexMesh* is not a stand alone meshing tool, it is a very useful utility.

12.0.3 Workflow

The topic of investigation is the drag force on an anvil.

Figure 9: Problem

To solve this problem, the object under investigaton – in this case the anvil – has to be modelled by means of CAD. *snappyHexMesh* needs a representation of the object – actually only the objects surface is important – in STL format.

Then, the computational domain has to be specified. Figure 10 shows the background domain (the rectangle) and the object under investigation (the anvil). To simulate this case, the space outside of the anvil has to be meshed. Mathematically skeaking, the anvil has to be subtracted from the background domain. The remainder is our computational domain.

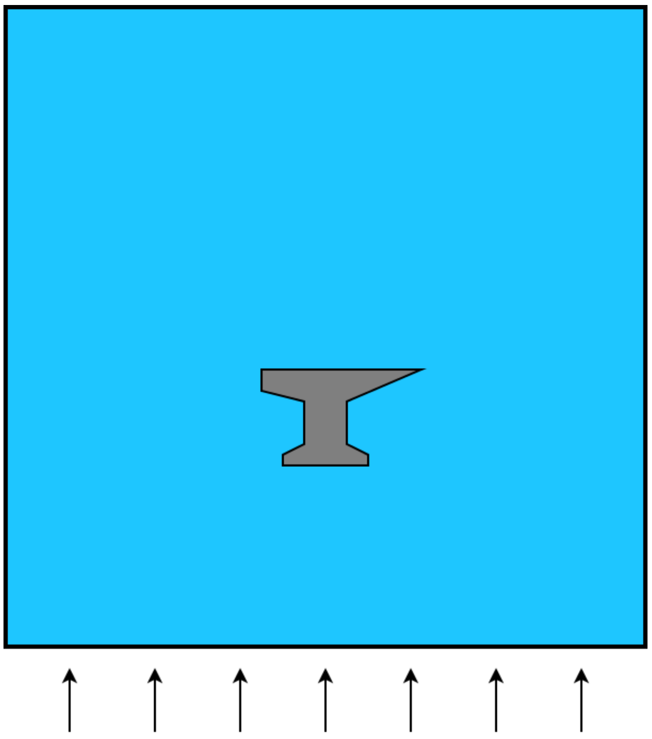


Figure 10: Problem geometry

The creation of the mesh in this case follows a two step approach:

1. The background domain is meshed using blockMesh, thus creating the background mesh
2. snappyHexMesh then perfoms several steps

Cell splitting The cells of the background mesh near the objects surface are refined.

Cell removal Cells of the background mesh inside the object are removed.

Cell snapping The remaining background mesh is modified in order to reconstruct the surface of the object.

Layer addition Additional hexahedral cells are introduced on the boundary surface of the object to ensure a good mesh quality.

12.0.4 Pitfalls

Run time

If *snappyHexMesh* is finished in less than a second, then something is wrong. As *snappyHexMesh* performs up to three work intensive steps (castellation, snapping and layer addition), a run of *snappyHexMesh* takes a couple of seconds or even longer (tens of seconds).

`convertToMeters` in `blockMeshDict`

When creating a mesh with snappyHexMesh different scales of the background mesh and the STL mesh are a frequent source of error. Check the following things:

1. The unit of the vertex coordinates in `blockMeshDict`
2. The value of the `convertToMeters` keyword in `blockMeshDict`
3. The unit in which the STL was created

13 *checkMesh*

checkMesh is a tool to perform tests on an existing mesh. *checkMesh* is simply invoked by its name. Like other tools, *checkMesh* assumes to be called from the case directory. When *checkMesh* is to called from an other location than the case directory, the path to the case directory has to be specified with the option `-case`.

Listing 102 shows an error message produced by *checkMesh*, if *checkMesh* has been called with no mesh present. In this case the tool can't find the files specified in Section 10.1.

```
--> FOAM FATAL ERROR:
Cannot find file "points" in directory "polyMesh" in times 0 down to constant

    From function Time::findInstance(const fileName&, const word&, const IOobject::readOption,
     const word&)
    in file db/Time/findInstance.C at line 188.

FOAM exiting
```

Listing 102: No mesh present

A more thorough testing is performed when *checkMesh* is called with two additional options. Then *checkMesh* performs some further tests.

```
checkMesh -allGeometry -allTopology
```

Listing 103: Do more checks

checkMesh has also the `-latestTime` option like many other OpenFOAM tools. This option is particularly useful when examining meshes created by *snappyHexMesh*. *snappyHexMesh* stores intermediate meshes if it is not told otherwise. By default, after a completed run of *snappyHexMesh* there are the background mesh and the results of the three basic stages of a *snappyHexMesh* run (castellation, snapping and layer addition). Depending on which of these steps are active up to four meshes may be present. Restricting *checkMesh* to the final mesh reduces runtime and avoids the unnecessary examination of an intermediate mesh.

13.1 Definitions

In order to understand the output of *checkMesh* it is necessary to define some quantities calculated by *checkMesh*.

13.1.1 Face non-orthogonality

Non-orthogonality is a property of the faces. Each face – with the exception of boundary faces – connects two cells. The non-orthogonality is the angle between the vector connecting the cell centres and the face normal vector. In Figure 11 the vector connecting the cell centres is denoted **d** and the face normal vector[23] **S**.

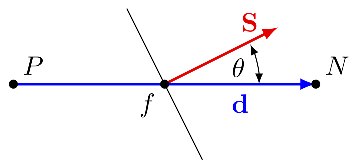


Figure 11: Definition of non-orthogonality

In a perfectly orthogonal mesh the vectors **d** and **S** are parallel. If a mesh is non-orthogonal these vectors draw an angle as in Figure 11. This angle can be calculated from **d** and **S** by Eq. 9.

[23]The face normal vector or face area vector is a vector normal to a face. The length of this vector is equal to the area of the face.

$$\mathbf{d} \cdot \mathbf{S} = ||\mathbf{d}||\ ||\mathbf{S}||\ \cos(\theta) \tag{7}$$

$$\frac{\mathbf{d} \cdot \mathbf{S}}{||\mathbf{d}||\ ||\mathbf{S}||} = \frac{||\mathbf{d}||\ ||\mathbf{S}||\cos(\theta)}{||\mathbf{d}||\ ||\mathbf{S}||} = \cos(\theta) \tag{8}$$

$$\theta = \arccos(\frac{\mathbf{d} \cdot \mathbf{S}}{||\mathbf{d}||\ ||\mathbf{S}||}) \tag{9}$$

Eq. 9 can also be found in the sources of OpenFOAM in the function `faceNonOrthogonality` in the file `cellQuality.C`. Listing shows 104 a loop over all faces. For each face the non-orthogonality is computed. The vectors `d` and `s` are the connecting vector between the cell centres and the face area vector. The scalar `cosDDotS` is the angle θ of Figure 11.

Note the two precautions that were taken to avoid numerical issues. First, the denominator is the sum of the product of the magnitudes and `VSMALL`. `VSMALL` is a number with a very small value to prevent division by zero. Second, the argument of the `acos` function is `min(1.0, (d & s)/(mag(d)*magS + VSMALL))`. Keeping the argument of the arc-cosine equal or below 1 makes perfectly sense, because the arc-cosine is defined only for values between -1 and 1. The limit of -1 is inherently ensured. The inner product of two vectors is always positive. `VSMALL` is also positive.

```
forAll(nei, faceI)
{
    vector d = centres[nei[faceI]] - centres[own[faceI]];
    vector s = areas[faceI];
    scalar magS = mag(s);

    scalar cosDDotS =
       radToDeg(Foam::acos(min(1.0, (d & s)/(mag(d)*magS + VSMALL))));
    result[own[faceI]] = max(cosDDotS, result[own[faceI]]);
    result[nei[faceI]] = max(cosDDotS, result[nei[faceI]]);
}
```

Listing 104: A detail of the function `faceNonOrthogonality` in the file `cellQuality.C`

The non-orthogonality reported by *checkMesh* is the angle θ of Figure 11. Therefore the reported non-orthogonality lies in the range between 0 and 90. A non-orthogonality of 0 means the mesh is orthogonal and consists of hexahedra (cudoids) or regular tetrahedra. Listing 106 shows the output of *checkMesh*. In this case the mesh is orthogonal, the maximum and average non-orthogonality is 0.

Listing 108 shows the output of *checkMesh* in case of a non-orthogonal mesh. Listing 109 indicates that a non-orthogonality of above 70 triggers *checkMesh* to issue a warning message.

13.1.2 Face skewness

OpenFOAM defines skewness in a mesh different than other tools, e.g. Gambit. The reason for this OpenFOAM-specific definition is that this definition is associated with the definition of a skewness error in [10] as part of mesh induced discretisation errors.

Skewness is a property of the faces of the mesh. Each face connects two cells – except boundary faces. Figure 12 shows the cell centres P and N of two adjacent cells. The face face_{PN} is the face connecting these two cells. The point F is the face centre of the face face_{PN}. The line $c = \overline{PN}$ connects the cell centres. This connecting line intersects with the face face_{PN}. This intersection point I divides the line c into the two parts c_1 and c_2.

To calculate the location of I the length of c_1 is of key interest because the skewness is defined in Eq. 10. The location (the vector to) the points P, N and F are easily obtained. From this three vectors $\mathbf{d}_o$, $\mathbf{d}_n$ and $\mathbf{c}$ is computed. With $\mathbf{d}_o$ and $\mathbf{d}_n$ the inner product with the face area vector $\mathbf{A}_f$ is computed to obtain `dOwn` and `dNei`[24].

[24]`dOwn` and `dNei` are actual variable names. Therefore these symbols are written in typewriter font.

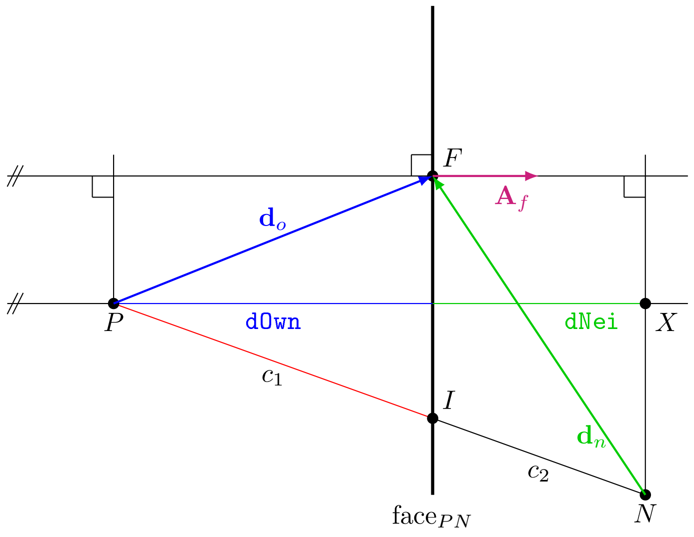


Figure 12: Definition of skewness

$$\texttt{skewness} = \frac{|\overline{IF}|}{|\overline{PN}|} \tag{10}$$

$$\mathbf{d}_o = \vec{F} - \vec{P} \tag{11}$$

$$\mathbf{d}_n = \vec{F} - \vec{N} \tag{12}$$

$$\mathbf{c} = \vec{N} - \vec{P} \tag{13}$$

$$\texttt{dOwn} = \frac{\mathbf{d}_o \cdot \mathbf{A}_f}{||\mathbf{A}_f||} \tag{14}$$

$$\texttt{dNei} = \frac{\mathbf{d}_n \cdot \mathbf{A}_f}{||\mathbf{A}_f||} \tag{15}$$

$$\angle(XPN) = \alpha \tag{16}$$

$$\cos(\alpha) = \frac{\texttt{dOwn}}{c_1} = \frac{\texttt{dOwn + dNei}}{c_1 + c_2} = \frac{\texttt{dOwn + dNei}}{||\mathbf{c}||} \tag{17}$$

$$c_1 = \frac{\texttt{dOwn}}{\texttt{dOwn + dNei}}||\mathbf{c}|| \tag{18}$$

$$\vec{I} = \vec{P} + c_1\mathbf{c} \tag{19}$$

$$\texttt{skewness} = \frac{||\vec{F} - \vec{I}||}{||\mathbf{c}||} \tag{20}$$

Note that both $\vec{P}$ and $\mathbf{c}$ are vectors. The reader hopefully excuses this lack of consistency in mathematical notation. $\vec{P}$ denotes the position vector of the point P. In this case the symbol $\vec{P}$ is prefered to $\mathbf{P}$ in order to use symbols that can be found in Figure 12.

Listing 105 shows a detail of the function `faceSkewness` from the file `cellQuality.C`. There a loop over all internal faces is traversed. The loop body contains the calculation of the skewness. First `dOwn` and `dNei` are computed. Then the location of the point I is determined. The variable `faceIntersection` of the type `point` contains the position vector to the point I – the point at which the connection line between the cell centres intersects the face. Finally, the skewness is calculated (compare Eq. 20). Notice the precaution against a possible division by zero (adding `VSMALL` to the denominator).

```
forAll(nei, faceI)
{
    scalar dOwn = mag
    (
        (faceCtrs[faceI] - cellCtrs[own[faceI]]) & areas[faceI]
    )/mag(areas[faceI]);

    scalar dNei = mag
    (
        (cellCtrs[nei[faceI]] - faceCtrs[faceI]) & areas[faceI]
    )/mag(areas[faceI]);

    point faceIntersection =
        cellCtrs[own[faceI]]
      + (dOwn/(dOwn+dNei))*(cellCtrs[nei[faceI]] - cellCtrs[own[faceI]]);

    result[faceI] =
       mag(faceCtrs[faceI] - faceIntersection)
       /(mag(cellCtrs[nei[faceI]] - cellCtrs[own[faceI]]) + VSMALL);
}
```

Listing 105: A detail of the function `faceSkewness` in the file `cellQuality.C`

13.1.3 Face concavity

pending

13.1.4 Cell concavity

When a cell is concave

13.2 Pitfalls

The results of *checkMesh* need to be taken with a grain of salt. Therefore, it is helpful to know how *checkMesh* defines the qualitity measures it tests for (Section 13.1) and also to know about the shortcomings of the tests performed by *checkMesh* (Section 13.2).

The tests performed by *checkMesh* do not necessarily guarantee the mesh to be suitable for simulation. Furthermore, if a mesh fails a test, that does not necessariliy mean that it is unsuitable for calculation.

13.2.1 Mesh quality - *aspect ratio*

checkMesh performs a number of quality checks. However, the user has to be careful. *checkMesh* does only check if a mesh makes a simulation impossible. There are some situations in which *checkMesh* does not issue an error or a warning, however, a mesh can nevertheless be unsuitable for a successful calculation.

The aspect ratio is the ratio of the largest and the smallest dimension of the cells. For the aspect ratio there are no limits. Listing 106 shows the output of *checkMesh* when a mesh with high aspect ratio cells is tested. Although *checkMesh* does not complain, the mesh is not suitable for simulation. Even with extremely small time steps numerical problems appear.

```
Checking geometry...
    Overall domain bounding box (0 0 0) (0.1 0.1 0.01)
    Mesh (non-empty, non-wedge) directions (1 1 1)
    Mesh (non-empty) directions (1 1 1)
    Boundary openness (-9.51633e-17 1.17791e-18 -4.51751e-17) OK.
    Max cell openness = 1.35525e-16 OK.
    Max aspect ratio = 100 OK.
    Minimum face area = 2.5e-07. Maximum face area = 2.5e-05.  Face area magnitudes OK.
    Min volume = 1.25e-09. Max volume = 1.25e-09.  Total volume = 0.0001.  Cell volumes OK.
    Mesh non-orthogonality Max: 0 average: 0
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 2e-06 OK.
    Coupled point location match (average 0) OK.

Mesh OK.

End
```

Listing 106: *checkMesh* output for a mesh with high aspect ratio

13.2.2 Mesh quality - *skewness*

There are different ways to calculate the skewness of a finite volume cell. To test whether *checkMesh* complains about high skewness, a mesh is distorted by the use of edge grading. Figure 13 shows this mesh. Parallel edges are graded alternately – alternating between the expand ratio and its reciprocal value. Listing 107 shows the grading settings. The test case for this examination is the *cavity* case of *icoFoam*. This case can be found in the tutorials.

```
hex (0 1 2 3 4 5 6 7) (20 20 2) edgeGrading (3 0.33 3 0.33  1 1 1 1  1 1 1 1)
```

Listing 107: Block definition in *blockMeshDict* to achieve high skewness

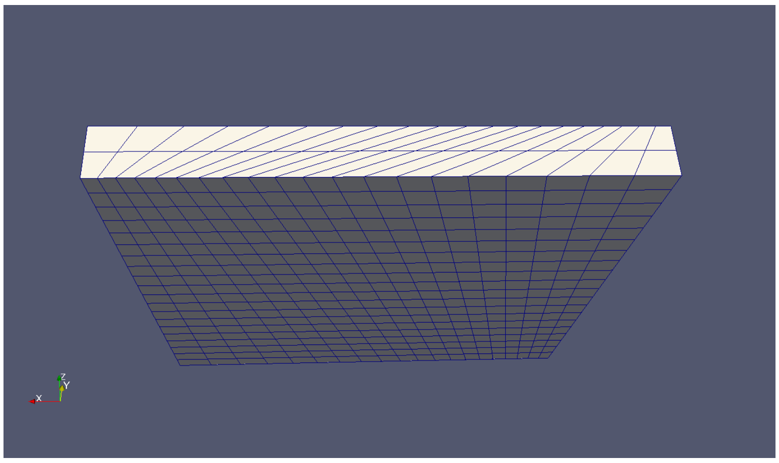


Figure 13: A distorted mesh

checkMesh issues no warnings for the value pair 3 and 0.33. The values 4 and 0.25 cause a warning about *severly non-orthogonal faces*.

However, a simulation is impossible for much lower values. The simulation runs for the value pair 1.33 and 0.75. The values 1.4 and 0.714 cause the simulation to crash. The limits of stability of a simulation are therefore reached earlier than the limits of *checkMesh*.

To conclude this section, the user should bear the folling statement in mind. Numerical problems of a simulation may be caused by bad mesh quality. In some cases – like the one presented above – bad mesh quality is the root of the problem, but *checkMesh* issues no warnings. However, the values of the quality characteristics may give a hint. Some manuals of CFD software propose numerical ranges for characteristics like aspect ratio to ensure good quality.

```
Checking geometry...
  Overall domain bounding box (0 0 0) (0.1 0.1 0.01)
  Mesh (non-empty, non-wedge) directions (1 1 1)
  Mesh (non-empty) directions (1 1 1)
  Boundary openness (4.23516e-18 9.03502e-18 1.60936e-16) OK.
  Max cell openness = 1.67251e-16 OK.
  Max aspect ratio = 3.63059 OK.
  Minimum face area = 1.42648e-05. Maximum face area = 7.1694e-05.  Face area magnitudes OK.
  Min volume = 1.03854e-07. Max volume = 1.69673e-07.  Total volume = 0.0001.  Cell volumes OK
    .
  Mesh non-orthogonality Max: 69.4798 average: 32.8092    Non-orthogonality check OK.
  Face pyramids OK.
  Max skewness = 2.35485 OK.
  Coupled point location match (average 0) OK.

Mesh OK.

End
```

Listing 108: *checkMesh* output for the distorted mesh; grading ratios 3 and 0.33

```
Checking geometry...
    Overall domain bounding box (0 0 0) (0.1 0.1 0.01)
    Mesh (non-empty, non-wedge) directions (1 1 1)
    Mesh (non-empty) directions (1 1 1)
    Boundary openness (4.23516e-18 -6.21157e-18 1.18585e-16) OK.
    Max cell openness = 2.37664e-16 OK.
    Max aspect ratio = 4.23706 OK.
    Minimum face area = 1.23181e-05. Maximum face area = 8.67874e-05.  Face area magnitudes OK.
    Min volume = 1.00882e-07. Max volume = 1.84055e-07.  Total volume = 0.0001.  Cell volumes OK
      .
    Mesh non-orthogonality Max: 73.1635 average: 36.2131
   *Number of severely non-orthogonal faces: 80.
    Non-orthogonality check OK.
  <<Writing 80 non-orthogonal faces to set nonOrthoFaces
    Face pyramids OK.
    Max skewness = 2.93978 OK.
    Coupled point location match (average 0) OK.

Mesh OK.

End
```

Listing 109: *checkMesh* output for the distorted mesh; grading ratios 4 and 0.25

13.2.3 Possible non-pitfall: `twoInternalFacesCells`

If a mesh for a two-dimensional simulation is created and checked using *checkMesh* with the `-allTopology` option enabled[25], then *checkMesh* will issue a message like in Listing 110. This message indicates, that there are cells present with only two internal faces. This message can be ignored when 2D meshes are concerned. The corner cells of a rectangular mesh have – by definition – only two internal faces.

```
Checking topology...
    Boundary definition OK.
    Cell to face addressing OK.
    Point usage OK.
    Upper triangular ordering OK.
    Face vertices OK.
    Topological cell zip-up check OK.
    Face-face connectivity OK.
  <<Writing 4 cells with with two non-boundary faces to set twoInternalFacesCells
    Number of regions: 1 (OK).
```

Listing 110: *checkMesh* output for a 2D mesh with `-allTopology` option set.

If this message appears when a 3D mesh is examined, then there is probably some error in the definition of the mesh. A cell in a 3D mesh should have at least three internal faces. A message stating the presence of cells with two internal faces in a 3D mesh indicates non-connected regions.

13.3 Useful output

The output of checkMesh in Listing 110 also shows another interesting thing to know about *checkMesh*. The line `«Writing 4 cells with with two non-boundary faces to set twoInternalFacesCells` tells the user that *checkMesh* created a set of cells that are found to have some problems.

Figure 14 shows the content of the case which resulted in Figure 13. There we see a directory named `sets` inside the `polyMesh` folder. The `sets` folder was created by *checkMesh* and inside this folder *checkMesh* stores any sets it creates. The file names are rather self-explanatory, e.g. the file `skewFaces` contains all faces which failed the test for skewness. All these cell or face sets can be viewed with *paraView*.

[25]When the `-allTopology` option is enabled, *checkMesh* performs two additional topological checks. Checking the face connectivity is one of these checks.

```
.
├── 0
│   ├── p
│   └── U
├── constant
│   ├── polyMesh
│   │   ├── blockMeshDict
│   │   ├── boundary
│   │   ├── faces
│   │   ├── neighbour
│   │   ├── owner
│   │   ├── points
│   │   └── sets
│   │       ├── lowQualityTetFaces
│   │       ├── nonOrthoFaces
│   │       ├── skewFaces
│   │       └── underdeterminedCells
│   └── transportProperties
└── system
    ├── controlDict
    ├── fvSchemes
    └── fvSolution
```

Figure 14: Sets created by *checkMesh* in the `sets` directory.

14 Other mesh manipulation tools

14.1 *topoSet*

The tool *topoSet* creates point, face or cell sets from a geometric definition. There are a number of ways to define the geometric region containing the intended points, faces or cells.

14.1.1 Usage

The dictionary `topoSetDict` is used to define the geometric region. Find some examples in the tutorials using the following command.

```
find $FOAM_TUTORIALS -name topoSetDict
```

Listing 111: Find examples for the use of *topoSet*

A face or cell set will contain only faces or cells whose centres lie within the specified geometric region.

14.1.2 Pitfall: The definition of the geometric region

To demonstrate the function of topoSet a cell set was defined for the cavity tutorial-case. The mesh of the cavity case is $1 \times 1 \times 0.1\,\mathrm{m}$ and the box defining the cell set was chosen to be $0.5 \times 0.5 \times 0.05\,\mathrm{m}$. The dimensions of this box are simply half the dimensions of the mesh. However, only cells whose cell centre is located in the box are contained in the cell set. As the mesh is one cell in depth and $0.1\,\mathrm{m}$ in depth, all the cell centres are exactly at $z = 0.05\,\mathrm{m}$. Due to inevitable numerical errors in calculating the cell centre[26], the numerical errors decided whether a cell was included into the cell set or not.

To avoid this error, always make sure the geometric region contains all the intended cells.

[26]The location of the cell centre is not stored in any file, thus this quantity has to be computed.

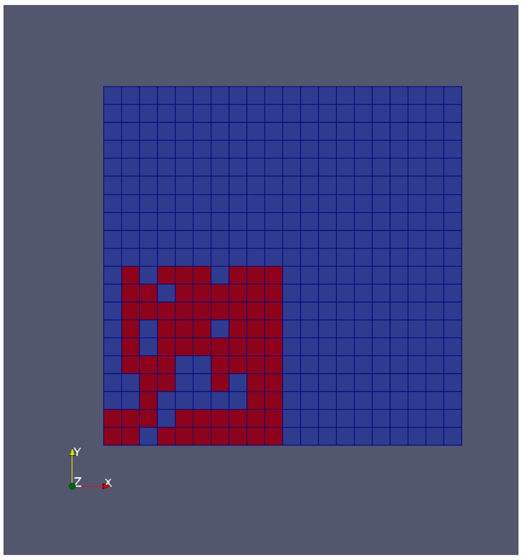


Figure 15: A faulty cell set definition. The red cells are part of the cell set. All other cells are blue.

14.2 *refineMesh*

The tool *refineMesh* is used – just as the name suggests – to refine a mesh.

14.2.1 Usage

First a cell set has to be defined, this can be done using the tool *topoSet*.

With the dictionary `refineMeshDict` the rules for refining a particular cell set can be stated. When rules have been defined in `refineMeshDict` , then the command line option `-dict` has to be used.

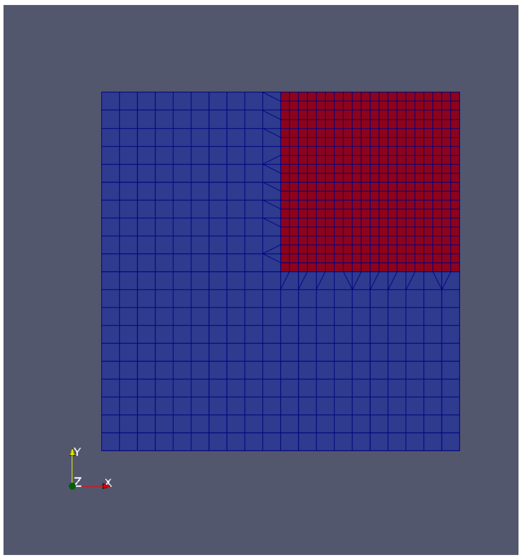


Figure 16: An example of a refined mesh. The refined region is marked in red.

14.2.2 Pifalls

If the tool *refineMesh* is called without any command line parameters then the whole mesh is refined. For *refineMesh* to obey the rules set in the `refineMeshDict` the command line option `-dict` has to used when calling *refineMesh*. See this useful post in the CFD-Online Forum `http://www.cfd-online.com/Forums/openfoam-meshing-utilities/61518-blockmesh-cellset-refinemesh.html#post195725`

Notice the different meaning of the `-dict` command line option of the tools *topoSet* and *refineMesh*. If you are in doubt about this difference, check the summary of the command line usage printed by the `-help` option.

14.3 *renumberMesh*

The tool *renumberMesh* modifies the arrangement of the cells of the mesh in order to create lower bandwidth for the numerical solution. For further information about the role and the influence of the bandwidth in numerical simulation see books on the numerical solution of large equation systems.

Renumbering the mesh can reduce computation times as it re-arranges the data to benefit the numerical solution of the resulting equation system.

14.4 *subsetMesh*

14.5 *createPatch*

14.6 *stitchMesh*

15 Initialize Fields

15.1 Basics

There are two ways to define the initial value of a field quantity. The first is to set the field to a uniform value. Listing 112 shows the `0/U` file of the *cavitiy* tutorial. There the internal field is set to a uniform value.

If a non-uniform initialisation is desired, then a list of values for all cells is needed instead. Listing 119 shows some lines of such a definition. Entering such a nonuniform list by hand would be very tiresome. To spare the user of such a painful and exhausting task, there are some tools to provide help.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }

    fixedWalls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    frontAndBack
    {
        type            empty;
    }
}
```

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

Listing 112: The file `0/U` of the *cavity* tutorial

15.2 *setFields*

setFields is a utility that allows to define geometrical regions within the domain and to assign field values to those regions. *setFields* reads this definitions from a file in the *system*-directory – the *setFieldsDict*. To initialize the field quantities `setFields` has to be executed after creating the mesh. *setFields* needs to read all files defining the mesh[27].

In Listing 113 a box is defined in which the field *alpha1* is set to a different value.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      setFieldsDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
defaultFieldValues
(
    volScalarFieldValue alpha1 1
);

regions
(
    boxToCell
    {
        box (-0.3 -0.3 0) (0.3 0.3 0.26);

        fieldValues
        (
            volScalarFieldValue alpha1 0
        );
    }
);
// ************************************************************************* //
```

Listing 113: *setFieldsDict*

Pitfall: Geometric region is not part of the domain

If the geometric region, in which to initialise a field with a specified value, lies outside the domain, *setFields* does not issue any warning or error message.

Pitfall: Geometric region covers the whole domain

This may happen if the geometric region is defined with respect to the vertex coordinates found in *blockMeshDict*. When the vertex coordinates are entered in millimeters – and *convertToMeters* is set appropiately – then it may happen, that the geometric region, based on the vertex coordinates in millimeters, is too large by the factor of 1000.

Listing 114 and 115 show the root of such a situation. The plan is to create a box and initialise it in a way, that the domain is half filled with one phase. The definition of the box in the *setFieldsDict* relies solely on the vertex coordinates ignoring the scaling factor *convertToMeters* resulting in a way too large box. After

[27] Only the file *neighbour* can be missing for *setFields* not to crash.

executing *setFields* the domain is completely filled with one phase instead of half filled.

```
convertToMeters 1e-3;

vertices
(
  (0     0     0)
  (50    0     0)
  (50    0     250)
  (0     0     250)
  (0     50    0)
  (50    50    0)
  (50    50    250)
  (0     50    250)
);
```

Listing 114: *blockMeshDict* entry for a box of $50 \times 50 \times 250\,\mathrm{mm}$

```
regions
(
  boxToCell
  {
    box (0.0 0.0 0.0) (50.0 50.0 125.0);

    fieldValues
    (
      volScalarFieldValue alpha1 0
    );
  }
);
```

Listing 115: *setFieldsDict* entry for a box of $50 \times 50 \times 125\,\mathrm{m}$

Pitfall: Field not found

If the *setFieldsDict* specifies a field which is not present, then OpenFOAM issues an error message similar to Listing 116. In this case the file *setFieldsDict* was copied from a case which uses the old naming scheme of *twoPhaseEulerFoam*, i.e. *alpha* instead of *alpha1*. See Section 32.1.1 for further information about the naming scheme. Therefore, the dictionary contained a definition for the field *alpha* which was not present in the *0*-directory.

```
Setting field default values
--> FOAM Warning :
    From function void setCellFieldType(const fvMesh& mesh, const labelList& selectedCells,
    Istream& fieldValueStream)
    in file setFields.C at line 103
    Field alpha not found


--> FOAM FATAL IO ERROR:
wrong token type - expected word, found on line 19 the label 1

file: /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bubbleColumn/system/setFieldsDict::
    defaultFieldValues at line 19.

    From function operator>>(Istream&, word&)
    in file primitives/strings/word/wordIO.C at line 74.

FOAM exiting
```

Listing 116: Missing field

15.3 *mapFields*

mapFields is a utility to transfer field data from a source mesh to target mesh. This may be useful after the mesh of case has been refined and existing solution data is to be used for initialising the case with the refined mesh. *mapFields* preserves the format of the data, if the source data was stored in binary format, the target data will also be binary.

To use *mapFields* the file *mapFieldsDict* has to be existent in the *system* folder of the case[28]. *mapFields* expects as the only mandatory argument the path to the source case. The current directory is assumed to be the case directory of the target case. If there is no specification regarding time, the latest time steps of both cases are processes. That means the latest time step of the source case is mapped to the latest time step of the target case.

Listing 117 shows the last lines of output of *mapFields*. With lines like `interpolating alpha` *mapFields* indicates that it is processing some field data. Even when source and target meshes are equal and no interpolation is needed, *mapFields* displays lines like `interpolating alpha` anyway.

```
Source time: 0.325
Target time: 0
Create meshes

Source mesh size: 81000 Target mesh size: 273375

Mapping fields for time 0.325

    interpolating alpha
    interpolating p
    interpolating k
    interpolating epsilon
    interpolating Theta
    interpolating Ub
    interpolating Ua

End
```

Listing 117: Output of *mapFields*

15.3.1 Pitfall: Missing files

mapFields issues no warning or error message when the source case contains no data. Listing 118 shows the output of *mapFields* as the target case contained no *0*-directory. Only the missing lines containing statements like `interpolating alpha` indicate that something is amiss and no field data is processed.

```
Source time: 0.325
Target time: 0
Create meshes

Source mesh size: 81000 Target mesh size: 273375

Mapping fields for time 0.325


End
```

Listing 118: Output of *mapFields*; Missing target *0*-directory

15.3.2 Pitfall: Unsuitable files

In the files containing the field data the values of the boundary fields as well as the values of the internal fields can be entered homogeneously (by the keyword `uniform`) or inhomogeneously (with the keyword `nonuniform`). Inhomogeneous field values have to be entered as a list of values. This list is preceded by the number of entries as well as the nature of the value. Listing 119 shows the beginning lines of the definition of a nonuniform vector field. The general syntax for such a list is the following:

```
nonuniform List<TYPE> COUNT ( VALUES )
```

[28]In the most basic case *mapFieldsDict* contains no other information than the header and empty definitions. Although this file may seem of no use, it has to exist in the *system* folder, and it has to contain the header and the empty definitions.

the list. A wrong value of COUNT leads to reading errors.

If data is to be mapped from a source case, the source case's data will always be stored as a nonuniform list. Otherwise, mapping the data would make no sense, as uniform fields are most easily defined. If the data of the target case is uniform, then mapping makes no problems.

If the data of the target case is nonuniform – for whatever reason – then it is necessary that the nonuniform lists have the same length. Otherwise, mapFields will exit with an error message like in Listing 120. The target case should always be set up with uniform fields to avoid such errors. This is most easily done by removing the definition of the internal field. In the tutorials sometimes files with an `.org` file extension can be found. This is a way to preserve the uniform field data in the *0*-directory without causing any trouble.

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

internalField   nonuniform List<vector>
1600
(
(0.000174291 -0.000171512 0)
(0.000171022 -0.000143648 0)
(-0.000259297 0.000305772 0)
(-0.000380671 0.000374937 0)
(-0.00182755 0.000930701 0)
```

Listing 119: An inhomogeneous internal field definition in the file `0/U`

```
Mapping fields for time 0.325

  interpolating alpha

--> FOAM FATAL IO ERROR:
size 81000 is not equal to the given value of 10125

file: /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/Case/0/alpha from line 18 to line
    39.

  From function Field<Type>::Field(const word& keyword, const dictionary&, const label)
  in file /home/user/OpenFOAM/OpenFOAM-2.1.x/src/OpenFOAM/lnInclude/Field.C at line 236.

FOAM exiting
```

Listing 120: Error message of *mapFields*; unequal number of values

15.3.3 Pitfall: Mapping data from a 2D to a 3D mesh

In this section we deal with some difficulties of the *mapFields* utility. We have finished a simulation on a 2D mesh. The geometry of the 2D case is 20 cm × 2 cm × 45 cm.

Now we want to transfer the 2D data to a 3D mesh to initialise the 3D simulation. The geometry of the 3D simulation is 20 cm × 5 cm × 45 cm. Note the different dimension in *y*-direction.

Listing 121 shows the `mapFieldsDict` that was used. Because of the great similarity of the geometry, no entries are necessary.

The problem

Figure 17 shows the result of the *mapFields* run. Only the field values inside the 2D domain were altered. The part of the 3D domain that lies outside the 2D domain remains unchanged. This behaviour is not satisfactory.

The work-around

One way to solve this problem would be to choose the 2D domain of a similar size as the 3D domain. However, if the 2D is already finished, then it would take some time to re-simulate the case with a redefined geometry.

Another solution is:

1. define the 3D domain to be of the same size as the 2D domain

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
location    "system";
object      mapFieldsDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

patchMap         ( );

cuttingPatches   ( );

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

Listing 121: The file `mapFieldsDict`

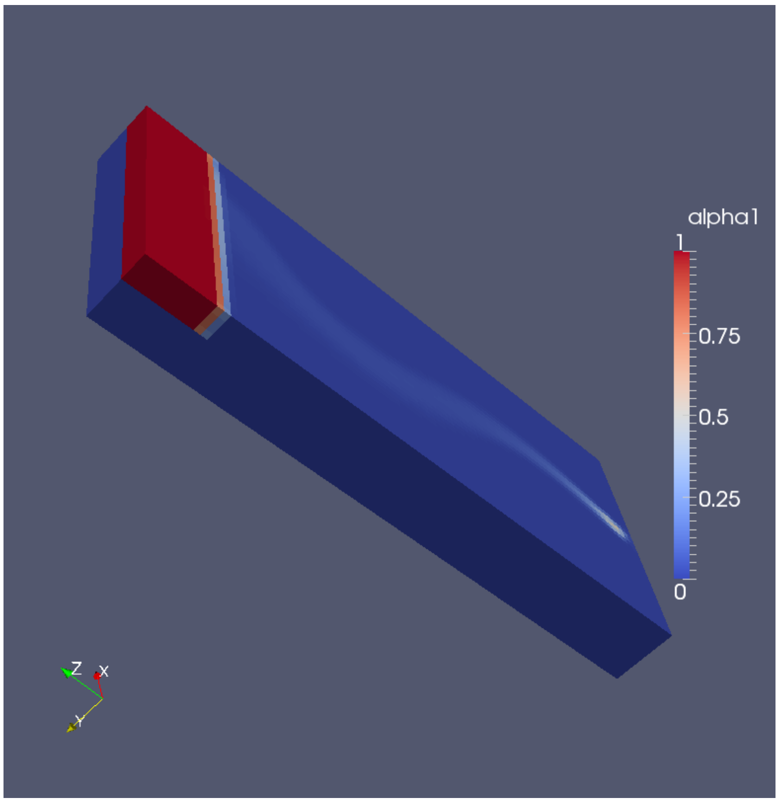


Figure 17: The mapped field

2. map the fields
3. redefine the 3D domain to its intended size, without changing the total number of cells

15.3.4 The work-around: Mapping data from a 2D to a 3D mesh

The work-around to the problem of the previous section is rather unelegant. A 2D mesh that has the same depth as the 3D mesh but is discretised with only 1 cell in depth will have a very bad aspect ratio.

A more elegant solution is to transform the mesh after the 2D simulation has finished. In our example, the 2D mesh has the dimensions 20 cm × 2 cm × 45 cm and the 3D mesh is 20 cm × 5 cm × 45 cm big.

With the tool *transformPoints* the mesh can be scaled selectively in the three dimensions of space. Listing 122 shows how *transformPoints* can be used to scale the 2D mesh in y-direction by the factor of 2.5. After this scaling operation the 2D mesh has the desired dimensions of 20 cm × 5 cm × 45 cm.

```
transformPoints -scale '(1.0 2.5 1.0)'
```

Listing 122: Scaling the 2D mesh in y-direction with *transformPoints*

After the mesh transformation the utility *mapFields* can be used to map the field from the scaled 2D mesh to the 3D mesh.

15.3.5 The importance of mapping

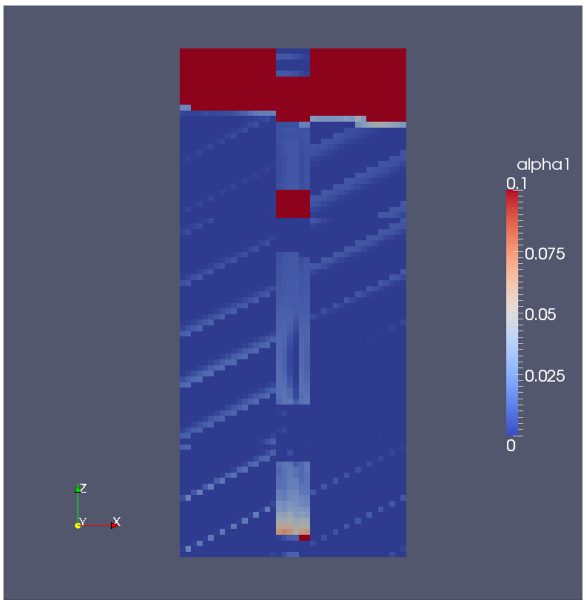
The purpose of this example is to highlight the need for the *mapFields* utility. A simulation of the bubble column has been made. Now, the user decides to change the size of the inlet patch. Thanks to the parametric mesh, this can be done easily only by changing some numbers in the file `blockMeshDict.m4`. See Section 11.5 for a discussion on creating a parametric mesh.

After the user changed the coordinates of some points, meshing yields a new mesh with the same number of cells as the old mesh had. Because the number of cells did not change, the data files from the finished simulation fit the new one. The user simply copies the necessary files from the latest time step of the finished simulation to the initial time step of the new simulation.

Starting the simulation resulted in a floating point exception. However, after reducing the time step, the simulation proceeded without any further errors. Figure 18 shows the initial `alpha` and `U1` fields of the new simulation. Due to a change in the numbering of the cells, the formerly smooth fields are now completely distorted. The single blocks of the mesh can be distinguished from the figures. This indicates, that OpenFOAM numbers the cells block-wise.

15.3.6 Pitfall: binary files

If the source case has binary data files, then the boundary conditions need to be defined before mapping the fields. Therefore, the boundary conditions need to be defined in a suitable ascii file. Then, the fields can be mapped. Editing a binary file with a text editor may render this file defective.



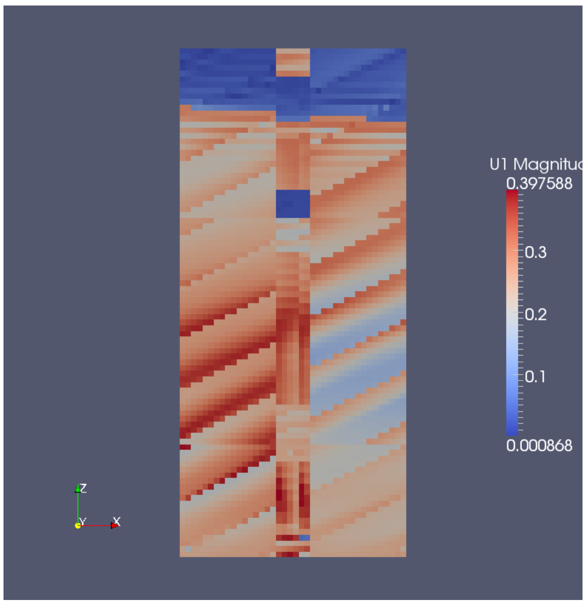


Figure 18: The unmapped fields

Part IV
Modelling

16 Turbulence-Models

16.1 Categories

The desired category of turbulence models can be specified in the file `turbulenceProperties`. There are three possible entries.

laminar The flow is modelled laminar

RASModel A Reynolds averaged turbulence model (RAS-model) is used.

LESModel Turbulence is modelled by a *large-eddy* model.

The file `turbulenceProperties` contains only one entry. In case of a large eddy simulation, this entry reads:

```
simulationType   LESModel;
```

Listing 123: *turbulenceProperties*

16.2 RAS-Models

The entry in the file `turbulenceProperties` specifies only the class of turbulence models. The exact turbulence model is specified in the file `RASProperties`. This file must contain all necessary parameters.

Listing 124 shows the content of `RASProperties`. In this case a k-ϵ model is used and no further parameters are necessary.

```
RASModel          kEpsilon;
turbulence        on;
printCoeffs       on;
```

Listing 124: *RASProperties*

Depending on the exact model more parameters can be necessary.

16.2.1 Keywords

RASModel The name of the turbulence model. At this place laminar can also be chosen. The banana test (see Section 7.1.1) delivers a list of available models.

```
--> FOAM FATAL ERROR:
Unknown RASModel type banana

Valid RASModel types:

17
(
  LRR
  LamBremhorstKE
  LaunderGibsonRSTM
  LaunderSharmaKE
  LienCubicKE
  LienCubicKELowRe
  LienLeschzinerLowRe
  NonlinearKEShih
  RNGkEpsilon
  SpalartAllmaras
  kEpsilon
  kOmega
  kOmegaSST
```

```
    kkLOmega
    laminar
    qZeta
    realizableKE
)
```

Listing 125: Possible RAS-model entries in *RASProperties*

turbulence This is a switch to activate or deactivate the turbulence modelling. Allowed values are: *on/off, true/false* or *yes/no*.
Is this switch is deactivated, then a laminar simulation is conducted. This way of choosing a laminar model is not recommended, see Section 16.4.1.

printCoeffs If this switch is enabled, then the solver will display the coefficients of the selected turbulence model.
Even if the switch `turbulence` is disabled, the solver will display the coefficients at the beginning of the simulation, see Listing 132. Only, when `RASModel laminar` is chosen, no coefficients are displayed.

optional parameters Some models accept optional parameters to override the default values of the model. Listing 126 shows how the coefficients of the k-ϵ model can be overridden.

```
kEpsilonCoeffs
{
    Cmu          0.09;
    C1           1.44;
    C2           1.92;
    C3           -0.33;
    sigmak       1.0;
    sigmaEps     1.11; //Original value:1.44
}
```

Listing 126: Definition of model parameters in *RASProperties*

16.2.2 Pitfall: meaningless Parameters

In the above section it was shown how to override default values of the model constants. In this procedure, there is one source of error hidden. This is not an actual error, but it can lead to a fruitless search for an error.

If nonsensical parameters are added to the `kEpsilonCoeffs` dictionary, these will be read and also printed. Listing 127 shows the `kEpsilonCoeffs` dictionary of the file `RASProperties`. This dictionary is used to override default values of the model constants. A fake model constant has been added to this dictionary.

Listing 128 shows parts of the solver output, when this dictionary is used in a simulation. All constants of the dictionary are read and printed again. It seems as if the constant `banana` is part of the turbulence model. Varying this parameter yields no results, which is no error.

The reason for this behaviour is, there is no check whether the defined constants in the dictionary make sense or not.

```
kEpsilonCoeffs
{
    Cmu          0.09;
    C1           1.44;
    C2           1.92;
    C3           -0.33;
    sigmak       1.0;
    sigmaEps     1.11; //Original value:1.44
    banana       0.815; // nonsense parameter
}
```

Listing 127: Definition of model parameters in *RASProperties*

```
Selecting RAS turbulence model kEpsilon
kEpsilonCoeffs
{
  Cmu                  0.09;
  C1                  1.44;
  C2                  1.92;
  C3                  -0.33;
  sigmak              1.0;
  sigmaEps            1.11;
  banana              0.815;
}

Starting time loop
```

Listing 128: Solver output

16.3 LES-Models

16.3.1 Keywords

The keywords `turbulence` and `printCoeffs` have the same meaning with LES models. There is also the possibility – depending on the selected model – of defining optional parameters.

LESModel The name of the turbulence model. At this place laminar can also be chosen. The banana test (see Section 7.1.1) delivers a list of available models. Listing 129 shows the result of such a banana test. The model dynamicSmagorinsky was loaded from an external library. See Section 7.2.3 for how to include external libraries.

```
--> FOAM FATAL ERROR:  Unknown LESModel type banana

Valid LESModel types:

16
(
  DeardorffDiffStress
  LRRDiffStress
  Smagorinsky
  SpalartAllmaras
  SpalartAllmarasDDES
  SpalartAllmarasIDDES
  dynLagrangian
  dynOneEqEddy
  dynamicSmagorinsky
  homogeneousDynOneEqEddy
  homogeneousDynSmagorinsky
  kOmegaSSTSAS
  laminar
  mixedSmagorinsky
  oneEqEddy
  spectEddyVisc
)
```

Listing 129: Possible LES-model entries in *LESProperties*

16.3.2 Algebraic sub-grid models

Algebraic sub-grid models introduce no further transport equation to the simulation. The turbulent viscosity is calculated from existing quantities.

16.3.3 Dynamic sub-grid models

The dynamic sub-grid models calculate the model constant C_S from known quantities instead of prescribing a fixed value. The way how C_S is calculated is determined by the sub-grid model.

16.3.4 One equation models

A further class of LES turbulence models are one equation models. These models add one further equation to the problem. Usually, an additional equation for the sub-grid scale turbulent kinetic energy is solved.

16.4 Pitfalls

16.4.1 Laminar Simulation

As already mentioned – see Section 16.2 – turbulence modelling can be deactivated in a some ways.

the list here lists different ways to conduct a laminar simulation. This list applys only to solvers that utilize the generic turbulence modelling of OpenFOAM:

1. **turbulenceProperties**: `simulationType laminar`
   This is the most general way to turn turbulence modelling off. `turbulenceProperties` controls the generic turbulence class. The generic turbulence class can take the form of the `laminar`,`RASModel` or `LESModel` class, see Figure 35. This is controlled by the parameter `simulationType`.

```
Selecting turbulence model type laminar
```

Listing 130: Solver output for `simulationType laminar`

2. **RASProperties**: `RASModel laminar`
   LESProperties: `LESModel laminar`
   In this case, a certain turbulence modelling strategy is chosen (`RASModel` or `LESModel`). However, there is a dummy turbulence model for laminar simulation. This dummy turbulence model is derived from the base class `RASModel` but it implements a laminar model. See Figure 36. Therefore, `RASModel laminar` selects the laminar RAS turbulence model. In this point `RASModel` and `LESModel` behave similar.

```
Selecting turbulence model type RASModel
Selecting RAS turbulence model laminar
```

Listing 131: Solver output for `RASModel laminar`

3. **RASProperties**: `turbulence off`
   The switch `turbulence` can be used to enable or disable turbulence modelling. When the calculation is started, the turbulence model specified is used. However, in the source code of the solver, there is the test whether turbulence modelling is active or not. See Listing 147.

```
Selecting turbulence model type RASModel
Selecting RAS turbulence model kEpsilon
kEpsilonCoeffs
{
    Cmu          0.09;
    C1           1.44;
    C2           1.92;
    sigmaEps     1.3;
}
```

Listing 132: Solver output for `turbulence off`

Solver output

The last two prossibilities to conduct a laminar simulation can lead to confusion because the solver output contains word like `RASmodel` or `RAS turbulence model`. See Listings ?? and ??. In both cases the simulation is laminar. In order to avoid this source of confusion, the user should use the parameter `simulationType` to perform a laminar calculation.

Independent from all other settings, `printCoeffs` prints the model constants of the selected turbulence model. This may also lead to confusion, when e.g. `turbulence off` is chosen to conduct a laminar simulation.

Exceptions

The above explanation only applies to solvers that utilize the generic turbulence models of OpenFOAM. However, there is no rule without its exceptions.

simpleFoam This solver uses only RAS turbulence models. Therefore, the entries of the file `turbulenceProperties` are redundant and the only ways to control turbulence modelling are items 2 and 3 of the list above.

twoPhaseEulerFoam This solver has the k-ϵ turbulence model hardcoded. Only item 3 of the list above applies to this solver. See Section 16.4.2 for a detailled discussion.

bubbleFoam The same as *twoPhaseEulerFoam*.

multiphaseEulerFoam This solver only uses LES turbulence models. Items 2 and 3 of the list above apply.

16.4.2 Turbulence models in *twoPhaseEulerFoam*

In the solver *twoPhaseEulerFoam,* the use of the k-ϵ turbulence model is hardcoded. This means that the solver does not use the generic turbulence modelling ususally used by OpenFOAMs solvers. The only choice the user of *twoPhaseEulerFoam* has is whether to enable or disable the k-ϵ turbulence model.

For this reason, the file `constant/turbulenceProperties` is not needed any more. This file can savely be deleted.

Another consequence of the k-ϵ turbulence model being hardcoded into *twoPhaseEulerFoam* is that the keyword `turbulenceProperties` in the file `RASproperties` is also not needed any more. This entry is only read if the generic turbulence modelling is used and if there is any choice of which RAS-model to use. The only mandatory keyword in `RASproperties` is the switch `turbulence`. This switch is the only way to decide whether to use turbulence modelling or not with *twoPhaseEulerFoam.* Solvers which use the generic turbulence modelling offer three possible ways to disable turbulence modelling, see Section 16.4.1.

16.4.3 Laminar simulation with *twoPhaseEulerFoam*

If *twoPhaseEulerFoam* is used and a laminar simulation is conducted, then the presence of the files like `0/k` or `0/epsilon` is mandatory. The solver read this files regardless of the fact, that a laminar simulation is conducted. This is due to the fact that the use of the k-ϵ model is hardcoded into *twoPhaseEulerFoam.*

Other solvers read this files based on the condition if and which turbulence model is used. Otherwise there would be the need for all possible files (`0/k`, `0/epsilon`, `0/omega`, etc.) to be present in any case, which would be utter madness.

16.4.4 Initial and boundary conditions

All turbulence models can be divided into classes depending on their mathematical properties.

Algebraic models These models add an algebraic equation to the problem. The turbulent viscosity is computed from known quantities using an algebraic equation (e.g. the Baldwin-Lomax model)

One equation models These models introduce an additional transport equation to the problem. The eddy viscosity is computed from this additional quantity (e.g. the Spalart-Allmaras model)

Two equation models These models introduce two additional transport equations to the problem. The eddy viscosity is computed from these additional quantities (z.B. k-ϵ, k-ω)

Every field quantity of a turbulence model needs its initial and boundary conditions. Consequently, there may be the need for additional files in the *0*-directory. One way to find out which files are needed is to look at the tutorials. There, a case may be found which utilises the needed turbulence model.

If a simulation is started and the solver is missing files – i.e. the solver tries to read files which are not present – then OpenFOAM will issue a corresponding error message. Listing 133 shows an error message of a case with a missing `0/k` file.

```
Selecting turbulence model type RASModel
Selecting RAS turbulence model kEpsilon
--> FOAM FATAL IO ERROR: cannot find file
file: /home/user/OpenFOAM/user-2.1.x/run/pisoFoam/cavity/0/k at line 0.

    From function regIOobject::readStream()
    in file db/regIOobject/regIOobjectRead.C at line 73.

FOAM exiting
```

Listing 133: Solver error message: missing file

16.4.5 Additional files

RAS turbulence models produce additional files. Most RAS models calculate the turbulent viscositiy from certain quantities. These quantities are usually field quantities and depend on the used turbulence model. However, the aim of all RAS turbulence models is to calculate the turbulent viscosity. The turbulent viscosity itself is a field quantity.

Listing 134 shows the folder contents before and after a simulation with *pisoFoam*. The *0*-directory contains only the mandatory files, in this case pressure and velocity as well as the turbulent quantities k and ϵ.

After the simulation has finished, the *0*-directory contains more files. The reason for creating the `*.old` files is not known. However, the turbulence model created the file `nut` for storing the turbulent viscosity.

The file `phi` as well as the folder `uniform` is created by the solver.

```
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$ ls
0  constant  system
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$ ls 0/
epsilon  k  p  U
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$ pisoFoam > /dev/null
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$ ls
0  0.5  1  constant  system
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$ ls 0/
epsilon  epsilon.old  k  k.old  nut  p  U
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$ ls 0.5/
epsilon  k  nut  p  phi  U  uniform
user@host:~/OpenFOAM/user-2.1.x/run/pisoFoam/ras/cavity$
```

Listing 134: Folder contents at the begin and the end of a simulation

The *0*-directories of some tutorial cases may already contain such additional files, e.g. `nut`. In some cases the 0-directory may also contain several of such files due to a change in the naming scheme. Listing 135 shows the contents of the *0*-directory of the *pitzDaily* tutorial case of *simpleFoam*. The case has not been run, so the files `nut` and `nuTilda` have not been generated by the solver. None of these two files is necessary to run the case with the k-ϵ turbulence model.

```
epsilon  k  nut  nuTilda  p  U
```

Listing 135: The content of the *0*-directory of the *pitzDaily* tutorial case of *simpleFoam*

16.4.6 Spalart-Allmaras

The Spalart-Allmaras is a one-equation turbulence model. Although it introduces only one additional equation to the problem it needs two additional files in the 0-directory. Listing 136 shows the content of the *0*-folder of the *airFoil2D* tutorial case of *simpleFoam*. The files `nut` and `nuTilda` are both necessary to run the case. The former contains the turbulent viscosity and the latter contains the transported quantity of the turbulence model. Therefore, the rule *one additional transport equation entails one additional data file* is not violated.

Because the viscosity is not constant it has to be defined in a file in the *0*-directory. And, because the viscosity is not the transported quantity of the Spalart-Allmaras model another file is added to the *0*-directory.

```
nut  nuTilda  p  U
```

Listing 136: The content of the *0*-directory of the *airFoil2D* tutorial case of *simpleFoam*

17 Boundary conditions

When the geometry of a problem is meshed, then the boundary patches – i.e. the faces delimiting the geometry – need to be specified. Every boundary patch is of a certain type. In Section 17.1 the possible types are discussed.

17.1 *Base types*

17.1.1 Geometric boundaries

Some kinds of boundary patches can be described purely geometrically. The numerical treatment of this kind of patches is inherently clear to the solver and needs no more modelling.

symmetry plane If a problem is symmetric, then only half of the domain needs to be modelled. The boundary that lies in the symmetry plane is of type *symmetry plane*.

empty OpenFOAM creates always three-dimensional meshes. If a two-dimensional simulation needs to be conducted, then the mesh must be one cell in thickness. The boundaries that are parallel to the considered plane must be of the type *empty* to cause the simulation to be two-dimensional.

wedge If a geometry is axisymmetric, then the problem can be simplified. In this case, only a part of the geometry – a wedge – is modelled. The additional boundaries are of type *wedge*.

cyclic Cyclic boundary.

processor A boundary between sub-domains created during the domain decomposition is of type *processor*.

17.1.2 Complex boundaries

Some kinds of boundary patches are more than just a geometric boundary of the domain. E.g. on a wall, the no-slip condition usually applies, therefore there is need for further modelling.

patch This is the generic type for all boundaries. A boundary is of this type, if none of the following types applies.

wall This is a special type for walls. This type is mandatory for using wall models when modelling turbulence.

The boundaries of the types *patch* and *wall* need to be specified further. These boundaries can have boundary conditions of the *primitive* or *derived* types.

17.2 *Primitive types*

The most important *primitive type* boundary conditions are:

fixedValue The value of a quantity is prescribed directly.

fixedGradient The gradient of a quantity is prescribed directly.

zeroGradient The gradient of a quantity is prescribed to zero.

```
type            fixedValue;
value           uniform (0 0 0);
```

Listing 137: `fixedValue` boundary condition

17.3 *Derived types*

The boundary condition of the *derived types* are derived from the boundary conditions of the *primitive types*. The boundary conditions of this type can be used to model more complex situations.

17.3.1 inletOutlet

The behaviour of the *inletOutlet* boundary condition depends of the flow direction. If the flow is directed outwards, then a *zeroGradient* boundary condition is applied. If the flow is inwards, then a fixed value is prescribed. The value of the inflowing quantity is provided by the `inletvalue` keyword. The `value` keyword has to be present, but it is not relevant.

```
type            inletOutlet;
inletValue      uniform (0 0 0);
value           uniform (0 0 0);
```

Listing 138: `inletOutlet` boundary condition

17.3.2 surfaceNormalFixedValue

The *surfaceNormalFixedValue* boundary condition prescribes the norm of a vector field. The direction is taken from the surface normal vector of the patch. A positive value for `refValue` means, that this quantity is directed in the same direction as the surface normal vector. A negative value means the opposite direction.

```
type            surfaceNormalFixedValue;
refValue        uniform -0.1;
```

Listing 139: `surfaceNormalFixedValue` boundary condition

17.3.3 pressureInletOutletVelocity

This boundary condition is a combination of *pressureInletVelocity* and *inletOutlet*.

17.4 Pitfalls

17.4.1 Syntax

When assigning a `fixedValue` boundary condition, OpenFOAM expects the keyword `uniform` or `nonuniform` after the `value` keyword.

Listing 140 shows the file `0/k`. There the inlet boundary definition differs from Listing 137. Note the missing `uniform` keyword. The reaction of OpenFOAM differs from the value after the keyword `version`.

Listing 141 shows the warning message OpenFOAM issues, when the value after the keyword `version` is `2.0` like in Listing 140. In this case, OpenFOAM assumes `uniform`.

If the value after the keyword `version` is `2.1`, then OpenFOAM will issue an error message like in Listing 142.

In both cases OpenFOAM-2.1.x was used. The author assumes the reason for this distinction between version 2.0 and 2.1 lies in an extension of the possible boundary conditions See the release notes of OpenFOAM-2.1.0 (`http://www.openfoam.org/version2.1.0/boundary-conditions.php`).

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      k;
}

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

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

internalField   uniform 1e-8;

boundaryField
{
    inlet
    {
        type        fixedValue;
        value       1e-8;
    }
```

Listing 140: The file `0/k`

```
--> FOAM Warning :
    From function Field<Type>::Field(const word& keyword, const dictionary&, const label)
    in file /home/user/OpenFOAM/OpenFOAM-2.1.x/src/OpenFOAM/lnInclude/Field.C at line 262
    Reading "/home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bubblePlume/case/0/k::
    boundaryField::inlet" from line 25 to line 26
    expected keyword 'uniform' or 'nonuniform', assuming deprecated Field format from Foam
    version 2.0.
```

Listing 141: Warning message: missing keywords

```
--> FOAM FATAL IO ERROR:
expected keyword 'uniform' or 'nonuniform', found on line 26 the doubleScalar 1e-08

file: /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bubblePlume/case/0/k::boundaryField
    ::inlet from line 25 to line 26.

    From function Field<Type>::Field(const word& keyword, const dictionary&, const label)
    in file /home/user/OpenFOAM/OpenFOAM-2.1.x/src/OpenFOAM/lnInclude/Field.C at line 278.

FOAM exiting
```

Listing 142: Warning message: missing keywords

17.5 Time-variant boundary conditions

Time-variant boundary conditions can help to avoid problems from an inept initialisation of the solution data. The most easy initialisation is to prescribe all values to be zero throughout the domain, see Listing 112 in Section 15.

At the start of a simulation when the non-zero values of some boundary meet the zero values of the neighbouring cells stability problems may arise due to the large relative velocities. One solution would be to choose a very small time step at the beginning. Another solution would be to prescribe a time-variant boundary condition. Thus, the field-values at the boundary are initially small and grow during a certain time span to their final value.

17.5.1 `uniformFixedValue`

This boundary condition is an generalisation of the `fixedValue` BC. See `http://www.openfoam.org/version2.1.0/boundary-conditions.php`.

Listing 143 shows the definition of a time-variant boundary condition with a fixed value. Between the time $t = 0.0\,\mathrm{s}$ and $t = 5.0\,\mathrm{s}$ the value of the boundary condition is linearly interpolated between the values for both ends of the interval. After this interval has ended, the value of the boundary condition remains constant.

```
inlet
{
  type            uniformFixedValue;
  uniformValue    table
  (
    ( 0.0   (0.0 0.0 0.0) )
    ( 5.0   (0.0 0.0 0.1) )
  );
}
```

Listing 143: Definition of a time-variant boundary condition

Pitfall: Two-phase solvers

This boundary condition does not work with two-phase solvers.

Part V
Solver

18 Solution Algorithms

The solution of the Navier-Stokes equations require the solution of the coupled equations for the velocities and the pressure field. In order to be able to gain a solution, there are several solution algorithms. All of these algorithms try to compute velocities and pressure seperately and therefore decouple the problem.

To decouple the computation of velocity and pressure a predictor-corrector strategy is followed.

18.1 SIMPLE

Figure 19 shows the flow chart of the SIMPLE algorithm. The SIMPLE algorithm predicts the velocity and then corrects both the pressure and the velocity. This is repeated until a convergence criteria is reached. The labels in Figure 19 are related to the terminology used in the source code of the `simpleFoam` solver. The solution procedure can be described as follows

1. Check if convergence is reached – `simple.loop()`
2. Predict the velocities using the momentum predictor– `UEqn.H`
3. Correct the pressure and the velocities– `pEqn.H`
4. Solve the transport equations for the turbulence model[29]– `turbulence->correct()`
5. Go back to step 1

In OpenFOAM the SIMPLE algorithm is used for steady-state solvers.

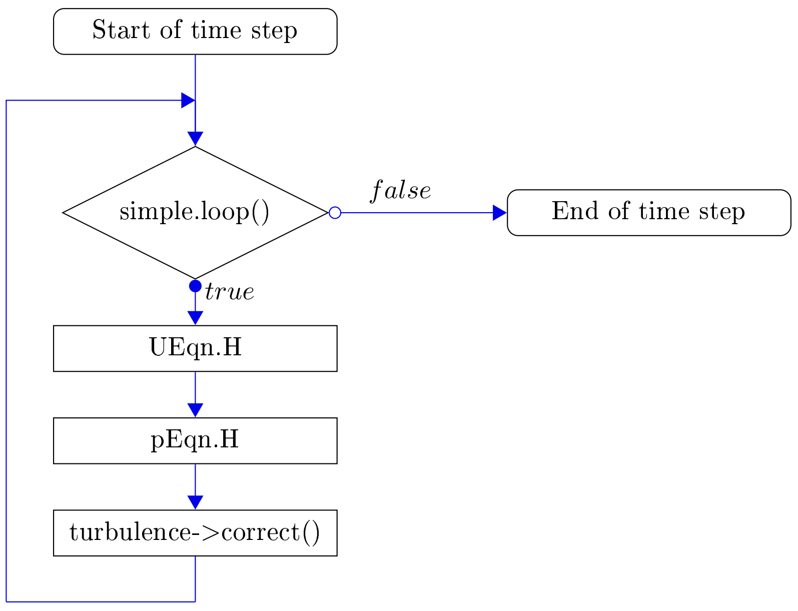


Figure 19: Flow chart of the SIMPLE algorithm

[29]In case of a laminar simulation an empty function is called. Turbulence is modelled in OpenFOAM in a very generic way. Therefore, a laminar simulation uses the `laminar` turbulence model.

18.1.1 Predictor

The predictor of *simpleFoam* is a momentum predictor.

```
// Momentum predictor
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevReff(U)
  ==
    sources(U)
);

UEqn().relax();

sources.constrain(UEqn());

solve(UEqn() == -fvc::grad(p));
```

Listing 144: Predictor in *UEqn.H* of *simpleFoam*

18.1.2 Corrector

The corrector is used to correct the pressure field by using the predicted velocity. This corrected pressure is used to correct the velocities by solving the continuity equation.

The non-orthogonal pressure corrector loop is necessary only for non-orthogonal meshes [14].

```
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
while (simple.correctNonOrthogonal())
{
  fvScalarMatrix pEqn
  (
    fvm::laplacian(rAU, p) == fvc::div(phi)
  );
  pEqn.setReference(pRefCell, pRefValue);

  pEqn.solve();

  if (simple.finalNonOrthogonalIter())
  {
    phi -= pEqn.flux();
  }
}

#include "continuityErrs.H"

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

// Momentum corrector
U -= rAU*fvc::grad(p);
U.correctBoundaryConditions();
sources.correct(U);
```

Listing 145: Corrector in *pEqn.H* of *simpleFoam*

18.2 PISO

The PISO algorithm also follows the predictor-corrector strategy. Figure 20 shows the flow chart of the PISO algorithm. The velocity is predicted using the momentum predictor. Then, the pressure and the velocity is corrected until a predefined number of iterations is reached. Afterwards, the transport equations of the turbulence model are solved.

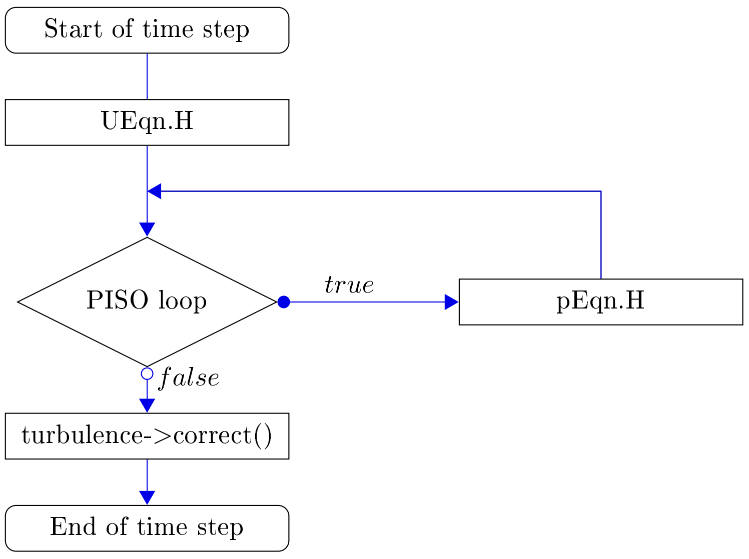


Figure 20: Flow chart of the PISO algorithm

19 *pimpleFoam*

pimpleFoam is a transient incompressible solver. The solver is described in the file `pimpleFoam.C` as follows:

```
Large time-step transient solver for incompressible, flow using the PIMPLE
(merged PISO-SIMPLE) algorithm.

Turbulence modelling is generic, i.e. laminar, RAS or LES may be selected.
```

19.1 Governing equations

19.1.1 Continuity equation

The general continuity equation reads as follows:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{21}$$

we now assume incompressible fluids: $\rho = const$

$$\nabla \cdot \mathbf{u} = 0 \tag{22}$$

or in alternative notation

$$\mathrm{div}(\mathbf{u}) = 0 \tag{23}$$

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{24}$$

19.1.2 Momentum equation

Departing from the Navier-Stokes equations, the momentum equation of *pimpleFoam* are derived.

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla(\rho \mathbf{u}\mathbf{u}) + \nabla \cdot \tau = -\nabla p + \mathbf{g} \tag{25}$$

because we assume a constant density we can divide by ρ

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \frac{1}{\rho}\nabla \cdot \tau = -\frac{\nabla p}{\rho} + \frac{\mathbf{g}}{\rho} \tag{26}$$

The last term is defined a general source term

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \frac{1}{\rho}\nabla \cdot \tau = -\frac{\nabla p}{\rho} + \mathbf{Q} \tag{27}$$

the shear stresses and the pressure are denoted by new symbols: $\frac{\tau}{\rho} = \mathbf{R}^{eff}$ und $\frac{p}{\rho} = p$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \nabla \cdot \mathbf{R}^{eff} = -\nabla p + \mathbf{Q} \tag{28}$$

The Boussinesq hypothesis allows us to add the Reynolds stresses to the shear stresses. This stress tensor – containing shear as well as Reynolds stresses – is denoted $\mathbf{R}^{eff}$, the effective stress tensor. Both RAS as well as LES turbulence models are based on the Boussinesq hypothesis.

$$\mathbf{R}^{eff} = -\nu^{eff}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right) \tag{29}$$

$$R_{ij}^{eff} = -\nu^{eff}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{30}$$

The trace of τ fulfills the continuity equation for incompressible fluids

$$\mathrm{tr}(\mathbf{R}^{eff}) = R_{ii}^{eff} = -2\nu^{eff}\left(\frac{\partial u_i}{\partial x_i}\right) = 0 \tag{31}$$

$$\frac{\partial u_i}{\partial x_i} = \nabla \cdot \mathbf{u} = 0 \tag{32}$$

Therefore, we can replace $\mathbf{R}^{eff}$ with the deviatoric part of $\mathbf{R}^{eff}$

$$\mathbf{R}^{eff} = \underbrace{\mathrm{dev}(\mathbf{R}^{eff})}_{\text{deviatoric part}} + \underbrace{\frac{1}{3}\mathrm{tr}(\mathbf{R}^{eff})\mathbf{I}}_{\text{hydrostatic part}} \tag{33}$$

$$\mathrm{dev}(\mathbf{R}^{eff}) = \mathbf{R}^{eff} - \frac{1}{3}\underbrace{\mathrm{tr}(\mathbf{R}^{eff})}_{=0}\mathbf{I} \tag{34}$$

Therefore, the momentum equation can be rewritten

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \underbrace{\nabla \cdot \left(\mathrm{dev}(\mathbf{R}^{eff})\right)}_{=\mathrm{div}(\mathrm{dev}(\mathbf{R}^{eff}))} = -\nabla p + \mathbf{Q} \tag{35}$$

Finally, we use Eq. (29)

$$\mathbf{R}^{eff} = -\nu^{eff}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right) \tag{29}$$

to gain

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \nabla \cdot \left(\mathrm{dev}(-\nu^{eff}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right))\right) = -\nabla p + \mathbf{Q} \tag{36}$$

19.1.3 Implementation

The momentum equation is implemented in the file `UEqn.H`. The first two terms of Eq. (36) can easily be identified in the source code in Listing 146.

The first term is the local derivative of the momentum – due to the incompressibility of the fluid, the density was eliminated – can be found in line 5 of Listing 146. Here, the instruction in the source code reads very much the same as the mathematical notation.

$$\frac{\partial \mathbf{u}}{\partial t} \qquad \Leftrightarrow \qquad \texttt{fvm::ddt(U)}$$

The second term of Eq. (36) is the convective transport of momentum. The use of the identifier `phi` should not lead to confusion. In order to read the equations from the source code, `phi` can be replaced with `U` without changing the meaning of the equations. The reason why `phi` is used in the source code lies in the solution procedure. See Section 41 for a detailled discussion about `phi`.

$$\underbrace{\nabla(\mathbf{uu})}_{\mathrm{div}(\mathbf{uu})} \qquad \Leftrightarrow \qquad \texttt{fvm::div(phi, U)}$$

The third term of Eq. (36) is the diffusive momentum transport term. Diffusive momentum transport is caused by the laminar viscosity as well as turbulence. Therefore, the turbulence model handles this term. See line 7 of Listing 146.

$$\underbrace{\nabla \cdot \left(\mathrm{dev}(\mathbf{R}^{eff})\right)}_{=\mathrm{div}(\mathrm{dev}(\mathbf{R}^{eff}))} \qquad \Leftrightarrow \qquad \texttt{turbulence->divDevReff(U)}$$

The terms on the *rhs* of Eq. (36) are the pressure gradient and the source term.

$$\underbrace{-\nabla p}_{=-\mathrm{grad}\, p} \qquad \Leftrightarrow \qquad \texttt{-fvc::grad(p))}$$

$$\mathbf{Q} \qquad \Leftrightarrow \qquad \texttt{sources(U)}$$

```
1  // Solve the Momentum equation
2
3  tmp<fvVectorMatrix> UEqn
4  (
5      fvm::ddt(U)
6    + fvm::div(phi, U)
7    + turbulence->divDevReff(U)
8  );
9
10 UEqn().relax();
11
12 sources.constrain(UEqn());
13
14 volScalarField rAU(1.0/UEqn().A());
15
16 if (pimple.momentumPredictor())
17 {
18     solve(UEqn() == -fvc::grad(p) + sources(U));
19 }
```

Listing 146: The file *UEqn.H* of *pimpleFoam*

19.2 The PIMPLE Algorithm – or, what's under the hood?

This Section deals with the way *pimpleFoam* and *twoPhaseEulerFoam*, which also uses the PIMPLE algorithm, work. Therefore, we examine the implementation of *pimpleFoam*. Listing 147 shows the main loop of *pimpleFoam*.

The first instruction is the loop over all time steps. Then there are some operations – the three `#include` instructions – concerning time step control. After incrementing the time step (Line 7), the PIMPLE loop comes (from Line 10 onwards).

Inside this loop, first the momentum equation is solved (Line 12), then the pressure correction loop is entered (Line 17).

At the end of the PIMPLE loop the turbulent equations[30] – if there are any present[31] – are solved (Line 22). At the end of each time step the data is written.

```
1  while (runTime.run())
2  {
3    #include "readTimeControls.H"
4    #include "CourantNo.H"
5    #include "setDeltaT.H"
6
7    runTime++;
8
9    // --- Pressure-velocity PIMPLE corrector loop
10   while (pimple.loop())
11   {
12     #include "UEqn.H"
13
14     // --- Pressure corrector loop
15     while (pimple.correct())
16     {
17       #include "pEqn.H"
18     }
19
20     if (pimple.turbCorr())
21     {
22       turbulence->correct();
23     }
24   }
25
26   runTime.write();
27 }
```

Listing 147: The main loop of *pimpleFoam*

Figure 21 shows the flow chart of the PIMPLE algorithm. This algorithm is executed every time step. If the PIMPLE loop is entered only once, then the algorithm is essentially the same as the PISO algorithm. Listing 154 draws this conclusion from the code itself.

19.2.1 `readTimeControls.H`

In line 3 of Listing 147 the file `readTimeControls.H` is included to the source code using the `#include` preprocessor macro. This is a very common way to give the code of OpenFOAM structure and order. Code which is used repeatedly is outsourced into a seperate file. This file is then included with the `#include` macro. Thus, code duplication is prevented. The file `readTimeControls.H` might be included into every solver that is able to use variable time steps. If this code was not outsourced into a seperate file, this code would be found in every variable time step solver. Maintaining this code, would be tiresome and prone to errors.

Listing 241 shows the contents of `readTimeControls.H`. The first instruction reads from *controlDict* the *adjustTimeStep* parameter. If there is no entry matching the name of the parameter (`"adjustTimeStep"`), then a default value is used. So, omitting the parameter *adjustTimeStep* in *controlDict* will result in a simulation with a fixed time step.

This is a very straight forward example of determining the behaviour of a solver using only the source code. In this case the names of the source file as well as variable and function names are rather self explaining. In other cases one has to dig deeply into the code to learn about what a certain command does.

[30] In case of a k-ϵ model, there are two transport equations to be solved. Other turbulence models require the solution of less or none transport equation.

[31] In case of a laminar simulation, no operation is carried out.

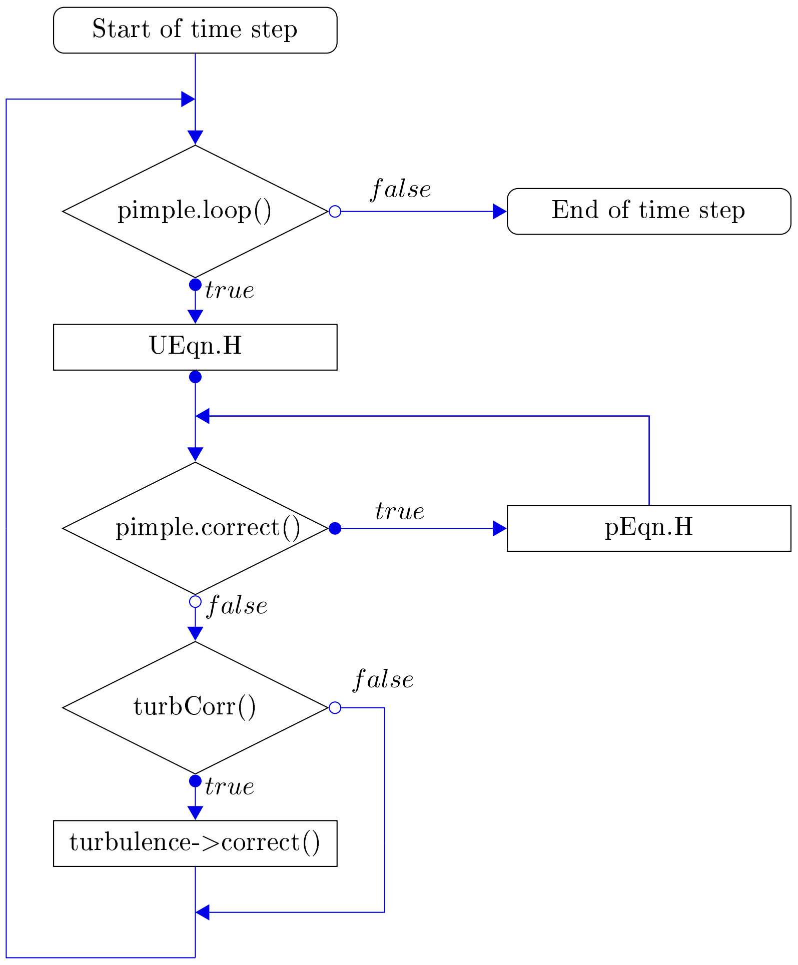


Figure 21: Flow chart of the PIMPLE algorithm

```
const bool adjustTimeStep =
  runTime.controlDict().lookupOrDefault("adjustTimeStep", false);
scalar maxCo =
  runTime.controlDict().lookupOrDefault<scalar>("maxCo", 1.0);
scalar maxDeltaT =
  runTime.controlDict().lookupOrDefault<scalar>("maxDeltaT", GREAT);
```

Listing 148: The content of `readTimeControls.H`

19.2.2 pimpleControl

Examining the files `pimpleControl.H` and `pimpleControl.C` will generate some knowledge of the inner life of *pimpleFoam*.

Solution controls

Listings 149 and 150 show parts of `pimpleControl.H` and `pimpleControl.C`. Listing 149 shows the declaration of protected[32] data in `pimpleControl.H`.

```
    // Protected data
        // Solution controls
            //- Maximum number of PIMPLE correctors
            label nCorrPIMPLE_;

            //- Maximum number of PISO correctors
            label nCorrPISO_;

            //- Current PISO corrector
            label corrPISO_;

            //- Flag to indicate whether to only solve turbulence on final iter
            bool turbOnFinalIterOnly_;

            //- Converged flag
            bool converged_;
```

Listing 149: Protected data in `pimpleControl.H`

```
void Foam::pimpleControl::read()
{
  solutionControl::read(false);

  // Read solution controls
  const dictionary& pimpleDict = dict();

  nCorrPIMPLE_ = pimpleDict.lookupOrDefault<label>("nOuterCorrectors", 1);

  nCorrPISO_ = pimpleDict.lookupOrDefault<label>("nCorrectors", 1);

  turbOnFinalIterOnly_ = pimpleDict.lookupOrDefault<Switch>("turbOnFinalIterOnly", true);
}
```

Listing 150: Read solution controls in `pimpleControl.C`

Reading the code we can see which keyword in the `PIMPLE` dictionary – it is a part of the `fvSolution` dictionary (see Section 7.4) – is connected to which variable in the code. Three of the protected variables of Listing 149 are assigned in Listing 150. One of them has the same name in both the code and the dictionary. The other two have different names.

The connection between keywords and the algorithm

The keyword `nOuterCorrectors` translates – with the help of Listing 150 to the variable `nCorrPIMPLE_`. This variable controls how often the PIMPLE loop is traversed. Listing 151 shows parts of the definition of the function `loop()` of the class `pimpleControl`. The return value of this function decides whether the PIMPLE loop is entered or not. In line 5 of Listing 151 an internal counter is incremented – the `++` operator of C++ adds 1 to the variable the operator is applied to. Afterwards, the internal counter is compared to the value of `nCorrPIMPLE_`. If this internal counter is then equal to the sum of `nCorrPIMPLE_ + 1`, then the function `loop()` returns `false`.

The internal counter is initialised to the value of 0. Listing 152 shows the constructor of the class `solutionControl`. The class `pimpleControl` is derived from `solutionControl`. So, every instance of `pimpleControl` has an internal counter `corr_` inherited from `solutionControl`. Line 9 of Listing 152 how the counter `corr_` is initialised

[32]Most programming languages provide *access specifiers* to specify the visibility of variables. The keyword `protected` means, that the variables can be accessed only inside the class `pimpleControl` and all classes inherited from `pimpleControl`.

to zero.

```
bool Foam::pimpleControl::loop()
{
  read();

  corr_++;

  /* code removed for the sake of brevity */

  if (corr_ == nCorrPIMPLE_ + 1)
  {
    if ((!residualControl_.empty()) && (nCorrPIMPLE_ != 1))
    {
      Info<< algorithmName_ << ": not converged within "
        << nCorrPIMPLE_ << " iterations" << endl;
    }

    corr_ = 0;
    mesh_.data::remove("finalIteration");
    return false;
  }

  /* code continues */
```

Listing 151: Some content of `pimpleControl.C`

```
Foam::solutionControl::solutionControl(fvMesh& mesh, const word& algorithmName)
:
mesh_(mesh),
residualControl_(),
algorithmName_(algorithmName),
nNonOrthCorr_(0),
momentumPredictor_(true),
transonic_(false),
corr_(0),
corrNonOrtho_(0)
{}
```

Listing 152: The constructor of the class `solutionControl` in `solutionControl.C`

The keyword `nCorrectors` translates – with the help of Listing 150 to the variable `nCorrPISO_`. This variable controls how often the PISO loop – or the corrector loop – is traversed. Listing 149 shows, that there are two variables related to the PISO loop, `nCorrPISO_` and `corrPISO_`. The first variable is the limit and the second is the counter.

`nCorrPISO_` is read from the `fvSolution` dictionary by the use of the `nCorrectors` keyword. This number tells the solver, how many times the corrector loop should be traversed. The corrector loop is a feature of the PISO algorithm. Hence, the maximum number of corrector loop iterations is called `nCorrPISO_`.

The variable `corrPISO_` is declared in the constructor of the class `pimpleControl`, see Listing 154. There the variable is initialised to zero.

Listing 153 shows the definition of the function `correct()` of the class `pimpleControl`. The return value of this function controls if the corrector loop is entered. In line 3 the counter `corrPISO_` is incremented every time this function is called. In line 10 the value of the counter is compared to the maximum number of corrector loop iterations.

```
1  inline bool Foam::pimpleControl::correct()
2  {
3    corrPISO_++;
4
5    if (debug)
6    {
7      Info<< algorithmName_ << " correct: corrPISO = " << corrPISO_ << endl;
8    }
9
10   if (corrPISO_ <= nCorrPISO_)
11   {
      return true;
    }
    else
    {
      corrPISO_ = 0;
      return false;
    }
}
```

Listing 153: The inline function `correct()` in `pimpleControlI.H`

PIMPLE or PISO algorithm

Listing 154 shows parts of the code of the constructor of the class `pimpleControl`. At first some data fields are set to initial values. Then the `read()` function is called, this function is shown in Listing 150. After reading the solution controls the variable `nCorrPIMPLE_` is tested. If this value is equal to one, then the solution algorithm equates the PISO algorithm. In this case an according message is printed to the Terminal.

```
Foam::pimpleControl::pimpleControl(fvMesh& mesh) :
    solutionControl(mesh, "PIMPLE"),
    nCorrPIMPLE_(0),
    nCorrPISO_(0),
    corrPISO_(0),
    turbOnFinalIterOnly_(true),
    converged_(false)
{
    read();

    if (nCorrPIMPLE_ > 1)
    {
        /* code removed for shortness of listing */
    }
    else
    {
        Info<< nl << algorithmName_ << ": Operating solver in PISO mode" << nl << endl;
    }
}
```

Listing 154: Constuctor of `pimpleControl` in `pimpleControl.C`

20 *twoPhaseEulerFoam*

This section is valid for OpenFOAM-2.0 til OpenFOAM-2.2.

20.1 General remarks

twoPhaseEulerFoam is a solver for two-phase problems. According to the CFD-Online Forum (`http://www.cfd-online.com/Forums/openfoam/`) this solver as well as *bubbleFoam* is based on the PhD thesis of Henrik Rusche [15]. In the course of an update of OpenFOAM-2.1.x in July 2012 the solution algorithm of the continuity equation was changed.

20.1.1 Turbulence

twoPhaseEulerFoam can only use the k-ϵ turbulence model. This model is so to say hardcoded and can only be turned on or off.

20.1.2 Kinetic theory

twoPhaseEulerFoam can make use of the kinetic theory for granular simulations, e.g. air flowing through a bed of small particles. This model can also be turned on or off.

In the following sections kinetic theory is ignored for the reason of keeping listings and explanations short.

20.2 Solver algorithm

twoPhaseEulerFoam is based on the PIMPLE algorithm. However, there are some modifications necessary for solving two-phase problems. Listing 155 shows the main part of this solver. The first two lines inside the main loop (`pimple.loop()`) differ from *pimpleFoam*. These lines deal with the two-phase continuity equation and the inter-phase momentum exchange coefficients.

Next, in line 6, comes the momentum predictor It contains the momentum equations for both phases and solves them subsequently, thus the filename `UEqns.H`.

After the predictor comes the corrector. The corrector is in fact a corrector loop. Inside this loop (`pimple.correct()`) the correction of pressure and velocity is computed. Inside the corrector loop (line 15) there is also a conditional second call of the continuity equation. The condition consists of two boolean statements. The first is a boolean variable, which is set in a dictionary by the user. The second is generated by the solution control.

After the corrector loop the total time derivatives of the velocities are calculated. Finally, the turbulent transport equations are solved. In this case it is the k-ϵ model that is called explicitly (line 23).

```
 1  // --- Pressure-velocity PIMPLE corrector loop
 2  while (pimple.loop())
 3  {
 4    #include "alphaEqn.H"
 5    #include "liftDragCoeffs.H"
 6    #include "UEqns.H"
 7
 8    // --- Pressure corrector loop
 9    while (pimple.correct())
10    {
11      #include "pEqn.H"
12
13      if (correctAlpha && !pimple.finalIter())
14      {
15        #include "alphaEqn.H"
16      }
17    }
18
19    #include "DDtU.H"
20
21    if (pimple.turbCorr())
22    {
23      #include "kEpsilon.H"
24    }
25  }
```

Listing 155: The main loop of *twoPhaseEulerFoam*

Figure 22 shows the flow chart of all operations that are performed during one time step.

20.2.1 Continuity

The continuity equation is implemented in the file `alphaEqn.H`.

Second call

In line 15 of Listing 155 the continuity equation is called again inside an `if`-statement. The condition depends on two boolean expressions.

The first, `correctAlpha`, is controlled by the `fvSolution` dictionary. Assigning a value to this keyword – the keyword has the same name as the boolean variable in the source code – is mandatory. The reading operation of this keyword from the dictionary can be found in the source file `readTwoPhaseEulerFoamControls.H` and is shown in Listing 156.

Three keywords are looked up from the `fvSolution` dictionary. All of them are related to the solving algorithm for the continuity equation. Those entries are read from the dictionary by invoking the function `lookup()`. See Section 34.3 for a detailed discussion about looking up keywords from dictionaries.

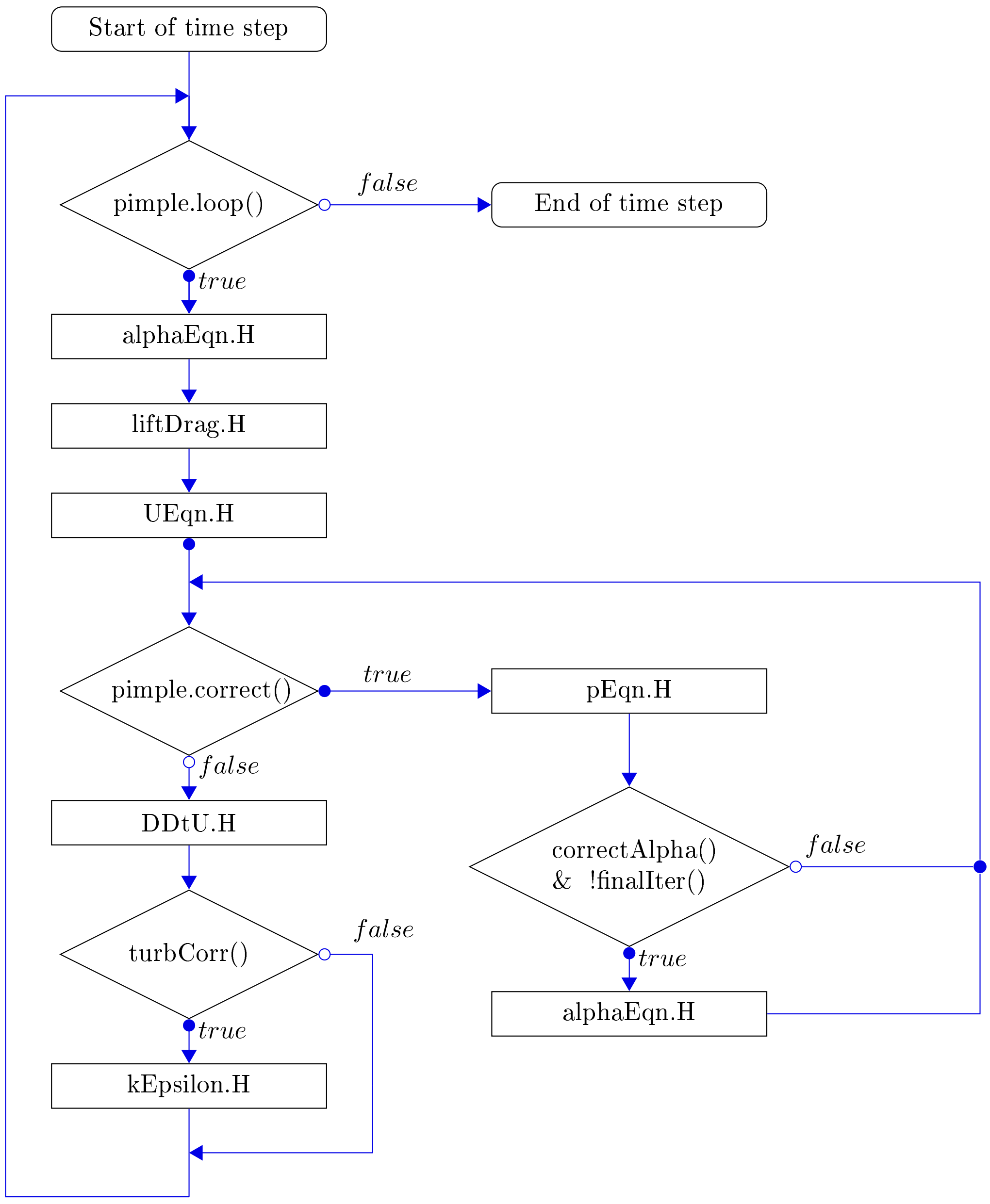


Figure 22: Flow chart of the main loop of *twoPhaseEulerFoam*

```
#include "readTimeControls.H"

int nAlphaCorr(readInt(pimple.dict().lookup("nAlphaCorr")));
int nAlphaSubCycles(readInt(pimple.dict().lookup("nAlphaSubCycles")));
Switch correctAlpha(pimple.dict().lookup("correctAlpha"));
```

Listing 156: The content of `readTwoPhaseEulerFoamControls.H`

The second boolean expression controlling the second call in line 15 of Listing 155 is controlled by the number of iterations of the PIMPLE loop. See Section 19.2 for a discussion about the PIMPLE algorithm.

The expression `pimple.finalIter()` is `true` when the last iteration of the PIMPLE algorithm is entered. Therefore, the expression `!pimple.finalIter()` is `true` if, and only if, the value of `nOuterCorrectors` or `nCorrPIMPLE_` is greater than one. Because only then, there is more than one PIMPLE iteration and only then, there is an iteration other than the final one.

If the PIMPLE loop is traversed only once, then `alphaEqn.H` is not entered a second time.

The file `alphaEqn.H`

The examination of the file `alphaEqn.H` results in the flow chart in Figure 23. The corrector loop is traversed a specified number of times. This number is set by the keyword `nAlphaCorr` of the `fvSolution` dictionary. The corrector loop is a simple `for` loop.

Inside the corrector loop is a sub-cycle loop. Inside this loop the continuity equation is solved. After the sub-cycle the volume fraction of the continuous phase is updated. The sub-cycle loop is also traversed a specified number of times. This number is set by the keyword `nAlphaSubCycles` of the `fvSolution` dictionary.

When the corrector loop is not entered anymore, the mixture density is updated.

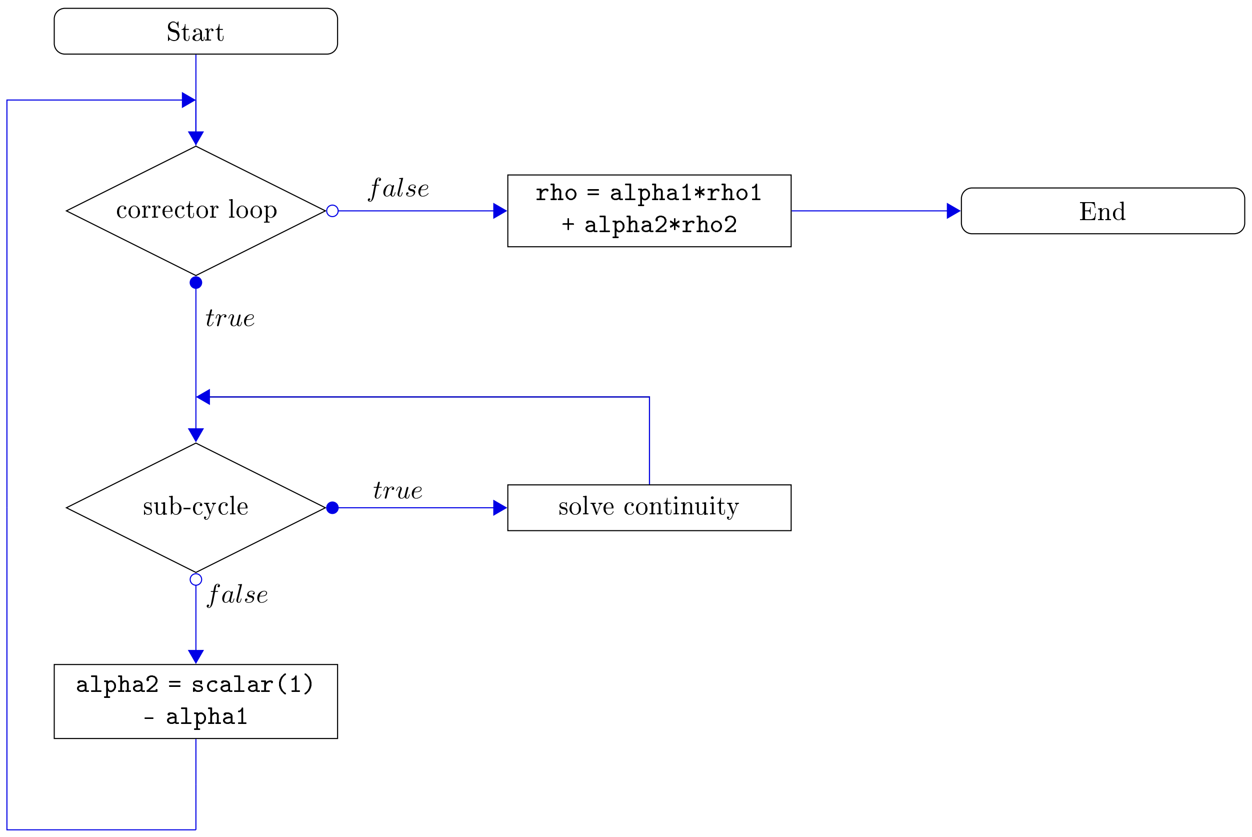


Figure 23: Flow chart of the operations in `alphaEqn.H`

20.3 Momentum exchange between the phases

20.3.1 Drag

The solver *twoPhaseEulerFoam* offers a number of drag models. In the sources of *twoPhaseEulerFoam* there are this models

- Ergun
- Gibilaro
- GidaspowErgunWenYu
- GidaspowSchillerNaumann
- SchillerNaumann
- SyamlalOBrien
- WenYu

The equations behind this models can be found in [8] or [18].

Drag is considered in the governing equations by the use of the so-called drag-function K. This drag-function is either computed directly, or it is computed by the use of the drag coefficient C_d. The drag force is the product of the drag-function and the relative velocity between the phases $\mathbf{U}_r$ [8].

Schiller-Naumann drag

We use the Schiller-Naumann drag model as an expample to demonstrate how OpenFOAM calculates the drag force. This drag model utilizes a drag coefficient that is a function of the Reynolds number.

$$C_d = \begin{cases} \frac{24}{Re}\left(1 + 0.15 Re^{0.687}\right) & \text{if } Re \leq 1000 \\ 0.44 & \text{if } Re > 1000 \end{cases} \tag{37}$$

$$K = \frac{3}{4} C_d \rho_B \frac{U_r}{d_A} \tag{38}$$

The drag coefficient is dimensionless, whereas the product of the drag-function K and the relative velocity has the dimension of a force density.

$$[K] = [C_d] \cdot [\rho_B] \cdot [\frac{U_r}{d_A}] = 1 \cdot \frac{\text{kg}}{\text{m}^3} \cdot \frac{\text{m}}{\text{s}} \frac{1}{\text{m}} = \frac{\text{kg}}{\text{m}^3\text{s}}$$
$$[K \cdot U_r] = \frac{\text{kg}}{\text{m}^3\text{s}} \cdot \frac{\text{m}}{\text{s}} = \frac{\text{kgm}}{\text{s}^2} \cdot \frac{1}{\text{m}^3} = \frac{\text{N}}{\text{m}^3}$$

Listing 157 shows, how the drag-function is computed by the Schiller-Naumann drag model.

```
Foam::tmp<Foam::volScalarField> Foam::SchillerNaumann::K
(
    const volScalarField& Ur
) const
{
    volScalarField Re(max(Ur*phasea_.d()/phaseb_.nu(), scalar(1.0e-3)));

    volScalarField Cds
    (
        neg(Re - 1000)*(24.0*(1.0 + 0.15*pow(Re, 0.687))/Re)
      + pos(Re - 1000)*0.44
    );

    return 0.75*Cds*phaseb_.rho()*Ur/phasea_.d();
}
```

Listing 157: Calculation of the drag-function in the file SchillerNaumann.H

The drag force contributes to the momentum balance. Probably for numerical reasons, one part of the drag is considered in the momentum equation and the other part is considered in the pressure equation.

20.3.2 Lift

The lift model of *twoPhaseEulerFoam* is described in [15]. The lift model computes the lift force on a rigid sphere in shear flow. The force density is calculated from the relative velocity between the phases and the vorticity of the mixture.

$$\frac{F_L}{V_B} = C_L \rho_c |\mathbf{U}_r \times (\nabla \times \mathbf{U}_c)| \tag{39}$$

mit

$$\mathbf{U}_r = \mathbf{U}_A - \mathbf{U}_B$$
$$\mathbf{U}_c = \alpha \mathbf{U}_A + \underbrace{(1-\alpha)}_{=\beta} \mathbf{U}_B$$
$$\rho_c = \alpha \rho_A + \beta \rho_B$$

The lift force is computed in the file `liftDragCoeffs.H`. The vector field `liftCoeff` contains the lift force density.

```
volVectorField liftCoeff(Cl*(beta*rhob + alpha*rhoa)*(Ur ^ fvc::curl(U)));
```

Listing 158: Berechnung Auftriebskraft; *liftDragCoeffs.H*

The dimensions of the field `liftCoeff` is the dimension of a force density.

$$[liftCoeff] = [C_L] \cdot [\rho_c] \cdot [\mathbf{U}_r \times (\nabla \times \mathbf{U}_c)] = 1 \cdot \frac{\mathrm{kg}}{\mathrm{m}^3} \cdot \frac{\mathrm{m}}{\mathrm{s}} \frac{1}{\mathrm{m}} \frac{\mathrm{m}}{\mathrm{s}} = \frac{\mathrm{kgm}}{\mathrm{s}^2} \cdot \frac{1}{\mathrm{m}^3} = \frac{\mathrm{N}}{\mathrm{m}^3}$$

20.3.3 Virtual mass

The virtual mass – an accelerating bubble needs not only to accelerate its own mass, it also needs to accelerate some of the displaced fluid – is considered in the momentum equation.

$$M_{A,VM} = \beta \frac{\rho_B}{\rho_A} C_{VM} \left(\frac{\mathrm{D}_B \mathbf{U}_B}{\mathrm{D}t} - \frac{\mathrm{D}_A \mathbf{U}_A}{\mathrm{D}t} \right) \tag{40}$$

In the source code, the momentum exchange term due to virtual mass is split into two parts. One part is included in the *rhs* of the momentum equation, the other is considered in the *lhs*. This seperation is probably for numerical reasons.

```
UaEqn =
(
  (scalar(1) + Cvm*rhob*beta/rhoa)*
  (
    fvm::ddt(Ua)
    + fvm::div(phia, Ua, "div(phia,Ua)")
    - fvm::Sp(fvc::div(phia), Ua)
  )
  + /* other terms */
  ==
  /* other terms */
  - beta/rhoa*(liftCoeff - Cvm*rhob*DDtUb)
);
```

Listing 159: Terms including virtual mass in the file `UEqns.H`

20.4 Kinetic Theory

For the simulation of dense gas-solid particulate flows the particulate phase can be modelled using the kinetic theory model.

21 *twoPhaseEulerFoam-2.3*

This section is valid for OpenFOAM-2.3.

With the release of OpenFOAM-2.3 the two-phase Eulerian solver *twoPhaseEulerFoam* has seen some major changes. See the release notes for further details: `http://www.openfoam.org/version2.3.0/multiphase.php`.

21.1 Naming scheme

The overhaul of *twoPhaseEulerFoam* in version 2.3 aims for reuseability and generality of the solver code itself as well as of the case data. A general distinction of data concerning a single phase and data concerning the whole simulation case can be made.

Case data is named as usual (e.g. `fvSchemes`, `controlDict`, `g`, etc.). Data related to a specific phase is now stored in files with a filename that consists of two parts. The naming scheme follows the well known `FILENAME.EXTENSION` naming scheme. In this case `FILENAME` denotes the type of information and `EXTENSION` denotes the phase itself. This naming scheme is much more general than other naming schemes that are/were used in OpenFOAM (cf. `U1, U2` vs. `Uwater, Uair` vs. `U.air, U.water`).

Listing 160 shows the contents of the *0* and *constant* folders of the bubble column tutorial case. There we see the `FILENAME.EXTENSION` naming scheme applied. As each phase has a velocity and a temperature, we see two files for velocity and temperature. The volume fraction is an exception, as there are only two phase considered, the volume fraction of water is easily calculated, i.e. `alpha.water = 1.0 - alpha.air`. As the pressure is share by all phases, the pressure file has no file-extension. In the *constant* folder there is also data that applies to one phase and data that applies to the simulation case. The files `g` and `phaseProperties` have no extensions because they contain no information specific to one phase. The thermophysical properties of the phases air and water are stored in the appropiate files.

The naming scheme that was introduced with *twoPhaseEulerFoam-2.3* is fit to create a material data library. The was the phases or the phase data is organized within the solver is now independent of the way the phase data is organized within the case.

```
user@host:~/OpenFOAM/OpenFOAM-2.3.x/tutorials/multiphase/twoPhaseEulerFoam/RAS/bubbleColumn$
    ls 0 -1
alpha.air
alpha.air.org
epsilon.air
epsilon.water
k.air
k.water
nut.air
nut.water
p
T.air
Theta
T.water
U.air
U.water
user@host:~/OpenFOAM/OpenFOAM-2.3.x/tutorials/multiphase/twoPhaseEulerFoam/RAS/bubbleColumn$
    ls constant -1
g
phaseProperties
polyMesh
thermophysicalProperties.air
thermophysicalProperties.water
turbulenceProperties.air
turbulenceProperties.water
```

Listing 160: Content of the *0* and *constant* folders of the bubble column tutorial case of *twoPhaseEulerFoam* in OpenFOAM-2.3.x

21.2 Solver capabilities

Not only the naming scheme is more general in version 2.3, also the solver itself is more generalized.

Compressibility all phases are treated as compressible. In the file `thermophysicalProperties` the behaviour of a phase can be specified.

Energy equation *twoPhaseEulerFoam* solves an energy equation for all phases. This can not be turned off.

Phase interaction has been extended. A great number of models specific for gas-liquid systems have been included.

Turbulence Turbulence is treated in a more general way. A number of turbulence models can be used in contrast to earlier versions of *twoPhaseEulerFoam* that had *kEpsilon* hard-coded.

21.3 Turbulence models

twoPhaseEulerFoam-2.3 uses a whole new class of turbulence models. As the governing equations of twoPhaseEulerFoam – namely the momentum equation – aren't phase intensive anymore, also the governing equations of the turbulence model are formulated in their general multi-phase form.

This limits the choice of turbulence models to a small number of multi-phase turbulence models. Listings 161 and 162 show the list of available turbulence models at the time of writing (May 2014).

```
Valid RASModel types:

6
(
LaheyKEpsilon
continuousGasKEpsilon
kEpsilon
kineticTheory
mixtureKEpsilon
phasePressure
)
```

Listing 161: Valid RAS turbulence models of *twoPhaseEulerFoam*.

```
Valid LESModel types:

5
(
NicenoKEqn
Smagorinsky
SmagorinskyZhang
continuousGasKEqn
kEqn
)
```

Listing 162: Valid LES turbulence models of *twoPhaseEulerFoam*.

21.3.1 `kEpsilon`

Listing 163 shows the governing equations of the compressible multi-phase formulation of the $k-\epsilon$ model. The governing equations are largely equivalent to the compressible formulation of the single-phase $k-\epsilon$ model. The formulation deviates from the compressible single-phase formulation in two aspects. First, the convective term is corrected with the continuity error, see Lines 5 and 18. Furthermore, there is an additional source term on the RHS, see Lines 11 and 24.

```
1      tmp<fvScalarMatrix> epsEqn
2      (
3          fvm::ddt(alpha, rho, epsilon_)
4        + fvm::div(alphaRhoPhi, epsilon_)
5        - fvm::Sp(fvc::ddt(alpha, rho) + fvc::div(alphaRhoPhi), epsilon_)
6        - fvm::laplacian(alpha*rho*DepsilonEff(), epsilon_)
7       ==
8          C1_*alpha*rho*G*epsilon_/k_
9        - fvm::SuSp(((2.0/3.0)*C1_ + C3_)*alpha*rho*divU, epsilon_)
10       - fvm::Sp(C2_*alpha*rho*epsilon_/k_, epsilon_)
```

```
        + epsilonSource()
    );

    tmp<fvScalarMatrix> kEqn
    (
        fvm::ddt(alpha, rho, k_)
      + fvm::div(alphaRhoPhi, k_)
      - fvm::Sp(fvc::ddt(alpha, rho) + fvc::div(alphaRhoPhi), k_)
      - fvm::laplacian(alpha*rho*DkEff(), k_)
     ==
        alpha*rho*G
      - fvm::SuSp((2.0/3.0)*alpha*rho*divU, k_)
      - fvm::Sp(alpha*rho*epsilon_/k_, k_)
      + kSource()
    );
```

Listing 163: Governing equations of the `kEpsilon` turbulence model.

21.3.2 `mixtureKEpsilon`

Usage

The $k - \epsilon$ model is computed for the mixture, i.e. the transport equations are solved for using the mixture properties. Thus, the solution variables are named `km` and `epsilonm`, see Listing 164.

```
DILUPBiCG:  Solving for epsilonm, Initial residual = 0.0114325, Final residual = 2.79117e-09,
    No Iterations 2
DILUPBiCG:  Solving for km, Initial residual = 0.0078252, Final residual = 6.13173e-09, No
    Iterations 2
```

Listing 164: Solver output of *twoPhaseEulerFoam* using the `mixtureKEpsilon` turbulence model.

In order to use the mixture $k-\epsilon$ model, it needs to be specified in both `turbulenceProperties` files. Listing 165 shows the resulting error message when `mixtureKEpsilon` is specified for only one of the phases. As the turbulence model for the mixture applies to both phases, it needs to be specified for both phases.

```
--> FOAM FATAL ERROR:

    lookup of turbulenceProperties.water from objectRegistry region0 successful
    but it is not a mixtureKEpsilon, it is a LaheyKEpsilon

    From function objectRegistry::lookupObject<Type>(const word&) const
    in file /home/user/OpenFOAM/OpenFOAM-2.3.x/src/OpenFOAM/lnInclude/objectRegistryTemplates.
    C at line 181.

FOAM aborting
```

Listing 165: Solver output of *twoPhaseEulerFoam* when the `mixtureKEpsilon` turbulence model is specified for only one of the two phases.

Theory

The governing equations of the mixture $k-\epsilon$ model can be found in the sources at `\$FOAM\_SRC/TurbulenceModels/phaseCompressible/RAS/mixtureKEpsilon` and in [6]. The biggest difference between the equations stated in [6] and the code of `mixtureKEpsilon` can be found in the Lines 5 and 18 of Listing 166. There, the continuity equation of the mixture appears on the of the governing equations. This minor difference between the formulation of the equation can be resolved in two steps. First, we take a look on the first two terms of the governing equations in [6] (local derivative and convective term), see Eqns. (41) to (44).

$$\frac{\partial \rho_m \epsilon_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}_m \epsilon_m) + \ldots \tag{41}$$

$$\rho_m \frac{\partial \epsilon_m}{\partial t} + \epsilon_m \frac{\partial \rho_m}{\partial t} + \epsilon_m \nabla \cdot (\rho_m \mathbf{u}_m) + \rho_m \mathbf{u}_m \cdot \nabla \epsilon_m + \ldots \tag{42}$$

$$\rho_m \frac{\partial \epsilon_m}{\partial t} + \underbrace{\epsilon_m \left(\frac{\partial \rho_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}_m) \right)}_{=0} + \rho_m \mathbf{u}_m \cdot \nabla \epsilon_m + \ldots \tag{43}$$

$$\rho_m \frac{\partial \epsilon_m}{\partial t} + \rho_m \mathbf{u}_m \cdot \nabla \epsilon_m + \ldots \tag{44}$$

In order to derive equations equivalent to the code implemented in OpenFOAM, we begin with Eq. (44) and use the product rule of differentiation, cf. Eqns. (41) and (42).

$$\rho_m \frac{\partial \epsilon_m}{\partial t} + \rho_m \mathbf{u}_m \cdot \nabla \epsilon_m + \ldots \tag{44}$$

$$\frac{\partial \rho_m \epsilon_m}{\partial t} - \epsilon_m \frac{\partial \rho_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}_m \epsilon_m) - \epsilon_m \nabla \cdot (\rho_m \mathbf{u}_m) + \ldots \tag{45}$$

$$\frac{\partial \rho_m \epsilon_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}_m \epsilon_m) - \epsilon_m \left(\frac{\partial \rho_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}_m) \right) + \ldots \tag{46}$$

Eq (46) is now equivalent to the first terms of the ϵ equation of Listing 166. The exact reason why this formulation was chosen is unknown to the author, a probable reason might be a better numerical behaviour.

```
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(rhom, epsilonm)
      + fvm::div(phim, epsilonm)
      - fvm::Sp(fvc::ddt(rhom) + fvc::div(phim), epsilonm)
      - fvm::laplacian(DepsilonEff(rhom*nutm), epsilonm)
     ==
        C1_*rhom*Gm*epsilonm/km
      - fvm::SuSp(((2.0/3.0)*C1_)*rhom*divUm, epsilonm)
      - fvm::Sp(C2_*rhom*epsilonm/km, epsilonm)
      + epsilonSource()
    );

    tmp<fvScalarMatrix> kmEqn
    (
        fvm::ddt(rhom, km)
      + fvm::div(phim, km)
      - fvm::Sp(fvc::ddt(rhom) + fvc::div(phim), km)
      - fvm::laplacian(DkEff(rhom*nutm), km)
     ==
        rhom*Gm
      - fvm::SuSp((2.0/3.0)*rhom*divUm, km)
      - fvm::Sp(rhom*epsilonm/km, km)
      + kSource()
    );
```

Listing 166: Governing equations of the `mixtureKEpsilon` turbulence model.

21.3.3 Pitfall: phase inversion

Phase inversion is the situation when the volume fraction of the continuous phase vanishes in some regions. As almost all terms of the governing equations are weighted with the volume fraction `alpha`, a vanishing volume fraction can lead to serious numerical problems.

The following example demonstrates the problems which may be faced when dealing with phase inversion. An air-water bubble column is modelled including some of the air above the water surface. Figure 24 shows the air volume fraction within the bubble column.

When `mixtureKEpsilon` is selected as turbulence model, the volume fraction is not included in the governing equation, so phase inversion poses no big problem, see Listing 166 or Eq. (46).

When `kEpsilon` is selected for the liquid phase, the volume fraction in the governing equations is the volume fraction of the liquid phase. This volume fraction vanishes above the water surface. Thus, in parts of the domain the solution of the governing equations faces numerical problems. The governing equations can still be solved in this case, but preconditioning the resulting matrix equation fails. Preconditioning is a step that is intended to improve the iterative solution of the resulting matrix equation. In the case of the `kEpsilon` turbulence model for the liquid phase, the only way to avoid crashing the simulation is to use a -solver with no preconditioning. The -solver and the smooth solver fail completely.

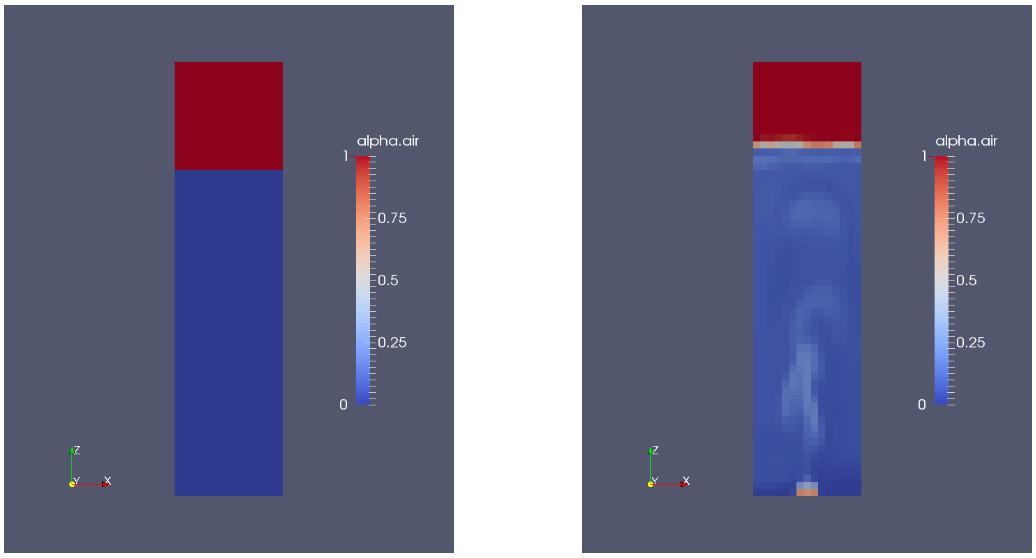


Figure 24: Air volume fraction of the bubble column. Initial field (left) and solution at $t = 10\,\mathrm{s}$ (right).

22 *multiphaseEulerFoam*

multiphaseEulerFoam is an Eulerian solver for n phases. This solver differs in some points from the solver *twoPhaseEulerFoam*.

22.1 Fields

The naming scheme of the fields differs from other multiphase solvers. *multiphaseEulerFoam* directly uses names (e.g. Uair, Uwater, Uoil, etc.).

22.1.1 alphas

A specialty of multiphaseEulerFoam is the field `alphas`. This field does not represent the volume fraction of a certain phase and is therefore not bounded by 0 and 1. This field is used to represent all phases in a single scalar field. `alphas` is computed by summing up the products of phase index and phase fraction.

$$\texttt{alphas} = \sum_{i=0}^{n-1} i * \alpha_i \tag{47}$$

Because `alphas` is computed quantity, the file `alphas` can be missing in the *0*-directory.

22.2 Momentum exchange

The parameters for the momentum exchange, e.g. the drag model, need to be specified pair-wise.

22.2.1 *drag*

```
drag
(
  (air water)
  {
    type blended;

    air
    {
      type SchillerNaumann;
      residualPhaseFraction 0;
      residualSlip 0;
    }

    water
    {
      type SchillerNaumann;
      residualPhaseFraction 0;
      residualSlip 0;
    }

    residualPhaseFraction 1e-2;
    residualSlip 1e-2;
  }

  /* further definitions */
```

Listing 167: Pair-wise definition of the drag model in the file `transportProperties`

22.2.2 *virtual mass*

The coefficients for considering virtual mass must also be specified pair-wise. Listing 168 shows how the coefficients for virtual mass are specified in the *damBreak* tutorial.

```
virtualMass
(
  (air water)     0.5
  (air oil)       0.5
  (air mercury)   0.5
  (water oil)     0.5
  (water mercury) 0.5
  (oil mercury)   0.5
);
```

Listing 168: Pair-wise definition of Coefficients for virtual mass in the file `transportProperties`

22.2.3 *lift force*

Currently (OpenFOAM 2.1.1) there is no lift model in *multiphaseEulerFoam*.

23 Multiphase modelling

23.1 Phase model class

One of the strenghts of object oriented programming is that the class structure in the source code can reflect the properties and relations of real-world things.

The phase model class in the various two- and multi-phase solvers of OpenFOAM is one example of how techniques of object oriented programming can be applied. In terms of a multi-phase problem in fluid dynamics we distinguish different phases.

We now violate the unwritten law of to not cite Wikipedia.

> Phase (matter), a physically distinctive form of a substance, such as the solid, liquid, and gaseous states of ordinary matter—also referred to as a "macroscopic state"
> `http://en.wikipedia.org/wiki/Phase`

In fluid dynamics phase is a commonly used term. When we intend our code to represent the reality we want to describe as closely as possible we need to introduce the concept of the phase into our source code. From a programming point of view properties of a phase – such as viscosity, velocity, etc. – are easy to implement. The viscosity of a phase is simply a field of values, velocity is another field of values.

Object orientation allows us to translate the idea of the phase into programming language. The basic idea is that a phase has a viscosity, it also has a velocity. We now create a class named `phaseModel` and this class needs to have a viscosity, a velocity and everthing else a phase needs to fit our needs.

The phase model classes follow the code of best practice in object oriented programming to hide internal data from the outer world and to provide access via the classes methods (data encapsulation, see `http://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm`).

No phases, please

In the single-phase solvers of OpenFOAM – such as *simpleFoam* – the concept of a phase is not used. As there is only one temperature and velocity to deal with, the concept of phases is not needed. In the single-phase solvers the phase-properties (viscosity, velocity, density, etc.) are linked according to the physical relations that are taken into account, but the concept of a phase is missing.

23.2 A comparison of the phase models in OpenFOAM-2.2

In this section we want to compare the implementation of the phase model class of the two solvers *twoPhaseEulerFoam* and *multiphaseEulerFoam*.

23.2.1 *twoPhaseEulerFoam*

The phase model class in *twoPhaseEulerFoam*-2.2.x collects the properties of a phase and offers an interface for accessing these properties. Listing 169 shows the essence of the header file of the phase model class. The listing is syntactically correct, however all pre-processor instruction (e.g. the `#include` statements) have been removed. Furthermore, most of the comments have been removed and the formatting has been adapted to reduce the line number. The purpose of Listing 169 is to present the data members and methods of the class by actual source code.

```
namespace Foam
{

class phaseModel
{
    // Private data
        dictionary dict_;
        word name_;
        dimensionedScalar d_;
        dimensionedScalar nu_;
        dimensionedScalar rho_;
        volVectorField U_;
        autoPtr<surfaceScalarField> phiPtr_;

public:
    // Member Functions
        const word& name() const { return name_; }

        const dimensionedScalar& d() const { return d_; }

        const dimensionedScalar& nu() const { return nu_; }

        const dimensionedScalar& rho() const { return rho_; }

        const volVectorField& U() const { return U_; }

        volVectorField& U() { return U_; }

        const surfaceScalarField& phi() const { return phiPtr_(); }

        surfaceScalarField& phi() { return phiPtr_(); }
};

} // End namespace Foam
```

Listing 169: A boiled-down version of the file `phaseModel.H`

The phase model class of *twoPhaseEulerFoam*-2.2.x contains all phase properties needed for an incompressible two-phase solver that makes use of an important consequence of being limited to two phase problems. By taking a look on the members of the class we see that there is no volume fraction field. In two phase problems one volume fraction field (`alpha1`) suffices as the volume fraction field of the other phase is instantly known (`alpha2 = 1 - alpha1`). Thus, the volume fraction can be treated seperately from other phase information.

Another missing item is the pressure. Most two- or multi-phase Eulerian solvers assume/use a common pressure for all phases. Thus, the pressure is independent of the phases and can be treated seperately.

23.2.2 *multiphaseEulerFoam*

One difference between the phase model class used in *twoPhaseEulerFoam* and the one used in *multiphaseEulerFoam* follows directly from the simplification made in the two-phase case. When dealing with an arbitrary number of phases, each phase must keep track of its own volume fraction. Thus, the volume fraction must be included into the phase model.

The straight-forward way would be to add another reference to the data members. As the volume fraction field is a scalar field, this reference would be a reference to a `volScalarField`. In *multiphaseEulerFoam* a more subtle approach was chosen. This also presents the application of another object-oriented programming technique.

The phase model class of *multiphaseEulerFoam* is derived from the class `volScalarField`. Thus, the phase model class is among other things its own the volume fraction field.

Listing 170 shows a stripped version of the header file of *multiphaseEulerFoam's* phase model class. Again, large parts of the file have been removed leaving only the data members and the methods of the class.

```
namespace Foam
{

class phaseModel
:
    public volScalarField
{
    // Private data
        word name_;
        dictionary phaseDict_;
        dimensionedScalar nu_;
        dimensionedScalar kappa_;
        dimensionedScalar Cp_;
        dimensionedScalar rho_;
        volVectorField U_;
        volVectorField DDtU_;
        surfaceScalarField phiAlpha_;
        autoPtr<surfaceScalarField> phiPtr_;
        autoPtr<diameterModel> dPtr_;

public:

    // Member Functions
        const word& name() const { return name_; }

        const word& keyword() const { return name(); }

        tmp<volScalarField> d() const;

        const dimensionedScalar& nu() const { return nu_; }

        const dimensionedScalar& kappa() const { return kappa_; }

        const dimensionedScalar& Cp() const { return Cp_; }

        const dimensionedScalar& rho() const { return rho_; }

        const volVectorField& U() const { return U_; }
        volVectorField& U() { return U_; }

        const volVectorField& DDtU() const { return DDtU_; }

        volVectorField& DDtU() { return DDtU_; }

        const surfaceScalarField& phi() const { return phiPtr_(); }

        surfaceScalarField& phi() { return phiPtr_(); }

        const surfaceScalarField& phiAlpha() const { return phiAlpha_; }

        surfaceScalarField& phiAlpha() { return phiAlpha_; }

        void correct();

        bool read(const dictionary& phaseDict);
};

} // End namespace Foam
```

Listing 170: A boiled-down version of the file `phaseModel.H`

The statements following the class keyword and the class name indicates the derivation of a class. The class name (`phaseModel`) and the name of the class we are deriving from (`volScalarField`) are separated by a colon (`:`). The name of the base class (`volScalarField`) is preceded by a visibility specifier (`public`). Here, we see a prototype of a class definition. The class we define (`phaseModel`) is derived from a base class (`volScalarField`).

```
class phaseModel : public volScalarField
{
  /* some c++ code */
}
```

This example highlights, that the class `phaseModel` is derived from the class `volScalarField`. This information alone does no proof that the phase model is its own volume fraction field. However, a glance on the constructor in the implementation file brings clarity.

In Listing 171 we see, that the first instruction in the initialisation list of the constructor reads the volume fraction field of the respective phase. This proofes that the phase model is in fact its own volume fraction field. For an explanation why we come to this conclusion we refer to any C++ textbook or online resource that covers the concept of inheritance, see e.g. `http://www.learncpp.com/cpp-tutorial/114-constructors-and-initialization-of-derived-classes/` or [17].

```
// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
Foam::phaseModel::phaseModel
(
    const word& name,
    const dictionary& phaseDict,
    const fvMesh& mesh
)
:
    volScalarField
    (
        IOobject
        (
            "alpha" + name,
            mesh.time().timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    ),
    name_(name),
    // code continues
```

Listing 171: The first few lines of the constructor of the phase model.

Besides being its own volume fraction field the phase model class of *multiphaseEulerFoam* was extended by several fields bearing information for the simulation of thermodynamics.

We can also observe the rudiment of giving the phase model a more active role. The phase model class of *twoPhaseEulerFoam* is simply an information carrier. The phase model of *multiphaseEulerFoam* features a method named `correct()`. The `correct()` method is used in many models for actions performed at every time step. However, in *multiphaseEulerFoam*-2.2.x this method is empty.

With OpenFOAM-2.1.0 the class `diameterModel` was introduced into *multiphaseEulerFoam* and *compressibleTwoPhaseEulerFoam*. The phase model class of multiphaseEulerFoam uses a diameter model class for keeping track of the dispersed phase's diameter. The diameter model offers the choice of computing the diameter of the dispersed phase elements from thermodynamic quantities besides using a constant diameter. Thus, the data member `dimensionedScalar d_` is replaced by a reference to a diameter model (`autoPtr<diameterModel> dPtr_`).

23.3 A comparison of the phase models in OpenFOAM-2.3

In this section we want to compare the implementation of the phase model class of the two solvers *twoPhaseEulerFoam* and *multiphaseEulerFoam*.

23.3.1 A comment on *multiphaseEulerFoam*

The phase model class used for *multiphaseEulerFoam* in OpenFOAM-2.2.x and OpenFOAM-2.3.x differs very little with respect to the class's methods and members. Listing 172 shows that the header files of the `phaseModel` class of *multiphaseEulerFoam* differs only in the copyright notice. The implementation file shows slightly greater differences[33]. However, the behaviour of this class can be considered nearly identical in OpenFOAM-2.2.x and OpenFOAM-2.3.x.

```
user@host:~/OpenFOAM$ diff
  OpenFOAM-2.2.x/applications/solvers/multiphase/multiphaseEulerFoam/phaseModel/phaseModel/
    phaseModel.H
  OpenFOAM-2.3.x/applications/solvers/multiphase/multiphaseEulerFoam/multiphaseSystem/
    phaseModel/phaseModel.H
5c5
<     \\  /    A nd           | Copyright (C) 2011 OpenFOAM Foundation
---
>     \\  /    A nd           | Copyright (C) 2011-2013 OpenFOAM Foundation
```

Listing 172: The output of *diff* for the file `phaseModel.H` of the solver *multiphaseEulerFoam* of the versions OpenFOAM-2.2.x and OpenFOAM-2.3.x as of May 2014[35].

23.3.2 *twoPhaseEulerFoam*

The two-phase model of *twoPhaseEulerFoam*-2.3.x makes heavy use of abstractions. The phase model class is used in conjunction with a class for the two-phase system.

```
namespace Foam
{

class phaseModel
:
    public volScalarField,
    public transportModel
{
    // Private data
        const twoPhaseSystem& fluid_;
        word name_;
        dictionary phaseDict_;
        scalar alphaMax_;
        autoPtr<rhoThermo> thermo_;
        volVectorField U_;
        surfaceScalarField alphaPhi_;
```

[33]The `diff` of the implementation file would be too long to be shown at this place. For general information on `diff` see Section 42.6.

[35]OpenFOAM Builds compared: 2.2.x-61b850bc107b and 2.3.x-0eb39ebe0f07.

```
        surfaceScalarField alphaRhoPhi_;
        autoPtr<surfaceScalarField> phiPtr_;
        autoPtr<diameterModel> dPtr_;
        autoPtr<PhaseCompressibleTurbulenceModel<phaseModel> > turbulence_;

public:

    // Member Functions
        const word& name() const { return name_; }

        const twoPhaseSystem& fluid() const { return fluid_; }

        const phaseModel& otherPhase() const;

        scalar alphaMax() const { return alphaMax_; }

        tmp<volScalarField> d() const;

        const PhaseCompressibleTurbulenceModel<phaseModel>&
            turbulence() const;

        PhaseCompressibleTurbulenceModel<phaseModel>&
            turbulence();

        const rhoThermo& thermo() const { return thermo_(); }

        rhoThermo& thermo() { return thermo_(); }

        tmp<volScalarField> nu() const { return thermo_->nu(); }

        tmp<scalarField> nu(const label patchi) const { return thermo_->nu(patchi); }

        tmp<volScalarField> mu() const { return thermo_->mu(); }

        tmp<scalarField> mu(const label patchi) const { return thermo_->mu(patchi); }

        tmp<volScalarField> kappa() const { return thermo_->kappa(); }

        tmp<volScalarField> Cp() const { return thermo_->Cp(); }

        const volScalarField& rho() const { return thermo_->rho(); }

        const volVectorField& U() const { return U_; }

        volVectorField& U() { return U_; }

        const surfaceScalarField& phi() const { return phiPtr_(); }

        surfaceScalarField& phi() { return phiPtr_(); }

        const surfaceScalarField& alphaPhi() const { return alphaPhi_; }

        surfaceScalarField& alphaPhi() { return alphaPhi_; }

        const surfaceScalarField& alphaRhoPhi() const { return alphaRhoPhi_; }

        surfaceScalarField& alphaRhoPhi() { return alphaRhoPhi_; }

        void correct();

        virtual bool read(const dictionary& phaseProperties);

        virtual bool read() { return true; }
};

} // End namespace Foam
```

Listing 173: A boiled-down version of the file `phaseModel.H`

The data members of the phase model class in *twoPhaseEulerFoam*-2.3.x contain a reference to the two-phase model class. This makes the phase model class aware of the other phase. The data members also contain a reference to a turbulence model and a thermophysical model. This is up to now the greatest generalisation we could observe in the multi-phase solvers of OpenFOAM.

Part VI
Postprocessing

There are two principal possibilities for post processing in OpenFOAM. First, there are tools that are executed after a simulation has finished. This tools work on the written data of the solution. *sample* and *paraView* are two examples for such tools.

Besides that, there is run-time post processing. Run-time post processing performs certain operations on the solution data as it is generated. Consequently, run-time post processing allows for a much finer time resolution. The functions objects – e.g. for calculating forces or force coefficients – are an example for run-time post processing. The big disadvantage of this method is, that the user has to know the intended post processing steps before starting a simulation. See `http://www.openfoam.com/features/runtime-postprocessing.php` for more information about run-time post processing.

24 *functions*

The functions are little programs that are part of OpenFOAM. A function object serves for one specific purpose, e.g. compute the time average of a field quantity. The function objects enable run-time post processing. At this point some function objects are explained.

fieldAverage compute the time average of field quantities

forces compute the forces on a body

forceCoeffs compute force coefficients, e.g. for drag, lift and momentum

sampledSet save the field values of a certain region, e.g. along a line

probes save field values at certain points

streamLine compute streamlines

24.1 Definition

function objects are defined in the file `controlDict`. There, a function dictionary is created which contains all necessary informations. Listing 174 shows the basic structure of such a definition.

Every function has a name. This name is stated at the place of the `NAME` placeholder in Listing 174. This name is also the name of the folder OpenFOAM creates in the case directory. There, all data generated by the function object is stored.

Each function object also has a type. This type needs to be specified at the place of the `TYPE` placeholder. The type needs to be from the list of the available functions. To find out, which functions are available, the banana-trick[36] can be used. Listing 175 shows the error message that is caused by the banana-trick.

The placeholder `LIBRARY` marks the place where the name of the library needs to entered. A function object is not a program that is executeable on its own. It is merely a library that is used by other programs. In our case, the function objects are called by the solvers. Therefore, the function objects are not compiled into executeables. The compiler creates libraries when the function objects are compiled. This libraries contain the functions in a machine readable form.

The keyword `enabled` is optional. With this keyword function objects can be excluded from execution.

```
functions
{
  NAME
  {
    type              TYPE;
    functionObjectLibs  ("LIBRARY");
    enabled             true;
    /*
      Definition
```

[36]If OpenFOAM expects a keyword from a limited set of allowed keywords, stating an invalid keyword usually causes OpenFOAM to print the list of allowed entries.

```
        */
    }
}
```

Listing 174: Definition of function objects in the file `controlDict`

```
--> FOAM FATAL ERROR:
Unknown function type banana

Valid functions are :

13
(
cellSource
faceSource
fieldAverage
fieldCoordinateSystemTransform
fieldMinMax
nearWallFields
patchProbes
probes
readFields
sets
streamLine
surfaceInterpolateFields
surfaces
)
```

Listing 175: Output of the *banana-trick*; applied to the keyword `type`

24.2 *probes*

The function *probes* saves the values of certain field quantities at specific points in space. Listing 176 shows an example of the definition of a *probes* function object.

This function object is of the type `probes`. The name of the function object is `probes1`. The data generated by this function is stored in the directory `probes1`. This directory contains a sub-directory. The name of this sub-directory corresponds to the time at which the simulation is started. This prevents files from being overwritten in case a simulation is continued at some point in time.

Figure 25 shows the directory tree after a simulation ended. There, the folder `probes1` contains a sub-directory named `0`. This is the time the simulation started. The `0` folder contains the files `p` and `U`.

The keywords `outputControl` and `outputInterval` are optional. They control – as their names suggest – the way the data is written to the hard drive.

`fields` contains the names of the fields that are of interest. `probeLocations` contains a set of points. The data of a specified field is computed for this locations and written to a file. The name of this file is the fields of interest. Listing 176 will result in two files. The file `p` contains the values of the pressure for all locations, the file `U` will contain the values of the velocity at all locations.

The function *probes* is contained in the file `libsampling.so`. This information can be gained from the tutorials. See Section 42.3 for more information about how to search the tutorials for specific information.

```
functions
{
  probes1
  {
    type              probes;
    functionObjectLibs ("libsampling.so");
    enabled           true;
    outputControl     timeStep;
    outputInterval    1;

    fields
    (
      p
      U
    );
    probeLocations
    (
      ( 0.0254 0.0253 0 )
      ( 0.0508 0.0253 0 )
    );
  }
}
```

Listing 176: The definition of *probes* in the file `controlDict`

```
caseDirectory
 ├── 0
 ├── More time steps
 ├── constant
 │    └── polyMesh
 ├── probes1
 │    └── 0
 │         ├── p
 │         └── U
 └── system
```

Figure 25: A part of the directory tree after the simulation ended

24.2.1 Pitfalls

Probe location outside the domain

If the probe location is outside of the domain OpenFOAM will issue a warning message and continue with the simulation.

```
--> FOAM Warning :
    From function findElements::findElements(const fvMesh&)
    in file probes/probes.C at line 102
    Did not find location (0.075 0 0.48) in any cell. Skipping location.
```

Listing 177: probe location outside of the domain

Unknown or non-existent field

If the probes dictionary contains fields that are not present to be probed, then no warning or error message will be issued. OpenFOAM simply continues computation. If the dictionary contains no valid fields to be probed, then the probe function will not be executed. Consequently no folder for storing the data will be created.

24.3 *fieldAverage*

fieldAverage computes time-averaged fields. Listing *lst:fieldAverageControlDict* shows an example of how this function is set up.

```
functions
{
  fieldAverage1
  {
    type                fieldAverage;
    functionObjectLibs ( "libfieldFunctionObjects.so" );
    enabled             true;
    outputControl       outputTime;
    fields
    (
      Ua
      {
        mean            on;
        prime2Mean      off;
        base            time;
      }
    );
  }
}
```

Listing 178: Definition of a *fieldAverage* function object in the file `controlDict`

24.4 *faceSource*

24.4.1 Average over a plane

faceSource extracts data from surfaces (faces). Listing 179 shows how the average of a field quantity over a cutting plane is set up.

```
functions
{
  faceObj1
  {
    type              faceSource;
    functionObjectLibs ("libfieldFunctionObjects.so");
    enabled           true;
    outputControl     outputTime;

    // Output to log&file (true) or to file only
    log               true;

    // Output field values as well
    valueOutput       false;

   // Type of source: patch/faceZone/sampledSurface
   source             sampledSurface;

   sampledSurfaceDict
   {
     // Sampling on triSurface
     type          cuttingPlane;
     planeType     pointAndNormal;
     pointAndNormalDict
     {
    basePoint ( 0 0 0.3 );
        normalVector ( 0 0 1 );
     }
     interpolate true;
   }

   // Operation: areaAverage/sum/weightedAverage ...
   operation          areaAverage;
   fields
   (
     alpha
   );
  }
}
```

Listing 179: Definition of a *faceSource* function object in the file `controlDict`

24.4.2 Compute volumetric flow over a boundary

Listing 180 shows the definition of a function object that is used to compute the volumetric flow over a boundary face. The key points for this are the definition of a weight field and the use of the summation operation. The weight field is automatically applied to the processed field, there is no need to specifically an operation such as *weightedSum*. If no weight field is defined, no weight field is used.

```
functions
{
    faceIn
    {
        type              faceSource;
        functionObjectLibs ("libfieldFunctionObjects.so");
        enabled           true;
        outputControl     timeStep;
        log               true;
        valueOutput       false;
        source            patch;
        sourceName        spargerInlet;
        surfaceFormat  raw;
        operation         sum;
        weightField       alpha1;

        fields
        (
            phi1
        );
    }
}
```
Listing 180: Definition of a *faceSource* function object in the file `controlDict`

24.4.3 Pitfall: `valueOutput`

The option `valueOutput` writes the field values on the sampled surface to disk. This can lead to massive disk space usage when setting `outputControl` to `timeStep`. In this case the field values are written for every time step. The option `valueOutput` should be disabled unless it is really needed.

Figure 26 shows the contents of the `postProcessing` folder after two time steps have been written to disk. For each sampled field the field values on the sampled patch are written to disk in files in the `surface` folder.

```
postProcessing
└─ faceObj1
   ├─ 0
   │  └─ faceSource.dat
   └─ surface
      ├─ 0.1
      │  ├─ phi_patch_outlet.raw
      │  ├─ p_patch_outlet.raw
      │  └─ U_patch_outlet.raw
      └─ 0.2
         ├─ phi_patch_outlet.raw
         ├─ p_patch_outlet.raw
         └─ U_patch_outlet.raw
```

Figure 26: The content of the `postProcessing` folder

24.5 Execute C++ code as functionObject

OpenFOAM makes it possible to execute C++ code as a functionObject[37]. This feature is disabled by default. To activate it a flag has to be changed. This is done for a single user in `~/.OpenFOAM/$WM_PROJECT_VERSION/controlDict` or system wide in `$WM_PROJECT_DIR/etc/controlDict`. In one of these files the flag shown in Listing 181 has to be set to one. It can be, that the first of these files does not exist, i.e. there are no user specific settings. The question of precedence (User setting over system wide setting) has not been pursued by the author.

Listing 182 shows an example of this feature. The field quantities $U1$, $U2$ and p are read in and some calculated values are printed to the Terminal.

[37]The *release notes* of OpenFOAM-2.0.0 suggest that this feature was introduced with version 2.0.0. See `http://www.openfoam.org/version2.0.0/`

```
// Allow case-supplied C++ code (#codeStream, codedFixedValue)
allowSystemOperations   1;
```

Listing 181: Allow case-supplied C++ code

```
extraInfo
{
  type                coded;
  functionObjectLibs ( "libutilityFunctionObjects.so" );
  redirectType        average;
  code
  #{
    const volVectorField& U1 = mesh().lookupObject<volVectorField>("U1");
    const volVectorField& U2 = mesh().lookupObject<volVectorField>("U2");
    Info << "max U1 = " << max(mag(U1)).value() << ", U2 = " << max(mag(U2)).value() << endl;
    const volScalarField& p = mesh().lookupObject<volScalarField>("p");
    Info << "p min/max = " << min(p).value() << ", " << max(p).value() << endl;
  #};
}
```

Listing 182: Define a *functionObject* using C++

When the solver is invoked, the so called coded functionObject is compiled on the fly. Listing 183 shows a portion of the solver output. Between the entry into the time loop and the first calculations, the code is read from `controlDict` and pasted into a template of a coded functionObject.

```
Starting time loop

Using dynamicCode for functionObject extraInfo at line 69 in "/home/user/OpenFOAM/user-2.1.x/
    run/twoPhaseEulerFoam/bubbleColumn/system/controlDict::functions::extraInfo"
Creating new library in "dynamicCode/average/platforms/linux64GccDPOpt/lib/
    libaverage_731fed868edc5a1d75988808649ac874cf00e044.so"
Invoking "wmake -s libso /home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bubbleColumn/
    dynamicCode/average"
wmakeLnInclude: linking include files to ./lnInclude
Making dependency list for source file functionObjectTemplate.C
Making dependency list for source file FilterFunctionObjectTemplate.C
'/home/user/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/bubbleColumn/dynamicCode/average/../
    platforms/linux64GccDPOpt/lib/libaverage_731fed868edc5a1d75988808649ac874cf00e044.so' is
    up to date.
Courant Number mean: 1.68517e-05 max: 0.00363
Max Ur Courant Number = 0.00363
Time = 0.001

MULES: Solving for alpha1
```

Listing 183: On the fly compilation of C++ coded functionObjects

OpenFOAM creates a directory named `dynamicCode` in the case directory. There, all files related to the coded functionObject can be found, source files as well as binaries. Figure 27 shows the directory tree after OpenFOAM compiled the coded functionObject.

```
caseDirectory
├── 0
├── constant
│   └── polyMesh
├── dynamicMesh
│   ├── average
│   │   ├── lnInclude
│   │   └── Make
│   │       └── linux64GccDPOpt
│   └── platforms
│       └── linux64GccDPOpt
│           └── libs
└── system
```

Figure 27: Directory tree after compilation of a coded functionObject

24.6 Execute *functions* after a simulation has finished

24.6.1 *execFlowFunctionObjects*

execFlowFunctionObjects is a post-processing tool of OpenFOAM. This tool allows the user to execute function objects after a simulation is finished. Normally, function objects are executed during the simulation. However, in some cases it is useful to apply a function to the data set of a already completed simulation, e.g. for testing the function.

Defining function objects in a seperate file

Listing 184 shows a file which contains only the definition of a function object. For the sake of clarity, this file is named `functionDict`. Defining functions in a seperate file reflects the division of labor in some way. The file `controlDict` is controlling the solver, whereas the file `functionDict` defines the function objects. The file `functionDict` can be included into the file `controlDict` by an `#include` statement. See Section 7.2.5 for examples.

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

functions
{
  probes1
  {
    type probes;
  functionObjectLibs ("libsampling.so");
    dictionary probesDict;
  }
}
```

Listing 184: Define functions in a seperate dictionary. The file `functionDict`

Run *execFlowFunctionObejcts*

execFlowFunctionObjects has to be told, that the functions are defined in a seperate file. By default, the tool reads the file `controlDict`. By using the parameter `-dict` the user can specify an alternative file containing the function dictionary.

```
execFlowFunctionObjects -noFlow -dict functionDict
```

Listing 185: Invokation of *execFlowFunctionObjects*

24.6.2 *postAverage*

postAverage is a small tool that is also designed to run functions on a already completed simulation. See Section 30.

25 *sample*

sample is a simple post processor. This tool is controlled by the file `sampleDict`. *sample* extracts data from the solution of a specific region. *sample* can extract data from the following geometric regions:

- from one or several points in space
- along a line
- on a face

sample is usually executed after a simulation has finished. See Section **??** for an example of using *sample*.

25.1 Usage

The simplest way to use *sample* is to call the command `sample`. In this case sample looks for a file named `sampleDict` located in the *system* directory. With the `-dict` an alternative file with a different name can be specified. However, this file has to reside in the *system* directory.

By default *sample* operates on all time steps. The option `-latestTime` can be used to sample only the latest solution data. The option `-time` can be used to specify a certain time or a time range to operate on.

Specifying a limited number of time steps to perform sampling on significantly reduces the time needed for this operation. The disk space used by the data generated by *sample* is usually in the order of up to a few megabytes. Therefore saving hard disk space is not an issue when using *sample*.

25.2 *sampleDict*

The file `sampleDict` controls what and where data is to be sampled.

25.2.1 Output format

There are 6 possible output formats (*csv, gnuplot, jplot, raw, vtk, xmgr*). The difference between the listed formats is the way how the data is organised inside the file.

sample creates one file for scalar quantities and one for vector quantities. The names of the data files are built from the names of the sampled fields, the output format and the name of the geometric set. E.g. `lineXuniform_Ua_Ub.csv`, this file contains the velocity fields `Ua` and `Ub` along the line `lineXuniform`. The data format of the sampled data is comma seperated values (`csv`).

25.2.2 Fields

The fields that are to be sampled are listed in the list `fields`.

Invalid entries are ignored, without any warning message. In the example of Listing 186 the list of fields contains the name `banana`. However, there is no field named `banana`, so *sample* will simply ignore this entry – *sample* will not issue any warning or error message. Thus, a typo in the `sampleDict` is not that easy to find. sample reports no warning but the intended field is not sampled. Always double check the entries in the fields sub-dictionary for typos, especially when sampling fields with composite names, e.g. `U2Mean` or `U2Prime2Mean`.

```
// Fields to sample.
fields
(
  alpha
  banana
  Ua
  Ub
);
```

Listing 186: Fields to sample in the file `sampleDict`

25.2.3 Geometric regions

The geometric regions on which *sample* can operate are

sets A set can contain one or several points or a line. Along a line, points can be distributed in an equidistant fashion.

surfaces A surface can be defined in several ways. Possible are, among others, cutting planes or iso-surfaces.

25.2.4 Pitfalls

Missing keywords

If the keywords *sets* and *surfaces* are missing in *sampleDict, sample* will run without producing any error messages or any data. If in Listing 187 the word *banana* would be replaced by *sets* and *orange* by *surfaces*, *sample* would work as expected. If *sample* is called with a *sampleDict* like in Listing 187, *sample* produces no data and issues no warning.

```
setFormat raw;

surfaceFormat vtk;

formatOptions
{
  ensight
  {
    format   ascii;
  }
}

interpolationScheme cellPoint;

fields
(
  p
  U
);

banana
(
  lineX1
  {
    type        uniform;
    axis        distance;
    start       (0.0015   0.5027  0.05);
    end         (0.0995   0.5027  0.05);
    nPoints     20;
  }
);

orange
(
);
```

Listing 187: Not working example of `sampleDict`

Faulty line definition

If the data along a line is to be sampled and the definition of the line is errorneous so that the line is outside the domain, *sample* will issue a warning message. Listing 188 shows an example of such a warning message. However, *sample* will not report an error and it will finish its run. So, when the output of *sample* is not checked, this might go unnoticed.

```
--> FOAM Warning :
     From function sampledSets::combineSampledSets(..)
    in file sampledSet/sampledSets/sampledSets.C at line 102
    Sample set lineX0 has zero points.
```

Listing 188: Warning message of *sample* due to a faulty line definition

26 *ParaView*

ParaView is a graphical post-processor. This program is called by invoking the command `paraFoam`. `paraFoam` is a script that calls *ParaView* with additional OpenFOAM libraries.

26.1 View the mesh

Besides viewing and post-processing simulation results, *ParaView* can be used to view the mesh. When refining a mesh it is important to check neighbouring blocks for the transition of mesh fineness. Figure 13 in Section 13 shows an example how *ParaView* displays a mesh.

Pitfall: default selection

If a user works on the refinement of the mesh and the definition of boundary conditions has not been made, then calling *ParaView* can crash because of its default selection of the pressure field. After pressing the Apply button *ParaView* tries to read in all selected fields. In case of a faulty definition of the boundary fields, this ends in the termination of the program. Listing 189 shows a corresponding error message.

```
--> FOAM FATAL IO ERROR:
keyword bottom is undefined in dictionary "/home/user/OpenFOAM/user-2.1.x/run/icoFoam/case01
    /0/p::boundaryField"

file: /home/user/OpenFOAM/user-2.1.x/run/icoFoam/case01/0/p::boundaryField from line 25 to
    line 35.

  From function dictionary::subDict(const word& keyword) const
  in file db/dictionary/dictionary.C at line 461.

FOAM exiting
```

Listing 189: Reading error due missing boundary field definition

Viewing the mesh

In this case the pressure field has to be manually unselected. If no fields are selected, *paraView* only reads the mesh information. Therefore, it is possible to view the mesh without the rest of the case properly set up. After the Apply button has been pressed and *paraView* has read all the data, the user has to choose from the representation drop-down menu in the toolbar the option Surface with edges.

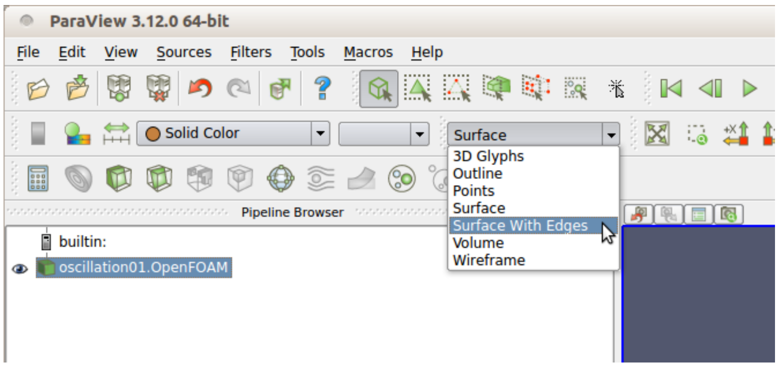


Figure 28: Select the proper representation to view the mesh

Part VII

External Tools

Besides *paraView*, there are a number of other useful tools, which do not come from the OpenFOAM Foundation. This section will cover such tools.

27 *pyFoam*

pyFoam is a collection of useful Python[38] scripts. These scripts are mostly written to serve one specific task. Further information can be found at `http://openfoamwiki.net/index.php/Contrib_PyFoam`.

27.1 Installation

The installation of *pyFoam* is described at `http://openfoamwiki.net/index.php/Contrib_PyFoam#Installation`. The major prerequisite for the use of *pyFoam* is, that a Python interpreter is installed. To check if a Python interpreter is installed on the system, simply type `python --version` in the Terminal. If a version number is displayed, like `Python 2.7.3`, then Python is installed. Otherwise, the operating system would display an error message, stating that the command `python` can not be found.

Further information about Python are found at `http://python.org/` and `http://docs.python.org/`.

27.2 *pyFoamPlotRunner*

The script *pyFoamPlotRunner* starts a simulation and plots the residuals like Fluent would do.

```
user@host:~/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/columnCase$ pyFoamPlotRunner.py
    twoPhaseEulerFoam
```

Listing 190: Calling *pyFoamPlotRunner*

27.3 *pyFoamPlotWatcher*

The script *pyFoamPlotWatcher* is intended to visualize solution data (e.g. residuals, time steps, Courant number, etc.) after the simulation has finished. This requires that the solver output is written into a file, see Section 8.1.1. *pyFoamPlotWatcher* does essentially the same job as *pyFoamPlotRunner* with the difference that the former tool is for finished simulations and the latter monitors a running simulation. So the description of the features of *pyFoamPlotWatcher* holds also true for *pyFoamPlotRunner*.

```
user@host:~/OpenFOAM/user-2.1.x/run/twoPhaseEulerFoam/columnCase$ pyFoamPlotWatcher.py LOGFILE
```

Listing 191: Calling *pyFoamPlotWatcher*

By default *pyFoamPlotWatcher* plots the curves of the residuals, continuity information and bounded variables. With options several other curves can be plotted (e.g. time step, iterations, Courant number, etc.). With regular expressions user specified data can be extracted from the log file.

Listing 192 shows the invokation of *pyFoamPlotWatcher* to plot additionally to the default selection also the Courant number. The processing of the solver output stored in the file `LOGFILE` is limited with the option `--end` with a specific value – 0.1 s in this case. There is also a `--start` option. The plot created by the command in Listing 192 is shown in Figure 29.

```
pyFoamPlotWatcher.py LOGFILE --end=0.1 --with-courant
```

Listing 192: Calling *pyFoamPlotWatcher* with some options

[38]Python is an interpreted programming language.

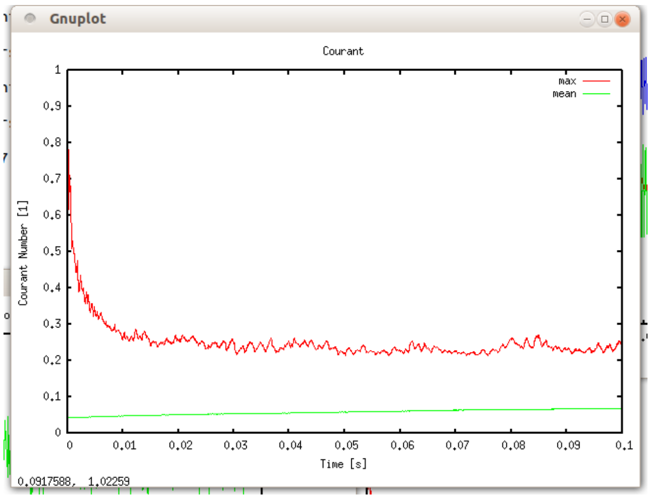


Figure 29: The Courant number plotted with *pyFoamPlotWatcher*.

27.3.1 Custom regular expressions

With regular expressions *pyFoamPlotWatcher* can extract arbitrary data from the solver output. This section elaborates this feature by the example of plotting the Courant number based on the relative velocity of a two-phase solver.

General information

pyFoamPlotWatcher has no option to display the history of the Courant number based on `Ur`, the relative velocity between the phases. Listing 193 shows some lines of the solver output of the two-phase solver *twoPhaseEulerFoam*. The line in red displays the Courant number based on the relative velocity `Ur`. The line above the red colored line displays the Courant number based on the mixture velocity, see Section 34.5.3 and 34.5.3 for information on the definition of the Courant number and the Courant number of the two-phase solver *twoPhaseEulerFoam*.

```
DILUPBiCG:  Solving for k, Initial residual = 0.000824921, Final residual = 1.47595e-06, No
    Iterations 2
ExecutionTime = 70870.7 s  ClockTime = 71186 s

Calculating averages

Courant Number mean: 0.103485 max: 0.422517
Max Ur Courant Number = 0.448791
deltaT = 0.00380929
Time = 72.5848

MULES: Solving for alpha1
MULES: Solving for alpha1
```

Listing 193: Some lines of the solver output of *twoPhaseEulerFoam*

Extracting the information

To extract the information from the log file we need to create a file containing the regular expression.

```
{"expr":"Max Ur Courant Number = (%f%)","name":"UrCoNum"}
```

Listing 194: The file `customRegexp`

If pyFoamPlotWatcher finds a file named `customRegexp` in the case directory, this file will be processed automatically. If the file containing the regular expression has another name or is located inanother place the option `--regexp-file=REG_EXP_FILE` can be used to specify the path to that file.

Listing 194 contains comma seperated entries (`"expr"` and `"name"`). The values are seperated by a colon from the name of the entries (e.g. `"name":"UrCoNum"`). The first entry contains the regular expression to extract the data. The second provides the name of the extracted data, but this entry can be omitted.

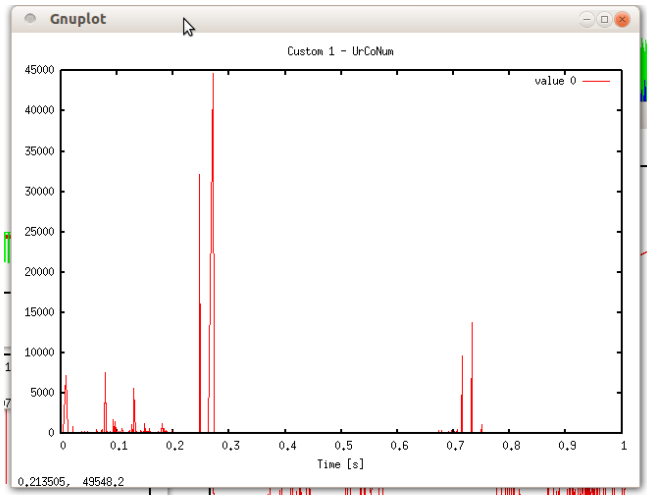


Figure 30: The Courant number based on the relative velocity plotted with *pyFoamPlotWatcher*

The absurdly high value of the Courant number indicates that the simulation did not go well. The need for plotting the Courant number based on `Ur` emanated from a trouble-shooting episode. Thus this section was written to preserve the gained knowledge.

27.3.2 Custom regular expression revisited

The plotting utilities of *pyFoam* (*pyFoamPlotRunner* and *pyFoamPlotWatcher*) accept custom regular expressions also in a different format than the format of Listing 194. This new format was introduced with version 0.5.3. See `http://openfoamwiki.net/index.php/Contrib_PyFoam#Plotting_with_customRegexp-files` for further information. The new format looks resembles an OpenFOAM dictionary.

Listing 195 shows an example of the solver output that will be post-processed. The goal is to draw curves of the quantities of the red line. Listing 196 shows the corresponding regular expression. The plotting utilities of *pyFoam* offer the `--dump-custom-regegexp` option to generate the custom regular expression in the new format from the old format. Listing 197 is the result of this operation.

```
DILUPBiCG:  Solving for beta, Initial residual = 0.000307666, Final residual = 7.36162e-08, No
     Iterations 2
DILUPBiCG:  Solving for T, Initial residual = 0.000514273, Final residual = 2.57279e-07, No
    Iterations 1
Concentration = 0.0509085 Min T = 0.00498731 Max T = 0.218343
Bubble load = 0.00623198  Min beta = 0  Max beta = 0.0677904
Time = 19.96
```

Listing 195: Some lines of the solver output to post-process

```
{"expr":"Concentration = (%f%)  Min T = (%f%)  Max T = (%f%)","name":"Concentration","titles
    ":["avg","min","max"]}
```

Listing 196: The custom regular expression in the odl format

```
Custom01
{
  accumulation first;
  enabled yes;
  expr "Concentration = (%f%)  Min T = (%f%)  Max T = (%f%)";
  name Custom01_Concentration;
  persist no;
  raisit no;
```

This offering is not approved or endorsed by ESI® Group, ESI-OpenCFD® or the OpenFOAM® Foundation, the producer of the OpenFOAM® software and owner of the OpenFOAM® trademark.

```
  theTitle "Custom 1 − Concentration";
  titles
    (
      avg
      min
      max
    );
  type regular;
  with lines;
  xlabel "Time [s]";
}
```

Listing 197: The custom regular expression in the new format

27.3.3 Special treatment of certain characters

Note that the solver output we processed so far contained no parentheses. The parentheses are interpreted by the regular expression. In order to deal with parentheses in the solver output they need to be escaped properly. The same is true for brackets. So the following example is also valid, when brackets are contained in the solver output that is to be processed with regular expressions.

Listing 198 shows some lines of solver output of twoPhaseEulerFoam. The line marked in red contains parentheses. In order to post-process these lines with regular expressions these parentheses need to be escaped in the regular epxression. Listing 199 shows the corresponding regular expression. Note the escaped parentheses marked in red.

```
Time = 19.9957

MULES: Solving for alpha1
MULES: Solving for alpha1
Dispersed phase volume fraction = 0.0168317 Min(alpha1) = 3.92503e-87 Max(alpha1) = 0.2
GAMG: Solving for p, Initial residual = 9.46269e−05, Final residual = 1.65711e−06, No
    Iterations 1
time step continuity errors : sum local = 2.08826e−05, global = 4.51574e−08, cumulative =
    −0.0334048
```

Listing 198: Some lines of the solver output of *twoPhaseEulerFoam*

```
{"expr":"Dispersed phase volume fraction = (%f%) Min\(alpha1\) = (%f%) Max\(alpha1\) = (%f%)
    ","name":"Volume fraction","titles":["avg","min","max"]}
```

Listing 199: The regular expression to extract the information about the volume fraction

27.3.4 Ignoring stuff

Listing 199 extracts three numbers from the line marked in Listing 198. Using this regular expression plots all three curves. If we are interested in only the first number – the average volume fraction – we replace the second and third `(%f%)` with a `.+` to ignore the second and third number. In this special case this seems an overkill – we could also delete parts of the expression since we are only interested in the first number – but if we are interested in the first and the third number, then we need to ignore the second number.

27.3.5 Producing images

The Figures 29 and 30 are screenshots of the images plotted by *pyFoamPlotWatcher*. However, there is the option `--hardcoded` that tells the *pyFoam* plot utilities to save the plots on the disk. By default a PNG image is produced but with the option `--format-of-hardcopy=HARDCOPYFORMAT` other formats can be chosen.

Figure 31 shows the plot produced by the regular expression of Listing 199.

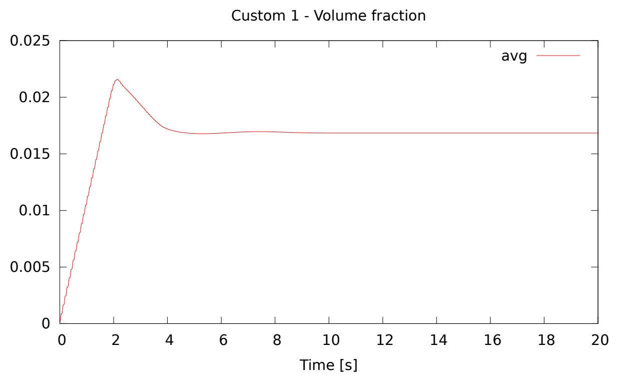


Figure 31: The average volume fraction plotted with *pyFoamPlotWatcher* and a custom regular expression

27.3.6 Writing data

Producing images is often not enough for post-processing. The option `--write-files` causes *pyFoam* to write the extracted data to the hard drive. Thus the extracted data can be processed by other programs.

27.4 *pyFoamClearCase*

As the name implies, *pyFoamClearCase* cleans the case directory. This script deletes all time directories save the *0* directory. By the use of command line options, a finer control of the actions of *pyFoamClearCase* is possible. Some of these options are:

–keep-last keep the last time step

–keep-regular keep all time steps

–after=T delete all time steps for $t > T$

–remove-processor delete the *processor** directories

The script is invoked by typing its name in the Terminal. Listing 200 shows how this script is executed. The options cause *pyFoamClearCase* to keep the last time directory and to remove all *processor** folders.

```
pyFoamClearCase.py . --keep-last --remove-processor
```

Listing 200: Calling *pyFoamClearCase*

Note the file ending `.py` after the name of the script. This ending indicates, that the script is written in Python. It also indicates, that *pyFoamClearCase* is an executable script rather than a program on its own.

27.5 *pyFoamCloneCase*

This script is used to copy a case. By default the *0*, the *constant* and the *system* directory are copied. Additionally, there are various command line arguments to control the operation of the script, e.g. copy also the latest time step or the *processor** directories.

27.6 *pyFoamDecompose*

This script is used to decompose the computational domain. Other than the tool *decomposePar*, this script does not need an existing `decomposeParDict`. This script receives command line arguments, generates the `decomposeParDict` and calls *decomposePar*.

In Listing 201 the script is called with two arguments. The first argument is the path to the case directory. In this case the dot refers to the currect directory. The second argument is the number of sub-domains. From this arguments, *pyFoamDecompose* creates a `decomposeParDict`. The first argument is necessary to tell the script where to save the newly created file. The second argument is the most fundamental information for domain decomposition – the number of sub-domains.

There is a large number of additional arguments which allow to exert more control over the way the domain is decomposed.

```
pyFoamDecompose.py . 4
```

Listing 201: Invokation of *pyFoamDecompose*

Listing 202 contains the `decomposeParDict` created by the command of Listing 201.

```
// * * * * * * * * * * //
FoamFile
{
  version 0.5;
  format ascii;
  root "ROOT";
  case "CASE";
  class dictionary;
  object nix;
}
method scotch;
numberOfSubdomains 4;
scotchCoeffs
{
}
```

Listing 202: The file `decomposeParDict` generated by *pyFoamDecompose decomposeParDict*

The output of *pyFoamDecompose* is stored in the file `Decomposer.logfile`.

27.7 *pyFoamDisplayBlockMesh*

If there is a problem with mesh topology and one isn't able to find the error in the *blockMeshDict*, this tool can be of great help. *pyFoamDisplayBlockMesh* does exactly what the name of the tool suggests. It reads *blockMeshDict* and displays the topology of the mesh. One might think, that that's exactly what is described in Section 11.6.1 (display the blocks with *paraView*). However, if the definition of the mesh is erroneous, *blockMesh* will not create a mesh and *paraView* is therefore not able to display the blocks.

pyFoamDisplayBlockMesh is a tool that allows the user to visualise a faulty mesh. This is of great help to find e.g. an error in the block definition, especially when there are more than one blocks. In Figure 32 a screenshot of the GUI of this tool is shown. In the main panel the vertices and the edges are displayed. With the two sliders below single blocks as well as patches can be marked and coloured. The local axes of a single block are displayed as tubes labelled with the corresponding names of the axes.

The blocks shown in Figure 32 have a faulty definition, so *blockMesh* produces an error message instead of creating a mesh. With the help of this tool, the cause for the error is easily found. The marked block should be in the right part of the geometry, so vertex number 5 should not be part of this block.

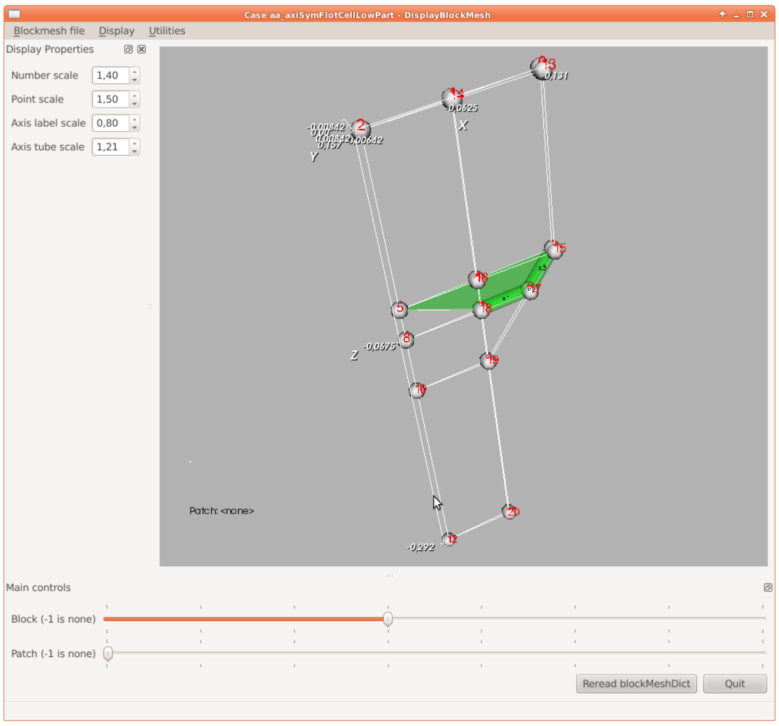


Figure 32: Screenshot of *pyFoamDisplayBlockMesh*

Right of the main panel the output of the standard meshing utilities *blockMesh* and *checkMesh* can be displayed (not shown in the picture). These utilities can be executed from the menu of this tool. Moreover, the *blockMeshDict* can be edited with this tool.

28 *swak4foam*

The name *swak4foam* comes from *SWiss Army Knife for Foam*. *swak4foam* evolved from a collection of tools like *groovyBC*, *funkySetFields* and *simpleFunctionObjects*. The documentation of *swak4foam* is located at `http://openfoamwiki.net/index.php/Contrib/swak4Foam`.

28.1 Installation

To install *swak4foam* one needs to download the source code and compile them. The source code of *swak4foam* is managed by the use of a *subversion*[39] repository. Listing 203 shows how the source code is downloaded by subversion. The first command changes the working directory of the terminal to `~/OpenFOAM`. The second command creates a directory named `swak4foam`. The third command changes the working directory of the terminal to the newly created folder and the last commands actually downloads the source code to the current directory.

```
cd ~/OpenFOAM
mkdir swak4foam
cd swak4foam
svn checkout https://openfoam-extend.svn.sourceforge.net/svnroot/openfoam-extend/trunk/
    Breeder_2.0/libraries/swak4Foam/
```

Listing 203: Installation of *swak4foam*

After downloading, the sources need to be compiled by calling `Allwmake`.

[39] *subversion*, abbreviated SVN, is a version control software to manage software projects.

28.2 *simpleSwakFunctionObjects*

simpleSwakFunctionObjects is an extension of *simpleFunctionObjects*. The functions of this library are used to post process data and extend functionality of OpenFOAM.

28.2.1 Extrema of a field quantity

If only the extrema of a field quantity are of interest, the tools of OpenFOAM (*probes*, *sample*) are of little use. One way of solving this problem could be, to modify the solver to write the extrema to the standard output. In Listing 204 some line of the standard output of *twoPhaseEulerFoam* are shown. This solver prints the mean value as well as the extrema of the volume fraction of the dispersed phase. The corresponding lines of source code can serve as a blueprint for a solver modification.

However, if the user is not inclined to modify and compile OpenFOAM solvers, *simpleSwakFunctionObjects* provide the solution.

```
DILUPBiCG:  Solving for alpha, Initial residual = 3.48391e-05, Final residual = 2.94111e-12,
    No Iterations 2
Dispersed phase volume fraction = 0.00824276  Min(alpha) = -1.66816e-19  Max(alpha) = 0.6
DILUPBiCG:  Solving for alpha, Initial residual = 3.71563e-07, Final residual = 8.16115e-14,
    No Iterations 2
Dispersed phase volume fraction = 0.00824276  Min(alpha) = -3.31819e-19  Max(alpha) = 0.6
```

Listing 204: Solver-Ausgabe von *twoPhaseEulerFoam*

swakExpression

The function to do the job is called *swakExpression*. This function is part of the library *libsimpleSwakFunctionObjects*. Listing 205 shows how this function is set up as a function object in the file `controlDict`. In this example the minimal value of the field `alpha` is saved. Notice the statement in last line of the Listing. This statement tells the solver to use the specified library. This library contains the function `swakExpression`. See Section 7.2.3 for further information about using external libraries.

```
functions
{
  minAlpha
  {
    type swakExpression;
    verbose true;
    accumulations ( min );
    valueType internalField;
    expression "min(alpha)";
  }
}

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

Listing 205: Definition of the function *swakExpression* in the file `controlDict`

Keywords

This section explains the most important keywords of Listing 205.

type specifies the type the function object

verbose a switch that controls whether the generated data is to be printed on the solver output or not. The data is written into a file anyway.

accumulations allowed entries: {`min,max,average,sum`}. Quote from the CFD-Online Forum[40]: *accumulations is only needed if you need "a single number" to print to the screen. For instance if you use a swakExpression-FO to print the maximum and minimum of your field to the screen.*

[40]`http://www.cfd-online.com/Forums/openfoam/103504-swak4foam-calculating-velocity-transformations.html`

valueType defines the type of the geometric region on which the function is applied. Allowed entries: {`internalField cellSet faceZone patch faceSet set surface cellZone`}

expression defines the quantity that is sought for. This can be a simple statement or a formula computing a quantity.

29 *blockMeshDG*

blockMeshDG is a modification of the meshing tool *blockMesh* to allow for double grading. Double grading means, that the ratio between the discretisation length of the middle and the ends of an edge is prescribed. This tool was developed by some users of OpenFOAM and is was published in the CFD-Online OpenFOAM Forum (`http://www.cfd-online.com/Forums/openfoam/70798-blockmesh-double-grading.html`). There is also a page in the OpenFOAM Wiki (`http://openfoamwiki.net/index.php/Contrib_blockMeshDG`).

29.1 Installation

The downloaded source code is ready for compilation after unpacking. All necessary entries have already been made to prevent the new utility to collide with the standard utilities of OpenFOAM. The make script creates an executable named *blockMeshDG*.

29.2 Usage

To discern between normal grading and double grading, the expansion ratio needs to be negative for double grading[41]. A positive entry causes normal grading to be applied just like it is the case with the standard utility.

29.3 Pitfalls

29.3.1 Uneven number of cells

blockMeshDG obviously has a problem with an uneven number of cells. Figure 33 shows the resulting mesh, when 15 cells are used for the double graded edge. In this case, although the mesh is of bad quality, *checkMesh* reports no error. However, the output of checkMesh contains some indications that something is not alright.

Listing 206 shows some lines of the output of *checkMesh*. The very high aspect ratio is an indicator that something is wrong with the mesh. Also the fact that the minimum and maximum values of face area or cell volume differ by up to three orders of magnitude should lead to the same conclusion. Unfortunately, *checkMesh* issues not even a warning message.

```
Checking geometry...

    Max aspect ratio = 81 OK.
    Minimum face area = 3.8395e-08. Maximum face area = 1.68746e-05.  Face area magnitudes OK.
    Min volume = 9.59875e-11. Max volume = 4.21864e-08.  Total volume = 4.92214e-05.  Cell
    volumes OK.
    Mesh non-orthogonality Max: 42.2304 average: 11.7938
    Non-orthogonality check OK.
    Min/max edge length = 3.079e-05 0.00508035 OK.
```

Listing 206: Some output of *checkMesh*

So far, the only solution to this problem is to use an even number of cells.

[41]A negative entry unequal to unity causes *blockMesh* to crash with a floating point exception. Therefore, using negative entries for double grading does not alter the standard behaviour.

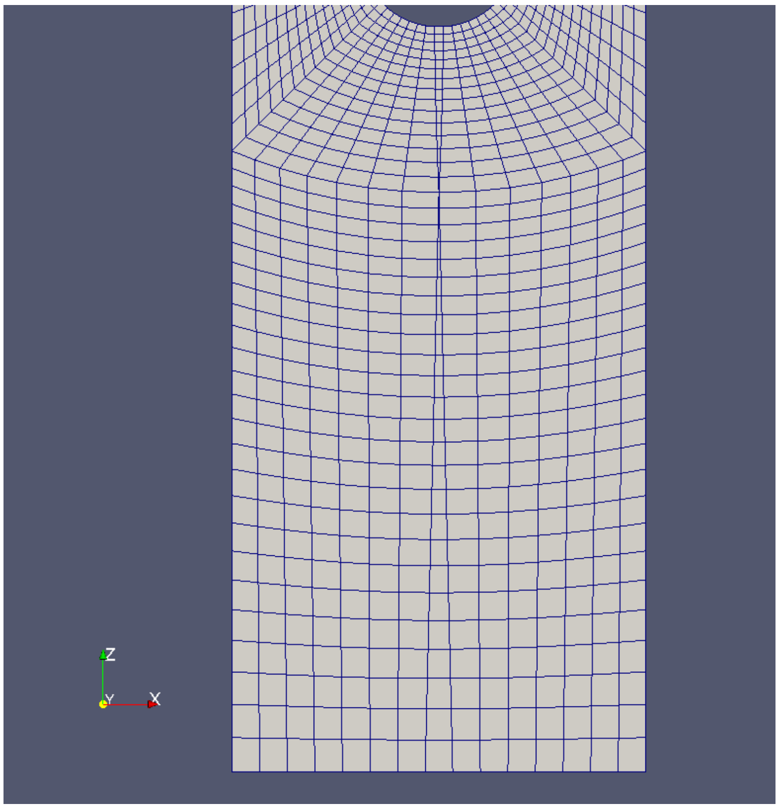


Figure 33: Double grading problem

30 *postAverage*

30.1 Motivation

This utility allows the user to execute functions after a simulation has finished. Normally, functions are executed during the run-time of the solver.

The idea and most of the source code for this tool stems from the CFD Online Forum [`http://www.cfd-online.com/Forums/openfoam-programming-development/70396-using-fieldaverage-library-average-postprocessing.html#post237751`]. This tool iterates over all time steps and executes the functions at run-time. Basically, this tool is a solver, that solves no equations.

30.2 Source code

The Listings 208 and 207 show the source code of this tool. The file `createFields.H` contains all statements responsible for reading the existing fields. The functions can only be applied to fields that were created in `createFields.H`.

The file `createFields.H` contains statements that allow the tool to be applied on simulation data following both the old and the new naming convention of *twoPhaseEulerFoam*. The source code contains the field names. In order to avoid writing a seperate tool for each naming scheme, the fields are read conditionally. I.e. the tool trys to read only if the corresponding file is present. Otherwise the tool would abort with an error for trying to access a non-existent file.

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

```
License
    This file is part of OpenFOAM.

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

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

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

Application
    postAverage

Gerhard Holzinger based on work by Eelco van Vliet

Description
    Post-processes data from flow calculations
    For each time: calculates the time average of a sequence of fields and
    writes time time average in the directory

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

#include "fvCFD.H"

int main(int argc, char *argv[])
{
    argList::noParallel();
    timeSelector::addOptions();

   #include "setRootCase.H"
   #include "createTime.H"

    instantList timeDirs = timeSelector::select0(runTime, args);
    runTime.setTime(timeDirs[0], 0);
   #include "createMesh.H"

    forAll(timeDirs, timeI)
    {
        runTime.setTime(timeDirs[timeI], timeI);
        Info<< "Adding fields for time " << runTime.timeName() << endl;
       #include "createFields.H"

        runTime.functionObjects().execute();
    }

    Info<< "\nEnd" << endl;

    return 0;
}

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

Listing 207: The file `postAverage.C`

```
    /* --------------------------------------------------
                        read always
    -------------------------------------------------*/
    Info<< "Reading field p\n" << endl;
    volScalarField p
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::NO_WRITE
            ),
            mesh
        );


        /* ------------------------------------------------------
                        read only if they exist
        ------------------------------------------------------*/
        IOobject UHeader
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ
        );

        autoPtr<volVectorField> U;

        if (UHeader.headerOk())
        {
            Info<< "Reading U.\n" << endl;

            U.set(new volVectorField
            (
                IOobject
                (
                    "U",
                    runTime.timeName(),
                    mesh,
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                mesh
            ));
        }


        IOobject UrHeader
        (
            "Ur",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ
        );

        autoPtr<volVectorField> Ur;

        if (UrHeader.headerOk())
        {
            Info<< "Reading Ur.\n" << endl;

            Ur.set(new volVectorField
            (
                IOobject
                (
                    "Ur",
                    runTime.timeName(),
                    mesh,
                    IOobject::MUST_READ,
                    IOobject::AUTO_WRITE
                ),
                mesh
            ));
        }


        /* old naming convention for two-phase solvers */
        /*   alpha, Ua, Ub, phia, phib */
        IOobject alphaHeader
        (
        "alpha",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<volScalarField> alpha;

    if (alphaHeader.headerOk())
    {
      Info<< "Reading field alpha\n" << endl;
      alpha.set(new volScalarField
      (
          IOobject
          (
              "alpha",
              runTime.timeName(),
              mesh,
              IOobject::READ_IF_PRESENT,
              IOobject::NO_WRITE
          ),
          mesh
      ));
    }


    IOobject UaHeader
    (
        "Ua",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<volVectorField> Ua;

    if (UaHeader.headerOk())
    {
        Info<< "Reading Ua.\n" << endl;

        Ua.set(new volVectorField
        (
            IOobject
            (
                "Ua",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }


    IOobject UbHeader
    (
        "Ub",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<volVectorField> Ub;

    if (UbHeader.headerOk())
    {
        Info<< "Reading Ub.\n" << endl;

        Ub.set(new volVectorField
        (
            IOobject
            (
                "Ub",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }


    IOobject phiaHeader
    (
        "phia",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<surfaceScalarField> phia;

    if (phiaHeader.headerOk())
    {
        Info<< "Reading phia.\n" << endl;

        phia.set(new surfaceScalarField
        (
            IOobject
            (
                "phia",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }


    IOobject phibHeader
    (
        "phib",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<surfaceScalarField> phib;

    if (phibHeader.headerOk())
    {
        Info<< "Reading phib.\n" << endl;

        phib.set(new surfaceScalarField
        (
            IOobject
            (
                "phib",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }

    /* new naming convention for two-phase solvers */
    /*   alpha1, U1, U2, phi1, phi2 */
    IOobject alpha1Header
    (
        "alpha1",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<volScalarField> alpha1;

    if (alpha1Header.headerOk())
    {
        Info<< "Reading alpha1.\n" << endl;

        alpha1.set(new volScalarField
        (
            IOobject
            (
                "alpha1",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }

    IOobject U1Header
    (
        "U1",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<volVectorField> U1;

    if (U1Header.headerOk())
    {
        Info<< "Reading U1.\n" << endl;

        U1.set(new volVectorField
        (
            IOobject
            (
                "U1",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }


    IOobject U2Header
    (
        "U2",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<volVectorField> U2;


    if (U2Header.headerOk())
    {
        Info<< "Reading U2.\n" << endl;

        U2.set(new volVectorField
        (
            IOobject
            (
                "U2",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }


    IOobject phi1Header
    (
        "phi1",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<surfaceScalarField> phi1;

    if (phi1Header.headerOk())
    {
        Info<< "Reading phi1.\n" << endl;

        phi1.set(new surfaceScalarField
        (
            IOobject
            (
                "phi1",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }

    IOobject phi2Header
    (
        "phi2",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ
    );

    autoPtr<surfaceScalarField> phi2;

    if (phi2Header.headerOk())
    {
        Info<< "Reading phi2.\n" << endl;

        phi2.set(new surfaceScalarField
        (
            IOobject
            (
                "phi2",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            mesh
        ));
    }
```

Listing 208: The file `createFields.H`

Naming scheme	old		new	
Phase	a	b	1	2
Volume fraction	***alpha***	*beta*	***alpha1***	*alpha2*
Velocity	***Ua***	***Ub***	***U1***	***U2***
Density	*rhoa*	*rhob*	*rho1*	*rho2*
Flux	*phia*	*phib*	*phi1*	*phi2*

Tabelle 4: Naming scheme of quanities of *twoPhaseEulerFoam*

Teil VIII
Updates

31 General remarks

OpenFOAM is like any other open source project continuously updated. Those updates are integrated relatively fast into the Git repository (e.g. OpenFOAM 2.1.x). In larger periods a new release of OpenFOAM is published (e.g. OpenFOAM 2.1.1).

In the course of the creation of this document OpenFOAM evolves as well. In this chapter changes relevant to this manual will be pointed out.

32 OpenFOAM

32.1 OpenFOAM-2.1.x

32.1.1 Naming scheme of two-phase solvers

The naming scheme of the two-phase solvers of OpenFOAM has been changed after the release of Version 2.1.1. This change affected OpenFOAM-2.1.x around July 2012. The velocities used by two-phase solvers are now named *U1* and *U2* instead of *Ua* and *Ub*. The volume fraction is consequently named *alpha1*. Other variables, e.g. density, also bear the number of the phase (*rho1* and *rho2*). Table 4 shows a selection of old and new names. The bold names are the names of files in the *0*-directory.

32.2 OpenFOAM-2.2.x

This section describes changes in behaviour or usage of OpenFOAM-2.2.x compared to OpenFOAM-2.1.x.

32.2.1 fvOptions

The *fvOptions* mechanism is an abstraction to allow for a generic treatment of physical models. See `http://www.openfoam.org/version2.2.0/fvOptions.php`.

32.2.2 postProcessing

The data generated by a *probes* function object or by the *sample* utility is now stored in a folder named `postProcessing`. This folder then contains a directory with the same name as the function object.

32.3 OpenFOAM-2.3.x

Although this manual is based on OpenFOAM-2.1 and OpenFOAM-2.2 this section lists some major differences to OpenFOAM-2.3.

32.3.1 *twoPhaseEulerFoam*

There have been major changes with the two-phase Eulerian solver *twoPhaseEulerFoam*. Simulation cases of OpenFOAM-2.1 or OpenFOAM-2.2 are not directly usable in OpenFOAM-2.3.

Part IX
Source Code & Programming

33 Understanding some C and C++

In this Section some features of the C++ programming language are discussed.

33.1 Definition vs. Declaration

In C and C++ there is the destinction between the declaration and the definition of a variable. Briefly explained, declaring a variable only tells the compiler that the variable exists and has a certain type. The declaration does not specify what the variable actually is.

A definition also tells the compiler what exactly a variable is. This does not necessarily mean that the variable is assigned a value.

Further information on that matter can be found in [17, 11] or `http://www.cprogramming.com/declare_vs_define.html`.

33.1.1 A classy example

In Listing ?? we define the class `phaseInterface`, i.e. we tell the compiler what the class looks like (data members, methods, etc.). Within the class `phaseInterface` we want to use the class `phaseModel`. This class already exists and is defined elsewhere, so there is no need for us to repeatedly define the class `phaseModel`. Creating our own definition of `phaseModel` would be useless and stupid.

To be able to use the existing class `phaseModel` we need to introduce this class to the compiler. In Line 4 of Listing ?? we do exactly this. We tell the compiler, that there is a class named `phaseModel`, that is all the information needed by now. This is sometimes referred to as *forward declaration*.

When we compile our class we need to make sure that we include the definition of `phaseModel`, e.g. via linking to the library in which `phaseModel` is defined.

```
1  namespace Foam
2  {
3
4  class phaseModel;
5
6  class phaseInterface
7  {
8    // lots of C++ code
9  };
10
11 }
```

Listing 209: Declaration and definition of classes

33.2 Namespaces

Namespaces are a feature of C++ to support a logical structure within the program. The basic idea behind namespaces put in simple words is *to keep things (variables and functions) visible where they need to be visible.* Like any other method of keeping things neat and tidy you could also survive without namespaces. However, to loosely quote Prof. Jasak one of the founders of OpenFOAM: *OpenFOAM is an example of how to make proper use of C++*. Therefore, we have a closer look on namespaces in OpenFOAM.

General information about the concept of namespaces can be found here:

- `http://www.cplusplus.com/doc/tutorial/namespaces/`
- `http://www.cprogramming.com/tutorial/namespaces.html`
- `http://www.learncpp.com/cpp-tutorial/711-namespaces/`

Some OpenFOAM specific aspects related to namespaces are discussed in Section 34.2.

33.3 `const` correctness

The `const` keyword has several uses and using `const` has some implications.

33.3.1 Constant variables

This is the most easy part. Any variable can be declared constant by using the `const` keyword. This can precede the datatype or the variable name. Both lines in Listing 210 are correct statements.

```
const int limit = 5;
int const answer = 42;
```

Listing 210: Constant variables

33.3.2 Constants and pointers

Pointing to a constant

A pointer can be used to point to a constant variable. The pointer itself is not constant and therefore changeable. However, the keyword `const` has to be used when declaring a pointer pointing to a constant variable. However, a pointer pointing to a constant can also point to a non-constant variable.

```
int const constVar1 = 42;
const int constVar2 = 13;
int variable = 11;

const int* pointer = &constVar1;

std::cout << "The pointer points to " << *pointer << std::endl;

// change the pointer
pointer = &constVar2;

std::cout << "The pointer points to " << *pointer << std::endl;

// point to a non-constant
pointer = &variable;

std::cout << "The pointer points to " << *pointer << std::endl;
```

Listing 211: Pointing to constant variables

```
The pointer points to 42
The pointer points to 13
```

Listing 212: Output of Listing 211

A constant pointer

A pointer can be constant regardless of the variable it points to. So, the address stored in the pointer can not be changed, the pointer will always point to the same variable. However, the variable itself can be altered. Listing 213 shows an example.

```
int variable = 11;

int* const constPointer1 = &variable;

std::cout << "The constant pointer points to " << *constPointer1 << std::endl;

variable = 79;

std::cout << "The constant pointer points to " << *constPointer1 << std::endl;
```

Listing 213: Using constant pointers

```
The constant pointer points to 11
The constant pointer points to 79
```

Listing 214: Output of Listing 213

A constant pointer to a constant

It is also possible to create a constant pointer pointing to a constant variable.

However, the last line of Listing 215 seems a bit unlogical but it isn't. To get the meaning of this line correctly, we need to read the left hand side of the assignment from right to left. First of all `constpointer4` is the name of the new variable. Secondly, `int* const` tells the compiler that the new variable is a constant pointer to an integer. This means, that the pointer itself – the location it points to – can not be changed. The last statement `const` at the very beginning of the line, means, that the variable the pointer points to can not be changed. However, `variable` is not a constant, so it can be altered anyway. The last line of Listing 215 does not change the nature of the variable `variable`, but it restricts the pointer to read-only operations. So, `variable` can be changed, but not using `constPointer4`.

```
int const constVar1 = 42;
int variable = 11;

const int* const constPointer2 = &constVar1;
const int* const constPointer4 = &variable;
```

Listing 215: A constant pointer to a constant

33.4 Function inlining

Motivation

Functions that carry out only a small number of operations are not very efficient, because the function call might take more time than the execution of all the operations. Especially if such a function is often called, the performance of the program suffers. However, writing functions is a good way to keep the code tidy.

On the one hand, functions enable the programmer to seperate code in a logical way. Code that is written for a specific task is outsourced into a function with a hopefully meaningful name. This improved readability and maintainability of the code.

One the other hand is writing functions a proper way to avoid code redundancy. Tasks that are carried out repeatedly are best put into a function. Therefore, the code has to be written only once and the function can be used wherever it is necessary.

The `inline` statement

The solution for this conflict is function inlining. The `inline` statement allowes the compiler to replace the function call with the function body, i.e. the operations performed by the function. This enables the programmer to keep the code tidy without the disadvantage of wasting time for time consuming function calls.

Listing 216 shows the definition of an inline function. The function body contains only two logical operations. The `inline` statement precedes the data type of the return value. So, writing inline functions is not different than writing ordinary functions.

```
inline bool Foam::pimpleControl::finalIter() const
{
  return converged_ || (corr_ == nCorrPIMPLE_);
}
```

Listing 216: The definition of an inline function

The use of the `inline` statement does not guarantee that the compiler replaces the function call. This depends on the compiler and the compiler settings.

OpenFOAM specifics

The OpenFOAM Code Style Guide (`http://www.openfoam.org/contrib/code-style.php`) demands from programmers to seperate the definition of inline and non-inline functions.

> Use inline functions where appropriate in a separate *classNameI.H* file.

Listing 217 shows the contents of the folder `pimpleControl`. Dividing the code of a program or a module into *.C and the *.H file is the common way to seperate declarations from the rest of the program. The *.dep file is generated by the compiler during compilation. The fourth file in the folder is a second header file as demanded by the Code Style Guide. Listing 216 is a part of `pimpleControlI.H`.

```
pimpleControl.C   pimpleControl.dep   pimpleControl.H   pimpleControlI.H
```

Listing 217: Content of the folder `pimpleControl`

33.5 Constructor (de)construction

In object oriented programming (OOP) everything is an object. All object are created by a constructor and if necessary destroyed by a destructor.

33.5.1 General syntax

The constructor is a method of a class like any other function or method[42]. However, the constructor is bound to comply some rules.

- The constructor always has the same name as its class
- The constructor has no return value

Listing 218 shows a simple class describing a point in a two-dimensional domain. This class has two constructors. The first constructor receives no arguments and initialises the member variables with zero. The second constructor receives two integer variables as arguments and uses this variables to initialize the member variables `xPos` and `yPos`.

Writing two or more constructors is possible because C++ supports function overloading. This means there can be several functions with the same name differing in the input arguments.

```
class Point
{
  int xPos;
  int yPos;

  public:
    Point()
    {
      /* constructor code */
      xPos = 0;
      yPos = 0;
    }
    Point(int x, int y)
    {
      xPos = x;
      yPos = y;
    }
};
```

Listing 218: A class for a 2D point

[42]The terms function and method are used interchangeably. However, the method indicates the use of object oriented programming. The term function is also used in procedural programming and does not automatically indicate the use of OOP.

Listing 219 demonstrates hot to create new variables of the type `Point`. The first line creates a variable of the type `Point`. Because no arguments are passed in this line, the first constructor of Listing 218 is called by the compiler.

The second line creates also a point. The numbers inside the parenthesis are passed to the constructor. Therefore the second constructor of Listing 218 is called and the member variables are initialised based on the arguments.

```
Point p1;
Point p2(3, 8);
```

Listing 219: Using the class for a 2D point

33.5.2 Copy-Constructor

The copy constructor is used to create a copy of an object. The C++ compiler will create a default copy constructor if the programmer does not write one. However, the default copy constructor has restrictions regarding the handling of complex classes.

```
Point::Point(Point & p)
{
  /* copy constructor code */
  xPos = p.xPos;
  yPos = p.yPos;
}
```

Listing 220: The copy constructor for the 2D point class

Hiding the copy constructor

A copy constructor can be hidden. Therefore, no copying is allowed. To do so, the copy constructor must be defined using a `private` modifier.

Listing 221 shows a simple example of a copy constructor that is declared as `private`. This means the copy constructor can only be called from within the class itself, i.e. only within the class `Point`.

Listing 222 shows an example from within the source code of OpenFOAM. There, the copy constructor of the class `turbulenceModel` is hidden by declaring it `private`.

```
class Point
{
  private:
    Point(Point & p);
};
```

Listing 221: Hiding the copy constructor

```
class turbulenceModel
:
    public regIOobject
{
private:
  // Private Member Functions

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

  /* code continues */
```

Listing 222: Hiding the copy constructor

33.5.3 Initialisation list

A class in C++ can have member variables of any type. Complex classes may need some kind of initialisation to ensure all variables have a defined state. When an instance of a class is created by the constructor, the initialisation list contains all statements to initialise member variables of the class.

Listing 223 shows a simple example of a constructor with an initialisation list. Listing 269 in Section 39.2.2 shows an usage example of an initialisation list in the OpenFOAM sources.

```
class Rectangle
{
  Point topLeft;
  Point bottomRight;

  public:
    Rectangle()
    {
      topLeft = Point();
      bottomRight = Point();
    }

    Rectangle(Point a, Point b)
    :
      topLeft(a),
      bottomRight(b)
    {
      /* constructor code */
    }
}
```

Listing 223: A constructor with an initialisation list

33.6 Object orientation

33.6.1 Abstract classes

See Section 34.6 for a discussion about the implementation of the generic turbulence models in OpenFOAM. This generic turbulence modelling makes heavy use of abstract classes and inheritance.

34 Under the hood of OpenFOAM

This section contains short code examples that in some way explain the behaviour of OpenFOAM in certain situations. All examples in this section are motivated by other parts of this manual. In some cases the source code of some applications is examined somewhere else.

34.1 Solver algorithms

See Sections 18, 19 and 20 in Part V.

34.2 Namespaces

34.2.1 Constants

Physics is full of constants. Therefore it would be nice to have a central location in which physical or mathematical constants are defined. OpenFOAM provides constants within the namespace `Foam::constant`. There the pre-defined constants are divided into the groups, such as

- electromagnetic
  - `mu0` - the magnetic permeability of vacuum
  - `epsilon0` - the electcial permittivity of vacuum
- physicoChemical

  - `R` - the universal gas constant

- mathematical
  - `pi` - π
  - `e` - the Euler number

In Listing 224 it is demonstrated how to access the constant pi within the source code. Listing 225 shows all the mathematical constants defined in OpenFOAM-2.2.x. From a computational performance point of view it makes perfect sense to pre-define often used constants such as two pi. Also note that instead of diving pi by 2.0 it is multiplied with 0.5. Mathematically these operations are equivalent, however, in terms of computational cost the floating point multiplication is to be preferred over the floating point division as it is much faster [2].

Also note that OpenFOAM does not define e and π on its own, it rather uses the constants provided by the system library. See e.g. `http://www.gnu.org/software/libc/manual/html_node/Mathematical-Constants.html` for the mathematical constants provided by the GNU C library (glibc). Thus `e` and `pi` are defined by accessing `M_E` and `M_PI`.

Further note that the constants are declared with the `const` specifier, which is the only sane way to define constants in C and C++.

```
scalar foo = constant::mathematical::pi;
```

Listing 224: A useless code example demonstrating the access to π with OpenFOAM's source code

```
const scalar e(M_E);
const scalar pi(M_PI);
const scalar twoPi(2*pi);
const scalar piByTwo(0.5*pi);
```

Listing 225: The mathematical constants provided by `mathematicalConstants.H`

In the FOAM-extend the access to e.g. the mathematical constants works the same way. Only the namespace is named `mathematicalConstants` instead of `constant::mathematical`. This is due to the fact that FOAM-extend is largely based on OpenFOAM-1.6.

34.3 Keyword lookup from dictionary

There are generally two kinds of keywords in a dictionary. There are mandatory keywords and optional ones.

34.3.1 Mandatory keywords

When a mandatory keyword is not found in a dictionary, OpenFOAM issues an error message and terminates.

Listing 226 shows the reading operation for three mandatory keywords. The function `lookup()` can be examined further in Listing 227.

```
#include "readTimeControls.H"

int nAlphaCorr(readInt(pimple.dict().lookup("nAlphaCorr")));
int nAlphaSubCycles(readInt(pimple.dict().lookup("nAlphaSubCycles")));
Switch correctAlpha(pimple.dict().lookup("correctAlpha"));
```

Listing 226: The content of `readTwoPhaseEulerFoamControls.H`

The code

Line 32 in Listing 227 shows, that the function `lookup()` simply calls value of `lookupEntry()`. This method also calls another method (`lookupEntryPtr()`) and does the error handling. The error handling routine clearly shows, that OpenFOAM will terminate in case the keyword wasn't found (see line 19).

```
const Foam::entry& Foam::dictionary::lookupEntry
(
  const word& keyword,
  bool recursive,
  bool patternMatch
) const
{
  const entry* entryPtr = lookupEntryPtr(keyword, recursive, patternMatch);

  if (entryPtr == NULL)
  {
    FatalIOErrorIn
    (
      "dictionary::lookupEntry(const word&, bool, bool) const",
      *this
    )
    << "keyword " << keyword << " is undefined in dictionary "
    << name()
    << exit(FatalIOError);
  }

  return *entryPtr;
}

Foam::ITstream& Foam::dictionary::lookup
(
  const word& keyword,
  bool recursive,
  bool patternMatch
) const
{
  return lookupEntry(keyword, recursive, patternMatch).stream();
}
```

Listing 227: Some content of `dictionary.C`

34.3.2 Optional keywords

A method that is used to read an optional keyword from a dictionary is usually provided with a default value. This default value is used in the case that the keyword is non-existent in the dictionary.

Listing 228 shows the reading operation for three optional keywords. The read function is called with two arguments. The first is the keyword and the second is the default value. If the function `lookupOrDefault()` finds no entry, then the default value is returned.

```
const bool adjustTimeStep =
  runTime.controlDict().lookupOrDefault("adjustTimeStep", false);
scalar maxCo =
  runTime.controlDict().lookupOrDefault<scalar>("maxCo", 1.0);
scalar maxDeltaT =
  runTime.controlDict().lookupOrDefault<scalar>("maxDeltaT", GREAT);
```

Listing 228: The content of `readTimeControls.H`

The code

Listing 229 shows the definition of the function `lookupOrDefault()`. This function also calls another function to lookup the keyword – actually it looks for the value assigned to the specified keyword in the dictionary – and enters a conditional branch. In case the keyword was found, the corresponding value is returned (line 14). If the keyword was not found, then the default value is returned (line 18).

In Listing 229 the function is defined with four input arguments. However, in Listing 228 this function is called with only two arguments.

The solution for this contradiction can be found in the file `dictionary.H`, where this function is declared. This declaration can also be found in Listing 230. There, in lines 6 and 7, default values for two arguments are specified. Therefore, the function can be called with only two arguments – with the two arguments that have no default value[43]. If the function is called with all its arguments, the passed argument overrides the default value.

When declaring a function that uses default values for its arguments, the arguments without default value must precede the arguments that have a default value. Otherwise, there could be ambiguity.

```
template<class T>
T Foam::dictionary::lookupOrDefault
(
  const word& keyword,
  const T& deflt,
  bool recursive,
  bool patternMatch
) const
{
  const entry* entryPtr = lookupEntryPtr(keyword, recursive, patternMatch);

  if (entryPtr)
  {
    return pTraits<T>(entryPtr->stream());
  }
  else
  {
    return deflt;
  }
}
```

Listing 229: Some content of `dictionaryTemplates.C`

```
template<class T>
T lookupOrDefault
(
  const word&,
  const T&,
  bool recursive=false,
  bool patternMatch=true
) const;
```

Listing 230: Some content of `dictionary.H`

34.4 OpenFOAM specific datatypes

34.4.1 The `Switch` datatype

A lot of settings in dictionaries are switches to activate or deactivate a feature. Listing 231 shows the part of the source code defining all valid values. Inside the source code a switch can only be true or false, as the class `Switch` is used as a boolean data type. However, in the dictionaries a switch can have more values – provided they denote a decision. Human languages usually have more ways of answering a yes-no question, this may be the motivation for allowing this range of values for switches.

```
// NB: values chosen such that bitwise '&' 0x1 yields the bool value
// INVALID is also evaluates to false, but don't rely on that
const char* Foam::Switch::names[Foam::Switch::INVALID+1] =
{
  "false", "true",
  "off",   "on",
  "no",    "yes",
  "n",     "y",
  "f",     "t",
  "none",  "true",  // is there a reasonable counterpart to "none"?
  "invalid"
};
```

Listing 231: Some content of `Switch.C`

[43]The function could also be called with three argmuents, then the default value of the third argument would be overridden and the fourth argument would have its default value.

Listing ?? shows an example of how the Switch datatype can be used in the code. This example reads from the transportProperties dictionary. If no valid entry named testSwitch is present, then the value of the switch is set to false. Notice the second argument of the method lookupOrDefault(), it reads Switch(false). This means, that a new object of the type Switch is created with the boolean value false being passed to the constructor of the class Switch. This new object of type Switch is then used – if necessary – as default value for the switch named testSwitch.

```
Switch testSwitch(transportProperties.lookupOrDefault<Switch>("testSwitch", Switch(false)));
```

Listing 232: Usage example of the Switch datatype

34.4.2 The label datatype

In nearly every program there is sometimes the need for a counter. When examining the solution algorithms, like in Section 19.2, counters can be found. OpenFOAM uses a datatype called label for such counters, e.g. see Listing 149.

The most obvious datatype for a counter would be the integer datatype. Listing 233 contains some lines of the file label.H, where this datatype is defined. Depending on system or compilation parameters, label is of the type int, long or long long[44].

Listing 233 shows the definition of label in case int is used as the underlying datatype.

```
namespace Foam
{
  typedef int label;

  static const label labelMin = INT_MIN;
  static const label labelMax = INT_MAX;

  inline label readLabel(Istream& is)
  {
    return readInt(is);
  }

} // End namespace Foam
```

Listing 233: Some content of label.H

34.4.3 The tmp<> datatype

There is a special class for all temporary data. Because there is no memory management in C++ the programmer has to delete unused variables. The author assumes that the tmp class for all kinds of temporary data is meant to distinguish temporary variables from other variables.

The tmp class uses a technique called generic programming.

34.4.4 The IOobject datatype

The class IOobject handles the behaviour of all kinds of data structures. Although, there are no variables of the type IOobject, understanding some parts of this class will help to understand certain aspects of OpenFOAM.

Listings 234 and 235 show some examples from the sources of the solver *twoPhaseEulerFoam*. There, the class IOobject is used in the creation of fields as well as the creation of dictionary objects.

In Listing 234 two volScalarField variables are created. The constructor of the class volScalarField receives two arguments. In both cases the first argument is an IOobject.

Let us read the arguments of the IOobject constructor call. The first argument is the name of the IOobject. The two last arguments are the read and write flags.

In the case of the fields alpha1 and alpha2 the read and write flags are different. The field alpha1 is read at the start of the application. The write flag causes the field alpha1 to be written to disk, whenever the data

[44]In C as well as in C++ the domain of long is greater or equal than the domain of int. long long was defined in the C99 standard of C and was later introduced to the C++11 standard. The domain of long long is again larger or equal than the domain of long. The type long long uses at least 64 bit. So it is on 64 bit systems the largest possible datatype. The datatype long can use – depending on the compiler – 32 or 64 bit. The type long long guarantees the use of 64 bit.

is written. The field alpha2 on the contrary is not written to disk and the application also does not try to read it.

The name of the IOobject is also the name which the application uses as file name. Therefore the field alpha1 will be written to disk in a file named alpha1. Also when the application tries to read alpha1, it tries to read from the file alpha1.

```
volScalarField alpha1
(
    IOobject
    (
        "alpha1",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

volScalarField alpha2
(
    IOobject
    (
        "alpha2",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    scalar(1) - alpha1
);
```

Listing 234: Definition of volume fraction fields in createFields.H

Listing 235 shows the definition of an IOdictionary. The constructor of the class IOdictionary receives also an IOobject as argument. Again, the name of the IOobject is also the name of the file the application tries to read when reading in the dictionary. Notice also the read flag. This flag causes the application to check if the file has been modified during run-time. If this is the case, the file will be read again.

```
1  IOdictionary ppProperties
2  (
3      IOobject
4      (
5          "ppProperties",
6          runTime.constant(),
7          mesh,
8          IOobject::MUST_READ_IF_MODIFIED,
9          IOobject::NO_WRITE
10     )
11 );
```

Listing 235: Definition of a dictionary in readPPProperties.H

Read & write flags

In the constructor so called read and write flags are provided as arguments, see e.g. Lines 8 and 9 of Listing 235.

Listing 236 shows the available read/write flags. The flag MUST_READ_IF_MODIFIED was introduced with OpenFOAM-2.0.0[45]. The available read flags offer quite some flexibility.

```
        //- Enumeration defining the valid states of an IOobject
        enum objectState
        {
```

[45]http://www.openfoam.org/version2.0.0/runtime-control.php

```
        GOOD,
        BAD
    };

    //- Enumeration defining the read options
    enum readOption
    {
        MUST_READ,
        MUST_READ_IF_MODIFIED,
        READ_IF_PRESENT,
        NO_READ
    };

    //- Enumeration defining the write options
    enum writeOption
    {
        AUTO_WRITE = 0,
        NO_WRITE = 1
    };
```

Listing 236: Definition of the object states and read/write flags of `IOobject` in `IOobject.H`

34.5 Time management

34.5.1 Time stepping

Transient solvers solve the governing equations each time step at least once. Depending on the solution algorithm there are several inner iterations (iterations within a time step) during one outer iteration.

pimpleFoam

Listing 237 shows the beginning of the main loop of *pimpleFoam*. After the three `include` instructions, the `runTime` object is incremented. This means, the current time step is incremented to the next time step.

```
/* code removed for the sake of brevity */

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

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

  runTime++;

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

  /* code continues */
```

Listing 237: The beginning of the main loop of *pimpleFoam* in `pimpleFoam.C`

pisoFoam

Listing 238 shows the beginning of the main loop of *pisoFoam*.

```
/* code removed for the sake of brevity */

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
    {
      /* code continues */
```

Listing 238: The beginning of the main loop of *pisoFoam* in `pisoFoam.C`

There, there is no incrementation of any `runTime` object. The explanation for this, lies in the condition of the `while` statement. In *pisoFoam*, the `while` statement is controlled by the return value of the function call `runTime.loop()`. Whereas, in pimpleFoam, the `while` statement is controlled by the return value of the function call `runTime.run()`.

Let's have a closer look on `runTime.loop()`. Listing 239 shows, that the function `loop()` calls the function `run()` and then increments the `runTime` object by calling `operator++()`.

The ++ operator of the `Time` class

Listing 240 shows the first lines of the definition of the ++ operator of the `Time` class. The last instruction of Listing 240 set the time value to the current time value plus the time step.

```
bool Foam::Time::loop()
{
    bool running = run();

    if (running)
    {
      operator++();
    }

    return running;
}
```

Listing 239: The definition of the function `loop()` in `Time.C`

```
Foam::Time& Foam::Time::operator++()
{
    deltaT0_ = deltaTSave_;
    deltaTSave_ = deltaT_;

    // Save old time name
    const word oldTimeName = dimensionedScalar::name();

    setTime(value() + deltaT_, timeIndex_ + 1);

    /* code removed for the sake of brevity */
```

Listing 240: The definition of the operator ++ in `Time.C`

34.5.2 Setting the new time step

Transient simulations can be run with fixed and variable time steps. In a simulation with fixed time step the time step is constant. The value of the time step must be set before the simulation is started. The time step influences the accuracy and stability of the simulation. The value of the time step determines the time scales that can be resolved in the simulation. Via the Courant-Friedrichs-Lewy (CFL) criterion the time step is linked to the stability of the time integration method.

Most transient OpenFOAM solvers offer the possibility of transient simulations with variable time steps. The user then provides the limits for the determination of the time steps. The most obvious limit is the maximum time step `maxDeltaT`. This is the upper limit for the value of each new time step. This is the parameter for the user to determine the time scale to be resolved.

The second limit for determining the time steps is the maximum Courant number. This parameters purpose is to maintain stability of the numerical solution.

Listing **??** shows the code that reads the time controls. The first instruction reads the entry in `controlDict` specifying whether to use variable time steps or not. This code is rather self-explanatory. If there is not entry in `controlDict` then a fixed time step is used. The other two instructions read values for the maximum Courant number and the maximum time step. The default value for the maximum Courant number is 1.0, which is the limit for the explicit Euler time integration method.

```
const bool adjustTimeStep =
    runTime.controlDict().lookupOrDefault("adjustTimeStep", false);
scalar maxCo =
    runTime.controlDict().lookupOrDefault<scalar>("maxCo", 1.0);
scalar maxDeltaT =
    runTime.controlDict().lookupOrDefault<scalar>("maxDeltaT", GREAT);
```

Listing 241: The content of the file `readTimeControls.H`

Determining the new time step

The value of the new time step has to obey both limit mentioned above, the maximum time step and the maximum Courant number. In order to prevent oscillations the increase of the time step is damped. Listing **??** shows how the time step is computed each time step.

```
if (adjustTimeStep)
{
    scalar maxDeltaTFact = maxCo/(CoNum + SMALL);
    scalar deltaTFact = min(min(maxDeltaTFact, 1.0 + 0.1*maxDeltaTFact), 1.2);

    runTime.setDeltaT
    (
        min
        (
            deltaTFact*runTime.deltaTValue(),
            maxDeltaT
        )
    );

    Info<< "deltaT = " <<  runTime.deltaTValue() << endl;
}
```

Listing 242: The content of the file `setDeltaT.H`

Let us have a look on what the code is actually doing.

$$\texttt{maxDeltaTFact} = \frac{\texttt{maxCo}}{\texttt{Co} + \texttt{SMALL}} \tag{48}$$

$$\texttt{deltaTFact} = \min(\ \min(\ \texttt{maxDeltaTFact},\ 1.0 + 0.1 * \texttt{maxDeltaTFact}),\ 1.2) \tag{49}$$

The scalar `maxDeltaTFact` (line **??** in Listing 242 and Eq. (48)) is the relation between the maximum Courant number and the current Courant number (see Section 34.5.3 on how the Courant number is determined). The role of the constant `SMALL` is to prevent division by zero, which would cause the solver to crash.

The scalar `deltaTFact` is computed from `maxDeltaTFact`. This line of code (line **??** and Eq. (49)) implements the damping, i.e. the rate of increase of the time step is limited. The nested use of two `min()` functions determines the minimum of three values. The most obvious of these three values is the last argument. If this value is the smallest, then the next time step is 20 % larger than the last one.

Eq. (49) shows the minimum of the first two arguments in a mathematical way. Figure 34 shows the three arguments of Eq. (49). We use the symbol x for the scalar `maxDeltaTFact`. In Figure 34 the values for x are greater than one. Eq. (51) elaborates why this is the case. x is the ratio of the maximum Courant number Co_{max} and the current Courant number Co. As the current Courant number is always smaller than the maximum Courant number we replace Co with fCo_{max}, with $f < 1$. After cancelling Co_{max} the inverse of f remains. Thus x is always greater than one.

$$\min(x,\ 1+0.1x) = \begin{cases} x & x < \frac{10}{9} \\ 1+0.1x & x > \frac{10}{9} \end{cases} \tag{50}$$

$$x = \frac{Co_{max}}{Co} = \frac{Co_{max}}{\underbrace{f}_{<1} Co_{max}} = \frac{1}{f} \tag{51}$$

$$\Rightarrow x > 1 \tag{52}$$

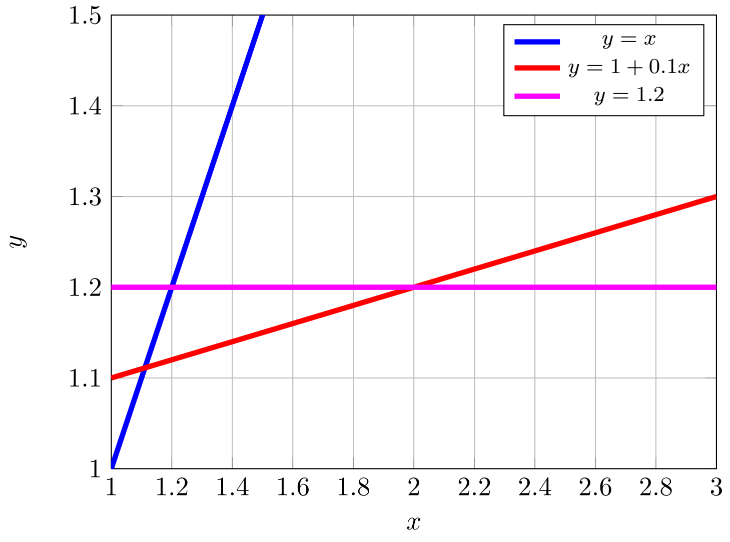


Figure 34: The three arguments of Eq. (49) plotted over x

The argument of the function `setDeltaT()` contains the abidance of the first limit, the maximum time step. There the minimum of the newly calculated and the maximum time step is passed on.

34.5.3 The Courant number

The Courant number Co is the ratio of the time step Δt and the characteristic convection time scale $^{u}/_{\Delta x}$. Eq. (53) shows the definition of the Courant number. However in a practical CFD code the Courant number will be computed in a slightly different way. Eq. (54) shows how Eq. (53) is expanded with $^{A}/_{A}$ to gain a formulation featuring the flux and the volume of the control volume instead of the velocity and the discretisation length. Eq. (55) shows the extension of Eq. (54) for a one-dimensional finite volume formulation. The mean of the fluxes of the faces E and W defines the convective time scale. This definition seems obvious in some way in the one-dimensional case. For two or three-dimensional cases the choice of how to define the characteristic flux seems not straight forward.

$$Co = \frac{u \Delta t}{\Delta x} \tag{53}$$

$$Co = \frac{u \Delta t}{\Delta x} = \frac{u \Delta t}{\Delta x} \frac{A}{A} = \frac{\phi \Delta t}{\Delta V} \tag{54}$$

$$Co = \frac{\frac{|\phi_E| - |\phi_W|}{2} \Delta t}{\Delta V} = \frac{1}{2} \frac{(|\phi_E| - |\phi_W|) \Delta t}{\Delta V} \tag{55}$$

The Courant number in OpenFOAM

In OpenFOAM the Courant number is computed for all cells. In fact OpenFOAM computes a maximum Courant number, i.e. the largest Courant number of all cells, and a mean Courant number, i.e. the mean Courant number of all cells.

Listing 243 shows the code responsible for computing the Courant number. Line 8 of Listing 243 translates to Eq. (56). `sumPhi` is a scalar field containing the sum of the magnitudes of all face fluxes of every cell, i.e. for each cell the magnitude of the face fluxes are summed up. Eq. (56) holds for every cell.

Eq. (57) is the mathematical representation of line 11. There the maximum value of the ratio between the values of `sumPhi` and the cell volume is determined. Both variables `sumPhi` and `mesh.V()` contain values for every cell. Therefore the `gMax()` function returns the maximum value.

Eq. (58) represents line 14.

```
 1  scalar CoNum = 0.0;
 2  scalar meanCoNum = 0.0;
 3
 4  if (mesh.nInternalFaces())
 5  {
 6      scalarField sumPhi
 7      (
 8          fvc::surfaceSum(mag(phi))().internalField()
 9      );
10
11      CoNum = 0.5*gMax(sumPhi/mesh.V().field())*runTime.deltaTValue();
12
13      meanCoNum =
14          0.5*(gSum(sumPhi)/gSum(mesh.V().field()))*runTime.deltaTValue();
15  }
16
17  Info<< "Courant Number mean: " << meanCoNum
18      << " max: " << CoNum << endl;
```

Listing 243: The content of the file `CourantNo.H`

$$\texttt{sumPhi} = \sum_{f_i} |\phi_{f_i}| \tag{56}$$

$$\texttt{CoNum} = \frac{1}{2} \max_{\text{all cells}} \left(\frac{\texttt{sumPhi}}{V_{cell}} \right) \Delta t \tag{57}$$

$$\texttt{meanCoNum} = \frac{1}{2} \frac{\sum \texttt{sumPhi}}{\sum V_{cell}} \Delta t \tag{58}$$

Discussion

The way to compute the Courant number in a three dimensional case is not straight forward as mentioned above. This section reflects the authors way of understanding. So there is no guarantee of validity. The factor of $1/2$ and the summation of ϕ_{f_i} is explained by the author as follows.

We base our reflections on a two dimensional control volume. Eq. (60) shows the summation written in the long form. This equation is then rearranged to yield Eq. (61). In Eq. (61) the summation is reduced to two terms. These terms are the arithmetic mean of the face flux in the principal directions $N - S$ and $W - E$. This summation is then identified as the L_1 norm of the mean face fluxes in the principal directions.

The reason for choosing the L_1 norm is not self-evident. In any case is the L_1 norm computationally cheaper than the Euklidian or L_2 norm. However, the use of the L_1 norm seems justified since it measures the distance covered by a movement, see `http://en.wikipedia.org/wiki/Taxicab_geometry`.

$$Co = \frac{1}{2} \frac{\sum_{f_i} |\phi_{f_i}|}{V_{cell}} \Delta t \tag{59}$$

$$Co = \frac{1}{2} \frac{|\phi_N| + |\phi_E| + |\phi_S| + |\phi_W|}{V_{cell}} \Delta t \tag{60}$$

$$Co = \frac{\frac{|\phi_N| + |\phi_S|}{2} + \frac{|\phi_E| + |\phi_W|}{2}}{V_{cell}} \Delta t \tag{61}$$

$$Co = \frac{\overline{|\phi|}^{NS} + \overline{|\phi|}^{WE}}{V_{cell}} \Delta t \tag{62}$$

$$Co = \frac{\| \overline{|\phi|}^{\mathbf{x}_i} \|_1}{V_{cell}} \Delta t \tag{63}$$

We indroduce the following symbols

$$\frac{1}{2}\sum_{f_i}|\phi_{f_i}| = \| \overline{|\phi|}^{\mathbf{x}_i} \|_1 = \|\Phi\|_1 \tag{64}$$

$$Co = \frac{\|\Phi\|_1}{V_{cell}}\Delta t \tag{65}$$

The way the mean Courant number is computed seems incorrect at the first glance but it isn't.

$$Co = \frac{\|\Phi\|_1}{V_{cell}}\Delta t \tag{65}$$

The mean value of the quantity x is defined as follows

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{66}$$

Next we write the mean value of the Courant number. An unmarked summation is a summation over all cells.

$$\overline{Co} = \frac{1}{N}\sum\left(\frac{\|\Phi\|_1}{V_{cell}}\right)\Delta t \tag{67}$$

$$\overline{Co} = \frac{1}{N}\underbrace{\frac{\sum V_{cell}}{\sum V_{cell}}}_{=1}\underbrace{\frac{\sum\|\Phi\|_1}{\sum\|\Phi\|_1}}_{=1}\sum\left(\frac{\|\Phi\|_1}{V_{cell}}\right)\Delta t \tag{68}$$

$$\overline{Co} = \frac{\sum\|\Phi\|_1}{\sum V_{cell}}\underbrace{\frac{1}{N}\frac{\sum V_{cell}}{\sum\|\Phi\|_1}\sum\left(\frac{\|\Phi\|_1}{V_{cell}}\right)}_{X}\Delta t \tag{69}$$

Eq. (69) now resembles Eq. (58). Now we concentrate on the term X which is the only difference between Eqns. (69) and (58).

$$X = \frac{1}{N}\frac{\sum V_{cell}}{\sum\|\Phi\|_1}\sum\left(\frac{\|\Phi\|_1}{V_{cell}}\right) \tag{70}$$

$$X = \underbrace{\frac{\sum V_{cell}}{N}}_{=\overline{V_{cell}}}\frac{1}{\sum\|\Phi\|_1}\sum\left(\frac{\|\Phi\|_1}{V_{cell}}\right) \tag{71}$$

$$X = \frac{\overline{V_{cell}}}{\sum\|\Phi\|_1}\sum\left(\frac{\|\Phi\|_1}{V_{cell}}\right) \tag{72}$$

$$X = \frac{1}{\sum\|\Phi\|_1}\sum\left(\frac{\|\Phi\|_1}{\frac{V_{cell}}{\overline{V_{cell}}}}\right) \tag{73}$$

We assume $\frac{V_{cell}}{\overline{V_{cell}}} \approx 1$

$$X = \frac{1}{\sum\|\Phi\|_1}\sum\left(\frac{\|\Phi\|_1}{1}\right) \tag{74}$$

$$X = \frac{\sum\|\Phi\|_1}{\sum\|\Phi\|_1} = 1 \tag{75}$$

Thus we have shown that the way the mean Courant number `meanCoNum` is computed is actually the mean Courant number $\overline{Co}$. However, this attempt of a proof is based on some assumptions.

First, the way the author explains the meaning of the summation of the face fluxes relies on hexahedral cells. The argument made seems not to be applicable on tetrahedral cells. Secondly, the assumption $\frac{V_{cell}}{\overline{V_{cell}}} \approx 1$ is valid for homogeneous grids. For a uniform grid this assumption would be ideally fulfilled. If the volume of the largest and smallest cells differs a lot this assumption is not justified.

Some thoughts on the computational costs

Why the formula for the mean Courant number is rearranged from

$$\overline{Co} = \frac{1}{N} \sum \left(\frac{\|\Phi\|_1}{V_{cell}} \right) \Delta t \tag{76}$$

to

$$\overline{Co} = \frac{\sum \|\Phi\|_1}{\sum V_{cell}} \Delta t \tag{77}$$

is unknown to the author.

It is the opinion of the author that this is made for reasons of computational cost. Two times the summation over all values of a field plus one division is computationally cheaper than an elementwise division of two fields and one subsequent summation over all elements of the resulting field.

This would be the case if the division operation takes more time than the summation operation which is very likely the case. Depending on the system the floating point division operation can take several times longer than a floating point multiplication.

In the first case n times one division and one addition needs to be made, with n the number of field values. In the second case $2n$ times additions and one division is to be made.

$$T_1 = n(T_d + T_s) \qquad\qquad T_2 = 2nT_s + T_d \tag{78}$$

We introduce the factor δ, that is the ratio between T_d and T_s.

$$T_1 = n(\delta T_s + T_s) \qquad\qquad T_2 = 2nT_s + \delta T_s \tag{79}$$

$$T_1 = nT_s(1 + \delta) \qquad\qquad T_2 = T_s(2n + \delta) \tag{80}$$

$$\frac{T_1}{T_s} = n(1 + \delta) \qquad\qquad \frac{T_2}{T_s} = (2n + \delta) \tag{81}$$

Next we assume that n is very large

$$\frac{T_1}{T_s} = n(1 + \delta) \qquad\qquad \frac{T_2}{T_s} \approx 2n \tag{82}$$

So the first formula takes $1 + \delta$ operations, whereas the second formula takes approximately $2n$ operations. If δ is larger than one, the second formula will take less time for computation. A δ smaller than one is highly unlikelyor even impossible as the addition is a very simple operation. Remember, δ is the ratio between the time a division takes and the time an addition takes. The actual ratio vary according to the system architecture, the compiler and the implementation, e.g. [2] reports a factor of 5 to 6 for single and double precision floating point division. This argument does not consider the memory usage of the operations involved, it only focuses on the number of floating point operations.

Because the Courant number is computed after every time step the time needed to calculate the Courant number has an impact on the simulation time.

34.5.4 The two-phase Courant number

In a two-phase simulation there are several choices of how to compute the Courant number. In total, there are 4 velocity fields (`U1`, `U2`, `U` and `Ur`). These are the velocities of the phases 1 and 2 as well as the mixture and relative velocities. The solver *twoPhaseEulerFoam* computes the Courant number for the mixture and the relative velocities.

Listing 244 shows the content of the file `CourantNos.H` which is part of the source code of this solver. Line 1 computes the mixture Courant number by including the file `CourantNo.H`. This is the file described in Section 34.5.3. As this code operates on the field `phi`, which happens to be the flux of the mixture, the mixture Courant number is computed.

The next lines compute the Courant number based on the relative phase flux. At line 11 the maximum of this two Courant numbers is determined and stored into the variable `CoNum`.

`CoNum` is the Courant number used by the time stepping mechanism. So the variable time steps of the *twoPhaseEulerFoam* solver are based on the maximum of the mixture and relative velocity Courant number.

```
#include "CourantNo.H"

{
    scalar UrCoNum = 0.5*gMax
    (
        fvc::surfaceSum(mag(phi1 - phi2))().internalField()/mesh.V().field()
    )*runTime.deltaTValue();

    Info<< "Max Ur Courant Number = " << UrCoNum << endl;

    CoNum = max(CoNum, UrCoNum);
}
```

Listing 244: The content of the file `CourantNos.H`

34.6 Turbulence models

In Section 16.1 it is stated that the user can choose between three options.

1. A laminar simulation
2. Using a RAS turbulence model
3. Using a LES turbulence model

This statement is reflected in the relationship between the classes implementing the turbulence models in OpenFOAM. Object oriented programming allowes the programmer to translate relationships directly from human language to source code. Two statements can be made about turbulence models

1. All RAS turbulence models are turbulence models, but not all turbulence models are RAS turbulence models.
2. A RAS turbulence model is not the same as an LES turbulence model, however, both are turbulence models.

Both statements are reflected by the class diagram of the turbulence models. On the top is the abstract class `turbulenceModel`. This abstract class, provides the framework for all derived turbulence classes. Also, all functionality common to all possible turbulence classes can be defined in this class. All derived classes will then inherit this functionality.

Each turbulence model is derived from this abstract base class. Each turbulence class will implement specific functionality individually.

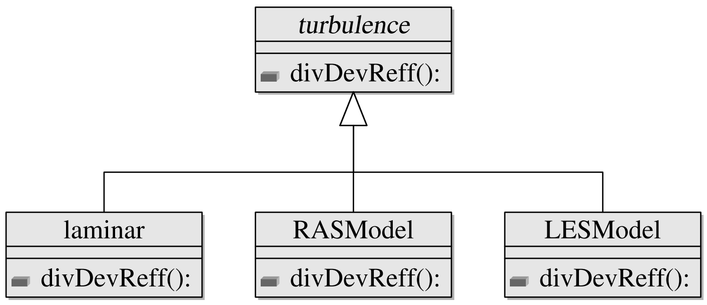


Figure 35: Vererbungsdiagramm der Turbulenz-Klassen

34.6.1 The abstract base class `turbulenceModel`

The base class `turbulenceModel` is an abstract class[46]. It contains several pure-virtual functions. To be able to call this functions, these functions must be overridden by the classes that are derived from the base class.

[46]A class that contains one or more abstract methods is called an abstract class. If a class contains only abstract methods, then it is sometimes called a pure-abstract class.

A pure-virtual class can not be called. Listing 245 shows the declaration of pure-virtual or abstract methods. The `= 0` indicates that a method is abstract.

```
//- Return the turbulence viscosity
virtual tmp<volScalarField> nut() const = 0;

//- Return the effective viscosity
virtual tmp<volScalarField> nuEff() const = 0;
```

Listing 245: Declaration of the virtual methods in `turbulenceModel.H`

The base class contains not only virtual functions. It also contains functions that are the same for all derived classes. Consequently, this functions are implemented by the base class. Listing 246 shows the implementation of the function `nu()`. This function is used to access the laminar or molecular viscosity. The laminar viscosity is a property of the fluid itself and has nothing to do with turbulence. However, the turbulence models need to access the laminar viscosity.

```
//- Return the laminar viscosity
inline tmp<volScalarField> nu() const
{
  return transportModel_.nu();
}
```

Listing 246: Implementation of `nu()` in `turbulenceModel.H`

Every class derived from an abstract class must at least override the abstract methods. The non-abstract methods of the base class – like `nu()` from Listing 246 – can be used by the derived classes. No matter if a RAS or a LES turbulence model is used, the laminar viscosity will always be the same.

34.6.2 The class `RASModel`

The class `RASModel` is derived from the abstract class `turbulenceModel`. The class `RASModel` itself is the base class for all RAS turbulence models. It is also an abstract class because it does not override all abstract methods inherited from `turbulenceModel`.

However, the class `RASModel` implements all methods that are common to all RAS turbulence models. Listing 247 shows the implementation of the method `nuEff()` in the class `RASModel`.

```
//- Return the effective viscosity
virtual tmp<volScalarField> nuEff() const
{
  return tmp<volScalarField>
  (
    new volScalarField("nuEff", nut() + nu())
  );
}
```

Listing 247: Implementation of `nuEff()` in `RASModel.H`

The effective viscosity `nuEff` is calculated from the laminar viscosity, which is a property of the fluid, and the turbulent viscosity. The turbulent viscosity is a property of the turbulence model. The function `nu()` in Listing 247 is implemented in the class `turbulenceModel`, see Listing 246. The function `nut()` is not implemented by the class `RASModel`. Therefore, this method must be implemented by the classes derived from `RASModel`.

34.6.3 RAS turbulence models

All RAS turbulence models are derived from the class `RASModel`. Each derived class must implement all remaining abstract methods. Figure 36 shows a simplified class diagram – there is a number of RAS turbulence models available in OpenFOAM.

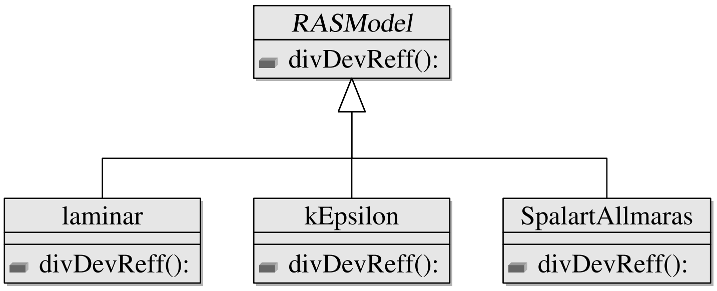


Figure 36: Inheritance of RAS turbulence models

34.6.4 The class kEpsilon

The class kEpsilon is derived from RASModel.

```
class kEpsilon
:
  public RASModel
{
  /* class definition */
}
```

Listing 248: Class definition of kEpsilon in kEpsilon.H

The function nut() has to be implemented by kEpsilon. Listing 249 shows how the function nut() is implemented. This function simply returns the class member nut_.

```
//- Return the turbulence viscosity
virtual tmp<volScalarField> nut() const
{
  return nut_;
}
```

Listing 249: Implementation of nut() in kEpsilon.H

The way how nut_ is calculated differs between the RAS turbulence models. See Listing 267 in Section 39.2.2.

35 General remarks on solver modifications

This section collects and documents solver modifications of the author.

35.1 Preparatory tasks

In order to be able to distinguish between the standard solvers and the solvers created by the user, a new directory has to be created. We follow the scheme of the standard solvers, of which the source code resides in OpenFOAM-2.1.x/applications/solvers. Therefore, we need to create some folders to place our sources in user-2.1.x/applications/solvers. Listing 250 lists the necessary commands. Open a Terminal and type the commands of the Listing to do the job.

```
cd $FOAM_INST_DIR
cd user-2.1.x
mkdir applications
cd applications
mkdir solvers
```

Listing 250: Create some directories

35.2 The next steps

When modifying a solver, there are some further steps necessary. These are described in Section 36.1. Although these steps are for a specific example, they represent the general steps that are necessary. In short these steps are

- copy the sources you want to base your new solver on
- make necessary adjustments to ensure that
  - the new solver compiles at all
  - the new solver does not corrupt existing solvers

Based on this steps the user can start to modify the sources in order to accomplish the intended function or feature.

36 *twoPhaseLESEulerFoam*

The solver *twoPhaseEulerFoam* can only use the k-ϵ turbulence model. The aim of this section is to document the necessary modifications to create a version of the *twoPhaseEulerFoam* solver that is capable to use the LES turbulence model. This new solver is called – like this section – *twoPhaseLESEulerFoam*.

36.1 Preparatory tasks

36.1.1 Copy the sources

As the new solver shall be a modification of the existing *twoPhaseEulerFoam* solver, we need to copy the `OpenFOAM-2.1.x/applications/solvers/multiphase/twoPhaseEulerFoam` folder to `user-2.1.x/applications/solvers.`
To do this via the Terminal

```
cd $FOAM_INST_DIR
cp -r OpenFOAM-2.1.x/applications/solvers/multiphase/twoPhaseEulerFoam user-2.1.x/applications
    /solvers/twoPhaseLESEulerFoam
```

Listing 251: Copy the sources

36.1.2 Rename files

Next, some files have to be renamed. This may not be mandatory in order to successfully compile the solver. However, for the sake of tidiness, all files containing the name of the solver have to be renamed. In this case, there are two files. The source of the solver itself `twoPhaseEulerFoam.C` and `readTwoPhaseEulerFoamControls.H`.

The names of these two files change to `twoPhaseLESEulerFoam.C` and `readTwoPhaseLESEulerFoamControls.H`. The latter of these files is included via an `#include` statement into the former one. Therefore, we need to change the according statement in `twoPhaseLESEulerFoam.C`. Listing 252 shows the affected lines of code. The first line is the old statement, which is commented. This line can also be deleted. However, the old statement is left in order to show the original statement. The second line is the modified statement.

```
/* #include "readTwoPhaseEulerFoamControls.H" */
#include "readTwoPhaseLESEulerFoamControls.H"
```

Listing 252: Change the include statement

36.1.3 Adjust `Make/files`

In order not to corrupt the existing solver the file `Make/files` has to be adapted. Listing 253 shows how the content has to look like. This file contains a list of *.C files that define the solver. In most cases there is only one such file, e.g. `twoPhaseEulerFoam.C`. The entry beginning with `EXE` defines the full path to the executable. The locations where changes have to be made are marked red in the Listing. These changes are:

- The name of the source file, `twoPhaseLESEulerFoam.C` instead of `twoPhaseEulerFoam.C`.
- The path to the executeable, `FOAM_USER_APPBIN` instead of `FOAM_APPBIN`.
- The name of the executeable[47], `twoPhaseLESEulerFoam` instead of `twoPhaseEulerFoam`.

```
twoPhaseLESEulerFoam.C

EXE = $(FOAM_USER_APPBIN)/twoPhaseLESEulerFoam
```

Listing 253: Content of *Make/files*

The reason for all these changes lies in the compilation process. A new solver is compiled by simply typing `wmake` in the Terminal. *wmake* reads from `Make/files` which file to compile and where to put the created executable.

36.1.4 The file `Make/options`

At this stage, there is no need to alter this file. The explanation of this file fits best at this location.

The file `Make/options` contains all compiler flags and parameters. Such parameters are, e.g.

- additional directories where included header files are located; the first group in Listing 254.
- libraries which have to be linked[48] to the executable of the solver; the second group of entries.

For the sake of completeness, Listing 254 shows the content of the file `Make/options`. This file is read by *wmake* to determine some parameters for the compiler. As you can see in Listing 255, the compiler is called with a lot more options. However, all the options listed in `Make/options` are related to the specific solver, e.g. which libraries the solver uses. Other options, e.g. the target platform, or the warning level, are elsewhere defined.

```
EXE_INC = \
    -I../bubbleFoam \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude \
    -IturbulenceModel \
    -IkineticTheoryModels/lnInclude \
    -IinterfacialModels/lnInclude \
    -IphaseModel/lnInclude \
    -Iaveraging

EXE_LIBS = \
    -lEulerianInterfacialModels \
    -lfiniteVolume \
    -lmeshTools \
    -lincompressibleTransportModels \
    -lphaseModel \
    -lkineticTheoryModel
```

Listing 254: Content of *Make/options*

36.2 Preliminary observations

First of all we have to bear in mind, that *twoPhaseEulerFoam* is based on the solver *bubbleFoam*. This fact becomes important now. At this stage, the sources of *twoPhaseEulerFoam* have been copied to a `user-2.1.x/applications/sol`
Then all necessary adjustment have been made to prepare compilation.

[47]The executeable does not necessarily have to have the same name as the source file. However, different names can lead to confusion and make code maintenance harder. Therefore, it is strongly recommended to use consistent names, i.e. to name the source file `SOLVER.C` and the executable `SOLVER`.

[48]Compilation of C or C++ programs is usually done in two steps. First all files are compiled and then the object files generated by the compiler are linked together to form the executable.

Compilation

Now, if we try to compile our new solver *twoPhaseLESEulerFoam*, – which is acutally just a copy of *twoPhaseEulerFoam*, because there are no real modifications yet – then compilation fails.

```
+ wmake
Making dependency list for source file twoPhaseLESEulerFoam.C
could not open file createRASTurbulence.H for source file twoPhaseLESEulerFoam.C
could not open file wallFunctions.H for source file twoPhaseLESEulerFoam.C
could not open file wallDissipation.H for source file twoPhaseLESEulerFoam.C
could not open file wallViscosity.H for source file twoPhaseLESEulerFoam.C
SOURCE=twoPhaseLESEulerFoam.C ; g++ -m64 -Dlinux64 -DWM_DP -Wall -Wextra -Wno-unused-
    parameter -Wold-style-cast -Wnon-virtual-dtor -O3 -DNoRepository -ftemplate-depth-100 -I
    ../bubbleFoam -I/home/user/OpenFOAM/OpenFOAM-2.1.x/src/finiteVolume/lnInclude -I/home/user
    /OpenFOAM/OpenFOAM-2.1.x/src/transportModels/incompressible/lnInclude -IturbulenceModel -
    IkineticTheoryModels/lnInclude -IinterfacialModels/lnInclude -IphaseModel/lnInclude -
    Iaveraging -IlnInclude -I. -I/home/user/OpenFOAM/OpenFOAM-2.1.x/src/OpenFOAM/lnInclude -I/
    home/user/OpenFOAM/OpenFOAM-2.1.x/src/OSspecific/POSIX/lnInclude   -fPIC -c $SOURCE -o
    Make/linux64GccDPOpt/twoPhaseLESEulerFoam.o
In file included from twoPhaseLESEulerFoam.C:60:0:
createFields.H:139:37: schwerwiegender Fehler: createRASTurbulence.H: Datei oder Verzeichnis
    nicht gefunden Kompilierung beendet.
make: *** [Make/linux64GccDPOpt/twoPhaseLESEulerFoam.o] Fehler 1
```

Listing 255: Compilation error message

The error message says, that the file `createRASTurbulence.H` and other could not be found. In the `user-2.1.x/applications/solvers/twoPhaseLESEulerFoam` directory, there are no such files. However, these files are included in `createFields.C` which is included in `twoPhaseLESEulerFoam.C`.

The reason

The solution to this mystery lies in the first statement of this section. The *twoPhaseEulerFoam* solver is based on *bubbleFoam*. If we have a look on the source directory of *bubbleFoam* (Listing 256) we find all files that are missing when compiling *twoPhaseLESEulerFoam*.

```
user@host:~/OpenFOAM/OpenFOAM-2.1.x/applications/solvers/multiphase/bubbleFoam$ ls
alphaEqn.H     bubbleFoam.dep   createPhi1.H   createRASTurbulence.H   kEpsilon.H
Make    readBubbleFoamControls.H   wallDissipation.H   wallViscosity.H bubbleFoam.C
createFields.H   createPhi2.H   DDtU.H   liftDragCoeffs.H   pEqn.H   UEqns.H
wallFunctions.H     write.H
user@host:~/OpenFOAM/OpenFOAM-2.1.x/applications/solvers/multiphase/bubbleFoam$
```

Listing 256: The source files of *bubbleFoam*

Now, there are source files of another solver included in *twoPhaseEulerFoam*. However, other than the standard solver *twoPhaseEulerFoam* our solver fails to compile. The explanation is this string of characters `-I../bubbleFoam`. This can be found in Listing 255 as a parameter in the call of *g++*. *g++* is the C++ compiler of the GNU compiler collection. *g++* is on Linux systems usually the standard C++ compiler. The `-I` flag tells the compiler where to find header files. In this case `../bubbleFoam` is specified.

This path is valid for the standard solver of OpenFOAM. However, in our case, there is no folder called `bubbleFoam` in the `user2.1.x/applications/solvers` directory. In the case of *twoPhaseLESEulerFoam*, `../bubbleFoam` refers to `user2.1.x/applications/solvers/bubbleFoam` which does not exist.

The solution

In a first attempt to ensure that our new solver compiles we can copy the missing files from the the sources of *bubbleFoam* to the sources of *twoPhaseLESEulerFoam*. We now can delete the line containing `-I../bubbleFoam` in `Make/options`, because the included files are now located in the same directory as `twoPhaseLESEulerFoam.C`. The directory of the main source file – of `twoPhaseLESEulerFoam.C` – is the a default location, where the compiler looks for included files.

The files from *bubbleFoam* all deal with the k-ϵ turbulence model. In our case – we want to include the LES turbulence model – we do not need this files. However, if we wanted to use the k-ϵ turbulence model, then copying the missing file from the sources of *bubbleFoam* would be the proper thing to do. Listing 257 shows the necessary commands for the Terminal. Notice the use of the wildcard `*`, this substitutes for zero or more characters. Therefore, the first `cp` command copies the files `wallDissipation.H`, `wallViscosity.H` and `wallFunctions.h` to the sources of our new solver. The second `cp` command copies the file `createRASTurbulence.H`. In this case the wildcard is used to save typing effort.

```
cd $FOAM_INST_DIR
cp OpenFOAM-2.1.x/applications/solvers/multiphase/bubbleFoam/wall* user-2.1.x/applications/
    solvers/twoPhaseLESEulerFoam/
cp OpenFOAM-2.1.x/applications/solvers/multiphase/bubbleFoam/createRAS* user-2.1.x/
    applications/solvers/twoPhaseLESEulerFoam/
```

Listing 257: Copy the missing file from the sources of *bubbleFoam*

36.3 How LES in OpenFOAM is used

If we want to integrate LES turbulence models into our solver, we should first have a look at other solvers. Looking at the source code of a solver that supports LES models out of the box, will provide us with some hints. Now, we have a look at the source code of *pimpleFoam*. *pimpleFoam* is a solver for an incompressible fluid. Because *twoPhaseEulerFoam* is a solver two incompressible fluids which also uses the PIMPLE algorithm, comparing *twoPhaseEulerFoam* with *pimpleFoam* is not a bad idea.

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

Listing 258: Including turbulence: *pimpleFoam*

The second and the third line are required for using a generic turbulence model. The header file `singlePhaseTransportMod` provides a transport model and the file `turbulenceModel.H` provides all definitons of the generic turbulence model.

36.4 Integrate LES

36.4.1 Include required models

In order to make use of the LES turbulence model we need to include the header file `singlePhaseTransportModel.H` because the turbulence models of OpenFOAM make use of the transport model. Instead of the file `turbulenceModel.H` we will include the file `LESModel.H`. This file defines the base class for all LES turbulence models.

Listing 259 shows the first group of `include` statements of the file `twoPhaseLESEulerFoam`. The last two lines include the transport model and the LES model.

```
#include "fvCFD.H"
#include "MULES.H"
#include "subCycle.H"
#include "nearWallDist.H"
#include "wallFvPatch.H"
#include "fixedValueFvsPatchFields.H"
#include "Switch.H"

#include "IFstream.H"
#include "OFstream.H"

#include "dragModel.H"
#include "phaseModel.H"
#include "kineticTheoryModel.H"

#include "pimpleControl.H"
#include "MRFZones.H"

// for using LES
```

```
#include "singlePhaseTransportModel.H"
#include "LESModel.H"
```

Listing 259: The first group of `include` statements in `twoPhaseLESEulerFoam`

36.4.2 Replace the k-ϵ model

In the file `twoPhaseLESEulerFoam` we need to replace the statement that includes the file `kEpsilon.H`. This file contains the k-ϵ turbulence model. Since we want to use the LES models provided by OpenFOAM we simply copied from other solver, see Listing 147.

In line 24 of Listing 261 we write a similar instruction like in e.g. *pimpleFoam*. However, the variable `sgsModel` is of type `LESModel`, whereas in the source code of solvers that use generic turbulence modelling this line would read `turbulence->correct()`.

In line 25 we update the field `nuEff2`, which is the effective viscosity of the continuous phase. This instruction is necessary because *twoPhaseEulerFoam* uses a distinct field for the effective viscosity. Other solvers access this quantity via their turbulence model. To keep the number of changes in the source code low, we stick to the original code of *twoPhaseEulerFoam* as far as it is feasible.

```
1   // --- Pressure-velocity PIMPLE corrector loop
2   while (pimple.loop())
3   {
4     #include "alphaEqn.H"
5     #include "liftDragCoeffs.H"
6     #include "UEqns.H"
7
8     // --- Pressure corrector loop
9     while (pimple.correct())
10    {
11      #include "pEqn.H"
12
13      if (correctAlpha && !pimple.finalIter())
14      {
15        #include "alphaEqn.H"
16      }
17    }
18
19    #include "DDtU.H"
20
21    if (pimple.turbCorr())
22    {
23      //#include "kEpsilon.H"
24      sgsModel->correct();
25      nuEff2 = sgsModel->nuEff();
26    }
27  }
```

Listing 260: The main loop in `twoPhaseLESEulerFoam`

36.4.3 Create a LES model

Now, we need to modify the file `createFields.H`. First we need to comment or delete the `include` statement including the file `createRASTurbulence.H`. Then, we need to create a transport model and a LES model.

Finally, we need to copy the instructions that create the fields `nuEff1` and `nuEff2` from the file `createRASTurbulence.H` into the file `createFields.H`. The question of how to model turbulence in two-phase flows is completely answered. So, this is just one possibility. See Section 39.3 for a discussion about turbulence in two-phase solvers.

```
/* lots of code */

//#include "createRASTurbulence.H"

/* even more code */

// new for LES
singlePhaseTransportModel fluid(U2, phi2);
```

```

autoPtr<incompressible::LESModel> sgsModel
(
  incompressible::LESModel::New(U2, phi2, fluid)
);


// new from createRASTurbulence.H
Info<< "Calculating field nuEff1\n" << endl;
volScalarField nuEff1
(
  IOobject
  (
    "nuEff1",
    runTime.timeName(),
    mesh,
    IOobject::NO_READ,
    IOobject::NO_WRITE
  ),
  sgsModel->nut() + nu1
  // nuEff1 will be overwritten at the end of the file
);

Info<< "Calculating field nuEff2\n" << endl;
volScalarField nuEff2
(
  IOobject
  (
    "nuEff2",
    runTime.timeName(),
    mesh,
    IOobject::NO_READ,
    IOobject::NO_WRITE
  ),
  sgsModel->nut() + nu2
);

// set nuEff1 according to Jakobsen 1997
nuEff1 = rho1*nuEff2/rho2;
```

Listing 261: The main loop in `twoPhaseLESEulerFoam`

36.4.4 Make ready for compiling

In order to be ready to compile the new solver, we need to adjust some more files. Listing 262 shows the necessary modifications of the file `Make/options`. These adjustments are necessary in order to enable the compiler to find all included files.

```
EXE_INC = \
  -I$(LIB_SRC)/finiteVolume/lnInclude \
  -I$(LIB_SRC)/transportModels \
  -I$(LIB_SRC)/transportModels/incompressible/lnInclude \
  -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
  -IkineticTheoryModels/lnInclude \
  -IinterfacialModels/lnInclude \
  -IphaseModel/lnInclude \
  -I$(LIB_SRC)/turbulenceModels \
  -I$(LIB_SRC)/turbulenceModels/incompressible/LES/LESModel \
  -I$(LIB_SRC)/turbulenceModels/LES/LESdeltas/lnInclude \
  -Iaveraging

EXE_LIBS = \
  -lEulerianInterfacialModels \
  -lfiniteVolume \
  -lmeshTools \
  -lincompressibleTransportModels \
  -lphaseModel \
  -lkineticTheoryModel \
  -lincompressibleLESModels
```

Listing 262: Content of *Make/options*

36.5 Compile

The solver can be compiled by invoking `wmake`.

Part X
Theory

This section covers more detailled topics and tries to look *under the hood* of OpenFOAM from a non-programming view.

37 Discretization

37.1 Temporal discretization

37.2 Spatial discretization

The purpose of spatial discretization schemes is to compute the face values of fields whose values are stored at the cell centre. The face values are then used e.g. for computing the spatial derivatives.

37.2.1 `upwind` scheme

An upwind scheme determines the face value of a quantity simply by choosing the cell centered value of the cell that is located upwind of the face in question.

37.2.2 `linearUpwind` scheme

The `linearUpwind` scheme is equivalent to FLUENTs *Second-Order Upwind Scheme*.

37.2.3 QUICK scheme

The FLUENT Theory Guide [1] states:

> For quadrilateral and hexahedral meshes, where unique upstream and downstream faces and cells can be identified, ANSYS FLUENT also provides the QUICK scheme for computing a higher-order value of the convected variable at a face.

37.2.4 MUSCL scheme

38 Momentum diffusion in an incompressible fluid

38.1 Governing equations

In Section 19.1 we discussed the governing equations of a solver for incompressible fluids.

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \underbrace{\nabla \cdot \left(\mathrm{dev}(\mathbf{R}^{eff})\right)}_{=\mathrm{div}(\mathrm{dev}(\mathbf{R}^{eff}))} = -\nabla p + \mathbf{Q} \tag{35}$$

$$\mathbf{R}^{eff} = -\nu^{eff}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right) \tag{29}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \nabla \cdot \left(\mathrm{dev}(-\nu^{eff}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right))\right) = -\nabla p + \mathbf{Q} \tag{36}$$

The momentum diffusion term is handled by the turbulence model.

$$\underbrace{\nabla \cdot \left(\mathrm{dev}(\mathbf{R}^{eff})\right)}_{=\mathrm{div}(\mathrm{dev}(\mathbf{R}^{eff}))} \quad \Leftrightarrow \quad \texttt{turbulence->divDevReff(U)}$$

38.2 Implementation

All turbulence model of OpenFOAM are based on a generic turbulence model class. Figure 35 in Section 34.6 shows a class diagram. There, it is shown, that all RAS turbulence model classes as well as all LES turbulence model classes are derived from the same base class. A lot of solvers of OpenFOAM allow the user to choose between laminar simulation as well as RAS or LES turbulence modelling. Therefore, by the time of writting the source code, nobody could have known, which turbulence exactly will handle the momentum diffusion term.

To overcome such problems, modern programming languages support a technique called polymorphism. In the source code the instruction `turbulence->divDevReff(U)` is called to compute the diffusive term. This instruction means, that the method `divDevReff()` of the object `turbulence` is called.

```
// Solve the Momentum equation

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

UEqn().relax();

sources.constrain(UEqn());

volScalarField rAU(1.0/UEqn().A());

if (pimple.momentumPredictor())
{
  solve(UEqn() == -fvc::grad(p) + sources(U));
}
```

Listing 263: The file *UEqn.H* of *pimpleFoam*

The source code of the file `createFields.H` tells us, that the object `turbulence` is of the data type `turbulenceModel`.

```
singlePhaseTransportModel laminarTransport(U, phi);

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

Listing 264: The file *createFields.H* of *pimpleFoam*

By the time of compilation, it is guaranteed that the object `turbulence` is of the data type `turbulenceModel`. However, `turbulence` will never actually be of the data type `turbulenceModel`. It will be of a data type derived from `turbulenceModel`. The decision which exact method `divDevReff()` has to be called, will be made at run-time based on the actual type of `turbulence`.

Listing 265 shows the declaration of the virtual method `divDevReff()`. See Section 34.6 for a discussion on virtual methods. Listing 266 shows how this method is actually implemented by the standard k-ϵ turbulence models of OpenFOAM.

```
//- Return the source term for the momentum equation
virtual tmp<fvVectorMatrix> divDevReff(volVectorField& U) const = 0;
```

Listing 265: Declaration of the virtual Method *divDevReff* in *turbulenceModel.H*

```
tmp<fvVectorMatrix> kEpsilon::divDevReff(volVectorField& U) const
{
  return
  (
    - fvm::laplacian(nuEff(), U)
    - fvc::div(nuEff()*dev(T(fvc::grad(U))))
  );
}
```

Listing 266: Implementation of the virtual Method *divDevReff* in *kEpsilon.H*

The calculation of `divDevReff()` is equivalent to Eq. (36).

$$\begin{aligned}\texttt{divDevReff} &= \nabla\cdot\left(\mathrm{dev}(-\nu\left(\nabla\mathbf{U}+(\nabla\mathbf{U})^T\right))\right)\\ &= \underbrace{-\nabla\cdot\left(\nu(\nabla\mathbf{U})\right)}_{\texttt{laplacian(nu,U)}} - \underbrace{\nabla\cdot\left(\nu(\nabla\mathbf{U})^T\right)}_{\texttt{div(nu*dev(T(grad(U))))}}\end{aligned}$$

The momentum diffusion term is most probably split into two parts for numerical reasons.

39 The incompressible k-ϵ turbulence model

39.1 The k-ϵ turbulence model in literature

The governing equations for the k-ϵ model for a single phase are taken from Wilcox [19].

Eddy viscosity

$$\mu_T = \rho C_\mu \frac{k^2}{\epsilon} \tag{83}$$

Turbulent kinetic energy

$$\rho\frac{\partial k}{\partial t} + \rho U_j\frac{\partial k}{\partial x_j} = \tau_{ij}\frac{\partial U_i}{\partial x_j} - \rho\epsilon + \frac{\partial}{\partial x_j}\left[(\mu + \frac{\mu_T}{\sigma_k})\frac{\partial k}{\partial x_j}\right] \tag{84}$$

Dissipation Rate

$$\rho\frac{\partial \epsilon}{\partial t} + \rho U_j\frac{\partial \epsilon}{\partial x_j} = C_{\epsilon 1}\frac{\epsilon}{k}\tau_{ij}\frac{\partial U_i}{\partial x_j} - C_{\epsilon 2}\rho\frac{\epsilon^2}{k} + \frac{\partial}{\partial x_j}\left[(\mu + \frac{\mu_T}{\sigma_\epsilon})\frac{\partial \epsilon}{\partial x_j}\right] \tag{85}$$

Closure coefficients

$$C_{\epsilon 1} = 1.44, \qquad C_{\epsilon 2} = 1.92, \qquad C_\mu = 0.09, \qquad \sigma_k = 1.0, \qquad \sigma_\epsilon = 1.3 \tag{86}$$

The transport equations for k and ϵ are reorganized to follow the basic structure

local derivative + convection + diffusion = source & sink terms

Turbulent kinetic energy

$$\rho\frac{\partial k}{\partial t} + \rho U_j\frac{\partial k}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\underbrace{(\mu + \frac{\mu_T}{\sigma_k})}_{D_k}\frac{\partial k}{\partial x_j}\right] = \underbrace{\tau_{ij}\frac{\partial U_i}{\partial x_j}}_{G} - \rho\epsilon \tag{87}$$

Dissipation Rate

$$\rho\frac{\partial \epsilon}{\partial t} + \rho U_j\frac{\partial \epsilon}{\partial x_j} - \frac{\partial}{\partial x_j}\left[\underbrace{(\mu + \frac{\mu_T}{\sigma_\epsilon})}_{D_\epsilon}\frac{\partial \epsilon}{\partial x_j}\right] = C_{\epsilon 1}\frac{\epsilon}{k}\underbrace{\tau_{ij}\frac{\partial U_i}{\partial x_j}}_{G} - C_{\epsilon 2}\rho\frac{\epsilon^2}{k} \tag{88}$$

Diffusivity constants

$$D_k = \mu + \frac{\mu_T}{\sigma_k} \tag{89}$$

$$D_\epsilon = \mu + \frac{\mu_T}{\sigma_\epsilon} \tag{90}$$

The constant expressions in the diffusive terms are combined into the diffusivity constants D_k and D_ϵ. The first term on the right hand side of the turbulent kinetic energy equation is the production of turbulent kinetic energy G.

39.2 The k-ϵ turbulence model in OpenFOAM

39.2.1 Governing equations

The governing equations of the k-ϵ model of OpenFOAM are basically the same equations as in Section 39.1. The vector notation is used in this section because the syntax OpenFOAM uses strongly resembles the vector notation. However, there are some modifications to the equations.

First, the transport equations for k and ϵ are divided by the density ρ. Therefore, all terms containing viscosity contain the kinematic viscosity ν instead of the dynamic viscosity μ.

Secondly, the standard k-ϵ model of OpenFOAM has eliminated the model constant σ_k. Since the value of this constant is one, this constant has been elimininated. This does not change the behaviour of the model. However, if the user tries to change this model constant, nothing actually happens. See Section 16.2.2 for a discussion and an example.

Finally, the convection term is converted into two term by the product rule of differentiation. See Eqn. (92).

Eddy viscosity, see Listing 267

$$\mu_T = \rho\,\nu_T$$

$$\nu_T = C_\mu \frac{k^2}{\epsilon} \tag{91}$$

Turbulent kinetic energy, see Listing 268

$$U_j \frac{\partial k}{\partial x_j} = \mathbf{U} \cdot \frac{\partial k}{\partial \mathbf{x}} = \mathbf{U} \cdot \nabla k$$

$$\mathbf{U} \cdot \frac{\partial k}{\partial \mathbf{x}} = \nabla \cdot (\mathbf{U}k) - (\nabla \cdot \mathbf{U})k \tag{92}$$

$$\frac{\partial k}{\partial t} + \nabla \cdot (\mathbf{U}k) - (\nabla \cdot \mathbf{U})k - \nabla \cdot (D_k \nabla k) = G - \epsilon \tag{93}$$

Dissipation Rate

$$\frac{\partial \epsilon}{\partial t} + \nabla \cdot (\mathbf{U}\epsilon) - (\nabla \cdot \mathbf{U})\epsilon - \nabla \cdot (D_\epsilon \nabla \epsilon) = C_1 G \frac{1}{k} - C_2 \frac{\epsilon^2}{k} \tag{94}$$

Diffusivity constants - Note that σ_k has been eliminated from the equations

$$D_k = \texttt{DkEff} = \nu + \nu_T \tag{95}$$

$$D_\epsilon = \texttt{DepsilonEff} = \nu + \frac{\nu_T}{\sigma_\epsilon} \tag{96}$$

Closure coefficients - default values

$$C_1 = 1.44, \qquad C_2 = 1.92, \qquad C_\mu = 0.09, \qquad \sigma_\epsilon = 1.3 \tag{97}$$

The default values of the model constants can be found in the constructor of the respective turbulence model class.

39.2.2 The source code

Listing 267 shows the calculation of the eddy viscosity. A (too) short glimpse on the code may lead to confusion, as the function `sqr()` meaning taking a variable to the power of two looks similar to `sqrt()`, which is the square root.

Listing 268 shows the transport equation for the turbulent viscosity. The last term on the right hand side is expanded.

$$\epsilon = \underbrace{\frac{\epsilon}{k}\, k}_{\texttt{fvm::Sp(epsilon/k, k)}} \tag{98}$$

```
nut_ = Cmu_*sqr(k_)/epsilon_;
```

Listing 267: Calculation of the eddy viscosity

```
tmp<fvScalarMatrix> kEqn
(
  fvm::ddt(k_)
  + fvm::div(phi_, k_)
  - fvm::Sp(fvc::div(phi_), k_)
  - fvm::laplacian(DkEff(), k_)
==
  G
  - fvm::Sp(epsilon_/k_, k_)
);
```

Listing 268: Transport equation for the turbulent kinetic energy

Constructor

Listing 269 shows the first lines of the constructor of the `kEpsilon` class. The constructor receives five arguments. After the colon (in line 9), the initialisation list follows. This list contains also the default values of the model constants. See Section 33.5 for details about constructors in C++. In line 18 the default value of the model constant C_μ is defined.

```
1  kEpsilon::kEpsilon
2  (
3    const volVectorField& U,
4    const surfaceScalarField& phi,
5    transportModel& transport,
6    const word& turbulenceModelName,
7    const word& modelName
8  )
9  :
10   RASModel(modelName, U, phi, transport, turbulenceModelName),
11
12   Cmu_
13   (
14     dimensioned<scalar>::lookupOrAddToDict
15     (
16       "Cmu",
17       coeffDict_,
18       0.09
19     )
20   ),
21   /* code continues */
```

Listing 269: The constructor of the `kEpsilon` class

39.3 The k-ϵ turbulence model in *bubbleFoam* and *twoPhaseEulerFoam*

The k-ϵ turbulence model is hardcoded in *bubbleFoam* and *twoPhaseEulerFoam*. This means, that these solvers do not use the generic turbulence modelling other than most OpenFOAM solvers.

The question of turbulence modelling in dispersed two-phase flows is not fully answered yet. There are several strategies:

Per phase The turbulence is modelled for both phases individually.

Mixture The turbulence is modelled based on mixture quantities.

Liquid phase Turbulence is modelled based in the quantites of the liquid phase. The turbulence of the dispersed phase is either neglected or considered by a model constant.

39.3.1 Governing equations

The k-ϵ turbulence model of *bubbleFoam* and *twoPhaseEulerFoam* is in some aspects different than the standard k-ϵ turbulence model of OpenFOAM.

1. The diffusivity constants are calculated from the effective viscosity. Compare Eqns. (89, 90) and (104, 105)
2. The model constants σ_k and σ_ϵ are replaced by their reciprocal values.
3. Other than in the standard k-ϵ model, the model constant σ_k is not dropped. By defining a value for the constant $\alpha_{1,k} = {}^1/_{\sigma_k}$, a value for σ_k is assigned.

Turbulence modelling in *bubbleFoam* and *twoPhaseEulerFoam* is based on the liquid quantities. Turbulence of the gas phase is considered by the use of the model constant C_t. This constant connects the turbulent viscosity of the liquid and the gas phase. By setting this constant to zero, turbulence is ignored in the gas phase.

Eddy viscosity

$$\nu_{2,T} = C_\mu \frac{k^2}{\epsilon} \tag{99}$$

$$\nu_{2,eff} = \nu_2 + \nu_{2,T} \tag{100}$$

$$\nu_{1,eff} = \nu_1 + C_t^2 \nu_{2,T} \tag{101}$$

Turbulent kinetic energy, see Listing 268

$$\frac{\partial k}{\partial t} + \nabla \cdot (\mathbf{U}_2 k) - (\nabla \cdot \mathbf{U}_2) k - \nabla \cdot (\alpha_{1,k} \nu_{2,eff} \nabla k) = G - \epsilon \tag{102}$$

Dissipation Rate

$$\frac{\partial \epsilon}{\partial t} + \nabla \cdot (\mathbf{U}_2 \epsilon) - (\nabla \cdot \mathbf{U}_2) \epsilon - \nabla \cdot (\alpha_{1,\epsilon} \nu_{2,eff} \nabla \epsilon) = C_1 G \frac{1}{k} - C_2 \frac{\epsilon^2}{k} \tag{103}$$

Diffusivity constants - Note the different definition

$$\alpha_{1,k} = \frac{1}{\sigma_k}$$

$$\alpha_{1,\epsilon} = \frac{1}{\sigma_\epsilon}$$

$$D_k = \alpha_{1,k} \nu_{2,eff} = \frac{\nu_{2,eff}}{\sigma_k} \tag{104}$$

$$D_\epsilon = \alpha_{1,\epsilon} \nu_{2,eff} = \frac{\nu_{2,eff}}{\sigma_\epsilon} \tag{105}$$

Closure coefficients - default values

$$C_1 = 1.44, \qquad C_2 = 1.92, \qquad C_\mu = 0.09, \qquad \alpha_{1,k} = 1, \qquad \alpha_{1,\epsilon} = 0.76923 \tag{106}$$

39.3.2 Source code

The transport equations of *bubbleFoam* and *twoPhaseEulerFoam* reside in the file `kEpsilon.H`. Listing 270 shows the most important lines of `kEpsilon.H`.

```
tmp<volTensorField> tgradU2 = fvc::grad(U2);
volScalarField G(2*nut2*(tgradU2() && dev(symm(tgradU2()))));

// Dissipation equation
fvScalarMatrix epsEqn
(
    fvm::ddt(epsilon)
  + fvm::div(phi2, epsilon)
  - fvm::Sp(fvc::div(phi2), epsilon)
10      - fvm::laplacian
11        (
12        alpha1Eps*nuEff2, epsilon,
13        "laplacian(DepsilonEff,epsilon)"
14        )
15      ==
16        C1*G*epsilon/k
17      - fvm::Sp(C2*epsilon/k, epsilon)
18  );
19
20  // Turbulent kinetic energy equation
21  fvScalarMatrix kEqn
22  (
23        fvm::ddt(k)
24      + fvm::div(phi2, k)
25      - fvm::Sp(fvc::div(phi2), k)
26      - fvm::laplacian
27      (
28        alpha1k*nuEff2, k,
29        "laplacian(DkEff,k)"
30      )
31      ==
32        G
33      - fvm::Sp(epsilon/k, k)
34  );
35
36  //- Re-calculate turbulence viscosity
37  nut2 = Cmu*sqr(k)/epsilon;
```

Listing 270: The turbulent transport equations of the `bubbleFoam` and `twoPhaseEulerFoam` solver

39.4 Modelling the production of turbulent kinetic energy

When comparing the turbulent equations From literature and the sources, the definition of the production of turbulent kinetic energy shows great differences.

39.4.1 Definitions from literature and source files

The production of turbulent kinetic energy seems to be differently defined.

Thesis of H. Rusche [15] - the basis of *bubbleFoam* and *twoPhaseEulerFoam*

$$P_b = 2\nu_{2,eff} \left(\nabla \mathbf{U}_b \cdot \operatorname{dev} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) \right) \tag{107}$$

Source code - `kEpsilon.H` of *bubbleFoam* - See Line 2 Listing 270

$$G = 2\nu_T \left(\nabla \mathbf{U}_2 : \operatorname{dev}(\operatorname{sym}(\nabla \mathbf{U}_2)) \right) \tag{108}$$

Source code - standard k-ϵ model, `kEpsilon.C`

$$G = 2\nu_T |\operatorname{sym}(\nabla \mathbf{U})|^2 \tag{109}$$

Ferzinger Peric [9]

$$P = \mu_T \nabla \mathbf{U} : \left(\nabla \mathbf{U} + (\nabla \mathbf{U})^T \right) \tag{110}$$

Wilcox [19]

$$G = \mu_T \nabla \mathbf{U} : \left(\nabla \mathbf{U} + (\nabla \mathbf{U})^T \right) - \frac{2}{3} \rho k \mathbf{I} : \nabla \mathbf{U} \tag{111}$$

Some definitions use the dynamic viscosity and some others use the kinematic viscosity. For incompressible fluids, this is no major difference between the definitions.

39.4.2 Different use of viscosity

Eq. (107) is the only definition that makes use of the [15] effective viscosity instead of the turbulent viscosity. The reason for this is not explained.

However, the FLUENT Theory Guide [1] states that the effective viscosity is used to calculate the production term when high-Reynolds number versions of the k-ϵ model are used. It is not further specified what is meant with high-Reynolds number versions of the k-ϵ model.

39.4.3 Notation

The definitions in Section 39.4.1 are written in vector notation. However, there seems to be a minor flaw in Eq. (107). There

$$P_b = 2\nu_{2,eff} \left(\nabla \mathbf{U}_b \cdot \operatorname{dev} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) \right) \tag{107}$$

The dot can not denote an inner product. The result only has the correct dimension, if the dot denotes a contraction. Therefore, the equation should read

$$P_b = 2\nu_{2,eff} \left(\nabla \mathbf{U}_b : \operatorname{dev} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) \right) \tag{112}$$

39.4.4 Definitions from literature

The definition of the production term in Eq. (110) and (111) differ only in the last term.

$$G = \mu_T \nabla \mathbf{U} : \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) - \frac{2}{3} \rho k \mathbf{I} : \nabla \mathbf{U} \tag{111}$$

Using the following identities, the contraction can be replaced by an inner product

$$\mathbf{I} : \nabla \mathbf{U} = \operatorname{tr}(\nabla \mathbf{U}) = \nabla \cdot \mathbf{U} \tag{113}$$

For incompressible fluids the divergence of the velocity must be zero due to the continuity equation

$$\nabla \cdot \mathbf{U} = 0 \tag{114}$$

$$G = \mu_T \nabla \mathbf{U} : \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) - \underbrace{\frac{2}{3} \rho k \mathbf{I} : \nabla \mathbf{U}}_{=0} \tag{115}$$

Therefore, Eqns. (110) and (111) are identical if the fluid is incompressible. We now can examine the differences of the definitions of the production term, using Eq. (110) as reference equation.

39.4.5 Definitions of Rusche and *bubbleFoam*

The solvers *bubbleFoam* and *twoPhaseEulerFoam* are based on the thesis of H. Rusche [15]. However, the production term is defined differently. Compare Eq. (107) and (108).

$$P_b = 2\nu_{2,eff} \left(\nabla \mathbf{U}_b : \operatorname{dev} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) \right) \tag{107}$$

$$G = 2\nu_T \left(\nabla \mathbf{U}_2 : \operatorname{dev}(\operatorname{sym}(\nabla \mathbf{U}_2)) \right) \tag{108}$$

We ignore the different symbols for the velocity of the continuous phase

$$\mathbf{U}_2 = \mathbf{U}_b \tag{116}$$

The second operator of the contraction is different in both equations. We ask, if the following equation holds

$$\nabla \mathbf{U}_2 : \operatorname{dev}(\operatorname{sym}(\nabla \mathbf{U}_2)) \overset{?}{=} \nabla \mathbf{U}_b : \operatorname{dev} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) \tag{117}$$

With the following identities the question is easily answered

$$\mathrm{dev}(\mathbf{T}) = \mathbf{T} - \frac{1}{3}\,\mathrm{tr}(\mathbf{T}) \tag{118}$$

$$\mathrm{sym}(\mathbf{T}) = \frac{1}{2}\left(\mathbf{T} + (\mathbf{T})^T\right) \tag{119}$$

$$\mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U}_2)\right) = \mathrm{dev}\left(\frac{1}{2}\left(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T\right)\right) \tag{120}$$

$$\mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U}_2)\right) = \frac{1}{2}\,\mathrm{dev}\left(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T\right) \tag{121}$$

$$\mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U}_2)\right) = \frac{1}{2}\underbrace{\left((\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T) - \frac{1}{3}\,\mathrm{tr}(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T)\right)}_{=\mathrm{dev}(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T)} \tag{122}$$

$$\mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U}_2)\right) = \frac{1}{2}\,\mathrm{dev}\left(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T\right) \tag{123}$$

This leads to the answer

$$\nabla\mathbf{U}_2 : \mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U}_2)\right) = \frac{1}{2}\nabla\mathbf{U}_b : \mathrm{dev}\left(\nabla\mathbf{U_b} + (\nabla\mathbf{U_b})^T\right) \tag{124}$$

The definition of the production term in the source code differs in two ways from the definition in the source code

1. The use of different viscosities, see Eqns. (107) and (108).
2. A factor of 2, compare Eqns. (117) and (124)

The reason for this differences is not clear. H. Rusche refers to an article which is not available to the author.

39.4.6 Definitions of Ferzinger and *bubbleFoam*

We now compare the definitions of Ferzinger and *bubbleFoam*. The definition of Ferzinger is – like the equations in most other book about turbulence – for single-phase systems. However, *bubbleFoam* is a two-phase solver. The question of considering turbulence in two-phase systems is not answered yet. *bubbleFoam* considers turbulence for the continuous phase by the use of a turbulence model. The turbulence of the disperse phase is linked to the continuous phase. Therefore, turbulence model equations of *bubbleFoam* are quite similar to single-phase turbulence equations.

$$G = 2\nu_T\left(\nabla\mathbf{U}_2 : \mathrm{dev}(\mathrm{sym}(\nabla\mathbf{U}_2))\right) \tag{108}$$

$$P = \mu_T\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) \tag{110}$$

We ignore the different viscosities and ask ourselves

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) \overset{?}{=} 2\left(\nabla\mathbf{U}_2 : \mathrm{dev}(\mathrm{sym}(\nabla\mathbf{U}_2))\right) \tag{125}$$

Inserting Eq. (123) gives

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = 2(\nabla\mathbf{U}_2 : \underbrace{\mathrm{dev}(\mathrm{sym}(\nabla\mathbf{U}_2)))}_{=\frac{1}{2}\mathrm{dev}(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T)} \tag{126}$$

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = \nabla\mathbf{U}_2 : \mathrm{dev}\left(\nabla\mathbf{U_2} + (\nabla\mathbf{U_2})^T\right) \tag{127}$$

Now we insert Eq. (118) into the *rhs* of Eq. (127)

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = \nabla\mathbf{U} : \left(\mathrm{dev}(\nabla\mathbf{U} + (\nabla\mathbf{U})^T) + \frac{1}{3}\,\mathrm{tr}(\nabla\mathbf{U} + (\nabla\mathbf{U})^T)\right) \tag{128}$$

Using the following identities and Eq. (113)

$$\mathrm{tr}(\mathbf{A}+\mathbf{B}) = \mathrm{tr}(\mathbf{A}) + \mathrm{tr}(\mathbf{B}) \tag{129}$$

$$\mathrm{tr}(\mathbf{A}^\mathbf{T}) = \mathrm{tr}(\mathbf{A}) \tag{130}$$

$$\mathbf{I} : \nabla\mathbf{U} = \mathrm{tr}(\nabla\mathbf{U}) = \nabla\cdot\mathbf{U} \tag{113}$$

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = \nabla\mathbf{U} : \left(\mathrm{dev}(\nabla\mathbf{U} + (\nabla\mathbf{U})^T) + \frac{2}{3}(\nabla\cdot\mathbf{U})\right) \tag{131}$$

The second term of the *rhs* vanishes according to the continuity equation for an incompressible fluid

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = \nabla\mathbf{U} : \left(\mathrm{dev}(\nabla\mathbf{U} + (\nabla\mathbf{U})^T) + \frac{2}{3}\underbrace{(\nabla\cdot\mathbf{U})}_{\nabla\cdot\mathbf{U}=0}\right) \tag{132}$$

Eq. (133) now resembles Eq. (127). Therefore, we proofed that the definition of *bubbleFoam* is equivalent to the definition of Ferzinger

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = \nabla\mathbf{U} : \mathrm{dev}\left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) \tag{133}$$

39.4.7 Definition of standard k-ϵ of OpenFOAM

We now compare the definition of the production term of the standard k-ϵ model implemented in OpenFOAM with the definition found in [9].

Source code - standard k-ϵ model, `kEpsilon.C`

$$G = 2\nu_T|\mathrm{sym}(\nabla\mathbf{U})|^2 \tag{109}$$

Ferzinger Peric [9]

$$P = \nu_T\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) \tag{110}$$

Starting from Eq. (110), we will use Eq. (133) and Eq. (123)

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = \nabla\mathbf{U} : \mathrm{dev}\left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) \tag{133}$$

$$\mathrm{dev}\left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = 2\ \mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U})\right) \tag{123}$$

to gain

$$\nabla\mathbf{U} : \left(\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right) = 2\,\nabla\mathbf{U} : \mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U})\right) \tag{134}$$

We use definition (135) to change Eq. (109)

$$|\mathrm{sym}(\nabla\mathbf{U})|^2 = \mathrm{sym}(\nabla\mathbf{U}) : \mathrm{sym}(\nabla\mathbf{U}) \tag{135}$$

Now we pose the question

$$\mathrm{sym}(\nabla\mathbf{U}) : \mathrm{sym}(\nabla\mathbf{U}) \overset{?}{=} \nabla\mathbf{U} : \mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U})\right) \tag{136}$$

The *lhs* of Eq. (136) corresponds to Eq. (109). The *rhs* of Eq. (136) was derived from Eq. (110). Now, we use some identities

$$\mathrm{dev}(\mathbf{T}) = \mathbf{T} - \frac{1}{3}\,\mathrm{tr}(\mathbf{T}) \tag{118}$$

$$\mathrm{tr}(\mathrm{sym}(\mathbf{T})) = \mathrm{tr}(\mathbf{T}) \tag{137}$$

to reformulate the *rhs* of Eq. (136)

$$\nabla\mathbf{U} : \mathrm{dev}\left(\mathrm{sym}(\nabla\mathbf{U})\right) = \nabla\mathbf{U} : \left(\mathrm{sym}(\nabla\mathbf{U}) - \frac{1}{3}\,\mathrm{tr}(\nabla\mathbf{U})\right) \tag{138}$$

As we now concentrate on incompressible single-phase problems, we can eliminate the second term of the *rhs* of Eq. (138) by the use of Eq. (113)

$$\mathbf{I} : \nabla \mathbf{U} = \mathrm{tr}(\nabla \mathbf{U}) = \nabla \cdot \mathbf{U} = 0 \tag{113}$$

We now have

$$\nabla \mathbf{U} : \mathrm{dev}\left(\mathrm{sym}(\nabla \mathbf{U})\right) = \nabla \mathbf{U} : \mathrm{sym}(\nabla \mathbf{U}) \tag{139}$$

The following equation remains, which is easily proofed by some tensor calculus

$$\mathrm{sym}(\nabla \mathbf{U}) : \mathrm{sym}(\nabla \mathbf{U}) = \nabla \mathbf{U} : \mathrm{sym}(\nabla \mathbf{U}) \tag{140}$$

Every tensor can be decomposed into a symmetric and a skew part

$$\mathbf{T} = sym(\mathbf{T}) + \mathrm{skew}(\mathbf{T}) \tag{141}$$

$$\mathrm{sym}(\mathbf{T}) = \frac{1}{2}\left(\mathbf{T} + \mathbf{T}^T\right) \tag{142}$$

$$\mathrm{skew}(\mathbf{T}) = \frac{1}{2}\left(\mathbf{T} - \mathbf{T}^T\right) \tag{143}$$

Therefore, we can write

$$\mathbf{T} : \mathrm{sym}(\mathbf{T}) = \mathrm{sym}(\mathbf{T}) : \mathrm{sym}(\mathbf{T}) + \mathrm{skew}(\mathbf{T}) : \mathrm{sym}(\mathbf{T}) \tag{144}$$

The following properties of skew tensors let the second contraction vanish

$$\underbrace{\mathrm{skew}(\mathbf{T})}_{a_{ij}} : \underbrace{\mathrm{sym}(\mathbf{T})}_{s_{ij}} \tag{145}$$

$$a_{ii} = 0 \tag{146}$$

$$a_{ij} = -a_{ji} \tag{147}$$

$$\mathrm{skew}(\mathbf{T}) : \mathrm{sym}(\mathbf{T}) = a_{ij} s_{ij} = 0 \tag{148}$$

Finally, we obtain

$$\mathbf{T} : \mathrm{sym}(\mathbf{T}) = \mathrm{sym}(\mathbf{T}) : \mathrm{sym}(\mathbf{T}) \tag{149}$$

Therefore, we proofed that the definition of the standard k-ϵ model is equivalent to the definition of Ferzinger.

40 Some theory behind the scenes of LES

40.1 LES model hierarchy

The large eddy simulation is based on the spatial filtering of the governing equations. Similar to the Reynolds-averaged modelling strategy (filtering with respect to time), the large eddy modelling strategy requires some closure models. In principle, the velocity is decomposed into a grid-scale and a sub-grid scale portion. The grid-scale portion is resolved by the governing equations. The sub-grid scale portion – or the influence of the sub-grid scale portion on the resolved velocity – needs to be modelled.

Similar to the RANS approach, the closure terms appear in the stress terms of the momentum equations. There are several modelling strategies to close the equations. The class hierarchy of the LES models of OpenFOAM reflects the different approaches. Figure 37 shows the first layer of the class hierarchy of the LES models in OpenFOAM. First layer means that a class derived from the abstract class `LESModel` may be an abstract class itself and therefore be the base for other classes[49][50].

The classification according to Figure 37 is not the only possible way to divide all existing LES models into categories.

[49]In a class diagram a class with an italic written name is an abstract class. A class with an upright written name is an actual class.

[50]This shows the great advantage of object oriented programming. The class hierarchy of the code reflects the relation between the objects in reality, e.g. every eddy viscosity model is an LES model, but not every LES model is an eddy viscosity model.

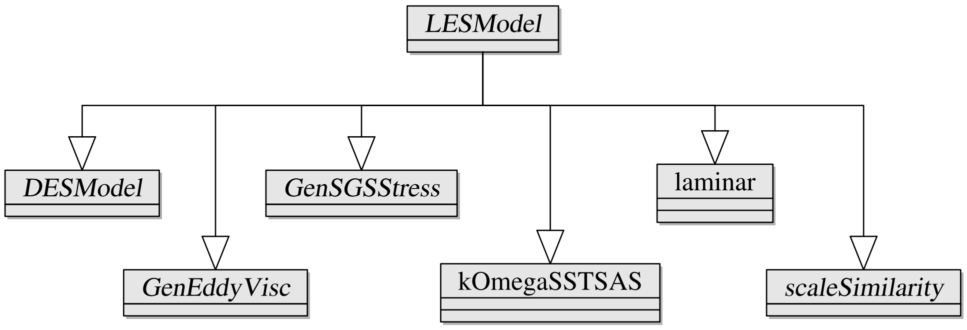


Figure 37: First layer of the class hierarchy of the LES models of OpenFOAM

40.2 Eddy viscosity models

One of the most common approaches of closing the governing equations when using an LES turbulence modelling strategy are eddy viscosity models. Like the RANS turbulence models, the eddy viscosity models make use of the Boussinesq hypothesis. The contribution of the sub-grid scale terms is modelled by an additional viscosity. The effective viscosity is the sum of the laminar viscosity and the sub-grid viscosity.

$$\nu_{eff} = \nu + \nu_{SGS} \tag{150}$$

40.2.1 Class hierarchy

The base class for all eddy viscosity models is `GenEddyVisc`. Figure 38 shows the class hierarchy with focus on `GenEddyVisc`.

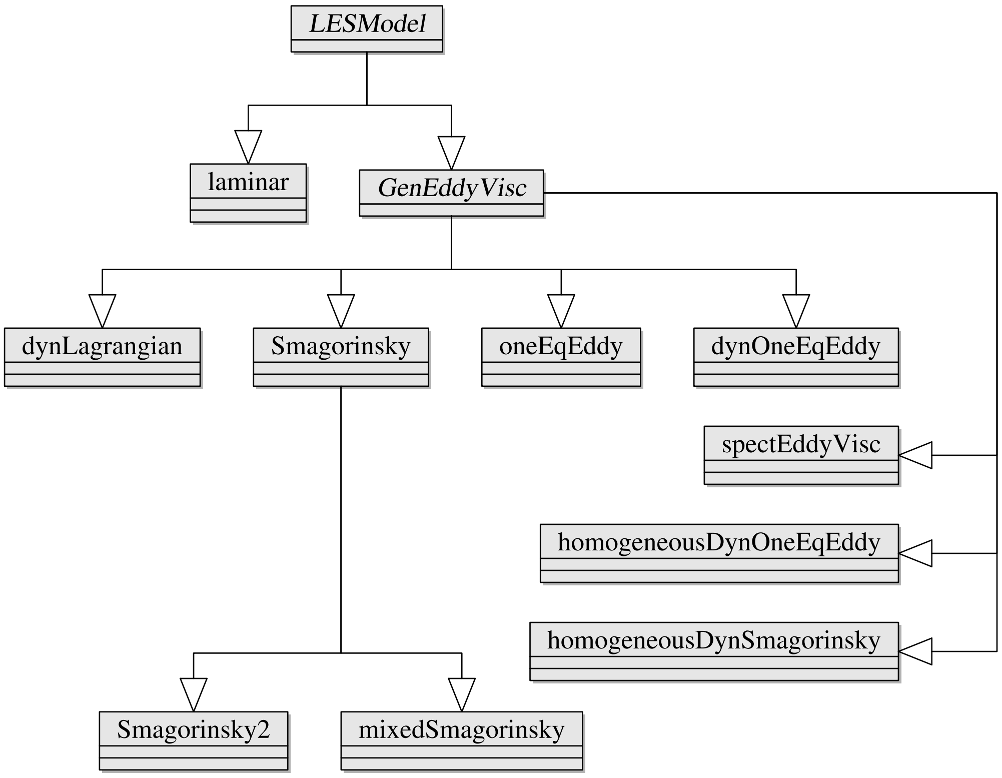


Figure 38: Class hierarchy of the eddy viscosity models in OpenFOAM

40.2.2 Classification

The eddy viscosity models can be divided further based on the way the sub-grid viscosity is computed and the complexity of the model.

	constant coefficient	dynamic coefficient
algebraic model	`Smagorinsky`	`homogeneousDynSmagorinsky`
	`Smagorinsky2`	`spectEddyVisc`
one equation model	`oneEqEddy`	`dynOneEqEddy`
		`homogeneousDynOneEqEddy`
		`dynLagrangian`

Table 5: Comparison of the eddy viscosity models of OpenFOAM

40.2.3 Eddy viscosity

For dimensional reasons, the eddy viscostiy must be a product of a length and a velocity scale [7]. Eq. (152) shows the generic equation for the sub-grid viscosity. An additional model constant is the third term in the product. The way the model constant is computed as well as the choice for the length and velocity scales is determined by the model.

$$[\nu_{SGS}] = \frac{\mathrm{m}^2}{\mathrm{s}} = \frac{\mathrm{m}}{\mathrm{s}} \cdot \mathrm{m} \tag{151}$$

$$\nu_{SGS} = C_{SGS}\, l_{SGS}\, q_{SGS} \tag{152}$$

A choice that is common to a number of eddy viscosity models in OpenFOAM is to choose the filter width as the length scale and the square root of the sub-grid kinetic energy as teh velocity scale. Algebraic models usually calculate the sub-grid kinetic energy from known quantities, e.g. based on the velocity gradient. One equation models typically solve a transport equation for the sub-grid scale kinetic energy.

$$l_{SGS} = \Delta \tag{153}$$

$$[l_{SGS}] = \mathrm{m} \tag{154}$$

$$q_{SGS} = \sqrt{k_{SGS}} \tag{155}$$

$$[q_{SGS}] = \sqrt{\frac{\mathrm{m}^2}{\mathrm{s}^2}} = \frac{\mathrm{m}}{\mathrm{s}} \tag{156}$$

40.2.4 The `Smagorinsky` LES model

The Smagorinsky eddy viscosity is one of the simplest LES models. From Table 5 we see that this is an algebraic model with a constant model coefficient. This model was published 1963 [16].

Eq. (157) shows the definition of the sub-grid scale viscosity according to the Samgorinsky model as it can be found in literature [7].

$$\nu_{SGS} = (C_S \Delta)^2 |\mathbf{S}| \tag{157}$$

with

$$\mathbf{S} = \mathrm{sym}(\nabla \mathbf{u}) = \mathrm{sym}(\mathrm{grad}(\mathbf{u}))$$
$$|\mathbf{T}| = \sqrt{\mathbf{T} : \mathbf{T}}$$

Some rearrangement of Eq. (157) is necessary to match the form of Definition (152) and (155). Eqns. (158) to (160) show the necessary steps to match the generic definition of ν_{SGS}.

$$\nu_{SGS} = C_S^2 \underbrace{\Delta}_{l_{SGS}} \underbrace{\Delta\sqrt{\mathbf{S}:\mathbf{S}}}_{q_{SGS}} \tag{158}$$

$$q_{SGS} = \sqrt{k_{SGS}} = \Delta\sqrt{\mathbf{S}:\mathbf{S}} \tag{159}$$

$$\Rightarrow k_{SGS} = \Delta^2\,\mathbf{S}:\mathbf{S} \tag{160}$$

Implementation

The implementation in the source code differs a little from the equations above.

```
void Smagorinsky::updateSubGridScaleFields(const volTensorField& gradU)
{
  nuSgs_ = ck_*delta()*sqrt(k(gradU));
  nuSgs_.correctBoundaryConditions();
}
```

Listing 271: The function `updateSubGridScaleFields()` in the file `Smagorinsky.C`

```
tmp<volScalarField> k(const tmp<volTensorField>& gradU) const
{
  return (2.0*ck_/ce_)*sqr(delta())*magSqr(dev(symm(gradU)));
}
```

Listing 272: The function `k()` in the file `Smagorinsky.H`

Listing 271 shows the implementation of how the sub-grid viscosity is computed by the Smagorinsky model in OpenFOAM. Listing 272 shows how the model calculates the sub-grid kinetic energy.

$$\texttt{nuSgs} = ck\Delta\sqrt{k} \tag{161}$$

$$k = 2\frac{ck}{ce}\Delta^2|\operatorname{dev}\mathbf{S}|^2 \tag{162}$$

with

$$\mathbf{S} = \operatorname{sym}\operatorname{grad}(\mathbf{u}) \tag{163}$$

it follows

$$\texttt{nuSgs} = ck\Delta\sqrt{2\frac{ck}{ce}\Delta^2|\operatorname{dev}\mathbf{S}|^2} \tag{164}$$

$$\texttt{nuSgs} = ck\sqrt{2\frac{ck}{ce}}\Delta^2|\operatorname{dev}\mathbf{S}| \tag{165}$$

the comparison with Eq. 157 shows

$$\nu_{SGS} = (C_S\Delta)^2|\mathbf{S}| \tag{157}$$

$$\Rightarrow C_S^2 = ck\sqrt{2\frac{ck}{ce}} \tag{166}$$

Eq. (166) shows how the Smagorinsky constant can be calculated from the model constants. The Smagorinsky constant is often stated in publications using or investigating the Smagorinsky model, because it is the only degree of freedom of the Smagorinsky model.

In OpenFOAM the Smagorinsky model has two model constants. ce is inherited from the class `GenEddyVisc`. This constant is used in the definition of the sub-grid dissipation rate. The default value of ce is 1.048 and is defined in the constructor of the class `GenEddyVisc` in the file `GenEddyVisc.C`.

Therefore, the model constant ck is the only degree of freedom of the Smagorinsky model of OpenFOAM. The default value of ck is 0.094. This results in a default value fofr C_S of $0.1995 \approx 0.2$. The value of C_S varies in literature depending on the publication from 0.07 to 0.33 [5, 12].

```
//- Return sub-grid disipation rate
virtual tmp<volScalarField> epsilon() const
{
  return tmp<volScalarField>
  (
    new volScalarField
    (
      IOobject
      (
        "epsilon",
        runTime_.timeName(),
        mesh_,
        IOobject::NO_READ,
        IOobject::NO_WRITE
      ),
      ce_*k()*sqrt(k())/delta()
    )
  );
}
```

Listing 273: The function `epsilon()` in the file `GenEddyVisc.H`

40.2.5 The `oneEqEddy` LES model

The `oneEqEddy` model is one of the standard LES models of OpenFOAM. This model is an one equation eddy viscosity model with a constant model coefficient. Eq. 167 shows how the sub-grid viscosity is calculated by the `oneEqEddy` model. The constant ck has a default value of 0.094.

$$\nu_{SGS} = ck \Delta \sqrt{k_{SGS}} \tag{167}$$

The transport equation for k_{SGS}

As this model is an one equation model, it introduces an additional equation to the set of equations. This additional equation is a transport equation for the sub-grid kinetic energy k_{SGS}. k_{SGS} is the kinetic energy of the unresolved protion of the velocity. Thus, k_{SGS} is called sub-grid kinetic energy.

$$\frac{\partial k_{SGS}}{\partial t} + \nabla \cdot (k_{SGS}\, \mathbf{u}) - \nabla \cdot (D_k \nabla k_{SGS}) = G - \epsilon_{SGS} \tag{168}$$

with

$$D_k = \nu + \nu_{SGS}$$
$$G = \nu_{SGS} \, |\mathrm{sym}(\nabla \mathbf{u})|^2$$
$$\epsilon_{SGS} = ce \frac{\sqrt{k_{SGS}}}{\Delta} k_{SGS}$$

Eq. 168 is similar to the transport equation for k of the k-ϵ model. Also the definition of the sub-grid viscosity is similar to the definition of the turbulent viscosity of the k-ϵ model. This is not very obvious. Therefore, we shall explore this matter further.

$$\nu_{SGS} = ck \Delta \sqrt{k_{SGS}} \tag{167}$$

$$\nu_{SGS} = ck \frac{ce}{ce} \frac{k_{SGS}}{k_{SGS}} \frac{\sqrt{k_{SGS}}}{\sqrt{k_{SGS}}} \Delta \sqrt{k_{SGS}} \tag{169}$$

$$\nu_{SGS} = ck\, ce \frac{k_{SGS} \sqrt{k_{SGS}} \sqrt{k_{SGS}}}{ce \frac{k_{SGS}\sqrt{k_{SGS}}}{\Delta}} \tag{170}$$

$$\nu_{SGS} = ck\, ce \frac{k_{SGS}^2}{\epsilon_{SGS}} \tag{171}$$

Eq. 171 is similar to Eq. 91 – the definition of the turbulent viscosity of the k-ϵ model

$$\nu_T = C_\mu \frac{k^2}{\epsilon} \tag{91}$$

The product of ck and ce when using their default values gives $ck \cdot ce = 0.0985$ which is approximately the default value of C_μ of the k-ϵ model, which is $C_\mu = 0.09$.

41 The use of `phi`

41.1 The question

The governing equations of the solvers of OpenFOAM are written in a special notation that makes it easy to compare the source codes with equations from a fluid dynamics textbook. In Section 19.1 the governing equations of the solver *pimpleFoam* are examined. There, the terms of Eq. 36 are compared with the source code, see Listing 146. Here, we repeat the comparison of how the convective term is written in the sources and how this term is expressed mathematically.

$$\underbrace{\nabla(\mathbf{uu})}_{\mathrm{div}(\mathbf{uu})} \Leftrightarrow \texttt{fvm::div(phi, U)}$$

We now examine how `phi` is defined and how we can find `phi` in the math.

41.2 Implementation

41.2.1 The origin of fields

One way to learn more about `phi` is to look for its definition in the source code of OpenFOAM.

Listing 274 shows the first lines of the `main` function of the solver *pimpleFoam*. The `main` function of any C or C++ program is entered, when this program is executed. So, the instructions of Listing 274 are the first instructions that are executed, when the solver is called.

In line 6 of Listing 274 the file `createFields.H` is included. This file contains instructions that create the data structures of all fields that are necessary for the solver (e.g. the pressure or the velocity field).

```
1  int main(int argc, char *argv[])
2  {
3    #include "setRootCase.H"
4    #include "createTime.H"
5    #include "createMesh.H"
6    #include "createFields.H"
7    #include "initContinuityErrs.H"
8
9    /* the rest of the solver */
```

Listing 274: The first few line of the `main` function of *pimpleFoam* in `pimpleFoam.C`

The file `createFields.H` contains the content of Listing 275. There, the velocity field `U` is created. In line 15 the file `createPhi.H` is included. There, the field `phi` is created.

```
1  Info<< "Reading field U\n" << endl;
2  volVectorField U
3  (
4    IOobject
5    (
6      "U",
7      runTime.timeName(),
8      mesh,
9      IOobject::MUST_READ,
10     IOobject::AUTO_WRITE
11   ),
12   mesh
13 );
```

```

#include "createPhi.H"
```

Listing 275: The creation of `U` and `phi` in the file `createFields.H`

41.2.2 How `phi` is defined

Listing 276 shows the content of the file `createPhi.H`. From this Listing we see the data type of `phi`, it is `surfaceScalarField`. This tells us, that `phi` is a scalar, that is defined on the faces of the control volumes (cells) of the mesh.

Line 13 tells us how `phi` is defined. There, we find out, that `phi` is the inner product of the velocity – we forget for the moment about the function `linearInterpolate` – and the face surface area vector. In Listing 277 we see the declaration of the function `Sf()`. In Listing 278 we see, that the variable `mesh` of Listing 276 is of the type `fvMesh`.

```
1  Info<< "Reading/calculating face flux field phi\n" << endl;
2
3  surfaceScalarField phi
4  (
5      IOobject
6      (
7          "phi",
8          runTime.timeName(),
9          mesh,
10         IOobject::READ_IF_PRESENT,
11         IOobject::AUTO_WRITE
12     ),
13     linearInterpolate(U) & mesh.Sf()
14 );
```

Listing 276: The creation of `phi` in the file `createPhi.H`

```
//- Return cell face area vectors
const surfaceVectorField& Sf() const;
```

Listing 277: The declaration of the method `Sf()` of the class `fvMesh` in the file `fvMesh.H`

```
Foam::Info
    << "Create mesh for time = "
    << runTime.timeName() << Foam::nl << Foam::endl;

Foam::fvMesh mesh
(
    Foam::IOobject
    (
        Foam::fvMesh::defaultRegion,
        runTime.timeName(),
        runTime,
        Foam::IOobject::MUST_READ
    )
);
```

Listing 278: The creation of the mesh in the file `createMesh.H`

41.3 The math

Now, let us examine the origin of `phi` from the mathematical point of view. We start with the governing equations of a solver for incompressible fluids. Therefore, Eq. 36 is repeated below.

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \nabla \cdot \operatorname{dev}(-\nu^{eff}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right)) = -\nabla p + \mathbf{Q} \tag{36}$$

This equation is written in diferential form and is valid everywhere in the fluid. In order to use the finite volume method, we need the governing equations in the integral form. Integrating Eq. (36) over a control volume yields:

$$\int_V \frac{\partial \mathbf{u}}{\partial t} + \nabla(\mathbf{u}\mathbf{u}) + \nabla\cdot\,\mathrm{dev}(-\nu^{eff}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right))\,\mathrm{d}V = \int_V -\nabla p + \mathbf{Q}\,\mathrm{d}V \tag{172}$$

Now we will have a closer look on the second term of Eq. (172). That is the convective term we already saw at the beginning of this section.

Using Gauss' theorem, we replace the integration over the volume of our control volume with the integration over the surface of the control volume.

$$\int_V \nabla(\mathbf{u}\mathbf{u})\,\mathrm{d}V = \oint_{\partial V} (\mathbf{u}\mathbf{u})\cdot\mathrm{d}\mathbf{S} \tag{173}$$

Because our control volume is a polyhedron (in most cases a hexahedron or a tetrahedron), the surface integral reduces to a sum of intergrals over the faces S_f of the polyhedron.

$$\oint_{\partial V} (\mathbf{u}\mathbf{u})\cdot\mathrm{d}\mathbf{S} = \sum_f \int_{S_f} (\mathbf{u}\mathbf{u})\cdot\mathrm{d}\mathbf{S}_f \tag{174}$$

$$\|\mathbf{S}_f\| = S_f \tag{175}$$

With $\mathbf{S}_f$ being the surface normal vector of the face f. The norm of this vector is equal to the area of the face f. We denote with the subscript f the mean face-value of a quantity. So, in Eq. (176) the term $(\mathbf{u}\mathbf{u})_f$ means the

$$\sum_f \int_{S_f} (\mathbf{u}\mathbf{u})\cdot\mathrm{d}\mathbf{S}_f = \sum_f (\mathbf{u}\mathbf{u})_f\cdot\mathbf{S}_f \tag{176}$$

$$(\mathbf{u}\mathbf{u})_f = \frac{1}{S_f}\int_{S_f} (\mathbf{u}\mathbf{u})\,\mathrm{d}\mathbf{S}_f \tag{177}$$

$$\sum_f (\mathbf{u}\mathbf{u})_f\cdot\mathbf{S}_f \approx \sum_f (\mathbf{u}_f\mathbf{u}_f)\cdot\mathbf{S}_f \tag{178}$$

Eq. (178) contains the fundamental assumption or approximation of the finite volume method. It is assumed, that the mean face-value of the product of the velocities is (approximately) equal to the product of the mean face-values of the velocity. In general, the operations averaging and multiplication are not commutative.

We are now nearly finished. The *rhs* of Eq. (178) contains all ingredients we need for `phi`. A surface area vector, a velocity and an inner vector product. See Listing 276. However, this ingredients are not in the order we need. Therefore, there is need for some more math to do.

A general rule of tensor calculus states:

$$\mathbf{a}\otimes\mathbf{b}\cdot\mathbf{c} = \mathbf{a}(\mathbf{b}\cdot\mathbf{c}) \tag{179}$$

In this document, we omit the symbol $\otimes$ for the sake of brevity.

$$\mathbf{a}\otimes\mathbf{b}\cdot\mathbf{c} = (\mathbf{a}\mathbf{b})\cdot\mathbf{c} \tag{180}$$

Eq. (180) looks like the *rhs* of Eq. (178).

$$(\mathbf{u}_f\mathbf{u}_f)\cdot\mathbf{S}_f = \mathbf{u}_f\,\underbrace{(\mathbf{u}_f\cdot\mathbf{S}_f)}_{=\phi_f} \tag{181}$$

$$\mathbf{u}_f(\mathbf{u}_f\cdot\mathbf{S}_f) = \mathbf{u}_f\,\phi_f \tag{182}$$

41.4 Summary

Now, after having dug deep into the sources and after having done some math, we can summarize all thoughts so far. We want to understand this equivalency.

$$\underbrace{\nabla(\mathbf{u}\mathbf{u})}_{\mathrm{div}(\mathbf{u}\mathbf{u})} \Leftrightarrow \texttt{fvm::div(phi, U)}$$

The math tells use the following identities.

$$\int_V \nabla(\mathbf{u}\mathbf{u})\,\mathrm{d}V = \oint_{\partial V} (\mathbf{u}\mathbf{u}) \cdot \mathrm{d}\mathbf{S} \tag{183}$$

$$\oint_{\partial V} (\mathbf{u}\mathbf{u}) \cdot \mathrm{d}\mathbf{S} = \sum_f (\mathbf{u}\mathbf{u})_f \cdot \mathbf{S}_f \tag{184}$$

$$\sum_f (\mathbf{u}\mathbf{u})_f \cdot \mathbf{S}_f \approx \sum_f (\mathbf{u}_f \mathbf{u}_f) \cdot \mathbf{S}_f \tag{185}$$

$$\sum_f (\mathbf{u}_f \mathbf{u}_f) \cdot \mathbf{S}_f = \sum_f \mathbf{u}_f (\mathbf{u}_f \cdot \mathbf{S}_f) \tag{186}$$

$$\sum_f \mathbf{u}_f (\mathbf{u}_f \cdot \mathbf{S}_f) = \sum_f \mathbf{u}_f \, \phi_f \tag{187}$$

We have shown, that the integral formulation of the convective term can be reformulated to incorporate ϕ and $\mathbf{u}$ instead of $\mathbf{u}\mathbf{u}$.

Part XI

Appendix

42 Useful Linux commands

42.1 Getting help

42.1.1 Display `-help`

Virtually all Linux commands display a summary of the programs purpose and usage. To display this message the command has to be invoked with one of those parameters: `-h`, `-help`, `--help`. If the wrong parameter is used the help message is displayed anyway or an error message naming the correct parameter to display the usage information, see Listing 279.

```
user@host:~$ ls -help
ls: invalid option -- e
Try 'ls --help' for more information.
user@host:~$
```

Listing 279: Displaying the help message

Apparently all of the tools and solvers of OpenFOAM[51] display such help messages. New Linux and OpenFOAM users are strongly encouraged to study the help messages to deepen their understanding and insight.

42.1.2 *man* pages

Many Linux commands have an additional, more detailed documentation[52]. This is written in the *man* pages (*man* is short for manual). To display the *man* pages of a certain command, simply put the name of the command or program behind the command `man`. Listing 280 shows how to display the *man* pages of the Linux command *cp*.

```
man cp
```

Listing 280: Displaying the *man* pages

The *man* pages cover general commands of Linux, system call, library function of the C standard library and much more. On some systems the man pages are only partially or not at all installed by default.

42.2 Finding files

42.2.1 Searching files system wide

Searching for a file on the whole file system can be done by *locate*. Listing 281 shows the result of the search for the source file of *icoFoam*.

```
user@host:~/OpenFOAM/user-2.1.x/run/icoTurb$ locate icoFoam.C
/home/user/OpenFOAM/OpenFOAM-2.0.x/applications/solvers/incompressible/icoFoam/icoFoam.C
/home/user/OpenFOAM/OpenFOAM-2.1.x/applications/solvers/incompressible/icoFoam/icoFoam.C
```

Listing 281: Looking for *icoFoam.C*

42.2.2 In a certain directory

To find a file in a certain directory and its sub-directories find can be used. Listing 282 shows the command to search the file LESProperties in the OpenFOAM tutorials.

[51] No exception is known to the author.
[52] As an example: the *man* pages of *gcc* are longer than 10000 lines.

```
find $FOAM_TUTORIALS -name LESProperties
```

Listing 282: Search *LESProperties* in the tutorials

42.3 Find files and scan them

How do I define probes? I have seen this already, but where?

To answer this question one has to find all files in which *probes* can be defined – the *controlDict* in this case. Additionally, all of the files returned by the search have to be scanned for the definition of *probes*. As an OpenFOAM case consists of a number of text files, it is easy to scan these files for certain keywords. So, the answer to the question above is: find all controlDicts and scan them for the word probe.

Instead of perfoming this task manually, a single one-liner in the Terminal does the magic. Listing 283 shows how all files named *controlDict* in the tutorials are located and scanned for the word *probes*.

```
find $FOAM_TUTORIALS -name controlDict | xargs grep 'probes' -sl
```

Listing 283: Find and scan files

find looks for respectively finds all files with the name passed with the option `-name` in the specified folder and its folders. *xargs* executes the passed command line. The output of *find* is passed to *grep* as input by a pipe. grep then scans all files for the word probes.

42.4 Scan a log file

grep can scan a text file for a certain pattern. In this example we want to scan the solver output for a certain pattern. The solver *twoPhaseEulerFoam* displays after every time step the minimum and maximum value of the volume fraction α. For α to be physically meaningful, its value has to be of the range $0 \leq \alpha \leq 1$.

In this example a simulation crashed and the main suspicion is, that there were values of α greater than one. Listing 284 shows two lines of solver output. The first line has a maximum value of one. In some cases, when regions evolve where the continuous phase vanishes, e.g. above a water surface, this value is perfectly reasonable. The second line comprises a maximum value of α greater than unity. This value is unphysical, because a phase can not occupy a certain amount of space – a cell – to more than 100%.

Due to the fact that simulations often do not crash immediately the log file containing the solver output is hundreds of thousands of lines long. To look for maximum values of α greater than unity manually is not an option. We need an one-liner that does that automatically for us. That's where *grep* comes in.

```
Dispersed phase volume fraction = 0.194351  Min(alpha) = 7.52826e-42  Max(alpha) = 1
Dispersed phase volume fraction = 0.060562  Min(alpha) = 2.30261e-52  Max(alpha) = 1.00003
```

Listing 284: Example: solver output regarding volume fraction

Listing 285 shows how the user can scan the log file for the appropiate pattern. grep expects as first argument the pattern to look for. The second argument is optional, it specifies the file from which to read. If no file was specified, grep would read from standard input. The option `-c` makes grep display only the number of number of matches. Otherwise, grep would display all lines in which a match was found. In a situation in which the number of hits could reach hundreds or thousands, displaying all lines with a match could be unwise.

The first command in Listing 285 would detect a match for both lines of Listings 284. So this pattern `'Max(alpha) = 1'` is not useful to find out whether α exceeded unity or not.

The second command in Listing 285 will only detect lines in which α is larger than unity. So, of the two lines of Listings 284, only the second one would result in a match.

```
grep 'Max(alpha) = 1' foamRun.log -c
grep 'Max(alpha) = 1.' foamRun.log -c
```

Listing 285: Scan the log using *grep*

42.5 Running in scripts

42.5.1 Starting a batch of jobs

To use the computing power of a computing cluster it is a good idea to let the cluster do the work in batches. To be able to do this, this section explains how to use a script to run a number of simulations sequentially. So, the cluster can calculate a great number of cases without the need for the user to start each job seperately. This would be unacceptable when simulating overnights.

The script in Listing 286 starts two parallel simulations inkluding domain decomposition and reconstruction. The script assumes to start from a directory which contains all two cases. The first group of commands changes into a subdirectory of the current directory (`cd './fullColumn_fineV01'`). The next commands perform all tasks of a parallel simulation. Then the script changes to the second case (`cd '../fullColumn_fineV02'`).

This is a very basic script. It contains no checks if a simulation has terminated prematurely or any other useful features.

```
#!/bin/bash
# fine 01

echo 'fine01'
cd './fullColumn_fineV01'

echo 'decomposing'
decomposePar > foamDecompose.log

mpirun -np 2 twoPhaseEulerFoam -parallel > foamRun.log

echo 'reconstructing'
reconstructPar > foamReconstruct.log

# fine 02
echo 'fine02'
cd '../fullColumn_fineV02'

echo 'decomposing'
decomposePar > foamDecompose.log

mpirun -np 2 twoPhaseEulerFoam -parallel > foamRun.log

echo 'reconstructing'
reconstructPar > foamReconstruct.log
```

Listing 286: Mit einem Shell-Skript mehrere Rechnungen nacheinander starten

42.5.2 Terminating a running script

There may be need to stop a script from any further execution without terminating the currently running simulation. This example assumes that a script with name runCalculations is to be terminated. First the PID of *runCalculations* has to be known. In Section 8.2.2 explains this bit in detail. Listing 286 shows how to look for the PID. The command in Listing 286 outputs two lines. The first line comes from the running script and the second line stems from the running parallel calculation. This is because all running processes matching the pattern `run` were searched for. Therefore, also the running instance of *mpirun* was found.

```
user@host:~$ ps -el | grep run
0 S  8553 14913 14517  0  80   0 -  2687 wait   pts/11   00:00:00 runCalculations
0 S  8553 14917 14913  0  80   0 -  2687 wait   pts/11   00:00:00 mpirun
user@host:~$
```

Listing 287: Search for PIDs using *ps* and *grep*

Terminate the script

If the script was terminated using `kill`, then the simulation would continue unaffected. Listing 288 shows how the script is terminated and mpirun continues to be running.

```
user@host:~$ ps -e | grep run
14913 pts/11    00:00:00 runCalculations
14917 pts/11    00:00:00 mpirun
user@host:~$ kill -KILL 14913
user@host:~$ ps -e | grep run
14917 pts/11    00:00:00 mpirun
```

Listing 288: Mit *kill* ein Skript beenden

Terminate the script and the simulation

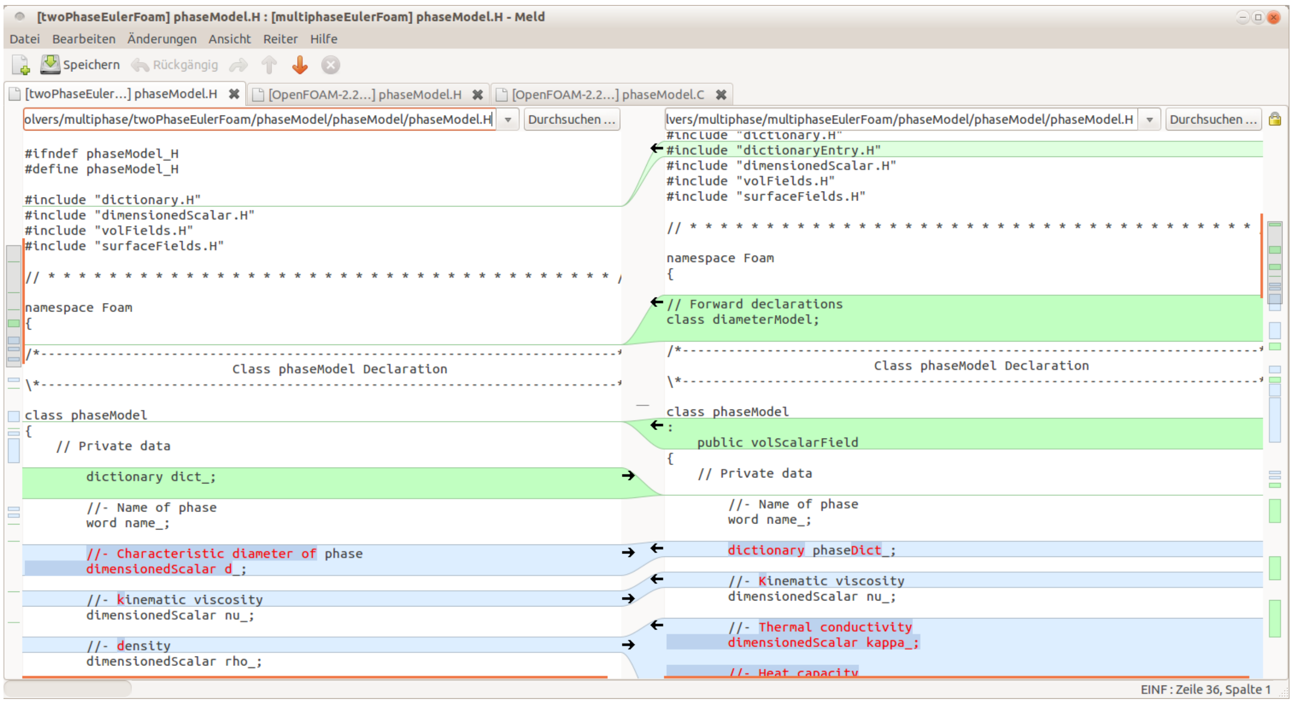
To terminate both the script and the simulation – in this example – the running simulation has to be terminated also. Terminating only the running simulation only, will cause the script to execute the next command. So, first the script and then the simulation need to be terminated.

42.6 diff

`diff` is a command line tool that analyses two files and prints a summary of the differences of those files. Further information on *diff* can be found in the man-pages or the help-message.

42.6.1 Meld

Meld is a graphical front-end to *diff*. This allows for a side-by-side comparison of both files under investigation. Parts of the file that differ are highlighted by colors. For more information about *Meld* see `http://meldmerge.org/`.


[twoPhaseEulerFoam] phaseModel.H : [multiphaseEulerFoam] phaseModel.H - Meld
Datei Bearbeiten Änderungen Ansicht Reiter Hilfe
Speichern Rückgängig
[twoPhaseEuler...] phaseModel.H
[OpenFOAM-2.2...] phaseModel.H
[OpenFOAM-2.2...] phaseModel.C
olvers/multiphase/twoPhaseEulerFoam/phaseModel/phaseModel/phaseModel.H
Durchsuchen ...
lvers/multiphase/multiphaseEulerFoam/phaseModel/phaseModel/phaseModel.H
Durchsuchen ...
EINF : Zeile 36, Spalte 1

```
#ifndef phaseModel_H
#define phaseModel_H

#include "dictionary.H"
#include "dimensionedScalar.H"
#include "volFields.H"
#include "surfaceFields.H"

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

namespace Foam
{

/*---------------------------------------------------------------------------*\
                       Class phaseModel Declaration
\*---------------------------------------------------------------------------*/

class phaseModel
{
    // Private data

        dictionary dict_;

        //- Name of phase
        word name_;

        //- Characteristic diameter of phase
        dimensionedScalar d_;

        //- kinematic viscosity
        dimensionedScalar nu_;

        //- density
        dimensionedScalar rho_;
```

```
#include "dictionary.H"
#include "dictionaryEntry.H"
#include "dimensionedScalar.H"
#include "volFields.H"
#include "surfaceFields.H"

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

namespace Foam
{

// Forward declarations
class diameterModel;

/*---------------------------------------------------------------------------*\
                       Class phaseModel Declaration
\*---------------------------------------------------------------------------*/

class phaseModel
:
    public volScalarField
{
    // Private data

        //- Name of phase
        word name_;

        dictionary phaseDict_;

        //- Kinematic viscosity
        dimensionedScalar nu_;

        //- Thermal conductivity
        dimensionedScalar kappa_;

        //- Heat capacity
```

Figure 39: A screenshot of *Meld*

42.7 Miscellaneous

This section contains references to useful scripts or commands explained elsewhere in this document.

Terminate a backround process

See Section 8.2.2.

Delete the *processor directories**

If one or several simulations have been conducted on a computing cluster, it makes sense so reconstruct the domain on the cluster. Otherwise the workstation of the user would be blocked for the time needed to complete reconstruction. After reconstructing the domain the *processor** directories still contain all the time step data. If the *processor** folders are deleted on the cluster, the user can afterwards copy the whole case directory to the workstation without transmitting the solution data twice.

See Section 8.4.2 for how to deal with *processor** directories.

Redirect output

Redirecting the output of a program is explained in Section 8.1.1.

43 Archive data

Parametric studies generate a great deal of data. After the post-processing is done all files could be compressed to save disk space. On Linux systems the *tar* archiving utility may be the agent of choice. The name *tar* comes from ***t****ape* ***ar****chive*, which is pretty descriptive in terms of the origins of this archiving program. A *tar* archive is a single file which contains all archived files and folders. This step alone is only a reorganisation of the data, fit for the usage of sequential data storage devices like magnetic tapes.

In a second step the tar archive needs to be compressed. For this task there are many possible choices. Linux systems usually provide programs like gzip, bzip2 or xz. The distinction between archiving and compressing is probably for historical as well as practical reasons. There is also one paradigm of the UNIX philosophy (*Make each program do one thing well*) which supports the segregation in archiving and compression. The compression programs usually differ in the utilised compression algorithms. There is one rule of thumb stating: The more data is to be compressed, the longer compression takes.

Table 6 lists the achieved compression of a parametric studies with 21 cases totalling in 50 GB of data. The data was written in ascii format. Compressing the data resulted in a 70+ % reduction of used disk space. If space consuming cases are to archived, slow algorithms that result in good compression rates should be prefered.

	used disk space	reduction	
21 cases uncompressed	50 GB		
compressed: *.tar.bz2	13.7 GB	36.3 GB	- 72.6 %

Table 6: Comparison of disk space reduction

Archive log files

In this example log files are archived. In this case the same algorithm achieves an even greater reduction of disk space usage. This example shows that the achieved compression rate strongly depends on the input data.

	used disk space	reduction	
16 log files uncompressed	2.0 GB		
compressed: *.tar.bz2	154.7 MB	1.85 GB	- 92.3 %

Table 7: Comparison of disk space reduction

Nomenclature

BC	boundary condition
CAD	computer aided design
CG	Conjugate gradient
EDF	Électricité de France
GAMG	Geometric algebraic multi-grid
gcc	GNU compiler collection
GNU	GNU is not Unix
GUI	graphical user interface
IGES	Initial Graphics Exchange Specification
MPI	message passing interface
OOP	object oriented programming
OS	operating system
Perl	An interpreted programming language
PID	process identifier
PIMPLE	An algorithm based on PISO and SIMPLE algorithm
PISO	Pressure Implicit with Split Operator
RHS	Right hand side
SAT	Standard ACIS Text
SI	Le Système Internationale d'Unités
SIMPLE	Semi-Implicit Method for Pressure-Linked Equations
STL	Surface Tesselation Language
UNIX	an operating system; ancestor of many modern operating systems, e.g. all kinds of Linux, Mac OS X.

References

[1] *FLUENT Theory Guide.*

[2] *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[3] The International System of Units, 2006.

[4] The International System of Units (SI), 2008.

[5] N. G. Deen B. Niceno, M. T. Dhotre. One.equation sub-grid scale (sgs) modelling for euler-euler large eddy simulation (eeles) of dispersed bubbly flow. *Chemical Engineering Science*, 63:3923–3931, 2008.

[6] A. Behzadi, R. I. Issa, and H. Rusche. Modelling of dispersed bubble and droplet flow at high phase fractions. *Chemical Engineering Science*, 59:759–770, 2004.

[7] J. Fröhlich. *Large Eddy Simulationen turbulenter Strömungen.* Teubner, 2006.

[8] E. Peirano & A.-E. Almstedt H. Enwald. Eulerian two-phase flow theory applied to fluidization. *Int. J. Multiphase Flow*, 22:21–66, 1996.

[9] M. Peric J. H. Ferzinger. *Computational Methods for Fluid Dynamics*. Springer, 2002.

[10] Hrvoje Jasak. *Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows*. PhD thesis, Imperial College of Science, Technology & Medicine, 1996.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 2nd edition, 1988.

[12] M. Milelli. *A numerical analysis of confined turbulent bubble plumes*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2002.

[13] OpenFOAM Foundation. *OpenFOAM - Programmer's Guide*, 2.1.0 edition, 2011.

[14] OpenFOAM Foundation. *OpenFOAM - User Guide*, 2.1.0 edition, 2011.

[15] Henrik Rusche. *Computational Fluid Dynamics of dispersed two-phase flows at high phase fractions*. PhD thesis, Imperial College of Science, Technology & Medicine, 2002.

[16] J. Smagorinsky. General circulation experiments with the primitive equations; i. the basic experiment. *Monthly Weather Review*, 91:99, 1963.

[17] Bjarne Stroustrup. *The C++ Programming language*. Addison-Wesley, 4th edition, 2013.

[18] Berend van Wachem. *Derivation, implementation and validation of computer simulation models for gas-solid fluidized beds*. PhD thesis, Delft University of Technology, 2000.

[19] David C. Wilcox. *Turbulence Modelling for CFD*. DCW Industries, Inc., 1994.