wolf dynamics

multiphysics simulations,
optimization & data analytics

www.wolfdynamics.com

OpenFOAM® Introductory Training Online session – 2020 Edition

- A help is needed, and much appreciated.
- If you find errors, have suggestions for better wording, figures, or new material, let us know.
- Also, if you find a tutorial that does not work, please let us know.
- Follow-up problems, questions, and suggestions at guerrero@wolfdynamics.com

Disclaimer

This offering is not approved or endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trademarks.

Wolf Dynamics makes no warranty, express or implied, about the completeness, accuracy, reliability, suitability, or usefulness of the information disclosed in this training material. This training material is intended to provide general information only. Any reliance the final user place on this training material is therefore strictly at his/her own risk. Under no circumstances and under no legal theory shall Wolf Dynamics be liable for any loss, damage or injury, arising directly or indirectly from the use or misuse of the information contained in this training material.

Revision 1-2020

JG

Acknowledgements

This training material and tutorials are based upon personal experience, OpenFOAM® source code, OpenFOAM® user guide, OpenFOAM® programmer's guide, and presentations from previous OpenFOAM® training sessions and OpenFOAM® workshops.

We gratefully acknowledge the following OpenFOAM® users for sharing online their material or for giving us their consent to use their material:

- Henry Weller and Chris Greenshields. The OpenFOAM Foundation.
- Hrvoje Jasak and Henrik Rusche. Wikki Ltd.
- Eugene de Villiers, Paolo Geremia, and Dan Combest. Engys.
- Hakan Nilsson. Chalmers University of Technology.
- Eric Paterson. Pennsylvania State University.
- Gavin Tabor. University of Exeter.
- Fumiya Nozaki. Yokohama, Japan.
- Marwan Darwish. American University of Beirut.
- Kevin Maki. University of Michigan.
- Tobias Holzmann. HolzmannCFD.

Acknowledgements

The following people have contributed directly to the development of this training material:

- Edoardo Alinovi.
- Matteo Bargiacchi.
- Mattia Cavaiola.
- Peyman Davvalo Khongar.
- Sehrish Naqvi.
- Damiano Natali.
- Stefano Olivieri.
- Biniyam Sishah.
- Giuseppe Zampogna.

On the training material

The following typographical conventions are used in this training material

- Text in `Courier new` font indicates Linux commands that should be typed literally by the user in the terminal.
- Text in **`Courier new bold`** font indicates directories.
- Text in *`Courier new italic`* font indicates human readable files or ascii files.
- Text in **Arial bold font** indicates program elements such as variables, function names, classes, statements and so on. It also indicate environment variables, and keywords. They also highlight important information.
- Text in Arial underline in blue font indicates URLs and email addresses.
- This icon ⚠ indicates a warning or a caution.
- This icon ☝ indicates a tip, suggestion, or a general note.
- This icon 🗀 indicates a folder or directory.
- This icon 🗎 indicates a human readable file (ascii file).
- This icon 🎞 indicates that the figure is an animation (animated gif).
- These characters `$>` indicate that a Linux command should be typed literally by the user in the terminal.

On the training material

The following typographical conventions are used in this training material

- Large code listing, ascii files listing, and screen outputs can be written in a square box, as follows:

```
1     #include <iostream>
2     using namespace std;
3
4     // main() is where program execution begins.  It is the main function.
5     // Every program in c++ must have this main function declared
6
7     int main ()
8     {
9           cout << "Hello world";                  //prints Hello world
10          return 0;                               //returns nothing
11    }
```

- To improve readability, the text might be colored.
- The font can be `Courier new` or **Arial bold**.
- And when required, the line number will be shown.

On the training material

Training material

- In the USB key you will find all the training material (tutorials, slides, quick reference guides, OpenFOAM® user guide, OpenFOAM® programmers manual, and lectures notes).
- You can extract the training material wherever you want. However, we highly recommend to extract all the training material in your OpenFOAM® user directory.
- From now on, we will refer to the directory where you extracted the training material as,
  - **$PTOFC**
    (abbreviation of **P**ath **T**o **O**pen**F**OAM® **C**ourse)

- To uncompress the tutorials go to the directory where you copied the training material and then type in the terminal,
  - `$> tar -zxvf file_name.tar.gz`

- In every single tutorial, you will find the file `README.FIRST`. In this file you will find the general instructions of how to run the case. You will also find some additional comments.
- In some cases, you will also find additional files with the extension .sh. These files can be used to run the case automatically, but we highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.
- You will find the automatic scripts in the cases explained in the lectures notes and some random cases.
- A word of caution, use the tutorials included in the training material just for recreational, instructional, or learning purposes and not for validation, benchmarking or as standard practices. ⚠

On the training material

Exercises

- At the end of each section, you will find an exercise section.
- The exercise section is optional, self-paced, and do it at anytime.
- The proposed exercises are designed to test your knowledge and to reinforce the concepts addressed during the lectures.
- All the concepts to be addressed in the exercise sections have been treated in the lecture notes, so the reader should not have problems answering the questions.
- If you have doubts, do not hesitate in asking.
- To help you answering the exercises, we might give you a few tips.
- And if it is necessary, the solution will be given.

Housekeeping issues

- **What OpenFOAM® version are we going to use?**
  - During this training we are going to use OpenFOAM® version 8.
  - The one developed by OpenCFD Ltd (http://www.openfoam.org/).

- **What Linux flavor should I use?**
  - We use OpenSUSE 15.1 or 15.2, but you are free to use any Linux flavor.

- **What Linux shell should I use?**
  - During this training we are going to use the BASH shell. If you want to know what shell you are using, type in the terminal
    - `$> echo $SHELL`
  - If the output is `/bin/bash`, you are using BASH shell.
  - If your output is something else, you are not using BASH shell. In this case, to start using BASH shell type in the terminal,
    - `$> bash`
  - If you do not know what is the terminal or how to use Linux, do not worry we are going to give a quick introduction later.

Training agenda

Module 0.

- Training agenda
- On the training material
- Housekeeping issues
- Additional information

Module 1.

- Introduction to OpenFOAM®
- A few OpenFOAM® simulations
- Library organization
- OpenFOAM® 101 – My first tutorial

Module 2.

- Solid modeling for CFD – Introduction to Onshape

Module 3.

- Meshing preliminaries
- Mesh quality assessment
- Meshing in OpenFOAM® – blockMesh and snappyHexMesh
- Mesh conversion and manipulation

Module 4.

- Running in parallel

Module 5.

- Sampling and plotting
- Data conversion

Module 6.

- The finite volume method. A crash introduction
- On the CFL number
- Boundary conditions and initial conditions
- Unsteady and steady simulations
- Assessing convergence
- Velocity pressure-coupling
- Linear solvers

Module 7.

- Implementing boundary conditions and initial conditions using codeStream

Module 8.

- Advanced modeling capabilities:
    - Turbulence modeling
    - Multiphase flows
    - Compressible flows
    - Moving reference frames and sliding grids
    - Moving bodies and rigid body motion
    - Source terms and passive scalars

Training agenda

Week 1.

- Course presentation, introduction to OpenFOAM®, running my first simulations, running in parallel

Week 2.

- Solid modeling using Onshape, mesh generation, mesh quality assessment, qualitative and quantitative postprocessing, scientific visualization

Week 3.

- Introduction to the finite volume method, numerical playground, best standard practices in CFD and OpenFOAM®, implementing boundary conditions and initial conditions using codeStream

Week 4.

- Implementing boundary conditions and initial conditions using codeStream (continuation), advanced physical models (turbulence, multiphase, compressible flows, dynamic meshes, source terms).

Week 5.

- Advanced physical models (continuation), extra topics (supplements), tips and tricks, closing remarks

Training agenda

Week 1.

- Module 0, Module 1, Module 4

Week 2.

- Module 2, Module 3, Module 5

Week 3.

- Module 6, Module 7

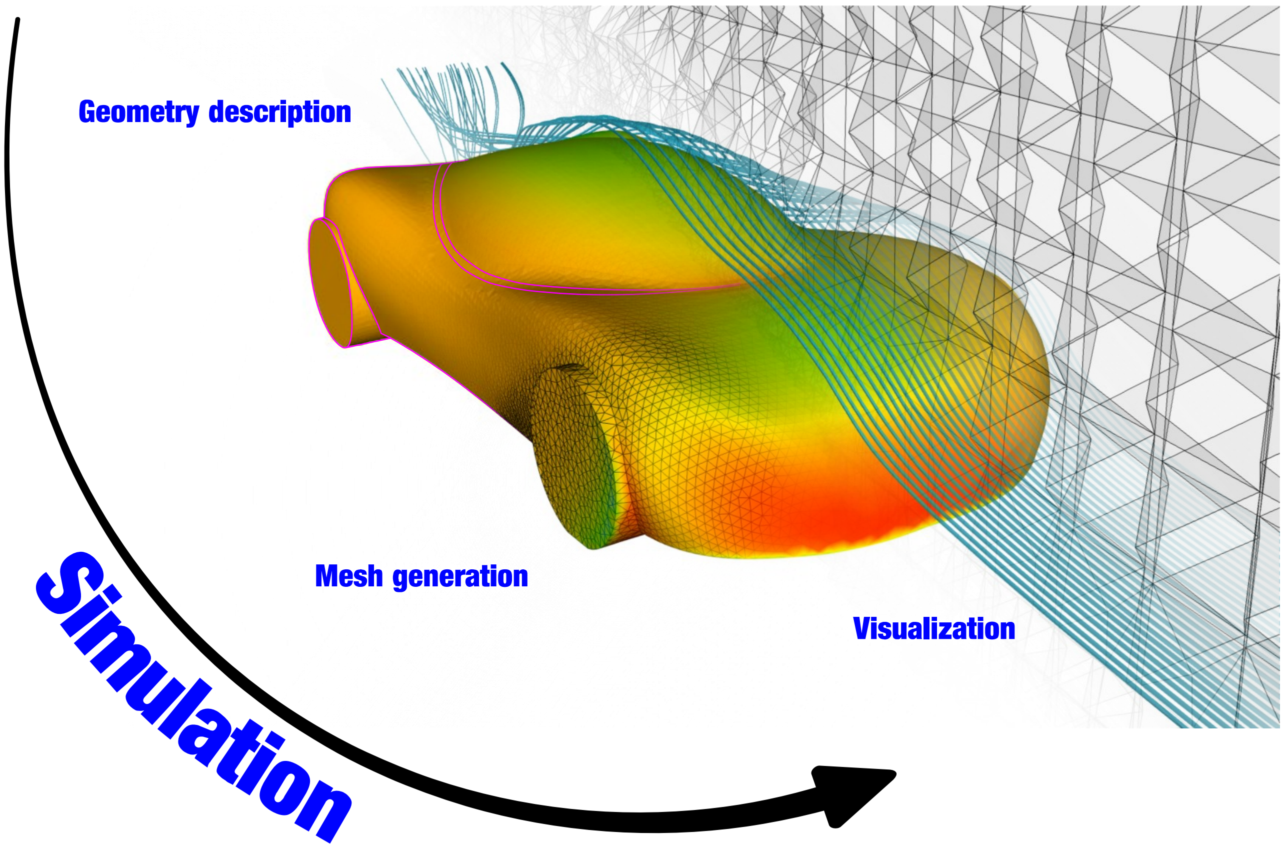
Week 4.

- Module 7, Module 8

Week 5.

- Module 8, extra topics

Training agenda



- The training agenda is organized in such a way that we will address the whole **CFD simulation workflow**.

Module 1

OpenFOAM® overview – First tutorial – Working our way in OpenFOAM®

Roadmap

1. **OpenFOAM® brief overview**
2. OpenFOAM® directory organization
3. Directory structure of an application/utility
4. Applications/utilities in OpenFOAM®
5. Directory structure of an OpenFOAM® case
6. Running my first OpenFOAM® case setup blindfold
7. A deeper view to my first OpenFOAM® case setup
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

OpenFOAM® brief overview

General description:

- OpenFOAM® stands for Open Source Field Operation and Manipulation.
- OpenFOAM® is first and foremost a C++ library used to solve partial differential equations (PDEs), and ordinary differential equations (ODEs).
- It comes with several ready-to-use or out-of-the-box solvers, pre-processing utilities, and post-processing utilities.
- It is licensed under the GNU General Public License (GPL). That means it is freely available and distributed with the source code.
- It can be used in massively parallel computers. No need to pay for separate licenses.
- It is under active development.
- It counts with a wide-spread community around the world (industry, academia and research labs).

OpenFOAM® brief overview

Multi-physics simulation capabilities:

- OpenFOAM® has extensive multi-physics simulation capabilities, among others:
  - Computational fluid dynamics (incompressible and compressible flows).
  - Computational heat transfer and conjugate heat transfer.
  - Combustion and chemical reactions.
  - Multiphase flows and mass transfer.
  - Particle methods (DEM, DSMC, MD) and lagrangian particles tracking.
  - Stress analysis and fluid-structure interaction.
  - Rotating frames of reference, arbitrary mesh interface, dynamic mesh handling, and adaptive mesh refinement.
  - 6 DOF solvers, ODE solvers, computational aero-acoustics, computational electromagnetics, computational solid mechanics, MHD.

OpenFOAM® brief overview

Physical modeling library:

- OpenFOAM® comes with many physical models, among others:
  - Extensive turbulence modeling capabilities (RANS, DES and LES).
  - Transport/rheology models. Newtonian and non-Newtonian viscosity models.
  - Thermophysical models and physical properties for liquids and gases.
  - Source terms models.
  - Lagrangian particle models.
  - Interphase momentum transfer models for multiphase flows.
  - Combustion, flame speed, chemical reactions, porous media, radiation, phase change.

OpenFOAM® brief overview

Under the hood you will find the following:

- Finite Volume Method (FVM) based solver.
- Collocated polyhedral unstructured meshes.
- Second order accuracy in space and time. Many discretization schemes available (including high order methods).
- Steady and transient solvers available.
- Pressure-velocity coupling via segregated methods (SIMPLE and PISO).
- But coupled solvers are under active development.
- Massive parallelism through domain decomposition.
- It comes with its own mesh generation tools.
- It also comes with many mesh manipulation and conversion utilities.
- It comes with many post-processing utilities.
- All components implemented in library form for easy re-use.

OpenFOAM® brief overview

OpenFOAM® vs. Commercial CFD applications:

- OpenFOAM® capabilities mirror those of commercial CFD applications.
- The main differences with commercial CFD applications are:
  - There is no native GUI.
  - It does not come with predefined setups. The users need to have a basic understanding of the CFD basics and be familiar with OpenFOAM® command line interface (CLI).
  - Knowing your way around the Linux bash shell is extremely useful.
  - It is not a single executable. Depending of what you are looking for, you will need to execute a specific application from the CLI.
  - It is not well documented, but the source code is available.
  - Access to complete source = no black magic. But to understand the source code you need to know object-oriented programming and C++.
  - Solvers can be tailored for a specific need, therefore OpenFOAM® is ideal for research and development.
  - It is free and has no limitation on the number of cores you can use.

OpenFOAM® brief overview

Developing new solvers (in case you need it):

- As the user has complete access to the source code, she/he has total freedom to modify existing solvers or use them as the starting point for new solvers.
- New solvers can be easily implemented using OpenFOAM® high level programming, *e.g.*:

$$\frac{\partial T}{\partial t} + \nabla \cdot (\phi T) - \nabla \cdot (\nu \nabla T) = 0 \quad \longrightarrow$$

```
solve
(
       fvm::ddt(T)
     + fvm::div(phi,T)
     - fvm::laplacian(nu,T)
       ==
       0
);
```

Correspondence between the implementation and the original equation is clear.

OpenFOAM® brief overview

> OpenFOAM® is an excellent piece of C++ and software engineering. Decent piece of CFD code.
>
> **H. Jasak**

Roadmap

1. ~~OpenFOAM® brief overview~~
2. **OpenFOAM® directory organization**
3. Directory structure of an application/utility
4. Applications/utilities in OpenFOAM®
5. Directory structure of an OpenFOAM® case
6. Running my first OpenFOAM® case setup blindfold
7. A deeper view to my first OpenFOAM® case setup
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

OpenFOAM® directory organization

```
$WM_PROJECT_DIR
├── Allwmake
├── applications
├── bin
├── COPYING
├── doc
├── etc
├── platforms
├── README.org
├── src
├── tutorials
└── wmake
```

If you installed OpenFOAM® in the default location, the environment variable **`$WM_PROJECT_DIR`** should point to the following directory (depending on the installed version):

`$HOME/OpenFOAM/OpenFOAM-8`

or

`$HOME/OpenFOAM/OpenFOAM-dev`

In this directory you will find all the files containing OpenFOAM® installation.

In this directory you will also find additional files (such as *`README.org`*, *`COPYING`*, etc.), but the most important one is *`Allwmake`*, which compiles OpenFOAM®.

OpenFOAM® directory organization

```
$WM_PROJECT_DIR
├── Allwmake
├── applications
├── bin
├── COPYING
├── doc
├── etc
├── platforms
├── README.org
├── src
├── tutorials
└── wmake
```

OpenFOAM® environment variables

The entries starting with the symbol $ are environment variables. You can find out the value of an environment variable by echoing its value, for example:

```
$> echo $WM_PROJECT_DIR
```

will print out the following information on the terminal,

```
$HOME/OpenFOAM/OpenFOAM-8
```

To list all the environment variables type in the terminal window,

```
$> env
```

To list all the environment variables related to OpenFOAM®, type in the terminal:

```
$> env | grep -i “OpenFOAM”
```

OpenFOAM® directory organization

```
$WM_PROJECT_DIR
├── Allwmake
├── applications
├── bin
├── COPYING
├── doc
├── etc
├── platforms
├── README.org
├── src
├── tutorials
└── wmake
```

OpenFOAM® aliases

You can go to any of these directories by using the predefined aliases set by OpenFOAM® (refer to *$WM_PROJECT_DIR/etc/config.sh/aliases* or *$WM_PROJECT_DIR/etc/config.csh/aliases*).

Just to name a few of the aliases defined:

```
alias foam=‘cd $WM_PROJECT_DIR’
alias app=‘cd $FOAM_APP’
alias src=‘cd $FOAM_SRC’
alias tut=‘cd $FOAM_TUTORIALS’
```

For a complete list type `alias` in the terminal.

To list all the aliases related to OpenFOAM®, type in the terminal:

```
$> alias | grep -i “FOAM”
```

OpenFOAM® directory organization

```
$WM_PROJECT_DIR
├── Allwmake
├── applications
├── bin
├── COPYING
├── doc
├── etc
├── platforms
├── README.org
├── src
├── tutorials
└── wmake
```

Let us study each directory inside **`$WM_PROJECT_DIR`**

- Any modification you add to the source code in **`WM_PROJECT_DIR`** will affect the whole library.
- Unless you know what are you doing, do not modify anything in the original installation (**`$WM_PROJECT_DIR`**), except for updates!

OpenFOAM® directory organization

The **applications** directory

```
$WM_PROJECT_DIR/applications
├── Allwmake
├── solvers
├── test
└── utilities
```

Let us visit the **`applications`** directory. Type in the terminal `app` or `$> cd $WM_PROJECT_DIR/applications`. You will find the following sub-directories:

- **`solvers`**, which contains the source code for the distributed solvers.
- **`test`**, which contains the source code of several test cases that show the usage of some of the OpenFOAM® classes.
- **`utilities`**, which contains the source code for the distributed utilities.

There is also an *`Allwmake`* script, which will compile all the content of **`solvers`** and **`utilities`**. To compile the test cases in **`test`** go to the desired test case directory and compile it by typing `wmake`.

OpenFOAM® directory organization

The **bin** directory

```
$WM_PROJECT_DIR/bin/
├── foamCleanPolyMesh
├── foamCleanTutorials
├── foamCloneCase
├── foamJob
├── foamLog
├── foamMonitor
├── foamNew
├── foamNewApp
├── foamNewBC
├── foamNewFunctionObject
├── paraFoam
├── ...
└── tools
```

The directory tree is not complete ⚠

Let us visit the **bin** directory:

- The **bin** directory contains many shell scripts, such as *foamNew*, *foamLog*, *foamJob*, *foamNewApp*, etc.

- This directory also contains the script *paraFoam* that will launch paraView.

OpenFOAM® directory organization

The **doc** directory

```
$WM_PROJECT_DIR/doc/
├── Allwmake
├── codingStyleGuide.org
├── Doxygen
├── Guides
└── tools
```

Let us visit the **doc** directory:

- The **doc** directory contains the documentation of OpenFOAM®, namely; user guide, programmer's guide and Doxygen generated documentation in html format.
- The Doxygen documentation needs to be compiled by typing `Allwmake doc` in **$WM_PROJECT_DIR**.
- You can access the Doxygen documentation online, http://cpp.openfoam.org/v8

OpenFOAM® directory organization

The **etc** directory

```
$WM_PROJECT_DIR/etc/
├── bashrc
├── caseDicts
├── cellModels
├── codeTemplates
├── config.csh
├── config.sh
├── controlDict
├── cshrc
├── paraFoam
├── README.org
├── templates
└── thermoData
```

Let us visit the **etc** directory:

- The **etc** directory contains the environment files, global OpenFOAM® instructions, templates, and the default thermochemical database *thermoData/thermoData*

- In the directory **caseDicts**, you will find many templates related to the input files used to setup a case in OpenFOAM®. We recommend you take some time and explore these files.

- It also contains the super dictionary *controlDict*, where you can set several debug flags and the defaults units.

OpenFOAM® directory organization

The **`platforms`** directory

```
$WM_PROJECT_DIR/platforms/
├── linux64GccDPInt32Opt
│   ├── applications
│   ├── bin
│   ├── lib
│   └── src
└── linux64GccDPInt32OptSYSTEMOPENMPI
    └── src
```

Let us visit the **`platforms`** directory.

- This directory contains the binaries generated when compiling the **`applications`** directory and the libraries generated by compiling the source code in the **`src`** directory.
- After compilation, the binaries will be located in the directory **`$WM_PROJECT_DIR/platforms/linux64GccDPInt32OptSYSTEMOPENMPI/bin`** **`$WM_PROJECT_DIR/platforms/linux64GccDPOpt/lib`**).
- After compilation, the libraries will be located in the directory **`$WM_PROJECT_DIR/platforms/linux64GccDPInt32OptSYSTEMOPENMPI/lib`**

OpenFOAM® directory organization

The **`src`** directory

```
$WM_PROJECT_DIR/src
├── Allwmake
├── combustionModels
├── finiteVolume
├── fvOptions
├── lagrangian
├── ...
├── OpenFOAM
├── parallel
├── MomentumTransportModels
├── sampling
├── sixDoFRigidBodyMotion
├── thermophysicalModels
├── transportModels
└── waves
```

The directory tree is not complete ⚠

Let us visit the **`src`** directory. Type in the terminal `src` or `$> cd $WM_PROJECT_DIR/src`

- This directory contains the source code for all the foundation libraries, this is the core of OpenFOAM®.
- It is divided in different subdirectories and each of them can contain several libraries.

A few interesting directories are:

- **`OpenFOAM`**. This core library includes the definitions of the containers used for the operations, the field definitions, the declaration of the mesh and mesh features such as zones and sets.

OpenFOAM® directory organization

The `src` directory

```
$WM_PROJECT_DIR/src
├── Allwmake
├── combustionModels
├── finiteVolume
├── fvOptions
├── lagrangian
├── ...
├── OpenFOAM
├── parallel
├── MomentumTransportModels
├── sampling
├── sixDoFRigidBodyMotion
├── thermophysicalModels
├── transportModels
└── waves
```

A few interesting directories are:

- **`finiteVolume`**. This library provides all the classes needed for the finite volume discretization, such as mesh handling, finite volume discretization operators (divergence, laplacian, gradient, fvc/fvm and so on), and boundary conditions. In the directory **`finiteVolume/lnInclude`** you also find the very important file *`fvCFD.H`*, which is included in most applications.
- **`MomentumTransportModels`**, which contains many turbulence models.
- **`sixDoFRigidBodyMotion`**. This core library contains the solver for rigid body motion.
- **`transportModels`**. This core library contains many transport models.

The directory tree is not complete ⚠

OpenFOAM® directory organization

The **tutorials** directory

```
$WM_PROJECT_DIR/tutorials/
├── Allclean
├── Allrun
├── Alltest
├── basic
├── combustion
├── compressible
├── discreteMethods
├── DNS
├── electromagnetics
├── financial
├── heatTransfer
├── incompressible
├── IO
├── lagrangian
├── mesh
├── multiphase
├── resources
└── stressAnalysis
```

Let us visit the **tutorials** directory. Type in the terminal `tut` or

```
$> cd $WM_PROJECT_DIR/tutorials
```

- The **tutorials** directory contains example cases for each solver. These are the tutorials distributed with OpenFOAM®.
- They are organized according to the physics involved.
- Use these tutorials as the starting point to develop you own cases.
- A word of caution, do not use these tutorials as best practices, they are there just to show how to use the applications. ⚠

OpenFOAM® directory organization

The **wmake** directory

```
$WM_PROJECT_DIR/wmake/
├── makefiles
├── platforms
├── rules
├── scripts
├── src
├── wclean
├── wcleanLnIncludeAll
├── wcleanPlatform
├── wdep
├── wmake
├── ...
├── wmakeFilesAndOptions
├── wmakeLnInclude
├── wmakeLnIncludeAll
├── ...
└── wrmo
```

The directory tree is not complete ⚠

Let us visit the **wmake** directory.

- OpenFOAM® uses a special make command: `wmake`
- `wmake` understands the file structure in OpenFOAM® and uses default compiler directives set in this directory.
- If you add a new compiler name in OpenFOAM® *bashrc* file, you should also tell `wmake` how to interpret that name.
- In **wmake/rules** you will find the default settings for the available compilers.
- In this directory, you will also find a few scripts that are useful when organizing your files for compilation, or for cleaning up.

OpenFOAM® directory organization

OpenFOAM® user directory

```
$HOME/OpenFOAM/
├── $WM_PROJECT_USER_DIR
└── $WM_PROJECT_DIR
```

- Let us now study OpenFOAM® user directory **`$WM_PROJECT_USER_DIR`**
- When working with OpenFOAM®, you can put your files wherever you want.
- To keep things in order, it is recommended to put your cases in your OpenFOAM® user directory or **`$WM_PROJECT_USER_DIR`**.
- It is also recommended to put your modified solvers, utilities, and libraries in your OpenFOAM® user directory or **`$WM_PROJECT_USER_DIR`**. This is done so you do not modify anything in the original installation.
- If you followed the standard installation instructions, the variable **`$WM_PROJECT_USER_DIR`** should point to the directory **`$HOME/OpenFOAM/USER_NAME-8`**, where **USER_NAME** is the name of the user (*e.g.*, johnDoe).

OpenFOAM® directory organization

Looking for information in OpenFOAM® source code

- To locate files you can use the `find` command.
- If you want to locate a directory inside **`$WM_PROJECT_DIR`** that contains the string **fvPatch** in its name, you can proceed as follows,
  - `$> find $WM_PROJECT_DIR -type d -name “*fvPatch*”`

    Where to look for

    Look for directories

    Case sensitive

    Look for this (using wildcards)

- If you want to locate a file inside **`$WM_PROJECT_DIR`** that contains the string **fvPatch** in its name, you can proceed as follows,
  - `$> find $WM_PROJECT_DIR -type f -name “*fvPatch*”`

    Where to look for

    Look for files

    Case sensitive

    Look for this (using wildcards)

- If you want to find a string inside a file, you can use the `grep` command.
- For example, if you want to find the string **LES** inside all the files within the directory **`$FOAM_SOLVERS`**, you can proceed as follows,
  - `$> grep -r -n “LES” $FOAM_SOLVERS`

  The argument `-r` means recursive and `-n` will output the line number.

OpenFOAM® directory organization

Looking for information in OpenFOAM® source code

- Dictionaries are input files required by OpenFOAM®.
- As you can imagine, there are many dictionaries in OpenFOAM®. The easiest way to find all of them is to do a local search in the installation directory as follows,
- For instance, if you are interested in finding all the files that end with the **Dict** word in the **`tutorials`** directory, in the terminal type:
  - `$> find $FOAM_TUTORIALS -name "*Dict"`
    (Case sensitive search)
  - `$> find $FOAM_TUTORIALS –iname '*dict'`
    (Non-case sensitive search)

- When given the search string, you can use single quotes ' ' or double-quotes " " (do not mixed them).
- We recommend to use single quotes, but it is up to you.

OpenFOAM® directory organization

Looking for information in OpenFOAM® source code

- A few more advanced commands to find information in your OpenFOAM® installation.
  - To find which tutorial files use the boundary condition **slip**:

    -
      ```
      $> find $FOAM_TUTORIALS -type f | xargs grep -sl ‘slip’
      ```

      This command will look for all files inside the directory **`$FOAM_TUTORIALS`**, then the output is used by grep to search for the string **slip**.
  - To find where the source code for the boundary condition **slip** is located:

    -
      ```
      $> find $FOAM_SRC -name “*slip*”
      ```
  - To find what applications do not run in parallel:

    -
      ```
      $> find $WM_PROJECT_DIR -type f | xargs grep -sl ‘noParallel’
      ```
- OpenFOAM® understands REGEX syntax.

OpenFOAM® directory organization

Environment variables

- Remember, OpenFOAM® uses its own environment variables.
- OpenFOAM® environment settings are contained in the **`OpenFOAM-8/etc`** directory.
- If you installed OpenFOAM® in the default location, they should be in:
  - **`$HOME/OpenFOAM/OpenFOAM-8/etc`**
- If you are running bash or ksh (if in doubt type in the terminal `echo $SHELL`), you sourced the *`$WM_PROJECT_DIR/etc/bashrc`* file by adding the following line to your *`$HOME/.bashrc`* file:
  - `source $HOME/OpenFOAM/OpenFOAM-8/etc/bashrc`

- By sourcing the file *`$WM_PROJECT_DIR/etc/bashrc`*, we start to use OpenFOAM® environment variables.
- By default, OpenFOAM® uses the system compiler and the system MPI compiler.
- When you use OpenFOAM® you are using its environment settings, that is, its path to libraries and compilers. So if you are doing software developing, and you are having compilation problems due to conflicting libraries or missing compilers, try to unload OpenFOAM® environment variables

Roadmap

1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. **Directory structure of an application/utility**
4. Applications/utilities in OpenFOAM®
5. Directory structure of an OpenFOAM® case
6. Running my first OpenFOAM® case setup blindfold
7. A deeper view to my first OpenFOAM® case setup
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

Directory structure of an OpenFOAM® application/utility

Directory structure of a general solver

```
$WM_PROJECT_DIR/applications/solvers/solverName/
├── createFields.H
├── appName.C
└── Make
    ├── files
    └── options
```

The **$WM_PROJECT_DIR/applications/solvers/solverName/** directory contains the source code of the solver.

- *solverName/appName.C*: is the actual source code of the solver.
- *solverName/createFields.H*: declares all the field variables and initializes the solution.
- The **Make** directory contains compilation instructions.
  - *Make/files*: names all the source files (*.C*), it specifies the solverName name and location of the output file.
  - *Make/options*: specifies directories to search for include files and libraries to link the solver against.

Directory structure of an OpenFOAM® application/utility

Directory structure of a general utility

```
$WM_PROJECT_DIR/applications/utilities/utilityName/
├── utility_dictionary
├── utilityName.C
├── header_files.H
└── Make
    ├── files
    └── options
```

The **`$WM_PROJECT_DIR/utilities/utilityName/`** directory contains the source code of the utility.

- *`utilityName/utilityName.C`*: is the actual source code of the application.
- *`utilityName/header_files.H`*: header files required to compile the application.
- *`utilityName/utility_dictionary`*: application's master dictionary.
- The **`Make`** directory contains compilation instructions.
    - *`Make/files`*: names all the source files `(.C)`, it specifies the `utilityName` name and location of the output file.
    - *`Make/options`*: specifies directories to search for include files and libraries to link the solver against.

Directory structure of an OpenFOAM® application/utility

- For your own solvers and utilities, it is recommended to put the source code in the directory **$WM_PROJECT_USER_DIR** following the same structure as in **$WM_PROJECT_DIR/applications/solvers** and **$WM_PROJECT_DIR/utilities/**.
- Also, you will need to modify the files *Make/files* and *Make/options* to point to the new name and location of the compiled binaries and libraries to link the solver against.
- You can do anything you want to your own copies, so you do not risk messing things up.
- This is done so you do not modify anything in the original installation, except for updates!

Roadmap

1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. ~~Directory structure of an application/utility~~
4. **Applications/utilities in OpenFOAM®**
5. Directory structure of an OpenFOAM® case
6. Running my first OpenFOAM® case setup blindfold
7. A deeper view to my first OpenFOAM® case setup
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

Applications/utilities in OpenFOAM®

- OpenFOAM® is not a single executable.
- Depending of what you want to do, you will need to use a specific application and there are many of them.
- If you are interested in knowing all the solvers, utilities, and libraries that come with your OpenFOAM® distribution, read the applications and libraries section in the user guide (chapter 3).
- In the directory **`$WM_PROJECT_DIR/doc`** you will find the documentation in pdf format.
- You can also access the online user guide. Go to the link http://cfd.direct/openfoam/user-guide/#contents, then go to chapter 3 (applications and libraries).
- If you want to get help on how to run an application, type in terminal

```
$> application_name -help
```

- The option `-help` will not run the application; it will only show all the options available.
- You can also get all the help you want from the source code.

Applications/utilities in OpenFOAM®

- You will find all the applications in the directory **`$FOAM_SOLVERS`** (you can use the alias `sol` to go there).
- You will find all the utilities in the directory **`$FOAM_UTILITIES`** (you can use the alias `util` to go there).
- For example, in the directory **`$FOAM_SOLVERS`**, you will find the directories containing the source code for the solvers available in the OpenFOAM® installation (version 8):

  - **basic**
  - **combustion**
  - **compressible**
  - **discreteMethods**
  - **DNS**
  - **electromagnetics**
  - **financial**
  - **heatTransfer**
  - **incompressible**
  - **lagrangian**
  - **multiphase**
  - **stressAnalysis**

- The solvers are subdivided according to the physics they address.
- The utilities are also subdivided in a similar way.

Applications/utilities in OpenFOAM®

- For example, in the sub-directory **`incompressible`** you will find several sub-directories containing the source code of the following solvers:

  - **adjointShapeOptimizationFoam**
  - **boundaryFoam**
  - **icoFoam**
  - **nonNewtonianIcoFoam**
  - **pimpleFoam**
  - **pisoFoam**
  - **shallowWaterFoam**
  - **simpleFoam**

- Inside each directory, you will find a file with the extension *`*.C`* and the same name as the directory. This is the main file, where you will find the top-level source code and a short description of the solver or utility.
- For example, in the file *`incompressible/icoFoam/icoFoam.C`* you will find the following description:

Transient solver for incompressible, laminar flow of Newtonian fluids.

Applications/utilities in OpenFOAM®

- Remember, OpenFOAM® is not a single executable.
- You will need to find the solver or utility that best fit what you want to do.
- A few solvers that we will use during this course:
  - **icoFoam**: laminar incompressible unsteady solver. Be careful, do not use this solver for production runs as it has many limitations.
  - **simpleFoam**: incompressible steady solver for laminar/turbulent flows.
  - **pimpleFoam**: incompressible unsteady solver for laminar/turbulent flows.
  - **rhoSimpleFoam**: compressible steady solver for laminar/turbulent flows.
  - **rhoPimpleFoam**: unsteady compressible solver for (laminar/turbulent flows.
  - **interFoam**: unsteady multiphase solver for separated flows using the VOF method (laminar and turbulent flows).
  - **laplacianFoam**: Laplace equation solver.
  - **potentialFoam**: potential flow solver.
  - **scalarTransportFoam**: steady/unsteady general transport equation solver.

Applications/utilities in OpenFOAM®

- Take your time and explore the source code.
- Also, while exploring the source code be careful not to add unwanted modifications in the original installation.
- If you modify the source code, be sure to do the modifications in your user directory instead of the main source code.

Roadmap

1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. ~~Directory structure of an application/utility~~
4. ~~Applications/utilities in OpenFOAM®~~
5. **Directory structure of an OpenFOAM® case**
6. Running my first OpenFOAM® case setup blindfold
7. A deeper view to my first OpenFOAM® case setup
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

Directory structure of an OpenFOAM® case

Directory structure of a general case

```
case_name
├── 0
│   ├── p
│   └── U
├── constant
│   ├── polyMesh
│   │   ├── boundary
│   │   ├── faces
│   │   ├── neighbour
│   │   ├── owner
│   │   └── points
│   └── transportProperties
├── system
│   ├── controlDict
│   ├── fvSchemes
│   └── fvSolution
└── time_directories
```

- OpenFOAM® uses a very particular directory structure for running cases.
- You should always follow the directory structure, otherwise, OpenFOAM® will complain.
- To keep everything in order, the case directory is often located in the path `$WM_PROJECT_USER_DIR/run`.
- This is not compulsory but highly advisable. You can copy the case files anywhere you want.
- The name of the case directory is given by the user (**do not use white spaces or strange symbols**).
- Depending of the solver or application you would like to use, you will need different files in each sub-directory.
- Remember, you always run the applications and utilities in the **top level** of the case directory (**the directory with the name case_name**). Not in the directory `system`, not in the directory `constant`, not in the directory `0`.

Directory structure of an OpenFOAM® case

Directory structure of a general case

```
case_name
├── 0
│   ├── p
│   └── U
├── constant
│   ├── polyMesh
│   │   ├── boundary
│   │   ├── faces
│   │   ├── neighbour
│   │   ├── owner
│   │   └── points
│   └── transportProperties
├── system
│    ├── controlDict
│    ├── fvSchemes
│    └── fvSolution
└── time_directories
```

`case_name`: the name of the case directory is given by the user (**do not use white spaces or strange symbols**).

This is the top-level directory, where you run the applications and utilities.

> `system`: contains run-time control and solver numerics.
>
> `constant`: contains physical properties, turbulence modeling properties, advanced physics and so on.
>
> > `constant/polyMesh`**:** contains the polyhedral mesh information.
>
> `0`: contains boundary conditions (BC) and initial conditions (IC).
>
> `time_directories`: contains the solution and derived fields. These directories are created by the solver automatically and according to the preset saving frequency, *e.g.*, 1, 2, 3, 4, … , 100.

Roadmap

1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. ~~Directory structure of an application/utility~~
4. ~~Applications/utilities in OpenFOAM®~~
5. ~~Directory structure of an OpenFOAM® case~~
6. **Running my first OpenFOAM® case setup blindfold**
7. A deeper view to my first OpenFOAM® case setup
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

Running my first OpenFOAM® case setup blindfold

Before we start – Always remember the directory structure

```
case_name
├── 0
├── constant
│   └── polyMesh
├── system
└── time_directories
```

- To keep everything in order, the case directory is often located in the path **`$WM_PROJECT_USER_DIR/run`**.
- This is not compulsory but highly advisable, you can put the case in any directory of your preference.
- The name of the case directory if given by the user (do not use white spaces).
- You run the applications and utilities in the top level of this directory.
- The directory **`system`** contains run-time control and solver numerics.
- The directory **`constant`** contains physical properties, turbulence modeling properties, advanced physics and so on.
- The directory **`constant/polyMesh`** contains the polyhedral mesh information.
- The directory **`0`** contains boundary conditions (BC) and initial conditions (IC).

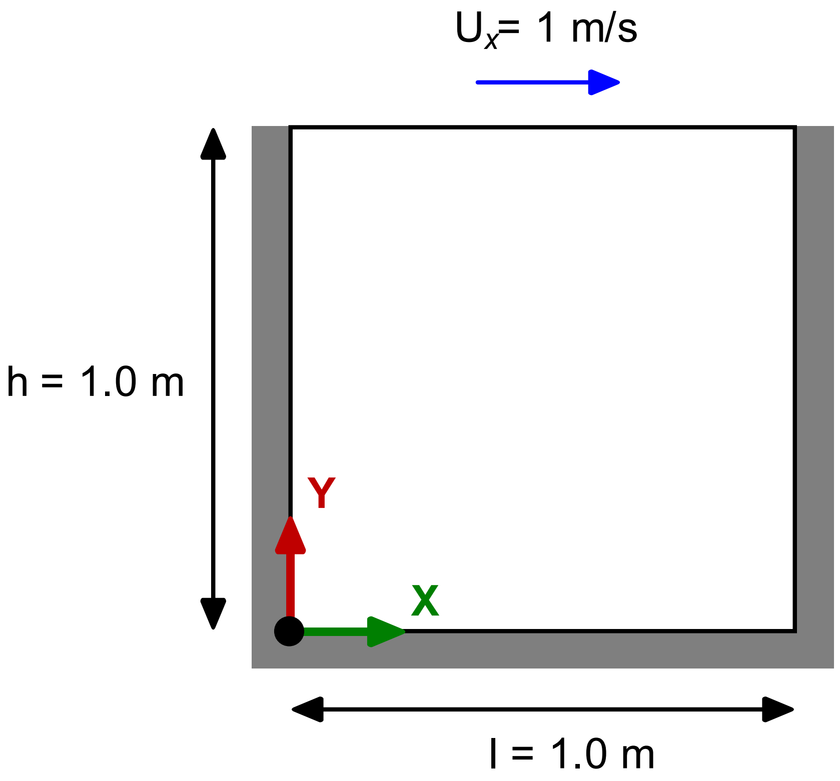
Running my first OpenFOAM® case setup blindfold

Before we start – Setting OpenFOAM® cases

- As you will see, it is quite difficult to remember all the dictionary files needed to run each application.
- It is even more difficult to recall the compulsory and optional entries of each input file.
- When setting a case from scratch in OpenFOAM®, what you need to do is find a tutorial or a case that close enough does what you want to do and then you can adapt it to your physics.
- Having this in mind, you have two sources of information:
  - **`$WM_PROJECT_DIR/tutorials`**
    (The tutorials distributed with OpenFOAM®)
  - **`$PTOFC`**
    (The tutorials used during this training)
- If you use a GUI, things are much easier. However, OpenFOAM® does not come with a native GUI interface.
- We are going to do things in the hard way (and maybe the smart way), we are going to use the Linux terminal

Running my first OpenFOAM® case setup blindfold

Flow in a lid-driven square cavity – Re = 100
Incompressible flow



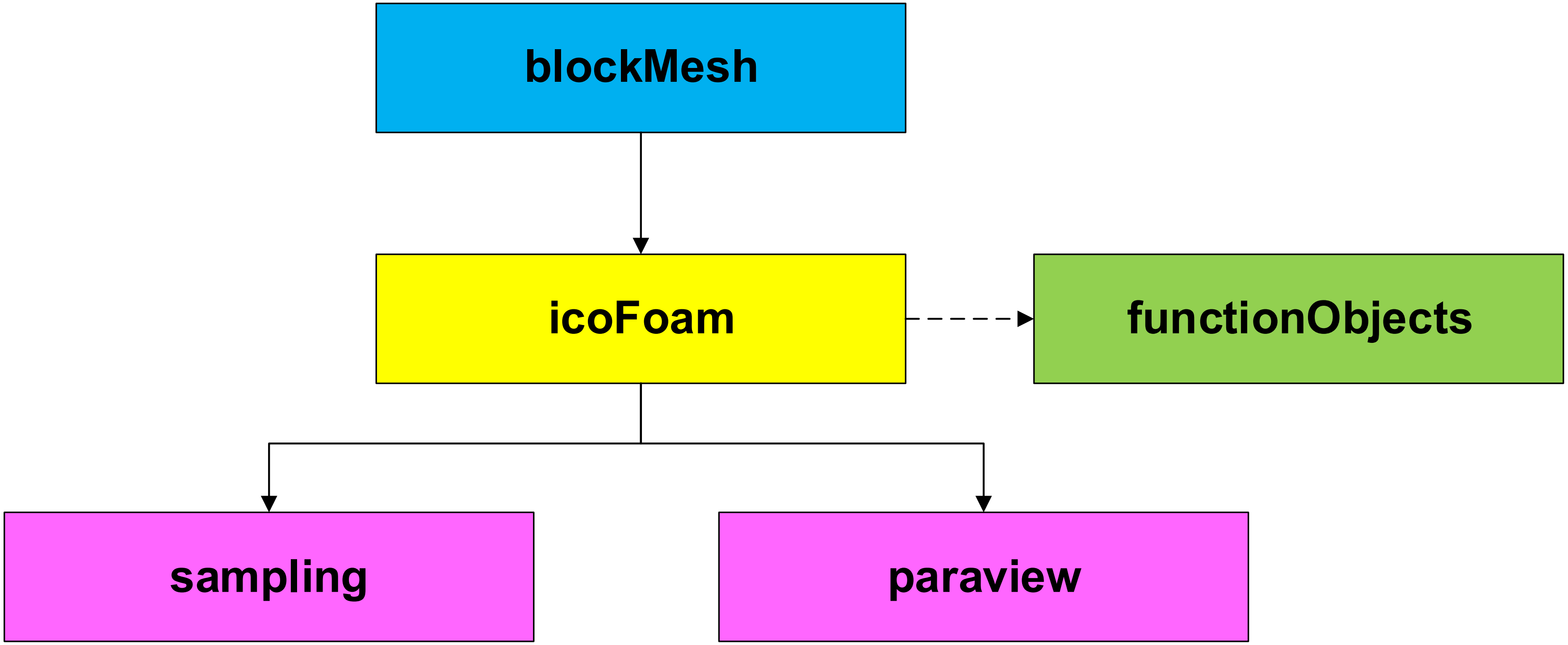
▨ **No-slip wall**

Physical and numerical side of the problem:

- The governing equations of the problem are the incompressible laminar Navier-Stokes equations.
- We are going to work in a 2D domain, but the problem can be easily extended to 3D.
- To find the numerical solution we need to discretize the domain (mesh generation), set the boundary and initial conditions, define the flow properties, setup the numerical scheme and solver settings, and set runtime parameters (time step, simulation time, saving frequency and so on).
- For convenience, when dealing with incompressible flows we will use relative pressure.
- All the dictionaries files have been already preset.

Running my first OpenFOAM® case setup blindfold

Workflow of the case



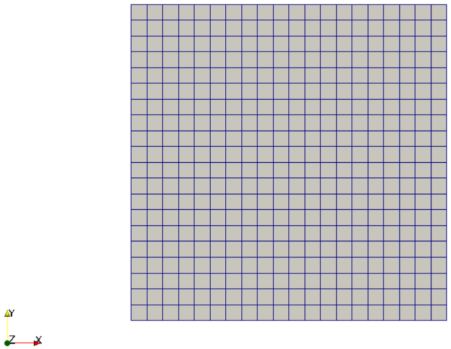
Running my first OpenFOAM® case setup blindfold

A word of caution about the solver icoFoam

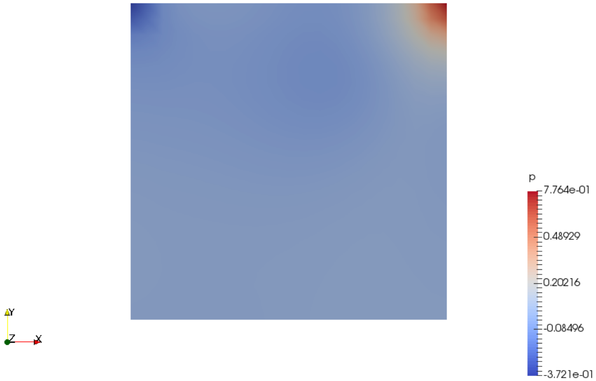
- The solver `icoFoam` is targeted for laminar incompressible unsteady solver.
- We do not recommend the use of this solver for production runs as it has no modeling capabilities and limited post-processing features.
- Instead of using `icoFoam`, you are better of with `pisoFoam` or `pimpleFoam`.

Running my first OpenFOAM® case setup blindfold

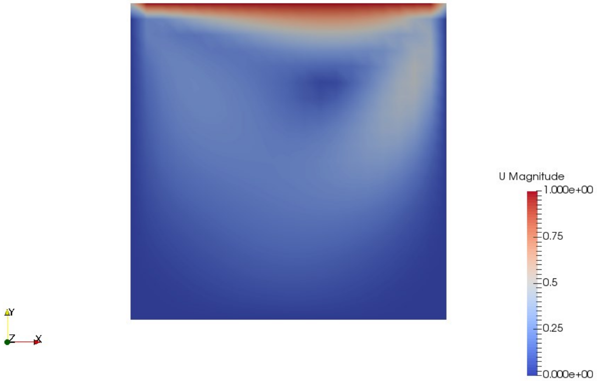
At the end of the day, you should get something like this



Mesh (very coarse and 2D)



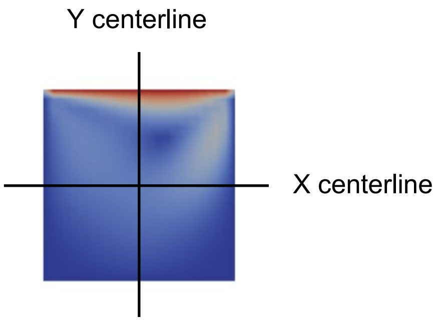
Pressure field (relative pressure)



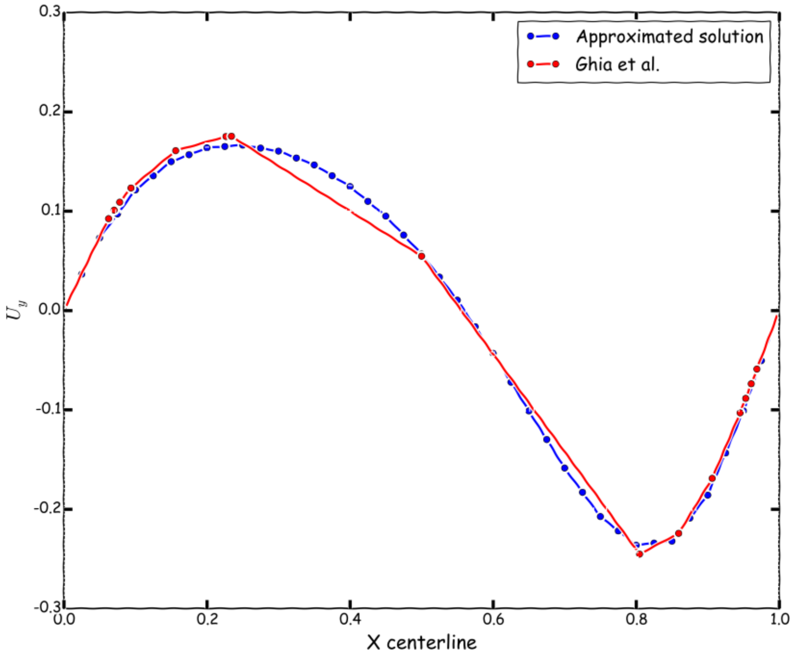
Velocity magnitude field

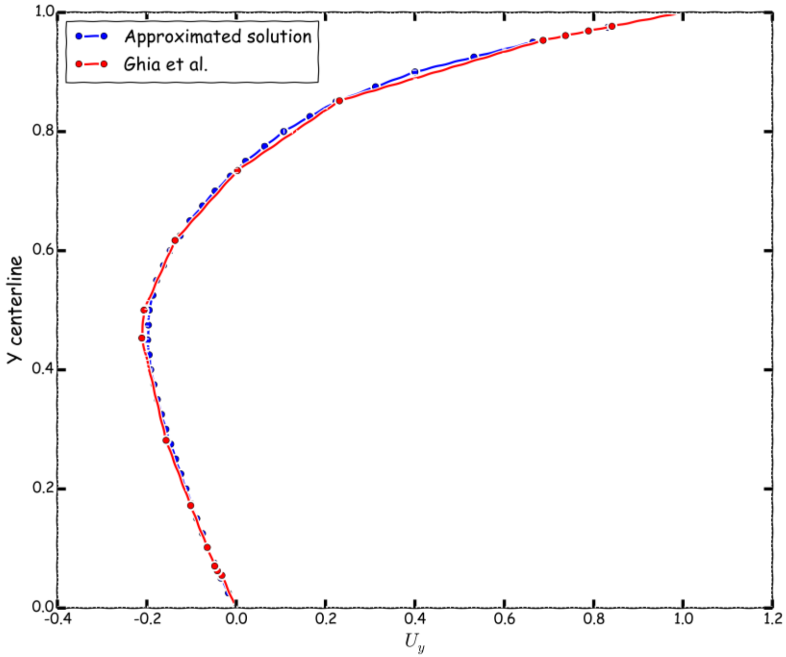
Running my first OpenFOAM® case setup blindfold

At the end of the day, you should get something like this



- And as CFD is not only about pretty colors, we should also validate the results





High-Re Solutions for incompressible flow using the navier-stokes equations and a multigrid method
U. Ghia, K. N. Ghia, C. T. Shin.
Journal of computational physics, 48, 387-411 (1982)

Running my first OpenFOAM® case setup blindfold

- Let us run our first case. Go to the directory:

```
$PTOFC/101OF/cavity2D
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Running my first OpenFOAM® case setup blindfold

Loading OpenFOAM® environment

- If you are using the lab workstations, you will need to source OpenFOAM® (load OpenFOAM® environment).

- To source OpenFOAM®, type in the terminal:
  - `$> of8`

- To use PyFoam (a plotting utility) you will need to source it. Type in the terminal:
  - `$> anaconda3`

- Remember, every time you open a new terminal window you need to source OpenFOAM® and PyFoam.
- Also, you might need to load OpenFOAM® again after loading PyFoam.
- By default, when installing OpenFOAM® and PyFoam you do not need to do this. This is our choice as we have many things installed and we want to avoid conflicts between applications.

Running my first OpenFOAM® case setup blindfold

What are we going to do?

- We will use the lid-driven square cavity tutorial as a general example to show you how to set up and run solvers and utilities in OpenFOAM®.
- In this tutorial we are going to generate the mesh using `blockMesh`.
- After generating the mesh, we will look for topological errors and assess the mesh quality. For this we use the utility `checkMesh`. Later on, we are going to talk about what is a good mesh.
- Then, we will find the numerical solution using `icoFoam`, which is a transient solver for incompressible, laminar flow of Newtonian fluids. By the way, we hope you did not forget where to look for this information.
- And we will finish with some quantitative post-processing and qualitative visualization using `paraFoam` and OpenFOAM® utilities.
- While we run this case, we are going to see a lot of information on the screen (standard output stream or stdout), but it will not be saved. This information is mainly related to convergence of the simulation, we will talk about this later on.
- A final word, we are going to use the solver `icoFoam` but have in mind that this is a very basic solver with no modeling capabilities and limited post-processing features.
- Therefore, is better to use `pisoFoam` or `pimpleFoam` which are equivalent to `icoFoam` but with many more features.

Running my first OpenFOAM® case setup blindfold

Running the case blindfold

- Let us run this case blindfold.
- Later we will study in detail each file and directory.
- Remember, the variable `$PTOFC` is pointing to the path where you unpacked the tutorials.
- You can create this environment variable or write down the path to the directory.
- In the terminal window type:

```
1.  $> cd $PTOFC/101OF/cavity
2.  $> ls -l
3.  $> blockMesh
4.  $> checkMesh
5.  $> icoFoam
6.  $> postProcess -func sampleDict -latestTime
7.  $> gnuplot gnuplot/gnuplot_script
8.  $> paraFoam
```

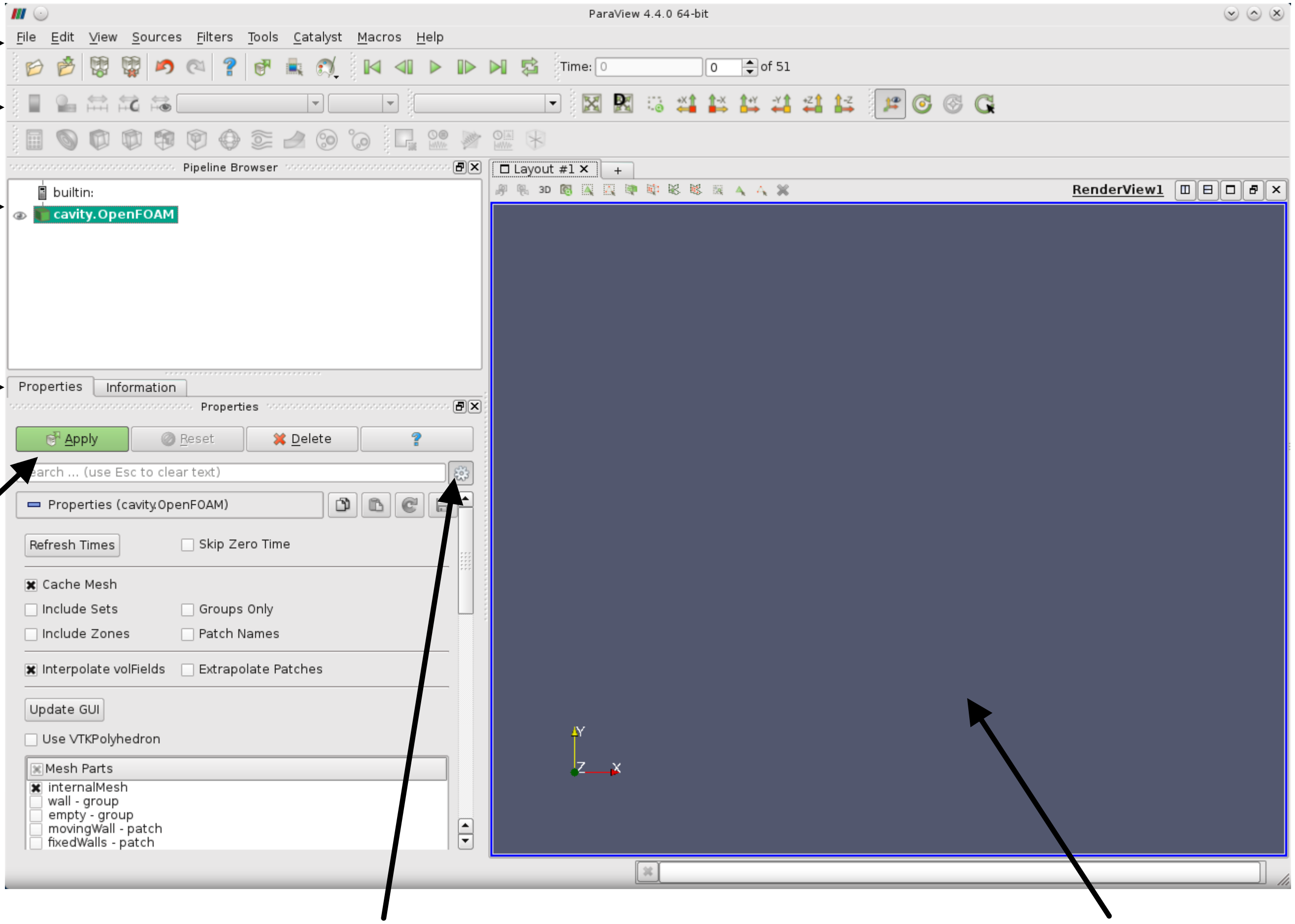
Running my first OpenFOAM® case setup blindfold

Running the case blindfold

- In step 1 we go to the case directory. Remember, `$PTOFC` is pointing to the path where you unpacked the tutorials.
- In step 2 we just list the directory structure (this step is optional). Does it look familiar to you? In the directory **`0`** you will the initial and boundary conditions, in the **`constant`** directory you will find the mesh information and physical properties, and in the directory **`system`** you will find the dictionaries that controls the numerics, runtime parameters and sampling.
- In step 3 we generate the mesh.
- In step 4 we check the mesh quality. We are going to address how to assess mesh quality later on.
- In step 5 we run the simulation. This will show a lot information on the screen, the standard output stream will not be saved.
- In step 6 we use the utility `postProcess` to do some sampling only of the last saved solution (the `latestTime` flag). This utility will read the dictionary file named *`sampleDict`* located in the directory **`system`**.
- In step 7 we use a gnuplot script to plot the sampled values. Feel free to take a look and reuse this script.
- Finally, in step 8 we visualize the solution using `paraFoam`. In the next slides we are going to briefly explore this application.

Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam



Time: 0 | 0 of 51

builtin:
cavity.OpenFOAM

Search ... (use Esc to clear text)

Properties (cavity.OpenFOAM)

Refresh Times □ Skip Zero Time

☒ Cache Mesh

□ Include Sets □ Groups Only

□ Include Zones □ Patch Names

☒ Interpolate volFields □ Extrapolate Patches

Update GUI

□ Use VTKPolyhedron

☒ Mesh Parts
- ☒ internalMesh
- □ wall - group
- □ empty - group
- □ movingWall - patch
- □ fixedWalls - patch

Menu Bar

Toolbars

Pipeline Browser

Properties panel

`Apply button`

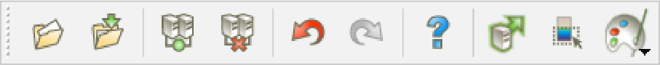
Press this button to load the case or to apply a filter

Advanced Toggle

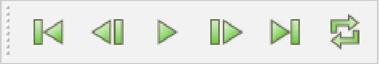
3D View/Canvas

Running my first OpenFOAM® case setup blindfold

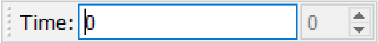
Crash introduction to paraFoam – Toolbars



- Main Controls

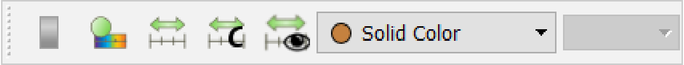


- VCR Controls (animation controls)



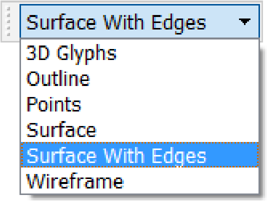
Time: 0 0

- Current Time Controls



Solid Color

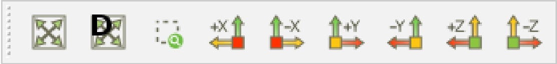
- Active Variable Controls



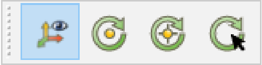
Surface With Edges

3D Glyphs
Outline
Points
Surface
Surface With Edges
Wireframe

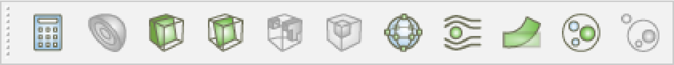
- Representation Toolbar



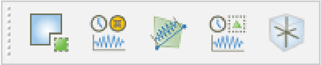
- Camera Controls (view orientation)



- Center Axes Controls



- Common Filters



- Data Analysis Toolbar

Crash introduction to paraFoam – Mesh visualization

Select `Surface With Edges` in the Representation Toolbar

Fit to screen

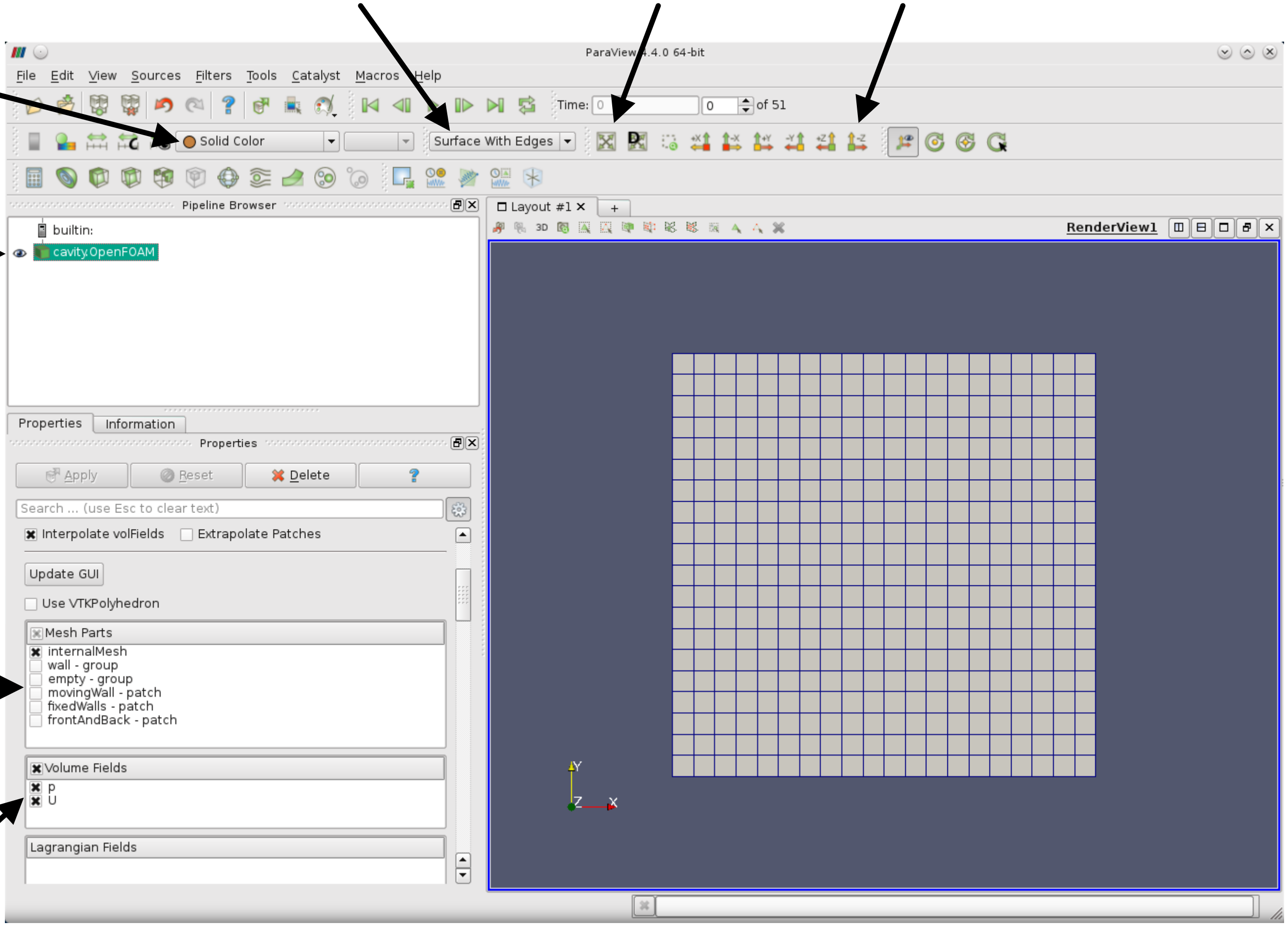
Select the -Z view

Select `Solid Color` in the Active Variable Controls

Click on the eyeball in the Pipeline Browser to hide/unhide the object

Select mesh parts to visualize. By default it will automatically select `internalMesh`

Select the volume fields to visualize. By default it will select **U** and **p**



☒ Interpolate volFields ☐ Extrapolate Patches

☐ Use VTKPolyhedron

Mesh Parts
- ☒ internalMesh
- ☐ wall - group
- ☐ empty - group
- ☐ movingWall - patch
- ☐ fixedWalls - patch
- ☐ frontAndBack - patch

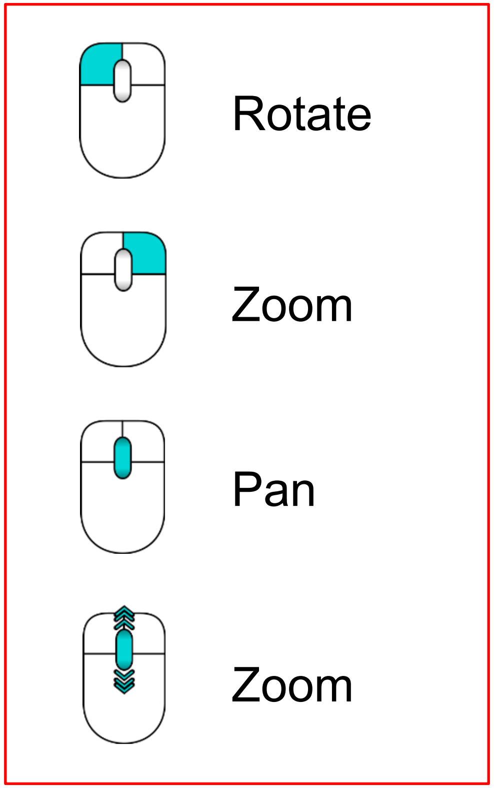
☒ Volume Fields
- ☒ p
- ☒ U

Lagrangian Fields

Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – 3D View and mouse interaction

Mouse interaction in the 3D view



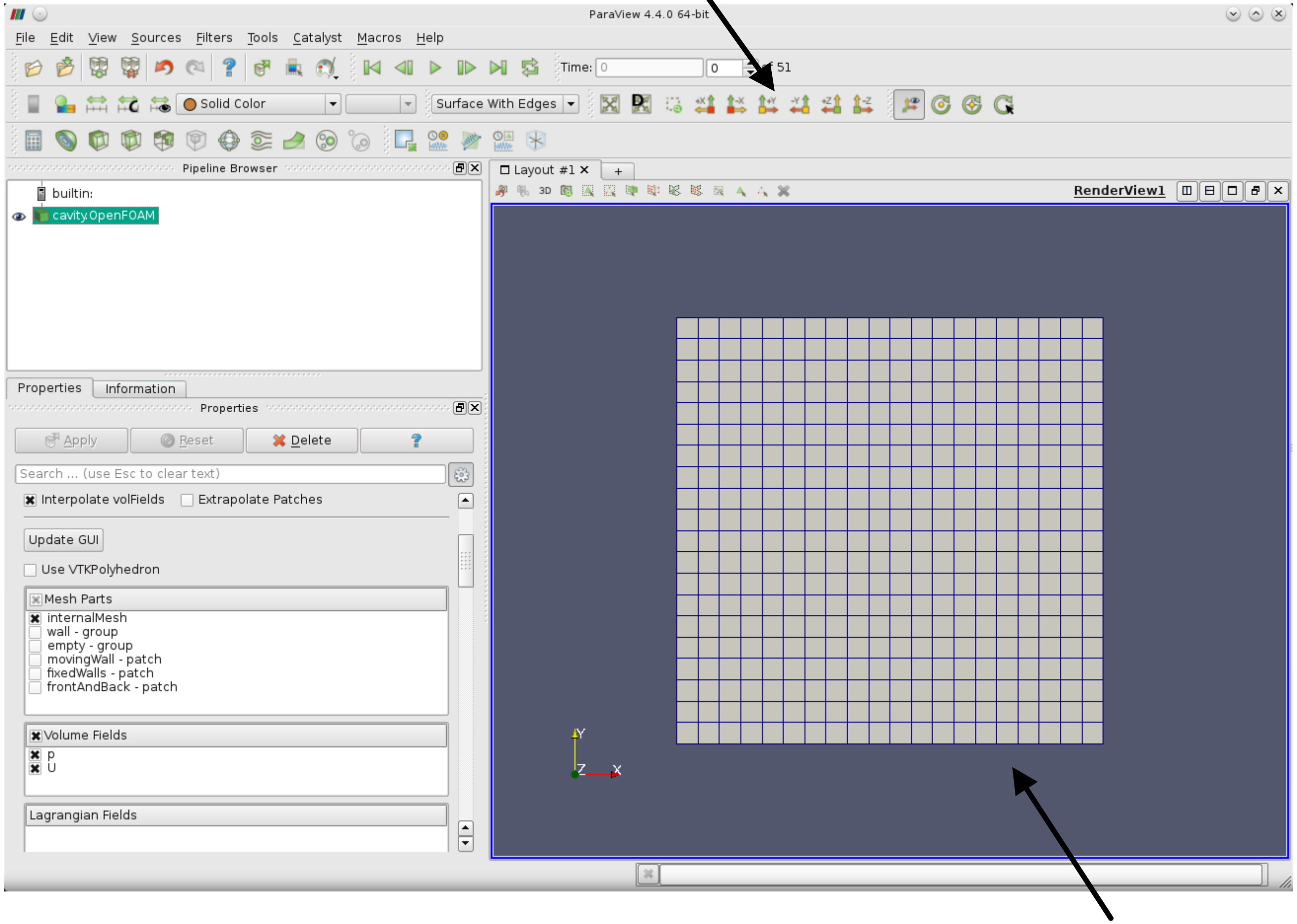
Rotate

Zoom

Pan

Zoom

Select view orientation in the Camera Controls



Time: 0 | 0 | of 51

Solid Color

Surface With Edges

builtin:

cavity.OpenFOAM

Properties

Search ... (use Esc to clear text)

☒ Interpolate volFields ☐ Extrapolate Patches

☐ Use VTKPolyhedron

☐ Mesh Parts
☒ internalMesh
☐ wall - group
☐ empty - group
☐ movingWall - patch
☐ fixedWalls - patch
☐ frontAndBack - patch

☒ Volume Fields
☒ p
☒ U

Lagrangian Fields

3D View/Canvas

Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – Fields visualization

Select `Last Frame` in the VCR Controls

Current Time Controls

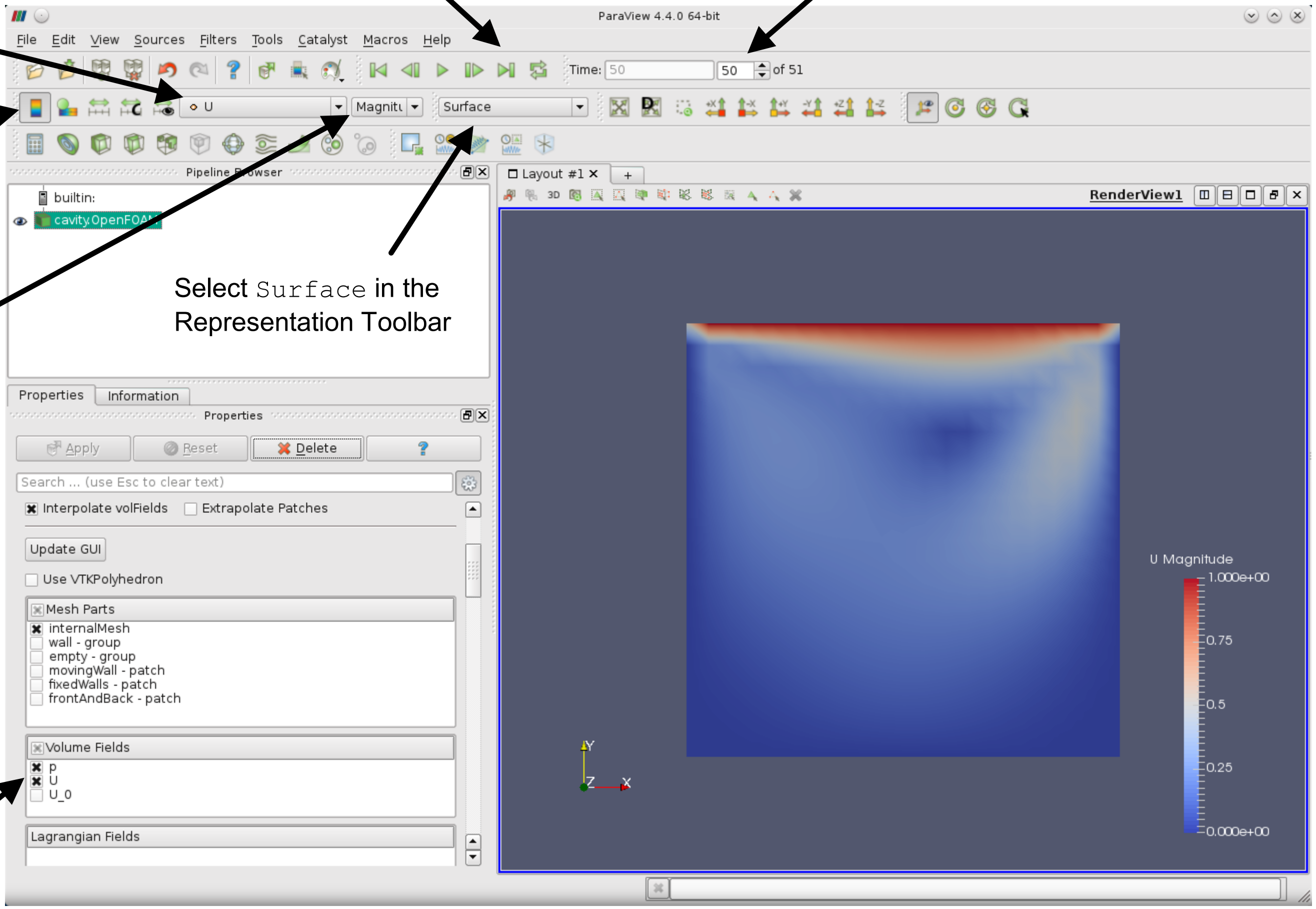
Select **U** in Active Variable Controls

Turn on/off color bar

Select `Surface` in the Representation Toolbar

Select `Magnitude` in the drop down menu

Select volume fields to visualize. By default it will select **U** and **p.**



Time: 50 | 50 of 51

○ U | Magnitu | Surface

Search ... (use Esc to clear text)

☒ Interpolate volFields ☐ Extrapolate Patches

☐ Use VTKPolyhedron

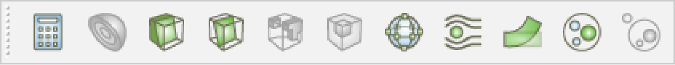
☒ Mesh Parts
- ☒ internalMesh
- ☐ wall - group
- ☐ empty - group
- ☐ movingWall - patch
- ☐ fixedWalls - patch
- ☐ frontAndBack - patch

☒ Volume Fields
- ☒ p
- ☒ U
- ☐ U_0

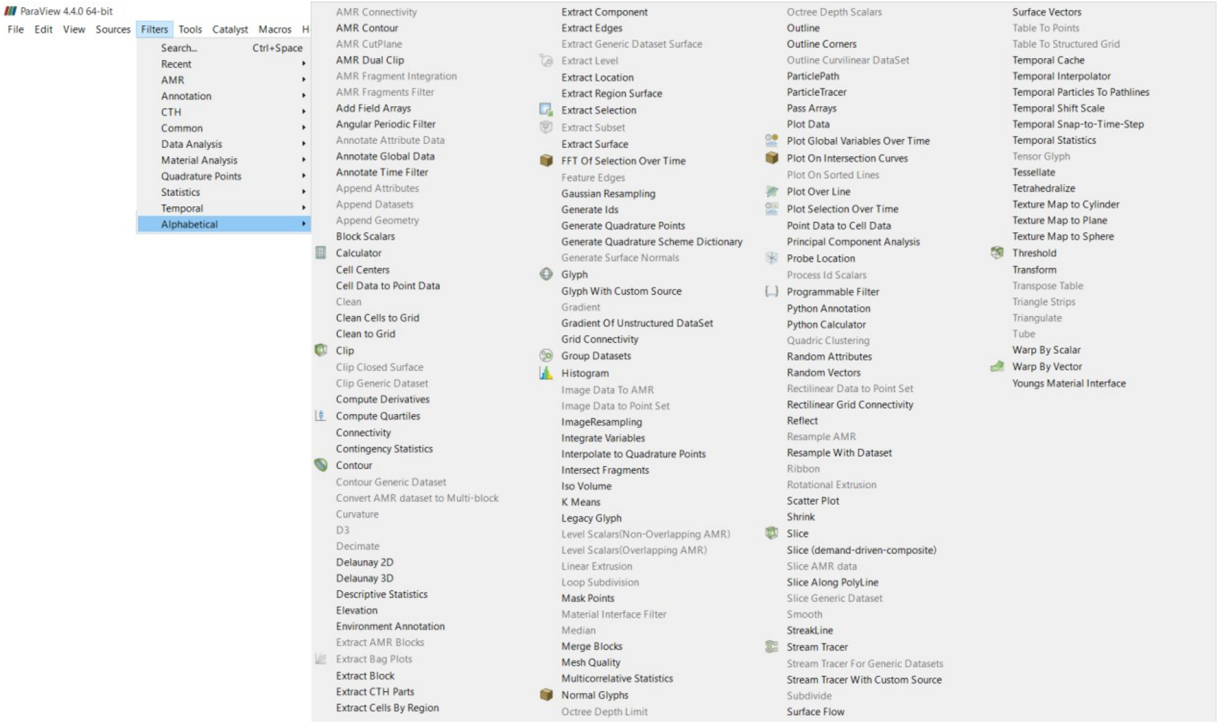
Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – Filters

- Filters are functions that generate, extract or derive features from the input data.
- They are attached to the input data.
- You can access the most commonly used filters from the `Common Filters` toolbar



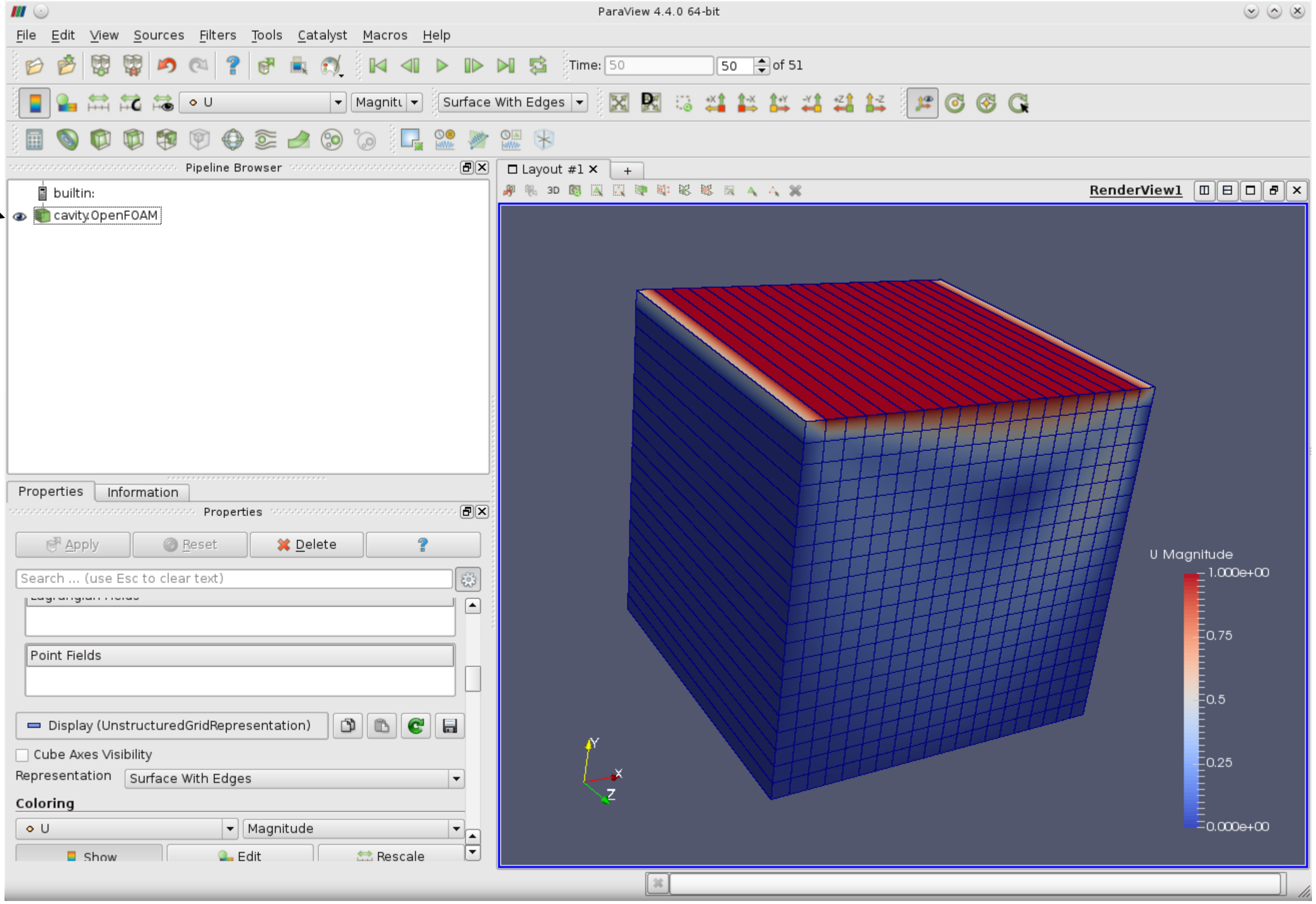
- You can access all the filters from the menu `Filter`.



Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – Filters

Filters are attached to the input data



- Even if the case is 2D, it will be visualized as if it were a 3D case.
- Notice that there is only one cell in the **Z** direction.
- Let us use the slice filter. This filter will create a cut plane.
- Let us create a slice normal to the **Z** direction.

Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – Slice filter

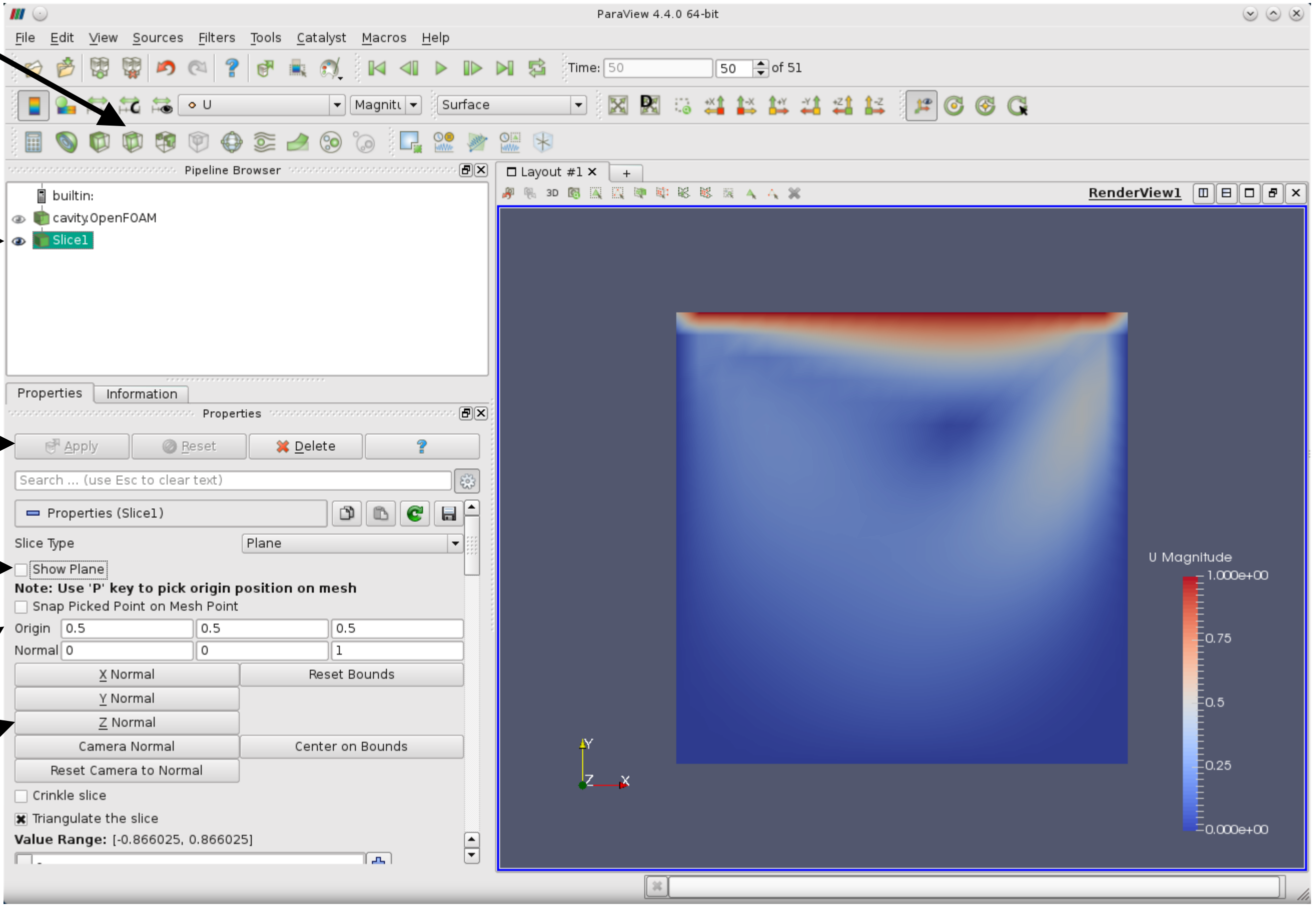
1. Select the `Slice` filter

If you want to erase a filter, right click on it and select `Delete`

4. Press `Apply`

3. Optional - Turn off the option `Show Plane`

2. Select the direction `Z Normal`. Additionally you can choose the origin of the plane (by default is the mid section)



Slice Type: Plane

☐ Show Plane

Note: Use 'P' key to pick origin position on mesh

☐ Snap Picked Point on Mesh Point

Origin 0.5 0.5 0.5

Normal 0 0 1

☐ Crinkle slice

☒ Triangulate the slice

Value Range: [-0.866025, 0.866025]

Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – Glyph filter

1. Select the `Glyph` filter. This filter will be applied on the `Slice1` filter

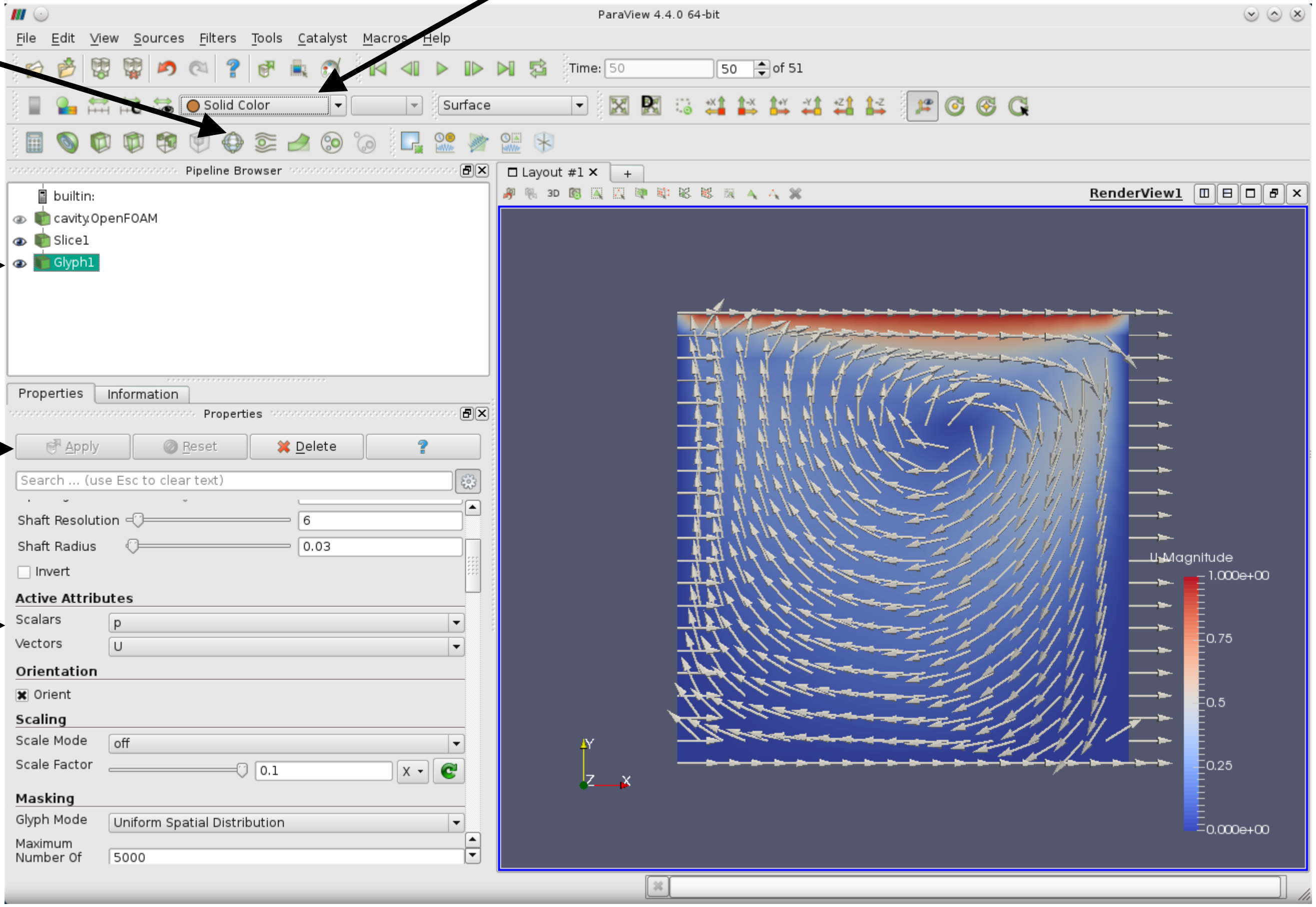
Notice that the filter `Glyph` was applied on the `Slice1` filter.

3. Press `Apply`

2. Filter options

Notice that the vectors are plotted in the cell vertices. To plot the vectors at the cell centers, use the filter `cell centers` and replot the vectors.

4. Color the colors using Solid Color



Properties:
- Shaft Resolution: 6
- Shaft Radius: 0.03
- □ Invert

Active Attributes
- Scalars: p
- Vectors: U

Orientation
- ☒ Orient

Scaling
- Scale Mode: off
- Scale Factor: 0.1

Masking
- Glyph Mode: Uniform Spatial Distribution
- Maximum Number Of: 5000

Running my first OpenFOAM® case setup blindfold

Crash introduction to paraFoam – Plot Over Line filter

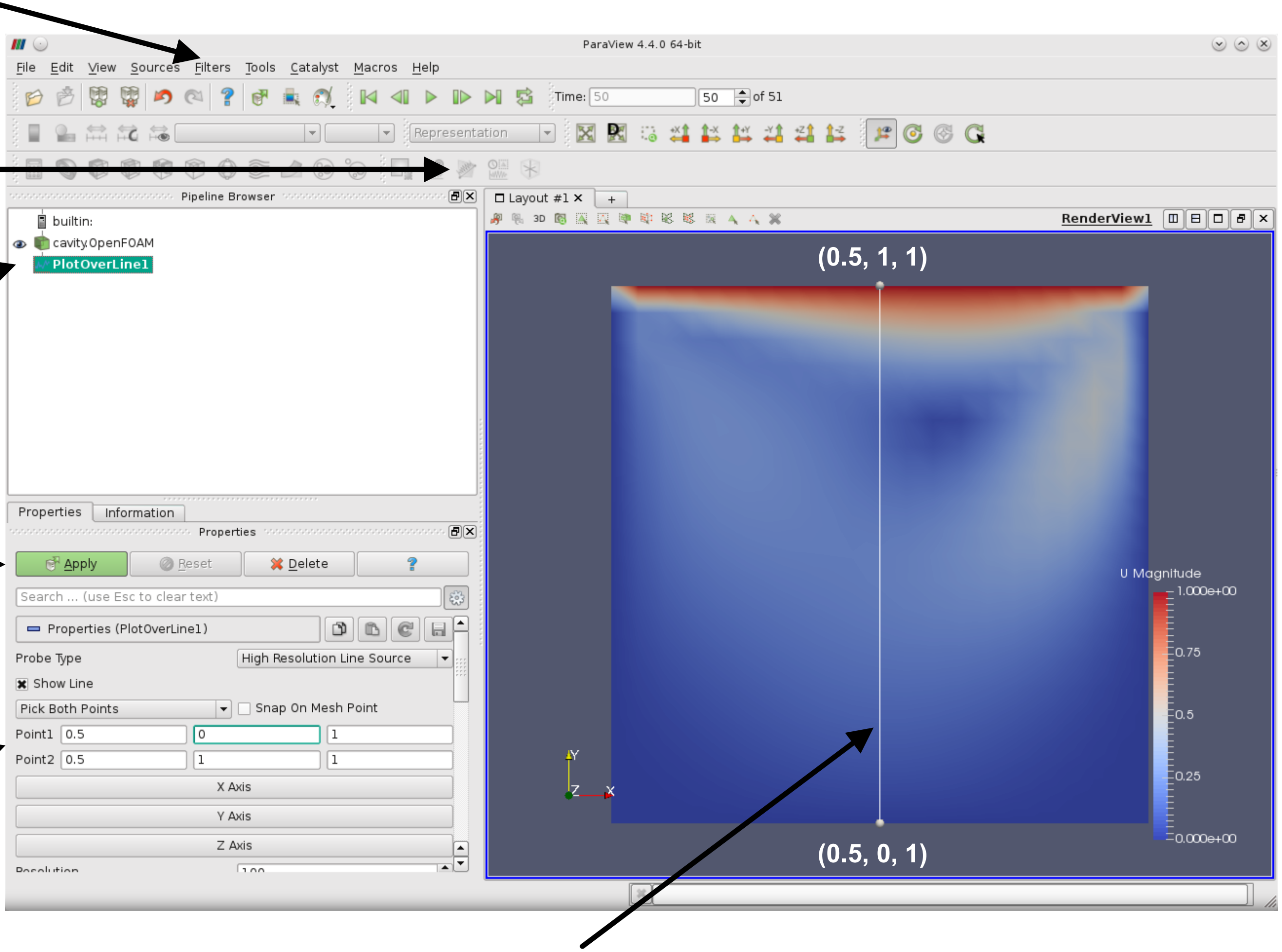
1.a. Select the `Plot Over Line` filter.

1.b. Alternative, you can select `Plot Over Line` filter from the Data Analysis Toolbar

Notice that we are using the filter in a clean `Pipeline`

3. Press `Apply`

2. Enter the coordinates of the line



Pipeline Browser: builtin:, cavity.OpenFOAM, PlotOverLine1

Properties (PlotOverLine1)

Probe Type: High Resolution Line Source

☒ Show Line

Pick Both Points

□ Snap On Mesh Point

Point1: 0.5, 0, 1

Point2: 0.5, 1, 1

Time: 50, 50 of 51

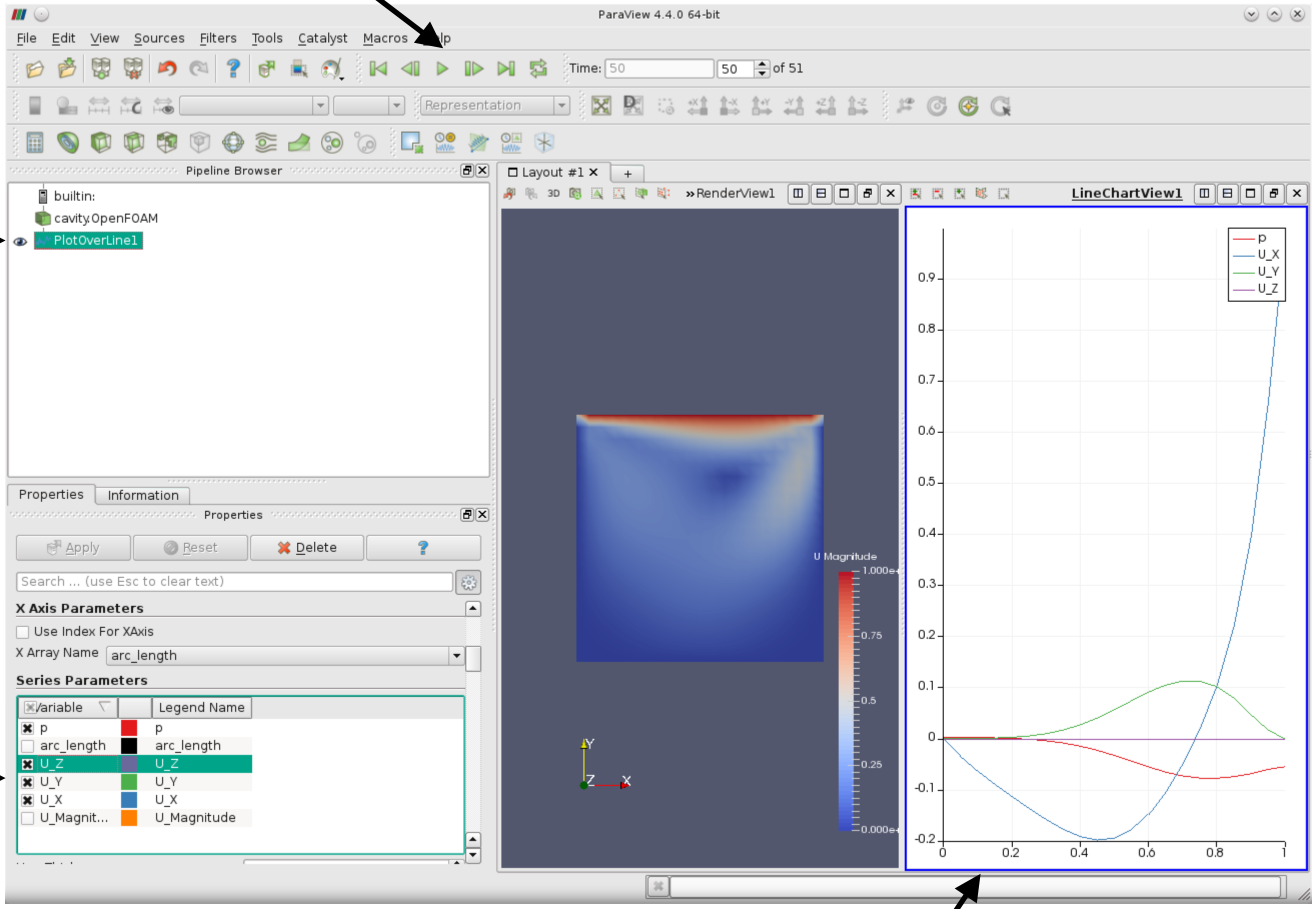
Line

Crash introduction to paraFoam – Filters

4. Optional – Use the VCR Control to change the frame.
The line chart view will be updated automatically

3. Optional - To save the sampled data in CSV format, click on the filter. Then click on the `File` menu and select the option `Save Data`

2. Select the variables to plot in the line chart view



Time: 50 | 50 of 51

X Axis Parameters

☐ Use Index For XAxis

X Array Name: arc_length

Series Parameters

☒ Variable		Legend Name
☒ p	🟥	p
☐ arc_length	⬛	arc_length
☒ U_Z	🟪	U_Z
☒ U_Y	🟩	U_Y
☒ U_X	🟦	U_X
☐ U_Magnit...	🟧	U_Magnitude

1. Click on the line chart view (the blue frame indicates that it is the active view)

Running my first OpenFOAM® case setup blindfold

Running the case blindfold with log files

- In the previous case, we ran the simulation but we did not save the standard output stream (stdout) in a `log` file.

- We just saw the information on-the-fly.

- Our advice is to always save the standard output stream (stdout) in a `log` file.

- It is of interest to always save the `log` as if something goes wrong and you would like to do troubleshooting, you will need this information.

- Also, if you are interested in plotting the residuals you will need the `log` file.

- By the way, if at any point you ask us what went wrong with your simulation, it is likely that we will ask you for this file.

- We might also ask for the standard error stream (stderr).

Running my first OpenFOAM® case setup blindfold

Running the case blindfold with log files

- There are many ways to save the *log* files.
- From now on, we will use the Linux `tee` command to save *log* files.
- To save a *log* file of the simulation or the output of any utility, you can proceed as follows:

```
1.  $> foamCleanTutorials
2.  $> blockMesh | tee log.blockMesh
3.  $> checkMesh | tee log.checkMesh
4.  $> icoFoam | tee log.icoFoam
```

The vertical bar or pipelining operator is used to concatenate commands

- You can use your favorite text editor to read the log file (e.g., gedit, vi, emacs).

Running my first OpenFOAM® case setup blindfold

Running the case blindfold with log files

- In step 1 we erase the mesh and all the folders, except for **`0`**, **`constant`** and **`system`**. This script comes with your OpenFOAM® installation.
- In step 2, we generate the mesh using the meshing tool `blockMesh`. We also redirect the standard output to an ascii file with the name *`log.blockMesh`* (it can be any name). The `tee` command will redirect the screen output to the file *`log.blockMesh`* and at the same time will show you the information on the screen.
- In step 3 we check the mesh quality. We also redirect the standard output to an ascii file with the name *`log.checkMesh`* (it can be any name).
- In step 4 we run the simulation. We also redirect the standard output to an ascii file with the name *`log.icoFoam`* (it can be any name). Remember, the `tee` command will redirect the screen output to the file *`log.icoFoam`* and at the same time will show you the information on the screen.
- To postprocess the information contained in the solver log file *`log.icoFoam`*, we can use the utility `foamLog`. Type in the terminal:
  - `$> foamLog log.icoFoam`
- This utility will extract the information inside the file *`log.icoFoam`*. The extracted information is saved in an editable/plottable format in the directory **`logs`**.
- At this point we can use `gnuplot` to plot the residuals. Type in the terminal:
  - `$> gnuplot`

Running my first OpenFOAM® case setup blindfold

Running the case blindfold with log files

- To plot the information extracted with `foamLog` using gnuplot, we can proceed as follows (remember, at this point we are using the gnuplot prompt):

1.
```
gnuplot> set logscale y
```
Set log scale in the y axis

2.
```
gnuplot> plot ‘logs/p_0’ using 1:2 with lines
```
Plot the file p_0 located in the directory logs, use columns 1 and 2 in the file p_0, use lines to output the plot.

3.
```
gnuplot> plot ‘logs/p_0’ using 1:2 with lines, ‘logs/pFinalRes_0’ using 1:2 with lines
```
Here we are plotting to different files. You can concatenate files using comma (,)

4.
```
gnuplot> reset
```
To reset the scales

5.
```
gnuplot> plot ‘logs/CourantMax_0’ u 1:2 w l
```
To plot file CourantMax_0. The letter u is equivalent to using. The letters w l are equivalent to with lines

6.
```
gnuplot> set logscale y
```

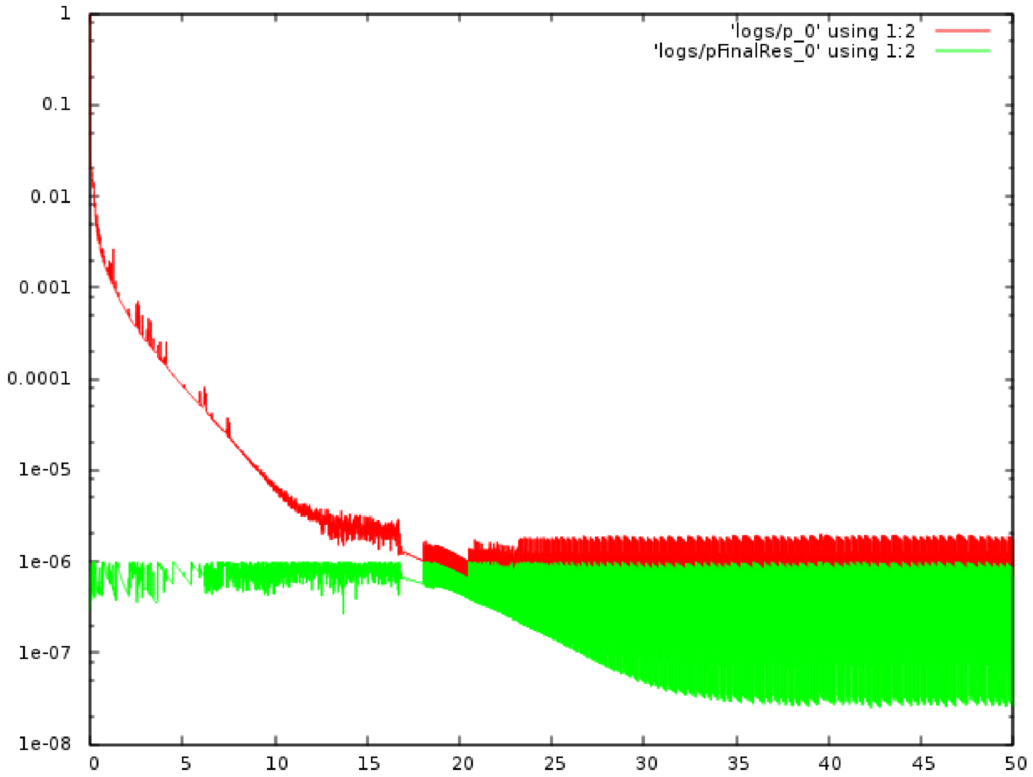
7.
```
gnuplot> plot [30:50][] ‘logs/Ux_0’ u 1:2 w l title ‘Ux’,‘logs/Uy_0’ u 1:2 w l title ‘Uy’
```
Set the x range from 30 to 50 and plot tow files and set legend titles

8.
```
gnuplot> exit
```
To exit gnuplot

Running my first OpenFOAM® case setup blindfold

Running the case blindfold with log files

- The output of step 3 is the following:



- The fact that the initial residuals (red line) are dropping to the same value of the final residuals (monotonic convergence), is a clear indication of a steady behavior.

Running my first OpenFOAM® case setup blindfold

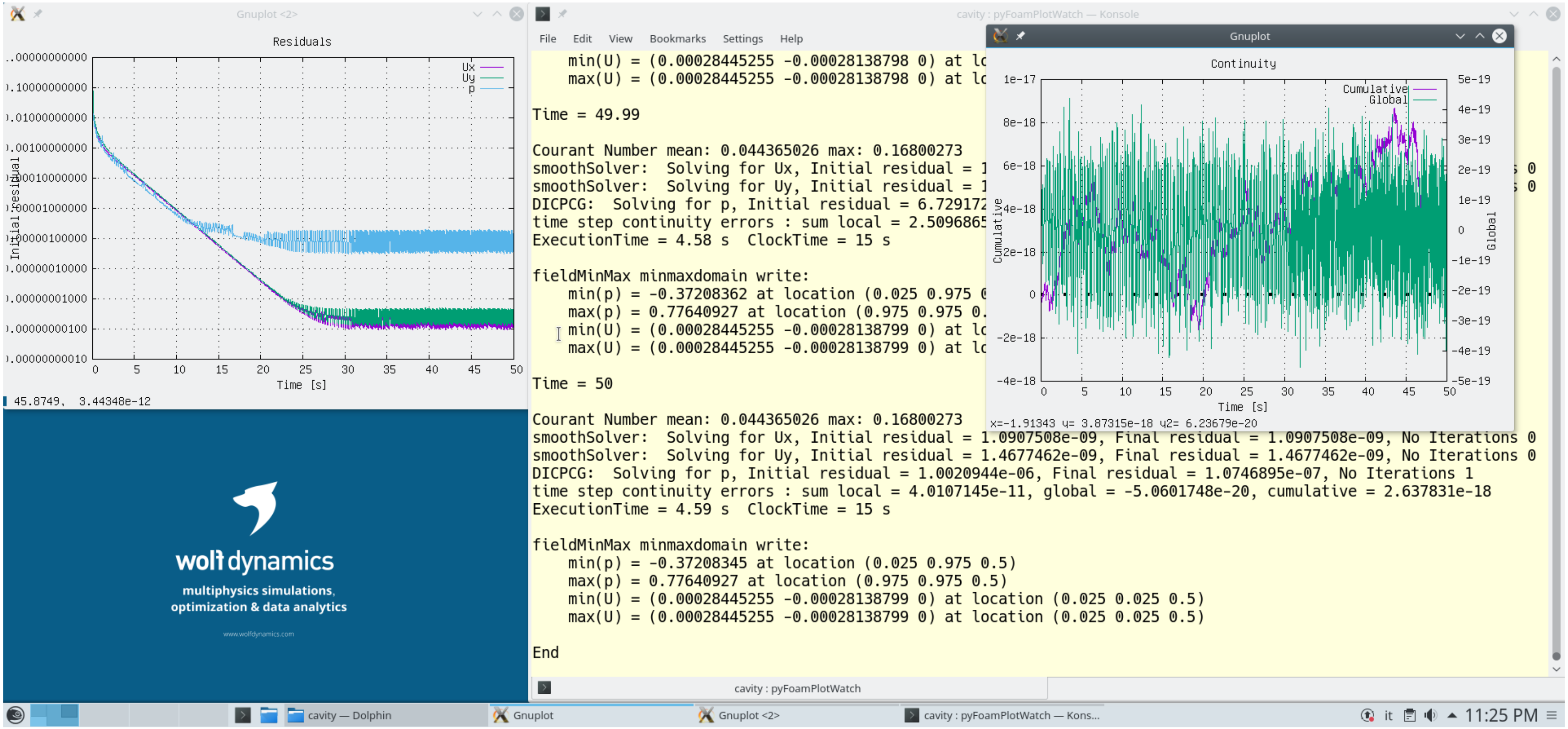
Running the case blindfold with log files and plotting the residuals

- It is also possible to plot the *log* information on the fly.
- The easiest way to do this is by using PyFoam (you will need to install it):
  - `$> pyFoamPlotRunner.py [options] <foamApplication>`
- If you are using the lab workstations, you will need to source PyFoam. To source PyFoam, type in the terminal:
  - `$> anaconda3`
- If you need help or want to know all the options available,
  - `$> pyFoamPlotRunner.py --help`
- To run this case with `pyFoamPlotRunner.py`, in the terminal type:
  - `$> pyFoamPlotRunner.py icoFoam`
- If you do not feel comfortable using `pyFoamPlotRunner.py` to run the solver, it is also possible to plot the information saved in the *log* file using PyFoam.
- To do so you will need to use the utility `pyFoamPlotWatcher.py`. For example,
  - `$> icoFoam | tee log.icoFoam`
- Then, in a new terminal window launch `pyFoamPlotWatcher`, as follows,
  - `$> pyFoamPlotWatcher.py log.icoFoam`
- You can also use `pyFoamPlotWatcher.py` to plot the information saved in an old *log* file.

Running my first OpenFOAM® case setup blindfold

Running the case blindfold with log files and plotting the residuals

- This is a screenshot on my computer. In this case, `pyFoamPlotRunner` is plotting the initial residuals and continuity errors on the fly.



```
    min(U) = (0.00028445255 -0.00028138798 0) at lo
    max(U) = (0.00028445255 -0.00028138798 0) at lo

Time = 49.99

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1
smoothSolver:  Solving for Uy, Initial residual = 1
DICPCG:  Solving for p, Initial residual = 6.729172
time step continuity errors : sum local = 2.5096865
ExecutionTime = 4.58 s  ClockTime = 15 s

fieldMinMax minmaxdomain write:
    min(p) = -0.37208362 at location (0.025 0.975 0
    max(p) = 0.77640927 at location (0.975 0.975 0.
    min(U) = (0.00028445255 -0.00028138799 0) at lo
    max(U) = (0.00028445255 -0.00028138799 0) at lo

Time = 50

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1.0907508e-09, Final residual = 1.0907508e-09, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.4677462e-09, Final residual = 1.4677462e-09, No Iterations 0
DICPCG:  Solving for p, Initial residual = 1.0020944e-06, Final residual = 1.0746895e-07, No Iterations 1
time step continuity errors : sum local = 4.0107145e-11, global = -5.0601748e-20, cumulative = 2.637831e-18
ExecutionTime = 4.59 s  ClockTime = 15 s

fieldMinMax minmaxdomain write:
    min(p) = -0.37208345 at location (0.025 0.975 0.5)
    max(p) = 0.77640927 at location (0.975 0.975 0.5)
    min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
    max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)

End
```

Running my first OpenFOAM® case setup blindfold

Stopping the simulation

- Your simulation will automatically stop at the time value you set using the keyword **endTime** in the *controlDict* dictionary.

endTime 50;

- If for any reason you want to stop your simulation before reaching the value set by the keyword **endTime**, you can change this value to a number lower than the current simulation time (you can use 0 for instance). This will stop your simulation, but it will not save your last time-step or iteration, so be careful.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     icoFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         50;
```

Running my first OpenFOAM® case setup blindfold

Stopping the simulation

- If you want to stop the simulation and save the solution, in the *controlDict* dictionary made the following modification,

stopAt writeNow;

This will stop your simulation and will save the current time-step or iteration.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     icoFoam;

startFrom       startTime;

startTime       0;

stopAt          writeNow;

endTime         50;
```

Running my first OpenFOAM® case setup blindfold

Stopping the simulation

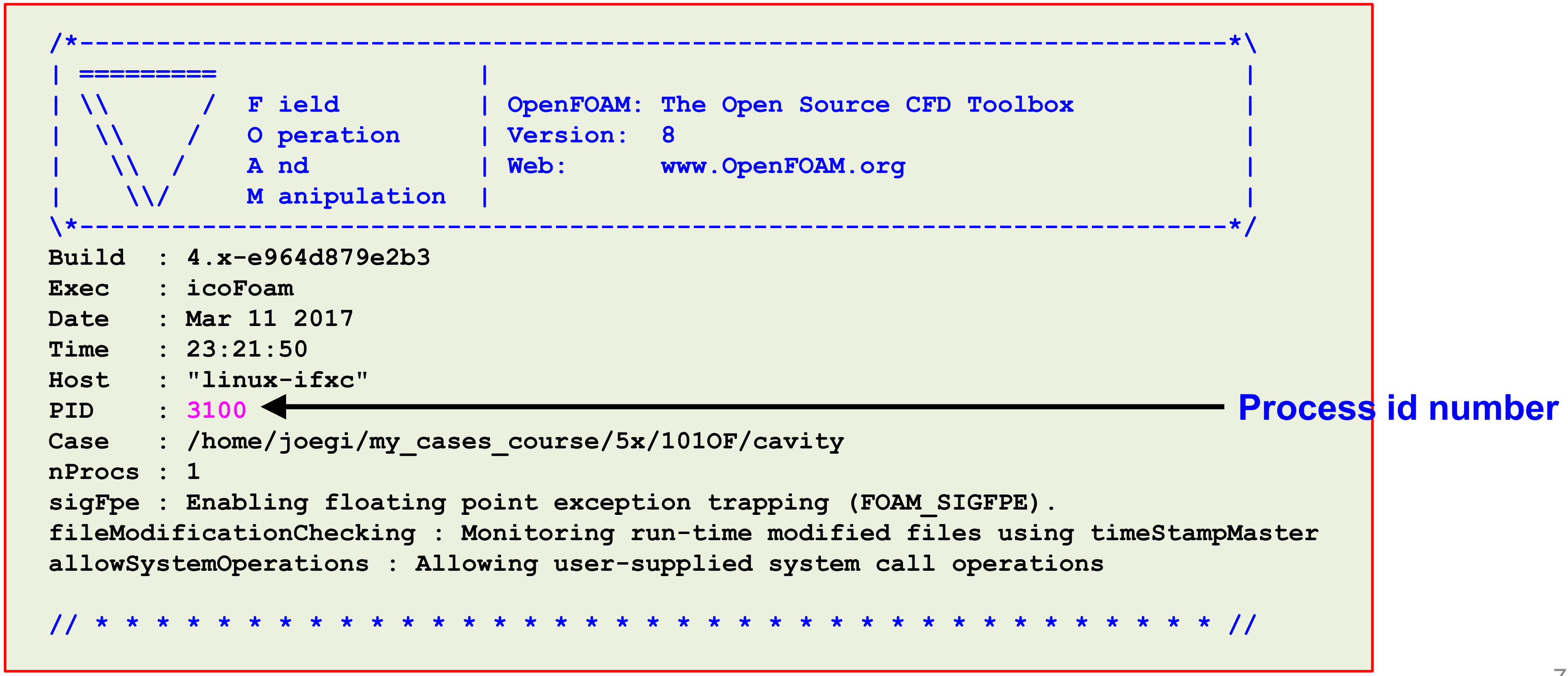
- The previous modifications can be done on-the-fly, but you will need to set the keyword **runTimeModifiable** to **true** in the *controlDict* dictionary.
- By setting the keyword **runTimeModifiable** to **true**, you will be able to modify most of the dictionaries on-the-fly.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}

runTimeModifiable true;
```

Running my first OpenFOAM® case setup blindfold

Stopping the simulation

- You can also kill the process. For instance, if you did not launch the solver in background, go to its terminal window and press `ctrl-c`. This will stop your simulation, but it will not save your last time-step or iteration, so be careful.
- If you launched the solver in background, just identify the process `id` using `top` or `htop` (or any other process manager) and terminate the associated process. Again, this will not save your last time-step or iteration.
- To identify the process `id` of the OpenFOAM® solver or utility, just read screen. At the beginning of the output screen, you will find the process `id` number.



```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build   : 4.x-e964d879e2b3
Exec    : icoFoam
Date    : Mar 11 2017
Time    : 23:21:50
Host    : "linux-ifxc"
PID     : 3100
Case    : /home/joegi/my_cases_course/5x/101OF/cavity
nProcs  : 1
sigFpe  : Enabling floating point exception trapping (FOAM_SIGFPE).
fileModificationChecking : Monitoring run-time modified files using timeStampMaster
allowSystemOperations : Allowing user-supplied system call operations

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

Process id number

Running my first OpenFOAM® case setup blindfold

Stopping the simulation

- When working locally, we usually proceed in this way:

  - `$> icoFoam | tee log.icofoam`

  This will run the solver `icoFoam` (by the way, this works for any solver or utility), it will save the standard output stream in the file *`log.icofoam`* and will show the solver output on the fly.

- If at any moment we want to stop the simulation, and we are not interested in saving the last time-step, we press `ctrl-c`.

- If we are interested in saving the last time step, we modify the *`controlDict`* dictionary and add the following keyword

  stopAt writeNow;

- Remember, this modification can be done on the fly. However, you will need to set the keyword **runTimeModifiable** to **yes** in the *`controlDict`* dictionary.

Running my first OpenFOAM® case setup blindfold

Cleaning the case folder

- If you want to erase the mesh and the solution in the current case folder, you can type in the terminal,

```
$> foamCleanTutorials
```

If you are running in parallel, this will also erase the **processorN** directories. We will talk about running in parallel later.

- If you are looking to only erase the mesh, you can type in the terminal,

```
$> foamCleanPolyMesh
```

- If you are only interested in erasing the saved solutions, in the terminal type,

```
$> foamListTimes -rm
```

- If you are running in parallel and you want to erase the solution saved in the **processorN** directories, type in the terminal,

```
$> foamListTimes -rm -processor
```

Roadmap

1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. ~~Directory structure of an application/utility~~
4. ~~Applications/utilities in OpenFOAM®~~
5. ~~Directory structure of an OpenFOAM® case~~
6. ~~Running my first OpenFOAM® case setup blindfold~~
7. **A deeper view to my first OpenFOAM® case setup**
8. 3D Dam break – Free surface flow
9. Flow past a cylinder – From laminar to turbulent flow

A deeper view to my first OpenFOAM® case setup

- We will take a close look at what we did by looking at the case files.
- The case directory originally contains the following sub-directories: **`0`**, **`constant`**, and **`system`**. After running `icoFoam` it also contains the time step directories **`1`**, **`2`**, **`3`**, `. . .`, **`48`**, **`49`**, **`50`**, the post-processing directory **`postProcessing`**, and the *`log.icoFoam`* file (if you chose to redirect the standard output stream).
  - The time step directories contain the values of all the variables at those time steps (the solution). The **`0`** directory is thus the initial condition and boundary conditions.
  - The **`constant`** directory contains the mesh and dictionaries for thermophysical, turbulence models and advanced physical models.
  - The **`system`** directory contains settings for the run, discretization schemes and solution procedures.
  - The **`postProcessing`** directory contains the information related to the **functionObjects** (we are going to address **functionObjects** later).
- The `icoFoam` solver reads these files and runs the case according to those settings.

A deeper view to my first OpenFOAM® case setup

- Before continuing, we want to point out the following:
  - Each dictionary file in the case directory has a header.
  - Lines 1-7 are commented.
  - You should always keep lines 8 to 14, if not, OpenFOAM® will complain.
  - According to the dictionary you are using, the **class** keyword (line 12) will be different. We are going to talk about this later on.
  - From now on and unless it is strictly necessary, we will not show the header when listing the dictionaries files.

```
1     /*--------------------------------*- C++ -*----------------------------------*\
2     | =========                 |                                                 |
3     | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4     |  \\    /   O peration     | Version:  8                                     |
5     |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6     |    \\/     M anipulation  |                                                 |
7     \*---------------------------------------------------------------------------*/
8     FoamFile
9     {
10        version     2.0;
11        format      ascii;
12        class       dictionary;
13        object      controlDict;
14    }
```

A deeper view to my first OpenFOAM® case setup

Let us explore the case directory

A deeper view to my first OpenFOAM® case setup

The **constant** directory

(and by the way, open each file and go thru its content)

- In this directory you will find the sub-directory **polyMesh** and the dictionary file *transportProperties*.
- The *transportProperties* file is a dictionary for the dimensioned scalar **nu**, or the kinematic viscosity.

```
17   nu        nu [ 0 2 -1 0 0 0 0 ] 0.01;      //Re 100
18   //nu       nu [ 0 2 -1 0 0 0 0 ] 0.001;    //Re 1000
```

- Notice that line 18 is commented.
- The values between square bracket are the units.
- OpenFOAM® is fully dimensional. You need to define the dimensions for each field dictionary and physical properties defined.
- Your dimensions shall be consistent.

A deeper view to my first OpenFOAM® case setup

Dimensions in OpenFOAM® (metric system)

No.	Property	Unit	Symbol
1	Mass	Kilogram	**kg**
2	Length	meters	**m**
3	Time	second	**s**
4	Temperature	Kelvin	**K**
5	Quantity	moles	**mol**
6	Current	ampere	**A**
7	Luminuous intensity	candela	**cd**

[1 (kg), 2 (m), 3 (s), 4 (K), 5 (mol), 6 (A), 7 (cd)]

A deeper view to my first OpenFOAM® case setup

The `constant` directory

(and by the way, open each file and go thru its content)

- Therefore, the dimensioned scalar **nu** or the kinematic viscosity,

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

has the following units

```
[ 0 m^2 s^-1 0 0 0 0 ]
```

Which is equivalent to

$$\nu = 0.01 \frac{m^2}{s}$$

A deeper view to my first OpenFOAM® case setup

The `constant` directory

(and by the way, open each file and go thru its content)

- In this case, as we are working with an incompressible flow, we only need to define the kinematic viscosity.

$$\nu = \frac{\mu}{\rho}$$

- Later on, we will ask you to change the Reynolds number, to do so you can change the value of **nu**. Remember,

$$Re = \frac{\rho \times U \times L}{\mu} = \frac{U \times L}{\nu}$$

- You can also change the free stream velocity U or the reference length L.

A deeper view to my first OpenFOAM® case setup

The **`constant`** directory

(and by the way, open each file and go thru its content)

- Depending on the physics involved and models used, you will need to define more variables in the dictionary *`transportProperties`*.
- For instance, for a multiphase case you will need to define the density **rho** and kinematic viscosity **nu** for each single phase. You will also need to define the surface tension σ.
- Also, depending of your physical model, you will find more dictionaries in the constant directory.
- For example, if you need to set gravity, you will need to create the dictionary *`g`*.
- If you work with compressible flows you will need to define the dynamic viscosity **mu**, and many other physical properties in the dictionary *`thermophysicalProperties`*.
- As we are not dealing with compressible flows (for the moment), we are not going into details.

A deeper view to my first OpenFOAM® case setup

The **`constant/polyMesh`** directory

(and by the way, open each file and go thru its content)

- In this case, the **`polyMesh`** directory is initially empty. After generating the mesh, it will contain the mesh in OpenFOAM® format.
- To generate the mesh in this case, we use the utility `blockMesh`. This utility reads the dictionary *`blockMeshDict`* located in the **`system`** folder.
- We will briefly address a few important inputs of the *`blockMeshDict`* dictionary.
- Do not worry, we are going to revisit this dictionary during the meshing session.
- However, have in mind that rarely you will use this utility to generate a mesh for complex geometries.
- Go to the directory **`system`** and open *`blockMeshDict`* dictionary with your favorite text editor, we will use gedit.

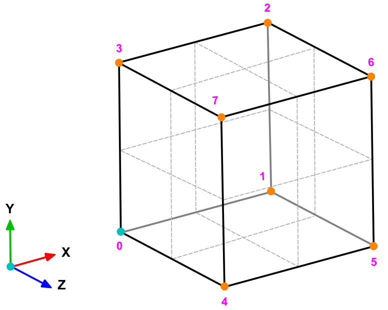
A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

- The *blockMeshDict* dictionary first defines a list with a number of vertices:

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
26      xcells 20;
27      ycells 20;
28      zcells 1;
29
37      vertices
38      (
39          ($xmin  $ymin  $zmin)      //vertex 0
40          ($xmax  $ymin  $zmin)      //vertex 1
41          ($xmax  $ymax  $zmin)      //vertex 2
42          ($xmin  $ymax  $zmin)      //vertex 3
43          ($xmin  $ymin  $zmax)      //vertex 4
44          ($xmax  $ymin  $zmax)      //vertex 5
45          ($xmax  $ymax  $zmax)      //vertex 6
46          ($xmin  $ymax  $zmax)      //vertex 7
47
48      /*
49          (0  0  0)
50          (1  0  0)
51          (1  1  0)
52          (0  1  0)
53          (0  0  0.1)
54          (1  0  0.1)
55          (1  1  0.1)
56          (0  1  0.1)
57      */
58      );
```

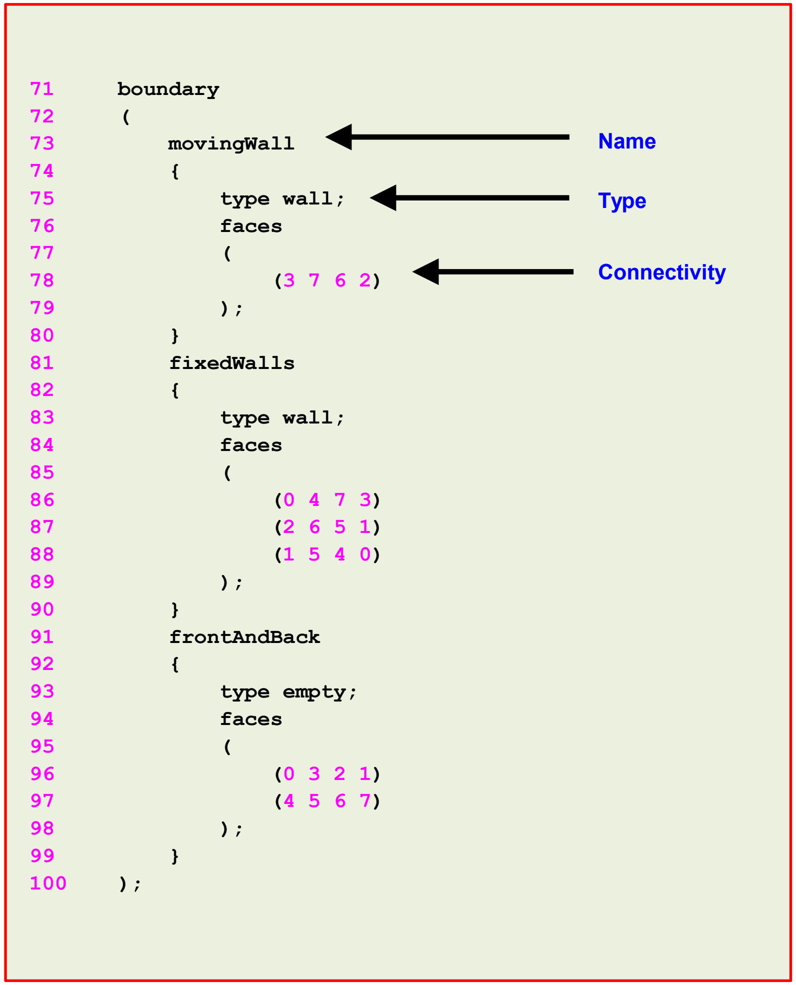
- The keyword **convertToMeters** (line 17), is a scaling factor. In this case we do not scale the dimensions.
- In the section vertices (lines 37-58), we define the vertices coordinates of the geometry. In this case, there are eight vertices defining the geometry. OpenFOAM® always uses 3D meshes, even if the simulation is 2D.
- We can directly define the vertex coordinates in the section vertices (commented lines 49-56), or we can use macro syntax.
- Using macro syntax we first define a variable and its value (lines 19-24), and then we can use them by adding the symbol **$** to the variable name (lines 39-46).
- In lines 26-28, we define a set of variables that will be used at a later time. These variables are related to the number of cells in each direction.
- Finally, notice that the vertex numbering starts from 0 (as the counters in c++). This numbering applies for blocks as well.



A deeper view to my first OpenFOAM® case setup

The `system/blockMeshDict` dictionary

- The `blockMeshDict` dictionary also defines the boundary patches:



```
71      boundary
72      (
73          movingWall
74          {
75              type wall;
76              faces
77              (
78                  (3 7 6 2)
79              );
80          }
81          fixedWalls
82          {
83              type wall;
84              faces
85              (
86                  (0 4 7 3)
87                  (2 6 5 1)
88                  (1 5 4 0)
89              );
90          }
91          frontAndBack
92          {
93              type empty;
94              faces
95              (
96                  (0 3 2 1)
97                  (4 5 6 7)
98              );
99          }
100     );
```

Name
Type
Connectivity

- In the section **boundary**, we define all the surface patches where we want to apply boundary conditions.
- This step is of paramount importance, because if we do not define the surface patches, we will not be able to apply the boundary conditions.
- For example:
  - In line 73 we define the patch name **movingWall** (the name is given by the user).
  - In line 75 we give a **base type** to the surface patch. In this case **wall** (do not worry we are going to talk about this later on).
  - In line 78 we give the connectivity list of the vertices that made up the surface patch or face, that is, **(3 7 6 2)**. Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.
- Remember, faces are defined by a list of 4 vertex numbers, e.g., **(3 7 6 2)**.

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

- To sum up, the *blockMeshDict* dictionary generates in this case a single block with:
  - **X/Y/Z** dimensions: **1.0**/**1.0**/**1.0**
  - Cells in the **X**, **Y** and **Z** directions: **20** x **20** x **1** cells.
  - One single **hex** block with straight lines.
  - Patch type **wall** and patch name **fixedWalls** at three sides.
  - Patch type **wall** and patch name **movingWall** at one side.
  - Patch type **empty** and patch name **frontAndBack** patch at two sides.

- If you are interested in visualizing the actual block topology, you can use `paraFoam` as follows,
  - `$> paraFoam -block`

A deeper view to my first OpenFOAM® case setup

🗎 The *system/blockMeshDict* dictionary

- As you can see, the *blockMeshDict* dictionary can be really tricky.

- If you deal with really easy geometries (rectangles, cylinders, and so on), then you can use `blockMesh` to do the meshing, but this is the exception rather than the rule.

- When using `snappyHexMesh`, (a body fitted mesher that comes with OpenFOAM®) you will need to generate a background mesh using `blockMesh`. We are going to deal with this later on.

- Our best advice is to create a template and reuse it.

- Also, take advantage of macro syntax for parametrization, and **#calc** syntax to perform inline calculations (lines 30-35 in the *blockMeshDict* dictionary we just studied).

- We are going to deal with **#codeStream** syntax and **#calc** syntax during the programming session.

A deeper view to my first OpenFOAM® case setup

🗎 The *`constant/polyMesh/boundary`* dictionary

- First of all, this file is automatically generated after you create the mesh using `blockMesh` or `snappyHexMesh`, or when you convert the mesh from a third-party format.
- In this file, the geometrical information related to the **base type** patch of each boundary (or surface patch) of the domain is specified.
- The **base type** boundary condition is the actual surface patch where we are going to apply a **numerical type** boundary condition (or numerical boundary condition).
- The **numerical type** boundary condition assign a field value to the surface patch (**base type**).
- We define the **numerical type** patch (or the value of the boundary condition), in the directory **0** or time directories.

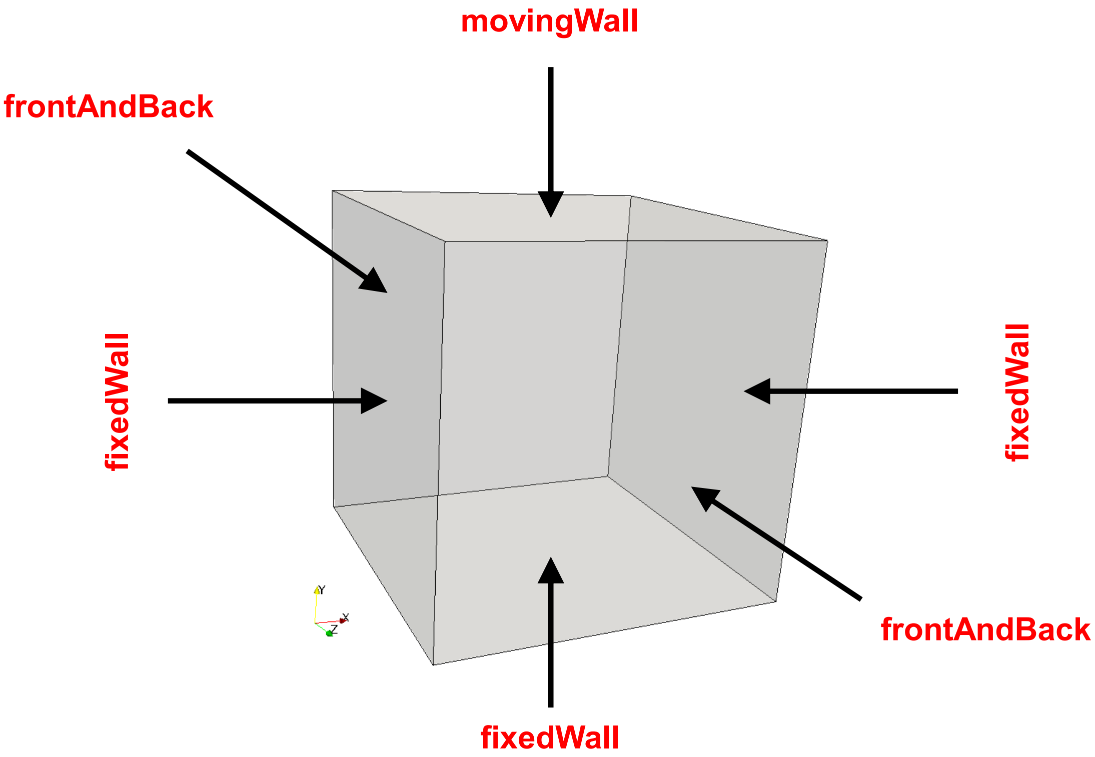
A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

- In this case, the file *boundary* is divided as follows

```
3
(
    movingWall
    {
        type                wall;
        inGroups            1(wall);
        nFaces              20;
        startFace           760;
    }
    fixedWalls
    {
        type                wall;
        inGroups            1(wall);
        nFaces              60;
        startFace           780;
    }
    frontAndBack
    {
        type                empty;
        inGroups            1(empty);
        nFaces              800;
        startFace           840;
    }
)
```

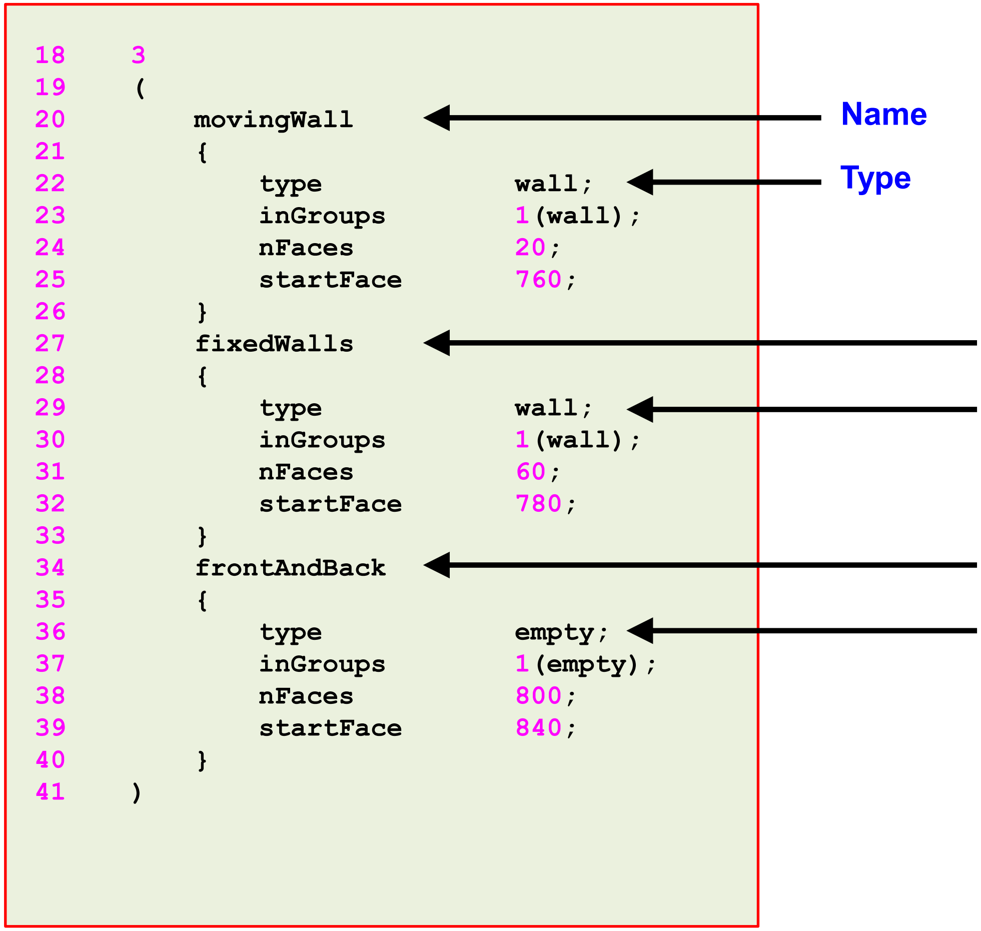
Number of surface patches
In the list bellow there must be 3 patches definition.



A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

- In this case, the file *boundary* is divided as follows



```
3
(
    movingWall
    {
        type            wall;
        inGroups        1(wall);
        nFaces          20;
        startFace       760;
    }
    fixedWalls
    {
        type            wall;
        inGroups        1(wall);
        nFaces          60;
        startFace       780;
    }
    frontAndBack
    {
        type            empty;
        inGroups        1(empty);
        nFaces          800;
        startFace       840;
    }
)
```

Name and type of the surface patches

- The name and type of the patch is given by the user.
- In this case the name and type was assigned in the dictionary *blockMeshDict*.
- You can change the name if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **movingWall** from **wall** to **patch**.
- When converting the mesh from a third party format, OpenFOAM® will try to recover the information from the original format. But it might happen that it does not recognizes the base type and name of the original file. In this case you will need to modify this file manually.

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

- In this case, the file *boundary* is divided as follows

```
3
(
    movingWall
    {
        type            wall;
        inGroups        1(wall);
        nFaces          20;
        startFace       760;
    }
    fixedWalls
    {
        type            wall;
        inGroups        1(wall);
        nFaces          60;
        startFace       780;
    }
    frontAndBack
    {
        type            empty;
        inGroups        1(empty);
        nFaces          800;
        startFace       840;
    }
)
```

inGroups keyword

- This keyword is optional. You can erase this information safely.
- It is used to group patches during visualization in ParaView/paraFoam. If you open this mesh in paraFoam you will see that there are two groups, namely: wall and empty.
- As usual, you can change the name.
- If you want to put a surface patch in two groups, you can proceed as follows:

  2(wall wall1)

  In this case the surface patch belongs to the groups **wall** and **wall1**.
- Groups can have more than one patch.

nFaces and startFace keywords

- Unless you know what you are doing, you do not need to modify this information. ⚠
- This information is related to the starting face and ending face of the boundary patch in the mesh data structure.
- This information is created automatically when generating the mesh or converting the mesh.

A deeper view to my first OpenFOAM® case setup

🗎 The *constant/polyMesh/boundary* dictionary

- There are a few **base type** patches that are constrained or paired. This means that the type should be the same in the *boundary* file and in the numerical boundary condition defined in the field files, *e.g.*, the files *0/U* and *0/p.*
- In this case, the **base type** of the patch **frontAndBack** (defined in the file *boundary*), is consistent with the **numerical type** patch defined in the field files *0/U* and *0/p*. They are of the type **empty**.
- Also, the **base type** of the patches **movingWall** and **fixedWalls** (defined in the file *boundary*), is consistent with the **numerical type** patch defined in the field files *0/U* and *0/p.*
- This is extremely important, especially if you are converting meshes as not always the type of the patches is set as you would like.
- Hence, it is highly advisable to do a sanity check and verify that the **base type** of the patches (the type defined in the file *boundary*), is consistent with the **numerical type** of the patches (the patch type defined in the field files contained in the directory **0** (or whatever time directory you defined the boundary and initial conditions).
- If the **base type** and **numerical type** boundary conditions are not consistent, OpenFOAM® will complain.
- Do not worry, we are going to address boundary conditions later on.

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

- The following **base type** boundary conditions are constrained or paired. That is, the type needs to be same in the *boundary* dictionary and field variables dictionaries (*e.g.* *U*, *p*).

constant/polyMesh/boundary	*0/U - 0/p (IC/BC)*
symmetry	**symmetry**
symmetryPlane	**symmetryPlane**
empty	**empty**
wedge	**wedge**
cyclic	**cyclic**
processor	**processor**

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

- The **base type patch** can be any of the **numerical** or **derived type** boundary conditions available in OpenFOAM®. Mathematically speaking; they can be Dirichlet, Neumann or Robin boundary conditions.

constant/polyMesh/boundary	*0/U - 0/p (IC/BC)*
patch	**fixedValue** **zeroGradient** **inletOutlet** **slip** **totalPressure** **supersonicFreeStream** and so on … Refer to the doxygen documentation for a list of all numerical type boundary conditions available.

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

- The **wall** base type boundary condition is defined as follows:

constant/polyMesh/boundary	*0/U (IC/BC)*	*0/p (IC/BC)*

wall	**type fixedValue;**
	value uniform (U V W);
		zeroGradient

- This boundary condition is not contained in the **patch** base type boundary condition group, because specialize modeling options can be used on this boundary condition.
- An example is turbulence modeling, where turbulence can be generated or dissipated at the walls.

A deeper view to my first OpenFOAM® case setup

🗎 The *constant/polyMesh/boundary* dictionary

- The name of the **base type** boundary condition and the name of the **numerical type** boundary condition needs to be the same, if not, OpenFOAM® will complain.
- Pay attention to this, specially if you are converting the mesh from another format.

constant/polyMesh/boundary	*0/U (IC/BC)*	*0/p (IC/BC)*
movingWall **fixedWalls** **frontAndBack**	**movingWall** **fixedWalls** **frontAndBack**	**movingWall** **fixedWalls** **frontAndBack**

- As you can see, all the names are the same across all the dictionary files.

A deeper view to my first OpenFOAM® case setup

The **system** directory

(and by the way, open each file and go thru its content)

- The **system** directory consists of the following compulsory dictionary files:
    - *controlDict*
    - *fvSchemes*
    - *fvSolution*
- *controlDict* contains general instructions on how to run the case.
- *fvSchemes* contains instructions for the discretization schemes that will be used for the different terms in the equations.
- *fvSolution* contains instructions on how to solve each discretized linear equation system.
- Do not worry, we are going to study in details the most important entries of each dictionary (the compulsory entries).
- If you forget a compulsory keyword or give a wrong entry to the keyword, OpenFOAM® will complain and it will let you what are you missing. This applies for all the dictionaries in the hierarchy of the case directory.
- There are many optional parameters, to know all of them refer to the doxygen documentation or the source code. Hereafter we will try to introduce a few of them.
- OpenFOAM® will not complain if you are not using optional parameters, after all, they are optional. However, if the entry you use for the optional parameter is wrong OpenFOAM® will let you know.

A deeper view to my first OpenFOAM® case setup

The `controlDict` dictionary

```
17    application       icoFoam;
18
19    startFrom         startTime;
20
21    startTime         0;
22
23    stopAt            endTime;
24
25    endTime           50;
26
27    deltaT            0.01;
28
29    writeControl      runTime;
30
31    writeInterval     1;
32
33    purgeWrite        0;
34
35    writeFormat       ascii;
36
37    writePrecision    8;
38
39    writeCompression off;
40
41    timeFormat        general;
42
43    timePrecision     6;
44
45    runTimeModifiable true;
```

- The `controlDict` dictionary contains runtime simulation controls, such as, start time, end time, time step, saving frequency and so on.
- Most of the entries are self-explanatory.
- This case starts from time 0 (keyword **startFrom** – line 19 – and keyword **startTime** – line 21 –). If you have the initial solution in a different time directory, just enter the number in line 21.
- The case will stop when it reaches the desired time set using the keyword **stopAt** (line 23).
- It will run up to 50 seconds (keyword **endTime** – line 25 –).
- The time step of the simulation is 0.01 seconds (keyword **deltaT** – line 27 –).
- It will write the solution every second (keyword **writeInterval** – line 31 –) of simulation time (keyword **runTime** – line 29 –).
- It will keep all the solution directories (keyword **purgeWrite** – line 33 –). If you want to keep only the last 5 solutions just change the value to 5.
- It will save the solution in ascii format (keyword **writeFormat** – line 35 –) with a precision of 8 digits (keyword **writePrecision** – line 37 –).
- And as the option **runTimeModifiable** (line 45) is on (**true**), we can modify all these entries while we are running the simulation.
- FYI, you can modify the entries on-the-fly for most of the dictionaries files.

A deeper view to my first OpenFOAM® case setup

The *controlDict* dictionary

```
17    application      icoFoam;
18
19    startFrom        startTime;
20
21    startTime        0;
22
23    stopAt           banana;
24
25    endTime          50;
26
27    deltaT           0.01;
28
29    writeControl     runTime;
30
31    writeInterval    1;
32
33    purgeWrite       0;
34
35    writeFormat      ascii;
36
37    writePrecision   8;
38
39    writeCompression off;
40
41    timeFormat       general;
42
43    timePrecision    6;
44
45    runTimeModifiable true;
```

- So how do we know what options are available for each keyword?
- The hard way is to refer to the source code.
- The easy way is to use the **banana method**.
- So what is the **banana method**? This method consist in inserting a dummy word (that does not exist in the installation) and let OpenFOAM® list the available options.
- For example. If you add **banana** in line 23, you will get this output:

```
banana is not in enumeration
4
(
nextWrite
writeNow
noWriteNow
endTime
)
```

- So your options are **nextWrite, writeNow, noWriteNow, endTime**
- And how do we know that banana does not exist in the source code? Just type in the terminal:
  - `$> src`
  - `$> grep -r -n banana .`
- If you see some bananas in your output someone is messing around with your installation.
- Remember, you can use any dummy word, but you have to be sure that it does not exist in OpenFOAM®.

A deeper view to my first OpenFOAM® case setup

The `controlDict` dictionary

```
17    application      icoFoam;
18
19    startFrom        startTime;
20
21    startTime        0;
22
23    stopAt           endTime;
24
25    //endTime          50;
26
27    deltaT           0.01;
28
29    writeControl     runTime;
30
31    writeInterval    1;
32
33    purgeWrite       0;
34
35    writeFormat      ascii;
36
37    writePrecision   8;
38
39    writeCompression off;
40
41    timeFormat       general;
42
43    timePrecision    6;
44
45    runTimeModifiable true;
```

- If you forget a compulsory keyword, OpenFOAM® will tell you what are you missing.
- So if you comment line 25, you will get this output:

```
--> FOAM FATAL IO ERROR
keyword endTime is undefined in dictionary …
```

- This output is just telling you that you are missing the keyword **endTime**.

- Do not pay attention to the words FATAL ERROR, maybe the developers of OpenFOAM® exaggerated a little bit.

A deeper view to my first OpenFOAM® case setup

The *fvSchemes* dictionary

```
ddtSchemes
{
    default         backward;
}

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

divSchemes
{
    default         none;
    div(phi,U)      Gauss linear;

    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    //default         Gauss linear orthogonal;
    default         Gauss linear limited 1;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    //default         orthogonal;
    default         limited 1;
}
```

- The *fvSchemes* dictionary contains the information related to the discretization schemes for the different terms appearing in the governing equations.
- As for the *controlDict* dictionary, the parameters can be changed on-the-fly.
- Also, if you want to know what options are available, just use the banana method.
- In this case we are using the **backward** method for time discretization (**ddtSchemes**). For gradients discretization (**gradSchemes**) we are using **Gauss linear** method. For the discretization of the convective terms (**divSchemes**) we are using **linear** interpolation for the term **div(phi,U)**.
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear** method with **limited 1** corrections (to handle mesh non-orthogonality and non-uniformity).
- The method we are using is second order accurate but oscillatory. We are going to talk about the properties of the numerical schemes later.
- Remember, at the end of the day we want a solution that is second order accurate.

A deeper view to my first OpenFOAM® case setup

The *fvSolution* dictionary

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;

    }

    pFinal
    {
        $p;
        relTol          0;
    }

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors      1;
    nNonOrthogonalCorrectors 0;
    pRefCell         0;
    pRefValue        0;
}
```

- The *fvSolution* dictionary contains the instructions of how to solve each discretized linear equation system. The equation solvers, tolerances, and algorithms are controlled from the sub-dictionary **solvers**.
- In the dictionary file *fvSolution* (and depending on the solver you are using), you will find the additional sub-dictionaries **PISO, PIMPLE, SIMPLE**, and **relaxationFactors**. These entries will be described later.
- As for the *controlDict and fvSchemes* dictionaries, the parameters can be changed on-the-fly.
- Also, if you want to know what options are available just use the banana method.
- In this case, to solve the pressure (**p**) we are using the **PCG** method, with the preconditioner **DIC**, an absolute **tolerance** equal to 1e-06 and a relative tolerance **relTol** equal to 0.
- The entry **pFinal** refers to the final pressure correction (notice that we are using macro syntax), and we are using a relative tolerance **relTol** equal to 0. We are putting more computational effort in the last iteration.
- In this case, we are using the same tolerances for **p** and **pFinal.** However, you can use difference tolerances, where usually you use a tighter tolerance in **pFinal**.

A deeper view to my first OpenFOAM® case setup

The *fvSolution* dictionary

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;

    }

    pFinal
    {
        $p;
        relTol          0;
    }

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors     1;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}
```

- To solve **U** we are using the **smoothSolver** method, with the smoother **symGaussSeidel**, an absolute **tolerance** equal to 1e-08 and a relative tolerance **relTol** equal to 0.
- The solvers will iterative until reaching any of the tolerance values set by the user or reaching a maximum value of iterations (optional entry).
- FYI, solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.
- The **PISO** sub-dictionary contains entries related to the pressure-velocity coupling method (the **PISO** method).
- In this case we are doing only one **PISO** correction and no orthogonal corrections.
- You need to do at least one **PISO** loop (**nCorrectors**).

A deeper view to my first OpenFOAM® case setup

The **`system`** directory

(optional dictionary files)

- In the **`system`** directory you will also find these two additional files:
  - *`decomposeParDict`*
  - *`sampleDict`*

- *`decomposeParDict`* is read by the utility `decomposePar`. This dictionary file contains information related to the mesh partitioning. This is used when running in parallel. We will address running in parallel later.
- *`sampleDict`* is read by the utility `postProcess`. This utility sample field data (points, lines or surfaces). In this dictionary file we specify the sample location and the fields to sample. The sampled data can be plotted using gnuplot or Python.

A deeper view to my first OpenFOAM® case setup

The *sampleDict* dictionary

```
type sets;

setFormat raw;

interpolationScheme cellPointFace;

fields
(
    U
);

sets
(

    l1
    {
        type                lineFace;
        axis                x;
        start               ( -1   0.5 0);
        end                 ( 2   0.5  0);
    }

    l2
    {
        type                lineFace;
        axis                y;
        start               (0.5 -1 0);
        end                 (0.5 2 0);
    }

);
```

Type of sampling, sets will sample along a line.

Format of the output file, raw format is a generic format that can be read by many applications. The output file is human readable (ascii format).

Interpolation method at the solution level (location of the interpolation points).

Fields to sample.

Sample method. How to interpolate the solution to the sample entity (line in this case)

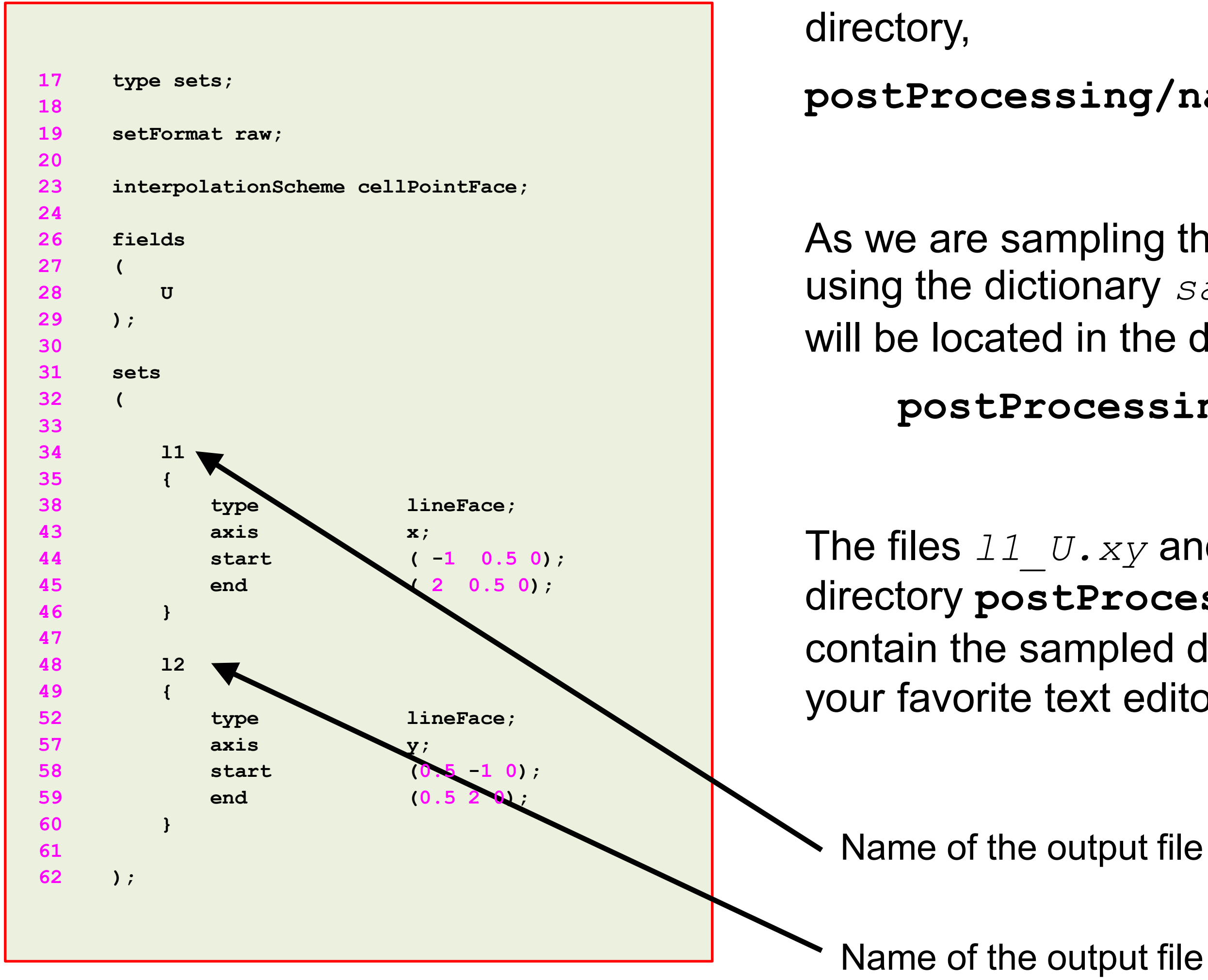
Location of the sample line. We define start and end point, and the axis of the sampling.

Sample method from the solution to the line.

Location of the sample line. We define start and end point, and the axis of the sampling.

A deeper view to my first OpenFOAM® case setup

The *sampleDict* dictionary



```
type sets;

setFormat raw;

interpolationScheme cellPointFace;

fields
(
    U
);

sets
(

    l1
    {
        type                lineFace;
        axis                x;
        start               ( -1  0.5 0);
        end                 ( 2  0.5 0);
    }

    l2
    {
        type                lineFace;
        axis                y;
        start               (0.5 -1 0);
        end                 (0.5 2 0);
    }

);
```

Name of the output file

Name of the output file

The sampled information is always saved in the directory,

postProcessing/name_of_input_dictionary

As we are sampling the latest time solution (50) and using the dictionary *sampleDict*, the sampled data will be located in the directory:

postProcessing/sampleDict/50

The files *l1_U.xy* and *l2_U.xy* located in the directory **postProcessing/sampleDict/50** contain the sampled data. Feel free to open them using your favorite text editor.

A deeper view to my first OpenFOAM® case setup

The 0 directory

(and by the way, open each file and go thru its content)

- The 0 directory contains the initial and boundary conditions for all primitive variables, in this case p and U. The U file contains the following information (velocity vector):

```
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;
    }
}
```

Dimensions of the field $\frac{m}{s}$

Uniform initial conditions.

The velocity field is initialize to **(0 0 0)** in all the domain

Remember velocity is a vector with three components, therefore the notation **(0 0 0)**.

Note:
If you take some time and compare the files *0/U* and *constant/polyMesh/boundary*, you will see that the name and type of each **numerical type** patch (the patch defined in *0/U*), is consistent with the **base type** patch (the patch defined in the file *constant/polyMesh/boundary*).

A deeper view to my first OpenFOAM® case setup

The 0 directory

(and by the way, open each file and go thru its content)

- The 0 directory contains the initial and boundary conditions for all primitive variables, in this case p and U. The U file contains the following information (velocity):

```
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;
    }
}
```

Dimensions of the field $\frac{m}{s}$

Numerical boundary condition for the patch **movingWall**

Numerical boundary condition for the patch **fixedWalls**

Numerical boundary condition for the patch **frontAndBack** (this is a constrained boundary condition).

A deeper view to my first OpenFOAM® case setup

The 0 directory

(and by the way, open each file and go thru its content)

- The 0 directory contains the initial and boundary conditions for all primitive variables, in this case p and U. The p file contains the following information (modified pressure):

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

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type            zeroGradient;
    }

    fixedWalls
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}

```

Dimensions of the field $\frac{m^2}{s^2}$

Uniform initial conditions.

The modified pressure field is initialize to 0 in all the domain. **This is relative pressure.**

Note:
If you take some time and compare the files *0/p* and *constant/polyMesh/boundary*, you will see that the name and type of each **numerical type** patch (the patch defined in *0/p*), is consistent with the **base type** patch (the patch defined in the file *constant/polyMesh/boundary*).

A deeper view to my first OpenFOAM® case setup

The 0 directory

(and by the way, open each file and go thru its content)

- The 0 directory contains the initial and boundary conditions for all primitive variables, in this case p and U. The p file contains the following information (modified pressure):

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

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type            zeroGradient;
    }

    fixedWalls
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}

```

Dimensions of the field $\frac{m^2}{s^2}$

Numerical boundary condition for the patch **movingWall**

Numerical boundary condition for the patch **fixedWalls**

Numerical boundary condition for the patch **frontAndBack** (this is a constrained boundary condition).

A deeper view to my first OpenFOAM® case setup

A very important remark on the pressure field ⚠

- We just used `icoFoam` which is an incompressible solver.
- **Let us be really loud on this.** All the incompressible solvers implemented in OpenFOAM® (`icoFoam`, `simpleFoam`, `pisoFoam`, and `pimpleFoam`), use the modified pressure, that is,

$$P = \frac{p}{\rho} \qquad \text{with units} \qquad \frac{m^2}{s^2}$$

- Or in OpenFOAM® jargon: `dimensions [0 2 -2 0 0 0 0]`
- So when visualizing or post processing the results **do not forget to multiply the pressure by the density** in order to get the right units of the physical pressure, that is,

$$\frac{kg}{m \cdot s^2}$$

- Or in OpenFOAM® jargon: `dimensions [1 -1 -2 0 0 0 0]`

A deeper view to my first OpenFOAM® case setup

- Coming back to the headers, and specifically the headers related to the field variable dictionaries (e.g. `U`, `p`, `gradU`, and so on).
- In the header of the field variables, the class type should be consistent with the type of field variable you are using.
- Be careful with this, specially if you are copying and pasting files.
- If the field variable is a scalar, the class should be **volScalarField**.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

A deeper view to my first OpenFOAM® case setup

- If the field variable is a vector, the class should be **volVectorField.**

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
```

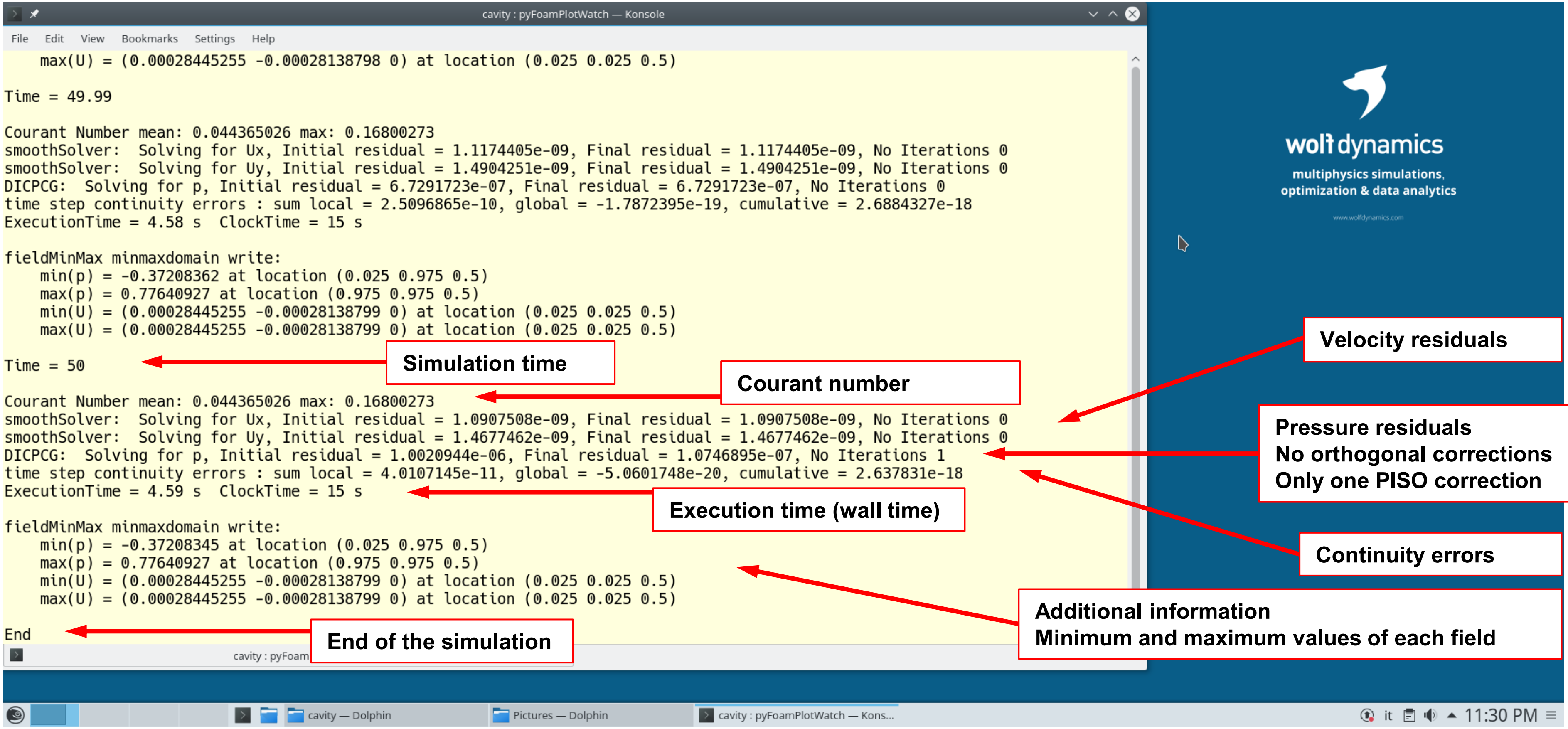
- If the field variable is a tensor (e.g. the velocity gradient tensor), the class should be **volTensorField**.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volTensorField;
    object      gradU;
}
```

A deeper view to my first OpenFOAM® case setup

The output screen

- Finally, let us talk about the output screen, which shows a lot of information.



```
    max(U) = (0.00028445255 -0.00028138798 0) at location (0.025 0.025 0.5)

Time = 49.99

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1.1174405e-09, Final residual = 1.1174405e-09, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.4904251e-09, Final residual = 1.4904251e-09, No Iterations 0
DICPCG:  Solving for p, Initial residual = 6.7291723e-07, Final residual = 6.7291723e-07, No Iterations 0
time step continuity errors : sum local = 2.5096865e-10, global = -1.7872395e-19, cumulative = 2.6884327e-18
ExecutionTime = 4.58 s  ClockTime = 15 s

fieldMinMax minmaxdomain write:
    min(p) = -0.37208362 at location (0.025 0.975 0.5)
    max(p) = 0.77640927 at location (0.975 0.975 0.5)
    min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
    max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)

Time = 50

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1.0907508e-09, Final residual = 1.0907508e-09, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.4677462e-09, Final residual = 1.4677462e-09, No Iterations 0
DICPCG:  Solving for p, Initial residual = 1.0020944e-06, Final residual = 1.0746895e-07, No Iterations 1
time step continuity errors : sum local = 4.0107145e-11, global = -5.0601748e-20, cumulative = 2.637831e-18
ExecutionTime = 4.59 s  ClockTime = 15 s

fieldMinMax minmaxdomain write:
    min(p) = -0.37208345 at location (0.025 0.975 0.5)
    max(p) = 0.77640927 at location (0.975 0.975 0.5)
    min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
    max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)

End
```

Simulation time

Courant number

Velocity residuals

Pressure residuals
No orthogonal corrections
Only one PISO correction

Execution time (wall time)

Continuity errors

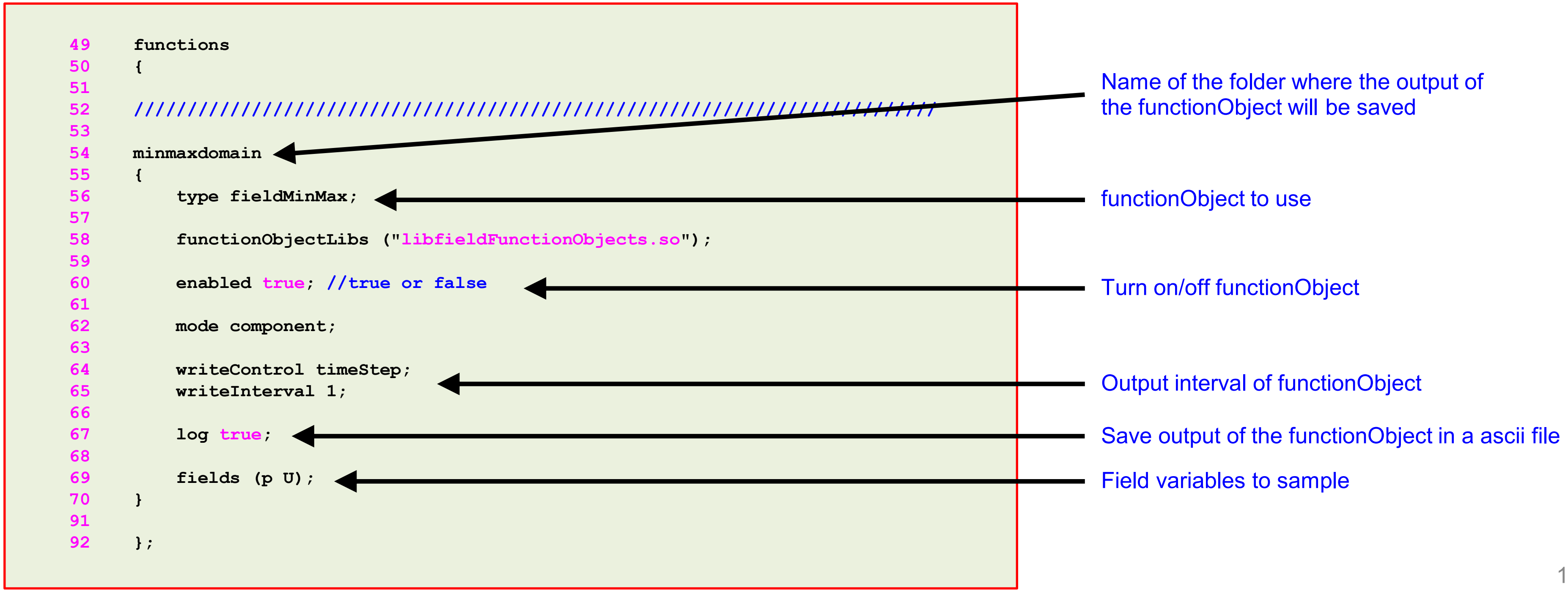
Additional information
Minimum and maximum values of each field

End of the simulation

A deeper view to my first OpenFOAM® case setup

The output screen

- By default, OpenFOAM® does not show the minimum and maximum information. To print out this information, we use **functionObjects**. We are going to address **functionObjects** in detail when we deal with post-processing and sampling.
- But for the moment, what we need to know is that we add **functionObjects** at the end of the `controlDict` dictionary. In this case, we are using a **functionObject** that prints the minimum and maximum information of the selected fields.
- This information complements the residuals information and it is saved in the **postProcessing** directory. It gives a better indication of stability, boundedness and consistency of the solution.



```
functions
{

////////////////////////////////////////////////////////////////////////////

minmaxdomain
{
    type fieldMinMax;

    functionObjectLibs ("libfieldFunctionObjects.so");

    enabled true; //true or false

    mode component;

    writeControl timeStep;
    writeInterval 1;

    log true;

    fields (p U);
}

};
```

Name of the folder where the output of the functionObject will be saved

functionObject to use

Turn on/off functionObject

Output interval of functionObject

Save output of the functionObject in a ascii file

Field variables to sample

A deeper view to my first OpenFOAM® case setup

The output screen

- Another very important output information is the CFL or Courant number.
- The Courant number imposes the **CFL number condition,** which is the maximum allowable CFL number a numerical scheme can use. For the n - dimensional case, the CFL number condition becomes,

$$CFL = \Delta t \sum_{i=1}^{n} \frac{u_i}{\Delta x_i} \leq CFL_{max}$$

- In OpenFOAM®, most of the solvers are implicit, which means they are **unconditionally stable**. In other words, they are not constrained to the **CFL number condition**.
- However, the fact that you are using a numerical method that is **unconditionally stable**, does not mean that you can choose a time step of any size.
- The time-step must be chosen in such a way that it resolves the time-dependent features, and it maintains the solver stability.
- For the moment and for the sake of simplicity, let us try to keep the CFL number below 5.0 and preferably close to 1.0 (for good accuracy).
- Other properties of the numerical method that you should observe are: conservationess, boundedness, transportiveness, and accuracy. We are going to address these properties and the CFL number when we deal with the FVM theory.

A deeper view to my first OpenFOAM® case setup

The output screen

- To control the CFL number you can change the time step or you can change the mesh.
- The easiest way is by changing the time step.
- For a time step of 0.01 seconds, this is the output you should get for this case,

```
Time = 49.99

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1.1174405e-09, Final residual = 1.1174405e-09, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.4904251e-09, Final residual = 1.4904251e-09, No Iterations 0
DICPCG:  Solving for p, Initial residual = 6.7291723e-07, Final residual = 6.7291723e-07, No Iterations 0
time step continuity errors : sum local = 2.5096865e-10, global = -1.7872395e-19, cumulative = 2.6884327e-18
ExecutionTime = 4.47 s  ClockTime = 5 s

fieldMinMax minmaxdomain output:
    min(p) = -0.37208362 at location (0.025 0.975 0.5)
    max(p) = 0.77640927 at location (0.975 0.975 0.5)
    min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
    max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)

Time = 50

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1.0907508e-09, Final residual = 1.0907508e-09, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.4677462e-09, Final residual = 1.4677462e-09, No Iterations 0
DICPCG:  Solving for p, Initial residual = 1.0020944e-06, Final residual = 1.0746895e-07, No Iterations 1
time step continuity errors : sum local = 4.0107145e-11, global = -5.0601748e-20, cumulative = 2.637831e-18
ExecutionTime = 4.47 s  ClockTime = 5 s

fieldMinMax minmaxdomain output:
    min(p) = -0.37208345 at location (0.025 0.975 0.5)
    max(p) = 0.77640927 at location (0.975 0.975 0.5)
    min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
    max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
```

CFL number at time step n - 1

CFL number at time step n

A deeper view to my first OpenFOAM® case setup

The output screen

- To control the CFL number you can change the time step or you can change the mesh.
- The easiest way is by changing the time step.
- For a time step of 0.1 seconds, this is the output you should get for this case,

```
Time = 49.9

Courant Number mean: 0.4441161 max: 1.6798756
smoothSolver:  Solving for Ux, Initial residual = 0.00016535808, Final residual = 2.7960145e-09, No Iterations 5
smoothSolver:  Solving for Uy, Initial residual = 0.00015920267, Final residual = 2.7704949e-09, No Iterations 5
DICPCG:  Solving for p, Initial residual = 0.0015842846, Final residual = 5.2788554e-07, No Iterations 26
time step continuity errors : sum local = 8.6128916e-09, global = 3.5439859e-19, cumulative = 2.4940081e-17
ExecutionTime = 0.81 s  ClockTime = 1 s

fieldMinMax minmaxdomain output:
    min(p) = -0.34322821 at location (0.025 0.975 0.5)
    max(p) = 0.73453489 at location (0.975 0.975 0.5)
    min(U) = (0.0002505779 -0.00025371425 0) at location (0.025 0.025 0.5)
    max(U) = (0.0002505779 -0.00025371425 0) at location (0.025 0.025 0.5)

Time = 50

Courant Number mean: 0.44411473 max: 1.6798833
smoothSolver:  Solving for Ux, Initial residual = 0.00016378098, Final residual = 2.7690608e-09, No Iterations 5
smoothSolver:  Solving for Uy, Initial residual = 0.00015720331, Final residual = 2.7354499e-09, No Iterations 5
DICPCG:  Solving for p, Initial residual = 0.0015662416, Final residual = 5.2290439e-07, No Iterations 26
time step continuity errors : sum local = 8.5379223e-09, global = -3.6676527e-19, cumulative = 2.4573316e-17
ExecutionTime = 0.81 s  ClockTime = 1 s

fieldMinMax minmaxdomain output:
    min(p) = -0.34244269 at location (0.025 0.975 0.5)
    max(p) = 0.73656831 at location (0.975 0.975 0.5)
    min(U) = (0.00025028679 -0.00025338014 0) at location (0.025 0.025 0.5)
    max(U) = (0.00025028679 -0.00025338014 0) at location (0.025 0.025 0.5)
```

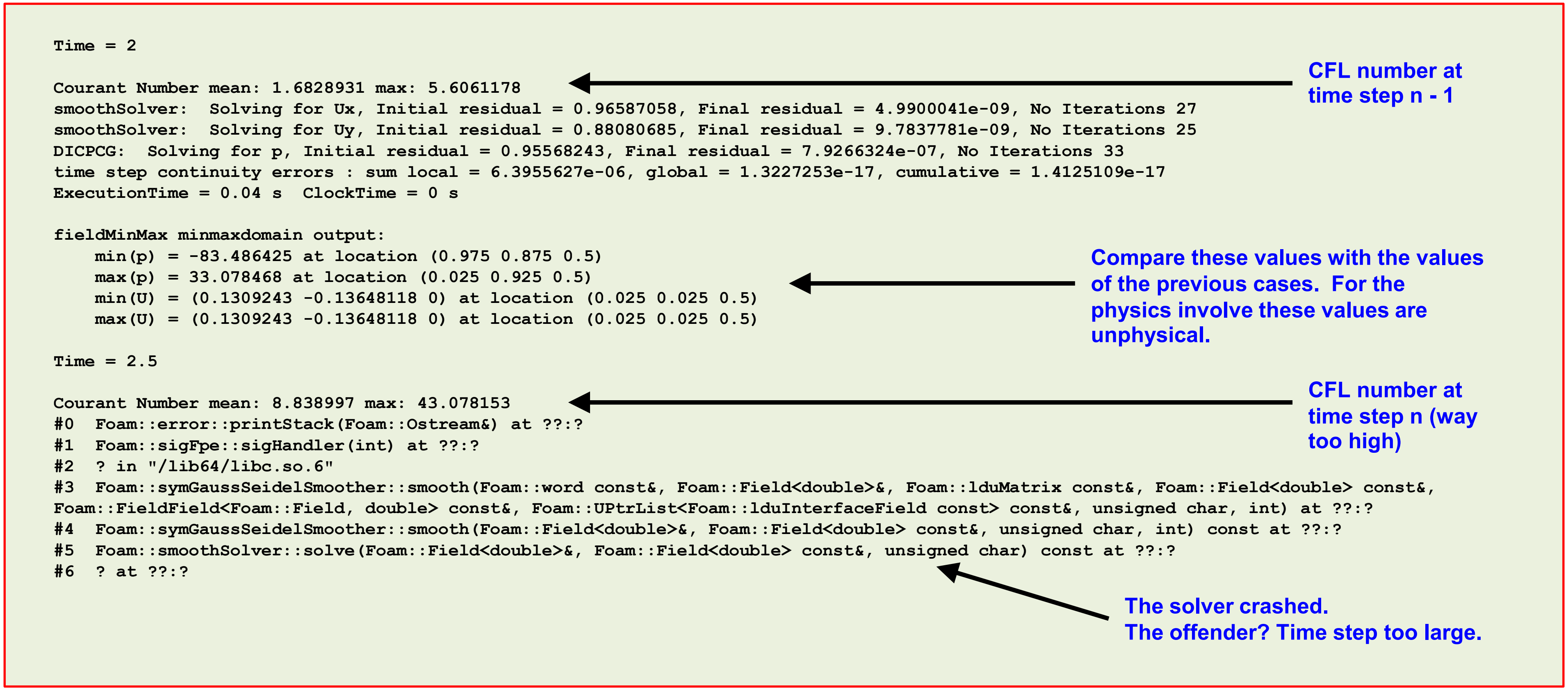
CFL number at time step n - 1

CFL number at time step n

A deeper view to my first OpenFOAM® case setup

The output screen

- To control the CFL number you can change the time step or you can change the mesh.
- The easiest way is by changing the time step.
- For a time step of 0.5 seconds, this is the output you should get for this case,



```
Time = 2

Courant Number mean: 1.6828931 max: 5.6061178
smoothSolver:  Solving for Ux, Initial residual = 0.96587058, Final residual = 4.9900041e-09, No Iterations 27
smoothSolver:  Solving for Uy, Initial residual = 0.88080685, Final residual = 9.7837781e-09, No Iterations 25
DICPCG:  Solving for p, Initial residual = 0.95568243, Final residual = 7.9266324e-07, No Iterations 33
time step continuity errors : sum local = 6.3955627e-06, global = 1.3227253e-17, cumulative = 1.4125109e-17
ExecutionTime = 0.04 s  ClockTime = 0 s

fieldMinMax minmaxdomain output:
    min(p) = -83.486425 at location (0.975 0.875 0.5)
    max(p) = 33.078468 at location (0.025 0.925 0.5)
    min(U) = (0.1309243 -0.13648118 0) at location (0.025 0.025 0.5)
    max(U) = (0.1309243 -0.13648118 0) at location (0.025 0.025 0.5)

Time = 2.5

Courant Number mean: 8.838997 max: 43.078153
#0  Foam::error::printStack(Foam::Ostream&) at ??:?
#1  Foam::sigFpe::sigHandler(int) at ??:?
#2  ? in "/lib64/libc.so.6"
#3  Foam::symGaussSeidelSmoother::smooth(Foam::word const&, Foam::Field<double>&, Foam::lduMatrix const&, Foam::Field<double> const&,
Foam::FieldField<Foam::Field, double> const&, Foam::UPtrList<Foam::lduInterfaceField const> const&, unsigned char, int) at ??:?
#4  Foam::symGaussSeidelSmoother::smooth(Foam::Field<double>&, Foam::Field<double> const&, unsigned char, int) const at ??:?
#5  Foam::smoothSolver::solve(Foam::Field<double>&, Foam::Field<double> const&, unsigned char) const at ??:?
#6  ? at ??:?
```

CFL number at time step n - 1

Compare these values with the values of the previous cases. For the physics involve these values are unphysical.

CFL number at time step n (way too high)

The solver crashed.
The offender? Time step too large.

A deeper view to my first OpenFOAM® case setup

The output screen

- Another output you should monitor are the continuity errors.
- These numbers should be small (it does not matter if they are negative or positive).
- If these values increase in time (about the order of 1e-2), you better control the case setup because something is wrong.
- The continuity errors are defined in the following file

$WM_PROJECT_DIR/src/finiteVolume/cfdTools/incompressible/continuityErrs.H

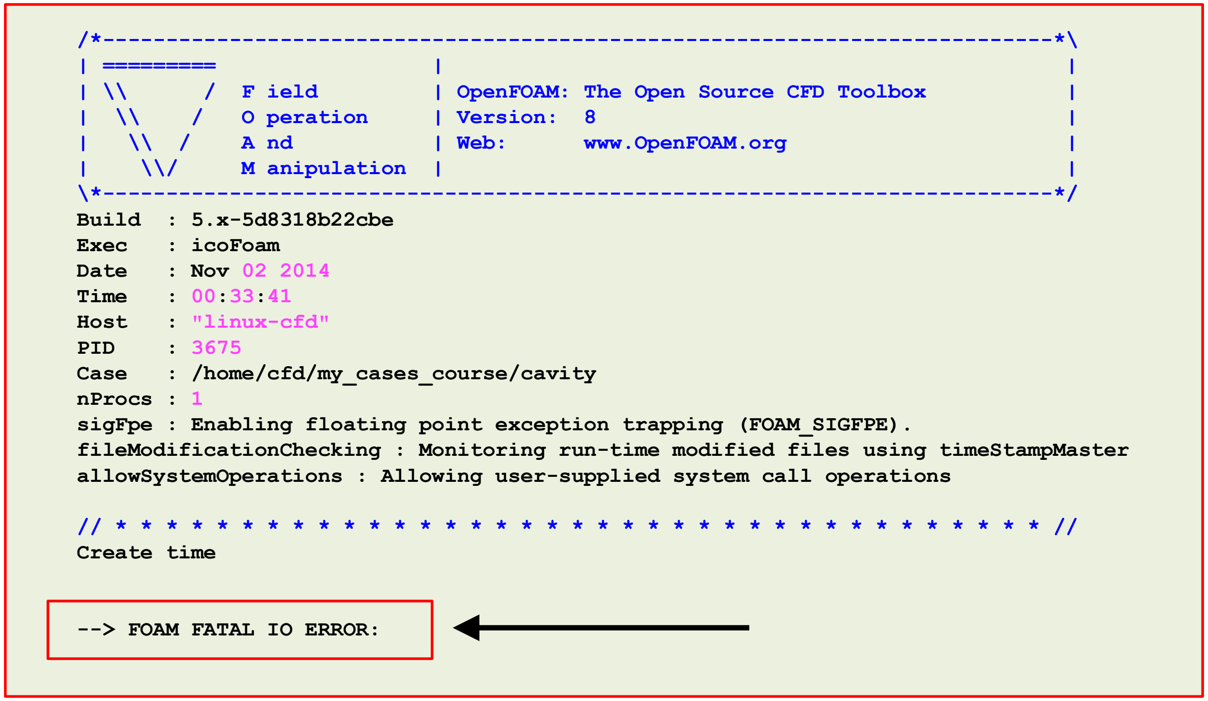
```
Time = 50

Courant Number mean: 0.44411473 max: 1.6798833
smoothSolver:  Solving for Ux, Initial residual = 0.00016378098, Final residual = 2.7690608e-09, No Iterations 5
smoothSolver:  Solving for Uy, Initial residual = 0.00015720331, Final residual = 2.7354499e-09, No Iterations 5
DICPCG:  Solving for p, Initial residual = 0.0015662416, Final residual = 5.2290439e-07, No Iterations 26
time step continuity errors : sum local = 8.5379223e-09, global = -3.6676527e-19, cumulative = 2.4573316e-17
ExecutionTime = 0.81 s  ClockTime = 1 s

fieldMinMax minmaxdomain output:
    min(p) = -0.34244269 at location (0.025 0.975 0.5)
    max(p) = 0.73656831 at location (0.975 0.975 0.5)
    min(U) = (0.00025028679 -0.00025338014 0) at location (0.025 0.025 0.5)
    max(U) = (0.00025028679 -0.00025338014 0) at location (0.025 0.025 0.5)
```

Continuity errors

A deeper view to my first OpenFOAM® case setup

Error output

- If you forget a keyword or a dictionary file, give a wrong option to a compulsory or optional entry, misspelled something, add something out of place in a dictionary, use the wrong dimensions, forget a semi-colon and so on, OpenFOAM® will give you the error `FOAM FATAL IO ERROR`.
- This error does not mean that the actual OpenFOAM® installation is corrupted. It is telling you that you are missing something or something is wrong in a dictionary.
- Maybe the guys of OpenFOAM® went a little bit extreme here.

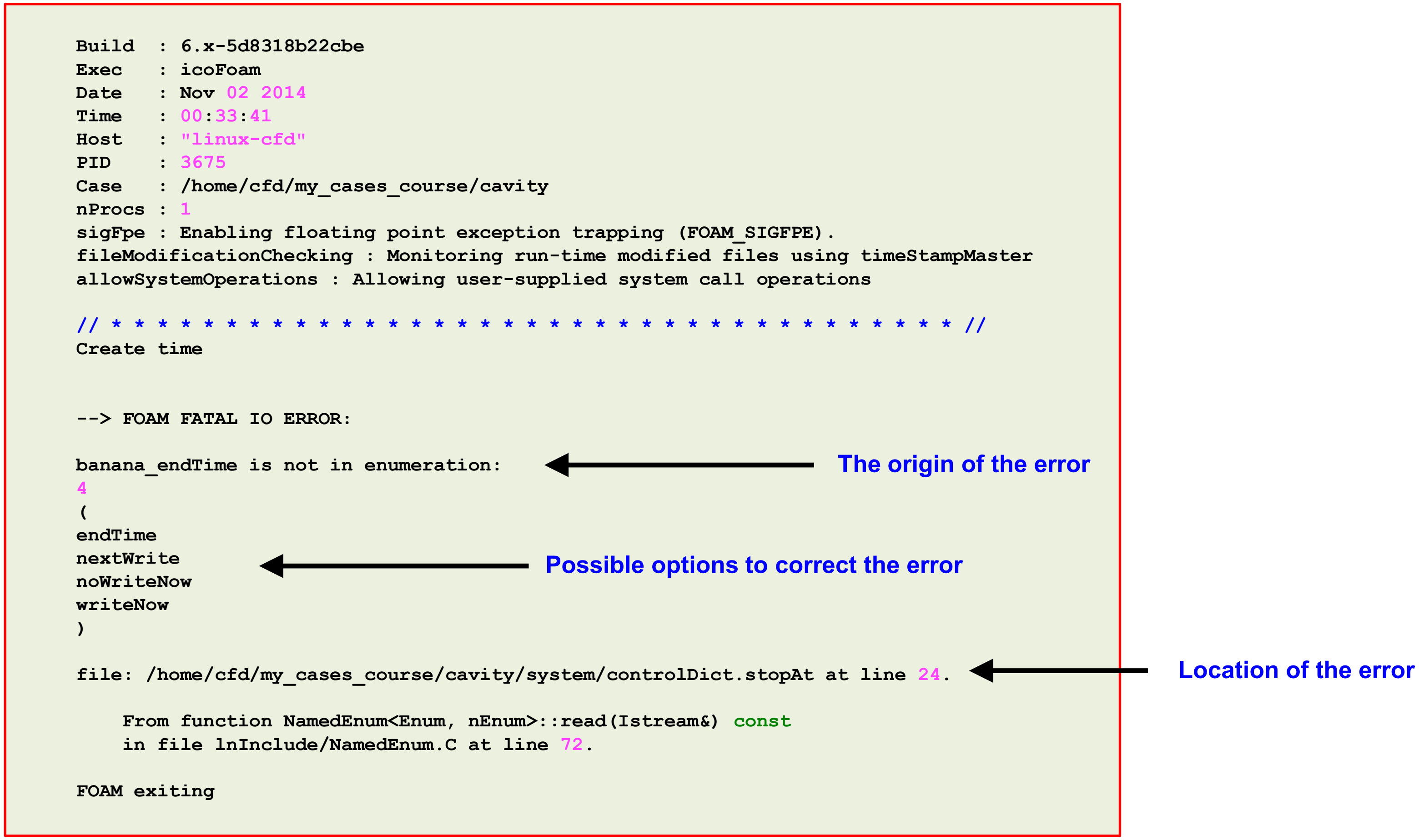


```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 5.x-5d8318b22cbe
Exec   : icoFoam
Date   : Nov 02 2014
Time   : 00:33:41
Host   : "linux-cfd"
PID    : 3675
Case   : /home/cfd/my_cases_course/cavity
nProcs : 1
sigFpe : Enabling floating point exception trapping (FOAM_SIGFPE).
fileModificationChecking : Monitoring run-time modified files using timeStampMaster
allowSystemOperations : Allowing user-supplied system call operations

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Create time


--> FOAM FATAL IO ERROR:
```

A deeper view to my first OpenFOAM® case setup

Error output

- Also, before entering into panic read carefully the output screen because OpenFOAM® is telling you what is the error and how to correct it.



```
Build   : 6.x-5d8318b22cbe
Exec    : icoFoam
Date    : Nov 02 2014
Time    : 00:33:41
Host    : "linux-cfd"
PID     : 3675
Case    : /home/cfd/my_cases_course/cavity
nProcs  : 1
sigFpe : Enabling floating point exception trapping (FOAM_SIGFPE).
fileModificationChecking : Monitoring run-time modified files using timeStampMaster
allowSystemOperations : Allowing user-supplied system call operations

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Create time


--> FOAM FATAL IO ERROR:

banana_endTime is not in enumeration:
4
(
endTime
nextWrite
noWriteNow
writeNow
)

file: /home/cfd/my_cases_course/cavity/system/controlDict.stopAt at line 24.

    From function NamedEnum<Enum, nEnum>::read(Istream&) const
    in file lnInclude/NamedEnum.C at line 72.

FOAM exiting
```

The origin of the error

Possible options to correct the error

Location of the error

A deeper view to my first OpenFOAM® case setup

Error output

- It is very important to read the screen and understand the output.

> *“Experience is simply the name we give our mistakes.”*

- Train yourself to identify the errors. Hereafter we list a few possible errors.
- Missing compulsory file p

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

file: /home/joegi/my_cases_course/6/101OF/cavity/0/p at line 0.

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

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- Mismatching patch name in file *p*

```
--> FOAM FATAL IO ERROR:
Cannot find patchField entry for xmovingWall

file: /home/joegi/my_cases_course/6/101OF/cavity/0/p.boundaryField from line 25 to line 35.

    From function GeometricField<Type, PatchField, GeoMesh>::GeometricBoundaryField::readField(const
DimensionedField<Type, GeoMesh>&, const dictionary&)
    in file /home/joegi/OpenFOAM/OpenFOAM-6/src/OpenFOAM/lnInclude/GeometricBoundaryField.C at line 209.

FOAM exiting
```

- Missing compulsory keyword in *fvSchemes*

```
--> FOAM FATAL IO ERROR:
keyword div(phi,U) is undefined in dictionary
"/home/joegi/my_cases_course/6/101OF/cavity/system/fvSchemes.divSchemes"

file: /home/joegi/my_cases_course/6/101OF/cavity/system/fvSchemes.divSchemes from line 30 to line 30.

    From function dictionary::lookupEntry(const word&, bool, bool) const
    in file db/dictionary/dictionary.C at line 442.

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- Missing entry in file *fvSolution* at keyword **PISO**

```
--> FOAM FATAL IO ERROR:
"ill defined primitiveEntry starting at keyword 'PISO' on line 68 and ending at line 68"

file: /home/joegi/my_cases_course/6/101OF/cavity/system/fvSolution at line 68.

    From function primitiveEntry::readEntry(const dictionary&, Istream&)
    in file lnInclude/IOerror.C at line 132.

FOAM exiting
```

- Incompatible dimensions. Likely the offender is the file *U*

```
--> FOAM FATAL ERROR:
incompatible dimensions for operation
    [U[0 1 -2 1 0 0 0] ] + [U[0 1 -2 2 0 0 0] ]

    From function checkMethod(const fvMatrix<Type>&, const fvMatrix<Type>&)
    in file /home/joegi/OpenFOAM/OpenFOAM-6/src/finiteVolume/lnInclude/fvMatrix.C at line 1295.

FOAM aborting

#0  Foam::error::printStack(Foam::Ostream&) at ??:?
#1  Foam::error::abort() at ??:?
#2  void Foam::checkMethod<Foam::Vector<double> >(Foam::fvMatrix<Foam::Vector<double> > const&,
Foam::fvMatrix<Foam::Vector<double> > const&, char const*) at ??:?
#3  ? at ??:?
#4  ? at ??:?
#5  __libc_start_main in "/lib64/libc.so.6"
#6  ? at /home/abuild/rpmbuild/BUILD/glibc-2.19/csu/../sysdeps/x86_64/start.S:125
Aborted
```

A deeper view to my first OpenFOAM® case setup

Error output

- Missing keyword **deltaT** in file *controlDict*

```
--> FOAM FATAL IO ERROR:
keyword deltaT is undefined in dictionary "/home/joegi/my_cases_course/6/101OF/cavity/system/controlDict"

file: /home/joegi/my_cases_course/6/101OF/cavity/system/controlDict from line 17 to line 69.

    From function dictionary::lookupEntry(const word&, bool, bool) const
    in file db/dictionary/dictionary.C at line 442.

FOAM exiting
```

- Missing file *points* in directory polyMesh. Likely you are missing the mesh.

```
--> 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 203.

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- Unknown boundary condition type.

```
--> FOAM FATAL IO ERROR:
Unknown patchField type sfixedValue for patch type wall

Valid patchField types are :

74
(
SRFFreestreamVelocity
SRFVelocity
SRFWallVelocity
activeBaffleVelocity

...
...
...

variableHeightFlowRateInletVelocity
waveTransmissive
wedge
zeroGradient
)


file: /home/joegi/my_cases_course/6/101OF/cavity/0/U.boundaryField.movingWall from line 25 to line 26.

    From function fvPatchField<Type>::New(const fvPatch&, const DimensionedField<Type, volMesh>&, const
dictionary&)
    in file /home/joegi/OpenFOAM/OpenFOAM-6/src/finiteVolume/lnInclude/fvPatchFieldNew.C at line 143.

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- This one is especially hard to spot

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  8                                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 6.x-5d8318b22cbe
Exec   : icoFoam
Date   : Nov 02 2014
Time   : 00:33:41
Host   : "linux-cfd"
PID    : 3675
fileName::stripInvalid() called for invalid fileName /home/cfd/my_cases_course/cavity0
    For debug level (= 2) > 1 this is considerd fatal
Aborted
```

- This error is related to the name of the working directory. In this case the name of the working directory is **`cavity 0`** (there is a blank space between the word cavity and the number 0).
- Do not use blank spaces or funny symbols when naming directories and files.
- Instead of **`cavity 0`** you could use **`cavity_0`**.

A deeper view to my first OpenFOAM® case setup

Error output

- You should worry about the `SIGFPE` error signal. This error signal indicates that something went really wrong (erroneous arithmetic operation).
- This message (that seems a little bit difficult to understand), is giving you a lot information.
- For instance, this output is telling us that the error is due to `SIGFPE` and the class associated to the error is `lduMatrix`. It is also telling you that the `GAMGSolver` solver is the affected one (likely the offender is the pressure).

```
#0  Foam::error::printStack(Foam::Ostream&) at ??:?
#1  Foam::sigFpe::sigHandler(int) at ??:?
#2   in "/lib64/libc.so.6"
#3  Foam::DICPreconditioner::calcReciprocalD(Foam::Field<double>&, Foam::lduMatrix const&) at ??:?
#4  Foam::DICSmoother::DICSmoother(Foam::word const&, Foam::lduMatrix const&, Foam::FieldField<Foam::Field, double>
const&, Foam::FieldField<Foam::Field, double> const&, Foam::UPtrList<Foam::lduInterfaceField const> const&) at ??:?
#5  Foam::lduMatrix::smoother::addsymMatrixConstructorToTable<Foam::DICSmoother>::New(Foam::word const&,
Foam::lduMatrix const&, Foam::FieldField<Foam::Field, double> const&, Foam::FieldField<Foam::Field, double> const&,
Foam::UPtrList<Foam::lduInterfaceField const> const&) at ??:?
#6  Foam::lduMatrix::smoother::New(Foam::word const&, Foam::lduMatrix const&, Foam::FieldField<Foam::Field, double>
const&, Foam::FieldField<Foam::Field, double> const&, Foam::UPtrList<Foam::lduInterfaceField const> const&,
Foam::dictionary const&) at ??:?
#7  Foam::GAMGSolver::initVcycle(Foam::PtrList<Foam::Field<double> >&, Foam::PtrList<Foam::Field<double> >&,
Foam::PtrList<Foam::lduMatrix::smoother>&, Foam::Field<double>&, Foam::Field<double>&) const at ??:?
#8  Foam::GAMGSolver::solve(Foam::Field<double>&, Foam::Field<double> const&, unsigned char) const at ??:?
#9  Foam::fvMatrix<double>::solveSegregated(Foam::dictionary const&) at ??:?
#10  Foam::fvMatrix<double>::solve(Foam::dictionary const&) at ??:?
#11
 at ??:?
#12  __libc_start_main in "/lib64/libc.so.6"
#13
 at /home/abuild/rpmbuild/BUILD/glibc-2.17/csu/../sysdeps/x86_64/start.S:126
Floating point exception
```

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- OpenFOAM® follows same general syntax rules as in C++.
- Commenting in OpenFOAM® (same as in C++):

```
// This is a line comment
```

```
/*
    This is a block comment
*/
```

- As in C++, you can use the **#include** directive in your dictionaries (do not forget to create the respective include file):

```
#include "initialConditions"
```

- Scalars, vectors, lists and dictionaries.
  - Scalars in OpenFOAM® are represented by a single value, e.g.,

    ```
    3.14159
    ```

  - Vectors in OpenFOAM® are represented as a list with three components, e.g.,

    ```
    (1.0  0.0  0.0)
    ```

  - A second order tensor in OpenFOAM® is represented as a list with nine components, e.g.,

    ```
    (
        1.0  0.0  0.0
        0.0  1.0  0.0
        0.0  0.0  1.0
    )
    ```

Dictionary files general features

- Scalars, vectors, lists and dictionaries.
  - List entries are contained within parentheses **( )**. A list can contain scalars, vectors, tensors, words, and so on.
    - A list of scalars is represented as follows:

```
name_of_the_list
(
      0
      1
      2
);
```

    - A list of vectors is represented as follows:

```
name_of_the_list
(
      (0 0 0)
      (1 0 0)
      (2 0 0)
);
```

    - A list of words is represented as follows

```
name_of_the_list
(
      “word1”
      “word2”
      “word3”
);
```

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- OpenFOAM® uses dictionaries to specify data in an input file (dictionary file).
- A dictionary in OpenFOAM® can contain multiple data entries and at the same time dictionaries can contain sub-dictionaries.
- To specify a dictionary entry, the name is followed by the keyword entries in curly braces:

```
solvers     ←――――――――――――――――――――――――――――  Dictionary solvers
{
    p       ←――――――――――――――――――――――――――――  Sub-dictionary p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }

    U       ←――――――――――――――――――――――――――――  Sub-dictionary U
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }
      ...
      ...
      ...
}
```

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- Macro expansion.
  - We first declare a variable (**x = 10**) and then we use it through the **$** macro substitution (**$x**).

```
vectorField         (20 0 0);                            //Declare variable

internalField        uniform $vectorField;               //Use declared variable


scalarField         101328;                              //Declare variable

type                fixedValue;
value               uniform      $scalarField;           //Use declared variable
```

  - You can use macro expansion to duplicate and access variables in dictionaries

```
p                                      // Declare/create the dictionary p
{
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
}

$p;                                    //To create a copy of the dictionary p
$p.solver;                             //To access the variable solver in the dictionary p
```

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- Instead of writing (the poor man's way):

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

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

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

- You can write (the lazy way):

```
"(left|right|top)Wall"
{
        type   fixedValue;
        value  uniform (0 0 0);
}
```

- You could also try (even lazier):

```
".*Wall"
{
        type   fixedValue;
        value  uniform (0 0 0);
}
```

- OpenFOAM® understands the syntax of regular expressions (regex or regeaxp).

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- Inline calculations.
  - You can use the directive **#calc** to do inline calculations, the syntax is as follows:

```
X = 10.0;                                   //Declare variable
Y = 3.0;                                    //Declare variable

Z   #calc   "$X*$Y – 12.0";                 //Do inline calculation. The result is saved in the variable Z
```

- With inline calculations you can access all the mathematical functions available in C++.
- Macro expansions and inline calculations are very useful to parametrize dictionaries and avoid repetitive tasks.
- Switches: they are used to enable or disable a function or a feature in the dictionaries.
- Switches are logical values. You can use the following values:

Switches	
false	true
off	on
no	yes
n	y
f	t
none	true

- You can find all the valid switches in the following file:

`OpenFOAM-6/src/OpenFOAM/primitives/bools/Switch/Switch.C`

A deeper view to my first OpenFOAM® case setup

Solvers and utilities help

- If you need help about a solver or utility, you can use the option `-help`. For instance:

    - `$> icoFoam -help`

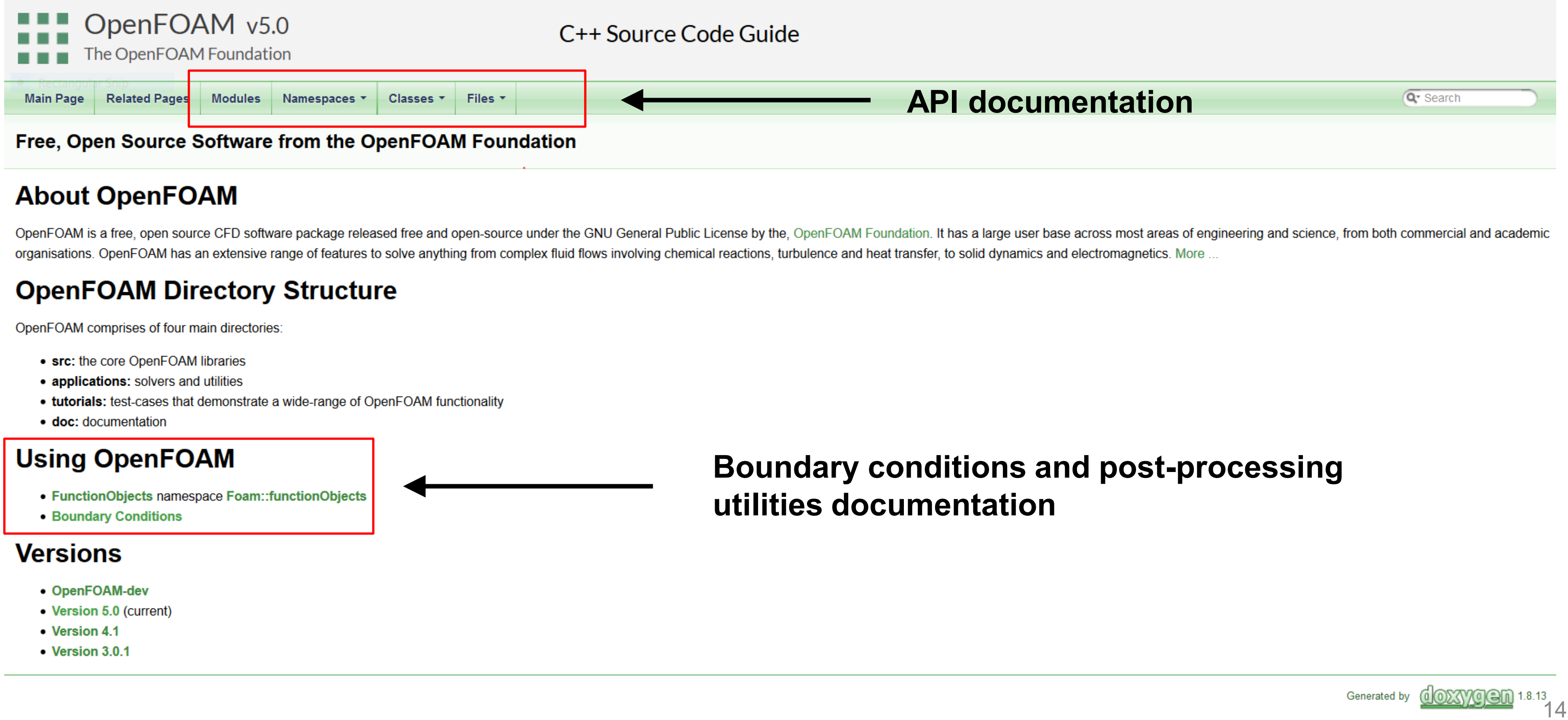
  will print some basic help and usage information about `icoFoam`

- Remember, you have the source code there so you can always check the original source.

A deeper view to my first OpenFOAM® case setup

Solvers and utilities help

- To get more information about the boundary conditions, post-processing utilities, and the API read the Doxygen documentation.
- If you did not compile the Doxygen documentation, you can access the information online, http://cpp.openfoam.org/v6/



Free, Open Source Software from the OpenFOAM Foundation

About OpenFOAM

OpenFOAM is a free, open source CFD software package released free and open-source under the GNU General Public License by the, OpenFOAM Foundation. It has a large user base across most areas of engineering and science, from both commercial and academic organisations. OpenFOAM has an extensive range of features to solve anything from complex fluid flows involving chemical reactions, turbulence and heat transfer, to solid dynamics and electromagnetics. More ...

OpenFOAM Directory Structure

OpenFOAM comprises of four main directories:

- **src:** the core OpenFOAM libraries
- **applications:** solvers and utilities
- **tutorials:** test-cases that demonstrate a wide-range of OpenFOAM functionality
- **doc:** documentation

Using OpenFOAM

- FunctionObjects namespace Foam::functionObjects
- Boundary Conditions

Versions

- OpenFOAM-dev
- Version 5.0 (current)
- Version 4.1
- Version 3.0.1

Generated by doxygen 1.8.13

API documentation

Boundary conditions and post-processing utilities documentation

A deeper view to my first OpenFOAM® case setup

Exercises

- Run the case with Re = 10 and Re = 1000. Feel free to change any variable to achieve the Re value (velocity, viscosity or length). Do you see an unsteady behavior in any of the cases? What about the computing time, what simulation is faster?
- Run the tutorial with Re = 100, a mesh with 120 x 120 x 1 cells, and using the default setup (original `controlDict`, `fvSchemes` and `fvSolution`). Did the simulation converge? Did it crash? Any comments.
- If your simulation crashed, try to solve the problem.
  (Hint: try to reduce the time-step to get a CFL less than 1)
- Besides reducing the time-step, can you find another solution?
  (Hint: look at the PISO options)
- Change the **base type** of the boundary patch **movingWall** to **patch**. (the `boundary` file). Do you get the same results? Can you comment on this?
- Try to extent the problem to 3D and use a uniform mesh (20 x 20 x 20). Compare the solution at the mid section of the 3D simulation with the 2D solution. Are the solutions similar?
- How many time discretization schemes are there in OpenFOAM®? Try to use a different discretization scheme.
- Run the simulation using **Gauss upwind** instead of **Gauss linear** for the term **div(phi,U)** (`fvSchemes`). Do you get the same quantitative results?
- Sample the field variables **U** and **P** at a different location and plot the results $\nu = \mu/\rho$ using gnuplot.
- What density value do you think we were using? What about dynamic viscosity?

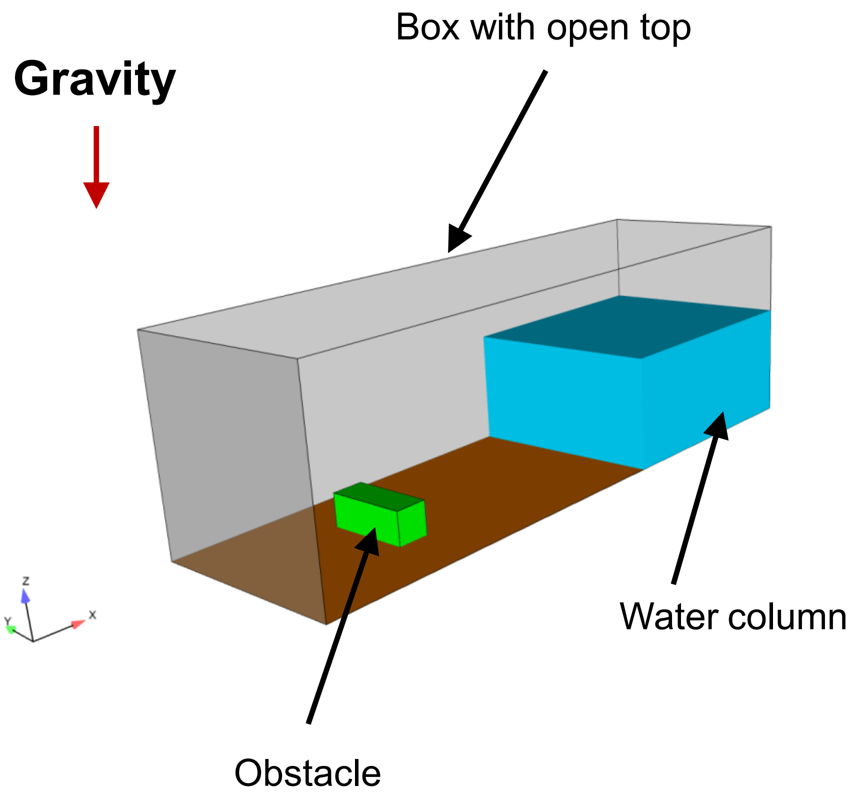
Hint: the physical pressure is equal to the modified pressure and

Roadmap

1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. ~~Directory structure of an application/utility~~
4. ~~Applications/utilities in OpenFOAM®~~
5. ~~Directory structure of an OpenFOAM® case~~
6. ~~Running my first OpenFOAM® case setup blindfold~~
7. ~~A deeper view to my first OpenFOAM® case setup~~
8. **3D Dam break – Free surface flow**
9. Flow past a cylinder – From laminar to turbulent flow

3D Dam break – Free surface flow

Dam break free surface flow



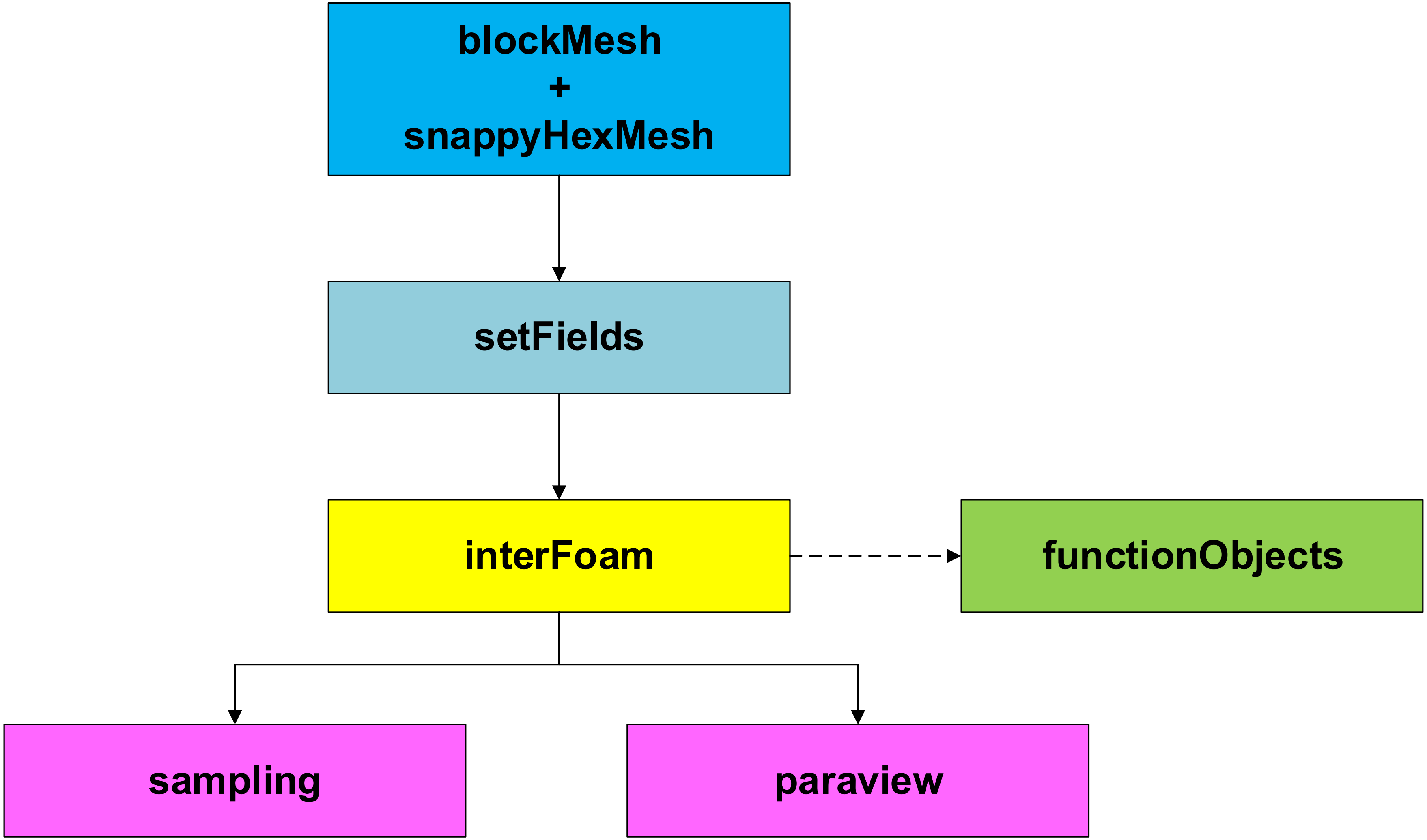
Physical and numerical side of the problem:

- In this case we are going to use the volume of fluid (VOF) method.
- This method solves the incompressible Navier-Stokes equations plus an additional equation to track the phases (free surface location).
- As this is a multiphase case, we need to define the physical properties for each phase involved (viscosity, density and surface tension).
- The working fluids are water and air.
- Additionally, we need to define the gravity vector and initialize the two flows.
- This is a three-dimensional and unsteady case.
- The details of the case setup can be found in the following reference:

  A Volume-of-Fluid Based Simulation Method for Wave Impact Problems.
  Journal of Computational Physics 206(1):363-393.
  June, 2005.

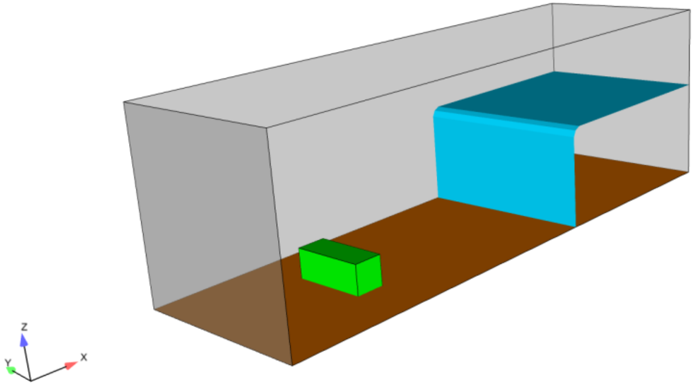
3D Dam break – Free surface flow

Workflow of the case

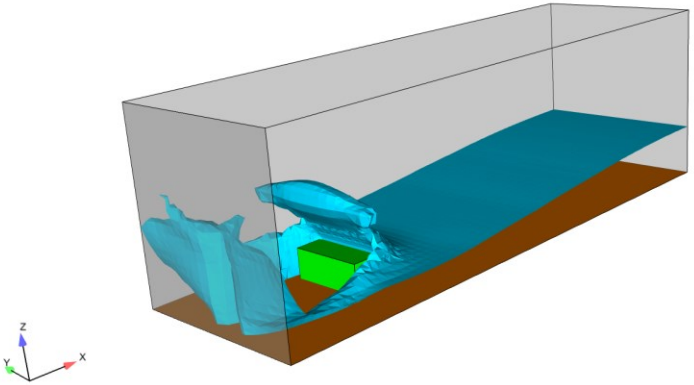


3D Dam break – Free surface flow

At the end of the day, you should get something like this



Initial conditions – Coarse mesh

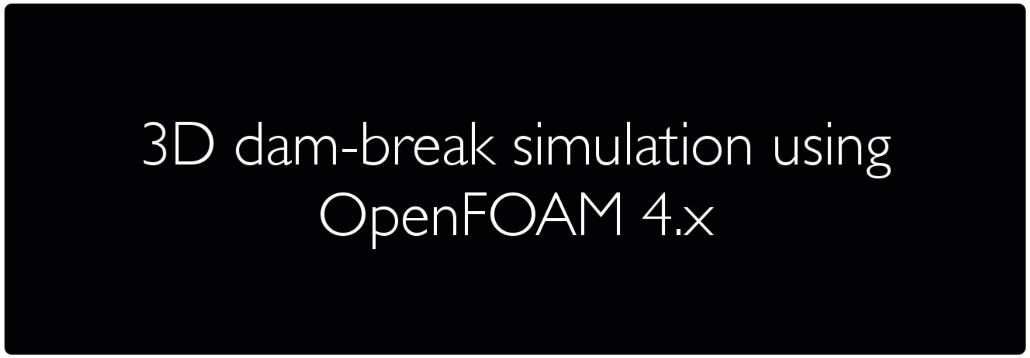


Solution at Time = 1 second – Coarse mesh

3D Dam break – Free surface flow

VOF Fraction (Free surface tracking) – Very fine mesh

http://www.wolfdynamics.com/validations/3d_db/dbreak.gif



3D dam-break simulation using OpenFOAM 4.x

3D Dam break – Free surface flow

- Let us run this case. Go to the directory:

```
$PTOFC/101OF/3d_damBreak
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

3D Dam break – Free surface flow

What are we going to do?

- We will use this case to introduce the multiphase solver `interFoam`.
- `interFoam` is a solver for 2 incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach
- We will define the physical properties of two phases, and we are going to initialize these phases.
- We will define the gravity vector in the dictionary *g*.
- After finding the solution, we will visualize the results. This is an unsteady case so now we are going to see things moving.
- We are going to briefly address how to post-process multiphase flows.
- We are going to generate the mesh using snappyHexMesh, but for the purpose of this tutorial we are not going to discuss the dictionaries.
- Remember, different solvers have different input dictionaries.

3D Dam break – Free surface flow

The **`constant`** directory

- In this directory, we will find the following compulsory dictionary files:

    - *`g`*
    - *`transportProperties`*
    - *`momentumTransport`*

- *`g`* contains the definition of the gravity vector.
- *`transportProperties`* contains the definition of the physical properties of each phase.
- *`momentumTransport`* contains the definition of the turbulence model to use.

3D Dam break – Free surface flow

The g dictionary file

```
8     FoamFile
9     {
10        version     2.0;
11        format      ascii;
12        class       uniformDimensionedVectorField;
13        location    "constant";
14        object      g;
15    }
17
18    dimensions        [0 1 -2 0 0 0 0];
19    value             (0 0 -9.81);
```

- This dictionary file is located in the directory **`constant`**.
- For multiphase flows, this dictionary is compulsory.
- In this dictionary we define the gravity vector (line 19).
- Pay attention to the **class** type (line 12).

3D Dam break – Free surface flow

The *transportProperties* dictionary file

Primary phase

```
17    phases (water air);
18
19    water
20    {
21        transportModel  Newtonian;
22        nu              [0 2 -1 0 0 0 0] 1e-06;
23        rho             [1 -3 0 0 0 0 0] 1000;
24    }
25
26    air
27    {
28        transportModel  Newtonian;
29        nu              [0 2 -1 0 0 0 0] 1.48e-05;
30        rho             [1 -3 0 0 0 0 0] 1;
31    }
32
33    sigma           [1 0 -2 0 0 0 0] 0.07;
```

- This dictionary file is located in the directory **`constant`**.
- We first define the name of the phases (line 17). In this case we are defining the names **water** and **air**. The first entry in this list is the primary phase (**water**).
- The name of the primary phase is the one you will use to initialize the solution.
- The name of the phases is given by the user.
- In this file we set the kinematic viscosity (**nu**), density (**rho**) and transport model (**transportModel**) of the phases.
- We also define the surface tension (**sigma**).

3D Dam break – Free surface flow

The *momentumTransport* dictionary file

- In this dictionary file we select what model we would like to use (laminar or turbulent).
- This dictionary is compulsory.
- In this case we use a RANS turbulence model (kEpsilon).

```
simulationType    RAS;

RAS
{
    RASModel kEpsilon;

    turbulence on;

    printCoeffs on;
}
```

3D Dam break – Free surface flow

The 0 directory

- In this directory, we will find the dictionary files that contain the boundary and initial conditions for all the primitive variables.
- As we are solving the incompressible RANS Navier-Stokes equations using the VOF method, we will find the following field files:

• *alpha.water*	(volume fraction of water phase)
• *p_rgh*	(pressure field minus hydrostatic component)
• *U*	(velocity field)
• *k*	(turbulent kinetic energy field)
• *epsilon*	(rate of dissipation of turbulence energy field)
• *nut*	(turbulence viscosity field)

3D Dam break – Free surface flow

The file *0/alpha.water*

```
17    dimensions      [0 0 0 0 0 0 0];
18
19    internalField   uniform 0;
20
21    boundaryField
22    {
23        front
24        {
25            type            zeroGradient;
26        }
27        back
28        {
29            type            zeroGradient;
30        }
31        left
32        {
33            type            zeroGradient;
34        }
35        right
36        {
37            type            zeroGradient;
38        }
39        bottom
40        {
41            type            zeroGradient;
42        }
43        top
44        {
45            type            inletOutlet;
46            inletValue      uniform 0;
47            value           uniform 0;
48        }
49        stlSurface
50        {
51            type             zeroGradient;
52        }
53
54    }
```

- This file contains the boundary and initial conditions for the non-dimensional scalar field **alpha.water**
- This file is named `alpha.water`, because the primary phase is water (we defined the primary phase in the `transportProperties` dictionary).
- Initially, this field is initialized as 0 in the whole domain (line 19). This means that there is no water in the domain at time 0. Later, we will initialize the water column and this file will be overwritten with a non-uniform field for the **internalField**.
- For the **front**, **back**, **left**, **right**, **bottom** and **stlSurface** patches we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).
- For the **top** patch we are using an **inletOutlet** boundary condition. This boundary condition avoids backflow into the domain. If the flow is going out it will use **zeroGradient** and if the flow is coming back it will assign the value set in the keyword **inletValue** (line 46).

3D Dam break – Free surface flow

The file *0/p_rgh*

```
17    dimensions       [1 -1 -2 0 0 0 0];
18
19    internalField    uniform 0;
20
21    boundaryField
22    {
23        front
24        {
25            type              fixedFluxPressure;
26            value             uniform 0;
27        }
28        back

33        left

38        right

43        bottom

48        top
49        {
50            type              totalPressure;
51            p0                uniform 0;
52            U                 U;
53            phi               phi;
54            rho               rho;
55            psi               none;
56            gamma             1;
57            value             uniform 0;
58        }
59        stlSurface
60        {
61            type              fixedFluxPressure;
62            value             uniform 0;
63        }
64
65    }
```

- This file contains the boundary and initial conditions for the dimensional scalar field **p_rgh**. The dimensions of this field are given in Pascal (line 17)
- This scalar field contains the value of the static pressure field minus the hydrostatic component.
- This field is initialized as 0 in the whole domain (line 19).
- For the **front**, **back**, **left, right, bottom** and **stlSurface** patches we are using the **fixedFluxPressure** boundary condition (refer to the source code or doxygen documentation to know more about this boundary condition).
- For the **top** patch we are using the **totalPressure** boundary condition (refer to the source code or doxygen documentation to know more about this boundary condition).

3D Dam break – Free surface flow

The file *0/U*

```
17    dimensions      [0 -1 -1 0 0 0 0];
18
19    internalField   uniform (0 0 0);
20
21    boundaryField
22    {
23        front
24        {
25            type            fixedValue;
26            value           uniform (0 0 0);
27        }
28        back

33        left

38        right

43        bottom

48        top
49        {
50            type            pressureInletOutletVelocity;
51            value           uniform (0 0 0);
52        }
53        stlSurface
54        {
55            type            fixedValue;
56            value           uniform (0 0 0);
57        }
58
59    }
```

- This file contains the boundary and initial conditions for the dimensional vector field **U**.
- We are using uniform initial conditions and the numerical value is **(0 0 0)** (keyword **internalField** in line 19).
- The **front**, **back**, **left, right, bottom** and **stlSurface** patches are no-slip walls, therefore we impose a **fixedValue** boundary condition with a value of **(0 0 0)** at the wall.
- For the **top** patch we are using the **pressureInlterOutletVelocity** boundary condition (refer to the source code or doxygen documentation to know more about this boundary condition).

3D Dam break – Free surface flow

The file *0/k*

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

internalField   uniform 0.1;

boundaryField
{
    "(front|back|left|right|bottom|stlSurface)"
    {
        type            kqRWallFunction;
        value           $internalField;
    }

    top
    {
        type            inletOutlet;
        inletValue      $internalField;
        value           $internalField;
    }

}
```

- This file contains the boundary and initial conditions for the dimensional scalar field **k.**
- This scalar (turbulent kinetic energy), is related to the turbulence model.
- This field is initialized as 0.1 in the whole domain, and all the boundary patches take the same value (**$internalField**).
- For the **front**, **back**, **left, right, bottom** and **stlSurface** patches we are using the **kqRWallFunction** boundary condition, which applies a wall function at the walls (refer to the source code or doxygen documentation to know more about this boundary condition).
- For the **top** patch we are using the **inletOutlet** boundary condition, this boundary condition handles backflow (refer to the source code or doxygen documentation to know more about this boundary condition).
- We will deal with turbulence modeling later.

3D Dam break – Free surface flow

The file *0/epsilon*

```
dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 0.1;

boundaryField
{
    "(front|back|left|right|bottom|stlSurface)"
    {
        type            epsilonWallFunction;
        value           $internalField;
    }

    top
    {
        type            inletOutlet;
        inletValue      $internalField;
        value           $internalField;
    }

}
```

- This file contains the boundary and initial conditions for the dimensional scalar field **epsilon**.
- This scalar (rate of dissipation of turbulence energy), is related to the turbulence model.
- This field is initialized as 0.1 in the whole domain, and all the boundary patches take the same value (**$internalField**).
- For the **front**, **back**, **left, right, bottom** and **stlSurface** patches we are using the **epsilonWallFunction** boundary condition, which applies a wall function at the walls (refer to the source code or doxygen documentation to know more about this boundary condition).
- For the **top** patch we are using the **inletOutlet** boundary condition, this boundary condition handles backflow (refer to the source code or doxygen documentation to know more about this boundary condition).
- We will deal with turbulence modeling later.

3D Dam break – Free surface flow

The file *0/nut*

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

internalField   uniform 0;

boundaryField
{
    “(front|back|left|right|bottom|stlSurface)”
    {
        type            nutkWallFunction;
        value            $internalField;
    }

    top
    {
        type            calculated;
        value            $internalField;;
    }

}
```

- This file contains the boundary and initial conditions for the dimensional scalar field **nut**.
- This scalar (turbulent viscosity), is related to the turbulence model.
- This field is initialized as 0 in the whole domain, and all the boundary patches take the same value (**$internalField**).
- For the **front**, **back**, **left, right, bottom** and **stlSurface** patches we are using the **nutkWallFunction** boundary condition, which applies a wall function at the walls (refer to the source code or doxygen documentation to know more about this boundary condition).
- For the **top** patch we are using the **calculated** boundary condition, this boundary condition computes the value of nut from k and epsilon (refer to the source code or doxygen documentation to know more about this boundary condition).
- We will deal with turbulence modeling later.

3D Dam break – Free surface flow

The **system** directory

- The **system** directory consists of the following compulsory dictionary files:
  - *controlDict*
  - *fvSchemes*
  - *fvSolution*

- *controlDict* contains general instructions on how to run the case.
- *fvSchemes* contains instructions for the discretization schemes that will be used for the different terms in the equations.
- *fvSolution* contains instructions on how to solve each discretized linear equation system.

3D Dam break – Free surface flow

The *controlDict* dictionary

```
17      application      interFoam;
18
19      startFrom        startTime;
20
21      startTime        0;
22
23      stopAt           endTime;
24
25      endTime          8;
26
27      deltaT           0.0001;
28
29      writeControl     adjustableRunTime;
30
31      writeInterval    0.02;
32
33      purgeWrite       0;
34
35      writeFormat      ascii;
36
37      writePrecision   8;
38
39      writeCompression uncompressed;
40
41      timeFormat       general;
42
43      timePrecision    8;
44
45      runTimeModifiable yes;
46
47      adjustTimeStep   yes;
48
49      maxCo            1.0;
50      maxAlphaCo       0.5;
51      maxDeltaT        0.01;
```

- This case starts from time 0 (**startTime**), and it will run up to 8 seconds (**endTime**).
- The initial time step of the simulation is 0.0001 seconds (**deltaT**).
- It will write the solution every 0.02 seconds (**writeInterval**) of simulation time (**runTime**). It will automatically adjust the time step (**adjustableRunTime**), in order to save the solution at the precise write interval.
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- The write precision is 8 digits (**writePrecision**). It will only save eight digits in the output files.
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.
- In line 47 we turn on the option **adjustTimeStep**. This option will automatically adjust the time step to achieve the maximum desired courant number (lines 49-50). We also set a maximum time step in line 51.
- Remember, the first time step of the simulation is done using the value set in line 27 and then it is automatically scaled to achieve the desired maximum values (lines 49-51).

3D Dam break – Free surface flow

The `controlDict` dictionary

```
55      functions
56      {

60      minmaxdomain
61      {
62          type fieldMinMax;
63
64          functionObjectLibs ("libfieldFunctionObjects.so");
65
66          enabled true; //true or false
67
68          mode component;
69
70          writeControl timeStep;
71          writeInterval 1;
72
73          log true;
74
75          fields (p p_rgh U alpha.water k epsilon);
76      }

144     };
```

- Let us take a look at the **functionObjects** definitions.
- In lines 60-76 we define the **fieldMinMax functionObject** which computes the minimum and maximum values of the field variables (**p p_rgh U alpha.water k epsilon**).

3D Dam break – Free surface flow

The `controlDict` dictionary

```
55      functions
56      {

81      water_in_domain
82      {
83          type                volRegion;
84          functionObjectLibs ("libfieldFunctionObjects.so");
85          enabled             true;
86
87          enabled             true;
88
89          //writeControl      outputTime;
90          writeControl    timeStep;
91          writeInterval   1;
92
93          log                 true;
94
95          regionType          all;
96
97         operation          volIntegrate;
98         fields
99         (
100             alpha.water
101         );
102     }

144     };
```

- Let us take a look at the **functionObjects** definitions.
- In lines 81-102 we define the **volRegion functionObject** which computes the volume integral (**volIntegrate**) of the field variable **alpha.water** in all the domain.
- Basically, we are monitoring the quantity of water in the domain.

3D Dam break – Free surface flow

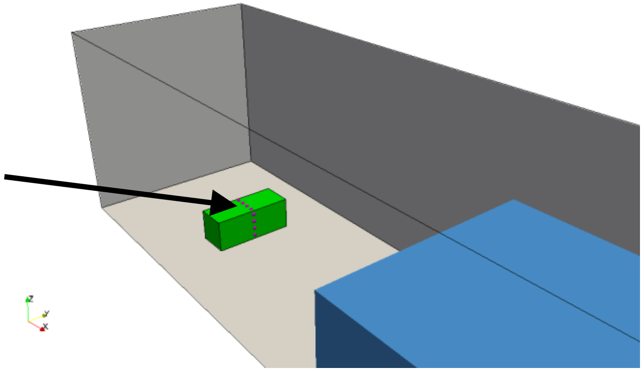
The `controlDict` dictionary

```
55      functions
56      {

107      probes1
108      {
109          type              probes;
110          functionObjectLibs ("libsampling.so");
111
112          pobeLocations
113          (
114              (0.82450002 0 0.021)
115              (0.82450002 0 0.061)
116              (0.82450002 0 0.101)
117              (0.82450002 0 0.141)
118              (0.8035 0 0.161)
119              (0.7635 0 0.161)
120              (0.7235 0 0.161)
121              (0.6835 0 0.161)
122          );
123
124         fields
125         (
126             p p_rgh
127         );
128
129          writeControl    timeStep;
130          writeInterval   1;
131     }

144     };
```

- Let us take a look at the **functionObjects** definitions.
- In lines 107-131 we define the **probes functionObject** which sample the selected fields (lines 124-127) at the selected locations (lines 112-122).
- This sampling is done on-the-fly. All the information sample by this **functionObject** is saved in the directory `./postProcessing/probes1`
- As we are sampling starting from time 0, the sampled data will be located in the directory:

  `postProcessing/probes1/0`

- Feel free to open the files located in the directory `postProcessing/probes1/0` using your favorite text editor.



Sampling locations (probeLocations)

3D Dam break – Free surface flow

The `controlDict` dictionary

```
55      functions
56      {

135      yplus
136      {
137          type              yPlus;
138          functionObjectLibs ("libutilityFunctionObjects.so ");
139           enabled true;
140           writeControl outputTime;
141      }

144     };
```

- Let us take a look at the **functionObjects** definitions.
- In lines 135-141 we define the **yplus functionObject** which computes the yplus value.
- This quantity is related to the turbulence modeling.
- This **functionObject** will save the yplus field in the solution directories with the same saving frequency as the solution (line 140).
- It will also save the minimum, maximum and mean values of yplus in the directory:

  `postProcessing/yplus`

3D Dam break – Free surface flow

The *fvSchemes* dictionary

```
ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
    grad(U)         cellLimited Gauss linear 1;
}

divSchemes
{
    div(rhoPhi,U)  Gauss linearUpwindV grad(U);

    div(phi,alpha)  Gauss interfaceCompression vanLeer 1;

    div(phi,k) Gauss upwind;
    div(phi,epsilon) Gauss upwind;
    div(((rho*nuEff)*dev2(T(grad(U)))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}
```

- In this case, for time discretization (**ddtSchemes**) we are using the **Euler** method.
- For gradient discretization (**gradSchemes**) we are using the **Gauss linear** as the default method and slope limiters (**cellLimited**) for the velocity gradient or **grad(U)**.
- For the discretization of the convective terms (**divSchemes**) we are using **linearUpwindV** interpolation method for the term **div(rhoPhi,U)**.
- For the term **div(phi,alpha)** we are using **interfaceCompression vanLeer** interpolation scheme.
  - This is an interface compression corrected scheme used to maintain sharp interfaces in VOF simulations.
  - The coefficient defines the degree of compression, where 1 is suitable for most VOF applications.
- For the terms **div(phi,k)** and **div(phi,epsilon)** we are using upwind (these terms are related to the turbulence modeling).
- **For the term div(((rho*nuEff)*dev2(T(grad(U))))** we are using **linear** interpolation (this term is related to the turbulence modeling).
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear corrected** method
- In overall, this method is second order accurate but a little bit diffusive. Remember, at the end of the day we want a solution that is second order accurate.

3D Dam break – Free surface flow

The *fvSolution* dictionary

```
17    solvers
18    {
19        "alpha.water.*"
20        {
21            nAlphaCorr      3;
22            nAlphaSubCycles 1;
23            cAlpha          1;
24
25            MULESCorr       yes;
26            nLimiterIter    10;
27
28            solver          smoothSolver;
29            smoother        symGaussSeidel;
30            tolerance       1e-8;
31            relTol          0;
32        }
33
34        “(pcorr|pcorrFinal)”
35        {
36            solver          PCG;
37            preconditioner  DIC;
38            tolerance       1e-8;
39            relTol          0;
40        }
41
42        p_rgh
43        {
44            solver          PCG;
45            preconditioner  DIC;
46            tolerance       1e-06;
47            relTol          0.01;
48            minIter         1;
49        }
```

- To solve the volume fraction or **alpha.water** (lines 19-32) we are using the **smoothSolver** method.
- In line 25 we turn on the semi-implicit method MULES. The keyword **nLimiterIter** controls the number of MULES iterations over the limiter.
- To have more stability it is possible to increase the number of loops and corrections used to solve **alpha.water** (lines 21-22).
- The keyword **cAlpha** (line 23) controls the sharpness of the interface (1 is usually fine for most cases).
- In lines 34-40 we setup the solver for **pcorr** and **pcorrFinal** (pressure correction).
- In this case **pcorr** is solved only one time at the beginning of the computation.
- In lines 42-49 we setup the solver for **p_rgh**.
- The keyword **minIter** (line 48), means that the linear solver will do at least one iteration.

3D Dam break – Free surface flow

The *fvSolution* dictionary

```
51        p_rghFinal
52        {
53            $p_rgh;
54            relTol          0;
55            minIter         1;
56        }
57
58        "(U|UFinal)"
59        {
60            solver          PBiCGStab;
61            Preconditioner  DILU;
62            tolerance       1e-08;
63            relTol          0;
72        }
73
74        "(k|epsilon).*"
75        {
76            solver          PBiCGStab;
77            Preconditioner  DILU;
78            tolerance       1e-08;
79            relTol          0;
80        }
81    }
82
```

- In lines 51-56 we setup the solver for **p_rghFinal**. This correspond to the last iteration in the loop (we can use a tighter convergence criteria to get more accuracy without increasing the computational cost)
- In lines 58-72 we setup the solvers for **U** and **UFInal**.
- In lines 74-80 we setup the solvers for the turbulent quantities, namely, **k** and **epsilon**.

3D Dam break – Free surface flow

The *fvSolution* dictionary

```
82
83    PIMPLE
84    {
85        momentumPredictor   yes;
86        nOuterCorrectors    1;
87        nCorrectors         3;
88        nNonOrthogonalCorrectors 1;
89    }
90
91    relaxationFactors
92    {
93        fields
94        {
95            ".*" 0.9;
96        }
97        equations
98        {
99            ".*" 0.9;
100        }
101    }
102
```

- In lines 83-89 we setup the entries related to the pressure-velocity coupling method used (**PIMPLE** in this case). Setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- To gain more stability we can increase the number of correctors (lines 87-88), however this will increase the computational cost.
- In lines 91-101 we setup the under-relaxation factors related to the **PIMPLE** method outer iterations.
    - The values defined correspond to the industry standard of the **SIMPLEC** method.
    - By using under-relaxation we ensure diagonal equality.
    - Be careful not use too low values as you will loose time accuracy.
    - If you want to disable under-relaxation, comment out these lines.
- The option **momentumPredictor** (line 85), is recommended for highly convective flows.

3D Dam break – Free surface flow

The **`system`** directory

- In the **`system`** directory you will find the following optional dictionary files:
  - *`decomposeParDict`*
  - *`setFieldsDict`*

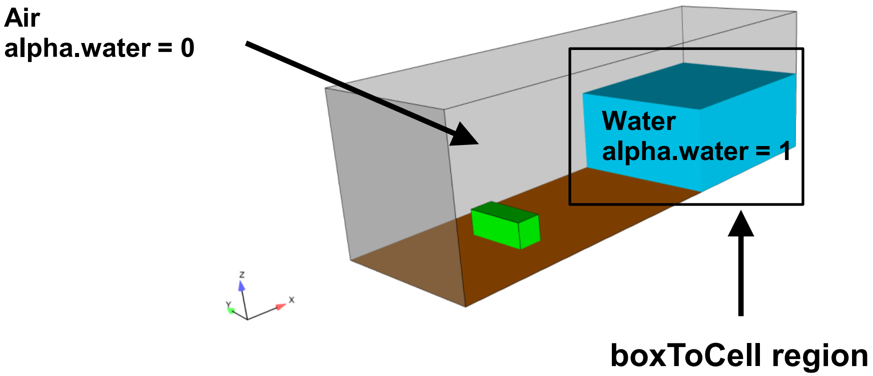
- *`decomposeParDict`* is read by the utility `decomposePar`. This dictionary file contains information related to the mesh partitioning. This is used when running in parallel.
- *`setFieldsDict`* is read by the utility `setFields`. This utility set values on selected cells/faces.

3D Dam break – Free surface flow

The `setFieldsDict` dictionary

```
17    defaultFieldValues
18    (
19        volScalarFieldValue alpha.water 0
20    );
21
22    regions
23    (
24        boxToCell
25        {
26            box (1.992 -10 0) (5 10 0.55);
27            fieldValues
28            (
29                volScalarFieldValue alpha.water 1
30            );
31        }
32    );
```

- This dictionary file is located in the directory **system**.
- In lines 17-20 we set the default value to be 0 in the whole domain (no water).
- In lines 22-32, we initialize a rectangular region (**box**) containing water (**alpha.water 1**).
- In this case, `setFields` will look for the dictionary file *`alpha.water`* and it will overwrite the original values according to the regions defined in *`setFieldsDict`*.
- We initialize the water phase because is the primary phase in the dictionary *`transportProperties`*.
- If you are interested in initializing the vector field **U**, you can proceed as follows **volVectorFieldValue U (0 0 0)**



3D Dam break – Free surface flow

The *decomposeParDict* dictionary

- This dictionary file is located in the directory **system**.
- This dictionary is used to decompose the domain in order to run in parallel.
- The keyword **numberOfSubdomains** (line 17) is used to set the number of cores we want to use in the parallel simulation.
- In this dictionary we also set the decomposition method (line 19).
- Most of the times the scotch method is fine.
- In this case we set the **numberOfSubdomains** to 4, therefore we will run in parallel using 4 cores.

```
17    numberOfSubdomains 4;
18
19    method scotch;
20
```

- When you run in parallel, the solution is saved in the directories **`processorN`**, where **`N`** stands for processor number. In this case you will find the following directories with the decomposed mesh and solution: **`processor0`**, **`processor1`**, **`processor2`**, and **`processor3`**.

3D Dam break – Free surface flow

Running the case

- Let us first generate the mesh.
- To generate the mesh will use snappyHexMesh (sHM), do not worry we will talk about sHM tomorrow.

```
1. $> foamCleanTutorials
2. $> rm –rf 0
3. $> blockMesh
4. $> surfaceFeatures
5. $> snappyHexMesh -overwrite
6. $> createPatch -dict system/createPatchDict.0 -overwrite
7. $> createPatch -dict system/createPatchDict.1 -overwrite
8. $> checkMesh
9. $> paraFoam
```

3D Dam break – Free surface flow

Running the case

- Let us run the simulation in parallel using the solver `interFoam`.
- We will talk more about running in parallel tomorrow
- To run the case, type in the terminal:

```
1.  $> rm -rf 0
2.  $> cp -r 0_org 0
3.  $> setFields
4.  $> paraFoam
5.  $> decomposePar
6.  $> mpirun -np 4 interFoam -parallel | tee log.interFoam
7.  $> reconstructPar
8.  $> paraFoam
```

3D Dam break – Free surface flow

Running the case

- In steps 1-2 we copy the information of the backup directory **`0_org`** into the directory **`0`**. We do this because in the next step the utility `setFields` will overwrite the file *`0/alpha.water`*, so it is a good idea to keep a backup.
- In step 3 we initialize the solution using the utility `setFields`. This utility reads the dictionary *`setFieldsDict`* located in the **`system`** directory.
- In step 4 we visualize the initialization using paraFoam.
- In step 5 we use the utility `decomposePar` to do the domain decomposition needed to run in parallel.
- In step 6 we run the simulation in parallel. Notice that np means number of processors and the value used should be the same number as the one you set in the dictionary *`decomposeParDict`*.
- If you want to run in serial, type in the terminal: `interFoam | tee log`
- In step 7 we reconstruct the parallel solution. This step is only needed if you are running in parallel.
- Finally, in step 8 we visualize the solution.

3D Dam break – Free surface flow

- To plot the sampled data using gnuplot you can proceed as follows. To enter to the gnuplot prompt type in the terminal:

```
$> gnuplot
```

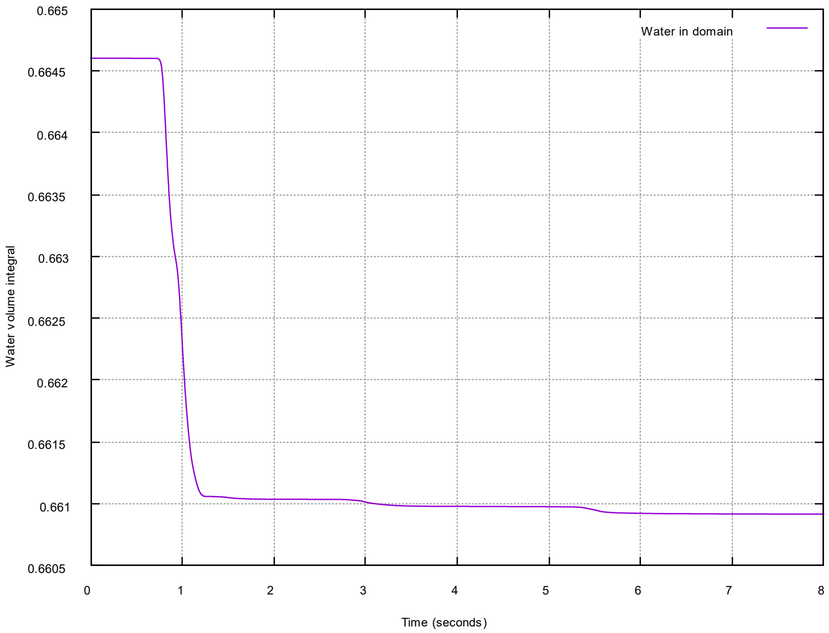
- Now that we are inside the gnuplot prompt, we can type,

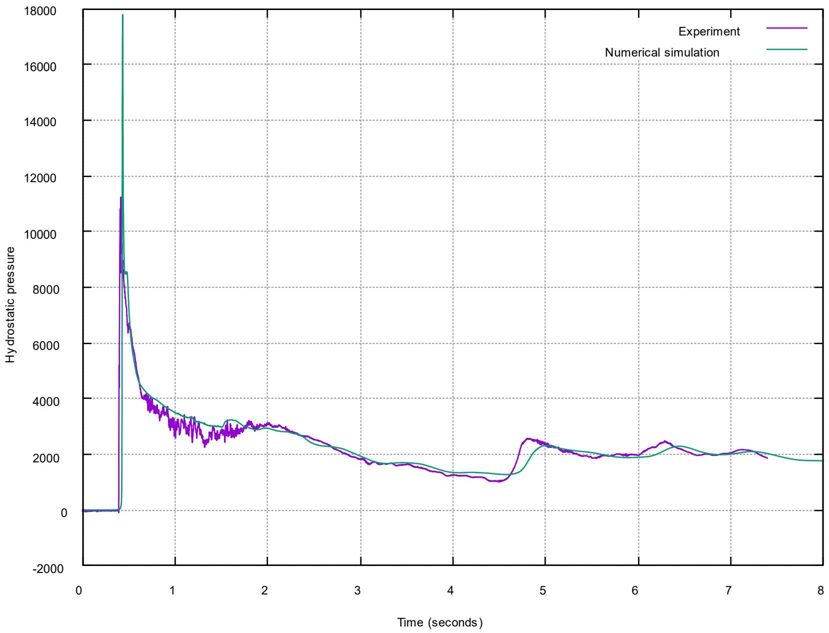
```
set xlabel 'Time (seconds)'

set ylabel 'Water volume integral'

gnuplot> plot 'postProcessing/water_in_domain/0/volRegion.dat' u 1:2 w l title
'Water in domain'
set xlabel 'Time (seconds)'

set ylabel 'Pressure'

plot 'SPHERIC_Test2/case.txt' u 1:2 w l title 'Experiment',
'postProcessing/probes1/0/p' u 1:2 w l title 'Numerical simulation'
gnuplot> exit
```

To exit gnuplot

3D Dam break – Free surface flow

- The output of steps 3 and 6 is the following:



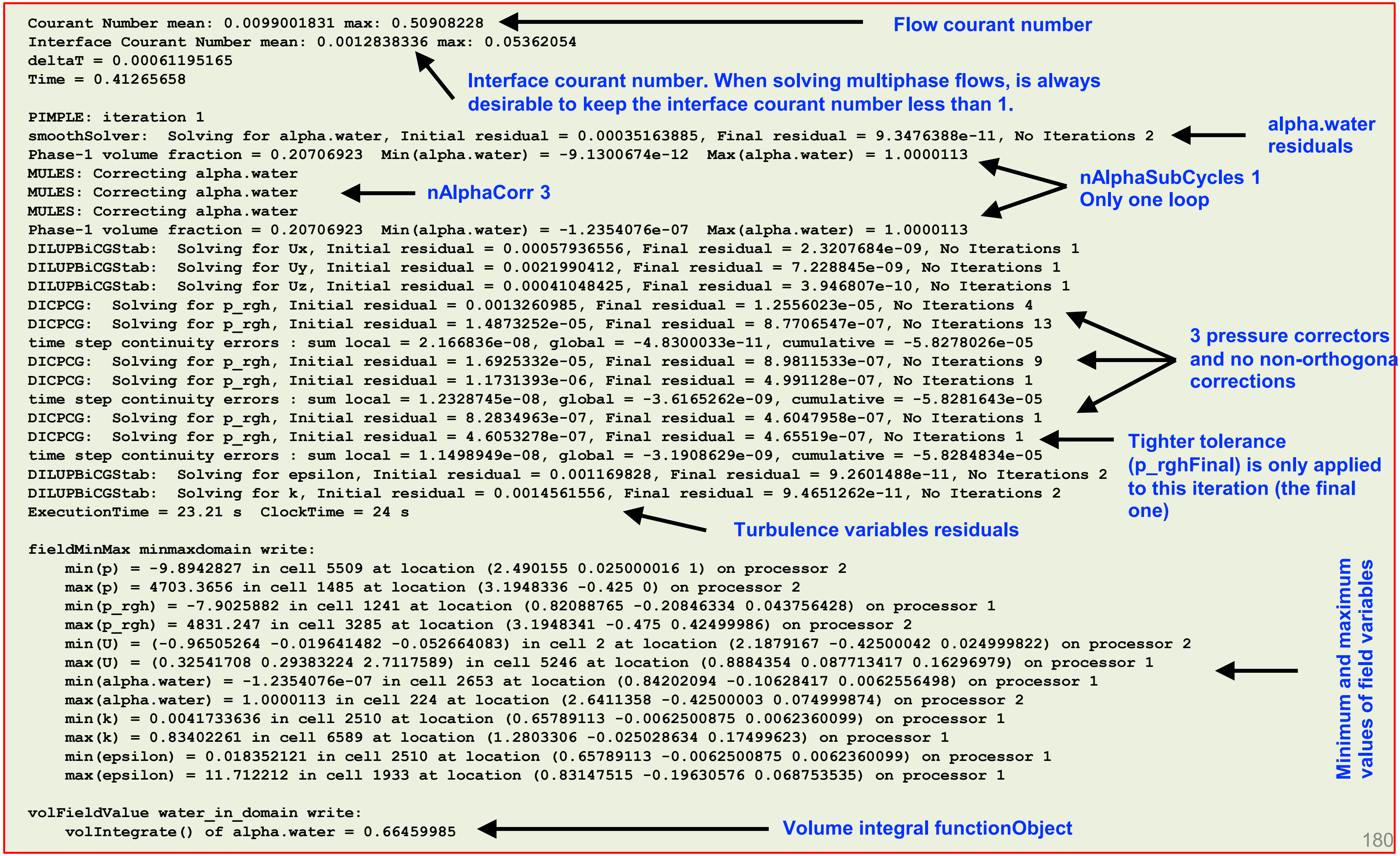


alpha.water vs. time

p vs. time (at probe 0)

3D Dam break – Free surface flow

The output screen



```
Courant Number mean: 0.0099001831 max: 0.50908228
Interface Courant Number mean: 0.0012838336 max: 0.05362054
deltaT = 0.00061195165
Time = 0.41265658

PIMPLE: iteration 1
smoothSolver:  Solving for alpha.water, Initial residual = 0.00035163885, Final residual = 9.3476388e-11, No Iterations 2
Phase-1 volume fraction = 0.20706923  Min(alpha.water) = -9.1300674e-12  Max(alpha.water) = 1.0000113
MULES: Correcting alpha.water
MULES: Correcting alpha.water
MULES: Correcting alpha.water
Phase-1 volume fraction = 0.20706923  Min(alpha.water) = -1.2354076e-07  Max(alpha.water) = 1.0000113
DILUPBiCGStab:  Solving for Ux, Initial residual = 0.00057936556, Final residual = 2.3207684e-09, No Iterations 1
DILUPBiCGStab:  Solving for Uy, Initial residual = 0.0021990412, Final residual = 7.228845e-09, No Iterations 1
DILUPBiCGStab:  Solving for Uz, Initial residual = 0.00041048425, Final residual = 3.946807e-10, No Iterations 1
DICPCG:  Solving for p_rgh, Initial residual = 0.0013260985, Final residual = 1.2556023e-05, No Iterations 4
DICPCG:  Solving for p_rgh, Initial residual = 1.4873252e-05, Final residual = 8.7706547e-07, No Iterations 13
time step continuity errors : sum local = 2.166836e-08, global = -4.8300033e-11, cumulative = -5.8278026e-05
DICPCG:  Solving for p_rgh, Initial residual = 1.6925332e-05, Final residual = 8.9811533e-07, No Iterations 9
DICPCG:  Solving for p_rgh, Initial residual = 1.1731393e-06, Final residual = 4.991128e-07, No Iterations 1
time step continuity errors : sum local = 1.2328745e-08, global = -3.6165262e-09, cumulative = -5.8281643e-05
DICPCG:  Solving for p_rgh, Initial residual = 8.2834963e-07, Final residual = 4.6047958e-07, No Iterations 1
DICPCG:  Solving for p_rgh, Initial residual = 4.6053278e-07, Final residual = 4.65519e-07, No Iterations 1
time step continuity errors : sum local = 1.1498949e-08, global = -3.1908629e-09, cumulative = -5.8284834e-05
DILUPBiCGStab:  Solving for epsilon, Initial residual = 0.001169828, Final residual = 9.2601488e-11, No Iterations 2
DILUPBiCGStab:  Solving for k, Initial residual = 0.0014561556, Final residual = 9.4651262e-11, No Iterations 2
ExecutionTime = 23.21 s  ClockTime = 24 s

fieldMinMax minmaxdomain write:
    min(p) = -9.8942827 in cell 5509 at location (2.490155 0.025000016 1) on processor 2
    max(p) = 4703.3656 in cell 1485 at location (3.1948336 -0.425 0) on processor 2
    min(p_rgh) = -7.9025882 in cell 1241 at location (0.82088765 -0.20846334 0.043756428) on processor 1
    max(p_rgh) = 4831.247 in cell 3285 at location (3.1948341 -0.475 0.42499986) on processor 2
    min(U) = (-0.96505264 -0.019641482 -0.052664083) in cell 2 at location (2.1879167 -0.42500042 0.024999822) on processor 2
    max(U) = (0.32541708 0.29383224 2.7117589) in cell 5246 at location (0.8884354 0.087713417 0.16296979) on processor 1
    min(alpha.water) = -1.2354076e-07 in cell 2653 at location (0.84202094 -0.10628417 0.0062556498) on processor 1
    max(alpha.water) = 1.0000113 in cell 224 at location (2.6411358 -0.42500003 0.074999874) on processor 2
    min(k) = 0.0041733636 in cell 2510 at location (0.65789113 -0.0062500875 0.0062360099) on processor 1
    max(k) = 0.83402261 in cell 6589 at location (1.2803306 -0.025028634 0.17499623) on processor 1
    min(epsilon) = 0.018352121 in cell 2510 at location (0.65789113 -0.0062500875 0.0062360099) on processor 1
    max(epsilon) = 11.712212 in cell 1933 at location (0.83147515 -0.19630576 0.068753535) on processor 1

volFieldValue water_in_domain write:
    volIntegrate() of alpha.water = 0.66459985
```

Flow courant number

Interface courant number. When solving multiphase flows, is always desirable to keep the interface courant number less than 1.

alpha.water residuals

nAlphaSubCycles 1
Only one loop

nAlphaCorr 3

3 pressure correctors and no non-orthogonal corrections

Tighter tolerance (p_rghFinal) is only applied to this iteration (the final one)

Turbulence variables residuals

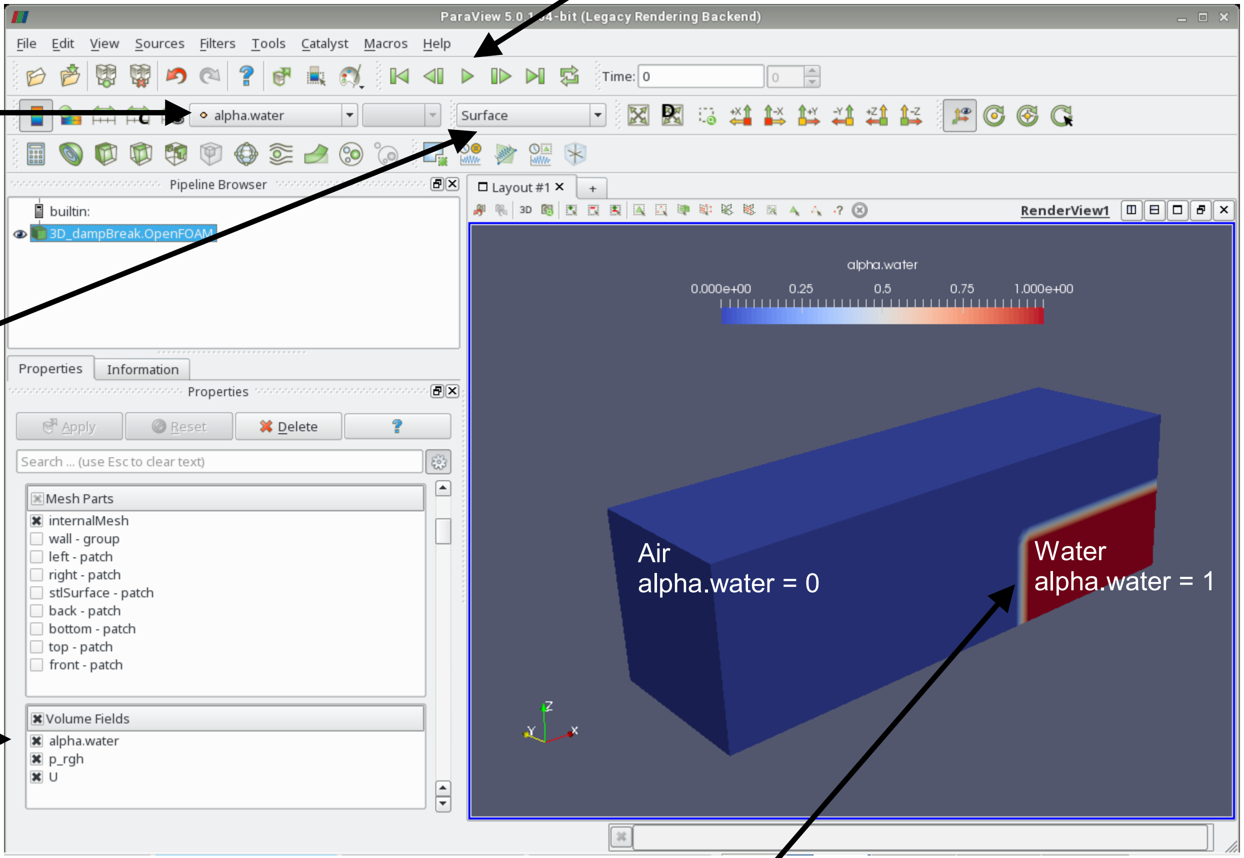
Minimum and maximum values of field variables

Volume integral functionObject

3D Dam break – Free surface flow

Post-processing multiphase flows in paraFoam

- To visualize the volume fraction, proceed as follows,



4. To animate the solution, press `Play` in the VCR Controls

2. Select **alpha.water** in the Active Variable drop-down menu

3. Select `Surface` in the Representation drop-down menu

1. In the Properties tab select **alpha.water** in Volume Fields

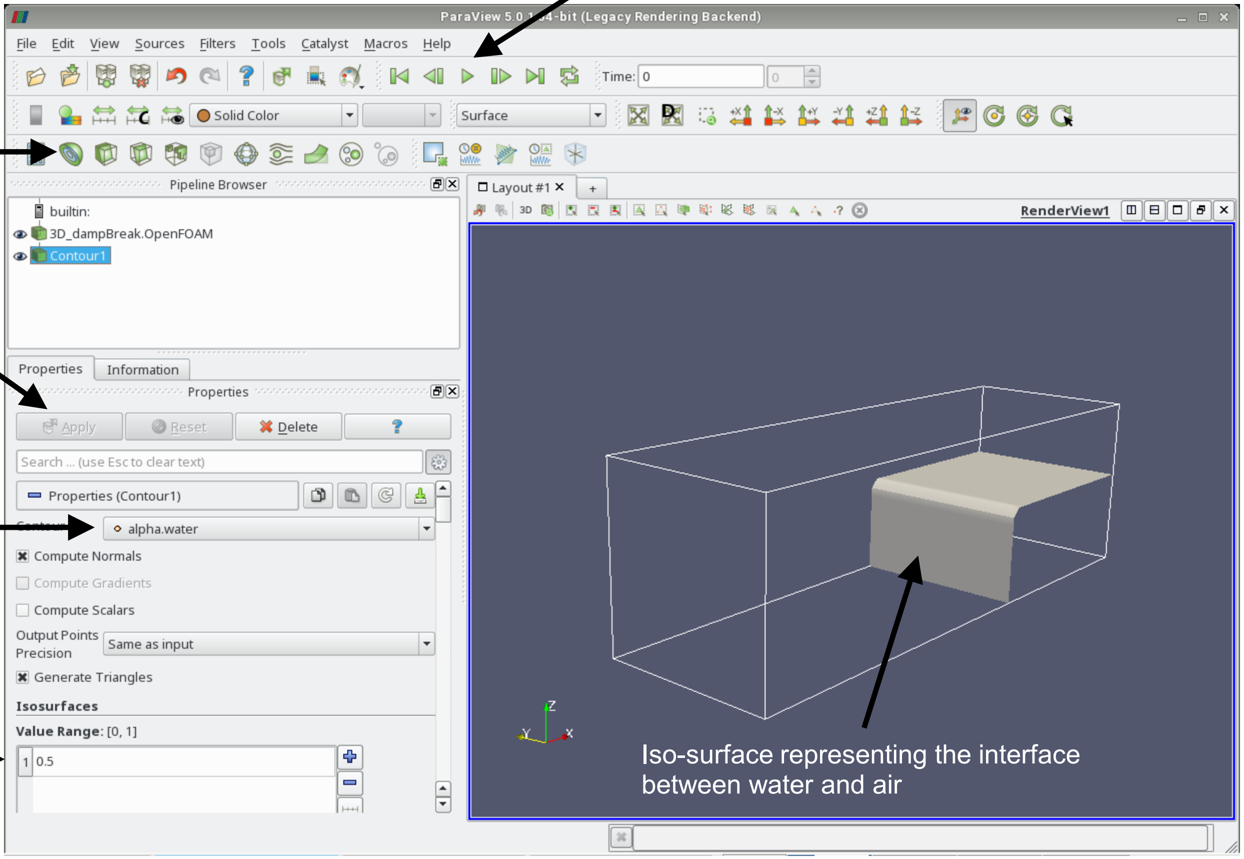
Interface
alpha.water = 0.5

3D Dam break – Free surface flow

Post-processing multiphase flows in paraFoam

- To visualize a surface representing the interface, proceed as follows,

5. To animate the solution, press `Play` in the VCR Controls



Properties (Contour1)

Contour By: alpha.water

☒ Compute Normals

☐ Compute Gradients

☐ Compute Scalars

Output Points Precision: Same as input

☒ Generate Triangles

Isosurfaces

Value Range: [0, 1]

1 0.5

Iso-surface representing the interface between water and air

1. Select the filter `Contour`

4. Press apply

2. Select **alpha.water** or the field you want to use to plot the iso-surface (it has to be a scalar)

3. Enter the value 0.5 which corresponds to the interface between water and air

3D Dam break – Free surface flow

Post-processing multiphase flows in paraFoam

- To visualize all the cells representing the water fraction, proceed as follows,

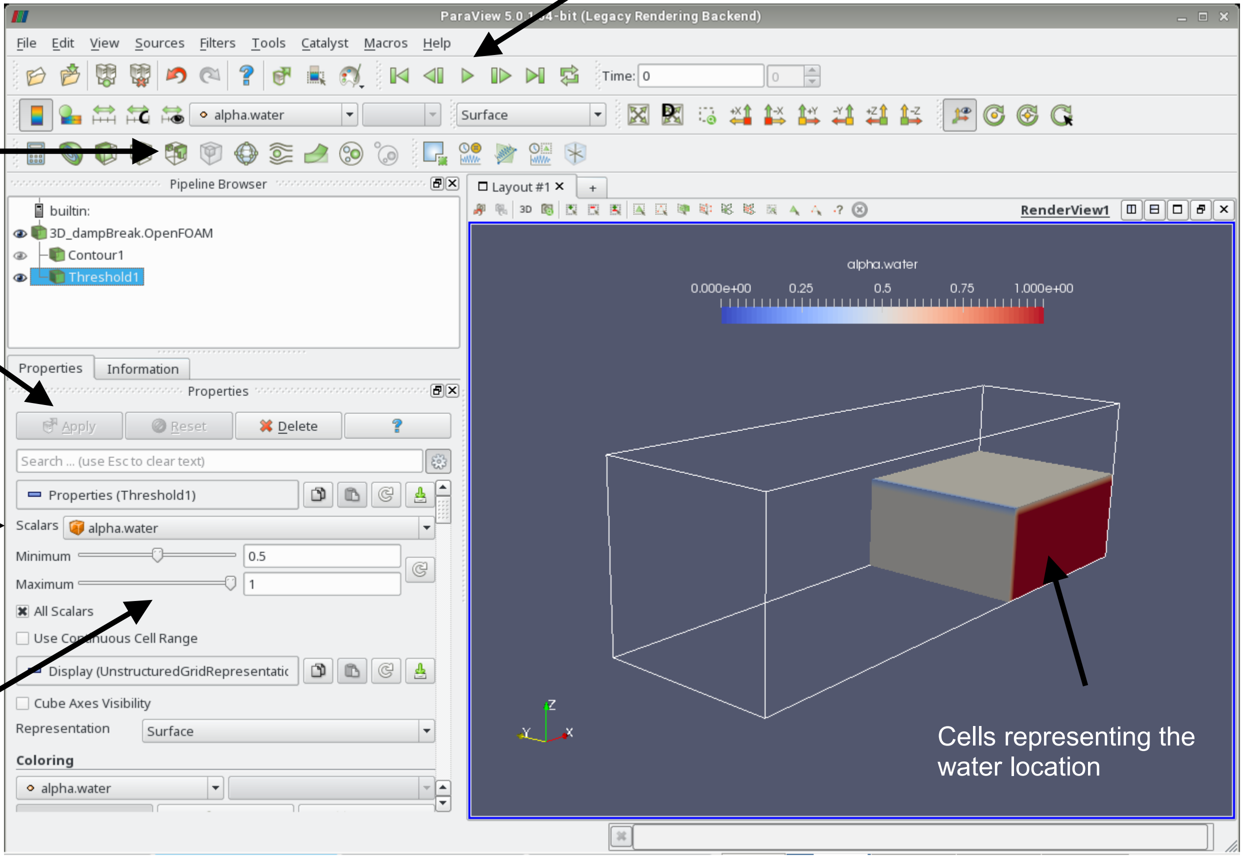
5. To animate the solution, press `Play` in the VCR Controls

1. Select the filter `Threshold`

4. Press apply

2. Select **alpha.water** or the field you want to use to visualize the cells (it has to be a scalar)

3. Select the range you want to visualize. To visualize the water select `Minimum` 0.5 and `Maximum` 1.



Time: 0

Pipeline Browser

builtin:
3D_dampBreak.OpenFOAM
Contour1
Threshold1

Properties (Threshold1)

Scalars: alpha.water

Minimum: 0.5

Maximum: 1

☒ All Scalars

□ Use Continuous Cell Range

Display (UnstructuredGridRepresentatic

□ Cube Axes Visibility

Representation: Surface

Coloring: alpha.water

Cells representing the water location

3D Dam break – Free surface flow

Exercises

- Instead of using the boundary condition **totalPressure** and **pressureInletOutletVelocity** for the patch **top**, try to use **zeroGradient**. Do you get the same results? Any comments?
  (Hint: this combination of boundary conditions might give you an error, if so, read carefully the screen and try to find a fix, you can start by looking at the file *`fvSolution`*)
- Instead of using the boundary condition **fixedFluxPressure** for the walls, try to use **zeroGradient**. Do you get the same results? Any comments?
- Run the simulation in a close domain. Does the volume integral of **alpha.water** remains the same? Why the value is not constant when the domain is open?
- Use a **functionObject** to measure the average pressure on the obstacle.
- How many initialization methods are there available in the dictionary *`setFieldsDict`*?
  (Hint: use the banana method)
- Run the simulation using **Gauss upwind** instead of **Gauss vanLeer** or **Gauss interfaceCompression vanLeer 1** for the term **div(phi,alpha)** (`fvSchemes`). Do you get the same quantitative results?

3D Dam break – Free surface flow

Exercises

- Run a numerical experiment for **cAlpha** equal to **0**, **1**, and **2**. Do you see any difference in the solution? What about computing time?
- Use the solver **GAMG** instead of using the solver **PCG** for the variable **p_rgh**. Do you see any difference on the solution or computing time?
- Increase the number of **nOuterCorrector** to 2 and study the output screen. What difference do you see?
- Turn off the MULES corrector (**MULESCorr**). Do you see any difference on the solution or computing time?
- If you set the gravity vector to (0 0 0), what do you think will happen?
- Try to break the solver and identify the cause of the error. You are free to try any kind of setup.

Roadmap

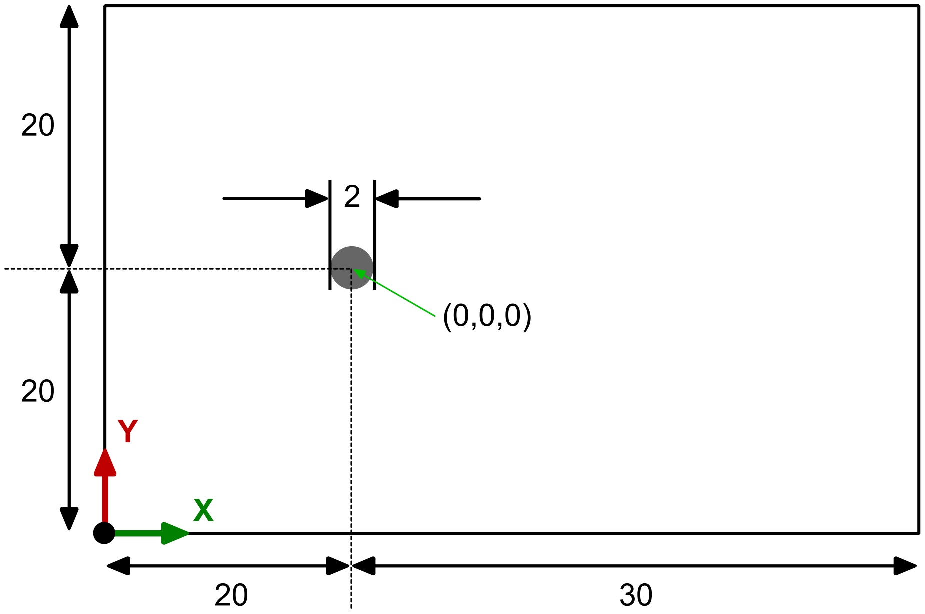
1. ~~OpenFOAM® brief overview~~
2. ~~OpenFOAM® directory organization~~
3. ~~Directory structure of an application/utility~~
4. ~~Applications/utilities in OpenFOAM®~~
5. ~~Directory structure of an OpenFOAM® case~~
6. ~~Running my first OpenFOAM® case setup blindfold~~
7. ~~A deeper view to my first OpenFOAM® case setup~~
8. ~~3D Dam break – Free surface flow~~
9. **Flow past a cylinder – From laminar to turbulent flow**

Flow past a cylinder – From laminar to turbulent flow

- At this point we all have a rough idea of what is going on with all these dictionary files.
- Unless it is strictly necessary, from now on we will not go into details about the dictionaries and files we are using.
- Remember, if you are using the lab computers, do not forget to load the environment variables.

Flow past a cylinder – From laminar to turbulent flow

Flow around a cylinder – 10 < Re < 2 000 000
Incompressible and compressible flow



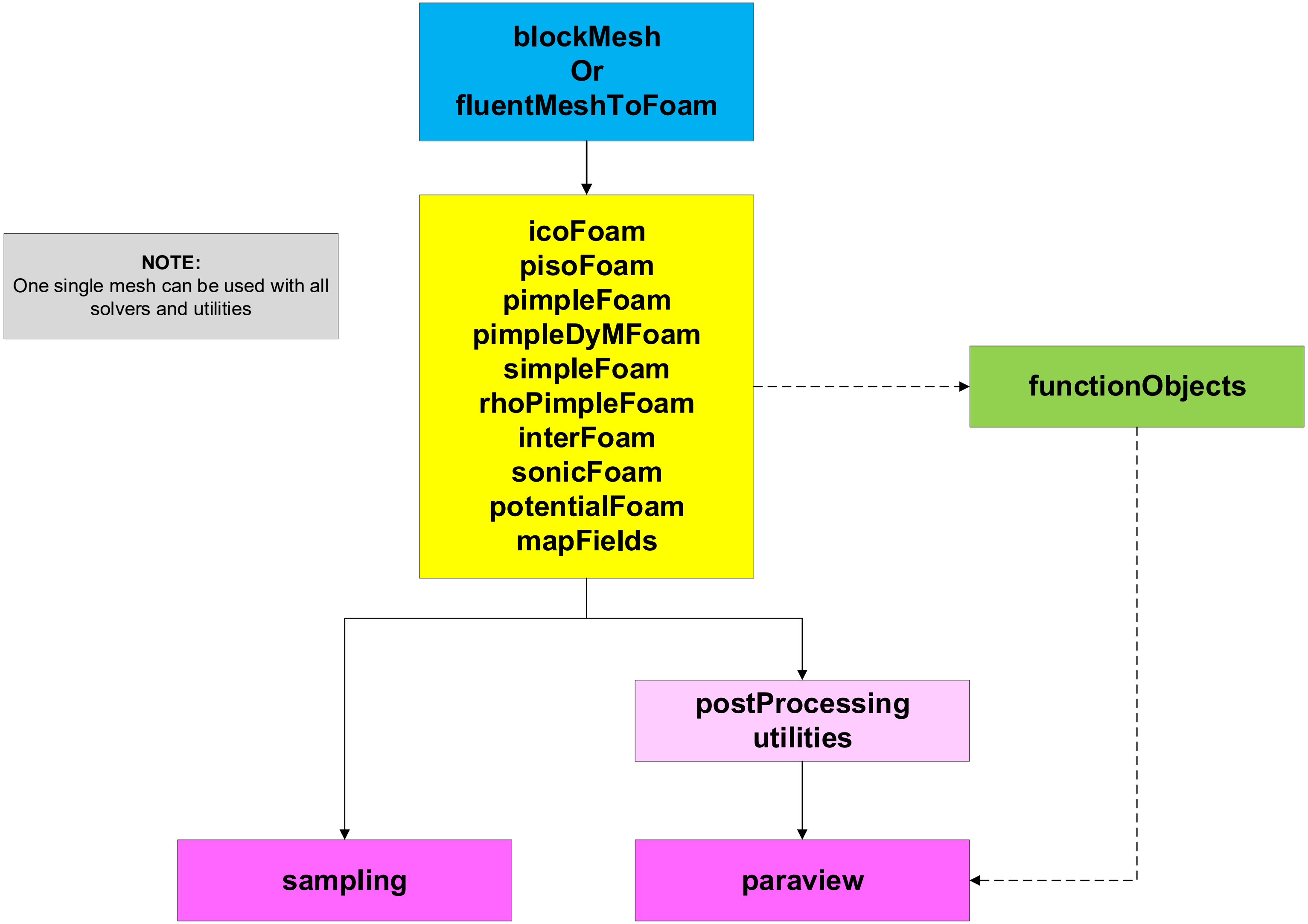
All the dimensions are in meters

Physical and numerical side of the problem:

- In this case we are going to solve the flow around a cylinder. We are going to use incompressible and compressible solvers, in laminar and turbulent regime.
- Therefore, the governing equations of the problem are the incompressible/compressible laminar/turbulent Navier-Stokes equations.
- We are going to work in a 2D domain.
- Depending on the Reynolds number, the flow can be steady or unsteady.
- This problem has a lot of validation data.

Flow past a cylinder – From laminar to turbulent flow

Workflow of the case



blockMesh
Or
fluentMeshToFoam

icoFoam
pisoFoam
pimpleFoam
pimpleDyMFoam
simpleFoam
rhoPimpleFoam
interFoam
sonicFoam
potentialFoam
mapFields

NOTE:
One single mesh can be used with all solvers and utilities

functionObjects

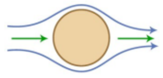
postProcessing
utilities

sampling

paraview

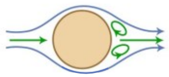
Flow past a cylinder – From laminar to turbulent flow

Vortex shedding behind a cylinder



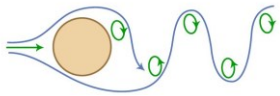
Creeping flow (no separation)
Steady flow

$Re < 5$



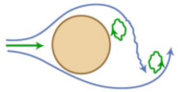
A pair of stable vortices in the wake
Steady flow

$5 < Re < 40 - 46$



Laminar vortex street (Von Karman street)
Unsteady flow

$40 - 46 < Re < 150$

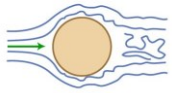


Laminar boundary layer up to the separation point, turbulent wake
Unsteady flow

$150 < Re < 300$

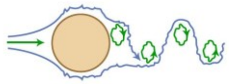
Transition to turbulence

$300 < Re < 3 \times 10^5$



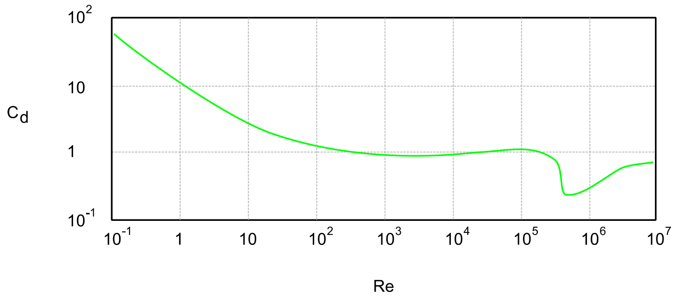
Boundary layer transition to turbulent
Unsteady flow

$3 \times 10^5 < Re < 3 \times 10^6$

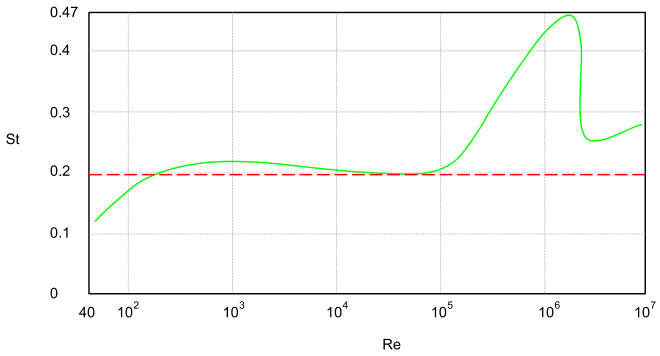


Turbulent vortex street, but the wake is narrower than in the laminar case
Unsteady flow

$3 \times 10^6 > Re$



Drag coefficient



Strouhal number

Flow past a cylinder – From laminar to turbulent flow

Some experimental (E) and numerical (N) results of the flow past a circular cylinder at various Reynolds numbers

Reference	c_d – Re = 20	L_{rb} – Re = 20	c_d – Re = 40	L_{rb} – Re = 40
[1] Tritton (E)	2.22	–	1.48	–
[2] Cuntanceau and Bouard (E)	–	0.73	–	1.89
[3] Russel and Wang (N)	2.13	0.94	1.60	2.29
[4] Calhoun and Wang (N)	2.19	0.91	1.62	2.18
[5] Ye et al. (N)	2.03	0.92	1.52	2.27
[6] Fornbern (N)	2.00	0.92	1.50	2.24
[7] Guerrero (N)	2.20	0.92	1.62	2.21

$\mathbf{L_{rb}}$ = length of recirculation bubble, $\mathbf{c_d}$ = drag coefficient, **Re** = Reynolds number,

[1] D. Tritton. Experiments on the flow past a circular cylinder at low Reynolds numbers. Journal of Fluid Mechanics, 6:547-567, 1959.

[2] M. Cuntanceau and R. Bouard. Experimental determination of the main features of the viscous flow in the wake of a circular cylinder in uniform translation. Part 1. Steady flow. Journal of Fluid Mechanics, 79:257-272, 1973.

[3] D. Rusell and Z. Wang. A cartesian grid method for modeling multiple moving objects in 2D incompressible viscous flow. Journal of Computational Physics, 191:177-205, 2003.

[4] D. Calhoun and Z. Wang. A cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. Journal of Computational Physics. 176:231-275, 2002.

[5] T. Ye, R. Mittal, H. Udaykumar, and W. Shyy. An accurate cartesian grid method for viscous incompressible flows with complex immersed boundaries. Journal of Computational Physics, 156:209-240, 1999.

[6] B. Fornberg. A numerical study of steady viscous flow past a circular cylinder. Journal of Fluid Mechanics, 98:819-855, 1980.

[7] J. Guerrero. Numerical simulation of the unsteady aerodynamics of flapping flight. PhD Thesis, University of Genoa, 2009.

Flow past a cylinder – From laminar to turbulent flow

Some experimental (E) and numerical (N) results of the flow past a circular cylinder at various Reynolds numbers

Reference	c_d – Re = 100	c_l – Re = 100	c_d – Re = 200	c_l – Re = 200
[1] Russel and Wang (N)	1.38 ± 0.007	± 0.322	1.29 ± 0.022	± 0.50
[2] Calhoun and Wang (N)	1.35 ± 0.014	± 0.30	1.17 ± 0.058	± 0.67
[3] Braza et al. (N)	1.386± 0.015	± 0.25	1.40 ± 0.05	± 0.75
[4] Choi et al. (N)	1.34 ± 0.011	± 0.315	1.36 ± 0.048	± 0.64
[5] Liu et al. (N)	1.35 ± 0.012	± 0.339	1.31 ± 0.049	± 0.69
[6] Guerrero (N)	1.38 ± 0.012	± 0.333	1.408 ± 0.048	± 0.725

c_l = lift coefficient, **c_d** = drag coefficient, **Re** = Reynolds number

[1] D. Rusell and Z. Wang. A cartesian grid method for modeling multiple moving objects in 2D incompressible viscous flow. Journal of Computational Physics, 191:177-205, 2003.

[2] D. Calhoun and Z. Wang. A cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. Journal of Computational Physics. 176:231-275, 2002.

[3] M. Braza, P. Chassaing, and H. Hinh. Numerical study and physical analysis of the pressure and velocity fields in the near wake of a circular cylinder. Journal of Fluid Mechanics, 165:79-130, 1986.

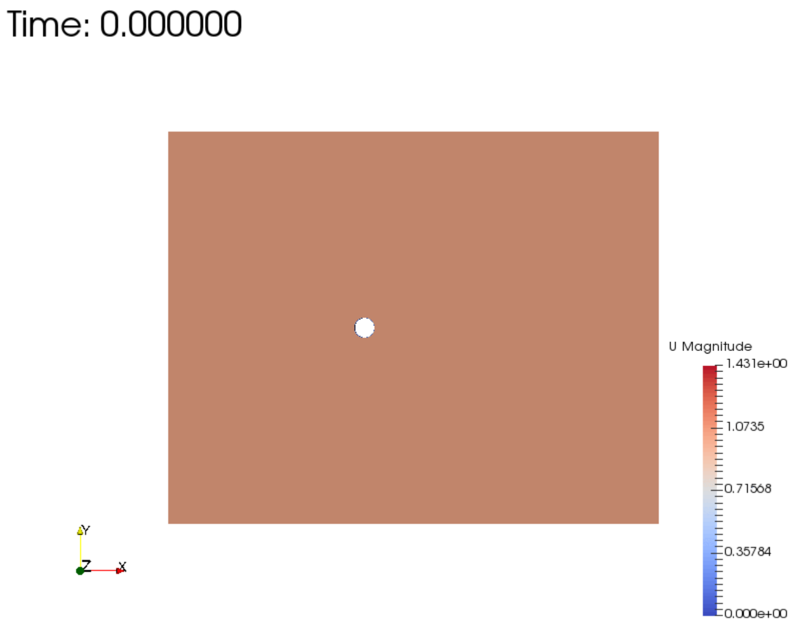
[4] J. Choi, R. Oberoi, J. Edwards, an J. Rosati. An immersed boundary method for complex incompressible flows. Journal of Computational Physics, 224:757-784, 2007.

[5] C. Liu, X. Zheng, and C. Sung. Preconditioned multigrid methods for unsteady incompressible flows. Journal of Computational Physics, 139:33-57, 1998.

[6] J. Guerrero. Numerical Simulation of the unsteady aerodynamics of flapping flight. PhD Thesis, University of Genoa, 2009.

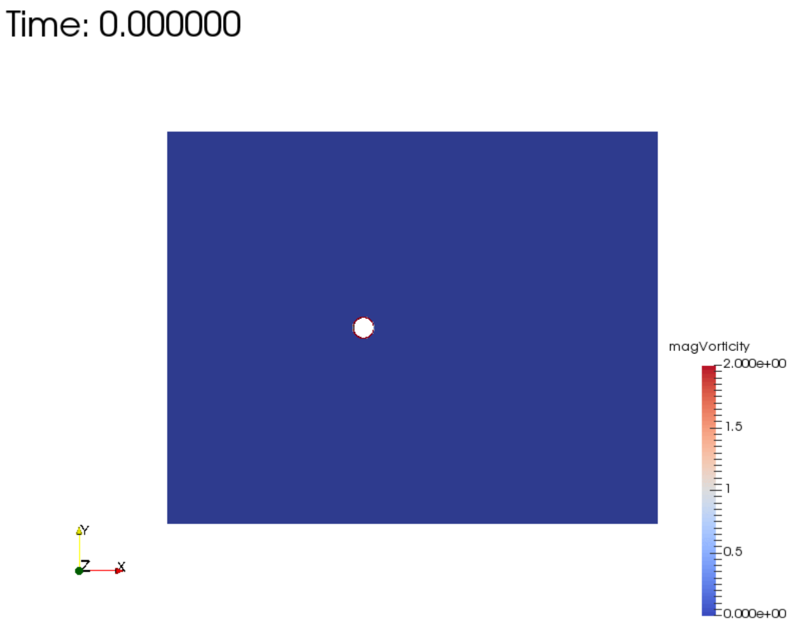
Flow past a cylinder – From laminar to turbulent flow

At the end of the day, you should get something like this



Instantaneous velocity magnitude field

www.wolfdynamics.com/wiki/cylinder_vortex_shedding/movvmag.gif



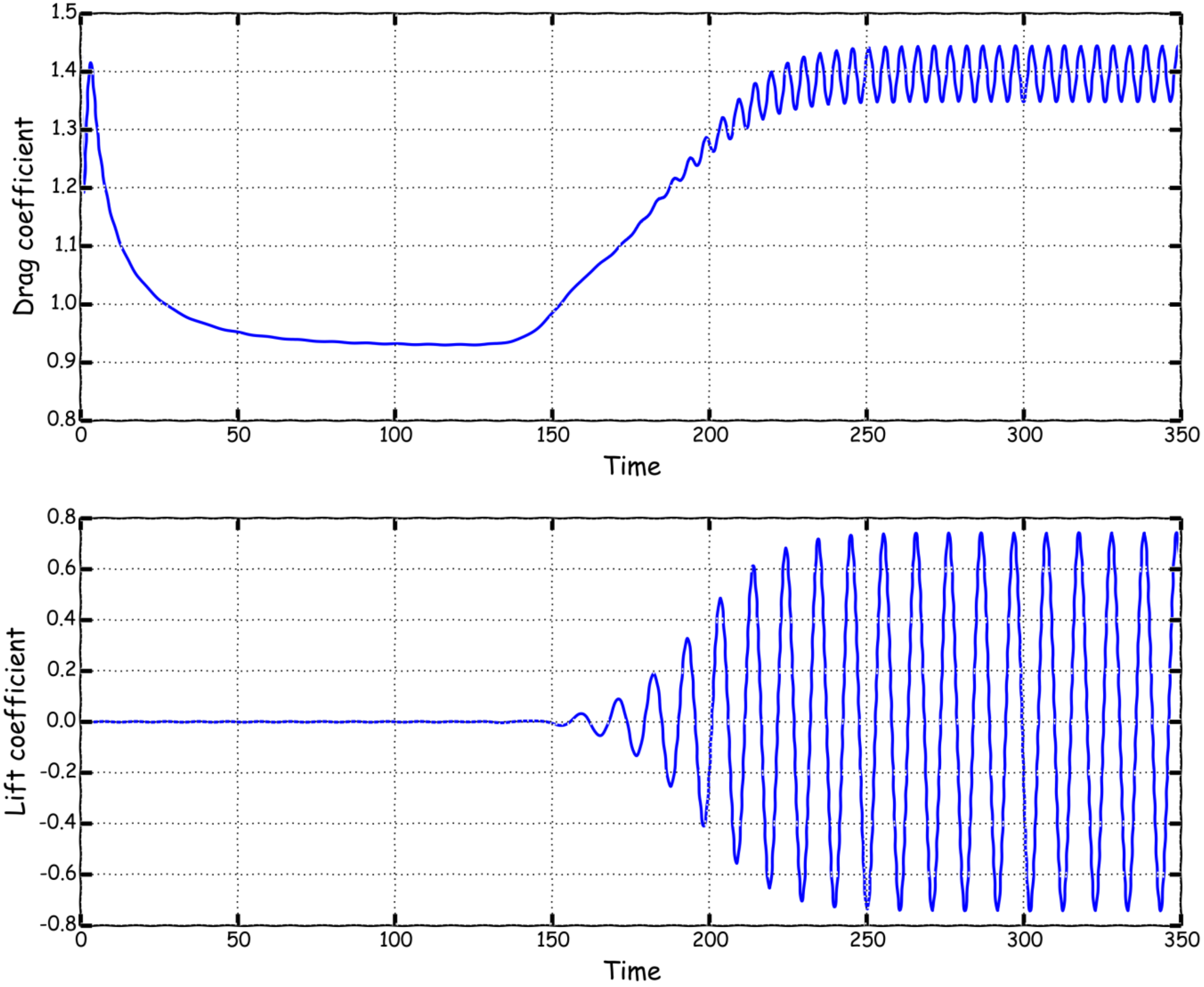
Instantaneous vorticity magnitude field

www.wolfdynamics.com/wiki/cylinder_vortex_shedding/movvort.gif

Incompressible flow – Reynolds 200

Flow past a cylinder – From laminar to turbulent flow

At the end of the day, you should get something like this



Incompressible flow – Reynolds 200

Flow past a cylinder – From laminar to turbulent flow

- Let us run this case. Go to the directory:

```
$PTOFC/101OF/vortex_shedding
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Flow past a cylinder – From laminar to turbulent flow

What are we going to do?

- We will use this case to learn how to use different solvers and utilities.
- Remember, different solvers have different input dictionaries.
- We will learn how to convert the mesh from a third-party software.
- We will learn how to use `setFields` to initialize the flow field and accelerate the convergence.
- We will learn how to map a solution from a coarse mesh to a fine mesh.
- We will learn how to setup a compressible solver.
- We will learn how to setup a turbulence case.
- We will use gnuplot to plot and compute the mean values of the lift and drag coefficients.
- We will visualize unsteady data.

Flow past a cylinder – From laminar to turbulent flow

Running the case

- Let us first convert the mesh from a third-party format (Fluent format).
- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c2`**
- In the terminal window type:

1. `$> foamCleanTutorials`
2. `$> fluent3DMeshToFoam ../../../meshes_and_geometries/vortex_shedding/ascii.msh`
3. `$> checkMesh`
4. `$> paraFoam`

- In step 2, we convert the mesh from Fluent format to OpenFOAM® format. Have in mind that the Fluent mesh must be in ascii format.
- If we try to open the mesh using `paraFoam` (step 4), it will crash. Can you tell what is the problem by just reading the screen?

Flow past a cylinder – From laminar to turbulent flow

Running the case

- To avoid this problem, type in the terminal,

1.
```
$> paraFoam -builtin
```

- Basically, the problem is related to the names and type of the patches in the file *boundary* and the boundary conditions (*U*, *p*). Notice that OpenFOAM® is telling you what and where is the error.

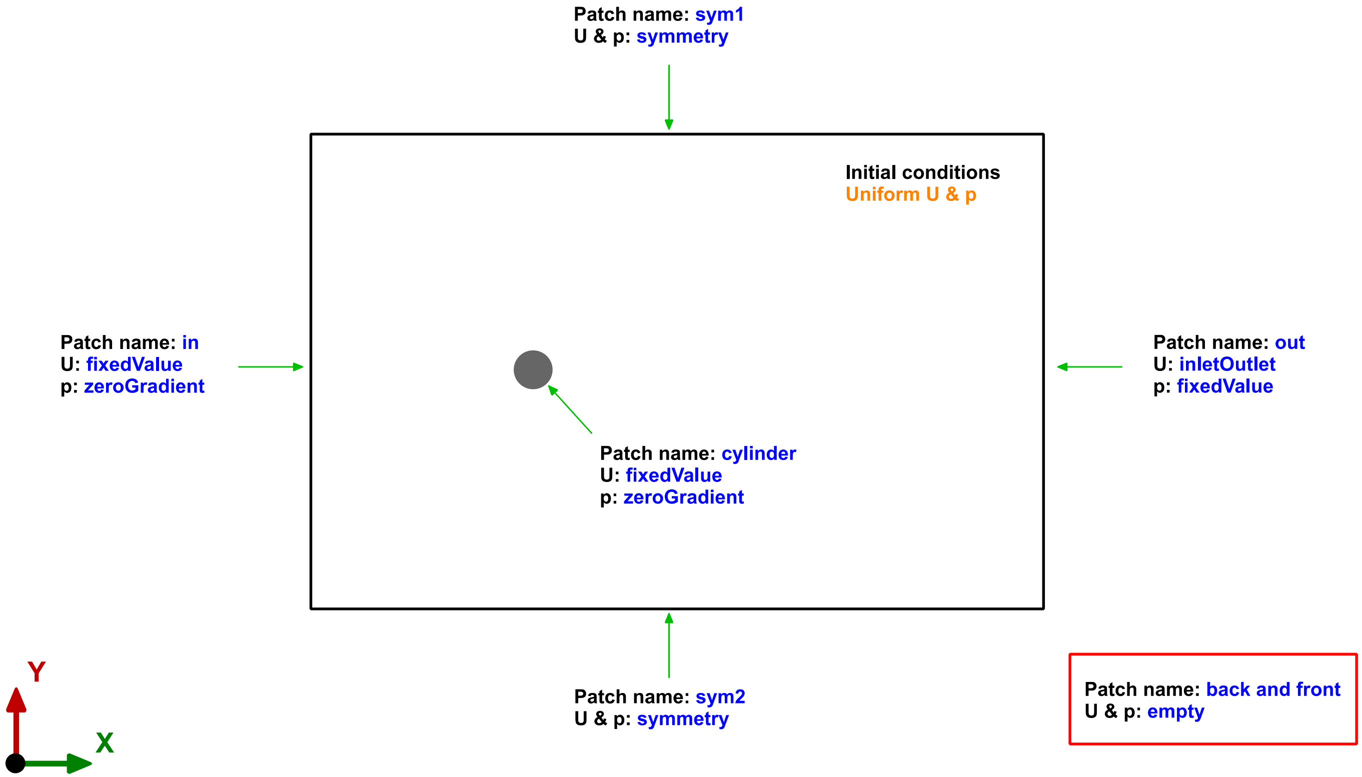
```
Created temporary 'c2.OpenFOAM'


--> FOAM FATAL IO ERROR:

    patch type 'patch' not constraint type 'empty'
    for patch front of field p in file "/home/joegi/my_cases_course/8/101OF/vortex_shedding/c2/0/p"

file: /home/joegi/my_cases_course/8/101OF/vortex_shedding/c2/0/p.boundaryField.front from line 60 to line 60.

    From function Foam::emptyFvPatchField<Type>::emptyFvPatchField(const Foam::fvPatch&, const
Foam::DimensionedField<Type, Foam::volMesh>&, const Foam::dictionary&) [with Type = double]
    in file fields/fvPatchFields/constraint/empty/emptyFvPatchField.C at line 80.

FOAM exiting
```

What

Where

Flow past a cylinder – From laminar to turbulent flow

- Remember, when converting meshes the **name** and **type** of the patches are not always set as you would like, so it is always a good idea to take a look at the file `boundary` and modify it according to your needs.
- Let us modify the `boundary` dictionary file.
- In this case, we would like to setup the following **numerical type** boundary conditions.



Patch name: sym1
U & p: symmetry

Initial conditions
Uniform U & p

Patch name: in
U: fixedValue
p: zeroGradient

Patch name: out
U: inletOutlet
p: fixedValue

Patch name: cylinder
U: fixedValue
p: zeroGradient

Patch name: sym2
U & p: symmetry

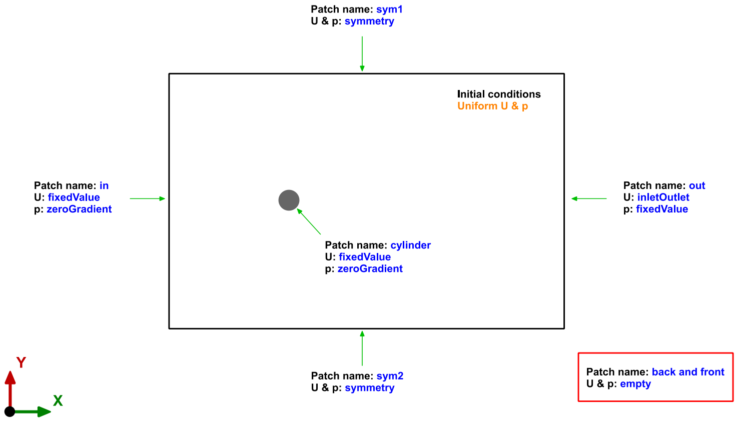
Patch name: back and front
U & p: empty

Flow past a cylinder – From laminar to turbulent flow

The *boundary* dictionary file

```
7
(
    out
    {
        type            patch;
        nFaces          80;
        startFace       18180;
    }
    sym1
    {
        type            symmetry;
        inGroups        1(symmetry);
        nFaces          100;
        startFace       18260;
    }
    sym2
    {
        type            symmetry;
        inGroups        1(symmetry);
        nFaces          100;
        startFace       18360;
    }
    in
    {
        type            patch;
        nFaces          80;
        startFace       18460;
    }
```

- This dictionary is located in the **`constant/polyMesh`** directory.
- This file is automatically created when converting or generating the mesh.
- To get a visual reference of the patches, you can visualize the mesh with paraFoam/paraview.
- The type of the **out** patch is OK.
- The type of the **sym1** patch is OK.
- The type of the **sym2** patch is OK.
- The type of the **in** patch is OK.

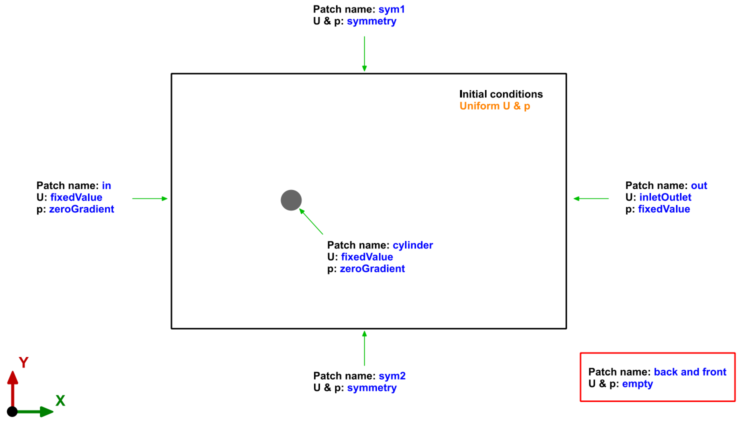


Flow past a cylinder – From laminar to turbulent flow

The *boundary* dictionary file

```
        cylinder
        {
            type            wall;
            inGroups        1(wall);
            nFaces          80;
            startFace       18540;
        }
        back
        {
            type            patch;
            nFaces          9200;
            startFace       18620;
        }
        front
        {
            type            patch;
            nFaces          9200;
            startFace       27820;
        }
    )
```

- The type of the **cylinder** patch is OK.
- The type of the **back** patch is **NOT OK**. Remember, this is a 2D simulation, therefore the type should be **empty**.
- The type of the **front** patch is **NOT OK**. Remember, this is a 2D simulation, therefore the type should be **empty**.
- Remember, we assign the **numerical type** boundary conditions (numerical values), in the field files found in the directory ***0***



Flow past a cylinder – From laminar to turbulent flow

- At this point, check that the **name** and **type** of the **base type** boundary conditions and **numerical type** boundary conditions are consistent. If everything is ok, we are ready to go.
- Do not forget to explore the rest of the dictionary files, namely:
  - *0/p* (**p** is defined as relative pressure)
  - *0/U*
  - constant/*transportProperties*
  - *system/controlDict*
  - *system/fvSchemes*
  - *system/fvSolution*
- Reminder:
  - The diameter of the cylinder is 2.0 m.
  - And we are targeting for a Re = 200.

$$\nu = \frac{\mu}{\rho} \qquad Re = \frac{\rho \times U \times D}{\mu} = \frac{U \times D}{\nu}$$

Flow past a cylinder – From laminar to turbulent flow

Running the case

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c2`**
- In the folder **`c1`** you will find the same setup, but to generate the mesh we use `blockMesh` (the mesh is identical).
- To run this case, in the terminal window type:

1.
```
$> renumberMesh -overwrite
```
2.
```
$> icoFoam | tee log.icofoam
```
3.
```
$> pyFoamPlotWatcher.py log.icofoam
```
   You will need to launch this script in a different terminal
4.
```
$> gnuplot scripts0/plot_coeffs
```
   You will need to launch this script in a different terminal
5.
```
$> paraFoam
```

Flow past a cylinder – From laminar to turbulent flow

Running the case

- In step 1 we use the utility `renumberMesh` to make the linear system more diagonal dominant, this will speed-up the linear solvers. This is inexpensive (even for large meshes), therefore is highly recommended to always do it.
- In step 2 we run the simulation and save the log file. Notice that we are sending the job to background.
- In step 3 we use `pyFoamPlotWatcher.py` to plot the residuals on-the-fly. As the job is running in background, we can launch this utility in the same terminal tab.
- In step 4 we use the gnuplot script `scripts0/plot_coeffs` to plot the force coefficients on-the-fly. Besides monitoring the residuals, is always a good idea to monitor a quantity of interest. Feel free to take a look at the script and to reuse it.
- The force coefficients are computed using **functionObjects**.
- After the simulation is over, we use `paraFoam` to visualize the results. Remember to use the VCR Controls to animate the solution.
- In the folder **`c1`** you will find the same setup, but to generate the mesh we use `blockMesh` (the mesh is identical).

Flow past a cylinder – From laminar to turbulent flow

- At this point try to use the following utilities. In the terminal type:

  -
    ```
    $> postProcess -func vorticity -noZero
    ```
    This utility will compute and write the vorticity field. The –noZero option means do not compute the vorticity field for the solution in the directory 0. If you do not add the -noZero option, it will compute and write the vorticity field for all the saved solutions, including 0

  -
    ```
    $> postprocess -func 'grad(U)' -latestTime
    ```
    This utility will compute and write the velocity gradient or grad(U) in the whole domain (including at the walls). The -latestTime option means compute the velocity gradient only for the last saved solution.

  -
    ```
    $> postprocess -func 'grad(p)'
    ```
    This utility will compute and write the pressure gradient or grad(U) in the whole domain (including at the walls).

  -
    ```
    $> foamToVTK -time 50:300
    ```
    This utility will convert the saved solution from OpenFOAM® format to VTK format. The -time 50:300 option means convert the solution to VTK format only for the time directories 50 to 300

These last three will give you and error message, try to fix it.

  -
    ```
    $> postProcess -func 'div(U)'
    ```
    This utility will compute and write the divergence of the velocity field or grad(U) in the whole domain (including at the walls).

  -
    ```
    $> pisoFoam -postProcess -func CourantNo
    ```
    This utility will compute and write the Courant number. This utility needs to access the solver database for the physical properties and additional quantities; therefore we need to tell what solver we are using. As the solver icoFoam does not accept the option -postProcess, we can use the solver pisoFoam instead. Remember, icoFoam is a fully laminar solver and pisoFoam is a laminar/turbulent solver.

  -
    ```
    $> pisoFoam -postProcess -func wallShearStress
    ```
    This utility will compute and write the wall shear stresses at the walls. As no arguments are given, it will save the wall shear stresses for all time steps.

Flow past a cylinder – From laminar to turbulent flow

Non-uniform field initialization

- In the previous case, it took about 150 seconds of simulation time to onset the instability.
- If you are not interested in the initial transient or if you want to speed-up the computation, you can add a perturbation in order to trigger the onset of the instability.
- Let us use the utility `setFields` to initialize a non-uniform flow.
- This case is already setup in the directory ,

  `$PTOFC/101OF/vortex_shedding/c3`

- As you saw in the previous example, `icoFoam` is a very basic solver that does not have access to all the advanced modeling or postprocessing capabilities that comes with OpenFOAM®.
- Therefore, instead of using `icoFoam` we will use `pisoFoam` (or `pimpleFoam`) from now on.
- To run the solver `pisoFoam` (or `pimpleFoam`) starting from the directory structure of an `icoFoam` case, you will need to add the followings modifications:
  - Add the file *`momentumTransport`* in the directory **`constant`**.
  - Add the **transportModel** to be used in the file *`constant/transportProperties`*.
  - Add the entry **div((nuEff*dev2(T(grad(U)))) Gauss linear;** to the dictionary *`system/fvSchemes`* in the section **divSchemes** (this entry is related to the Reynodls stresses).

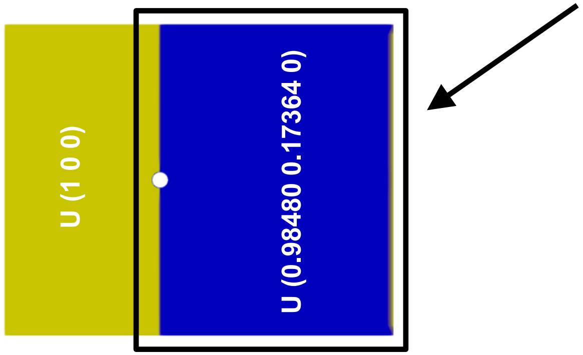
Flow past a cylinder – From laminar to turbulent flow

- Let us run the same case but using a non-uniform field

The `setFieldsDict` dictionary

```
17  defaultFieldValues
18  (
19      volVectorFieldValue U (1 0 0)
20  );
21
22  regions
23  (
24      boxToCell
25      {
26          box (0 -100 -100) (100 100 100);
27          fieldValues
28          (
29              volVectorFieldValue U (0.98480 0.17364 0)
30          );
31      }
32  );
```

- This dictionary file is located in the directory **system**.
- In lines 17-20 we set the default value of the velocity vector to be **(0 0 0)** in the whole domain.
- In lines 24-31, we initialize a rectangular region (**box**) just behind the cylinder with a velocity vector equal to **(0.98480 0.17364 0)**
- In this case, `setFields` will look for the dictionary file *U* and it will overwrite the original values according to the regions defined in `setFieldsDict`.



Flow past a cylinder – From laminar to turbulent flow

- Let us run the same case but using a non-uniform field.
- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c3`**
- Feel free to use the Fluent mesh or the mesh generated with `blockMesh`. Hereafter, we will use `blockMesh`.
- To run this case, in the terminal window type:

1.
```
$> foamCleanTutorials
```
2.
```
$> blockMesh
```
3.
```
$> rm -rf 0 > /dev/null 2>&1
```
4.
```
$> cp -r 0_org/ 0
```
5.
```
$> setFields
```
6.
```
$> renumberMesh -overwrite
```
7.
```
$> pisoFoam | tee log.solver
```
8.
```
$> pyFoamPlotWatcher.py log.pisofoam
```
   You will need to launch this script in a different terminal
9.
```
$> gnuplot scripts0/plot_coeffs
```
   You will need to launch this script in a different terminal
10.
```
$> paraFoam
```

Flow past a cylinder – From laminar to turbulent flow

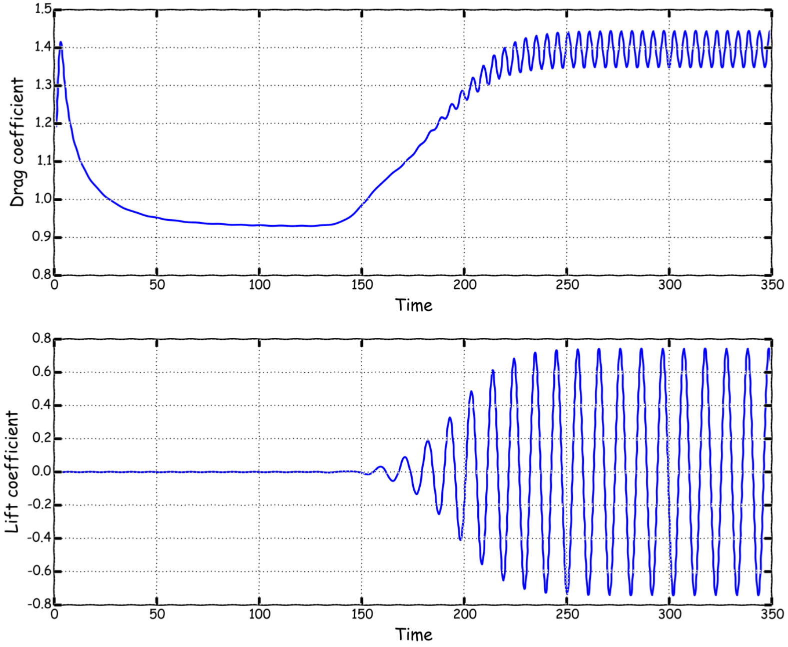
Running the case – Non-uniform field initialization

- In step 2 we generate the mesh using `blockMesh`. The **name** and **type** of the patches are already set in the dictionary `blockMeshDict` so there is no need to modify the `boundary` file.
- In step 4 we copy the original files to the directory **0**. We do this to keep a backup of the original files as the file *0/U* will be overwritten when using `setFields`.
- In step 5 we initialize the solution using `setFields`.
- In step 6 we use the utility `renumberMesh` to make the linear system more diagonal dominant, this will speed-up the linear solvers.
- In step 7 we run the simulation and save the log file. Notice that we are sending the job to background.
- In step 8 we use `pyFoamPlotWatcher.py` to plot the residuals on-the-fly. As the job is running in background, we can launch this utility in the same terminal tab.
- In step 9 we use the gnuplot script `scripts0/plot_coeffs` to plot the lift and drag coefficients on-the-fly. Besides monitoring the residuals, is always a good idea to monitor a quantity of interest. Feel free to take a look at the script and to reuse it.

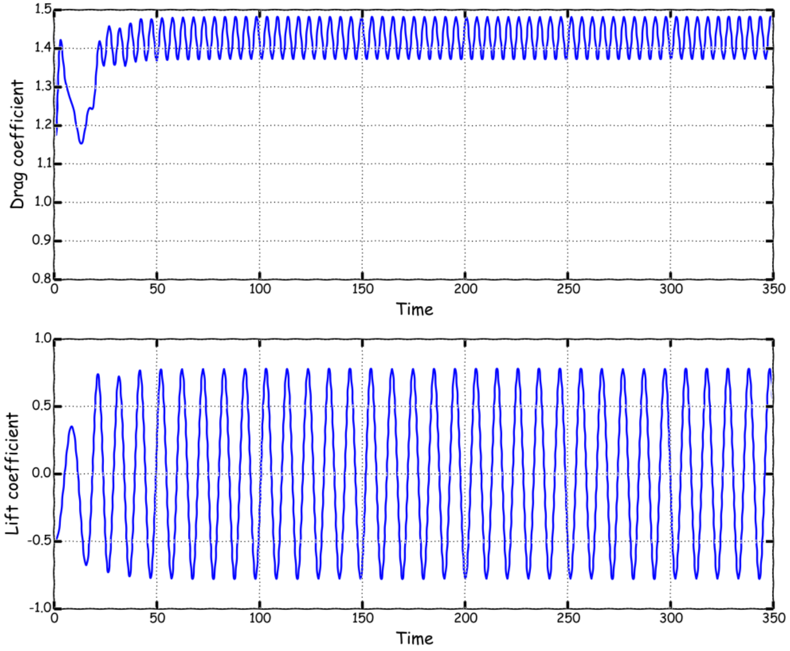
Flow past a cylinder – From laminar to turbulent flow

Does non-uniform field initialization make a difference?

- A picture is worth a thousand words. No need to tell you yes, even if the solutions are slightly different.
- This bring us to the next subject, for how long should we run the simulation?



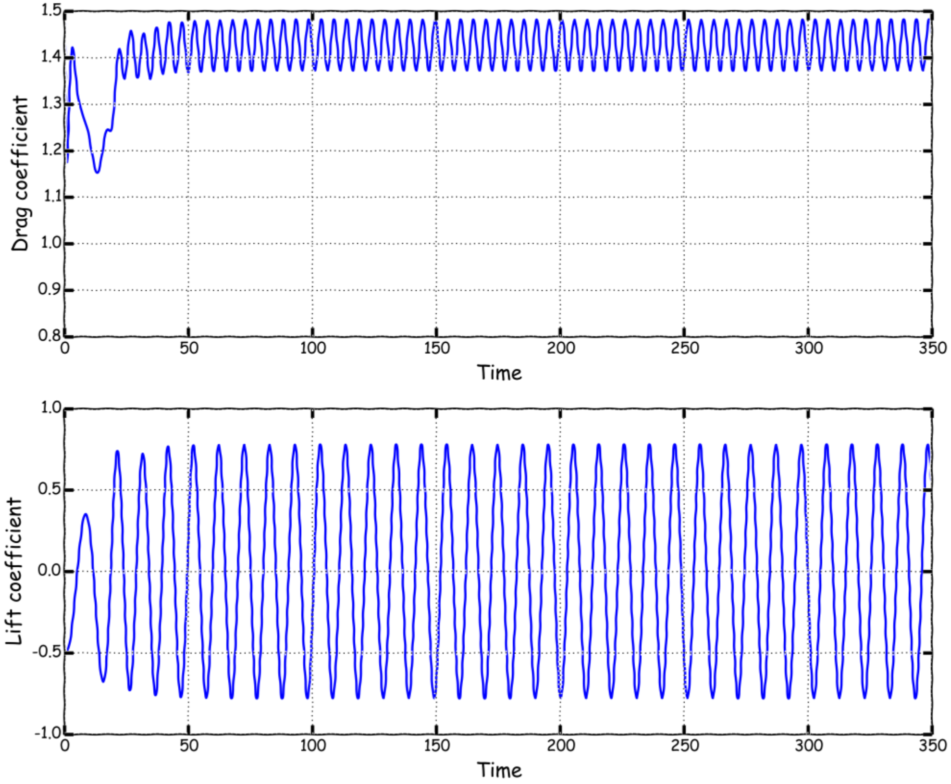
No field initialization



With field initialization

Flow past a cylinder – From laminar to turbulent flow

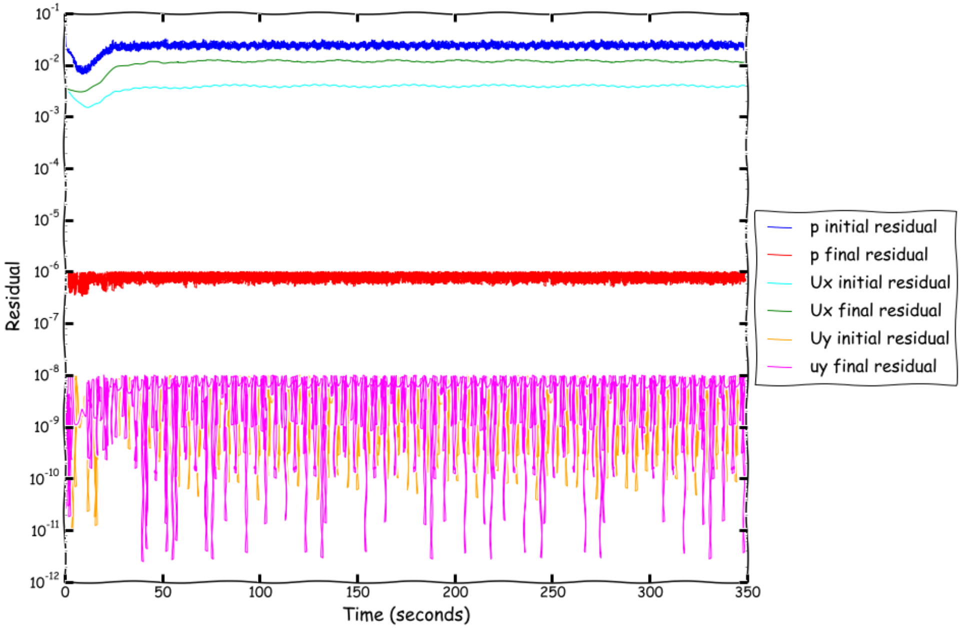
For how long should run the simulation?



- This is the difficult part when dealing with unsteady flows.
- Usually you run the simulation until the behavior of a quantity of interest does not oscillates or it becomes periodic.
- In this case we can say that after the 50 seconds mark the solution becomes periodic, therefore there is no need to run up to 350 seconds (unless you want to gather a lot of statistics).
- We can stop the simulation at 150 seconds (or maybe less), and do the average of the quantities between 100 and 150 seconds.

Flow past a cylinder – From laminar to turbulent flow

What about the residuals?



- Residuals are telling you a lot, but they are difficult to interpret.
- In this case the fact that the initial residuals are increasing after about 10 seconds, does not mean that the solution is diverging. This is in indication that something is happening (in this case the onset of the instability).
- Remember, the residuals should always drop to the tolerance criteria set in the *fvSolution* dictionary (final residuals). If they do not drop to the desired tolerance, we are talking about unconverged time-steps.
- Things that are not clear from the residuals:
  - For how long should we run the simulation?
  - Is the solution converging to the right value?

Flow past a cylinder – From laminar to turbulent flow

How to compute force coefficients

```
51      functions
52      {

178         forceCoeffs_object
179         {
188             type forceCoeffs;
189             functionObjectLibs ("libforces.so");
191             patches (cylinder);

193             pName p;
194             Uname U;
195             rhoName rhoInf;
196             rhoInf 1.0;

198             //// Dump to file
199             log true;

201             CofR (0.0 0 0);
202             liftDir (0 1 0);
202             dragDir (1 0 0);
204             pitchAxis (0 0 1);
205             magUInf 1.0;
206             lRef 1.0;
207             Aref 2.0;

209             outputControl   timeStep;
210             outputInterval  1;
211         }

237     };
```

- To compute the force coefficients we use **functionObjects**.
- Remember, **functionObjects** are defined at the end of the *controlDict* dictionary file.
- In line 178 we give a name to the **functionObject**.
- In line 191 we define the patch where we want to compute the forces.
- In lines 195-196 we define the reference density value.
- In line 201 we define the center of rotation (for moments).
- In line 202 we define the lift force axis.
- In line 203 we define the drag force axis.
- In line 204 we define the axis of rotation for moment computation.
- In line 206 we give the reference length (for computing the moments)
- In line 207 we give the reference area (in this case the frontal area).
- The output of this **functionObject** is saved in the file *forceCoeffs.dat* located in the directory **forceCoeffs_object/0/**

Flow past a cylinder – From laminar to turbulent flow

Can we compute basic statistics of the force coefficients using gnuplot?

- Yes we can. Enter the gnuplot prompt and type:

1.
```
gnuplot> stats ‘postProcessing/forceCoeffs_object/0/forceCoeffs.dat’ u 3
```
This will compute the basic statistics of all the rows in the file forceCoeffs.dat (we are sampling column 3 in the input file)

2.
```
gnuplot> stats ‘postProcessing/forceCoeffs_object/0/forceCoeffs.dat’ every ::3000::7000 u 3
```
This will compute the basic statistics of rows 3000 to 7000 in the file forceCoeffs.dat (we are sampling column 3 in the input file)

3.
```
gnuplot> plot ‘postProcessing/forceCoeffs_object/0/forceCoeffs.dat’ u 3 w l
```
This will plot column 3 against the row number (iteration number)

4.
```
gnuplot> exit
```
To exit gnuplot

- Remember the force coefficients information is saved in the file `forceCoeffs.dat` located in the directory **`postProcessing/forceCoeffs_object/0`**

Flow past a cylinder – From laminar to turbulent flow

On the solution accuracy

```
ddtSchemes
{
  default         backward;
}

gradSchemes
{
    default           cellLimited leastSquares 1;
}

divSchemes
{
    default         none;
    div(phi,U)      Gauss linearUpwindV default;
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear limited 1;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         limited 1;
}
```

- At the end of the day we want a solution that is second order accurate.
- We define the discretization schemes (and therefore the accuracy) in the dictionary *fvSchemes*.
- In this case, for time discretization (**ddtSchemes**) we are using the **backward** method.
- For gradient discretization (**gradSchemes**) we are using the **leastSquares** method with slope limiters (**cellLimited**) for all terms (**default** option).
- Sometimes adding a gradient limiter to the pressure gradient or **grad(p)** can be too diffusive, so it is better not to use gradient limiters for **grad(p)**, e.g., **grad(p) leastSquares**.
- For the discretization of the convective terms (**divSchemes**) we are using **linearUpwindV** interpolation method for the term **div(rho,U)**.
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear limited 1** method
- In overall, this method is second order accurate (this is what we want).

Flow past a cylinder – From laminar to turbulent flow

On the solution tolerance and linear solvers

```
solvers
{
    p
    {
        solver           GAMG;
        tolerance        1e-6;
        relTol           0;
        smoother         GaussSeidel;
        nPreSweeps       0;
        nPostSweeps      2;
        cacheAgglomeration on;
        agglomerator     faceAreaPair;
        nCellsInCoarsestLevel 100;
        mergeLevels      1;
    }

    pFinal
    {
        $p;
        relTol          0;
    }

    U
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 2;
}
```

- We define the solution tolerance and linear solvers in the dictionary *fvSolution*.
- To solve the pressure (**p**) we are using the **GAMG** method with an absolute **tolerance** of 1e-6 and a relative tolerance **relTol** of 0.01.
- The entry **pFinal** refers to the final correction of the **PISO** loop. It is possible to use a tighter convergence criteria only in the last iteration.
- To solve **U** we are using the solver **PBiCGStab** and the **DILU** preconditioner**,** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0 (the solver will stop iterating when it meets any of the conditions).
- Solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.
- The **PISO** sub-dictionary contains entries related to the pressure-velocity coupling (in this case the **PISO** method). Hereafter we are doing two **PISO** correctors (**nCorrectors**) and two non-orthogonal corrections (**nNonOrthogonalCorrectors**).

Flow past a cylinder – From laminar to turbulent flow

On the runtime parameters

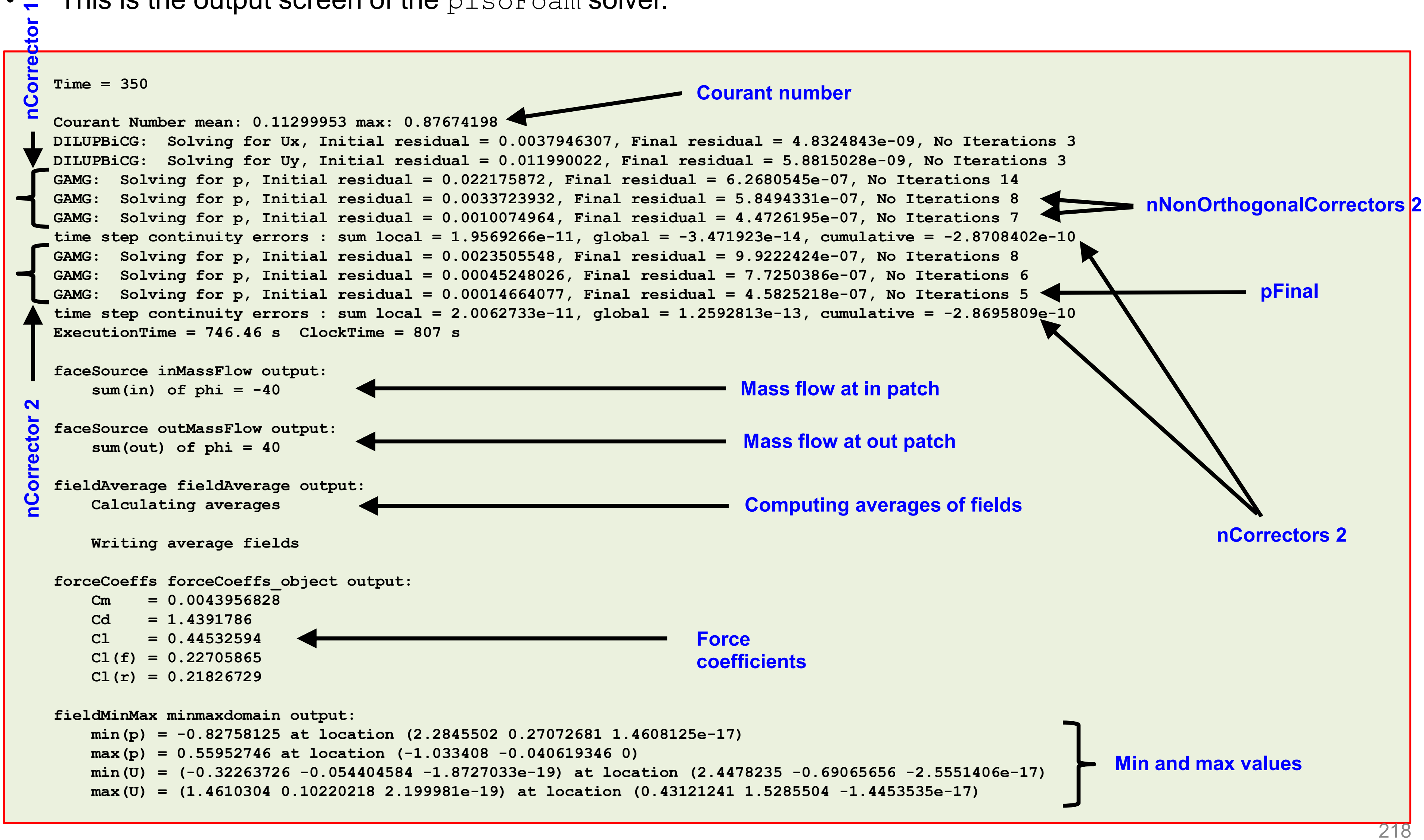
```
application     pisoFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         350;

deltaT          0.05;

writeControl    runTime;

writeInterval   1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  8;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

- This case starts from the latest saved solution (**startFrom**).
- In this case as there are no saved solutions, it will start from 0 (**startTime**).
- It will run up to 350 seconds (**endTime**).
- The time step of the simulation is 0.05 seconds (**deltaT**). The time step has been chosen in such a way that the Courant number is less than 1
- It will write the solution every 1 second (**writeInterval**) of simulation time (**runTime**).
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- The write precision is 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.

Flow past a cylinder – From laminar to turbulent flow

The output screen

- This is the output screen of the `pisoFoam` solver.



```
Time = 350

Courant Number mean: 0.11299953 max: 0.87674198
DILUPBiCG:  Solving for Ux, Initial residual = 0.0037946307, Final residual = 4.8324843e-09, No Iterations 3
DILUPBiCG:  Solving for Uy, Initial residual = 0.011990022, Final residual = 5.8815028e-09, No Iterations 3
GAMG:  Solving for p, Initial residual = 0.022175872, Final residual = 6.2680545e-07, No Iterations 14
GAMG:  Solving for p, Initial residual = 0.0033723932, Final residual = 5.8494331e-07, No Iterations 8
GAMG:  Solving for p, Initial residual = 0.0010074964, Final residual = 4.4726195e-07, No Iterations 7
time step continuity errors : sum local = 1.9569266e-11, global = -3.471923e-14, cumulative = -2.8708402e-10
GAMG:  Solving for p, Initial residual = 0.0023505548, Final residual = 9.9222424e-07, No Iterations 8
GAMG:  Solving for p, Initial residual = 0.00045248026, Final residual = 7.7250386e-07, No Iterations 6
GAMG:  Solving for p, Initial residual = 0.00014664077, Final residual = 4.5825218e-07, No Iterations 5
time step continuity errors : sum local = 2.0062733e-11, global = 1.2592813e-13, cumulative = -2.8695809e-10
ExecutionTime = 746.46 s  ClockTime = 807 s

faceSource inMassFlow output:
    sum(in) of phi = -40

faceSource outMassFlow output:
    sum(out) of phi = 40

fieldAverage fieldAverage output:
    Calculating averages

    Writing average fields

forceCoeffs forceCoeffs_object output:
    Cm    = 0.0043956828
    Cd    = 1.4391786
    Cl    = 0.44532594
    Cl(f) = 0.22705865
    Cl(r) = 0.21826729

fieldMinMax minmaxdomain output:
    min(p) = -0.82758125 at location (2.2845502 0.27072681 1.4608125e-17)
    max(p) = 0.55952746 at location (-1.033408 -0.040619346 0)
    min(U) = (-0.32263726 -0.054404584 -1.8727033e-19) at location (2.4478235 -0.69065656 -2.5551406e-17)
    max(U) = (1.4610304 0.10220218 2.199981e-19) at location (0.43121241 1.5285504 -1.4453535e-17)
```

nCorrector 1

nCorrector 2

Courant number

nNonOrthogonalCorrectors 2

pFinal

Mass flow at in patch

Mass flow at out patch

Computing averages of fields

nCorrectors 2

Force coefficients

Min and max values

Flow past a cylinder – From laminar to turbulent flow

Let us use a potential solver to find a quick solution

- In this case we are going to use the potential solver `potentialFoam` (remember potential solvers are inviscid, irrotational and incompressible)
- This solver is super fast, and it can be used to find a solution to be used as initial conditions (non-uniform field) for an incompressible solver.
- A good initial condition will accelerate and improve the convergence rate.
- This case is already setup in the directory

```
$PTOFC/101OF/vortex_shedding/c4
```

- Do not forget to explore the dictionary files.
- The following dictionaries are different
  - *system/fvSchemes*
  - *system/fvSolution*

Try to spot the differences.

Flow past a cylinder – From laminar to turbulent flow

Running the case – Let us use a potential solver to find a quick solution

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c4`**
- Feel free to use the Fluent mesh or the mesh generated with `blockMesh`. In this case we will use `blockMesh`.
- To run this case, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> rm -rf 0
4.  $> cp -r 0_org 0
5.  $> potentialFoam -noFunctionObjects -initialiseUBCs -writep -writePhi
6.  $> paraFoam
```

Flow past a cylinder – From laminar to turbulent flow

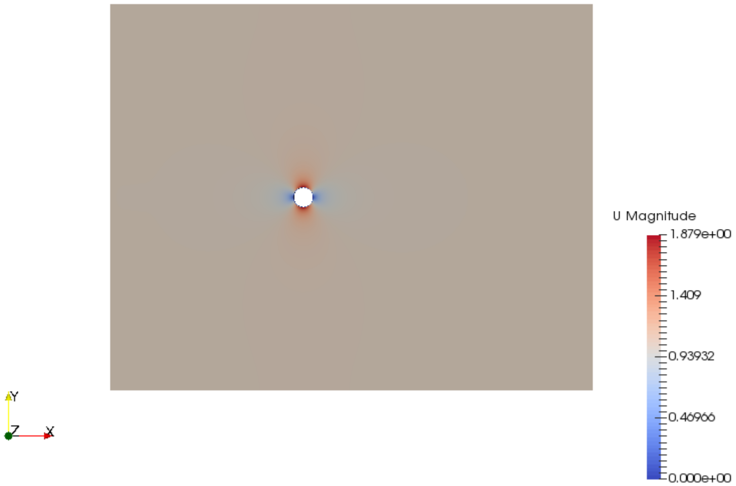
Running the case – Let us use a potential solver to find a quick solution

- In step 2 we generate the mesh using `blockMesh`. The **name** and **type** of the patches are already set in the dictionary `blockMeshDict` so there is no need to modify the *`boundary`* file.
- In step 4 we copy the original files to the directory **`0`**. We do this to keep a backup of the original files as they will be overwritten by the solver `potentialFoam`.
- In step 5 we run the solver. We use the option `–noFunctionObjects` to avoid conflicts with the **functionobjects**. The options `–writep` and `–writePhi` will write the pressure field and fluxes respectively.
- At this point, if you want to use this solution as initial conditions for an incompressible solver, just copy the files *`U`* and *`p`* into the start directory of the incompressible case you are looking to run. Have in mind that the meshes need to be the same.
- Be careful with the **name** and **type** of the boundary conditions, they should be same between the potential case and incompressible case.

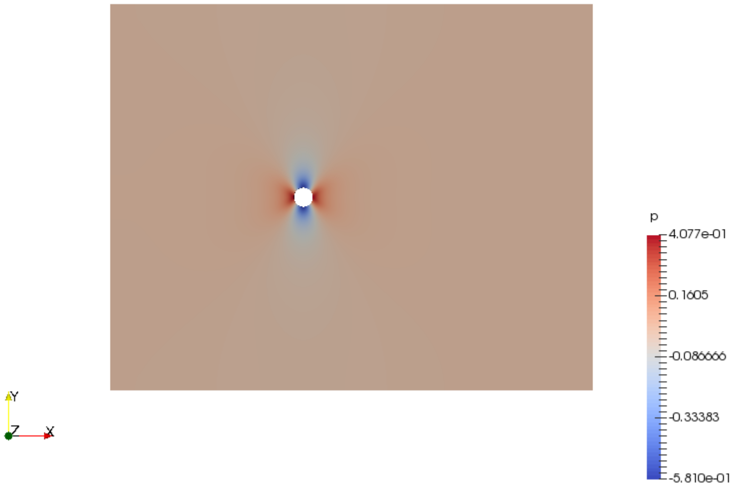
Flow past a cylinder – From laminar to turbulent flow

Potential solution

- Using a potential solution as initial conditions is much better than using a uniform flow. It will speed up the solution and it will give you more stability.
- Finding a solution using the potential solver is inexpensive.



Velocity field

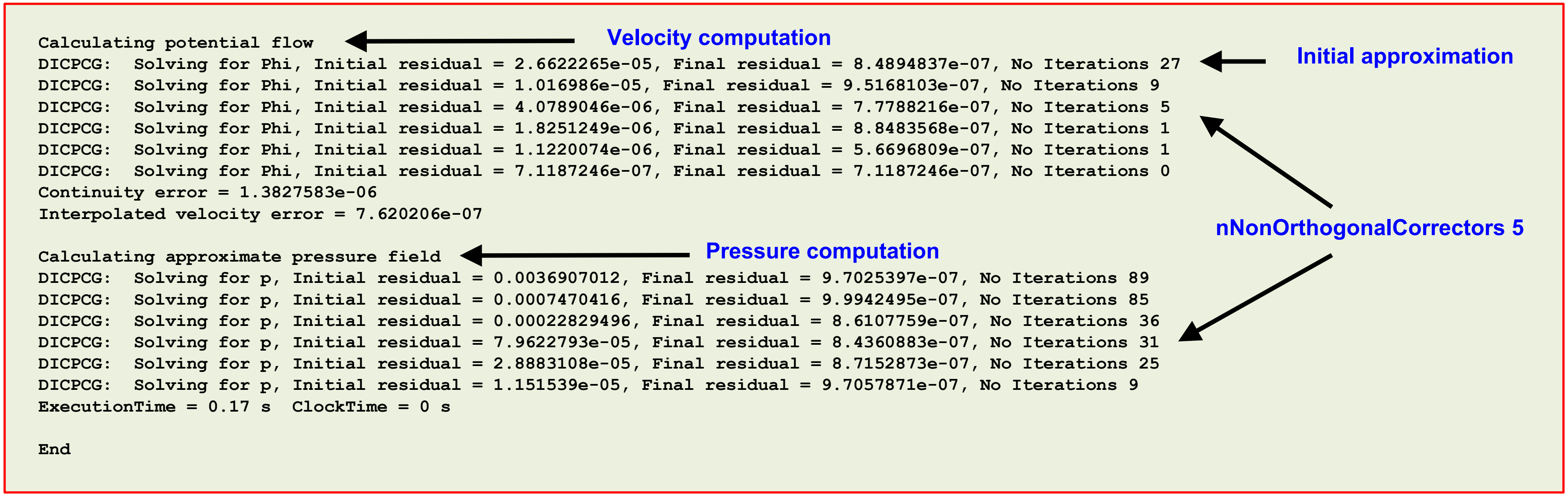


Pressure field

Flow past a cylinder – From laminar to turbulent flow

The output screen

- This is the output screen of the `potentialFoam` solver.
- The output of this solver is also a good indication of the sensitivity of the mesh quality to gradients computation. If you see that the number of iterations are dropping iteration after iteration, it means that the mesh is fine.
- If the number of iterations remain stalled, it means that the mesh is sensitive to gradients, so you should use non-orthogonal correction.
- In this case we have a good mesh.



```
Calculating potential flow
DICPCG:  Solving for Phi, Initial residual = 2.6622265e-05, Final residual = 8.4894837e-07, No Iterations 27
DICPCG:  Solving for Phi, Initial residual = 1.016986e-05, Final residual = 9.5168103e-07, No Iterations 9
DICPCG:  Solving for Phi, Initial residual = 4.0789046e-06, Final residual = 7.7788216e-07, No Iterations 5
DICPCG:  Solving for Phi, Initial residual = 1.8251249e-06, Final residual = 8.8483568e-07, No Iterations 1
DICPCG:  Solving for Phi, Initial residual = 1.1220074e-06, Final residual = 5.6696809e-07, No Iterations 1
DICPCG:  Solving for Phi, Initial residual = 7.1187246e-07, Final residual = 7.1187246e-07, No Iterations 0
Continuity error = 1.3827583e-06
Interpolated velocity error = 7.620206e-07

Calculating approximate pressure field
DICPCG:  Solving for p, Initial residual = 0.0036907012, Final residual = 9.7025397e-07, No Iterations 89
DICPCG:  Solving for p, Initial residual = 0.0007470416, Final residual = 9.9942495e-07, No Iterations 85
DICPCG:  Solving for p, Initial residual = 0.00022829496, Final residual = 8.6107759e-07, No Iterations 36
DICPCG:  Solving for p, Initial residual = 7.9622793e-05, Final residual = 8.4360883e-07, No Iterations 31
DICPCG:  Solving for p, Initial residual = 2.8883108e-05, Final residual = 8.7152873e-07, No Iterations 25
DICPCG:  Solving for p, Initial residual = 1.151539e-05, Final residual = 9.7057871e-07, No Iterations 9
ExecutionTime = 0.17 s  ClockTime = 0 s

End
```

Velocity computation

Initial approximation

nNonOrthogonalCorrectors 5

Pressure computation

Flow past a cylinder – From laminar to turbulent flow

Let us map a solution from a coarse mesh to a finer mesh

- It is also possible to map the solution from a coarse mesh to a finer mesh (and all the way around).
- For instance, you can compute a full Navier-Stokes solution in a coarse mesh (fast solution), and then map it to a finer mesh.
- Let us map the solution from the potential solver to a finer mesh (if you want you can map the solution obtained using `pisoFoam` or `icoFoam`). To do this we will use the utility `mapFields`.
- This case is already setup in the directory

```
$PTOFC/101OF/vortex_shedding/c6
```

Flow past a cylinder – From laminar to turbulent flow

Running the case – Let us map a solution from a coarse mesh to a finer mesh

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c6`**
- To generate the mesh, use `blockMesh` (remember this mesh is finer).
- To run this case, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> rm -rf 0
4.  $> cp -r 0_org 0
5.  $> mapfields ../c4 -consistent -noFunctionObjects -mapMethod cellPointInterpolate -sourceTime 0
6.  $> paraFoam
```

- To run step 5 you need to have a solution in the directory **`../c4`** ⚠

Flow past a cylinder – From laminar to turbulent flow

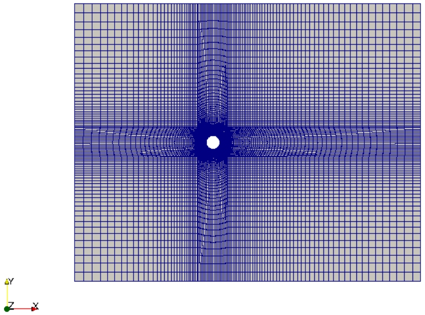
Running the case – Let us map a solution from a coarse mesh to a finer mesh

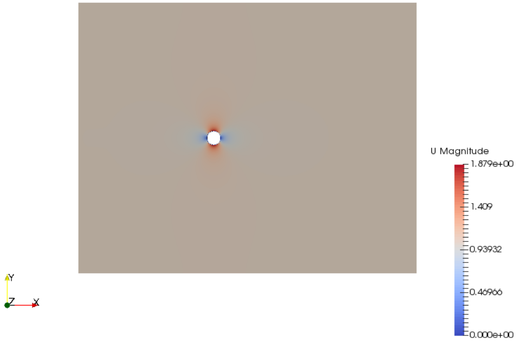
- In step 2 we generate a finer mesh using `blockMesh`. The **name** and **type** of the patches are already set in the dictionary `blockMeshDict` so there is no need to modify the `boundary` file.
- In step 4 we copy the original files to the directory `0`. We do this to keep a backup of the original files as they will be overwritten by the utility `mapFields`.
- In step 5 we use the utility `mapFields` with the following options:
  - We copy the solution from the directory `../c4`
  - The options `-consistent` is used when the domains and BCs are the same.
  - The option `-noFunctionObjects` is used to avoid conflicts with the **functionObjects**.
  - The option `-mapMethod cellPointInterpolate` defines the interpolation method.
  - The option `-sourceTime 0` defines the time from which we want to interpolate the solution.

Flow past a cylinder – From laminar to turbulent flow

The meshes and the mapped fields

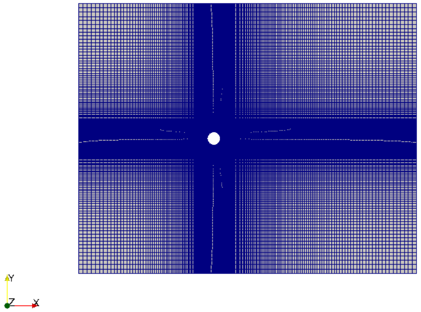
Coarse mesh

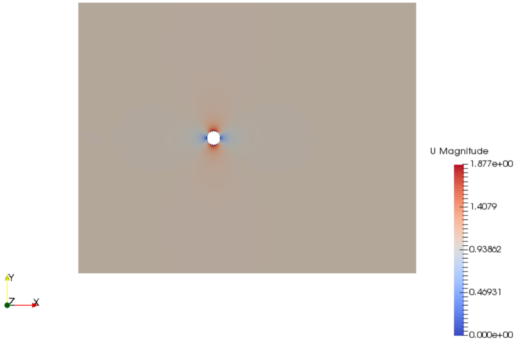




mapFields

Fine mesh

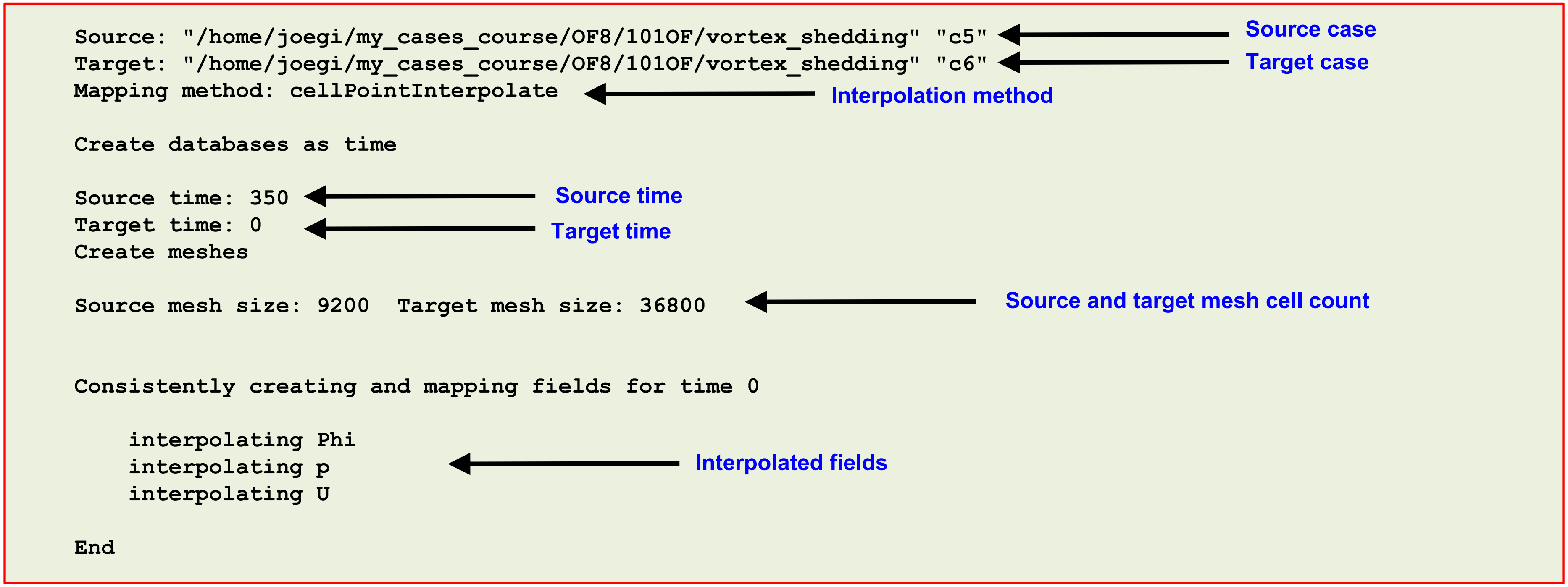




Flow past a cylinder – From laminar to turbulent flow

The output screen

- This is the output screen of the `mapFields` utility.
- The utility `mapFields`, will try to interpolate all fields in the source directory.
- You can control the target time via the **startFrom** and **startTime** keywords in the *`controlDict`* dictionary file.



```
Source: "/home/joegi/my_cases_course/OF8/101OF/vortex_shedding" "c5"
Target: "/home/joegi/my_cases_course/OF8/101OF/vortex_shedding" "c6"
Mapping method: cellPointInterpolate

Create databases as time

Source time: 350
Target time: 0
Create meshes

Source mesh size: 9200  Target mesh size: 36800


Consistently creating and mapping fields for time 0

    interpolating Phi
    interpolating p
    interpolating U

End
```

- Finally, after mapping the solution, you can run the solver in the usual way. The solver will use the mapped solution as initial conditions.

Flow past a cylinder – From laminar to turbulent flow

Setting a turbulent case

- So far we have used laminar incompressible solvers.
- Let us do a turbulent simulation.
- When doing turbulent simulations, we need to choose the turbulence model, define the boundary and initial conditions for the turbulent quantities, and modify the `fvSchemes` and `fvSolution` dictionaries to take account for the new variables we are solving (the transported turbulent quantities).
- This case is already setup in the directory

```
$PTOFC/101OF/vortex_shedding/c14
```

Flow past a cylinder – From laminar to turbulent flow

- The following dictionaries remain unchanged
  - *system/blockMeshDict*
  - *constant/polyMesh/boundary*
  - *0/p*
  - *0/U*

- The following dictionaries need to be adapted for the turbulence case
  - *constant/transportProperties*
  - *system/controlDict*
  - *system/fvSchemes*
  - *system/fvSolution*

- The following dictionaries need to be adapted for the turbulence case
  - *constant/momentumTransport*

Flow past a cylinder – From laminar to turbulent flow

The *transportProperties* dictionary file

- This dictionary file is located in the directory **constant**.
- In this file we set the transport model and the kinematic viscosity (**nu**).

```
transportModel  Newtonian;

nu              nu [ 0 2 -1 0 0 0 0 ] 0.0002;
```

- Reminder:
  - The diameter of the cylinder is 2.0 m.
  - And we are targeting for a Re = 10000.

$$\nu = \frac{\mu}{\rho} \qquad Re = \frac{\rho \times U \times D}{\mu} = \frac{U \times D}{\nu}$$

Flow past a cylinder – From laminar to turbulent flow

The *momentumTransport* dictionary file

- This dictionary file is located in the directory **constant**.
- In this dictionary file we select what model we would like to use (laminar or turbulent).
- In this case we are interested in modeling turbulence, therefore the dictionary is as follows

```
simulationType  RAS;

RAS
{
    RASModel        kOmegaSST;

    turbulence      on;

    printCoeffs     on;
}
```

RANS type simulation

RANS sub-dictionary

RANS model to use

Turn on/off turbulence. Runtime modifiable

Print coefficients at the beginning

- If you want to know the models available use the banana method.

Flow past a cylinder – From laminar to turbulent flow

The `controlDict` dictionary

```
17      application      pimpleFoam;
18
20      startFrom        latestTime;
21
22      startTime        0;
23
24      stopAt           endTime;
25
26      endTime          500;
27
28      deltaT           0.001;
29
30      writeControl     runTime;
31
32      writeInterval    1;
33
34      purgeWrite       0;
35
36      writeFormat      ascii;
37
38      writePrecision   8;
39
40      writeCompression off;
41
42      timeFormat       general;
43
44      timePrecision    6;
45
46      runTimeModifiable yes;
47
48      adjustTimeStep   yes;
49
50      maxCo            0.9;
51      maxDeltaT        0.1;
```

- This case will start from the last saved solution (**startFrom**). If there is no solution, the case will start from time 0 (**startTime**).
- It will run up to 500 seconds (**endTime**).
- The initial time step of the simulation is 0.001 seconds (**deltaT**).
- It will write the solution every 1 second (**writeInterval**) of simulation time (**runTime**).
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- The write precision is 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.
- In line 48 we turn on the option **adjustTimeStep**. This option will automatically adjust the time step to achieve the maximum desired courant number **maxCo** (line 50).
- We also set a maximum time step **maxDeltaT** in line 51.
- Remember, the first time-step of the simulation is done using the value set in line 28 and then it is automatically scaled to achieve the desired maximum values (lines 50-51).
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.

Flow past a cylinder – From laminar to turbulent flow

The *fvSchemes* dictionary

```
ddtSchemes
{
    default         CrankNicolson 0.7;
}
gradSchemes
{
    default           cellLimited leastSquares 1;
    grad(U)           cellLimited Gauss linear 1;
}
divSchemes
{
    default         none;
    div(phi,U)      Gauss linearUpwindV grad(U);
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
    div(phi,k)          Gauss linearUpwind default;
    div(phi,omega)      Gauss linearUpwind default;
}
laplacianSchemes
{
     default         Gauss linear limited 1;
}
interpolationSchemes
{
    default         linear;
}
snGradSchemes
{
    default         limited 1;
}
wallDist
{
    method meshWave;
}
```

- In this case, for time discretization (**ddtSchemes**) we are using the blended **CrankNicolson** method. The blending coefficient goes from 0 to 1, where 0 is equivalent to the **Euler** method and 1 is a pure **Crank Nicolson**.
- For gradient discretization (**gradSchemes**) we are using as default option the **leastSquares** method. For **grad(U)** we are using **Gauss linear** with slope limiters (**cellLimited**). You can define different methods for every term in the governing equations, for example, you can define a different method for **grad(p)**.
- For the discretization of the convective terms (**divSchemes**) we are using **linearUpwindV** interpolation method with slope limiters for the term **div(phi,U)**.
- For the terms **div(phi,k)** and **div(phi,omega)** we are using **linearUpwind** interpolation method with no slope limiters. These terms are related to the turbulence modeling.
- For the term **div((nuEff*dev2(T(grad(U)))))** we are using **linear** interpolation (this term is related to turbulence modeling).
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear limited 1** method.
- To compute the distance to the wall and normals to the wall, we use the method **meshWave**. This only applies when using wall functions (turbulence modeling).
- This method is second order accurate.

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
solvers
{
    p
    {
        solver           GAMG;
        tolerance        1e-6;
        relTol           0.001;
        smoother         GaussSeidel;
        nPreSweeps       0;
        nPostSweeps      2;
        cacheAgglomeration on;
        agglomerator     faceAreaPair;
        nCellsInCoarsestLevel 100;
        mergeLevels      1;
        minIter         2;
    }

    pFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
        minIter        3;
    }

    U
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
        minIter        3;
    }
```

- To solve the pressure (**p**) we are using the **GAMG** method, with an absolute **tolerance** of 1e-6 and a relative tolerance **relTol** of 0.001. Notice that we are fixing the number of minimum iterations (**minIter**).
- To solve the final pressure correction (**pFinal**) we are using the **PCG** method with the **DIC** preconditioner, with an absolute **tolerance** of 1e-6 and a relative tolerance **relTol** of 0.
- Notice that we can use different methods between **p** and **pFinal**. In this case we are using a tighter tolerance for the last iteration.
- We are also fixing the number of minimum iterations (**minIter**). This entry is optional.
- To solve **U** we are using the solver **PBiCGStab** with the **DILU** preconditioner, an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0. Notice that we are fixing the number of minimum iterations (**minIter**).

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
    solvers
    {

        UFinal
        {
            solver          PBiCGStab;
            preconditioner  DILU;
            tolerance       1e-08;
            relTol          0;
            minIter          3;
        }

        omega
        {
            solver          PBiCGStab;
            preconditioner  DILU;
            tolerance       1e-08;
            relTol          0;
            minIter          3;
        }

        omegaFinal
        {
            solver          PBiCGStab;
            preconditioner  DILU;
            tolerance       1e-08;
            relTol          0;
            minIter          3;
        }

        k
        {
            solver          PBiCGStab;
            preconditioner  DILU;
            tolerance       1e-08;
            relTol          0;
            minIter          3;
        }
```

- To solve **UFinal** we are using the solver **PBiCGStab** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0. Notice that we are fixing the number of minimum iterations (**minIter**).
- To solve **omega** and **omegaFinal** we are using the solver **PBiCGStab** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0. Notice that we are fixing the number of minimum iterations (**minIter**).
- To solve **k** we are using the solver **PBiCGStab** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0. Notice that we are fixing the number of minimum iterations (**minIter**).

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
113         kFinal
114         {
115             solver          PBiCGStab;
116             preconditioner  DILU;
117             tolerance       1e-08;
118             relTol          0;
119             minIter          3;
120         }
121     }
122
123     PIMPLE
124     {
126             nOuterCorrectors 1;
127             //nOuterCorrectors 2;
128
129             nCorrectors 3;
130             nNonOrthogonalCorrectors 1;
133     }
134
135     relaxationFactors
136     {
137         fields
138         {
139             p               0.3;
140         }
141         equations
142         {
143             U               0.7;
144             k               0.7;
145             omega           0.7;
146         }
147     }
```

- To solve **kFinal** we are using the solver **PBiCGStab** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0. Notice that we are fixing the number of minimum iterations (**minIter**).
- In lines 123-133 we setup the entries related to the pressure-velocity coupling method used (**PIMPLE** in this case). Setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- To gain more stability we are using 1 outer correctors (**nOuterCorrectors**), 3 inner correctors or **PISO** correctors (**nCorrectors**), and 1 correction due to non-orthogonality (**nNonOrthogonalCorrectors**).
- Remember, adding corrections increase the computational cost.
- In lines 135-147 we setup the under-relaxation factors used during the outer corrections of the **PIMPLE** method.
    - The values defined correspond to the industry standard of the **SIMPLE** method.
    - By using under-relaxation we ensure diagonal equality.
    - Be careful not use too low values as you will loose time accuracy.
    - If you want to disable under-relaxation, comment out these lines.

Flow past a cylinder – From laminar to turbulent flow

- The following dictionaries are new
  - *0/k*
  - *0/omega*
  - *0/nut*

  These are the field variables related to the closure equations of the turbulent model.

- As we are going to use the $\kappa - \omega \; SST$ model we need to define the initial conditions and boundaries conditions.
- To define the IC/BC we will use the free stream values of κ and ω
- In the following site, you can find a lot information about choosing initial and boundary conditions for the different turbulence models:
  - https://turbmodels.larc.nasa.gov/

Flow past a cylinder – From laminar to turbulent flow

$\kappa - \omega \; SST$ Turbulence model free-stream boundary conditions

- The initial value for the turbulent kinetic energy κ can be found as follows

$$\kappa = \frac{3}{2}(UI)^2$$

- The initial value for the specific kinetic energy ω can be found as follows

$$\omega = \frac{\rho\kappa}{\mu}\frac{\mu_t}{\mu}^{-1}$$

- Where $\frac{\mu_t}{\mu}$ is the viscosity ratio and $I = \frac{u'}{\bar{u}}$ is the turbulence intensity.

- If you are working with external aerodynamics or virtual wind tunnels, you can use the following reference values for the turbulence intensity and the viscosity ratio. They work most of the times, but it is a good idea to have some experimental data or a better initial estimate.

	Low	Medium	High
I	1.0 %	5.0 %	10.0 %
μ_t/μ	1	10	100

Flow past a cylinder – From laminar to turbulent flow

The file *0/k*

```
19    internalField   uniform 0.00015;
20
21    boundaryField
22    {
23        out
24        {
25            type            inletOutlet;
26            inletValue      uniform 0.00015;
27            value           uniform 0.00015;
28        }
29        sym1
30        {
31            type            symmetryPlane;
32        }
33        sym2
34        {
35            type            symmetryPlane;
36        }
37        in
38        {
39            type            fixedValue;
40            value           uniform 0.00015;
41        }
42        cylinder
43        {
44            type            kqRWallFunction;
45            value           uniform 0.00015;
46        }
47        back
48        {
49            type            empty;
50        }
51        front
52        {
53            type            empty;
54        }
55    }
```

- We are using uniform initial conditions (line 19).
- For the **in** patch we are using a **fixedValue** boundary condition.
- For the **out** patch we are using an **inletOutlet** boundary condition (this boundary condition avoids backflow).
- For the **cylinder** patch (which is **base type wall**), we are using the **kqRWallFunction** boundary condition. This is a wall function, we are going to talk about this when we deal with turbulence modeling. Remember, we can use wall functions only if the patch is of **base type wall**.
- The rest of the patches are constrained.
- FYI, the inlet velocity is 1 and the turbulence intensity is equal to 1%.
- We will study with more details how to setup the boundary conditions when we deal with turbulence modeling in the advanced modules.

Flow past a cylinder – From laminar to turbulent flow

The file *0/omega*

```
19      internalField   uniform 0.075;
20
21      boundaryField
22      {
23          out
24          {
25              type            inletOutlet;
26              inletValue      uniform 0.075;
27              value           uniform 0.075;
28          }
29          sym1
30          {
31              type            symmetryPlane;
32          }
33          sym2
34          {
35              type            symmetryPlane;
36          }
37          in
38          {
39              type            fixedValue;
40              value           uniform 0.075;
41          }
42          cylinder
43          {
44              type            omegaWallFunction;
45              Cmu             0.09;
46              kappa           0.41;
47              E               9.8;
48              beta1           0.075;
49              value           uniform 0.075;
50          }
51          back
52          {
53              type            empty;
54          }
55          front
56          {
57              type            empty;
58          }
59      }
```

- We are using uniform initial conditions (line 19).
- For the **in** patch we are using a **fixedValue** boundary condition.
- For the **out** patch we are using an **inletOutlet** boundary condition (this boundary condition avoids backflow).
- For the **cylinder** patch (which is **base type wall**), we are using the **omegaWallFunction** boundary condition. This is a wall function; we are going to talk about this when we deal with turbulence modeling. Remember, we can use wall functions only if the patch is of **base type wall**.
- The rest of the patches are constrained.
- FYI, the inlet velocity is 1 and the eddy viscosity ratio is equal to 10.
- We will study with more details how to setup the boundary conditions when we deal with turbulence modeling in the advanced modules.

Flow past a cylinder – From laminar to turbulent flow

The file *0/nut*

```
19    internalField   uniform 0;
20
21    boundaryField
22    {
23        out
24        {
25            type            calculated;
26            value           uniform 0;
27        }
28        sym1
29        {
30            type            symmetryPlane;
31        }
32        sym2
33        {
34            type            symmetryPlane;
35        }
36        in
37        {
38            type            calculated;
39            value           uniform 0;
40        }
41        cylinder
42        {
43            type            nutkWallFunction;
44            Cmu             0.09;
45            kappa           0.41;
46            E               9.8;
47            value           uniform 0;
48        }
49        back
50        {
51            type            empty;
52        }
53        front
54        {
55            type            empty;
56        }
57    }
```

- We are using uniform initial conditions (line 19).
- For the **in** patch we are using the **calculated** boundary condition (nut is computed from kappa and omega)
- For the **out** patch we are using the **calculated** boundary condition (nut is computed from kappa and omega)
- For the **cylinder** patch (which is **base type wall**), we are using the **nutkWallFunction** boundary condition. This is a wall function, we are going to talk about this when we deal with turbulence modeling. Remember, we can use wall functions only if the patch is of **base type wall**.
- The rest of the patches are constrained.
- Remember, the turbulent viscosity ν_t (nut) is equal to

$$\frac{\kappa}{\omega}$$

- We will study with more details how to setup the boundary conditions when we deal with turbulence modeling in the advanced modules.

Flow past a cylinder – From laminar to turbulent flow

Running the case – Setting a turbulent case

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c14`**
- Feel free to use the Fluent mesh or the mesh generated with `blockMesh`. In this case we will use `blockMesh`.
- To run this case, in the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> renumberMesh -overwrite`
4. `$> pimpleFoam | tee log.solver`

   You will need to launch this script in a different terminal
5. `$> pyFoamPlotWatcher.py log.solver`

   You will need to launch this script in a different terminal
6. `$> gnuplot scripts0/plot_coeffs`

   You will need to launch this script in a different terminal
7. `$> pimpleFoam -postprocess -func yPlus -latestTime -noFunctionObjects`
8. `$> paraFoam`

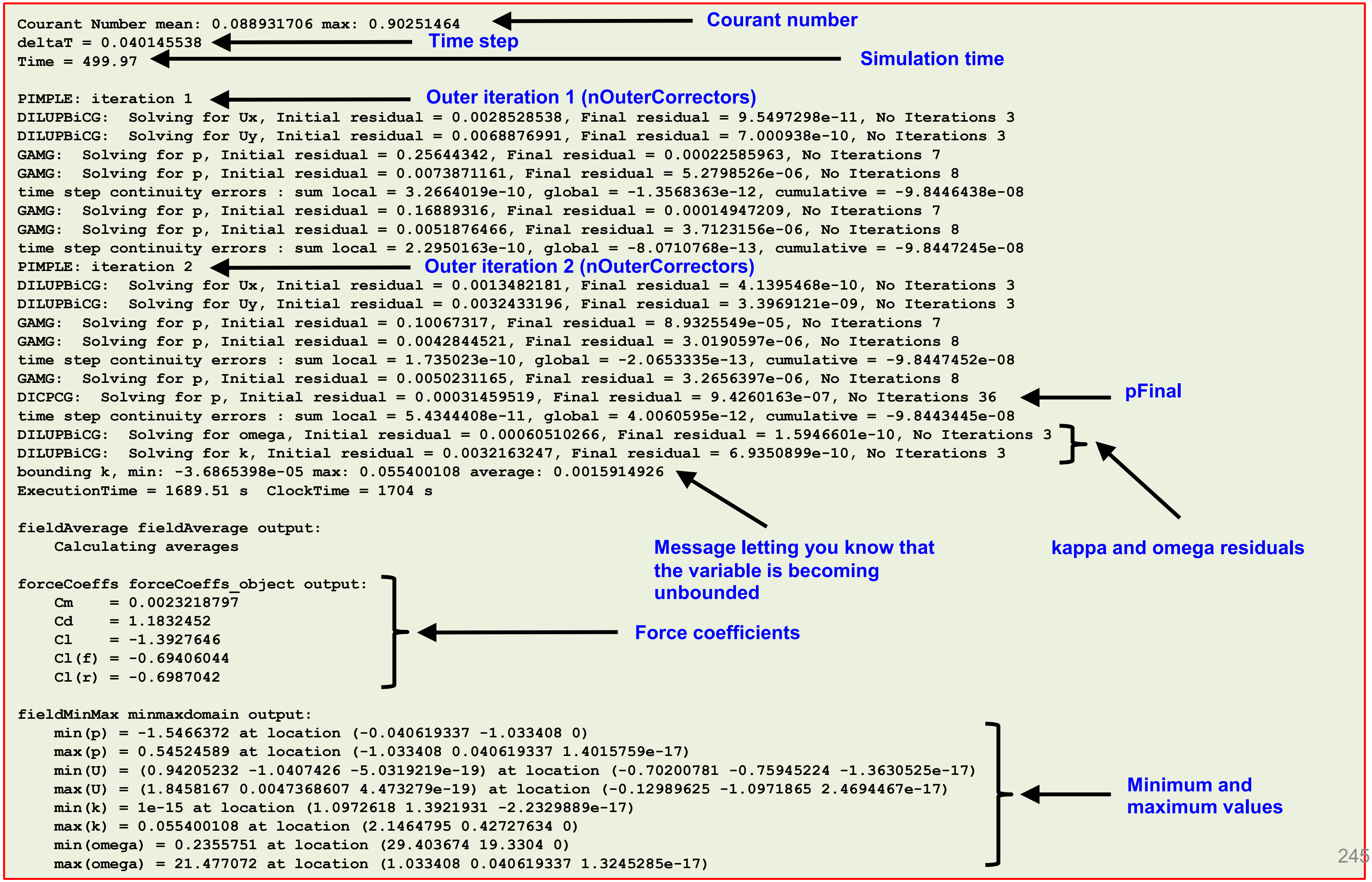
Flow past a cylinder – From laminar to turbulent flow

Running the case – Setting a turbulent case

- In step 3 we use the utility `renumberMesh` to make the linear system more diagonal dominant, this will speed-up the linear solvers.
- In step 4 we run the simulation and save the log file. Notice that we are sending the job to background.
- In step 5 we use `pyFoamPlotWatcher.py` to plot the residuals on-the-fly. As the job is running in background, we can launch this utility in the same terminal tab.
- In step 6 we use the gnuplot script `scripts0/plot_coeffs` to plot the force coefficients on-the-fly. Besides monitoring the residuals, is always a good idea to monitor a quantity of interest. Feel free to take a look at the script and to reuse it.
- In step 7 we use the utility `postProcess` to compute the y^+ value of each saved solution (we are going to talk about y^+ when we deal with turbulence modeling).

Flow past a cylinder – From laminar to turbulent flow

`pimpleFoam` **output screen**



```
Courant Number mean: 0.088931706 max: 0.90251464
deltaT = 0.040145538
Time = 499.97

PIMPLE: iteration 1
DILUPBiCG:  Solving for Ux, Initial residual = 0.0028528538, Final residual = 9.5497298e-11, No Iterations 3
DILUPBiCG:  Solving for Uy, Initial residual = 0.0068876991, Final residual = 7.000938e-10, No Iterations 3
GAMG:  Solving for p, Initial residual = 0.25644342, Final residual = 0.00022585963, No Iterations 7
GAMG:  Solving for p, Initial residual = 0.0073871161, Final residual = 5.2798526e-06, No Iterations 8
time step continuity errors : sum local = 3.2664019e-10, global = -1.3568363e-12, cumulative = -9.8446438e-08
GAMG:  Solving for p, Initial residual = 0.16889316, Final residual = 0.00014947209, No Iterations 7
GAMG:  Solving for p, Initial residual = 0.0051876466, Final residual = 3.7123156e-06, No Iterations 8
time step continuity errors : sum local = 2.2950163e-10, global = -8.0710768e-13, cumulative = -9.8447245e-08
PIMPLE: iteration 2
DILUPBiCG:  Solving for Ux, Initial residual = 0.0013482181, Final residual = 4.1395468e-10, No Iterations 3
DILUPBiCG:  Solving for Uy, Initial residual = 0.0032433196, Final residual = 3.3969121e-09, No Iterations 3
GAMG:  Solving for p, Initial residual = 0.10067317, Final residual = 8.9325549e-05, No Iterations 7
GAMG:  Solving for p, Initial residual = 0.0042844521, Final residual = 3.0190597e-06, No Iterations 8
time step continuity errors : sum local = 1.735023e-10, global = -2.0653335e-13, cumulative = -9.8447452e-08
GAMG:  Solving for p, Initial residual = 0.0050231165, Final residual = 3.2656397e-06, No Iterations 8
DICPCG:  Solving for p, Initial residual = 0.00031459519, Final residual = 9.4260163e-07, No Iterations 36
time step continuity errors : sum local = 5.4344408e-11, global = 4.0060595e-12, cumulative = -9.8443445e-08
DILUPBiCG:  Solving for omega, Initial residual = 0.00060510266, Final residual = 1.5946601e-10, No Iterations 3
DILUPBiCG:  Solving for k, Initial residual = 0.0032163247, Final residual = 6.9350899e-10, No Iterations 3
bounding k, min: -3.6865398e-05 max: 0.055400108 average: 0.0015914926
ExecutionTime = 1689.51 s  ClockTime = 1704 s

fieldAverage fieldAverage output:
    Calculating averages

forceCoeffs forceCoeffs_object output:
    Cm    = 0.0023218797
    Cd    = 1.1832452
    Cl    = -1.3927646
    Cl(f) = -0.69406044
    Cl(r) = -0.6987042

fieldMinMax minmaxdomain output:
    min(p) = -1.5466372 at location (-0.040619337 -1.033408 0)
    max(p) = 0.54524589 at location (-1.033408 0.040619337 1.4015759e-17)
    min(U) = (0.94205232 -1.0407426 -5.0319219e-19) at location (-0.70200781 -0.75945224 -1.3630525e-17)
    max(U) = (1.8458167 0.0047368607 4.473279e-19) at location (-0.12989625 -1.0971865 2.4694467e-17)
    min(k) = 1e-15 at location (1.0972618 1.3921931 -2.2329889e-17)
    max(k) = 0.055400108 at location (2.1464795 0.42727634 0)
    min(omega) = 0.2355751 at location (29.403674 19.3304 0)
    max(omega) = 21.477072 at location (1.033408 0.040619337 1.3245285e-17)
```

Courant number

Time step

Simulation time

Outer iteration 1 (nOuterCorrectors)

Outer iteration 2 (nOuterCorrectors)

pFinal

Message letting you know that the variable is becoming unbounded

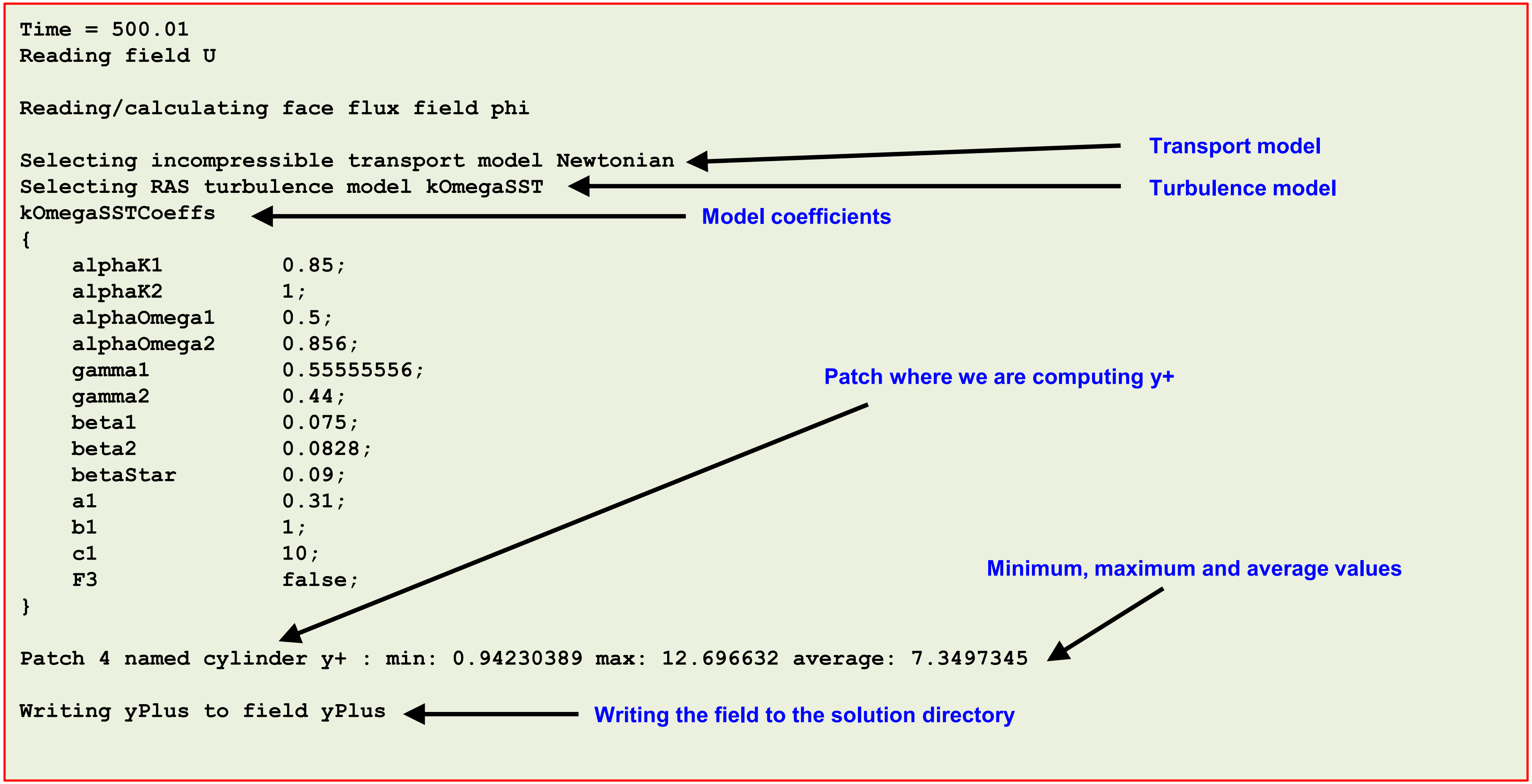
kappa and omega residuals

Force coefficients

Minimum and maximum values

The output screen

- This is the output screen of the `yPlus` utility.



```
Time = 500.01
Reading field U

Reading/calculating face flux field phi

Selecting incompressible transport model Newtonian
Selecting RAS turbulence model kOmegaSST
kOmegaSSTCoeffs
{
    alphaK1         0.85;
    alphaK2         1;
    alphaOmega1     0.5;
    alphaOmega2     0.856;
    gamma1          0.55555556;
    gamma2          0.44;
    beta1           0.075;
    beta2           0.0828;
    betaStar        0.09;
    a1              0.31;
    b1              1;
    c1              10;
    F3              false;
}

Patch 4 named cylinder y+ : min: 0.94230389 max: 12.696632 average: 7.3497345

Writing yPlus to field yPlus
```

Transport model

Turbulence model

Model coefficients

Patch where we are computing y+

Minimum, maximum and average values

Writing the field to the solution directory

Flow past a cylinder – From laminar to turbulent flow

Using a compressible solver

- So far, we have only used incompressible solvers.
- Let us use the compressible solver `rhoPimpleFoam`, which is a,

  Transient solver for laminar or turbulent flow of compressible fluids for HVAC and similar applications. Uses the flexible PIMPLE (PISO-SIMPLE) solution for time-resolved and pseudo-transient simulations.

- When working with compressible solver we need to define the thermodynamical properties of the working fluid and the temperature field (we are also solving the energy equation).
- Also remember, compressible solvers use absolute pressure. Conversely, incompressible solvers use relative pressure.
- This case is already setup in the directory

```
$PTOFC/101OF/vortex_shedding/c24
```

Flow past a cylinder – From laminar to turbulent flow

- The following dictionaries remain unchanged
  - *`system/blockMeshDict`*
  - *`constant/polyMesh/boundary`*

- Reminder:
  - The diameter of the cylinder is 0.002 m.
  - The working fluid is air at 20° Celsius and at a sea level.
  - Isothermal flow.
  - And we are targeting for a Re = 200.

$$\nu = \frac{\mu}{\rho} \qquad Re = \frac{\rho \times U \times D}{\mu} = \frac{U \times D}{\nu}$$

Flow past a cylinder – From laminar to turbulent flow

The **`constant`** directory

- In this directory, we will find the following compulsory dictionary files:

  - *`thermophysicalProperties`*
  - *`momentumTransport`*

- *`thermophysicalProperties`* contains the definition of the physical properties of the working fluid.
- *`momentumTransport`* contains the definition of the turbulence model to use.

Flow past a cylinder – From laminar to turbulent flow

The *thermophysicalProperties* dictionary file

```
18    thermoType
19    {
20        type            hePsiThermo;
21        mixture         pureMixture;
22        transport       const;
23        thermo          hConst;
24        equationOfState perfectGas;
25        specie          specie;
26        energy          sensibleEnthalpy;
27    }
28
29    mixture
30    {
31        specie
32        {
33            nMoles      1;
34            molWeight   28.9;
35        }
36        thermodynamics
37        {
38            Cp          1005;
39            Hf          0;
40        }
41        transport
42        {
43            mu          1.84e-05;
44            Pr          0.713;
45        }
46    }
```

- This dictionary file is located in the directory **constant**. Thermophysical models are concerned with energy, heat and physical properties.
- In the sub-dictionary **thermoType** (lines 18-27), we define the thermophysical models.
- The **transport** modeling concerns evaluating dynamic viscosity (line 22). In this case the viscosity is constant.
- The thermodynamic models (**thermo**) are concerned with evaluating the specific heat Cp (line 23). In this case Cp is constant
- The **equationOfState** keyword (line 24) concerns to the equation of state of the working fluid. In this case

$$\rho = \frac{p}{RT}$$

- The form of the energy equation to be used in the solution is specified in line 26 (**energy**). In this case we are using enthalpy (**sensibleEnthalpy**).

Flow past a cylinder – From laminar to turbulent flow

The *thermophysicalProperties* dictionary file

```
18    thermoType
19    {
20        type            hePsiThermo;
21        mixture         pureMixture;
22        transport       const;
23        thermo          hConst;
24        equationOfState perfectGas;
25        specie          specie;
26        energy          sensibleEnthalpy;
27    }
28
29    mixture
30    {
31        specie
32        {
33            nMoles      1;
34            molWeight   28.9;
35        }
36        thermodynamics
37        {
38            Cp          1005;
39            Hf          0;
40        }
41        transport
42        {
43            mu          1.84e-05;
44            Pr          0.713;
45        }
46    }
```

- In the sub-dictionary **mixture** (lines 29-46), we define the thermophysical properties of the working fluid.
- In this case, we are defining the properties for air at 20° Celsius and at a sea level.

Flow past a cylinder – From laminar to turbulent flow

The `turbulenceProperties` dictionary file

- In this dictionary file we select what model we would like to use (laminar or turbulent).
- This dictionary is compulsory.
- As we do not want to model turbulence, the dictionary is defined as follows,

```
simulationType    laminar;
```

Flow past a cylinder – From laminar to turbulent flow

The `0` directory

- In this directory, we will find the dictionary files that contain the boundary and initial conditions for all the primitive variables.
- As we are solving the compressible laminar Navier-Stokes equations, we will find the following field files:

  - `p` (pressure)
  - `T` (temperature)
  - `U` (velocity field)

Flow past a cylinder – From laminar to turbulent flow

The file *0/p*

```
17    dimensions      [1 -1 -2 0 0 0 0];
18
19    internalField   uniform 101325;
20
21    boundaryField
22    {
23        in
24        {
25            type            zeroGradient;
26        }
28        out
29        {
30            type            fixedValue;
31            value           uniform 101325;
32        }
34        cylinder
35        {
36            type            zeroGradient;
37        }
39        sym1
40        {
41            type            symmetryPlane;
42        }
44        sym2
45        {
46            type            symmetryPlane;
47        }
49        back
50        {
51            type            empty;
52        }
54        front
55        {
56            type            empty;
57        }
58    }
```

- This file contains the boundary and initial conditions for the scalar field pressure (**p**). **We are working with absolute pressure.**
- Contrary to incompressible flows where we defined relative pressure, this is the absolute pressure.
- Also, pay attention to the units (line 17). The pressure is defined in Pascal.
- We are using uniform initial conditions (line 19).
- For the **in** patch we are using a **zeroGradient** boundary condition.
- For the **outlet** patch we are using a **fixedValue** boundary condition.
- For the **cylinder** patch we are using a **zeroGradient** boundary condition.
- The rest of the patches are constrained.

Flow past a cylinder – From laminar to turbulent flow

The file *0/T*

```
17   dimensions      [0 0 0 -1 0 0 0];
18
19   internalField   uniform 293.15;
20
21   boundaryField
22   {
23       in
24       {
25           type            fixedValue;
26           value           $internalField;
27       }
29       out
30       {
31           type            inletOutlet;
32           value           $internalField;
33           inletValue      $internalField;
34       }
36       cylinder
37       {
38           type            zeroGradient;
39       }
41       sym1
42       {
43           type            symmetryPlane;
44       }
46       sym2
47       {
48           type            symmetryPlane;
49       }
51       back
52       {
53           type            empty;
54       }
56       front
57       {
58           type            empty;
59       }
60   }
```

- This file contains the boundary and initial conditions for the scalar field temperature (**T**).
- Also, pay attention to the units (line 17). The temperature is defined in Kelvin.
- We are using uniform initial conditions (line 19).
- For the **in** patch we are using a **fixedValue** boundary condition.
- For the **out** patch we are using a **inletOutlet** boundary condition (in case of backflow).
- For the **cylinder** patch we are using a **zeroGradient** boundary condition.
- The rest of the patches are constrained.

Flow past a cylinder – From laminar to turbulent flow

The file *0/U*

```
17    dimensions      [0 1 -1 0 0 0 0];
18
19    internalField   uniform (1.5 0 0);
20
21    boundaryField
22    {
23        in
24        {
25            type            fixedValue;
26            value           uniform (1.5 0 0);
27        }
29        out
30        {
31             type            inletOutlet;
32             phi             phi;
33             inletValue      uniform (0 0 0);
34             value           uniform (0 0 0);
35        }
37        cylinder
38        {
39            type            fixedValue;
40            value           uniform (0 0 0);
41        }
43        sym1
44        {
45            type            symmetryPlane;
46        }
48        sym2
49        {
50            type            symmetryPlane;
51        }
53        back
54        {
55            type            empty;
56        }
58        front
59        {
60            type            empty;
61        }
62    }
```

- This file contains the boundary and initial conditions for the dimensional vector field **U**.
- We are using uniform initial conditions and the numerical value is **(1.5 0 0)** (keyword **internalField** in line 19).
- For the **in** patch we are using a **fixedValue** boundary condition.
- For the **out** patch we are using a **inletOutlet** boundary condition (in case of backflow).
- For the **cylinder** patch we are using a **zeroGradient** boundary condition.
- The rest of the patches are constrained.

Flow past a cylinder – From laminar to turbulent flow

The **`system`** directory

- The **`system`** directory consists of the following compulsory dictionary files:
  - *`controlDict`*
  - *`fvSchemes`*
  - *`fvSolution`*

- *`controlDict`* contains general instructions on how to run the case.
- *`fvSchemes`* contains instructions for the discretization schemes that will be used for the different terms in the equations.
- *`fvSolution`* contains instructions on how to solve each discretized linear equation system.

The *controlDict* dictionary

```
17     application      rhoPimpleFoam;
18
19     startFrom        startTime;
20     //startFrom        latestTime;
21
22     startTime        0;
23
24     stopAt           endTime;
25     //stopAt   writeNow;
26
27     endTime          0.3;
28
29     deltaT           0.00001;
30
31     writeControl     adjustableRunTime;
32
33     writeInterval    0.0025;
34
35     purgeWrite       0;
36
37     writeFormat      ascii;
38
39     writePrecision   10;
40
41     writeCompression off;
42
43     timeFormat       general;
44
45     timePrecision    6;
46
47     runTimeModifiable true;
48
49     adjustTimeStep   yes;
50     maxCo            1;
51     maxDeltaT        1;
```

- This case will start from the last saved solution (**startFrom**). If there is no solution, the case will start from time 0 (**startTime**).
- It will run up to 0.3 seconds (**endTime**).
- The initial time step of the simulation is 0.00001 seconds (**deltaT**).
- It will write the solution every 0.0025 seconds (**writeInterval**) of simulation time (**adjustableRunTime**). The option **adjustableRunTime** will adjust the time-step to save the solution at the precise intervals. This may add some oscillations in the solution as the CFL is changing.
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.
- In line 49 we turn on the option **adjustTimeStep**. This option will automatically adjust the time step to achieve the maximum desired courant number (line 50).
- We also set a maximum time step in line 51.
- Remember, the first time step of the simulation is done using the value set in line 28 and then it is automatically scaled to achieve the desired maximum values (lines 66-67).
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.

Flow past a cylinder – From laminar to turbulent flow

The `controlDict` dictionary

```
55     functions
56     {

178    forceCoeffs_object
179    {
188     type forceCoeffs;
189     functionObjectLibs ("libforces.so");
190     patches (cylinder);
191
192     pName p;
193     Uname U;
194     //rhoName rhoInf;
195     rhoInf 1.205;
196
197     //// Dump to file
198     log true;
199
200     CofR (0.0 0 0);
201     liftDir (0 1 0);
202     dragDir (1 0 0);
203     pitchAxis (0 0 1);
204     magUInf 1.5;
205     lRef 0.001;
206     Aref 0.000002;
207
208      outputControl   timeStep;
209      outputInterval  1;
210      }


235
236    };
```

- As usual, at the bottom of the `controlDict` dictionary file we define the **functionObjects** (lines 55-236).
- Of special interest is the **functionObject forceCoeffs_object**.
- As we changed the domain dimensions and the inlet velocity, we need to update the reference values (lines 204-206).
- It is also important to update the reference density (line 195).

Flow past a cylinder – From laminar to turbulent flow

The *fvSchemes* dictionary

```
ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default           cellLimited leastSquares 1;
}

divSchemes
{
    default         none;
    div(phi,U)      Gauss linearUpwindV default;

    div(phi,K)      Gauss linear;
    div(phi,h)      Gauss linear;

    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear limited 1;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         limited 1;
}
```

- In this case, for time discretization (**ddtSchemes**) we are using the **Euler** method.
- For gradient discretization (**gradSchemes**) we are using the **leastSquares** method.
- For the discretization of the convective terms (**divSchemes**) we are using **linearUpwind** interpolation with no slope limiters for the term **div(phi,U)**.
- For the terms **div(phi,K)** (kinetic energy) and **div(phi,h)** (enthalpy) we are using **linear** interpolation method with no slope limiters.
- For the term **div(((rho*nuEff)*dev2(T(grad(U)))))** we are using **linear** interpolation (this term is related to the turbulence modeling).
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear limited 1** method.
- This method is second order accurate.

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.01;
        minIter          2;
    }
    pFinal
    {
        $p;
        relTol          0;
        minIter          2;
    }
    "U.*"
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
        minIter          2;
    }
    hFinal
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
        minIter          2;
    }
    "rho.*"
    {
        solver         diagonal;
    }
}
```

- To solve the pressure (**p**) we are using the **PCG** method with an absolute **tolerance** of 1e-6 and a relative tolerance **relTol** of 0.01.
- The entry **pFinal** refers to the final correction of the **PISO** loop. Notice that we are using macro expansion (**$p)** to copy the entries from the sub-dictionary **p**.
- To solve **U** and **UFinal** (**U.***) we are using the solver **PBiCGStab** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0.
- To solve **hFinal** (enthalpy) we are using the solver **PBiCGStab** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0.
- To solve **rho** and **rhoFinal** (**rho.***) we are using the **diagonal** solver (remember rho is found from the equation of state, so this is a back-substitution).
- FYI, solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.
- Be careful with the enthalpy, it might cause oscillations.

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
88
89      PIMPLE
90      {
91          momentumPredictor yes;
92          nOuterCorrectors  1;
93          nCorrectors      2;
94          nNonOrthogonalCorrectors 1;
103          pMinFactor           0.5;
104          pMaxFactor           2.0;
105      }
```

- The **PIMPLE** sub-dictionary contains entries related to the pressure-velocity coupling (in this case the **PIMPLE** method).
- Setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- Hereafter we are doing 2 **PISO** correctors (**nCorrectors**) and 1 non-orthogonal corrections (**nNonOrthogonalCorrectors**).
- In lines 95-96 we set the minimum and maximum physical values of rho (density).
- If we increase the number of **nCorrectors** and **nNonOrthogonalCorrectors** we gain more stability but at a higher computational cost.
- The choice of the number of corrections is driven by the quality of the mesh and the physics involve.
- You need to do at least one **PISO** loop (**nCorrectors**).

Flow past a cylinder – From laminar to turbulent flow

Running the case – Using a compressible solver

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c24`**
- Feel free to use the Fluent mesh or the mesh generated with `blockMesh`. In this case we will use `blockMesh`.
- To run this case, in the terminal window type:

1.
   ```
   $> foamCleanTutorials
   ```

2.
   ```
   $> blockMesh
   ```

3.
   ```
   $> transformPoints -scale ‘(0.001 0.001 0.001)’
   ```

4.
   ```
   $> renumberMesh -overwrite
   ```

5.
   ```
   $> rhoPimpleFoam | tee log
   ```

6.
   ```
   $> pyFoamPlotWatcher.py log
   ```
   You will need to launch this script in a different terminal

7.
   ```
   $> gnuplot scripts0/plot_coeffs
   ```
   You will need to launch this script in a different terminal

8.
   ```
   $> rhoPimpleFoam -postProcess -func MachNo
   ```

9.
   ```
   $> paraFoam
   ```

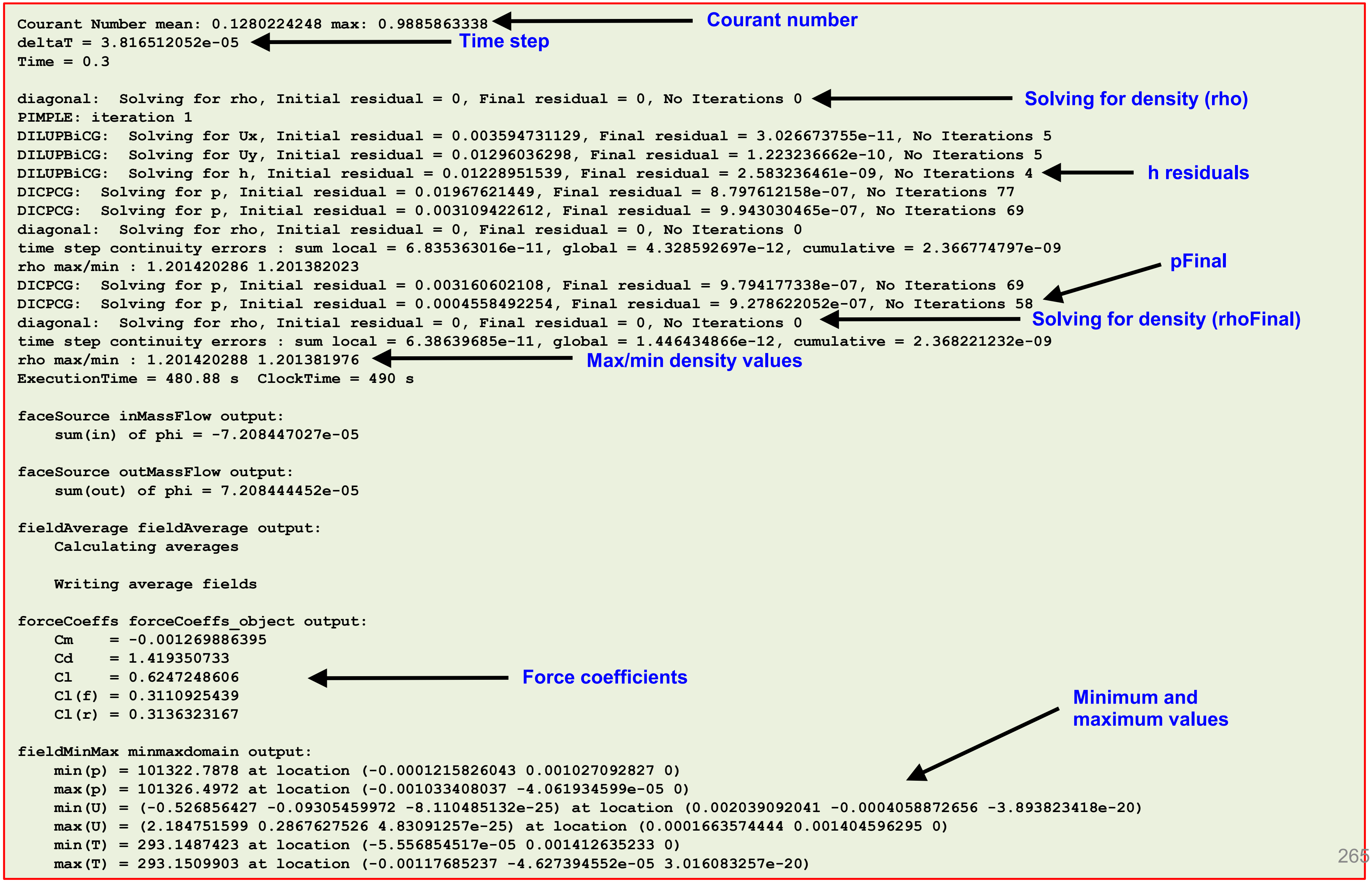
Flow past a cylinder – From laminar to turbulent flow

Running the case – Using a compressible solver

- In step 3 we scale the mesh.
- In step 4 we use the utility `renumberMesh` to make the linear system more diagonal dominant, this will speed-up the linear solvers.
- In step 5 we run the simulation and save the log file. Notice that we are sending the job to background.
- In step 6 we use `pyFoamPlotWatcher.py` to plot the residuals on-the-fly. As the job is running in background, we can launch this utility in the same terminal tab.
- In step 7 we use the gnuplot script `scripts0/plot_coeffs` to plot the force coefficients on-the-fly. Besides monitoring the residuals, is always a good idea to monitor a quantity of interest. Feel free to take a look at the script and to reuse it.
- In step 8 we use the utility `MachNo` to compute the Mach number.

Flow past a cylinder – From laminar to turbulent flow

`rhoPimpleFoam` **output screen**



```
Courant Number mean: 0.1280224248 max: 0.9885863338
deltaT = 3.816512052e-05
Time = 0.3

diagonal:  Solving for rho, Initial residual = 0, Final residual = 0, No Iterations 0
PIMPLE: iteration 1
DILUPBiCG:  Solving for Ux, Initial residual = 0.003594731129, Final residual = 3.026673755e-11, No Iterations 5
DILUPBiCG:  Solving for Uy, Initial residual = 0.01296036298, Final residual = 1.223236662e-10, No Iterations 5
DILUPBiCG:  Solving for h, Initial residual = 0.01228951539, Final residual = 2.583236461e-09, No Iterations 4
DICPCG:  Solving for p, Initial residual = 0.01967621449, Final residual = 8.797612158e-07, No Iterations 77
DICPCG:  Solving for p, Initial residual = 0.003109422612, Final residual = 9.943030465e-07, No Iterations 69
diagonal:  Solving for rho, Initial residual = 0, Final residual = 0, No Iterations 0
time step continuity errors : sum local = 6.835363016e-11, global = 4.328592697e-12, cumulative = 2.366774797e-09
rho max/min : 1.201420286 1.201382023
DICPCG:  Solving for p, Initial residual = 0.003160602108, Final residual = 9.794177338e-07, No Iterations 69
DICPCG:  Solving for p, Initial residual = 0.0004558492254, Final residual = 9.278622052e-07, No Iterations 58
diagonal:  Solving for rho, Initial residual = 0, Final residual = 0, No Iterations 0
time step continuity errors : sum local = 6.38639685e-11, global = 1.446434866e-12, cumulative = 2.368221232e-09
rho max/min : 1.201420288 1.201381976
ExecutionTime = 480.88 s  ClockTime = 490 s

faceSource inMassFlow output:
    sum(in) of phi = -7.208447027e-05

faceSource outMassFlow output:
    sum(out) of phi = 7.208444452e-05

fieldAverage fieldAverage output:
    Calculating averages

    Writing average fields

forceCoeffs forceCoeffs_object output:
    Cm    = -0.001269886395
    Cd    = 1.419350733
    Cl    = 0.6247248606
    Cl(f) = 0.3110925439
    Cl(r) = 0.3136323167

fieldMinMax minmaxdomain output:
    min(p) = 101322.7878 at location (-0.0001215826043 0.001027092827 0)
    max(p) = 101326.4972 at location (-0.001033408037 -4.061934599e-05 0)
    min(U) = (-0.526856427 -0.09305459972 -8.110485132e-25) at location (0.002039092041 -0.0004058872656 -3.893823418e-20)
    max(U) = (2.184751599 0.2867627526 4.83091257e-25) at location (0.0001663574444 0.001404596295 0)
    min(T) = 293.1487423 at location (-5.556854517e-05 0.001412635233 0)
    max(T) = 293.1509903 at location (-0.00117685237 -4.627394552e-05 3.016083257e-20)
```

Courant number
Time step
Solving for density (rho)
h residuals
pFinal
Solving for density (rhoFinal)
Max/min density values
Force coefficients
Minimum and maximum values

Flow past a cylinder – From laminar to turbulent flow

- In the directory **`$PTOFC/101OF/vortex_shedding`**, you will find 29 variations of the cylinder case involving different solvers and models. Feel free to explore all them.
- This is what you will find in each directory,
  - c1 = blockMesh – icoFoam – Unsteady solver – Re = 200.
  - c2 = fluentMeshToFoam – icoFoam – Unsteady solver – Re = 200.
  - c3 = blockMesh – pisoFoam – Unsteady solver – Field initialization – Re = 200.
  - c4 = blockMesh – potentialFoam – Re = 200.
  - c5 = blockMesh – mapFields – pisoFoam – Unsteady solver – original mesh – Re = 200.
  - c6 = blockMesh – mapFields – pisoFoam – Unsteady solver – Finer mesh – Re = 200.
  - c7 = blockMesh – pimpleFoam – Unsteady solver – Re = 200 – No turbulent model.
  - c8 = blockMesh – pisoFoam – Unsteady solver – Re = 200 – No turbulent model.
  - c9 = blockMesh – pisoFoam – Unsteady solver – Re = 200 – K-Omega SST turbulent model.
  - c10 = blockMesh – simpleFoam – Steady solver – Re = 200 – No turbulent model.
  - c11 = blockMesh – simpleFoam – Steady solver – Re = 40 – No turbulent model.
  - c12 = blockMesh – pisoFoam – Unsteady solver – Re = 40 – No turbulent model.
  - c14 = blockMesh – pimpleFoam – Unsteady solver – Re = 10000 – K-Omega SST turbulence model with wall functions.
  - c15 = blockMesh – pimpleFoam – Unsteady solver – Re = 100000 – K-Omega SST turbulence model with wall functions
  - c16 = blockMesh – simpleFoam – Steady solver – Re = 100000 – K-Omega SST turbulence model no wall functions.
  - c17 = blockMesh – simpleFoam – Steady solver – Re = 100000 – K-Omega SST turbulent model with wall functions.
  - c18 = blockMesh – pisoFoam – Unsteady solver – Re = 100000, LES Smagorinsky turbulent model.

Flow past a cylinder – From laminar to turbulent flow

- This is what you will find in each directory,
  - c19 = blockMesh – pimpleFoam – Unsteady solver – Re = 1000000 – Spalart Allmaras turbulent model no wall functions.
  - c20 = blockMesh – rhoPimpleFoam – Unsteady solver – Mach = 2.0 – Compressible – Laminar.
  - c21 = blockMesh – rhoPimpleFoam –Unsteady solver – Mach = 2.0 – Unsteady solver – Compressible – K-Omega SST turbulent model with wall functions.
  - c22 = blockMesh – rhoSimpleFoam – Mach = 2.0 – Steady solver – Compressible – K-Omega SST turbulent model with wall functions.
  - c23 = blockMesh – rhoPimpleFoam – Mach = 2.0 – LTS Pseudo-transient solver – Compressible – K-Omega SST turbulent model with wall functions.
  - c24 = blockMesh – pimpleFoam – Unsteady solver – Re = 200 – No turbulent model – Source terms (momentum)
  - c25 = blockMesh – pimpleFoam – Unsteady solver – Re = 200 – No turbulent model – Source terms (scalar transport)
  - c26 = blockMesh – rhoPimpleFoam – Unsteady solver – Re = 200 – Laminar, isothermal
  - c27 = blockMesh – rhoPimpleFoam – Unsteady solver – Re = 20000 – Turbulent, compressible
  - c28 = blockMesh – pimpleDyMFoam – Unsteady solver – Re = 200 – Laminar, moving cylinder (oscillating).
  - c29 = blockMesh – pimpleDyMFoam/pimpleFoam – Unsteady solver – Re = 200 – Laminar, rotating cylinder using AMI patches.
  - c30 = blockMesh – interFoam – Unsteady solver – Laminar, multiphase, free surface.
  - c31 = blockMesh – pimpleFoam – Unsteady solver – Laminar with source terms and AMR.

Module 2

Solid modeling

Roadmap

1. **Solid modeling preliminaries and introduction to Onshape**

Solid modeling – Preliminaries

- There is no wrong or right way when doing solid modeling for CFD. The only rule you should keep in mind is that by the end of the day you should get a smooth, clean, and watertight geometry.
- A watertight geometry means a close body with no holes or overlapping surfaces.
- Have in mind that the quality of the mesh and hence of the solution, greatly depends on the geometry. So always do your best when creating the geometry.
- During this solid modeling session we are going to show you how to get started with the geometry generation tools. The rest is on you.
- The best way to learn how to use these tools is by doing.
- The tool of our choice is Onshape (www.onshape.com). However, have in mind that all CAD and solid modeling applications have similar capabilities.

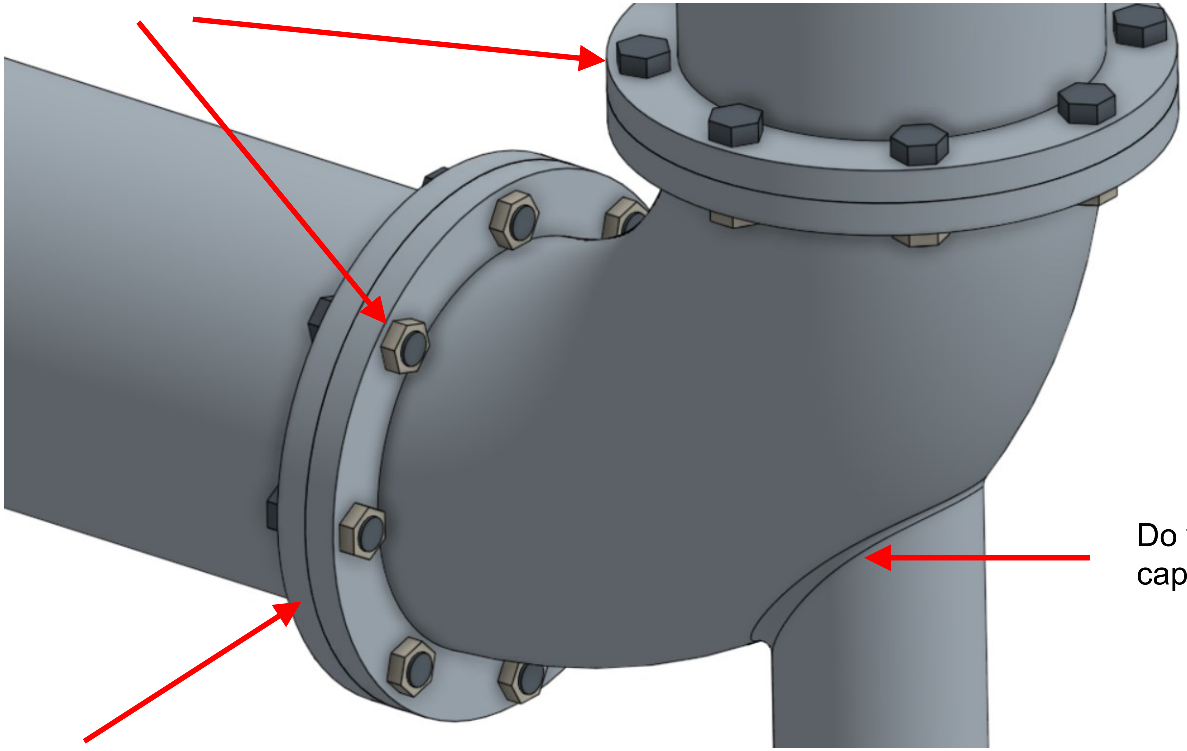
Solid modeling – Preliminaries

- There are always many ways to accomplish a task when creating a geometry, this give you the freedom to work in a way that is comfortable to you. Hereafter we are going to show you our way.
- Before starting to create your geometry, think about a strategy to employ to create your design, this is what we call design intent.
    - Choose one feature over other.
    - Dimensioning strategy.
    - Order of the operations.
    - Parametrization.
    - Single or multiple parts.
    - Do I need to parametrize my design, or should I use direct modeling?
- We are going to work with design intent during the hands-on sessions.

Solid modeling – Preliminaries

Geometry defeaturing

- Many times, it is not necessary to model all the details of the geometry. In these cases you should consider simplifying the geometry (geometry defeaturing).
- **Geometry defeaturing** can save you a lot of time when generating the mesh. So be smart, and use it whenever is possible.



Are the nuts and bolts necessary in my simulation?

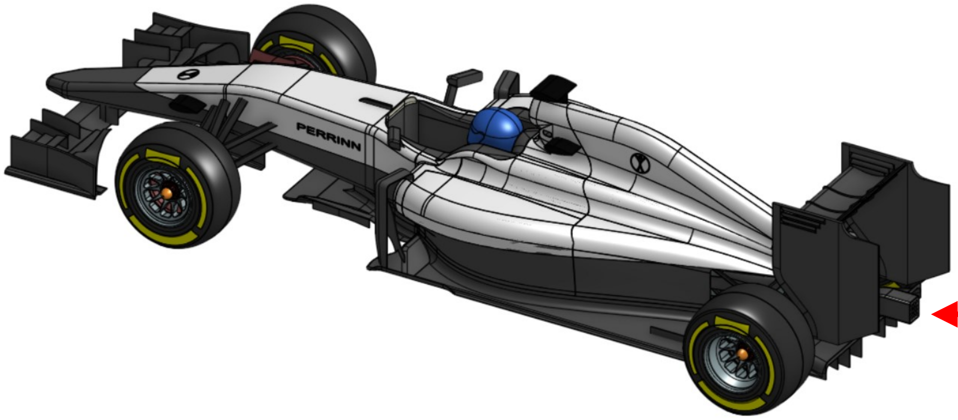
Do we really need to capture the fillet details?

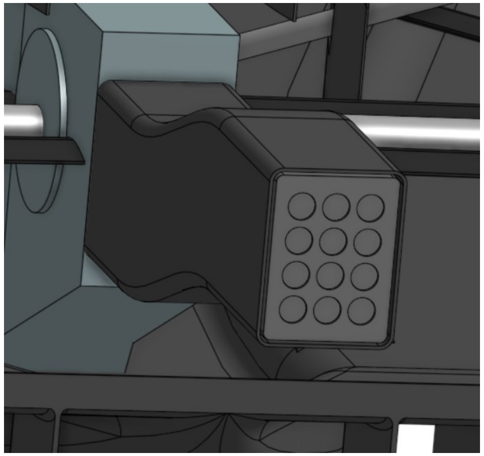
Do we need to model the flange?

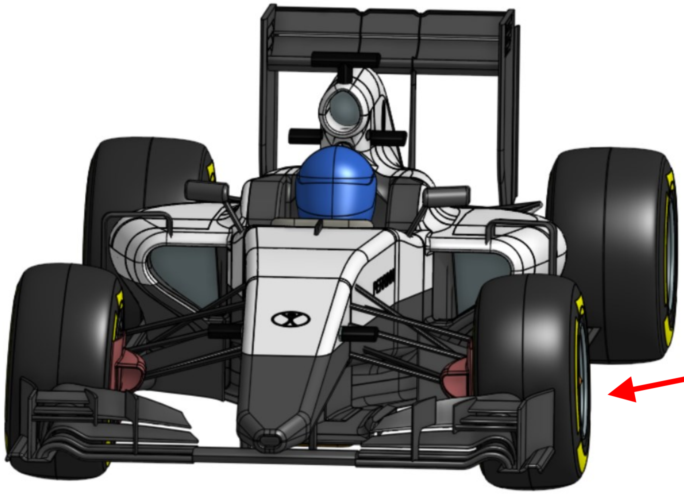
Solid modeling – Preliminaries

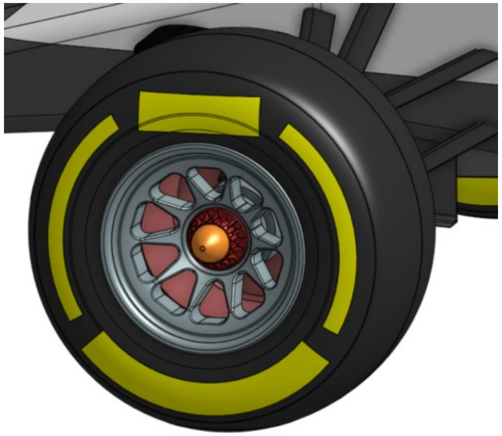
Geometry defeaturing

- Would you use all these geometry details for a CFD simulations?







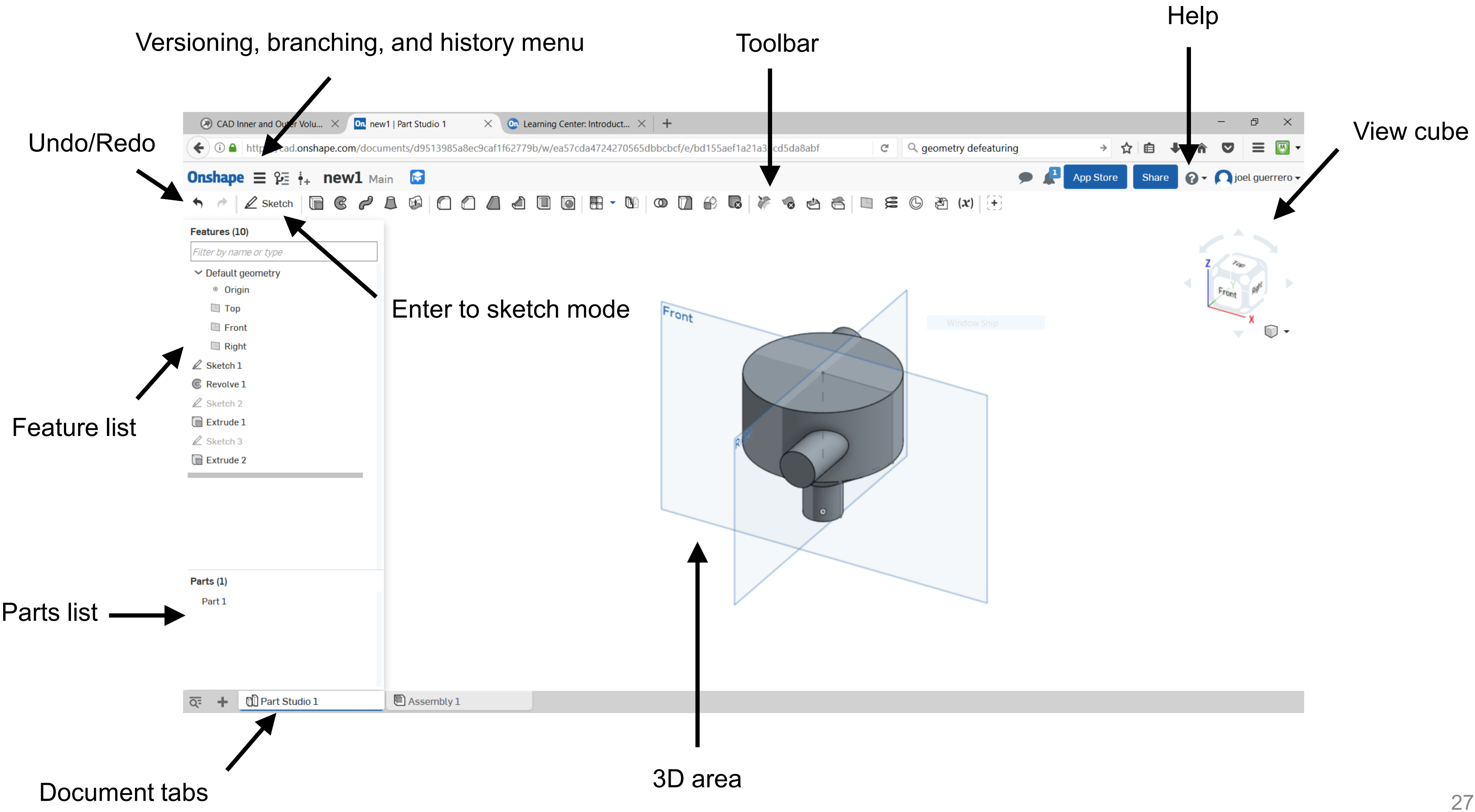


Solid modeling using Onshape

- Onshape is a professional CAD/solid modeling application.
- It provides powerful parametric and direct modeling capabilities.
- It is cloud based therefore you do not need to install any software.
- Documents are shareable.
- Multiple users can work in the same document at the same time (simultaneous editing).
- It runs in any device with a working web browser.
- Users can implement their own features using Onshape scripting language (featureScript).
- Users can access and modify documents in a programmatic way using python or nodejs.
- It is freely available for educational use and personal use.
- To start using Onshape register at: https://cad.onshape.com/

Solid modeling using Onshape

- Even if you have not used a CAD software before, you will find the GUI easy to use.
- You will notice that there is no save button because everything you do is automatically saved.



Help

Versioning, branching, and history menu

Toolbar

View cube

Undo/Redo

Enter to sketch mode

Feature list

Parts list

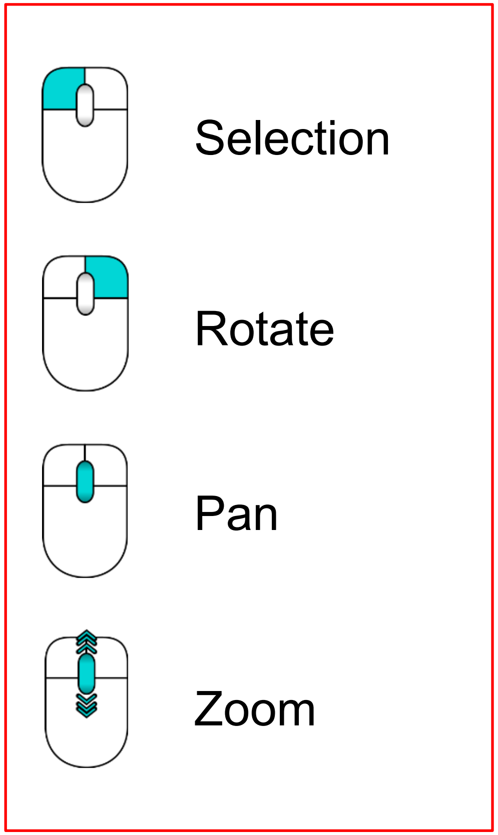
Document tabs

3D area

Solid modeling using Onshape

- Mouse interaction in the 3D area (it can be configured in the preference area).

Mouse interaction in the 3D viewer

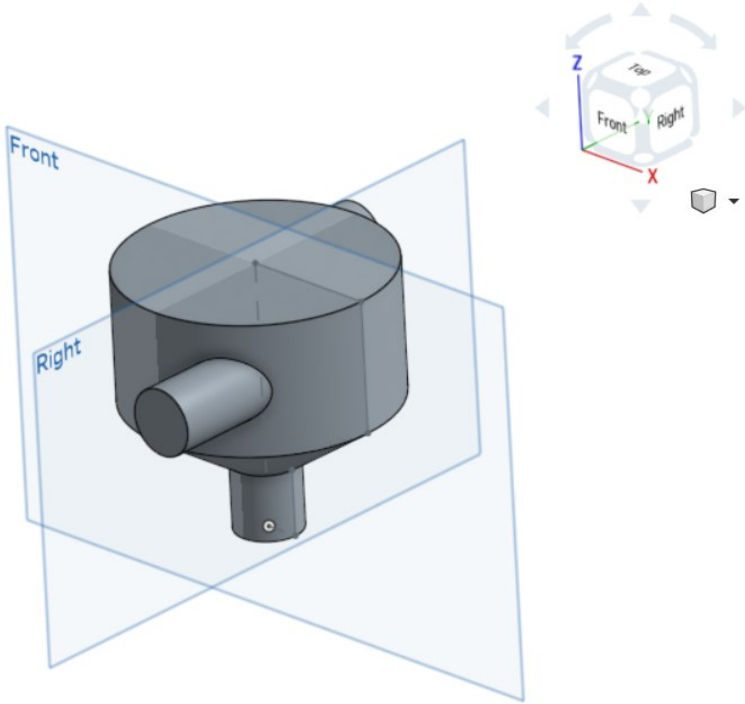


Selection

Rotate

Pan

Zoom

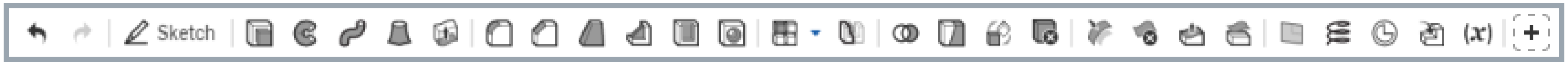


- To deselect click in an empty region in the 3D area

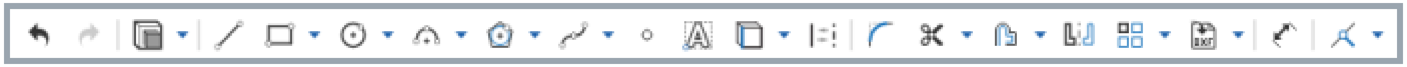
Solid modeling using Onshape

- Parametric modeling and feature based modeling are crucial components in the design experience.
- In Onshape you will find the following features:

Feature toolbar:



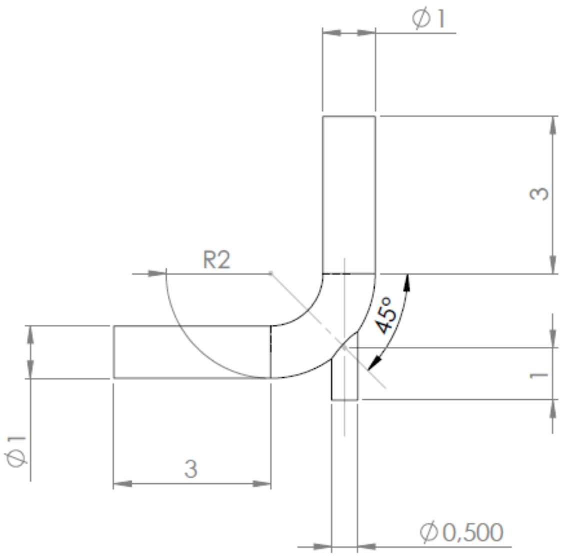
Sketch toolbar:

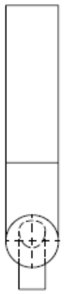


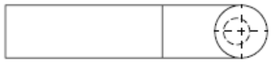
- Remember, sketches are the core of good 3D designs and parametrization.
- This is all we need to know about Onshape.
- Let us work with a simple geometry to understand how Onshape works.
- We also will show you a few clicks and picks you should be aware of.

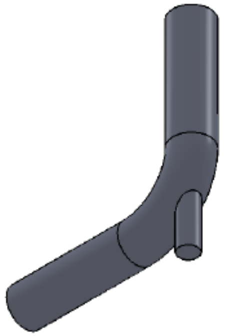
Solid modeling using Onshape

- Let us create this solid using the dimensions illustrated.







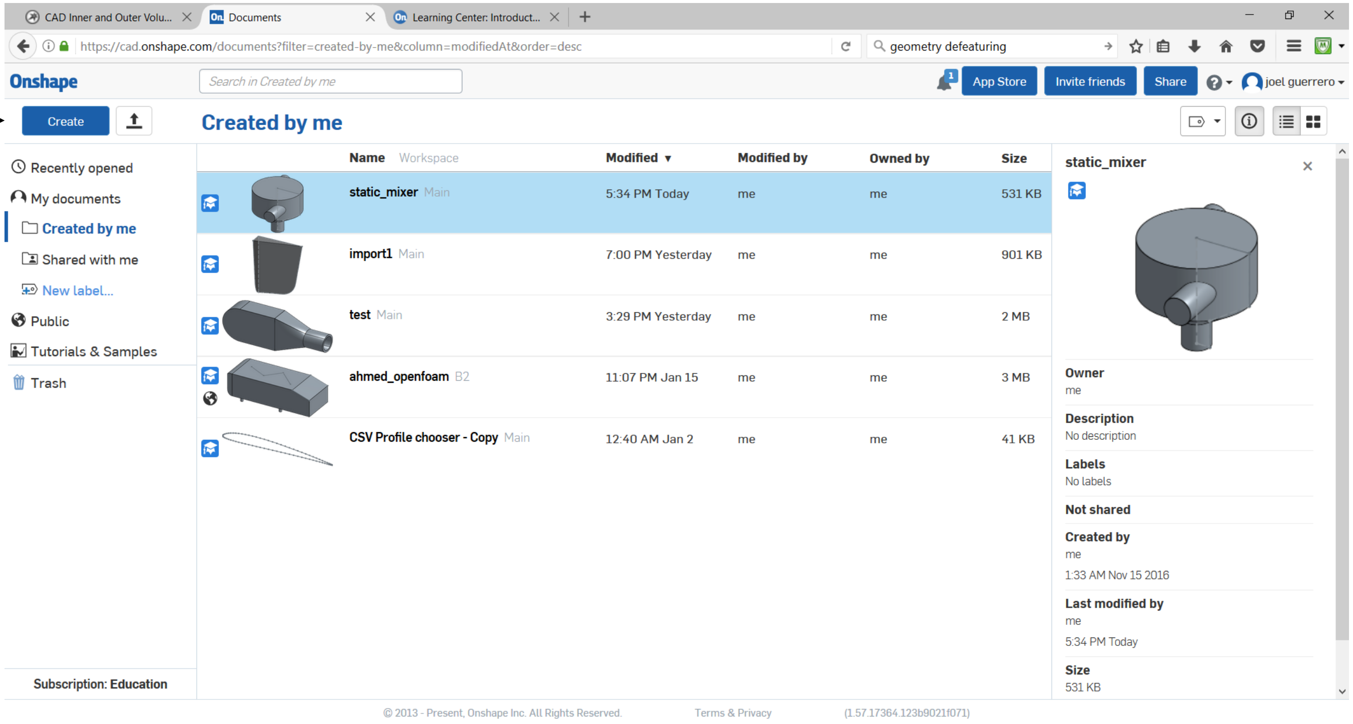


Note: all the dimensions are in meters

Solid modeling using Onshape

- Enter the document page and create a new design

Create new document



Created by me

Name	Workspace	Modified ▾	Modified by	Owned by	Size
static_mixer	Main	5:34 PM Today	me	me	531 KB
import1	Main	7:00 PM Yesterday	me	me	901 KB
test	Main	3:29 PM Yesterday	me	me	2 MB
ahmed_openfoam	B2	11:07 PM Jan 15	me	me	3 MB
CSV Profile chooser - Copy	Main	12:40 AM Jan 2	me	me	41 KB

static_mixer

Owner
me

Description
No description

Labels
No labels

Not shared

Created by
me
1:33 AM Nov 15 2016

Last modified by
me
5:34 PM Today

Size
531 KB

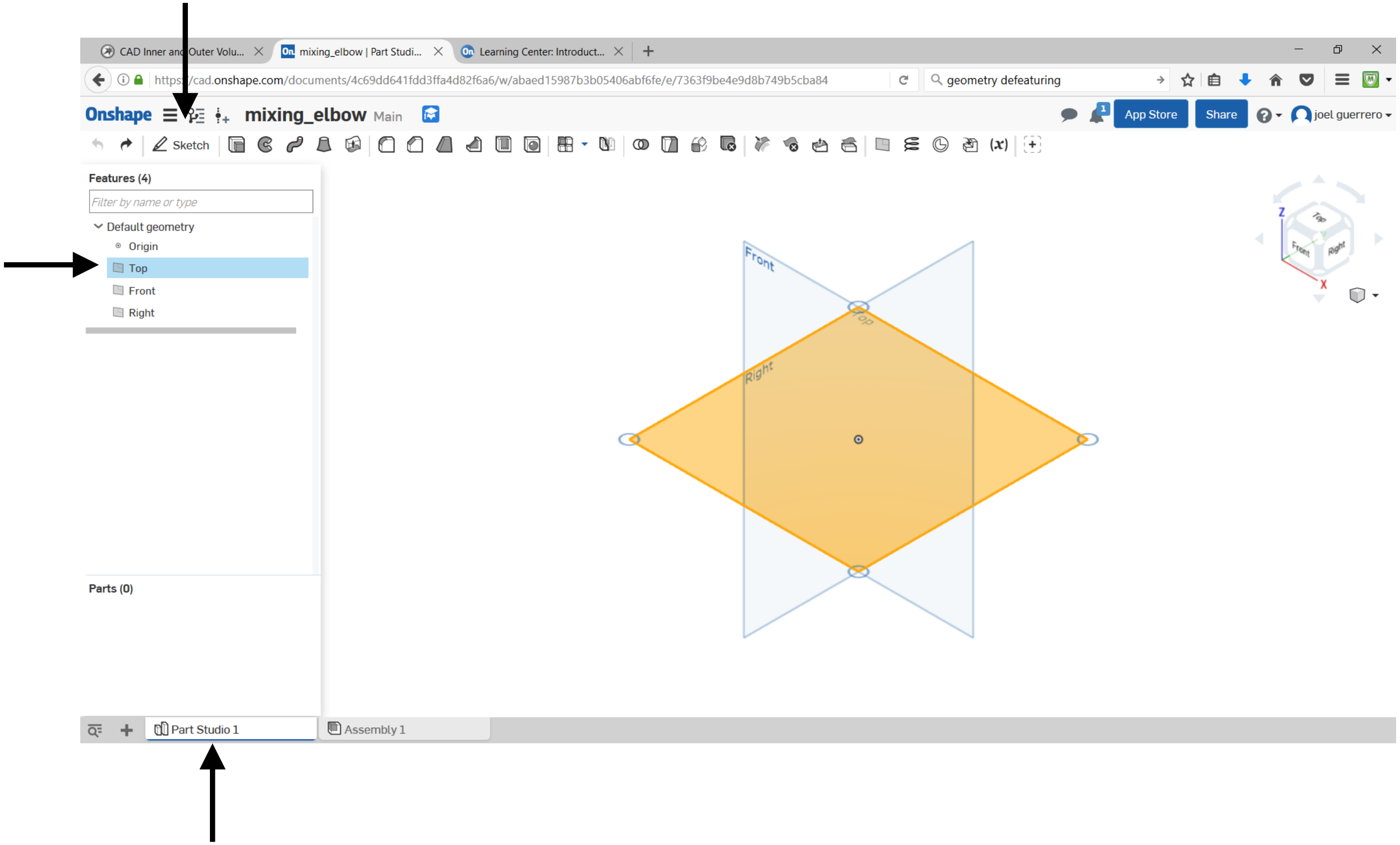
© 2013 - Present, Onshape Inc. All Rights Reserved. Terms & Privacy (1.57.17364.123b9021f071)

Solid modeling using Onshape

- In the part studio page, select the top plane and start a new sketch.

2. Start a new sketch

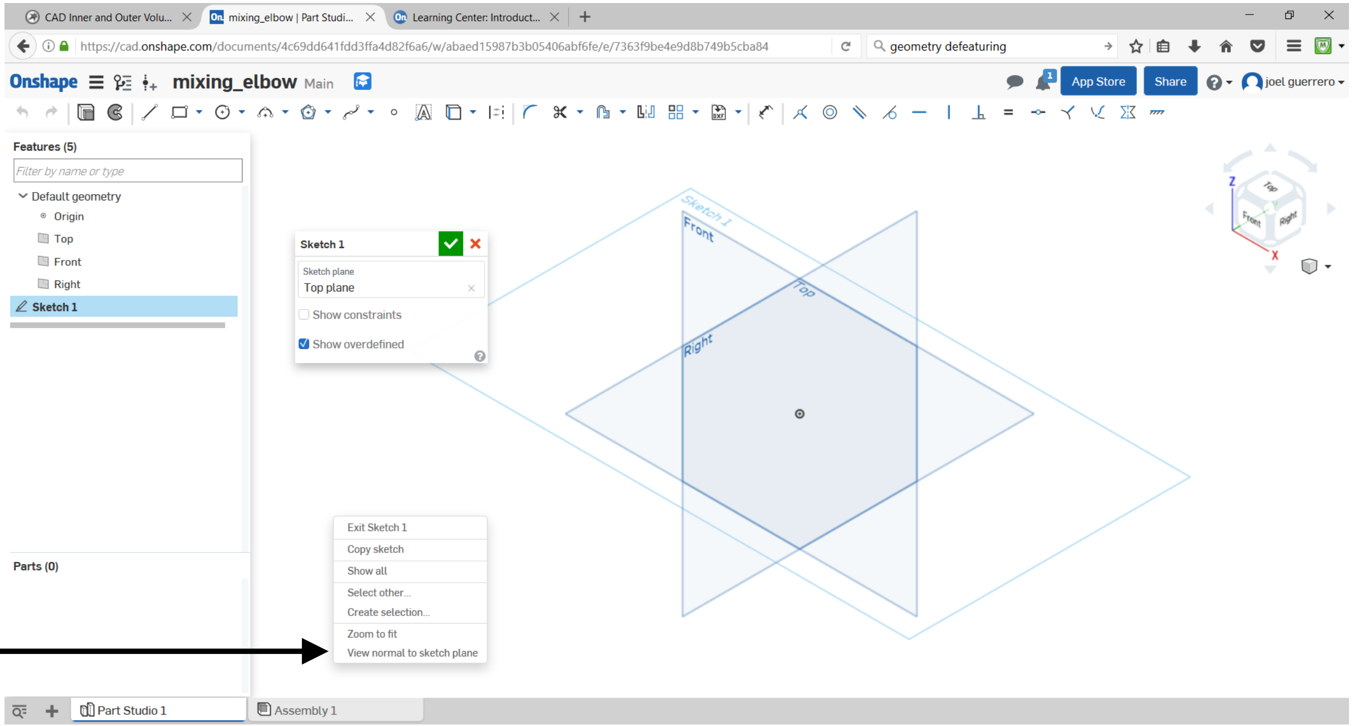
1. Select this plane


CAD Inner and Outer Volu...
mixing_elbow | Part Studi...
Learning Center: Introduct...
https://cad.onshape.com/documents/4c69dd641fdd3ffa4d82f6a6/w/abaed15987b3b05406abf6fe/e/7363f9be4e9d8b749b5cba84
geometry defeaturing
Onshape
mixing_elbow Main
App Store
Share
joel guerrero
Sketch
Features (4)
Filter by name or type
Default geometry
Origin
Top
Front
Right
Parts (0)
Part Studio 1
Assembly 1
Front
Top
Right

Part studio page

Solid modeling using Onshape

- In the part studio page, select the top plane and start a new sketch.



Sketch 1

Sketch plane: Top plane

□ Show constraints

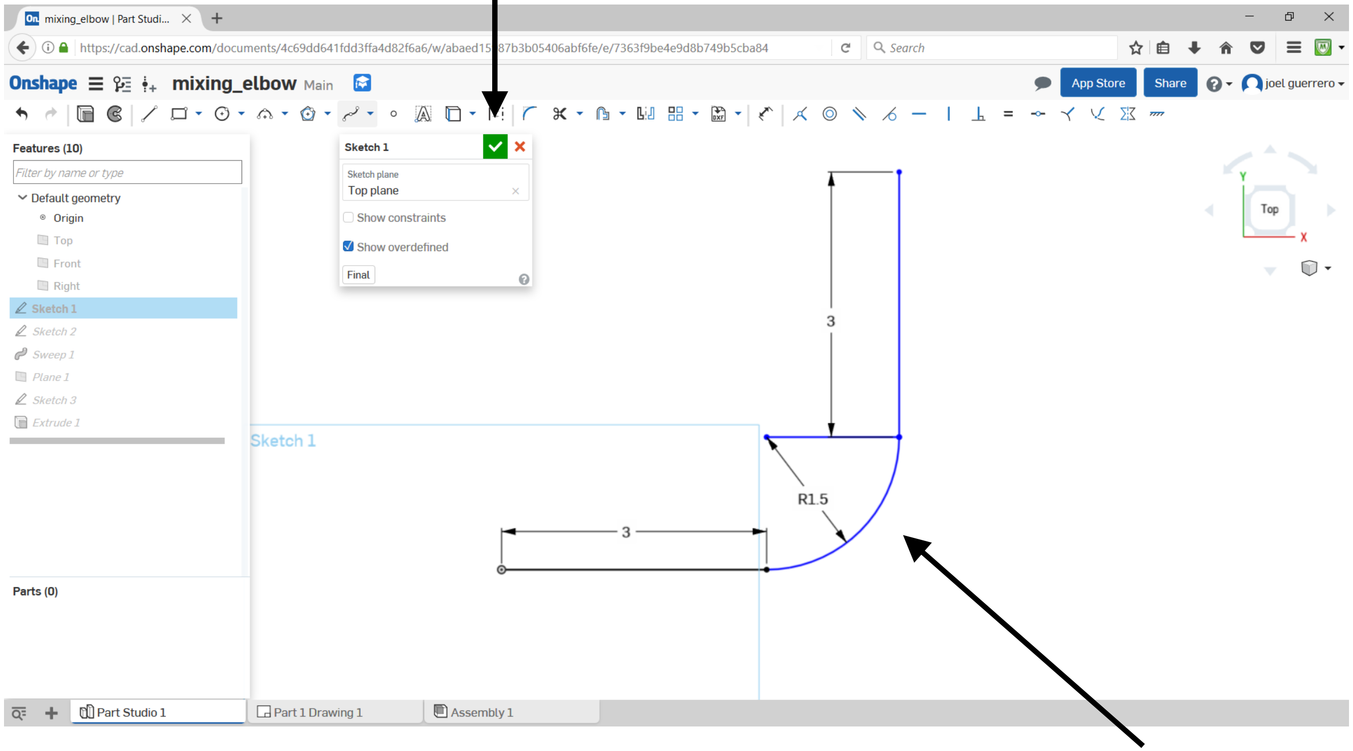
☑ Show overdefined

Right click on the 3D area and select view normal to sketch plane

Solid modeling using Onshape

- Using the sketching features, draw the following line.

When you are done sketching press the checkmark



mixing_elbow | Part Studi...

https://cad.onshape.com/documents/4c69dd641fdd3ffa4d82f6a6/w/abaed15?87b3b05406abf6fe/e/7363f9be4e9d8b749b5cba84

Onshape mixing_elbow Main

Features (10)

Filter by name or type

- Default geometry
  - Origin
  - Top
  - Front
  - Right
- Sketch 1
- Sketch 2
- Sweep 1
- Plane 1
- Sketch 3
- Extrude 1

Parts (0)

Sketch 1

Sketch plane: Top plane

☐ Show constraints

☑ Show overdefined

Final

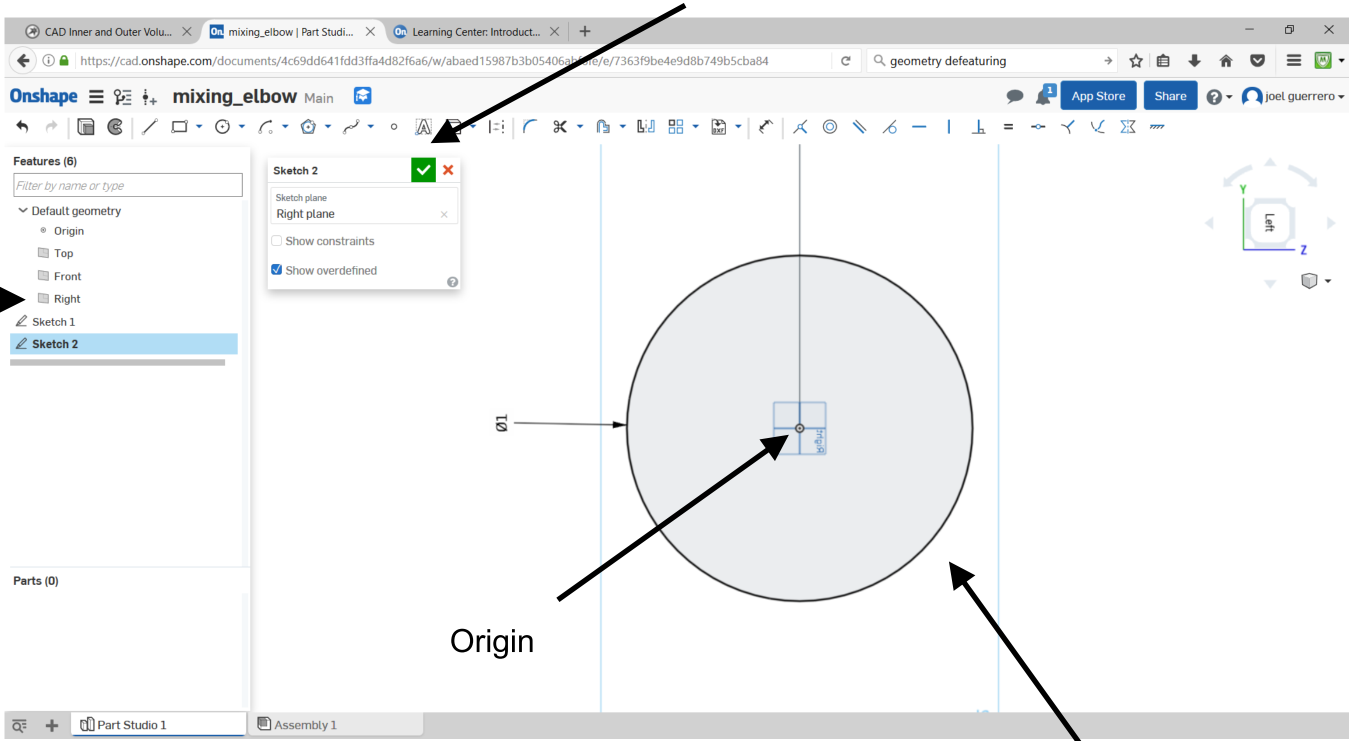
In sketch mode:

- Blue geometry is free to move.
- Black geometry is fully defined.
- Red geometry is over-constrained.

Use the dimensions illustrated to draw this polyline

Solid modeling using Onshape

- Select the right plane and start a new sketch.
- Draw a circle with the center in the origin (the white point).



Sketch 2

Sketch plane: Right plane

☐ Show constraints

☑ Show overdefined

When you are done sketching press the checkmark

Select this plane and start a new sketch

Origin

Use the dimensions illustrated to draw the circle

Solid modeling using Onshape

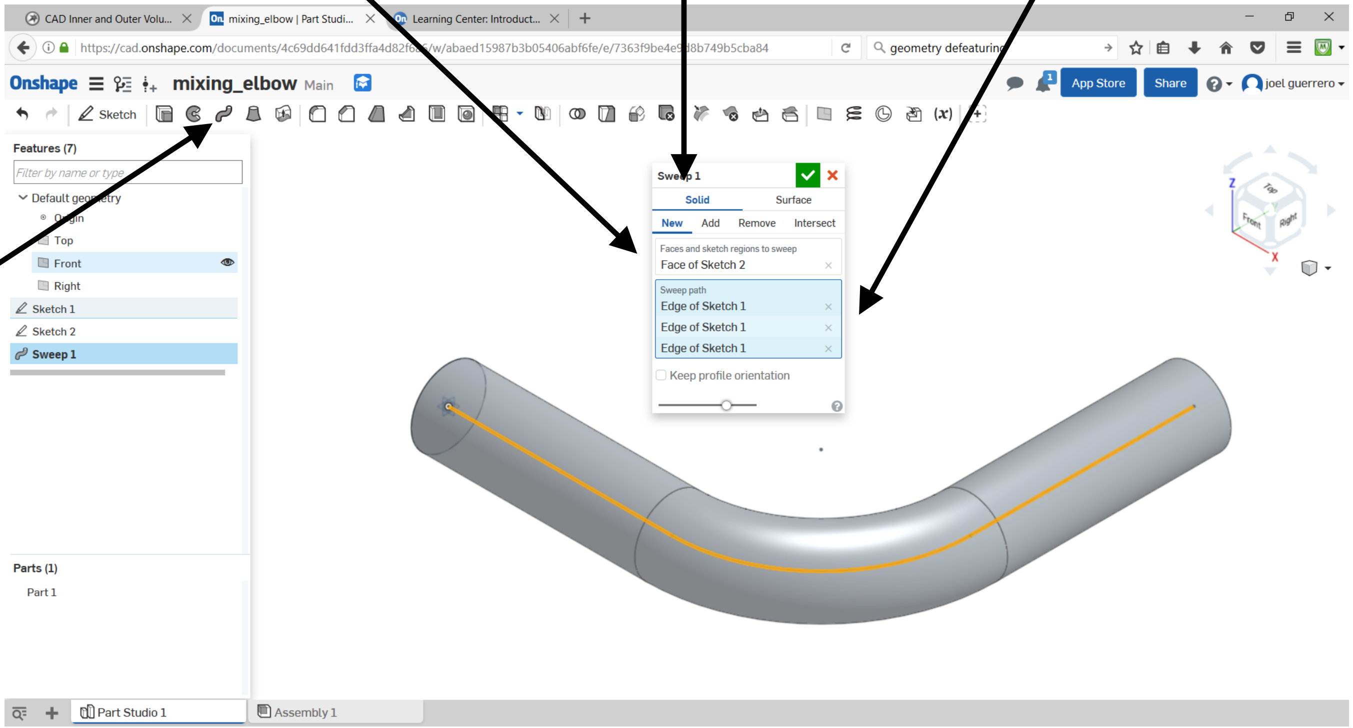
- Use the sweep feature to create a new solid.

Select the circle as the face to sweep

Select new solid

Select the lines as the sweep patch

Sweep feature



Sweep 1

Solid | Surface

New | Add | Remove | Intersect

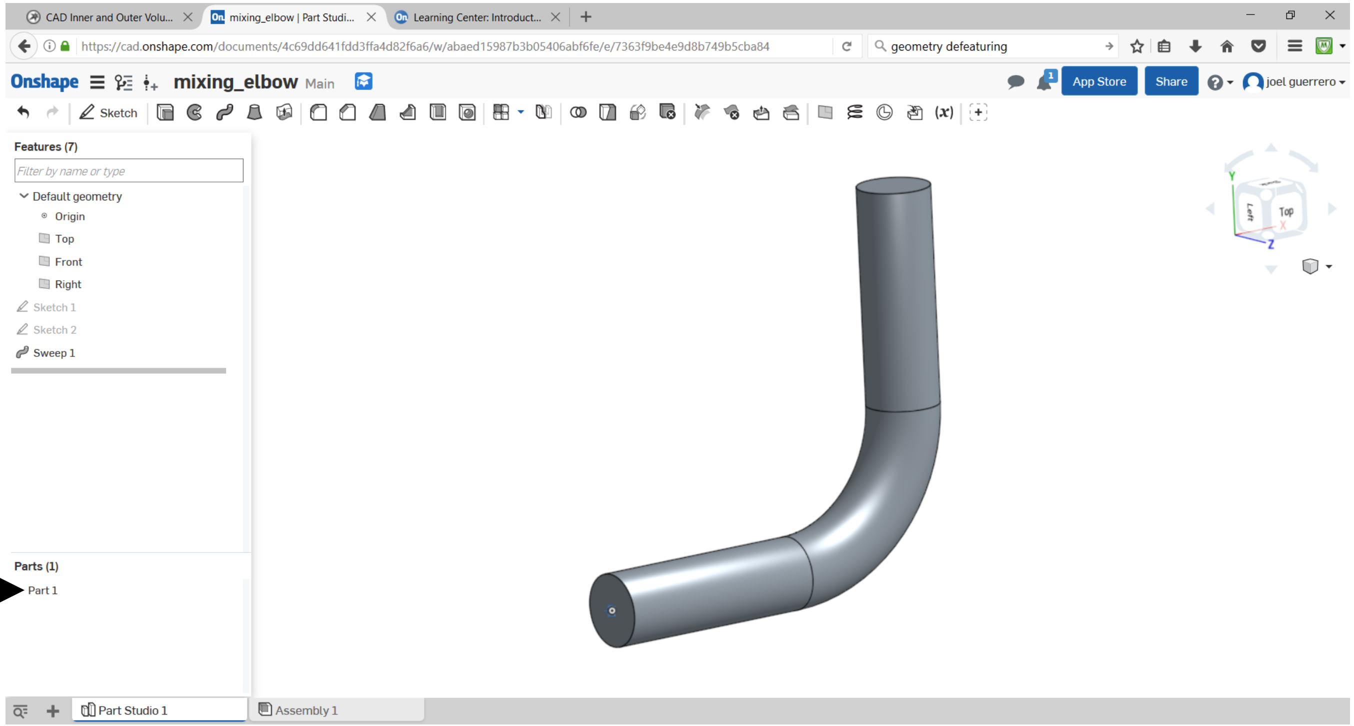
Faces and sketch regions to sweep: Face of Sketch 2

Sweep path: Edge of Sketch 1; Edge of Sketch 1; Edge of Sketch 1

□ Keep profile orientation

Solid modeling using Onshape

- At this point, you should have this solid.

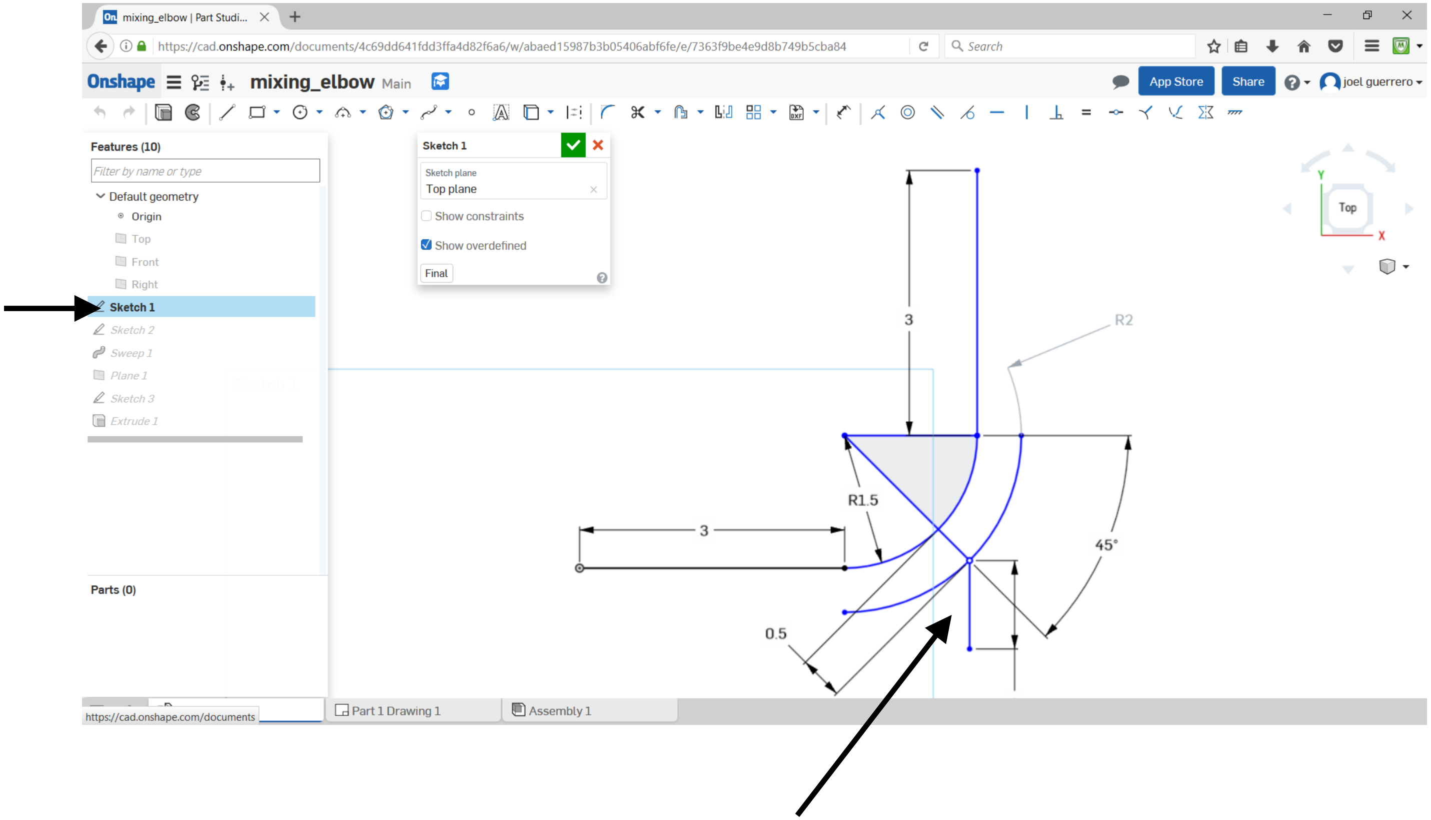


Solid name.
Right click to rename
or view the properties

Solid modeling using Onshape

- Let us add the new inlet to the pipe.
- Create a new sketch in the top plane or edit the initial sketch.
- Hereafter we will edit the initial sketch.

Right click and choose the option edit



Sketch 1

Sketch plane: Top plane

□ Show constraints

☑ Show overdefined

Sketch these new lines using the dimensions illustrated. Pay attention to the angle and the offset distance.

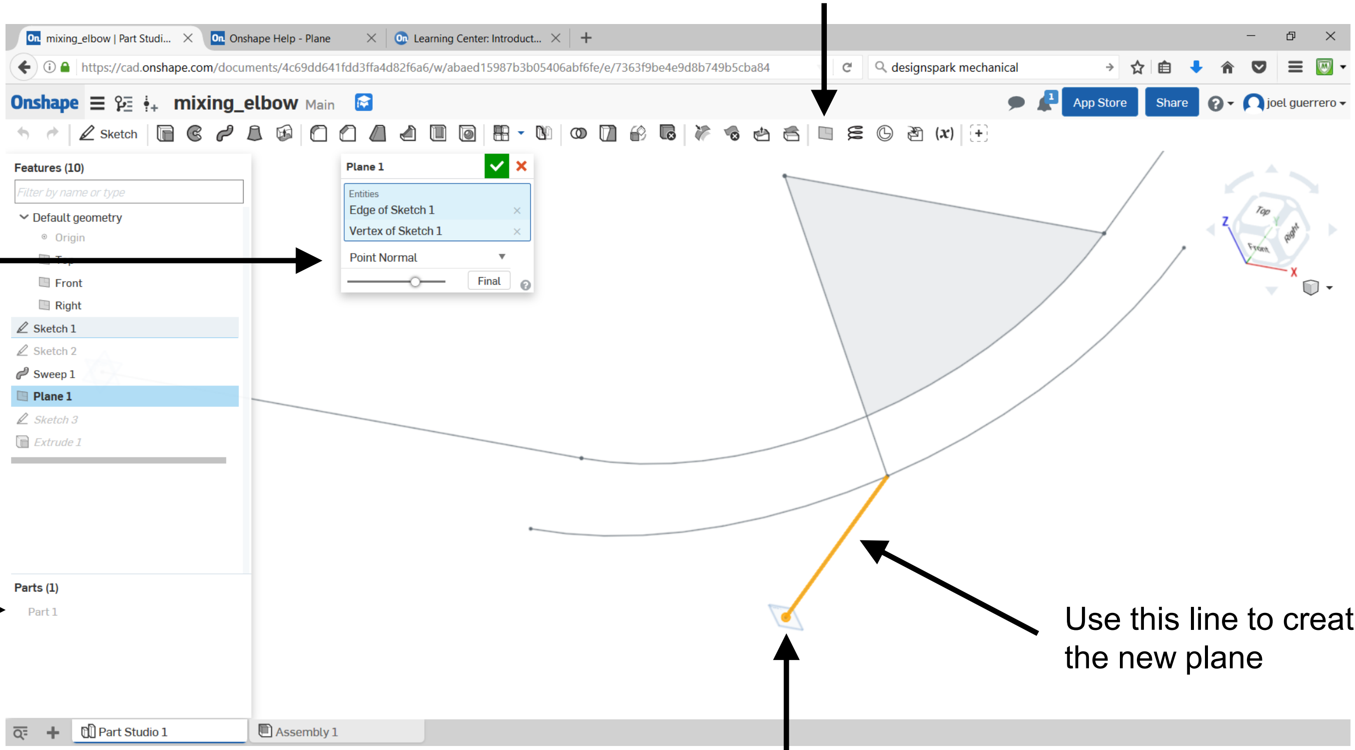
Solid modeling using Onshape

- Create a plane normal to a line and passing through a point

1. Create new plane

2. Select point normal, and select the line and point as illustrated

To get better visibility, you can hide the solid or adjust the transparency



Plane 1

Entities
Edge of Sketch 1
Vertex of Sketch 1

Point Normal

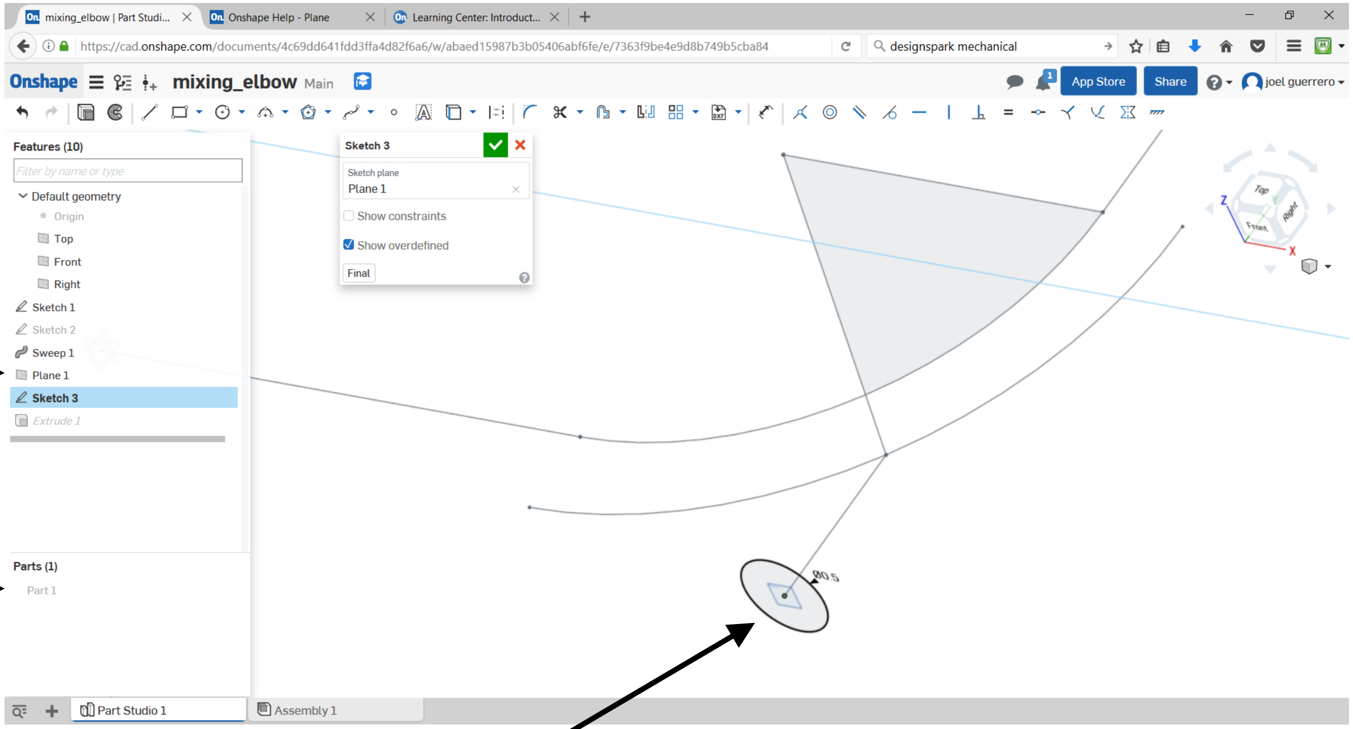
Final

Use this line to create the new plane

Use this point to create the new plane

Solid modeling using Onshape

- Sketch a circle in the newly created plane.



New plane

To get better visibility, you can hide the solid or adjust the transparency

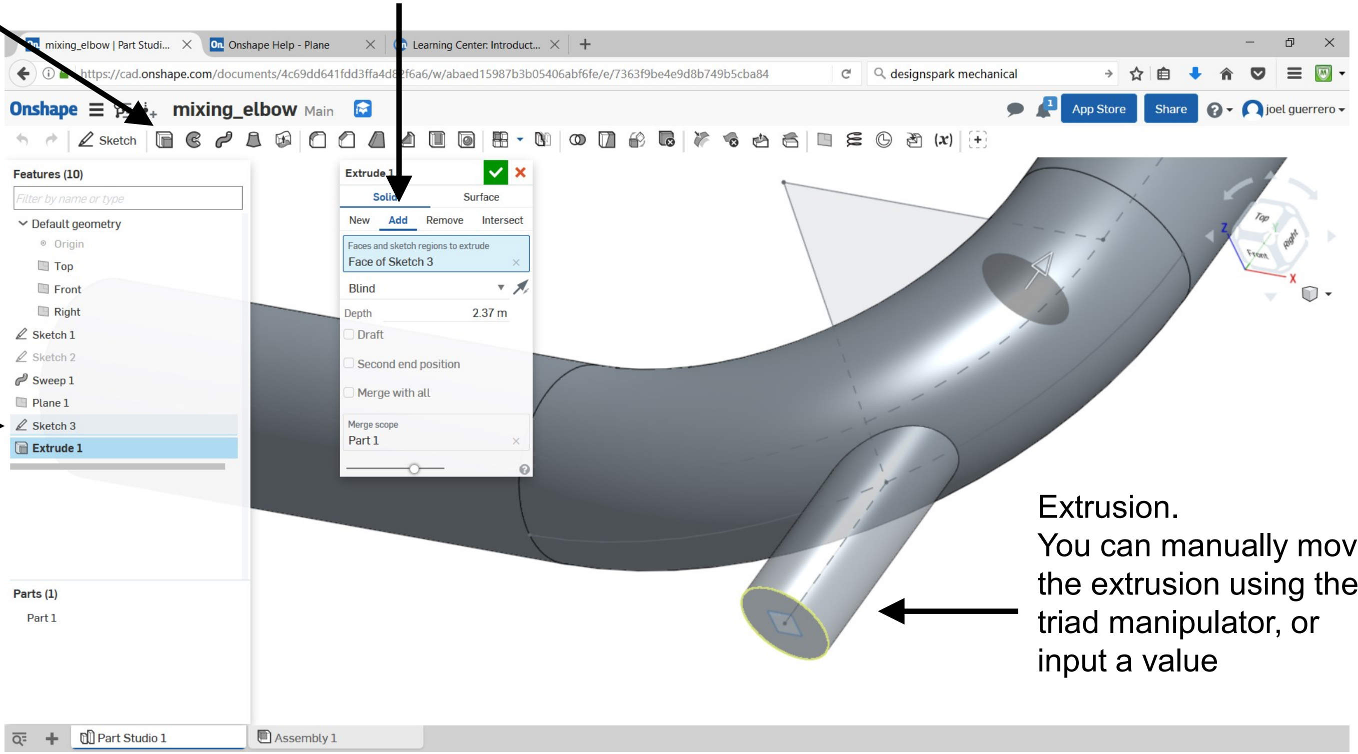
Sketch this circle in the newly created plane

Solid modeling using Onshape

- Using the feature extrude to create a new solid using the previous sketch.
- Extrude the circle until in intercept the solid.

Add the new solid to the previous part (boolean operation)

Feature extrusion



mixing_elbow | Part Studi...
Onshape Help - Plane
Learning Center: Introduct...
https://cad.onshape.com/documents/4c69dd641fdd3ffa4d8f6a6/w/abaed15987b3b05406abf6fe/e/7363f9be4e9d8b749b5cba84
designspark mechanical
Onshape
mixing_elbow Main
App Store
Share
joel guerrero
Sketch
Part Studio 1
Assembly 1
Top
Front
Right

Features (10)

Filter by name or type

- Default geometry
  - Origin
  - Top
  - Front
  - Right
- Sketch 1
- Sketch 2
- Sweep 1
- Plane 1
- Sketch 3
- Extrude 1

Parts (1)

Part 1

Extrude 1

Solid | Surface

New | Add | Remove | Intersect

Faces and sketch regions to extrude: Face of Sketch 3

Blind

Depth 2.37 m

□ Draft

□ Second end position

□ Merge with all

Merge scope: Part 1

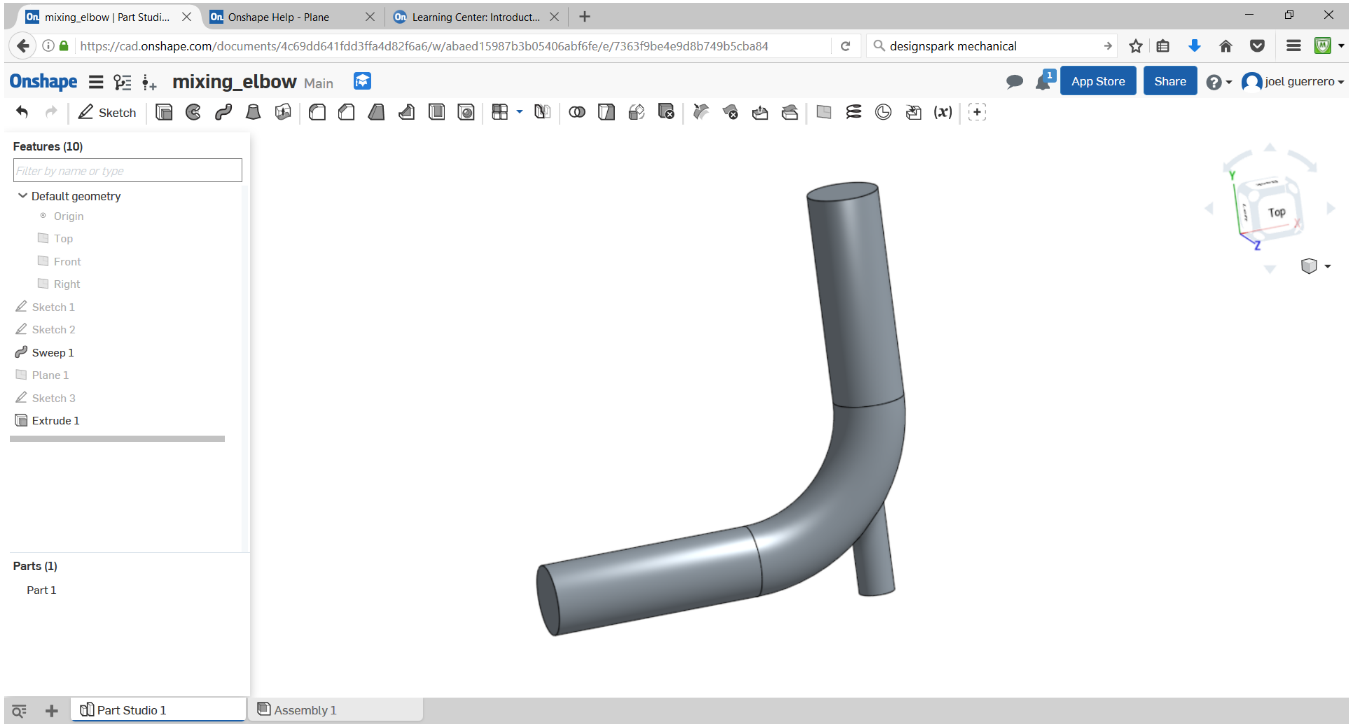
Use this sketch as the base for the extrusion

Extrusion.
You can manually move the extrusion using the triad manipulator, or input a value

Instead of the extrusion feature, you could use the sweep feature. You will need to create a longer sweep path.

Solid modeling using Onshape

- At this point you should have the following solid.



mixing_elbow | Part Studi...
Onshape Help - Plane
Learning Center: Introduct...
https://cad.onshape.com/documents/4c69dd641fdd3ffa4d82f6a6/w/abaed15987b3b05406abf6fe/e/7363f9be4e9d8b749b5cba84
designspark mechanical
Onshape
mixing_elbow Main
App Store
Share
joel guerrero
Sketch
Top
Part Studio 1
Assembly 1

Features (10)

Filter by name or type

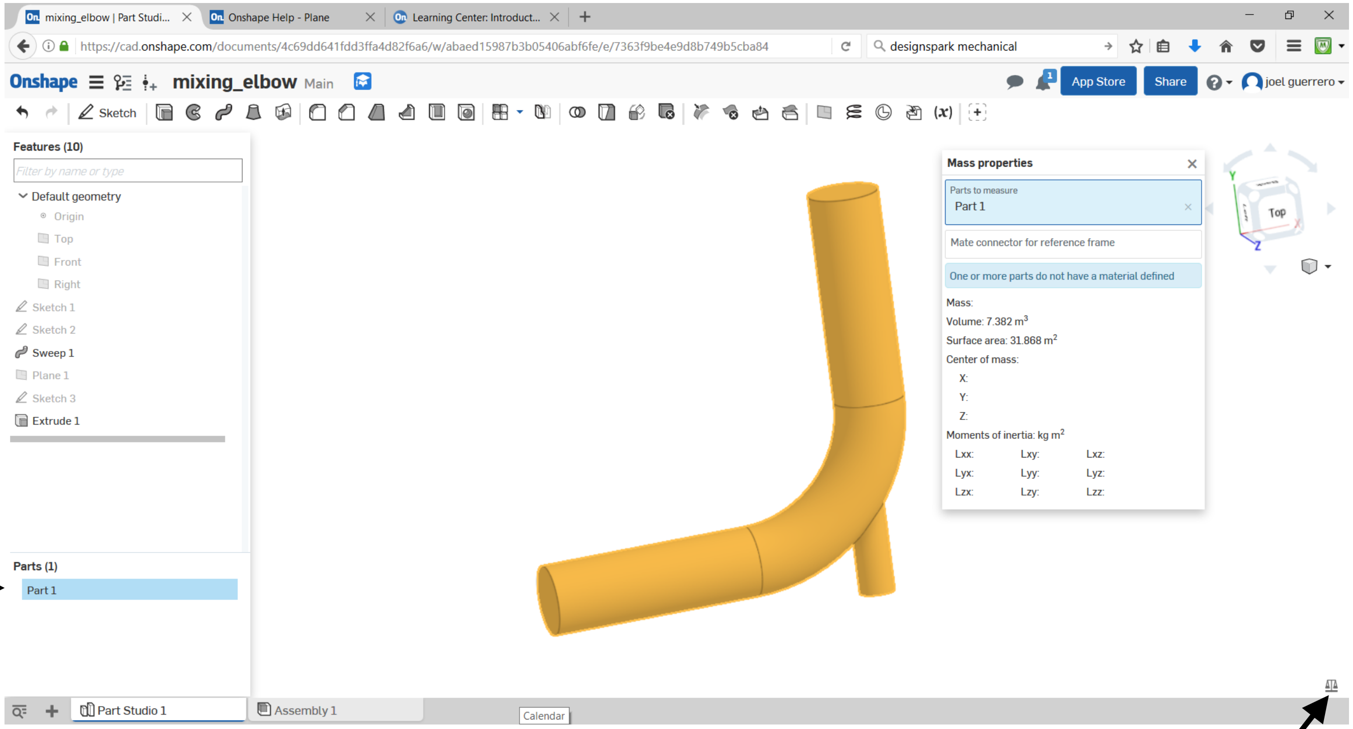
Default geometry
Origin
Top
Front
Right
Sketch 1
Sketch 2
Sweep 1
Plane 1
Sketch 3
Extrude 1

Parts (1)

Part 1

Solid modeling using Onshape

- If you want to know the mass properties of the solid, select it, and then click on the mass properties icon.
- To get the inertia, you will need to assign a material.



Mass properties

Parts to measure
Part 1

Mate connector for reference frame

One or more parts do not have a material defined

Mass:

Volume: 7.382 m^3

Surface area: 31.868 m^2

Center of mass:

X:

Y:

Z:

Moments of inertia: kg m^2

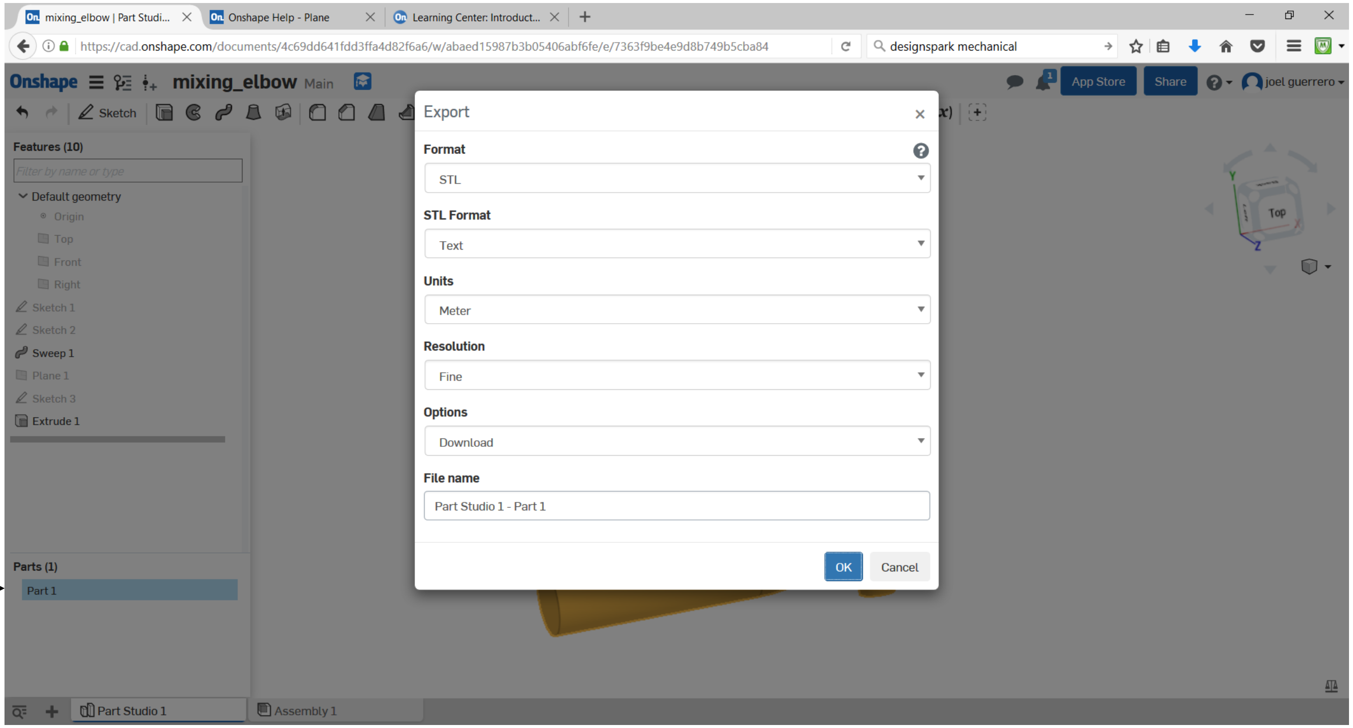
Lxx:	Lxy:	Lxz:
Lyx:	Lyy:	Lyz:
Lzx:	Lzy:	Lzz:

Select the part.
Right click and select assign material.

Mass properties icon

Solid modeling using Onshape

- To export the solid model, right click on the part name and select the option export.
- Choose the desired format. In this case choose STL.



Export

Format: STL

STL Format: Text

Units: Meter

Resolution: Fine

Options: Download

File name: Part Studio 1 - Part 1

Right click and select the option export.

Solid modeling using Onshape

- Parametric modeling and feature-based modeling are two of the most powerful tools available in any CAD/solid modeling applications.
- They are crucial components in the design experience, especially when dealing with design intent.
- Experimenting with dimension schemes is one of the best ways to improve your understanding of design intent.

- To learn more about Onshape, you can visit their learning center:

  https://learn.onshape.com/

- Finally, feel free to visit our youtube channel where you will find a few solid modeling videos in the context of CFD and OpenFOAM® :

  https://www.youtube.com/channel/UCNNBm3KxVS1rGeCVUU1p61g

Solid modeling using Onshape

- At the following links, you can find a few geometries that you can use to setup cases from scratch:

  - Sailing boat:
https://cad.onshape.com/documents/ad885ed6298e6d95e372f573/w/1cfda457fe3ad410332aad9c/e/8dac4fcaf6e34a43e9676fbc

  - Mixing elbow:
https://cad.onshape.com/documents/1cc919d8e75c2e47e8c1d50e/w/0efa002648eb2fb80ec4bec4/e/a742bf4113c626735e1d8f1a

  - Static mixer:
https://cad.onshape.com/documents/58f7930861743e1074559ea6/w/96672317c9167265f9d10181/e/e4b6b1baffa90ca207afe974

  - Ahmed body:
https://cad.onshape.com/documents/e1ecbacd95be9ed0962aa410/w/f0295899197e2f3d851000fd/e/aa40f8f5d26b7117dd0a5111

  - Mixing tank:
https://cad.onshape.com/documents/e00307c191ce168d1d8c2e05/w/fc5d69b18559ec3893a1a80a/e/ba4b5ca5b34ad7335a2915a3

  - Onera M6 wing:
https://cad.onshape.com/documents/e176caaa70bfd4719cafe3d7/w/9d8b5771d7382000b0762e65/e/ec4669f8c28761e60e35b32f

  - Three element airfoil:
https://cad.onshape.com/documents/590a4195a6145a1089cfb96f/w/838a3095da90d5dd66a3150e/e/5d65878bed975a1a94102846

Module 3

Meshing preliminaries – Mesh quality assessment – Meshing in OpenFOAM®

Before we begin

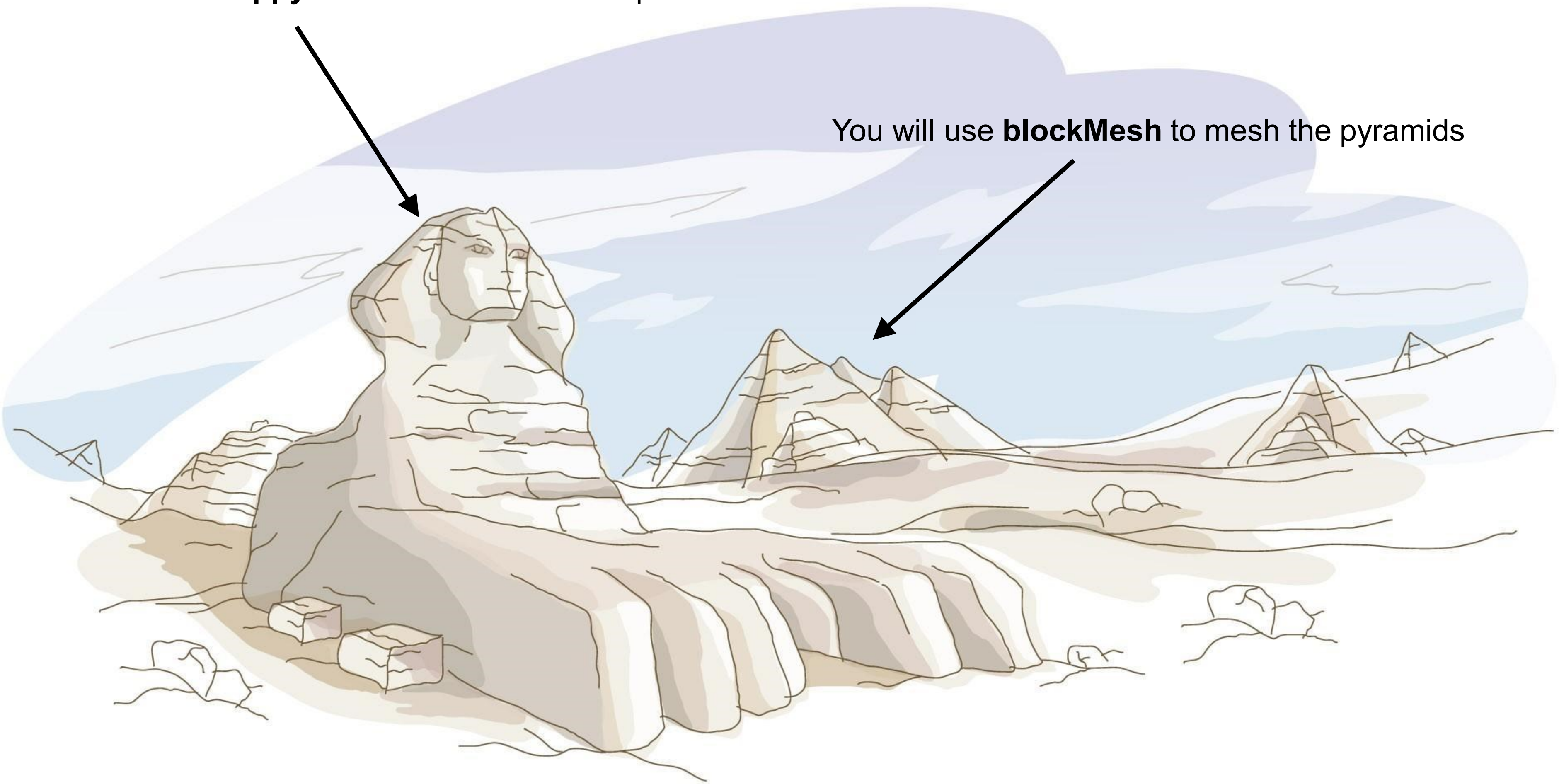
- OpenFOAM® comes with the following meshing applications:
  - `blockMesh`
  - `snappyHexMesh`
  - `foamyHexMesh`
  - `foamyQuadMesh`
- We are going to work with `blockMesh` and `snappyHexMesh`.
- **blockMesh** is a multi-block mesh generator.
- **snappyHexMesh** is an automatic split hex mesher, refines and snaps to surface.
- If you are not comfortable using OpenFOAM® meshing applications, you can use an external mesher.
- OpenFOAM® comes with many mesh conversion utilities. Many popular meshing formats are supported. To name a few: gambit, cfx, fluent, gmsh, ideas, netgen, plot3d, starccm, VTK.
- In this module, we are going to address how to mesh using OpenFOAM® technology, how to convert meshes to OpenFOAM® format, and how to assess mesh quality in OpenFOAM®.

Before we begin

By the end of this module, you will realize that

You will use **snappyHexMesh** to mesh the sphinx

You will use **blockMesh** to mesh the pyramids

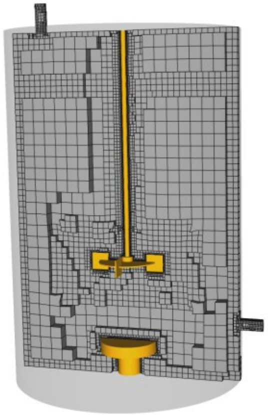


Roadmap

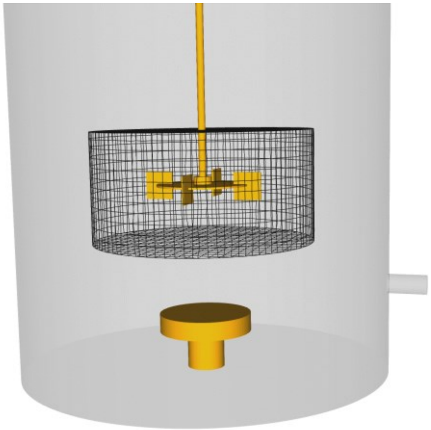
1. **Meshing preliminaries**
2. What is a good mesh?
3. Mesh quality assessment in OpenFOAM®
4. Mesh generation using blockMesh.
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

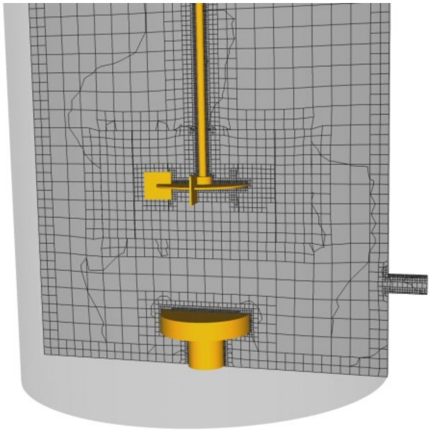
Meshing preliminaries

- Mesh generation or domain discretization consist in dividing the physical domain into a finite number of discrete regions, called control volumes or cells in which the solution is sought



🎞





www.wolfdynamics.com/wiki/moving/ani1.gif

www.wolfdynamics.com/wiki/moving/ani2.gif

Meshing preliminaries

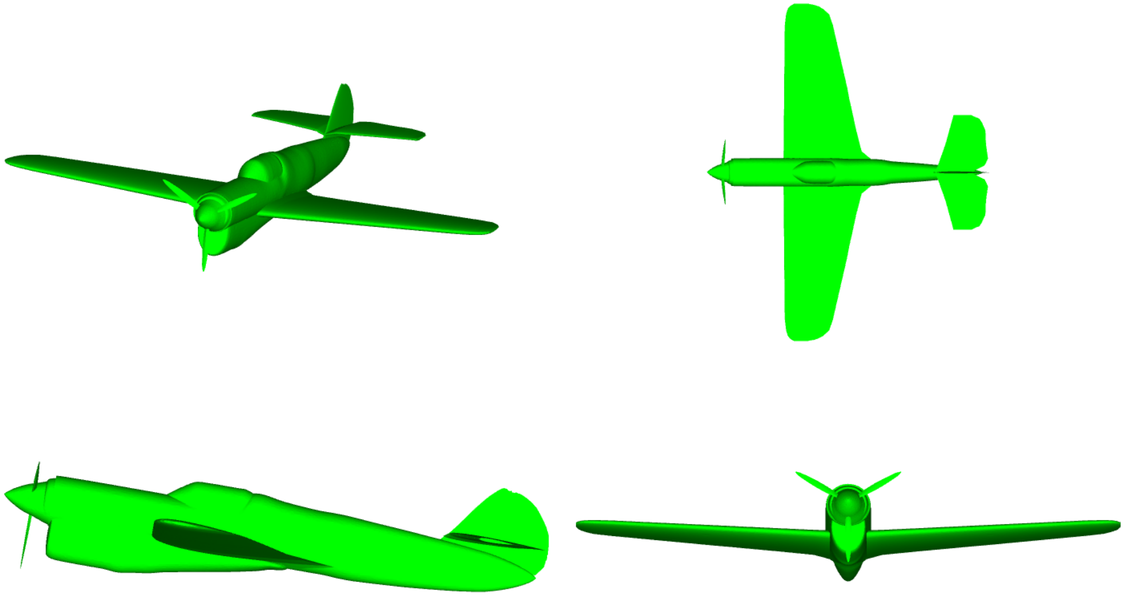
Mesh generation process

- Generally speaking, when generating the mesh we follow these three simple steps:
  - **Geometry generation:** we first generate the geometry that we are going to feed into the meshing tool.
  - **Mesh generation:** the mesh can be internal or external. We also define surface and volume refinement regions. We can also add inflation layers to better resolve the boundary layer. During the mesh generation process we also check the mesh quality.
  - **Definition of boundary surfaces:** in this step we define physical surfaces where we are going to apply the boundary conditions. If you do not define these individual surfaces, you will have one single surface and it will not be possible to apply different boundary conditions.

Meshing preliminaries

Geometry generation - Input geometry

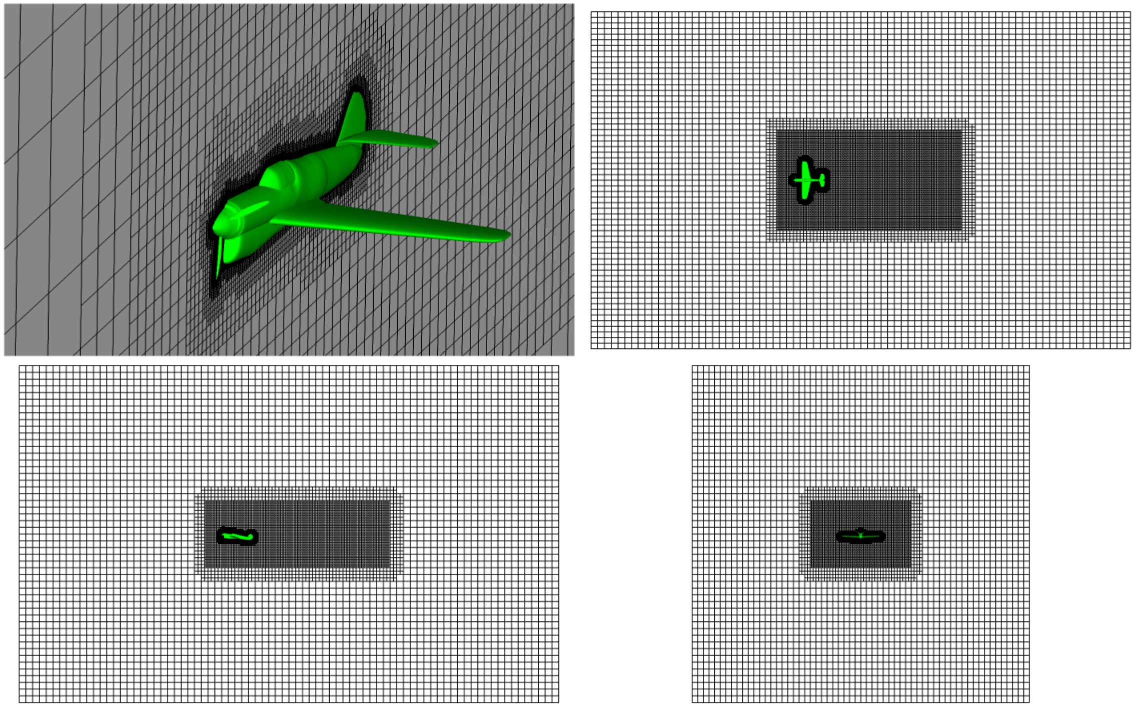
- The geometry must be watertight.
- Remember, the quality of the mesh and hence the quality of the solution greatly depends on the geometry. So always do your best when creating the geometry.



Meshing preliminaries

Mesh generation

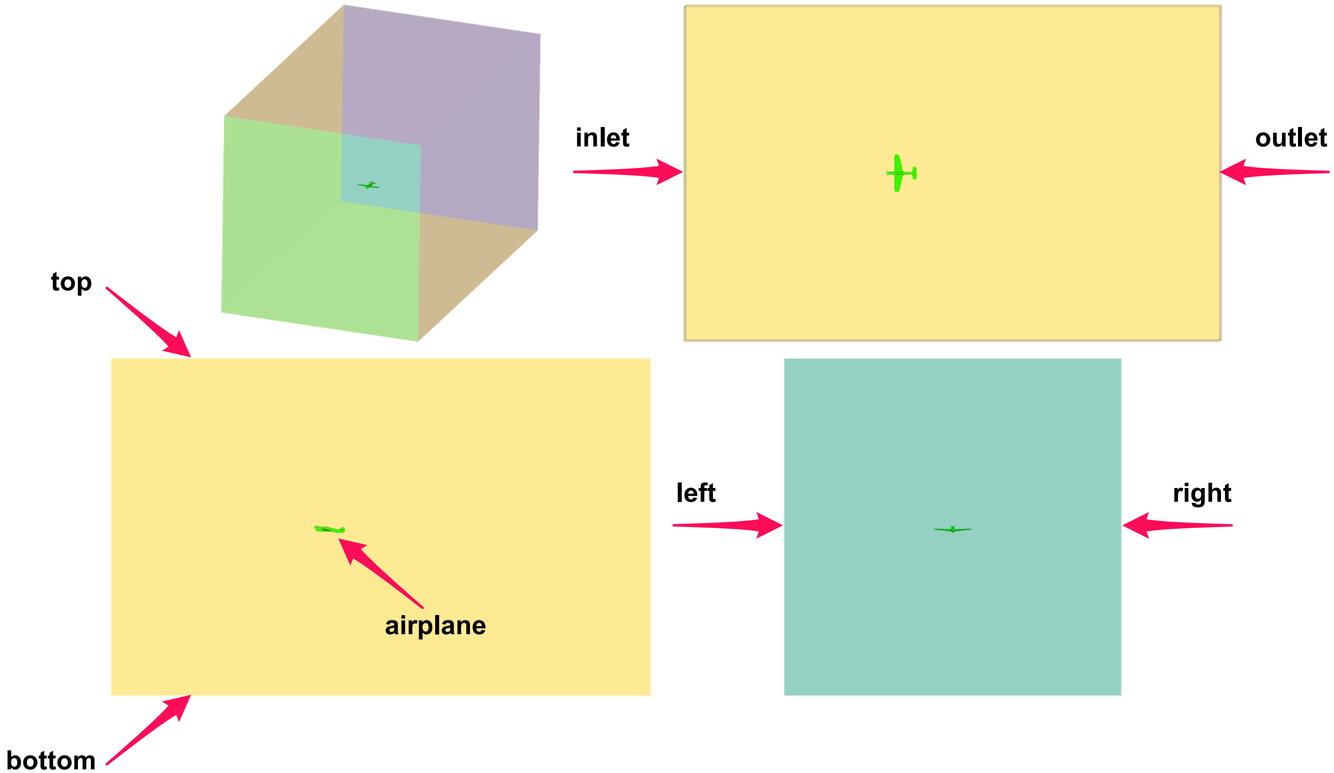
- If we are interested in external aerodynamics, we define a physical domain and we mesh the region around the body.
- If we are interested in internal aerodynamics, we simply mesh the internal volume of the geometry.
- To resolve better the flow features, we can add surface and volume refinement.
- Remember to always check the mesh quality.



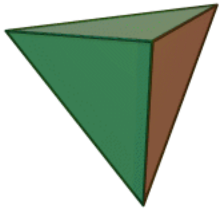
Meshing preliminaries

Definition of boundary surfaces (patches)

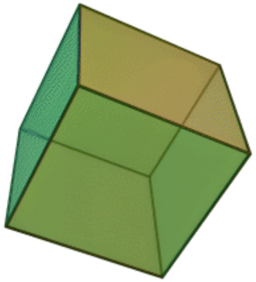
- In order to assign boundary conditions, we need to create boundary surfaces (patches) where we are going to apply the boundary values.
- The boundary surfaces (patches) are created at meshing time.
- In OpenFOAM®, you will find this information in the `boundary` dictionary file which is located in the directory **`constant/polyMesh`**. This dictionary is created automatically at meshing time.



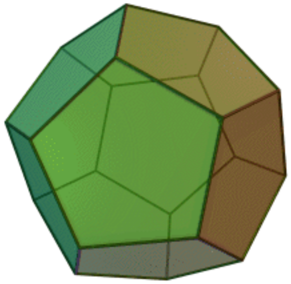
What cell type should I use?



http://www.wolfdynamics.com/wiki/cells/ani_tetra.gif



http://www.wolfdynamics.com/wiki/cells/ani_hexa.gif



http://www.wolfdynamics.com/wiki/cells/ani_poly.gif

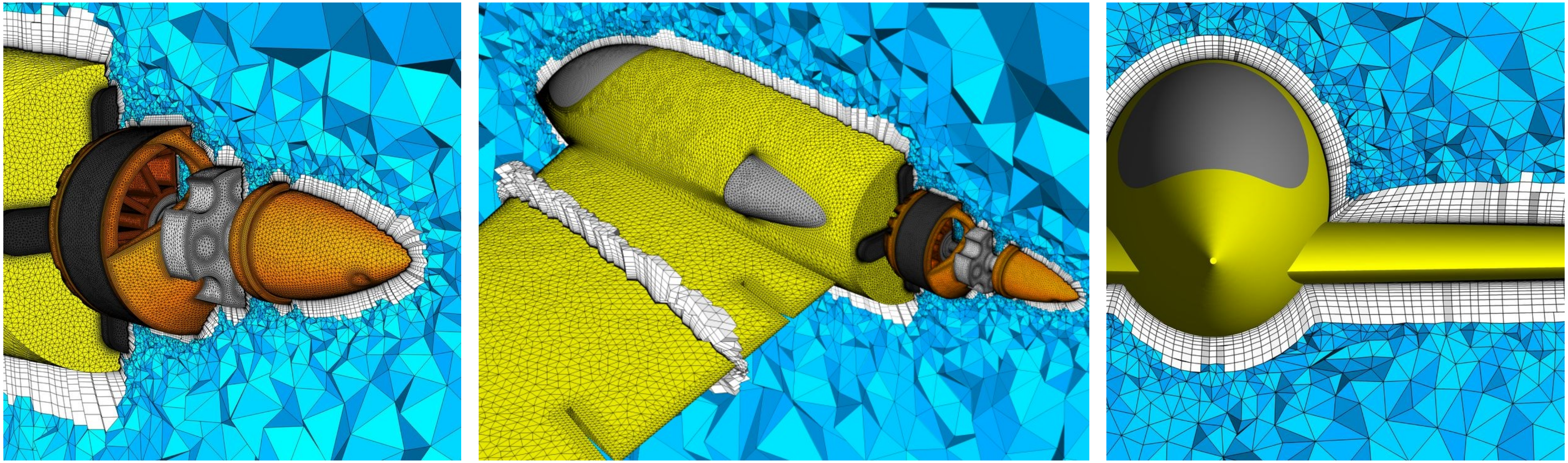
- In the meshing world, there are many cell types. Just to name a few: tetrahedrons, pyramids, hexahedrons, prisms, polyhedral.
- Each cell type has its very own properties when it comes to approximate the gradients and fluxes, we are going to talk about this later on when we deal with the FVM.
- Generally speaking, hexahedral cells will give more accurate solutions under certain conditions.
- However, this does not mean that tetra/poly cells are not good.
- What cell type do I use? It is up to you; at the end of the day the overall quality of the final mesh should be acceptable, and your mesh should resolve the physics

Roadmap

1. ~~Meshing preliminaries~~
2. **What is a good mesh?**
3. Mesh quality assessment in OpenFOAM®
4. Mesh generation using blockMesh.
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

What is a good mesh?

- There is no written theory when it comes to mesh generation.
- Basically, the whole process depends on user experience and good standard practices.
- A standard rule of thumb is that the elements shape and distribution should be pleasing to the eye.



22rd IMR Meshing Maestro Contest Winner
Travis Carrigan, John Chawner and Carolyn Woeber. Pointwise.
http://imr.sandia.gov/22imr/MeshingContest.html

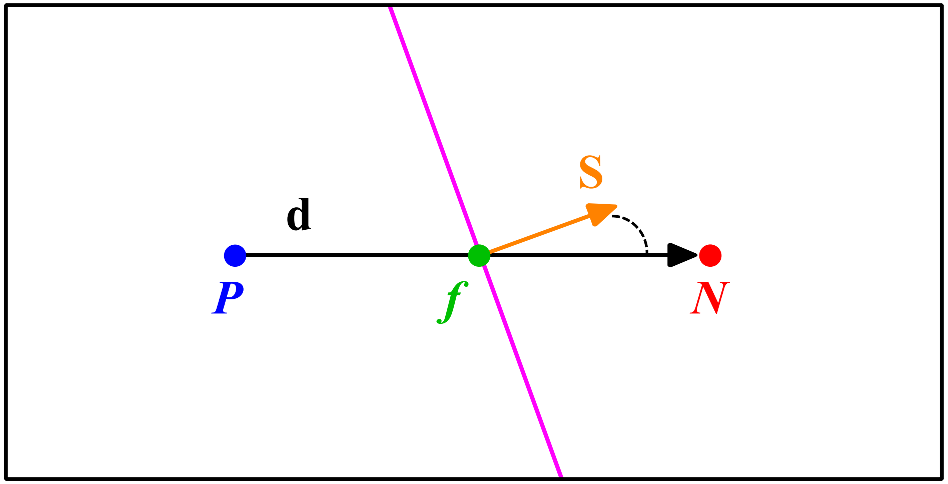
What is a good mesh?

- In a more sounded way, the user can rely in mesh metrics.
- However, no single standard benchmark or metric exists that can effectively assess the quality of a mesh, but the user can rely on suggested best practices.
- Hereafter, we will present the most common mesh quality metrics:
  - Orthogonality.
  - Skewness.
  - Aspect Ratio.
  - Smoothness.
- After generating the mesh, we measure these quality metrics to assess the mesh quality.
- Have in mind that there are many more mesh quality metrics out there, and some of them are not very easy to interpret (*e.g.*, jacobian matrix, determinant, flatness, equivalence, condition number, and so on).
- <u>It seems that it is much easier diagnosing bad meshes than good meshes.</u>

What is a good mesh?

Mesh quality metrics. Mesh orthogonality

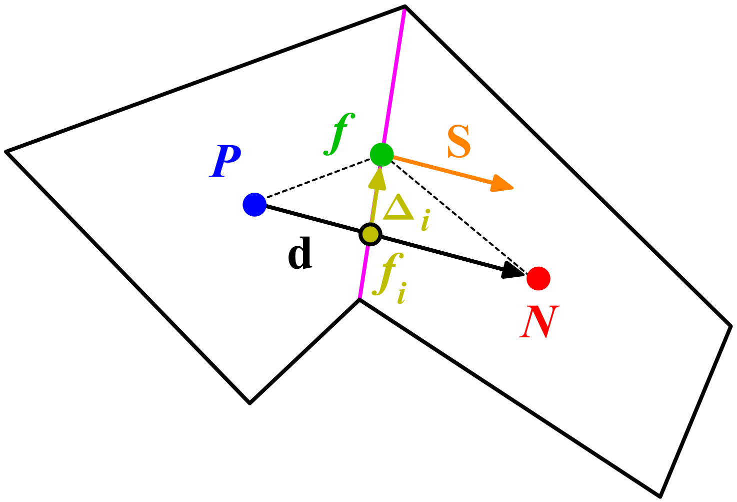
- Mesh orthogonality is the angular deviation of the vector **S** (located at the face center f) from the vector **d** connecting the two cell centers **P** and **N**. In this case is 20°.
- It mainly affects the Laplacian and gradient terms at the face center f.
- It adds numerical diffusion to the solution.



What is a good mesh?

Mesh quality metrics. Mesh skewness

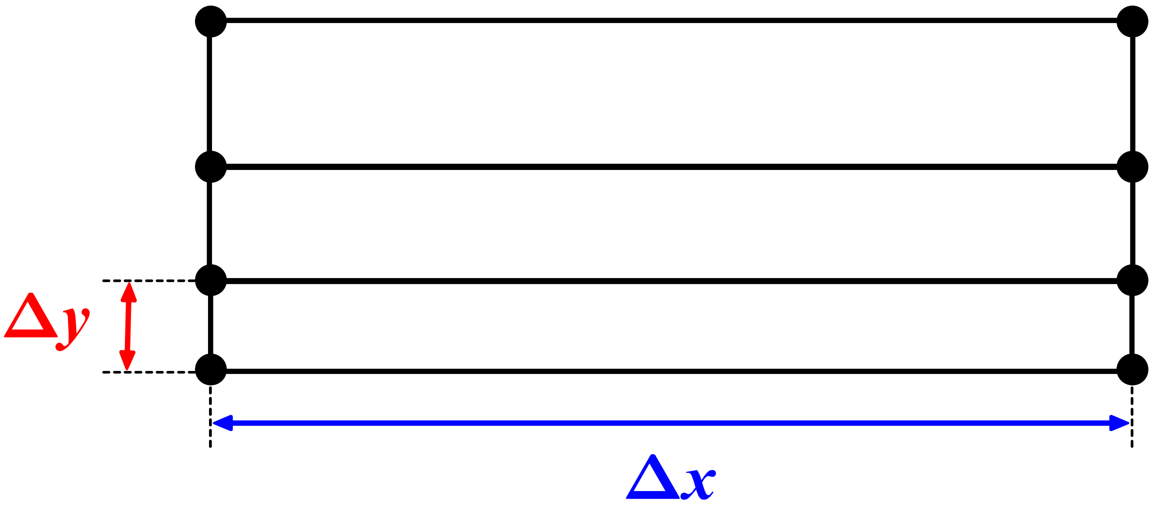
- Skewness (also known as non-conjunctionality) is the deviation of the vector **d** that connects the two cells **P** and **N**, from the face center f.
- The deviation vector is represented with Δ and f_i is the point where the vector **d** intersects the face f.
- It affects the interpolation of the cell centered quantities at the face center f.
- It affects the computation of the convective, diffusive, and gradient terms.
- It adds numerical diffusion and wiggles to the solution.



What is a good mesh?

Mesh quality metrics. Mesh aspect ratio AR

- Mesh aspect ratio AR is the ratio between the longest side Δx and the shortest side Δy .
- Large AR are ok if gradients in the largest direction are small.
- High AR smear gradients.
- Large AR can add numerical diffusion to the solution.

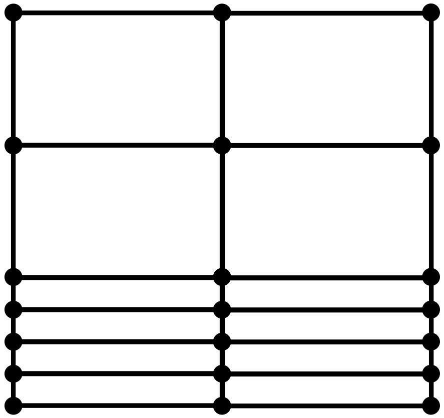


What is a good mesh?

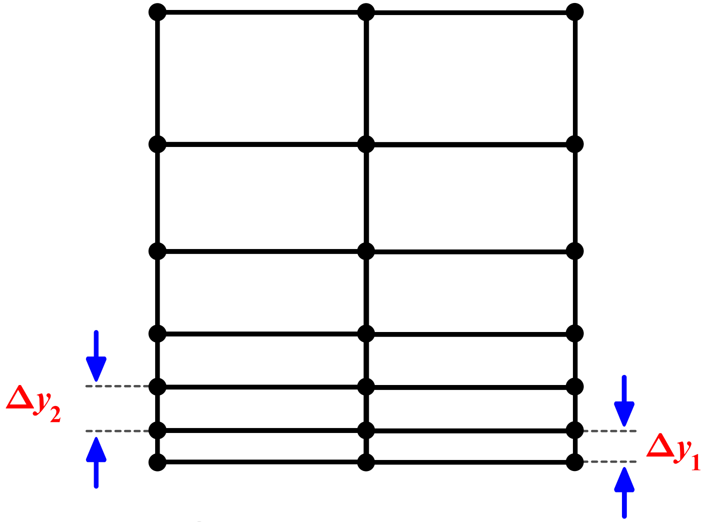
Mesh quality metrics. Smoothness

- Smoothness, also known as expansion rate, growth factor or uniformity, defines the transition in size between contiguous cells.
- Large transition ratios between cells add diffusion to the solution.
- Ideally, the maximum change in mesh spacing should be less than 20%:

$$\frac{\Delta y_2}{\Delta y_1} \leq 1.2$$



Steep transition

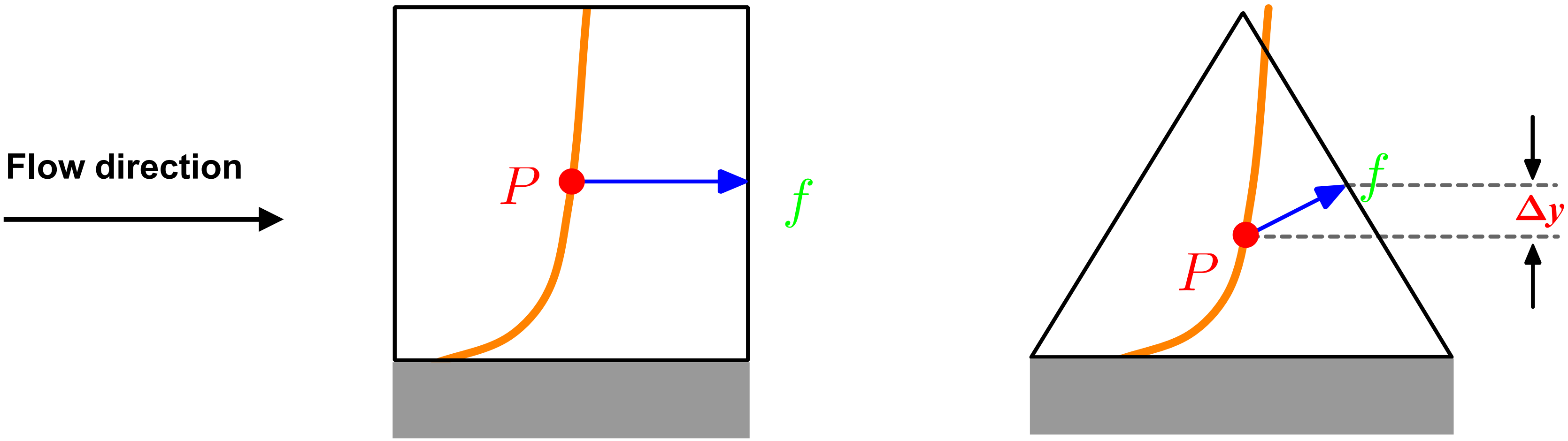


Smooth transition

What is a good mesh?

Mesh quality metrics. <u>Element type close to the walls - Cell/Flow alignment</u>

- Hexes, prisms, and quadrilaterals can be stretched easily to resolve boundary layers without losing quality.
- Triangular and tetrahedral meshes have inherently larger truncation error.
- Less truncation error when faces aligned with flow direction and gradients.



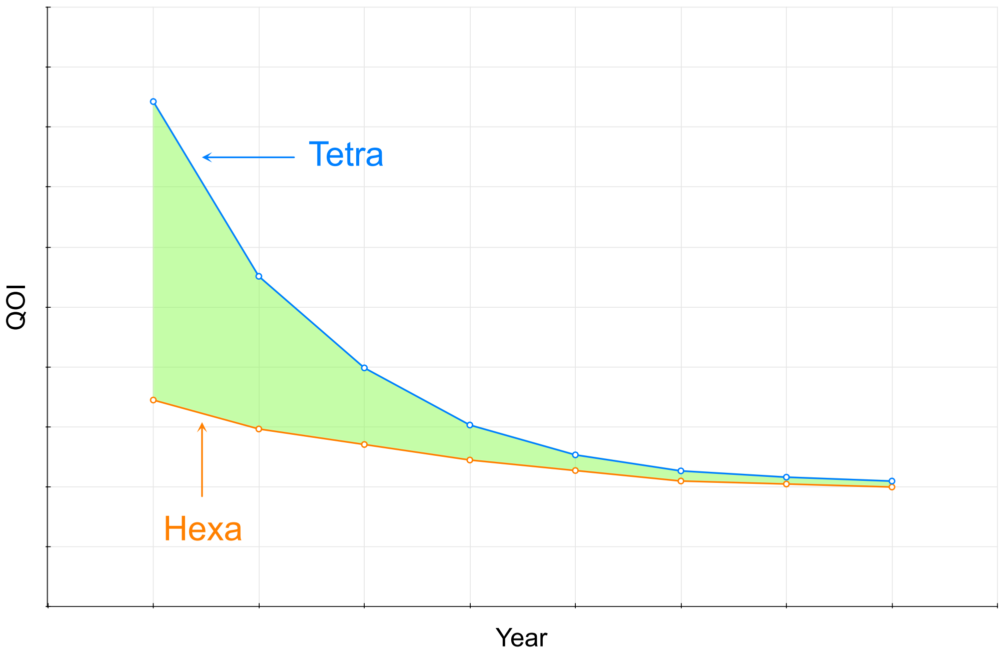
$$\phi_f = \phi_P + \cancel{\frac{\partial \phi}{\partial y}\Delta y} + \cancel{\mathcal{O}(\Delta y^2)}$$

$$\phi_f = \phi_P + \frac{\partial \phi}{\partial y}\Delta y + \mathcal{O}(\Delta y^2)$$

What is a good mesh?

Striving for quality

- For the same cell count, hexahedral meshes will give more accurate solutions, especially if the grid lines are aligned with the flow.
- But this does not mean that tetrahedral meshes are not good, by carefully choosing the numerical scheme you can get the same level of accuracy as in hexahedral meshes.
- The problem with tetrahedral meshes is mainly related to the way gradients are computed.

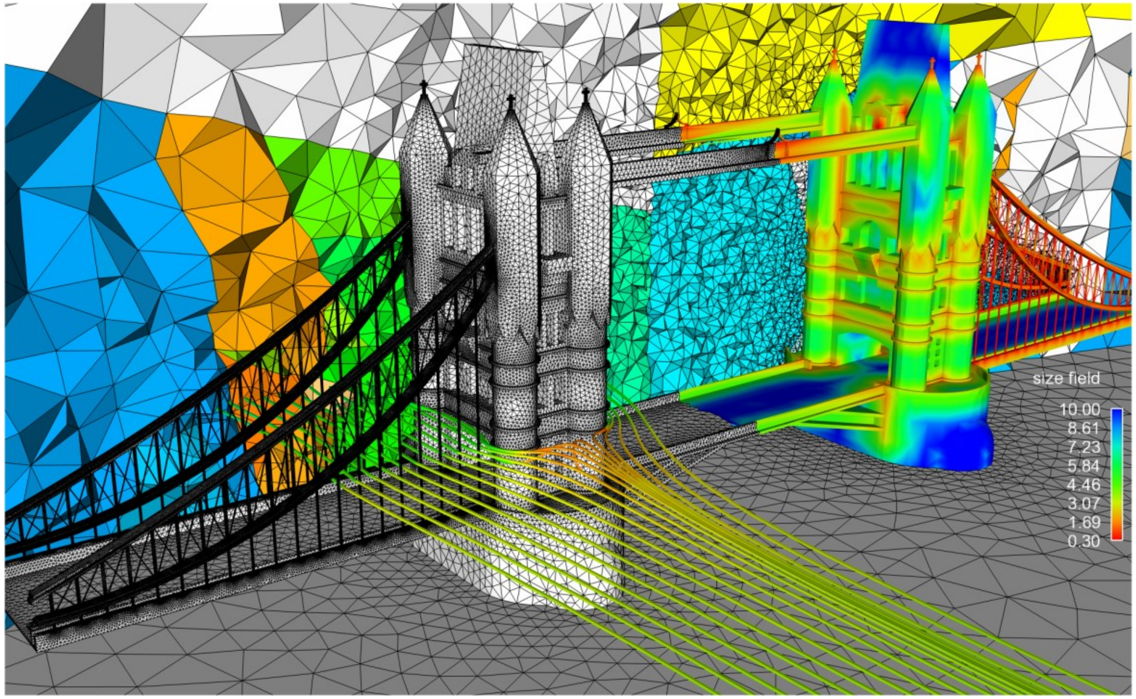


- In the early years of CFD, there was a huge gap between the outcome of tetra and hex meshes.
- But with time and thanks to developments in numerical methods and computer science (software and hardware), today all cell types give the same results.

What is a good mesh?

Striving for quality

- The mesh density should be high enough to capture all relevant flow features.
- In areas where the solution change slowly, you can use larger elements.
- A good mesh does not rely in the fact that the more cells we use the better the solution.



23rd IMR Meshing Maestro Contest Winner
Zhoufang Xiao , Jianjing Zheng, Dawei Zhao, Lijuan Zeng, Jianjun Chen, Yao Zheng
Center for Engineering & Scientific Computation, Zhejiang University, China.
http://www.sandia.gov/imr/MeshingContest.html

What is a good mesh?

Striving for quality

- Hexes, prisms, and quadrilaterals can be easily aligned with the flow.
- They can also be stretched to resolve boundary layers without losing much quality.
- Triangular and tetrahedral meshes can easily be adapted to any kind of geometry. The mesh generation process is almost automatic.
- Tetrahedral meshes normally need more computing resources during the solution stage. But this can be easily offset by the time saved during the mesh generation stage.
- Increasing the cells count will likely improve the solution accuracy, but at the cost of a higher computational cost. However, a finer mesh does not mean a better mesh.
- To keep cell count low, use non-uniform meshes to cluster cells only where they are needed. Use local refinements and solution adaption to further refine only on selected areas.
- In boundary layers, quads, hexes, and prisms/wedges cells are preferred over triangles, tetrahedrons, or pyramids.
- If you are not using wall functions (turbulence modeling), the mesh next to the walls should be fine enough to resolve the boundary layer flow. Have in mind that this will rocket the cell count and increase the computing time.

What is a good mesh?

Striving for quality

- Use hexahedral meshes whenever is possible, specially if high accuracy in predicting forces is your goal (drag prediction) or for turbo machinery applications.
- For complex flows without dominant flow direction, quad and hex meshes loose their advantages.
- Keep orthogonality, skewness, and aspect ratio to a minimum.
- Change in cell size should be smooth.
- Always check the mesh quality. Remember, one single cell can cause divergence or give you inaccurate results.
- When you strive for quality, you avoid the GIGO syndrome (garbage in, garbage out).
- Just to end for good the mesh quality talk:
  - A good mesh is a mesh that serves your project objectives.
  - So, as long as your results are physically realistic, reliable and accurate; your mesh is good.
  - Know your physics and generate a mesh able to resolve the physics involve, without over-doing.

What is a good mesh?

A good mesh might not lead to the ideal solution, but a bad mesh will always lead to a bad solution.

P. Baker – Pointwise

Who owns the mesh, owns the solution.

H. Jasak – Wikki Ltd.

Avoid the GIGO syndrome (Garbage In – Garbage Out).
As I am a really positive guy I prefer to say,
good mesh – good results.

J. G. – WD

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. **Mesh quality assessment in OpenFOAM®**
4. Mesh generation using blockMesh.
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

Mesh quality metrics in OpenFOAM®

- In the file *primitiveMeshCheck.C* located in the directory **$WM_PROJECT_DIR/src/OpenFOAM/meshes/primitiveMesh/primitiveMeshCheck/** you will find the quality metrics hardwired in OpenFOAM®. Their maximum (or minimum) values are defined as follows:

```
Foam::scalar Foam::primitiveMesh::closedThreshold_  = 1.0e-6;
Foam::scalar Foam::primitiveMesh::aspectThreshold_  = 1000;
Foam::scalar Foam::primitiveMesh::nonOrthThreshold_ = 70;    // deg
Foam::scalar Foam::primitiveMesh::skewThreshold_    = 4;
Foam::scalar Foam::primitiveMesh::planarCosAngle_   = 1.0e-6;
```

- You will be able to run simulations with mesh quality errors such as high skewness, high aspect ratio, and high non-orthogonality. But remember, they will affect the solution accuracy, might give you strange results, and eventually can made the solver blow-up.
- Have in mind that if you have bad quality meshes, you will need to adapt the numerics to deal with this kind of meshes. We will give you our recipe later when we deal with the numerics.
- You should avoid as much as possible non-orthogonality values close to 90. This is an indication that you have zero-volume cells.
- In overall, large aspect ratios do not represent a problem. It is just an indication that you have very fine meshes (which is the case when you are resolving the boundary layer).
- The default quality metrics in OpenFOAM® seems to be a little bit conservative. In our experience, we have found that you can run simulations with no numerical tricks with a non-orthogonality values up to 80 and skewness values up to 8.

Checking the mesh quality in OpenFOAM®

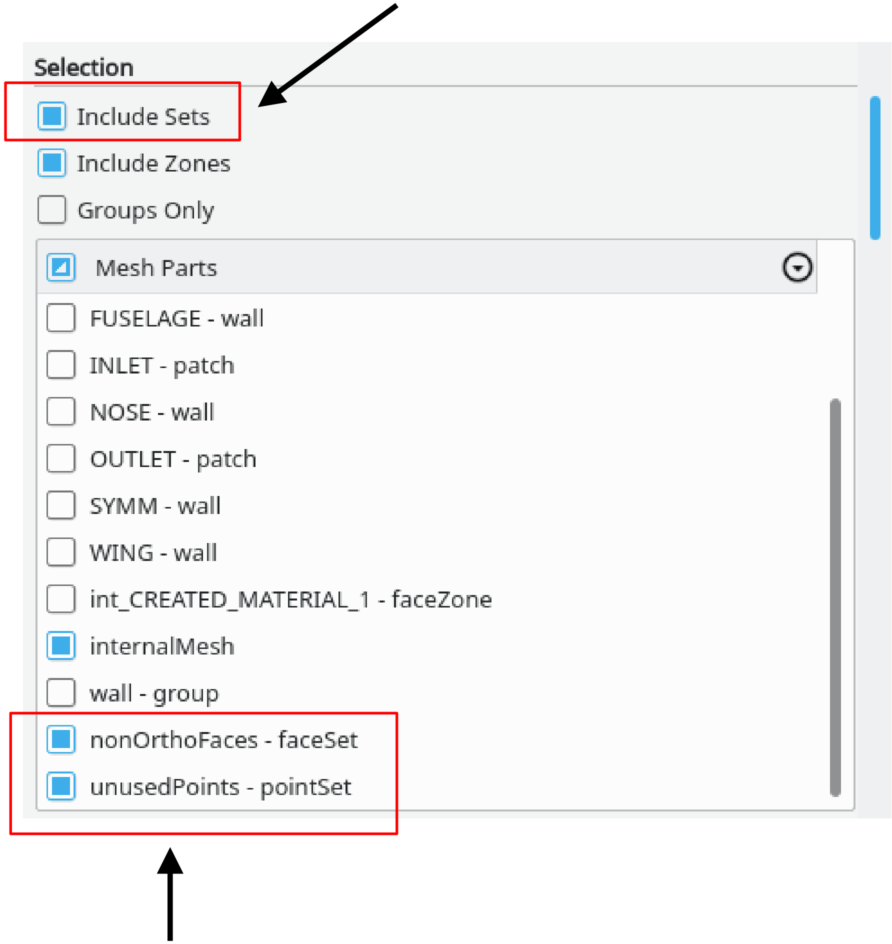
- To check the mesh quality and validity, OpenFOAM® comes with the utility `checkMesh`.
- To use this utility, just type in the terminal `checkMesh`, and read the screen output.
- `checkMesh` will look for/check for:
  - Mesh stats and overall number of cells of each type.
  - Check topology (boundary conditions definitions).
  - Check geometry and mesh quality (bounding box, cell volumes, skewness, orthogonality, aspect ratio, and so on).

- If for any reason `checkMesh` finds errors, it will give you a message and it will tell you what check failed.
- It will also write a set with the faulty cells, faces, and/or points.
- These sets are saved in the directory **`constant/polyMesh/sets/`**
- Mesh topology and patch topology errors must be repaired.
- You will be able to run with mesh quality errors such as skewness, aspect ratio, minimum face area, and non-orthogonality.
- But remember, they will severely tamper the solution accuracy, might give you strange results, and eventually can made the solver blow-up.
- Unfortunately, `checkMesh` does not repair these errors.
- You will need to check the geometry for possible errors and generate a new mesh.
- You can visualize the failed sets directly in `paraFoam` .
- You can also convert the failed sets into VTK format by using the utility `foamToVTK`.

Visualizing the failed sets in OpenFOAM®

- You can load the failed sets directly within `paraFoam`.
- Remember, you will need to create the sets. To do so, just run the `checkMesh` utility.
- If there are problems in the mesh, `checkMesh` will automatically save the sets in the directory **`constant/polyMesh/sets`**
- In `paraFoam`, simply select the option **Include Sets** and then select the sets you want to visualize.
- This method only works when using the wrapper `paraFoam`.
- If you are using paraview or a different scientific visualization application, you will need to convert the failed sets to VTK format or an alternative format.
- Also, when working with large meshes we prefer to convert the faulty sets to VTK format.
- To convert the faulty sets to VTK format you can use the utility `foamToVTK`.

Check this box to include sets



Selection

☑ Include Sets

☑ Include Zones

☐ Groups Only

☑ Mesh Parts

☐ FUSELAGE - wall

☐ INLET - patch

☐ NOSE - wall

☐ OUTLET - patch

☐ SYMM - wall

☐ WING - wall

☐ int_CREATED_MATERIAL_1 - faceZone

☑ internalMesh

☐ wall - group

☑ nonOrthoFaces - faceSet

☑ unusedPoints - pointSet

Failed sets

Mesh quality assessment in OpenFOAM®

Visualizing the failed sets in OpenFOAM®

- To convert the failed faces/cells/points to VTK format, you can proceed as follows:

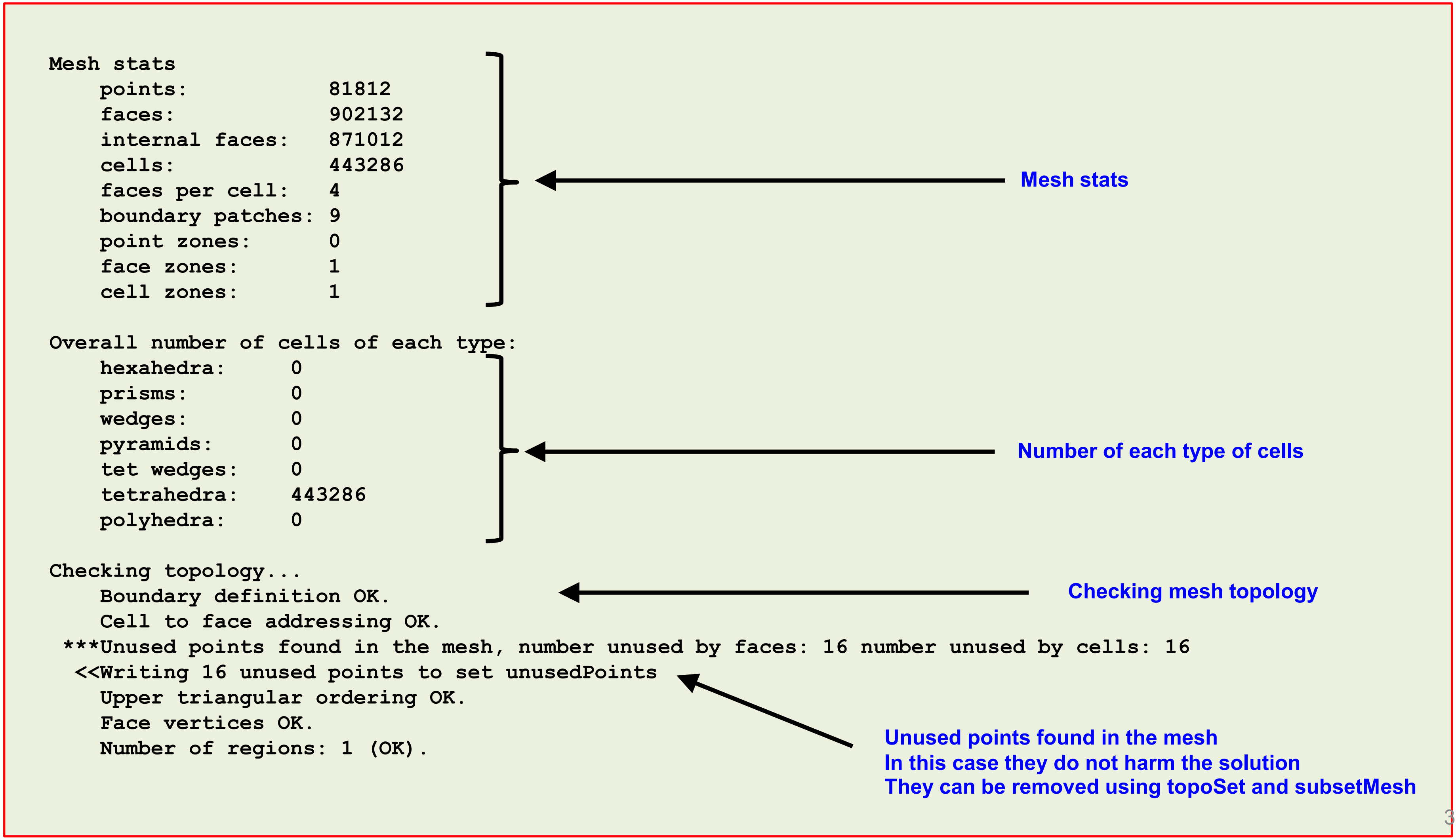
  - `$> foamToVTK -set_type name_of_sets`

  where **set_type** is the type of sets (faceSet, cellSet, pointSet, surfaceFields) and **name_of_sets** is the name of the set located in the directory **`constant/polyMesh/sets`** (highAspectRatioCells, nonOrthoFaces, wrongOrientedFaces, skewFaces, unusedPoints, and so on).

- At the end, `foamToVTK` will create a directory named **`VTK`**, where you will find the failed faces/cells/points in VTK format.
- At this point you can use `paraview/paraFoam` or any scientific visualization application to open the VTK files and visualize the failed sets.

Checking mesh quality in OpenFOAM®

- Sample `checkMesh` output,



```
Mesh stats
    points:           81812
    faces:            902132
    internal faces:   871012
    cells:            443286
    faces per cell:   4
    boundary patches: 9
    point zones:      0
    face zones:       1
    cell zones:       1

Overall number of cells of each type:
    hexahedra:     0
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    443286
    polyhedra:     0

Checking topology...
    Boundary definition OK.
    Cell to face addressing OK.
 ***Unused points found in the mesh, number unused by faces: 16 number unused by cells: 16
  <<Writing 16 unused points to set unusedPoints
    Upper triangular ordering OK.
    Face vertices OK.
    Number of regions: 1 (OK).
```

Mesh stats

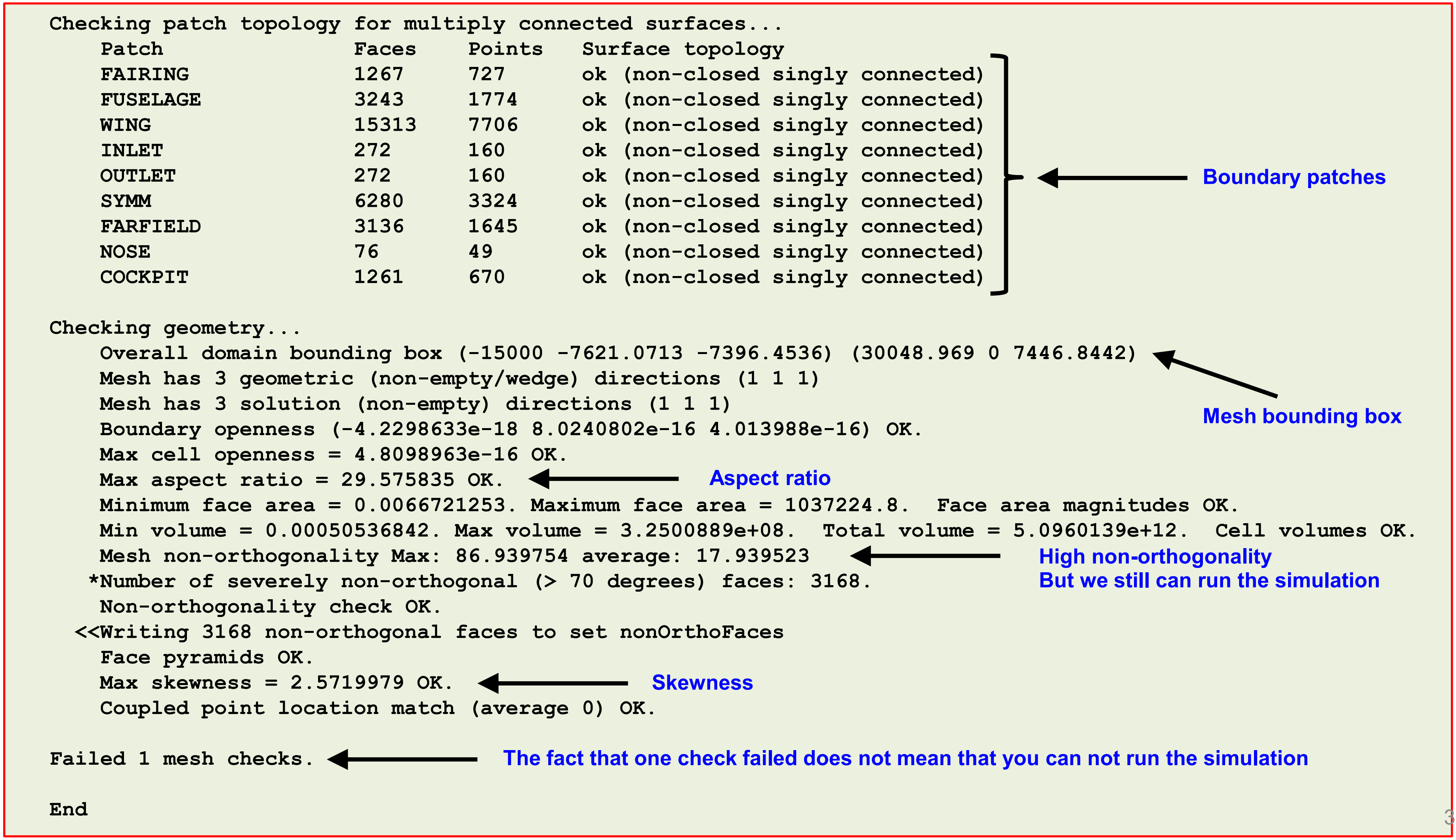
Number of each type of cells

Checking mesh topology

Unused points found in the mesh
In this case they do not harm the solution
They can be removed using topoSet and subsetMesh

Checking mesh quality in OpenFOAM®

- Sample `checkMesh` output,



```
Checking patch topology for multiply connected surfaces...
    Patch                   Faces     Points    Surface topology
    FAIRING                 1267      727       ok (non-closed singly connected)
    FUSELAGE                3243      1774      ok (non-closed singly connected)
    WING                    15313     7706      ok (non-closed singly connected)
    INLET                   272       160       ok (non-closed singly connected)
    OUTLET                  272       160       ok (non-closed singly connected)
    SYMM                    6280      3324      ok (non-closed singly connected)
    FARFIELD                3136      1645      ok (non-closed singly connected)
    NOSE                    76        49        ok (non-closed singly connected)
    COCKPIT                 1261      670       ok (non-closed singly connected)

Checking geometry...
    Overall domain bounding box (-15000 -7621.0713 -7396.4536) (30048.969 0 7446.8442)
    Mesh has 3 geometric (non-empty/wedge) directions (1 1 1)
    Mesh has 3 solution (non-empty) directions (1 1 1)
    Boundary openness (-4.2298633e-18 8.0240802e-16 4.013988e-16) OK.
    Max cell openness = 4.8098963e-16 OK.
    Max aspect ratio = 29.575835 OK.
    Minimum face area = 0.0066721253. Maximum face area = 1037224.8.  Face area magnitudes OK.
    Min volume = 0.00050536842. Max volume = 3.2500889e+08.  Total volume = 5.0960139e+12.  Cell volumes OK.
    Mesh non-orthogonality Max: 86.939754 average: 17.939523
   *Number of severely non-orthogonal (> 70 degrees) faces: 3168.
    Non-orthogonality check OK.
  <<Writing 3168 non-orthogonal faces to set nonOrthoFaces
    Face pyramids OK.
    Max skewness = 2.5719979 OK.
    Coupled point location match (average 0) OK.

Failed 1 mesh checks.

End
```

Boundary patches

Mesh bounding box

Aspect ratio

High non-orthogonality
But we still can run the simulation

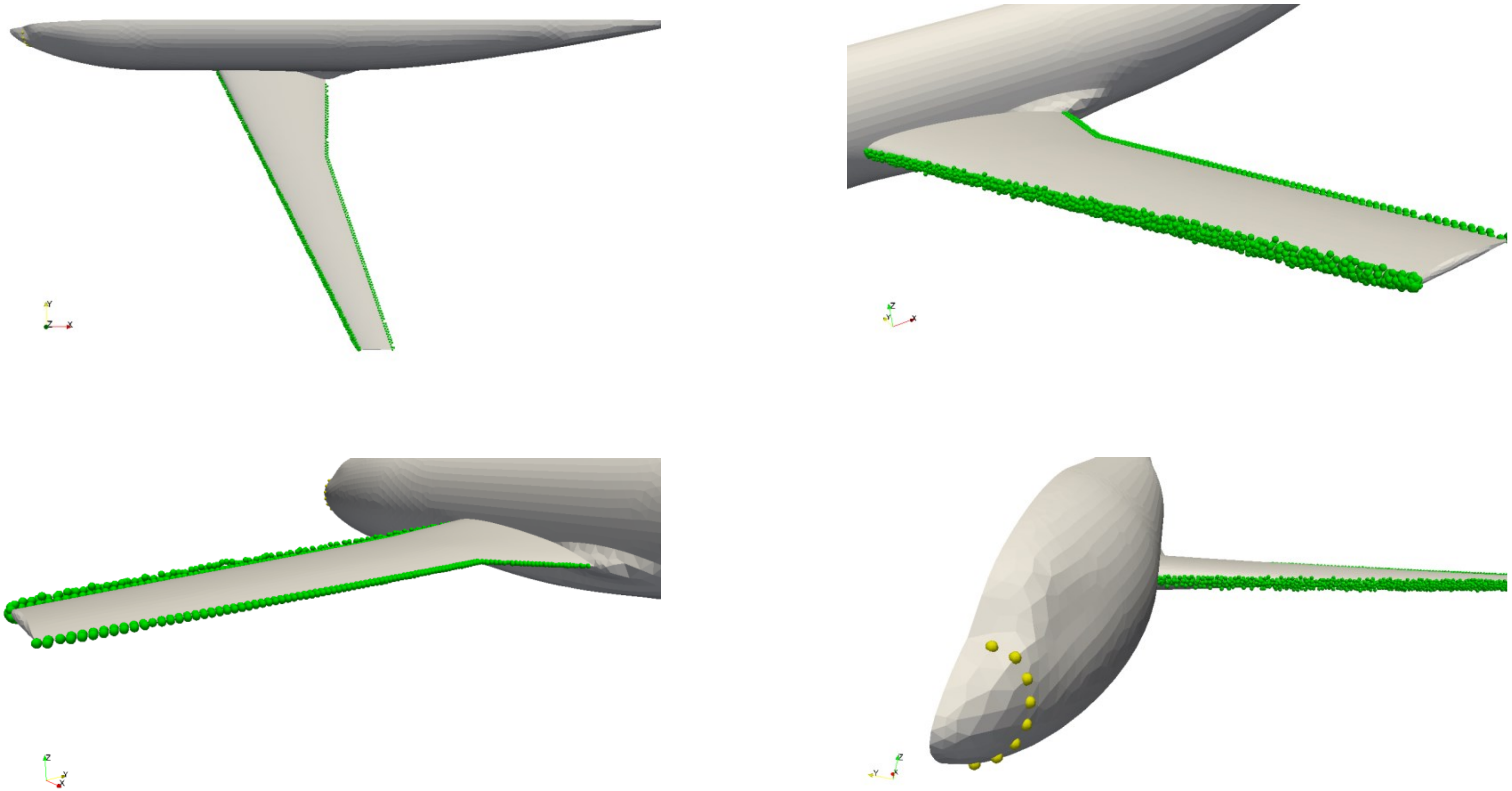
Skewness

The fact that one check failed does not mean that you can not run the simulation

Mesh quality assessment in OpenFOAM®

Visualization of faulty sets in paraFoam

- You will find this case ready to use in the directory, **`$PTOFC/mesh_quality_manipulation/M1_wingbody`**
- To run the case, just follow the instructions in the *`README.FIRST`* files.



Non orthogonal faces (green spheres) and unused points (yellow spheres)

Roadmap

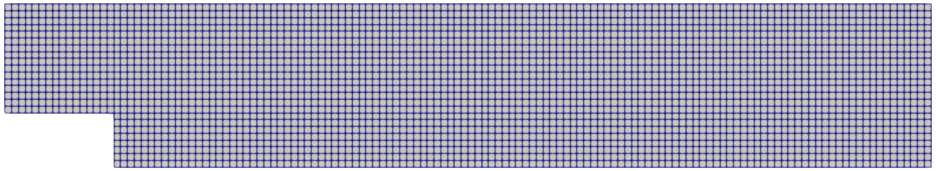
1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. **Mesh generation using blockMesh.**
5. Mesh generation using snappyHexMesh.
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

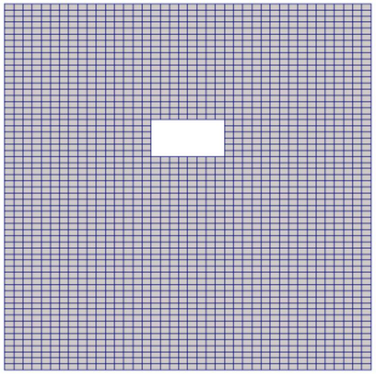
blockMesh

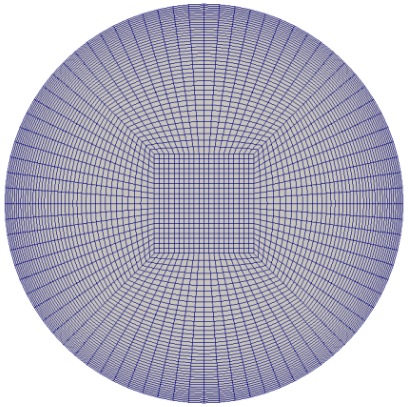
- *“blockMesh is a multi-block mesh generator.”*
- For simple geometries, the mesh generation utility `blockMesh` can be used.
- The mesh is generated from a dictionary file named *`blockMeshDict`* located in the **`system`** directory.
- This meshing tool generates high quality meshes.
- It is the tool to use for very simple geometries. As the complexity of the geometry increases, the effort and time required to setup the dictionary increases a lot.
- Usually, the background mesh used with `snappyHexMesh` consist of a single rectangular block; therefore, `blockMesh` can be used with no problem.
- It is highly recommended to create a template of the dictionary *`blockMeshDict`* that you can change according to the dimensions of your domain.
- You can also use m4 or Python scripting to automate the whole process.

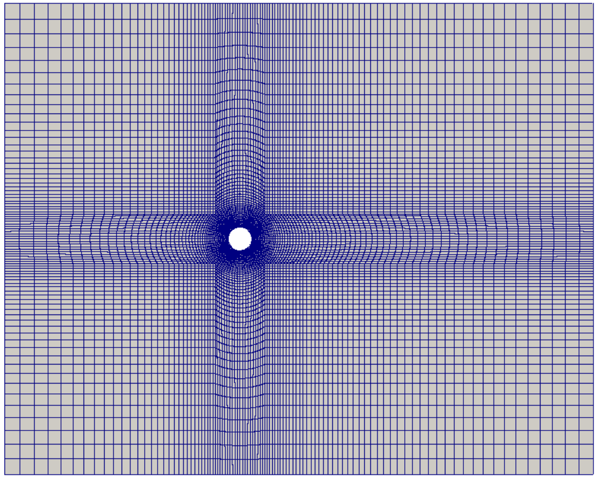
blockMesh

- These are a few meshes that you can generate using `blockMesh`.
- As you can see, they are not very complex.
- However, generating the blocking topology requires some effort.



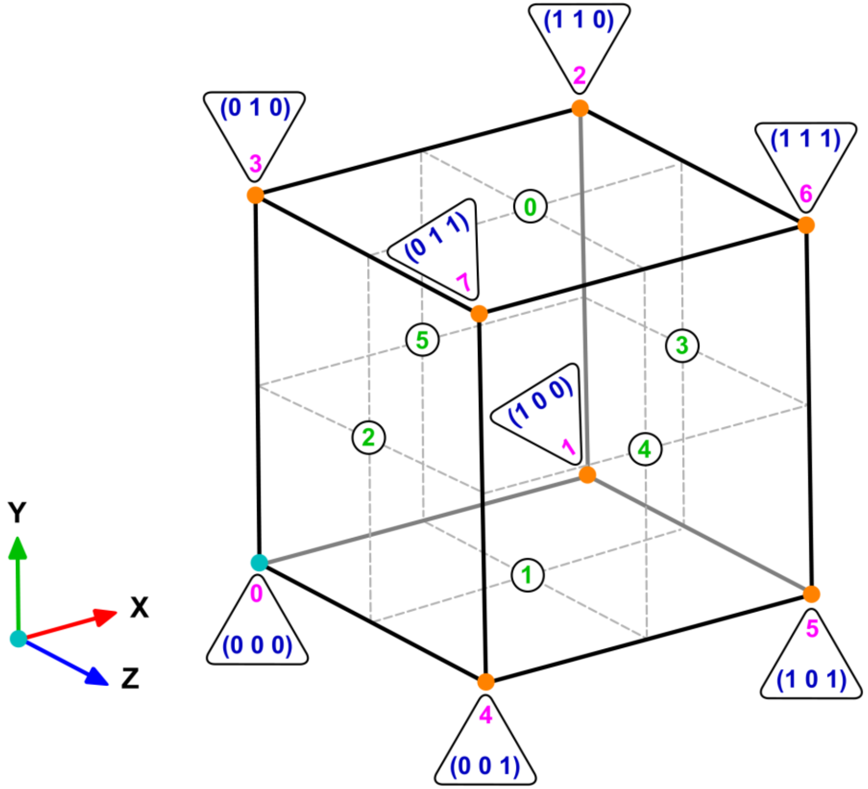






Mesh generation using blockMesh

blockMesh workflow



VERTICES

● 0	0 0 0
1	1 0 0
2	1 1 0
3	0 1 0
4	0 0 1
5	1 0 1
6	1 1 1
7	0 1 1

BLOCK (HEX)

0 1 2 3 4 5 6 7

FACES

0	3 7 6 2
1	1 5 4 0
2	0 4 7 3
3	2 6 5 1
4	0 3 2 1
5	4 5 6 7

FACE INDEX IS NOT IMPORTANT

- To generate a mesh with `blockMesh`, you will need to define the vertices, block connectivity and number of cells in each direction.
- To assign boundary patches, you will need to define the faces connectivity

blockMesh guided tutorials

- Meshing with blockMesh – Case 1.
- We will use the square cavity case.
- You will find this case in the directory:

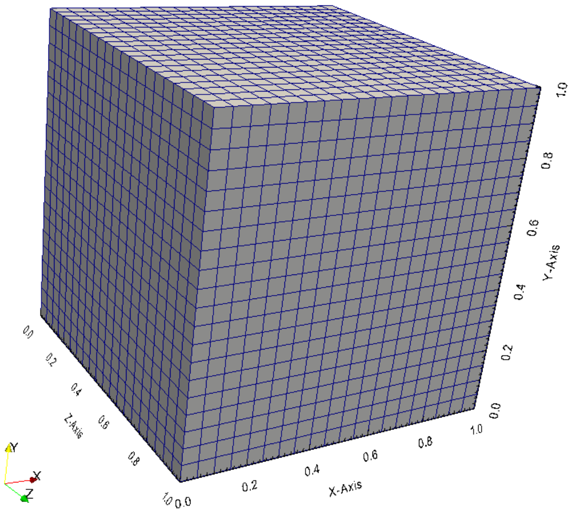
```
$PTOFC/101BLOCKMESH/C1
```

- In the case directory, you will find the README.FIRST file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, run_all.sh, run_mesh.sh, run_sampling.sh, run_solver.sh, and so on. These files can be used to run the case automatically by typing in the terminal, for example, sh run_solver.
- We highly recommend you to open the README.FIRST file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

blockMesh guided tutorials

What are we going to do?

- We will use this simple case to take a close look at a *blockMeshDict* dictionary.
- We will study all sections in the *blockMeshDict* dictionary.
- We will introduce two features useful for parameterization, namely, macro syntax and inline calculations.
- You can use this dictionary as a *blockMeshDict* template that you can change automatically according to the dimensions of your domain and the desired cell spacing.

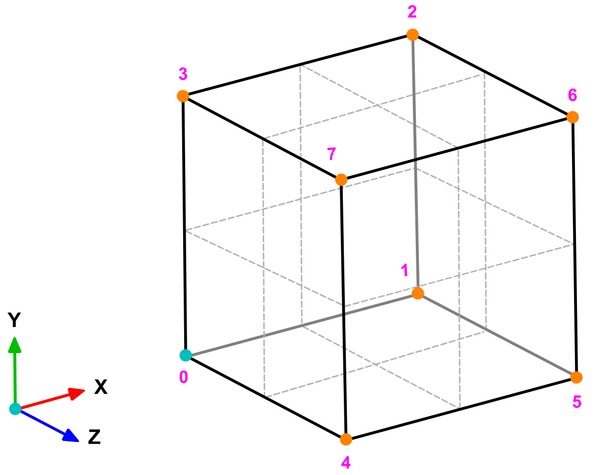


blockMesh guided tutorials

The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax – $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
44      vertices
45      (
46      //BLOCK 0
47          ($xmin  $ymin  $zmin)      //0
48          ($xmax  $ymin  $zmin)      //1
49          ($xmax  $ymax  $zmin)      //2
50          ($xmin  $ymax  $zmin)      //3
51          ($xmin  $ymin  $zmax)      //4
52          ($xmax  $ymin  $zmax)      //5
53          ($xmax  $ymax  $zmax)      //6
54          ($xmin  $ymax  $zmax)      //7
66      );
```

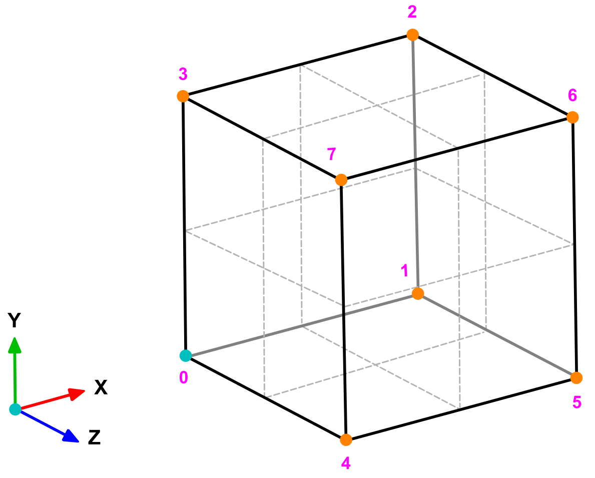
- The keyword **convertToMeters** (line 17), is a scaling factor. In this case we do not scale the dimensions.
- In lines 19-24 we declare some variables using macro syntax notation. With macro syntax, we first declare the variables and their values (lines 19-24), and then we can use the variables by adding the symbol **$** to the variable name (lines 47-54).
- In lines 30-32 we use macro syntax to declare another set of variables that will be used later.
- Macro syntax is a very convenient way to parameterize dictionaries.



The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
44      vertices
45      (
46      //BLOCK 0
47          ($xmin  $ymin  $zmin)      //0
48          ($xmax  $ymin  $zmin)      //1
49          ($xmax  $ymax  $zmin)      //2
50          ($xmin  $ymax  $zmin)      //3
51          ($xmin  $ymin  $zmax)      //4
52          ($xmax  $ymin  $zmax)      //5
53          ($xmax  $ymax  $zmax)      //6
54          ($xmin  $ymax  $zmax)      //7
66      );
```

- In lines 34-40 we are doing inline calculations using the directive **#calc**.
- Basically we are programming directly in the dictionary. OpenFOAM® will compile this function as it reads it.
- With inline calculations and **codeStream** you can access many OpenFOAM® functions from the dictionaries.
- Inline calculations and **codeStream** are very convenient ways to parameterize dictionaries and program directly on the dictionaries.



blockMesh guided tutorials

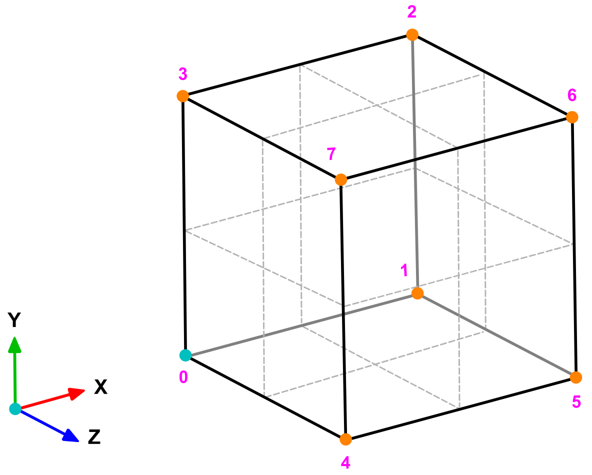
The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
44      vertices
45      (
46      //BLOCK 0
47          ($xmin  $ymin  $zmin)      //0
48          ($xmax  $ymin  $zmin)      //1
49          ($xmax  $ymax  $zmin)      //2
50          ($xmin  $ymax  $zmin)      //3
51          ($xmin  $ymin  $zmax)      //4
52          ($xmax  $ymin  $zmax)      //5
53          ($xmax  $ymax  $zmax)      //6
54          ($xmin  $ymax  $zmax)      //7
66      );
```

- To do inline calculations using the directive **#calc**, we proceed as follows (we will use line 35 as example):

ly #calc "$ymax - $ymin";

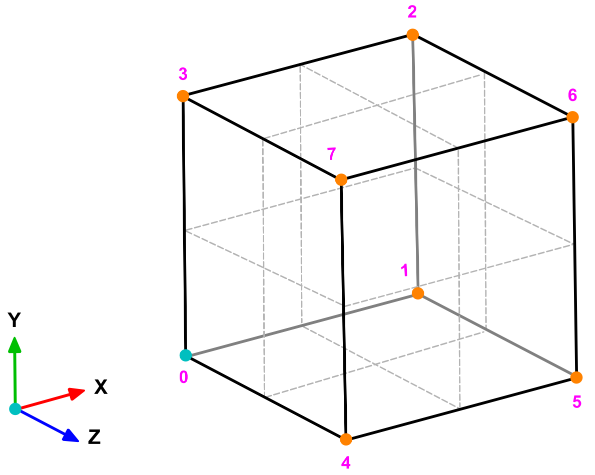
- We first give a name to the new variable (**ly**), we then tell OpenFOAM® that we want to do an inline calculation (**#calc**), and then we do the inline calculation (**"$ymax-$ymin";)**. Notice that the operation must be between double quotation marks.



The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax - $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
44      vertices
45      (
46      //BLOCK 0
47          ($xmin  $ymin  $zmin)      //0
48          ($xmax  $ymin  $zmin)      //1
49          ($xmax  $ymax  $zmin)      //2
50          ($xmin  $ymax  $zmin)      //3
51          ($xmin  $ymin  $zmax)      //4
52          ($xmax  $ymin  $zmax)      //5
53          ($xmax  $ymax  $zmax)      //6
54          ($xmin  $ymax  $zmax)      //7
66      );
```

- In lines lines 34-36, we use inline calculations to compute the length in each direction.
- Then we compute the number of cells to be used in each direction (lines 38-40).
- To compute the number of cells we use as cell spacing the values declared in lines 30-32.
- By proceeding in this way, we can compute automatically the number of cells needed in each direction according to the desired cell spacing.

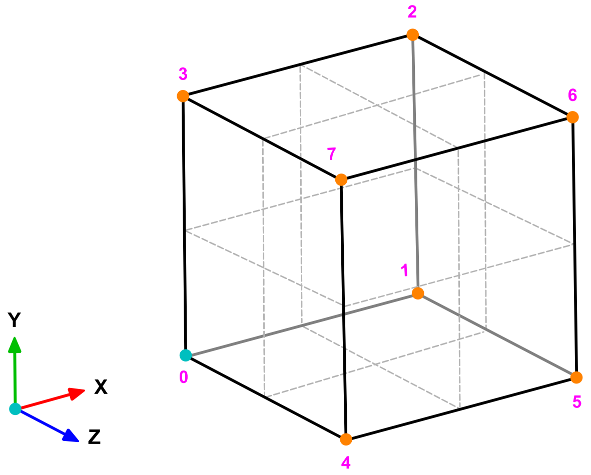


blockMesh guided tutorials

The *blockMeshDict* dictionary.

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 1;
21      ymin 0;
22      ymax 1;
23      zmin 0;
24      zmax 1;
25
30      deltax 0.05;
31      deltay 0.05;
32      deltaz 0.05;
33
34      lx #calc "$xmax - $xmin";
35      ly #calc "$ymax - $ymin";
36      lz #calc "$zmax – $zmin";
37
38      xcells #calc "round(($lx)/($deltax))";
39      ycells #calc "round(($ly)/($deltay))";
40      zcells #calc "round(($lz)/($deltaz))";
41
44      vertices
45      (
46      //BLOCK 0
47          ($xmin  $ymin  $zmin)      //0
48          ($xmax  $ymin  $zmin)      //1
49          ($xmax  $ymax  $zmin)      //2
50          ($xmin  $ymax  $zmin)      //3
51          ($xmin  $ymin  $zmax)      //4
52          ($xmax  $ymin  $zmax)      //5
53          ($xmax  $ymax  $zmax)      //6
54          ($xmin  $ymax  $zmax)      //7
66      );
```

- In the vertices section (lines 44-66), we define the vertex coordinates of the geometry.
- In this case, there are eight vertices defining a 3D block.
- Remember, OpenFOAM® always uses 3D meshes, even if the simulation is 2D. For 2D meshes, you only add one cell in the third dimension.
- Notice that the vertex numbering starts from 0 (as the counters in c++). This numbering applies for blocks as well.

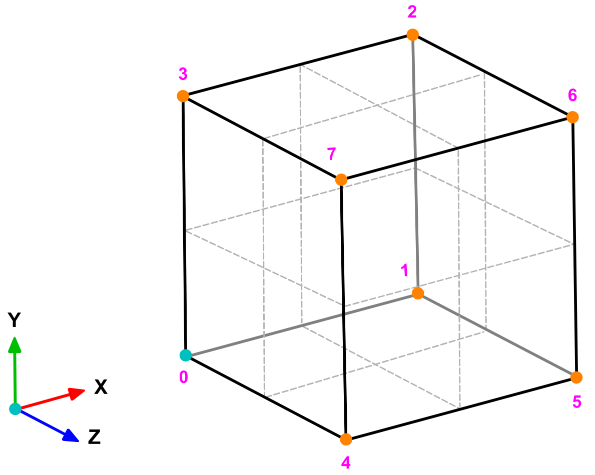


blockMesh guided tutorials

The *blockMeshDict* dictionary.

- In lines 68-71, we define the block topology, **hex** means that it is a structured hexahedral block. In this case, we are generating a rectangular mesh.
- In line 70, **(0 1 2 3 4 5 6 7)** are the vertices used to define the block (and yes, the order is important). Each hex block is defined by eight vertices, in sequential order. Where the first vertex in the list represents the origin of the coordinate system (vertex **0** in this case).
- **($xcells $ycells $zcells)** is the number of mesh cells in each direction (**X Y Z**). Notice that we are using macro syntax, and we compute the values using inline calculations.
- **simpleGrading (1 1 1)** is the grading or mesh stretching in each direction (**X Y Z**), in this case the mesh is uniform. We will deal with mesh grading/stretching in the next case.

```
68      blocks
69      (
70          hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
71      );
72
73      edges
74      (
75
76      );
```

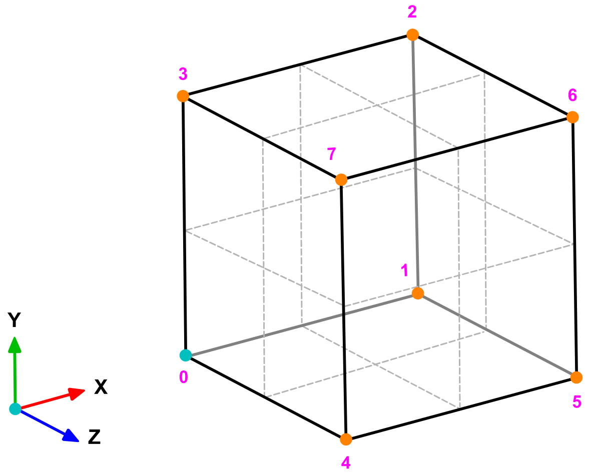


blockMesh guided tutorials

The *blockMeshDict* dictionary.

- Let us talk about the block ordering **hex (0 1 2 3 4 5 6 7)**, which is extremely important.
- **hex** blocks are defined by eight vertices in sequential order. Where the first vertex in the list represents the origin of the coordinate system (vertex **0** in this case).
- Starting from this vertex, we construct the block topology. So in this case, the first part of the block is made up by vertices **0 1 2 3** and the second part of the block is made up by vertices **4 5 6 7** (notice that we start from vertex **4** which is the projection in the **Z**-direction of vertex **0**).
- In this case, the vertices are ordered in such a way that if we look at the screen/paper (-z direction), the vertices rotate counter-clockwise.
- If you add a second block, you must identify the first vertex and starting from it, you should construct the block topology. In this case, you will need to merges faces, you will find more information about merging face in the supplement lectures.

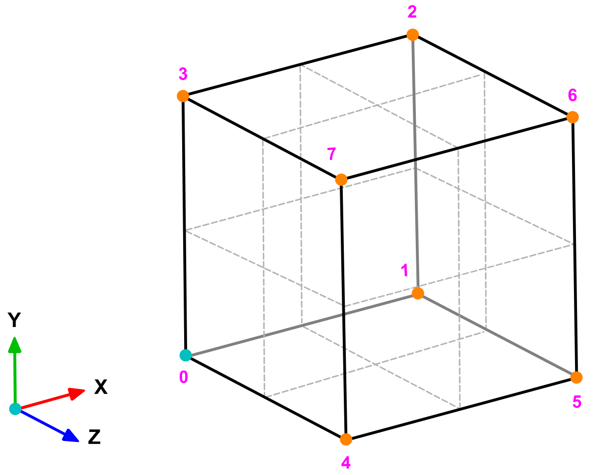
```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
);

edges
(

);
```



The *blockMeshDict* dictionary.

- Edges, are constructed from the vertices definition.
- Each edge joining two vertices is assumed to be straight by default.
- The user can specify any edge to be curved by entries in the section **edges**.
- Possible options are Bspline, arc, line, polyline, project, projectCurve, spline.
- For example, to define an arc we first define the vertices to be connected to form an edge and then we give an interpolation point.
- To define a polyline we first define the vertices to be connected to form an edge and then we give a list of the coordinates of the interpolation points.
- In this case and as we do not specify anything, all edges are assumed to be straight lines.

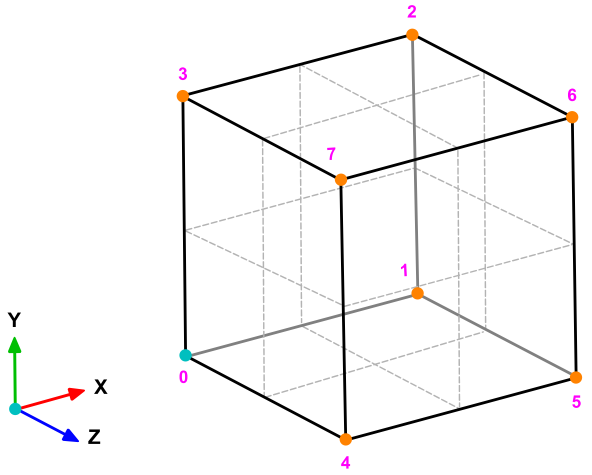
```
blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
);

edges
(

);
```



The *blockMeshDict* dictionary.

```
    boundary
    (
        top
        {
            type wall;
            faces
            (
                (3 7 6 2)
            );
        }
        left
        {
            type wall;
            faces
            (
                (0 4 7 3)
            );
        }
        right
        {
            type wall;
            faces
            (
                (2 6 5 1)
            );
        }
        bottom
        {
            type wall;
            faces
            (
                (0 1 5 4)
            );
        }
```

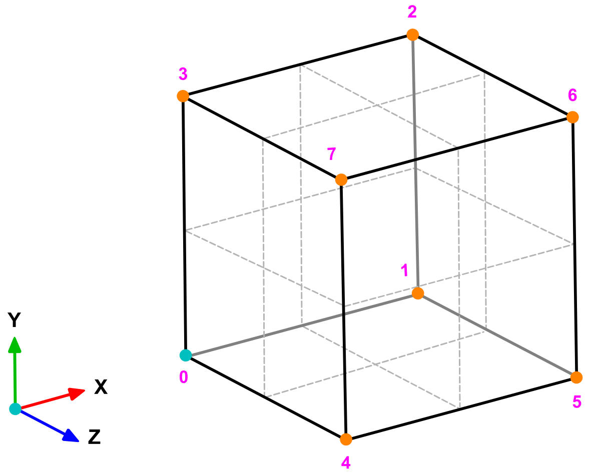
- In the section **boundary**, we define all the patches where we want to apply boundary conditions.
- This step is of paramount importance, because if we do not define the surface patches, we will not be able to apply the boundary conditions to individual surface patches.



The *blockMeshDict* dictionary.

```
78      boundary
79      (
80          top
81          {
82              type wall;
83              faces
84              (
85                  (3 7 6 2)
86              );
87          }
88          left
89          {
90              type wall;
91              faces
92              (
93                  (0 4 7 3)
94              );
95          }
96          right
97          {
98              type wall;
99              faces
100             (
101                 (2 6 5 1)
102             );
103         }
104         bottom
105         {
106             type wall;
107             faces
108             (
109                 (0 1 5 4)
110             );
111         }
```

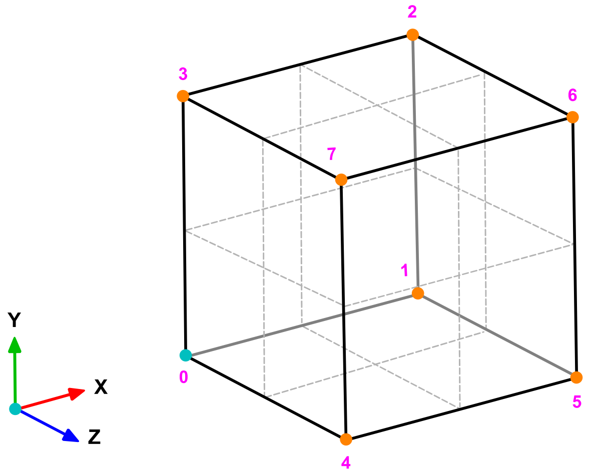
- In lines 80-87 we define a boundary patch.
- In line 80 we define the patch name **top** (the name is given by the user).
- In line 82 we give a **base type** to the surface patch. In this case **wall** (do not worry we are going to talk about this later).
- In line 85 we give the connectivity list of the vertices that made up the surface patch or face, that is, **(3 7 6 2)**.
- Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.



The *blockMeshDict* dictionary.

```
78    boundary
79    (
80        top
81        {
82            type wall;
83            faces
84            (
85                (3 7 6 2)
86            );
87        }
88        left
89        {
90            type wall;
91            faces
92            (
93                (0 4 7 3)
94            );
95        }
96        right
97        {
98            type wall;
99            faces
100           (
101               (2 6 5 1)
102           );
103       }
104       bottom
105       {
106           type wall;
107           faces
108           (
109               (0 1 5 4)
110           );
111       }
```

- Have in mind that the vertices need to be neighbors and it does not matter if the ordering is clockwise or counterclockwise.
- Remember, faces are defined by a list of 4 vertex numbers, e.g., **(3 7 6 2)**.
- In lines 88-95 we define the patch **left**.
- In lines 96-103 we define the patch **right**.
- In lines 104-11 we define the patch **bottom**.

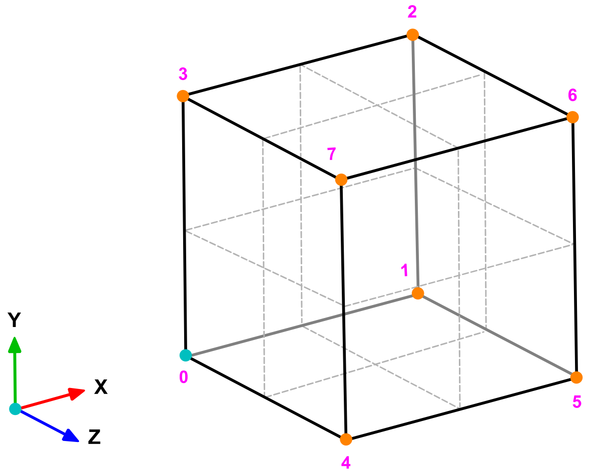


The *blockMeshDict* dictionary.

```
112        front
113        {
114            type wall;
115            faces
116            (
117                (4 5 6 7)
118            );
119        }
120        back
121        {
122            type wall;
123            faces
124            (
125                (0 3 2 1)
126            );
127        }
128    );
129
130    mergePatchPairs
131    (
132
133    );
```

- In lines 112-119 we define the patch **front**.
- In lines 120-127 we define the patch **back**.
- You can also group many faces into one patch, for example, instead of creating the patches **front** and **back**, you can group them into a single patch named **backAndFront**, as follows,

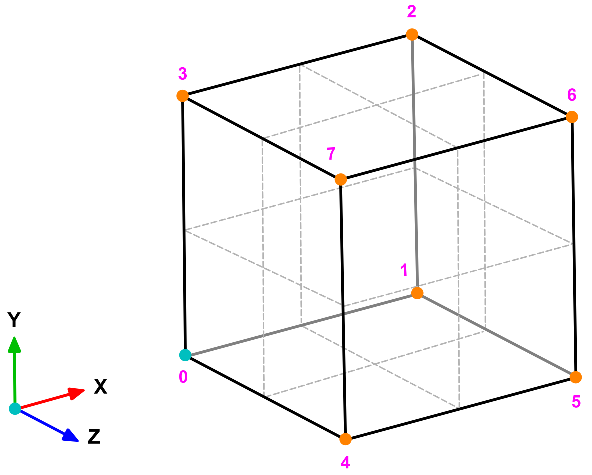
```
backAndFront
{
    type wall;
    faces
    (
        (4 5 6 7)
        (0 3 2 1)
    );
}
```



The *blockMeshDict* dictionary.

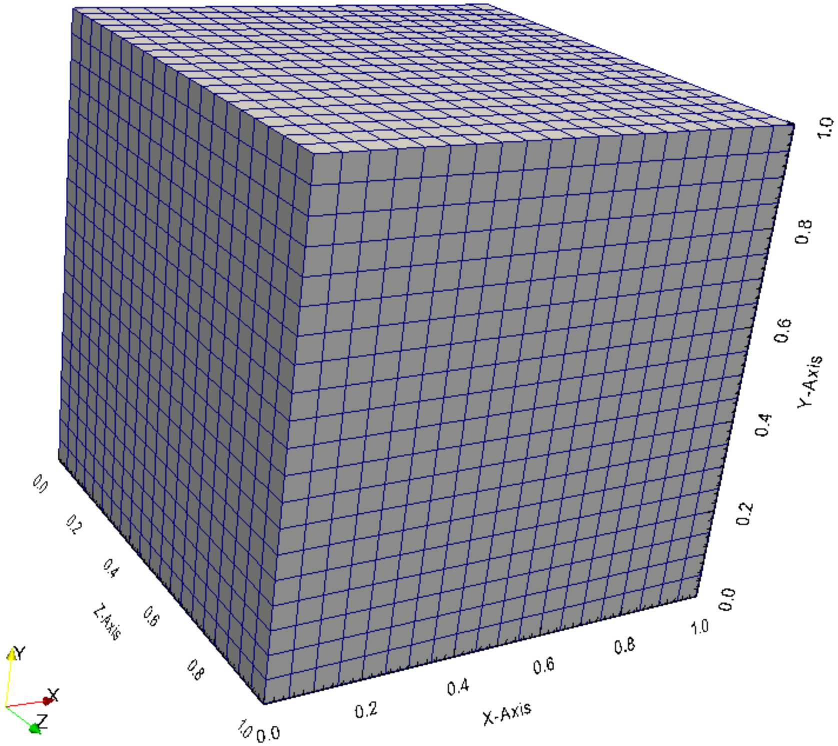
```
112         front
113         {
114             type wall;
115             faces
116             (
117                 (4 5 6 7)
118             );
119         }
120         back
121         {
122             type wall;
123             faces
124             (
125                 (0 3 2 1)
126             );
127         }
128     );
129
130     mergePatchPairs
131     (
132
133     );
```

- We can merge blocks in the section **mergePatchPairs** (lines 130-133).
- The block patches to be merged must be first defined in the **boundary** list, `blockMesh` then connect the two blocks.
- In this case, as we have one single block there is no need to merge patches.



The *blockMeshDict* dictionary.

- To sum up, the `blockMeshDict` dictionary generates a single block with:
  - **X/Y/Z** dimensions: **1.0**/**1.0**/**1.0**
  - As the cell spacing in all directions is defined as **0.05**, it will use the following number of cells in the **X**, **Y** and **Z** directions: **20** x **20** x **20** cells.
  - One single **hex** block with straight lines.
  - Six patches of base type **wall**, namely, **left**, **right**, **top**, **bottom**, **front** and **back**.
- The information regarding the patch **base type** and patch **name** is saved in the file `boundary`. Feel free to modify this file to fit your needs.
- Remember to use the utility `checkMesh` to check the quality of the mesh and look for topological errors.
- Topological errors must be repaired.



blockMesh guided tutorials

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

- First of all, this file is automatically generated after you create the mesh or you convert it from a third-party format.
- In this file, the geometrical information related to the **base type** patch of each boundary of the domain is specified.
- The **base type** boundary condition is the actual surface patch where we are going to apply a **primitive type** boundary condition (or numerical boundary condition).
- The **primitive type** boundary condition assign a field value to the surface patch.
- You define the **numerical type** patch (or the value of the boundary condition), in the directory **0** or time directories.
- The **name** and **base type** of the patches was defined in the dictionary *blockMeshDict* in the section **boundary**.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **top** from **wall** to **patch**.

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

- If you do not define the boundary patches in the dictionary *blockMeshDict*, they are grouped automatically in a default group named **defaultFaces** of type **empty**.
- For instance, if you do not assign a **base type** to the patch **front**, it will be grouped as follows:

```
defaultFaces
{
        type                    empty;
        inGroups                1(empty);
        nFaces                  400;
        startFace               24800;
}
```

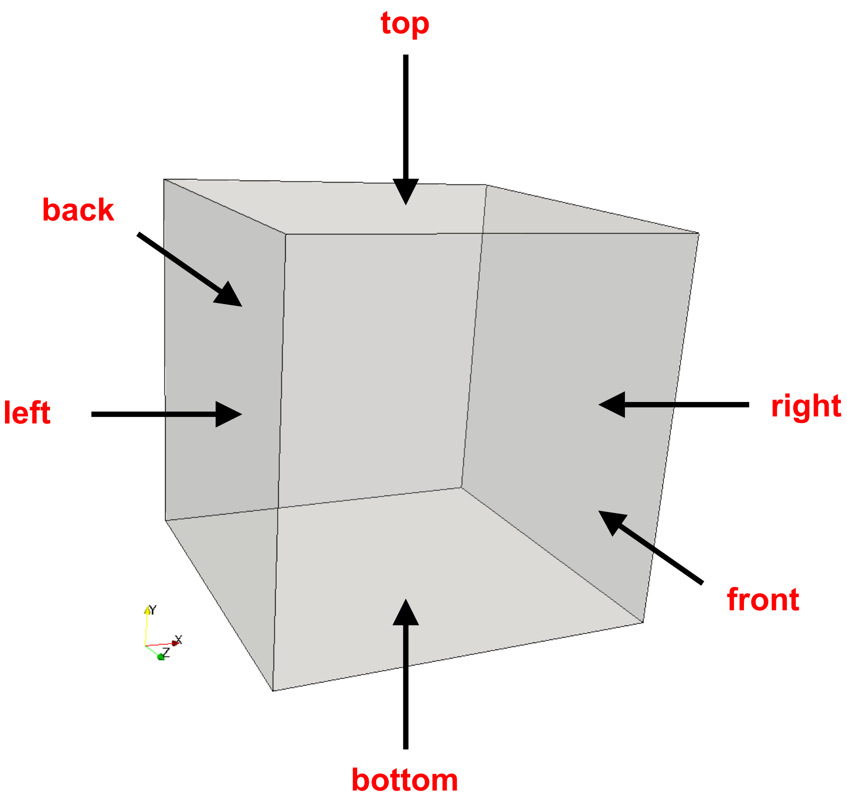
- Remember, you can manually change the name and type.

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

Number of surface patches

In the list bellow there must be 6 patches definition.



blockMesh guided tutorials

The *constant/polyMesh/boundary* dictionary

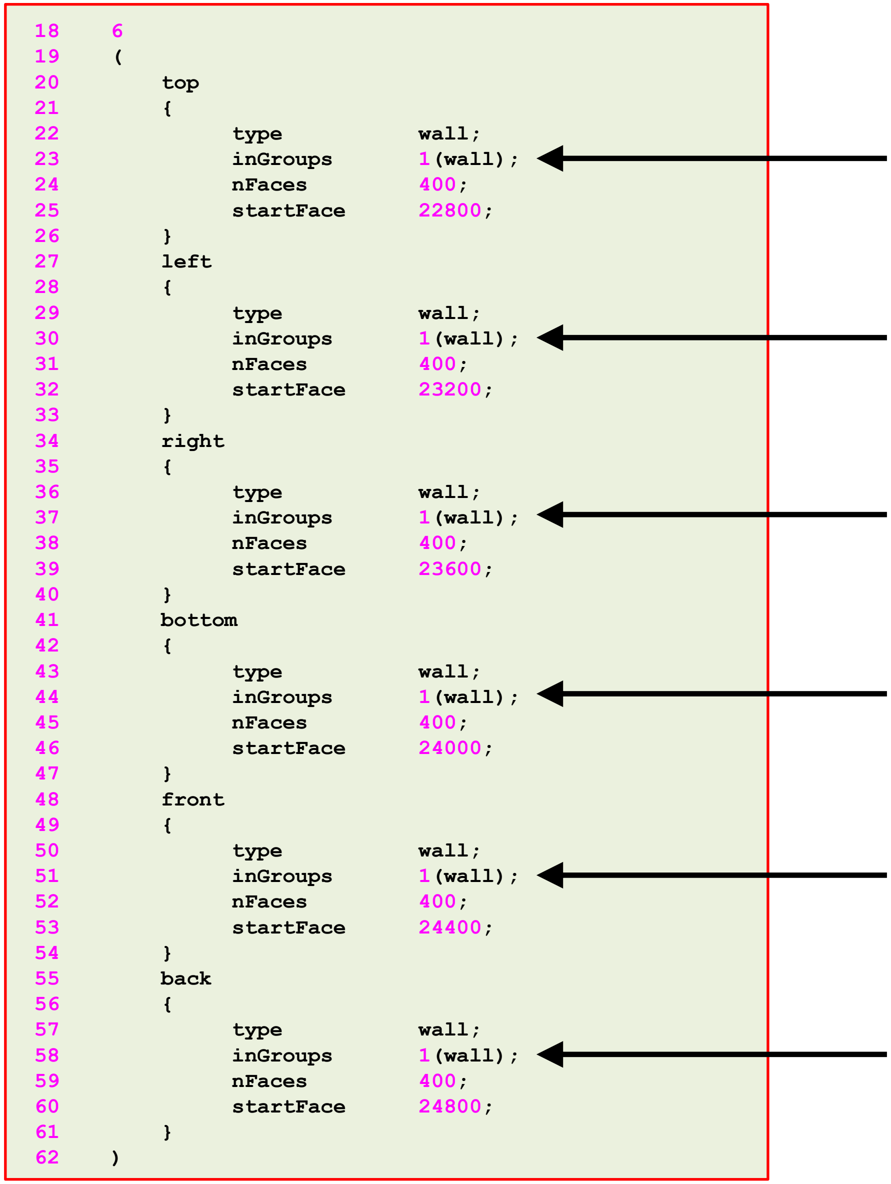
```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

Name and type of the surface patches

- The **name** and **base type** of the patch is given by the user.
- In this case the **name** and **base type** was assigned in the dictionary *blockMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **top** from **wall** to **patch**.

blockMesh guided tutorials

The *constant/polyMesh/boundary* dictionary



```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

inGroups keyword

- This is optional.
- You can erase this information safely.
- It is used to group patches during visualization in ParaView/paraFoam. If you open this mesh in paraFoam you will see that there are two groups, namely: **wall** and **empty**.
- As usual, you can change the name.
- If you want to put a surface patch in two groups, you can proceed as follows:

2(wall wall1)

In this case the surface patch belongs to the group **wall** (which can have another patch) and the group **wall1**

The *constant/polyMesh/boundary* dictionary

```
6
(
    top
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       22800;
    }
    left
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23200;
    }
    right
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       23600;
    }
    bottom
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24000;
    }
    front
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24400;
    }
    back
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       24800;
    }
)
```

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information.**
- Basically, this is telling you the starting face and ending face of the patch.
- This information is created automatically when generating the mesh or converting the mesh.

blockMesh guided tutorials

Running the case

- To generate the mesh, in the terminal window type:

```
1. $> foamCleanTutorials
2. $> blockMesh
3. $> checkMesh
4. $> paraFoam
```

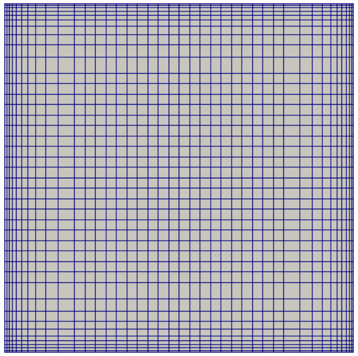
- If you want to visualize the blocking topology, type in the terminal

```
1. $> paraFoam -block
```

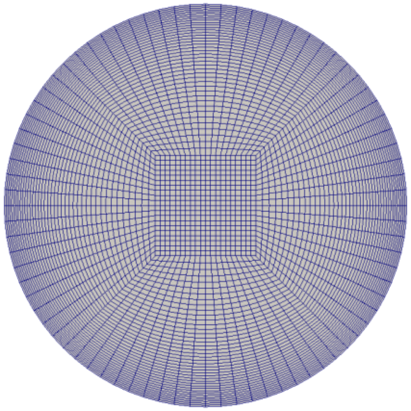
- You can run the rest of the cases following the same steps.

Final remarks on blockMesh

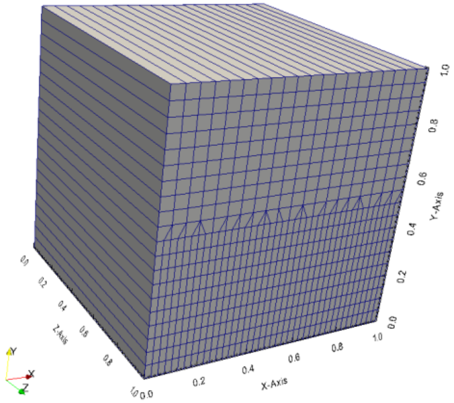
- For the moment, we will limit the use of `blockMesh` to single-block mesh topologies, which are used to run some simple cases and are the starting point for `snappyHexMesh`.
- But have in mind that you can do more elaborated meshes, however, it requires careful setup of the input dictionary.
- Have in mind that it can be really tricky to generate multi-block meshes with curve edges.
- With the training material, you will find a set of supplement slides where we explain how to create multi-block meshes, add stretching, and how to define curve edges.



Single-block mesh with multi-stretching



Multi-block mesh with curved edges and multi-stretching



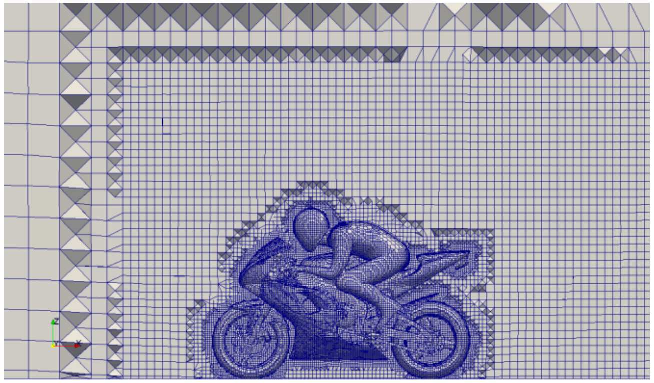
Multi-block mesh with face merging

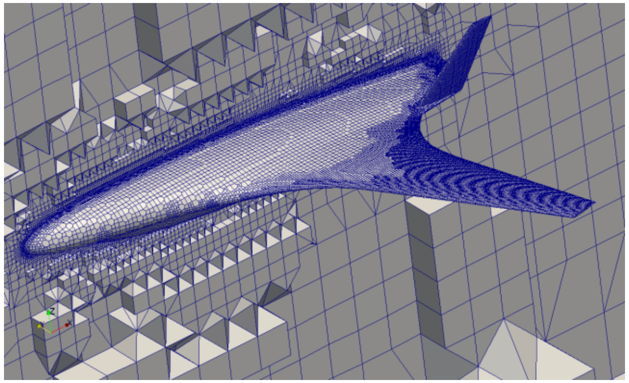
Roadmap

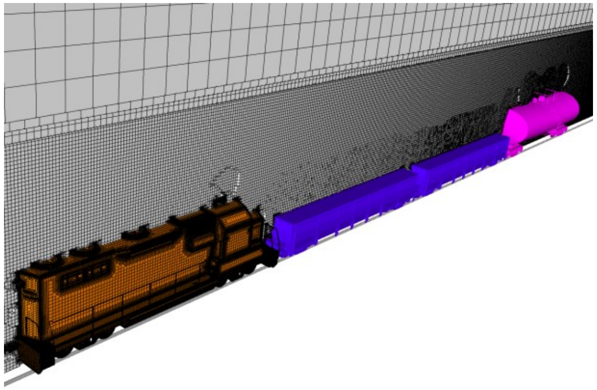
1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. **Mesh generation using snappyHexMesh.**
6. snappyHexMesh guided tutorials.
7. Mesh conversion
8. Geometry and mesh manipulation utilities

snappyHexMesh

- *"Automatic split hex mesher. Refines and snaps to surface."*
- For complex geometries, the mesh generation utility `snappyHexMesh` can be used.
- The `snappyHexMesh` utility generates 3D meshes containing hexahedra and split-hexahedra from a triangulated surface geometry in Stereolithography (STL) format.
- The mesh is generated from a dictionary file named *`snappyHexMeshDict`* located in the system directory and a triangulated surface geometry file located in the directory **`constant/triSurface`**.



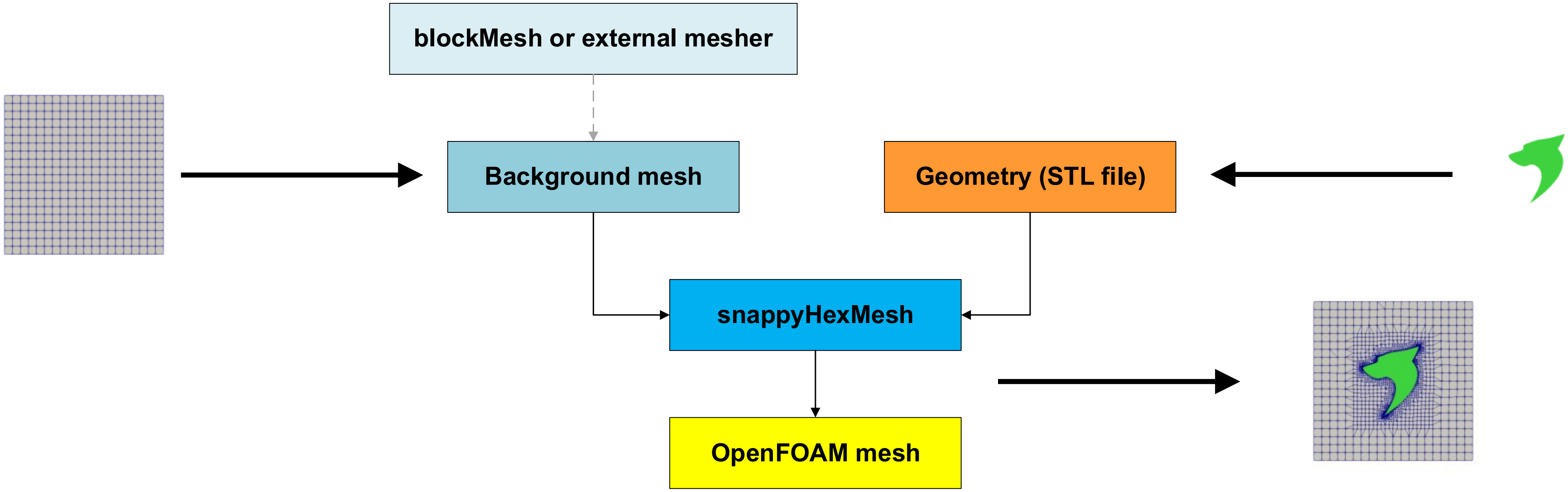




Mesh generation using snappyHexMesh

snappyHexMesh workflow

- To generate a mesh with snappyHexMesh we proceed as follows:
    - Generation of a background or base mesh.
    - Geometry definition.
    - Generation of a castellated mesh or cartesian mesh.
    - Generation of a snapped mesh or body fitted mesh.
    - Addition of layers close to the surfaces or boundary layer meshing.
    - Check/enforce mesh quality.



blockMesh or external mesher

Background mesh

Geometry (STL file)

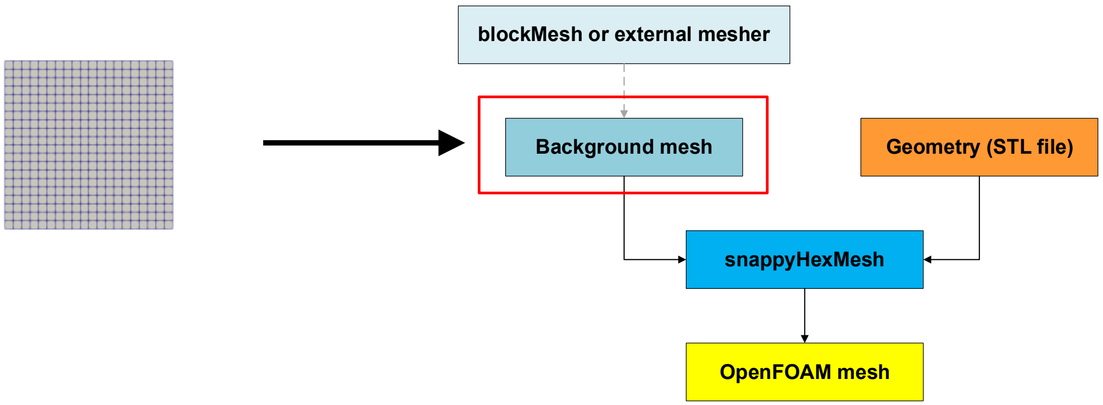
snappyHexMesh

OpenFOAM mesh

Mesh generation using snappyHexMesh

snappyHexMesh workflow – Background mesh

- The background or base mesh can be generated using blockMesh or an external mesher.
- The following criteria must be observed when creating the background mesh:
    - The mesh must consist purely of hexes.
    - The cell aspect ratio should be approximately 1, at least near the STL surface.
    - There must be at least one intersection of a cell edge with the STL surface.



blockMesh or external mesher

Background mesh

Geometry (STL file)

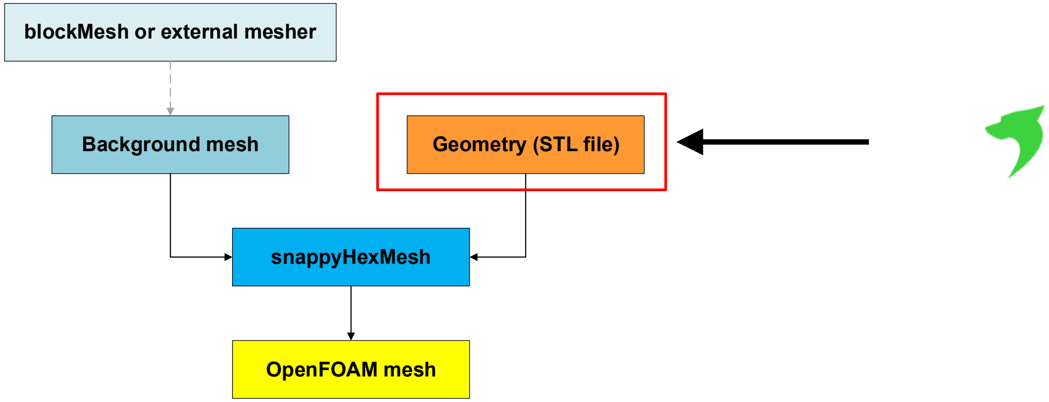
snappyHexMesh

OpenFOAM mesh

Mesh generation using snappyHexMesh

snappyHexMesh workflow – Geometry (STL file)

- The STL geometry can be obtained from any geometry modeling tool.
- The STL file can be made up of a single surface describing the geometry, or multiple surfaces that describe the geometry.
- In the case of a STL file with multiple surfaces, we can use local refinement in each individual surface. This gives us more control when generating the mesh.
- The STL geometry is always located in the directory **`constant/triSurface`**



blockMesh or external mesher

Background mesh

Geometry (STL file)

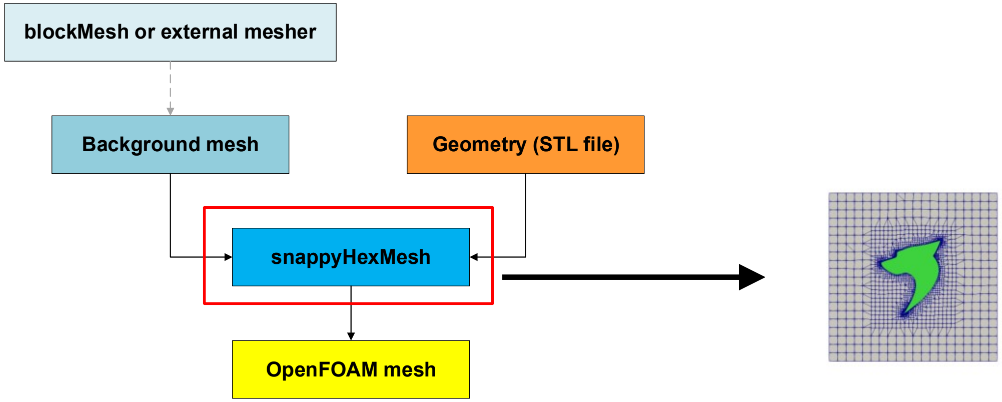
snappyHexMesh

OpenFOAM mesh

Mesh generation using snappyHexMesh

snappyHexMesh workflow

- The meshing utility `snappyHexMesh` reads the dictionary *`snappyHexMeshDict`* located in the directory system.
- The castellation, snapping, and boundary layer meshing steps are controlled by the dictionary *`snappyHexMeshDict`*.
- The final mesh is always located in the directory **`constant/polyMesh`**



blockMesh or external mesher

Background mesh

Geometry (STL file)

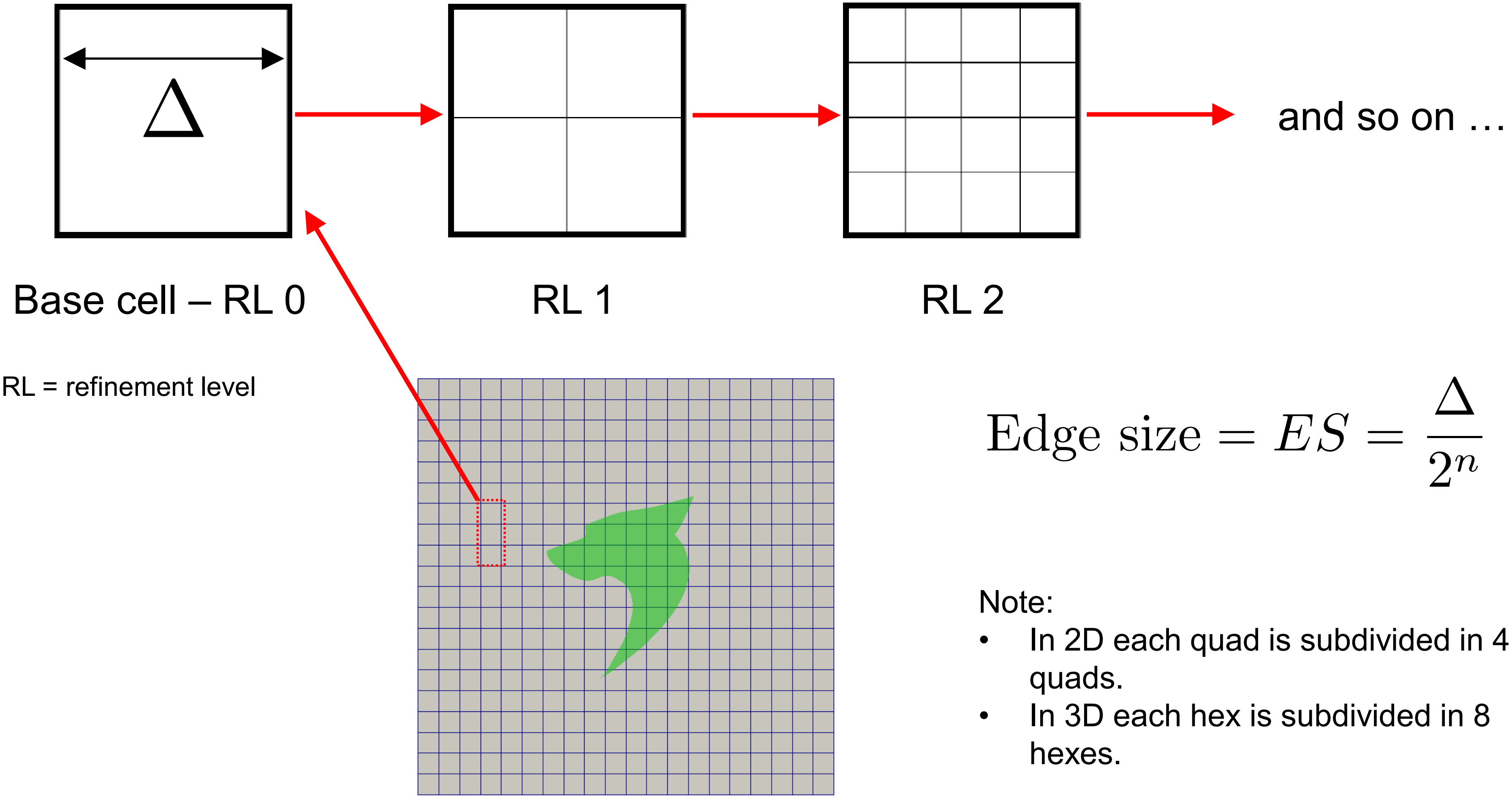
snappyHexMesh

OpenFOAM mesh

Mesh generation using snappyHexMesh

snappyHexMesh workflow

- All the volume and surface refinement is done in reference to the background or base mesh.



* RL = refinement level

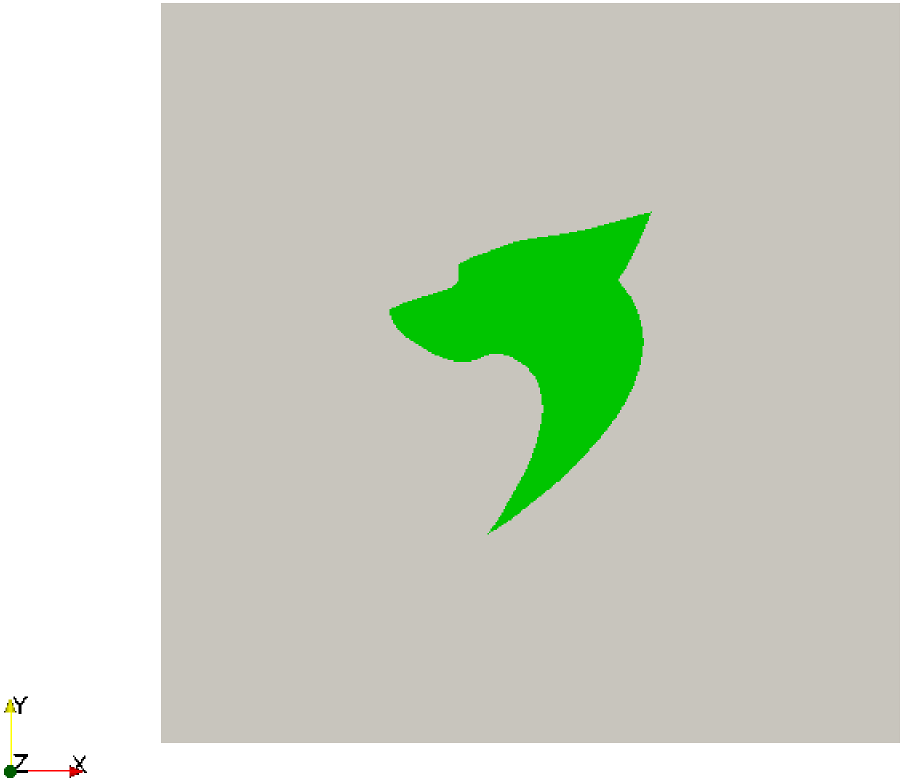
$$\text{Edge size} = ES = \frac{\Delta}{2^n}$$

Note:

- In 2D each quad is subdivided in 4 quads.
- In 3D each hex is subdivided in 8 hexes.

Mesh generation using snappyHexMesh

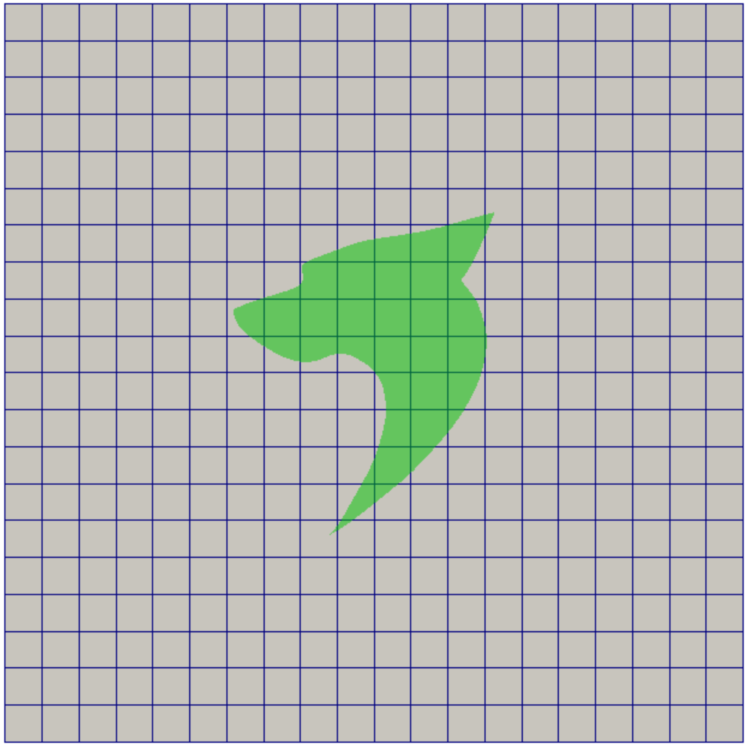
snappyHexMesh workflow



- The process of generating a mesh using `snappyHexMesh` will be described using this figure.
- The objective is to mesh a rectangular shaped region (shaded grey in the figure) surrounding an object described by a STL surface (shaded green in the figure).
- This is an external mesh (*e.g.* for external aerodynamics).
- You can also generate an internal mesh (*e.g.* flow inside a pipe).

Mesh generation using snappyHexMesh

snappyHexMesh workflow

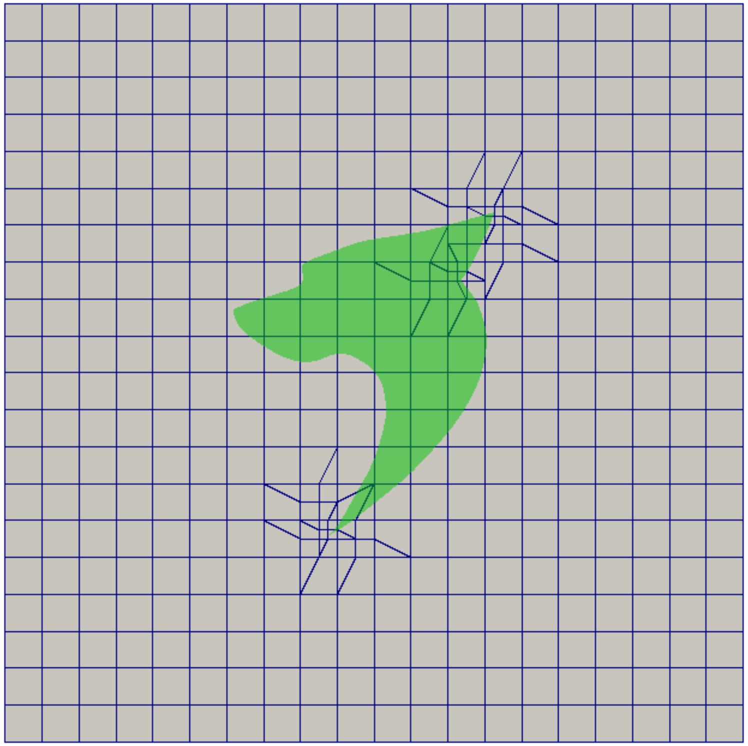


Step 1. Creating the background hexahedral mesh

- Before `snappyHexMesh` is executed the user must create a background mesh of hexahedral cells that fills the entire region as shown in the figure. This can be done by using `blockMesh` or any other mesher.
- The following criteria must be observed when creating the background mesh:
  - The mesh must consist purely of hexes.
  - The cell aspect ratio should be approximately 1, at least near the STL surface.
  - There must be at least one intersection of a cell edge with the STL surface.

Mesh generation using snappyHexMesh

snappyHexMesh workflow

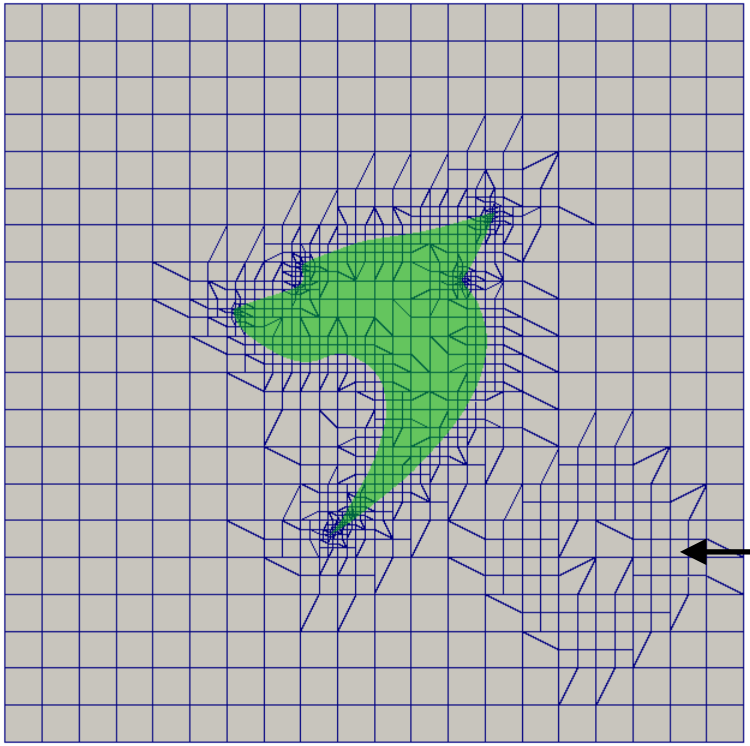


Step 2. Cell splitting at feature edges

- Cell splitting is performed according to the specification supplied by the user in the **castellatedMeshControls** sub-dictionary in the *`snappyHexMeshDict`* dictionary.
- The splitting process begins with cells being selected according to specified edge features as illustrated in the figure.
- The feature edges can be extracted from the STL geometry file using the utility `surfaceFeatures`.

Mesh generation using snappyHexMesh

snappyHexMesh workflow



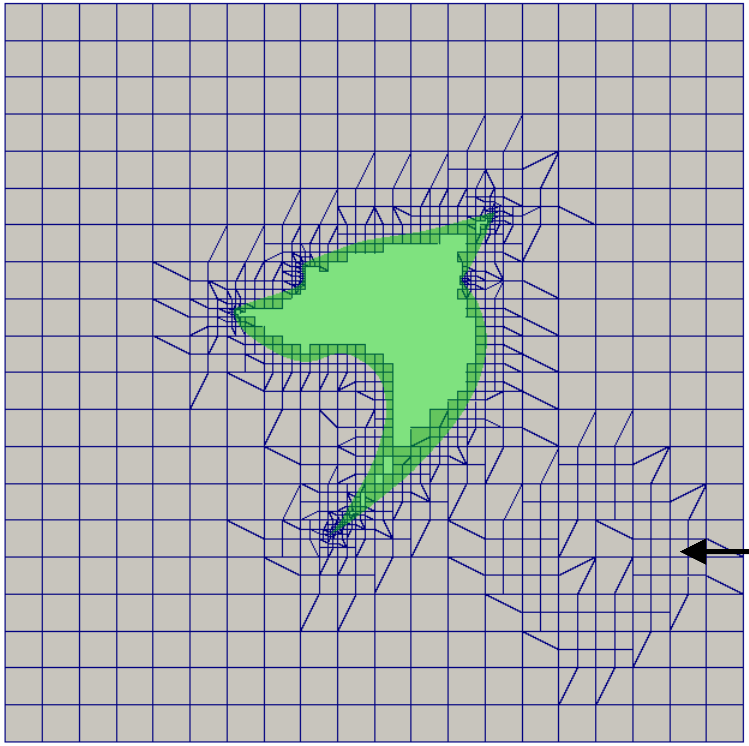
Additional internal cells splitting

Step 3. Cell splitting at surfaces

- Following feature edges refinement, cells are selected for splitting in the locality of specified surfaces as illustrated in the figure.
- The surface refinement (splitting) is performed according to the specification supplied by the user in the **refinementMeshControls** in the **castellatedMeshControls** sub-dictionary in the `snappyHexMeshDict` dictionary.
- Notice that we added additional internal cells splitting. This new cell region can be used to define a source term, or it can be put into motion.

Mesh generation using snappyHexMesh

snappyHexMesh workflow



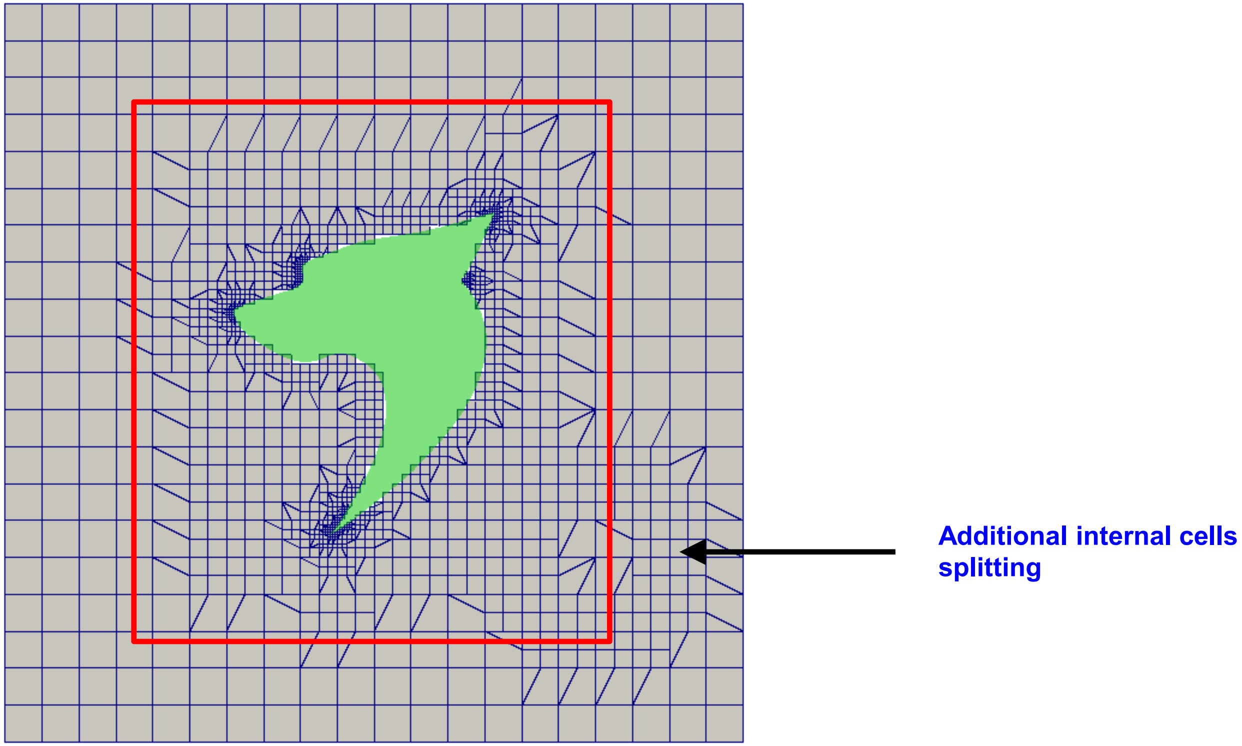
Additional internal cells splitting

Step 4. Cell removal

- Once the feature edges and surface splitting is complete, a process of cell removal begins.
- The region in which cells are retained are simply identified by a location point within the region, specified by the **locationInMesh** keyword in the **castellatedMeshControls** sub-dictionary in the `snappyHexMeshDict` dictionary.
- Cells are retained if, approximately speaking, 50% or more of their volume lies within the region.

Mesh generation using snappyHexMesh

snappyHexMesh workflow

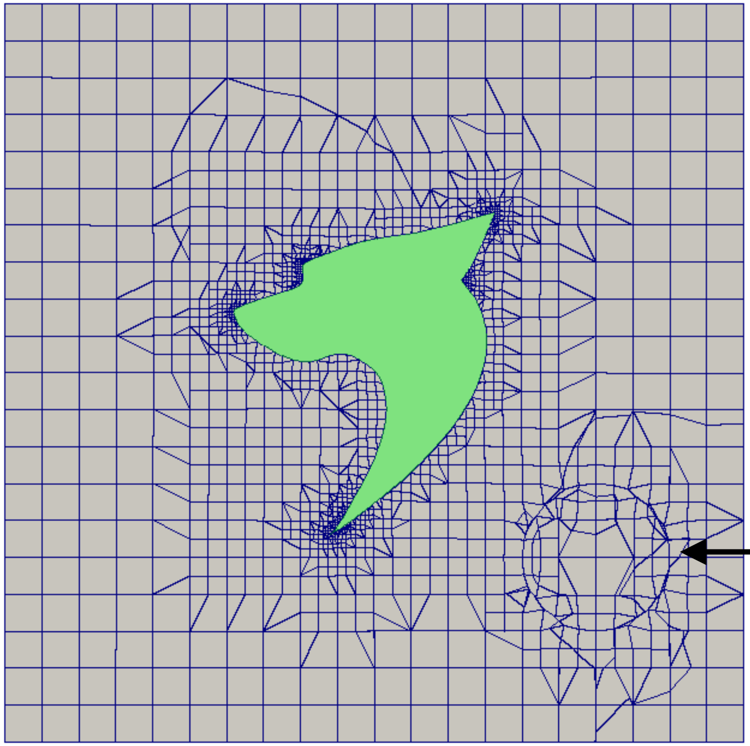


Step 5. Cell splitting in specified regions

- Those cells that lie within one or more specified volume regions can be further split by a region (in the figure, the rectangular region within the red rectangle).
- The information related to the refinement of the volume regions is supplied by the user in the **refinementRegions** block in the **castellatedMeshControls** sub-dictionary in the `snappyHexMeshDict` dictionary.
- This is a valid castellated or cartesian mesh that can be used for a simulation.

Mesh generation using snappyHexMesh

snappyHexMesh workflow



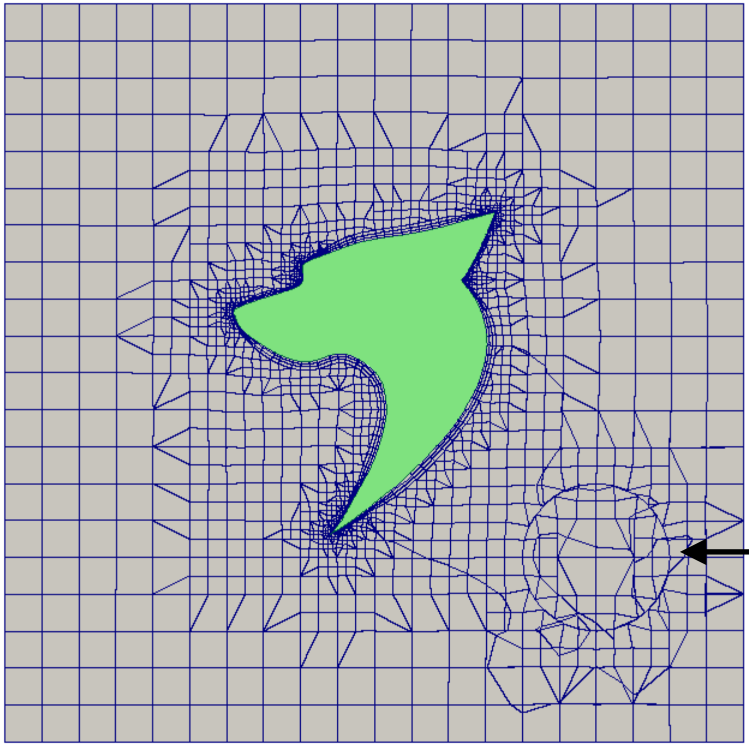
Additional internal cells splitting

Step 6. Snapping to surfaces

- After deleting the cells in the region specified and refining the volume mesh, the points are snapped on the surface to create a conforming mesh.
- The snapping is controlled by the user supplied information in the **snapControls** sub-dictionary in *snappyHexMeshDict*.
- Sometimes, the default **snapControls** options are not enough, so you will need to adjust the values to get a better mesh (not guarantee). It is advisable to save the intermediate steps with a high writing precision (*controlDict*).
- This is a valid snapped or body fitted mesh that can be used for a simulation.

Mesh generation using snappyHexMesh

snappyHexMesh workflow



Additional internal cells splitting

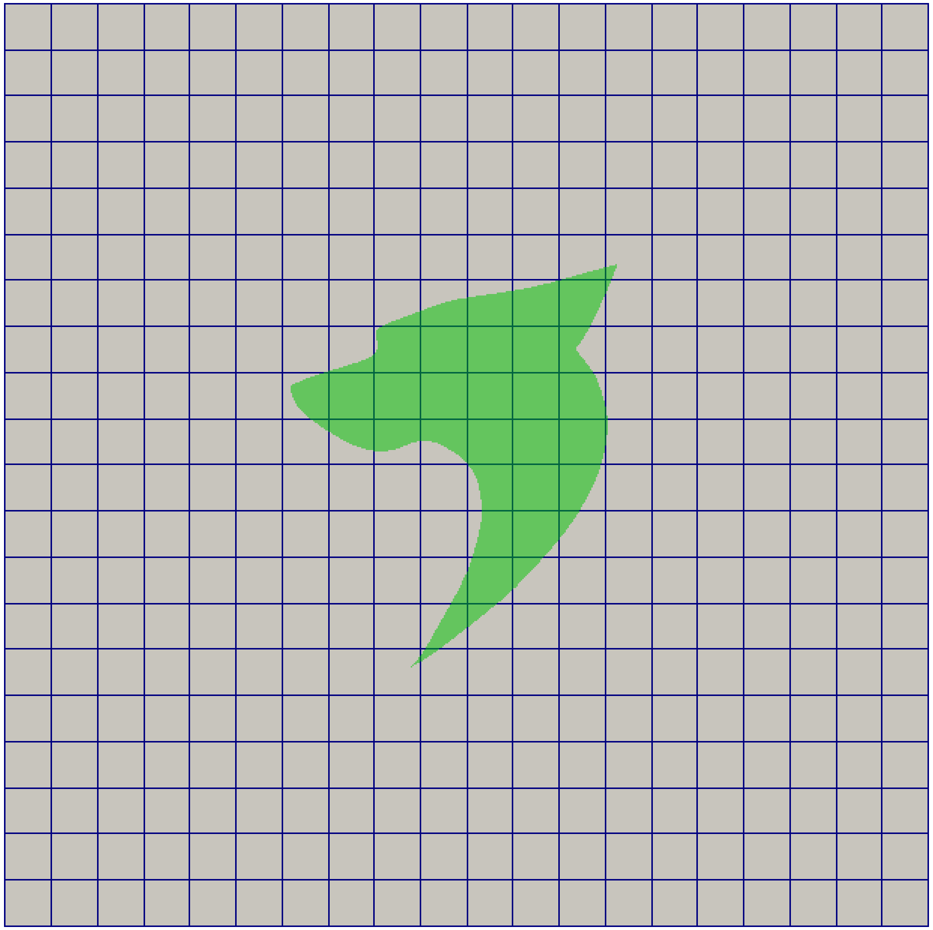
Step 7. Mesh layers

- The mesh output from the snapping stage it is suitable for simulation, although it can produce some irregular cells along boundary surfaces.
- There is an optional stage of the meshing process which introduces boundary layer meshing in selected parts of the mesh.
- This information is supplied by the user in the **addLayersControls** sub-dictionary in the *`snappyHexMeshDict`* dictionary.
- This is the final step of the mesh generation process using `snappyHexMesh`.
- This is a valid body fitted mesh with boundary layer meshing, that can be used for a simulation.

Mesh generation using snappyHexMesh

snappyHexMesh in action

www.wolfdynamics.com/wiki/shm/ani.gif



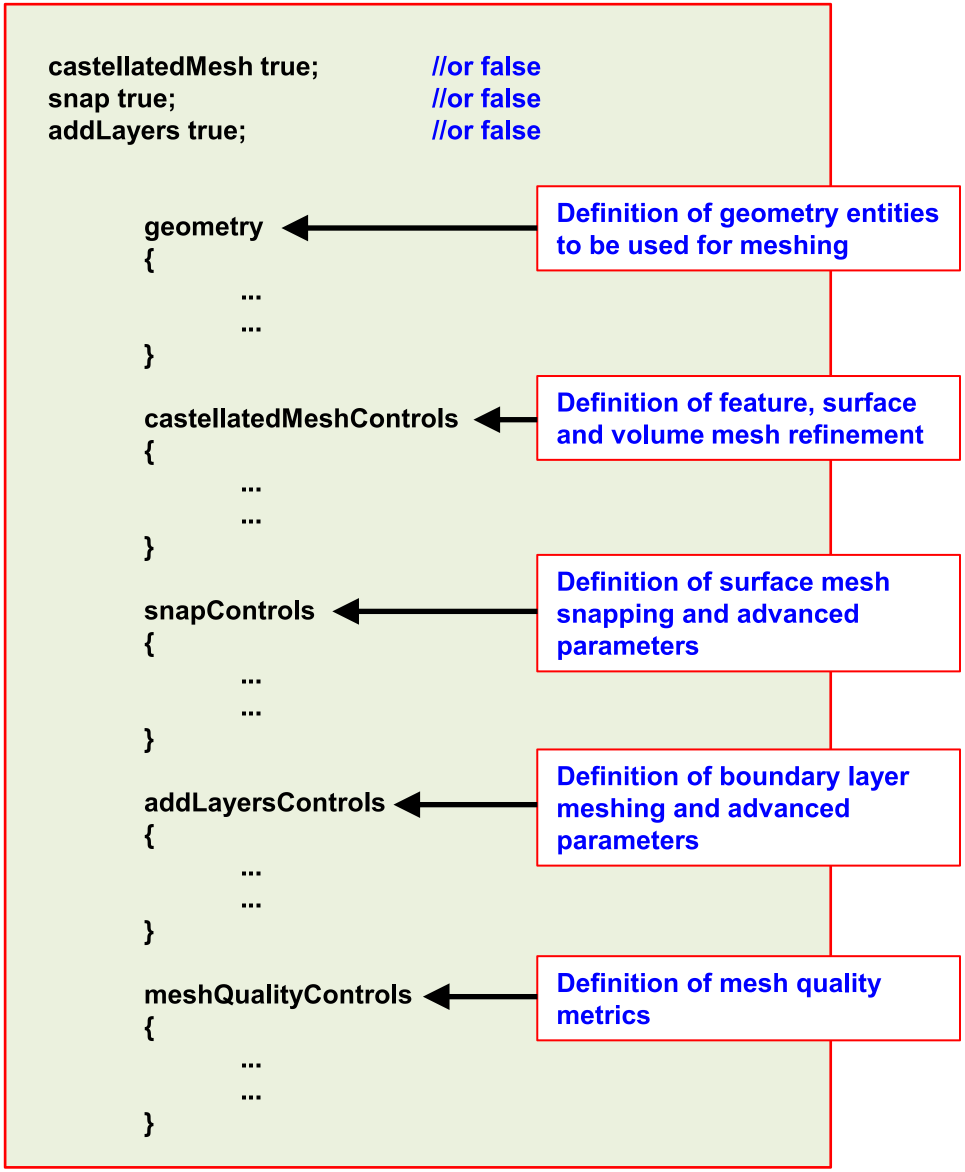
Mesh generation using snappyHexMesh

- Let us study the `snappyHexMesh` dictionary in details.
- We are going to work with the case we just saw in action.
- You will find this case in the directory:

```
$PTOFC/101SHM/M101_WD
```

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.



```
castellatedMesh true;          //or false
snap true;                     //or false
addLayers true;                //or false


        geometry
        {
                ...
                ...
        }

        castellatedMeshControls
        {
                ...
                ...
        }

        snapControls
        {
                ...
                ...
        }

        addLayersControls
        {
                ...
                ...
        }

        meshQualityControls
        {
                ...
                ...
        }
```

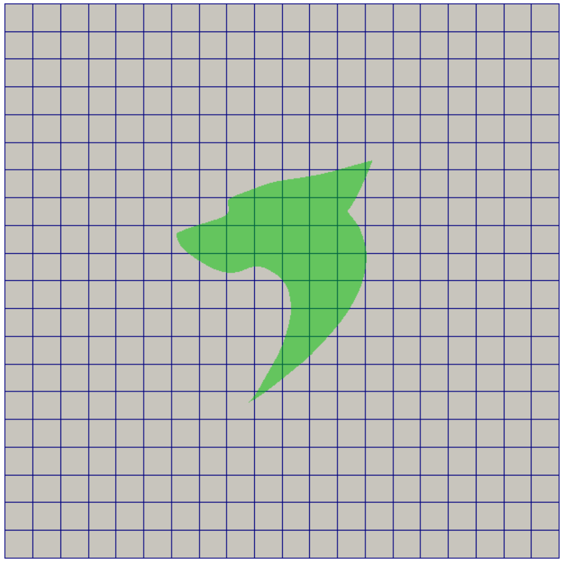
Definition of geometry entities to be used for meshing

Definition of feature, surface and volume mesh refinement

Definition of surface mesh snapping and advanced parameters

Definition of boundary layer meshing and advanced parameters

Definition of mesh quality metrics



- Open the dictionary *snappyHexMeshDict* with your favorite text editor (we will use gedit).
- The `snappyHexMesh` dictionary is made up of five sections, namely: **geometry**, **castellatedMeshControls**, **snapControls**, **addLayersControls** and **meshQualityControls**. Each section controls a step of the meshing process.
- In the first three lines we can turn off and turn on the different meshing steps. For example, if we want to generate a body fitted mesh with no boundary layer we should proceed as follows:

```
castellatedMesh true;
snap true;
addLayers false;
```

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMesh true;            //or false
snap true;                       //or false
addLayers true;                  //or false


        geometry
        {
                ...
                ...
        }

        castellatedMeshControls
        {
                ...
                ...
        }

        snapControls
        {
                ...
                ...
        }

        addLayersControls
        {
                ...
                ...
        }

        meshQualityControls
        {
                ...
                ...
        }
```

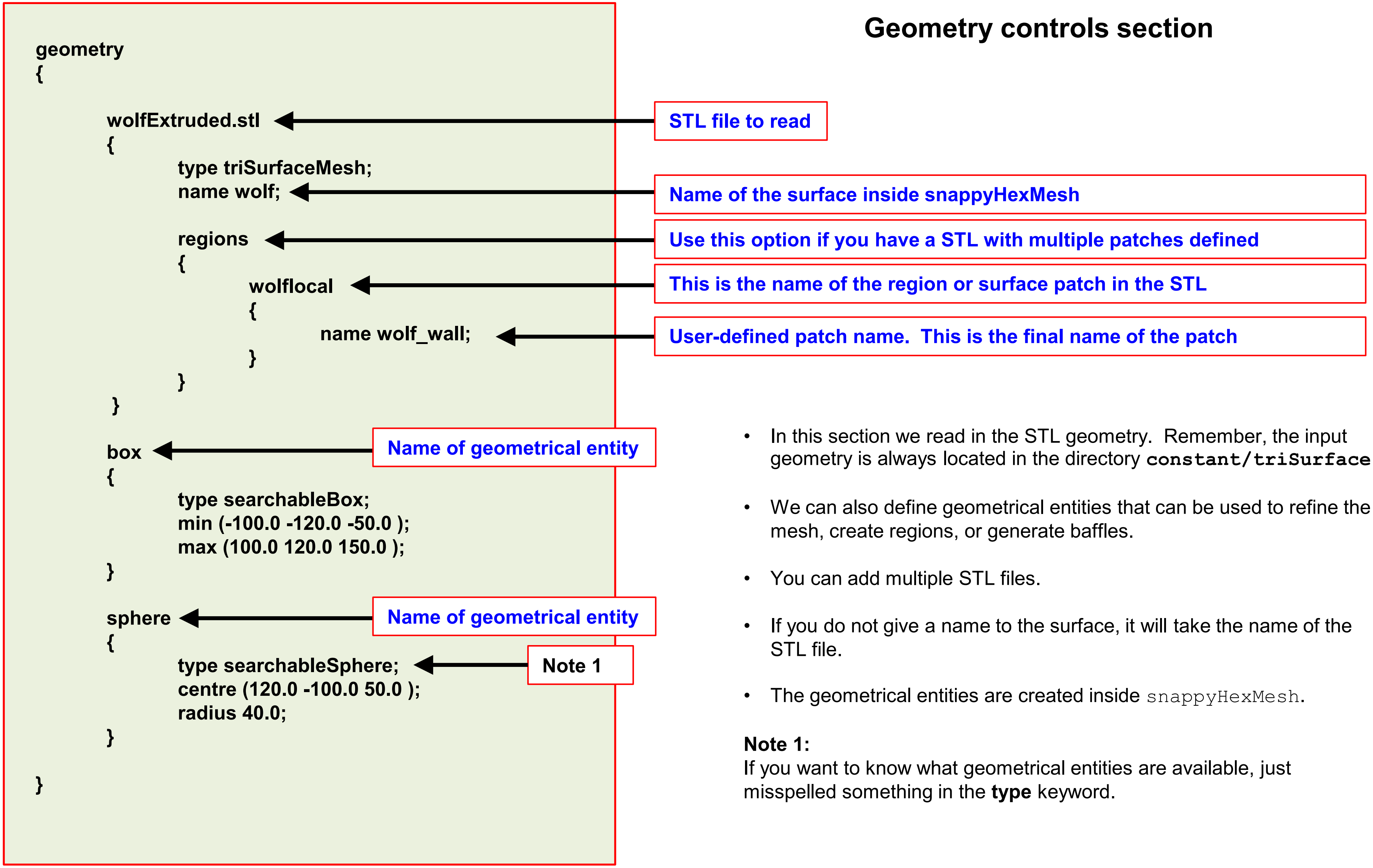
It can be located In a separated file

- Have in mind that there are more than 70 parameters to control in `snappyHexMeshDict` dictionary.
- Adding the fact that there is no native GUI, it can be quite tricky to control the mesh generation process.
- Nevertheless, `snappyHexMesh` generates very good hexa dominant meshes.
- Hereafter, we will only comment on the most important parameters.
- The parameters that you will find in the *`snappyHexMeshDict`* dictionaries distributed with the tutorials, in our opinion are robust and will work most of the times.

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

Geometry controls section



```
geometry
{

          wolfExtruded.stl
          {
                    type triSurfaceMesh;
                    name wolf;

                    regions
                    {
                              wolflocal
                              {
                                        name wolf_wall;
                              }
                    }
          }

          box
          {
                    type searchableBox;
                    min (-100.0 -120.0 -50.0 );
                    max (100.0 120.0 150.0 );
          }

          sphere
          {
                    type searchableSphere;
                    centre (120.0 -100.0 50.0 );
                    radius 40.0;
          }

}
```

STL file to read

Name of the surface inside snappyHexMesh

Use this option if you have a STL with multiple patches defined

This is the name of the region or surface patch in the STL

User-defined patch name. This is the final name of the patch

Name of geometrical entity

Name of geometrical entity

Note 1

- In this section we read in the STL geometry. Remember, the input geometry is always located in the directory `constant/triSurface`
- We can also define geometrical entities that can be used to refine the mesh, create regions, or generate baffles.
- You can add multiple STL files.
- If you do not give a name to the surface, it will take the name of the STL file.
- The geometrical entities are created inside `snappyHexMesh`.

Note 1:
If you want to know what geometrical entities are available, just misspelled something in the **type** keyword.

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{
        //Refinement parameters
        maxLocalCells  100000;
        maxGlobalCells 2000000;
        nCellsBetweenLevels 3;
                ...
                ...

        //Explicit feature edge refinement
        features
        (
                ...
                ...
        );

        //Surface based refinement
        refinementSurfaces
        {
                ...
                ...
        }

        //Region-wise refinement
        refinementRegions
        {
                ...
                ...
        }

        //Mesh selection
        locationInMesh (-100.0 0.0 50.0 );
}
```

Note 1

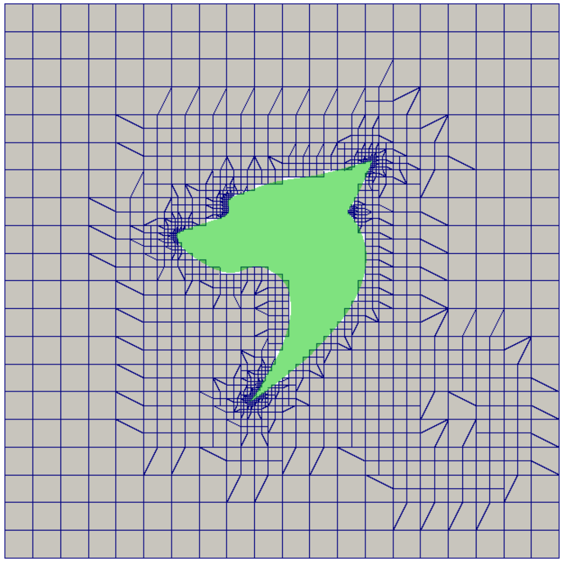
Dictionary block

Dictionary block

Dictionary block

Note 2

Castellated mesh controls section



- In the **castellatedMeshControls** section, we define the global refinement parameters, explicit feature edge refinement, surface-based refinement, region-wise refinement and the material point.
- **In this step, we are generating the castellated mesh.**

Note 1:
Maximum number of cells in the domain. If the mesher reach this number, it will not add more cells.

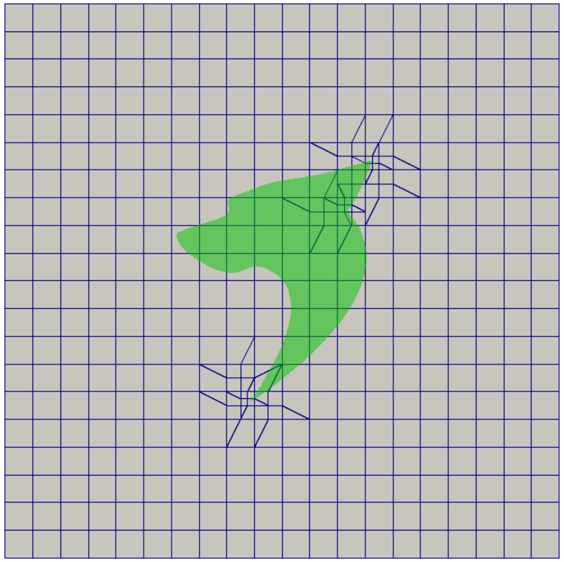
Note 2:
The material point indicates where we want to create the mesh, that is, inside or outside the body to be meshed.

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

        // Refinement parameters
        maxLocalCells  100000;
        maxGlobalCells 2000000;
        minRefinementCells 0;
        maxLoadUnbalance 0.10;
        nCellsBetweenLevels 3;                        Note 1

        //Local curvature and
        //feature angle refinement
        resolveFeatureAngle 30;                       Note 2

        planarAngle 30;

        allowFreeStandingZoneFaces true;


        //Explicit feature edge refinement
        features          Dictionary block
        (
                {
                        file "wolfExtruded.eMesh";                Note 3
                        level 2;
                }
        );

        ...
        ...
        ...

}
```

Castellated mesh controls section



Note 1:
This parameter controls the transition between cell refinement levels.

Note 2:
This parameter controls the local curvature refinement. The higher the value, the less features it captures. For example, if you use a value of 100 it will not add refinement in high curvature areas. It also controls edge feature snapping; high values will not resolve sharp angles in surface intersections.

Note 3:
This file is automatically created when you use the utility `surfaceFeatures`. The file is located in the directory **`constant/triSurface`**

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

    ...
    ...
    ...

    //Surface based refinement
    refinementSurfaces
    {

        //wolf was defined in the geometry section
        wolf
        {

            level (1 1);        //Global refinement

            regions
            {

                wolflocal
                {
                    level (2 4);

                    patchInfo
                    {
                        type wall;

                    }
                }
            }
        }
    ...
    ...
    }
```

Dictionary block → refinementSurfaces

Note 1 → wolf

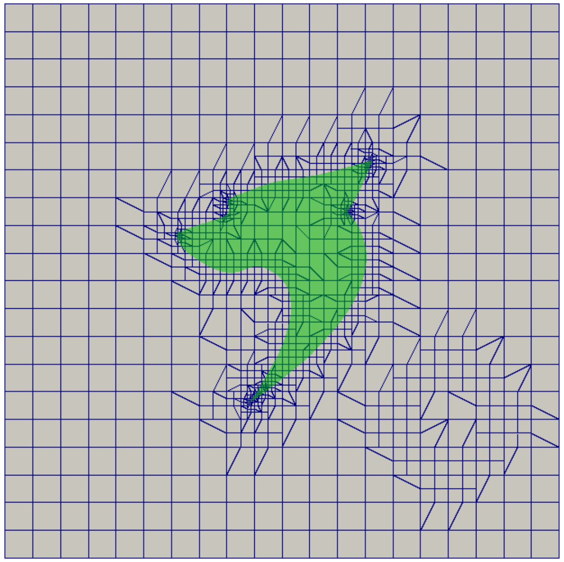
Note 2 → regions

Note 3 → wolflocal

Local refinement → level (2 4);

Note 4 → type wall;

Castellated mesh controls section



Note 1:
The surface wolf was defined in the geometry section.

Note 2:
The region **wolflocal** was defined in the geometry section.

Note 3:
Named region in the STL file. This refinement is local.
To use the surface refinement in the regions, the local regions must exist in STL file. We created a pointer to this region in the **geometry** section.

Note 4:
You can only define patches of type wall or patch.

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

          //Surface based refinement
          refinementSurfaces
          {

          ...
          ...
          ...
                    //This surface or geometrical entity
                    //was defined in geometry section
                    sphere
                    {
                              level (1 1);

                              faceZone face_inner;
                              cellZone cell_inner;

                              cellZoneInside inside;

                              //faceType internal;

                    }

          }

          ...
          ...
}
```

Dictionary block

Note 1

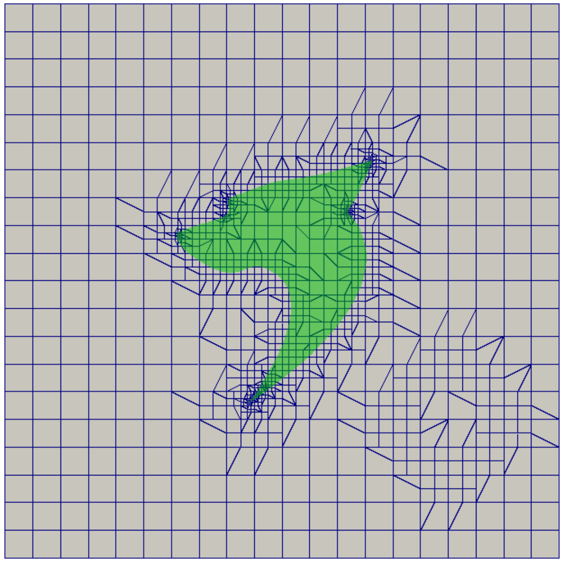
Name of faceZone

Name of cellZone

Create inner cellZone

Create internal faces from faceZone
Uncomment to create the internal faceZone

Castellated mesh controls section



Note 1:
Optional specification of what to do with **faceZone** faces:

internal: keep them as internal faces (default)
baffle: create baffles from them. This gives more freedom in mesh motion
boundary: create free-standing boundary faces (baffles but without the shared points)

e.g., **faceType** internal;

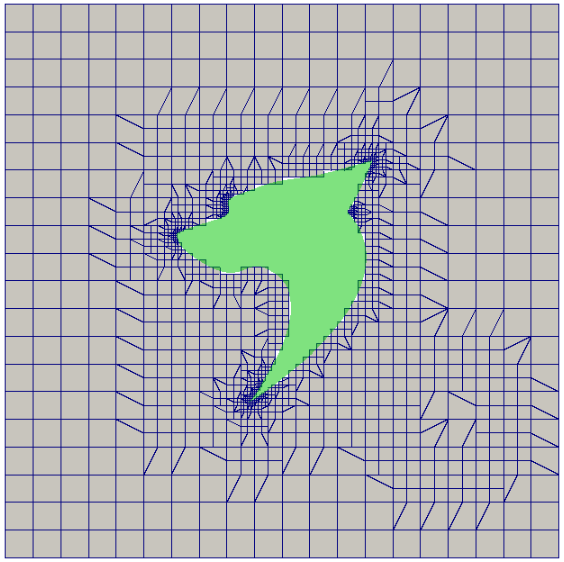
Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

        ...
        ...
        ...

        //Region-wise refinement
        refinementRegions
        {

                //This region or geometrical entity
                //was defined in the geometry section

                box
                {
                        mode inside;
                        levels  (( 1 1 ));
                }

        }


        //Mesh selection
        locationInMesh (-100.0 0.0 50.0 );

}
```

Dictionary block

Note 1

Castellated mesh controls section



Note 1:
This region or geometrical entity was created in the **geometry** section.

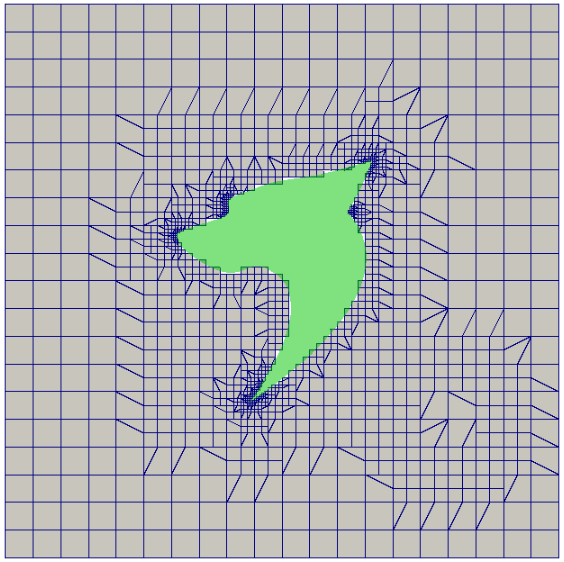
Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
castellatedMeshControls
{

          ...
          ...
          ...

          //Region-wise refinement
          refinementRegions
          {

                    //This region or geometrical entity
                    //was defined in the geometry section

                    box
                    {
                              mode inside;
                              levels  (( 1 1 ));
                    }

          }


          //Mesh selection
          locationInMesh (-100.0 0.0 50.0 );

}
```

Dictionary block

Castellated mesh controls section



This point defines where do you want the mesh.
Can be internal mesh or external mesh.

- If the point is inside the STL it is an internal mesh.
- If the point is inside the background mesh and outside the STL it is an external mesh.

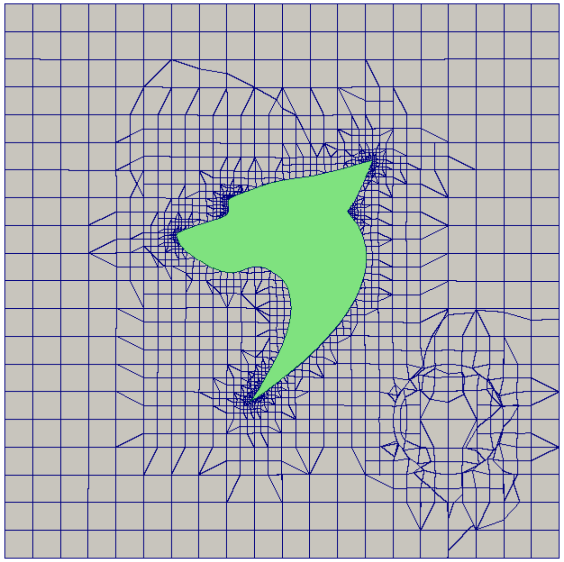
At this point we have a valid mesh (cartesian)

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
snapControls
{

        //Number of patch smoothing iterations
        //before finding correspondence to surface
        nSmoothPatch 3;

        tolerance 2.0;

        //- Number of mesh displacement relaxation
        //iterations.
        nSolveIter 100;                 Note 1

        //- Maximum number of snapping relaxation
        //iterations. Should stop before upon
        //reaching a correct mesh.
        nRelaxIter 10;                  Note 2

        // Feature snapping

                //Number of feature edge snapping iterations.
                nFeatureSnapIter 10;                          Note 3

                //Detect (geometric only) features by
                //sampling the surface (default=false).
                implicitFeatureSnap false;

                // Use castellatedMeshControls::features
                // (default = true)
                explicitFeatureSnap true;

                multiRegionFeatureSnap false;

}
```

Snap mesh controls section



Note 1:
The higher the value the better the body fitted mesh. The default value is 30. If you are having problems with the mesh quality (related to the snapping step), try to increase this value to 300. Have in mind that this will increase the meshing time.

Note 2:
Increase this value to improve the quality of the body fitted mesh.

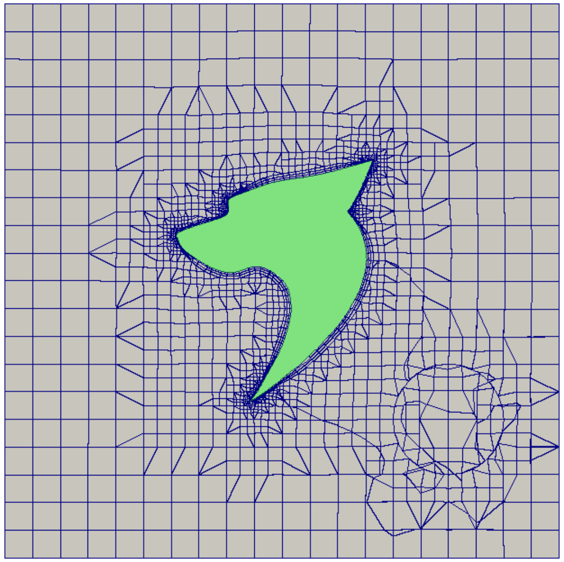
Note 3:
Increase this value to improve the quality of the edge features.

- **In this step, we are generating the body fitted mesh.**

Let us explore the snappyHexMeshDict dictionary.

```
addLayersControls
{
        //Global parameters
        relativeSizes true;
        expansionRatio 1.2;
        finalLayerThickness 0.5;
        minThickness 0.01;

        layers              ←  Note 1
        {
                wolf_wall   ←  Note 2
                {
                        nSurfaceLayers 3;
                        //Local parameters
                        //expansionRatio        1.3;
                        //finalLayerThickness   0.3;
                        //minThickness          0.1;
                }
        }
        // Advanced settings
        nGrow 0;
        featureAngle 130;   ←  Note 3
        maxFaceThicknessRatio 0.5;
        nSmoothSurfaceNormals 1;
        nSmoothThickness 10;
        minMedianAxisAngle 90;
        maxThicknessToMedialRatio 0.3;
        nSmoothNormals 3;
        slipFeatureAngle 30;
        nRelaxIter 5;
        nBufferCellsNoExtrude 0;
        nLayerIter 50;
        nRelaxedIter 20;
}
```

Boundary layer mesh controls section



Note 1:
In this section we select the patches where we want to add the layers. We can add multiple patches (if they exist).

Note 2:
This patch was created in the **geometry** section.

Note 3:
Specification of feature angle above which layers are collapsed automatically.

- **In this step, we are generating the boundary layer mesh.**

Mesh generation using snappyHexMesh

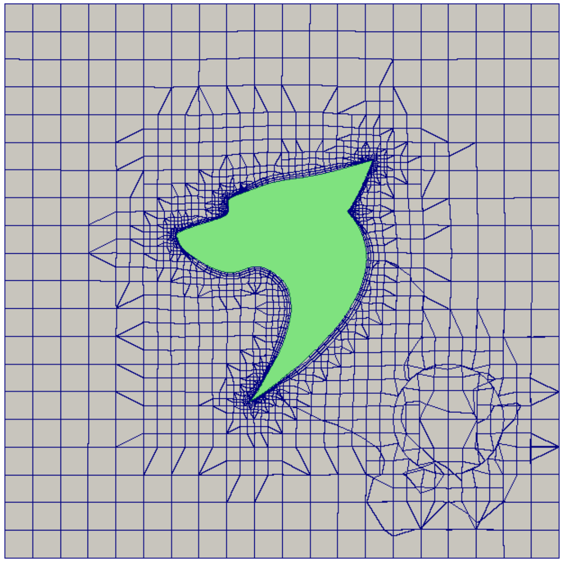
Let us explore the snappyHexMeshDict dictionary.

```
meshQualityControls
{
        maxNonOrtho 75;

        maxBoundarySkewness 20;

        maxInternalSkewness 4;

        maxConcave 80;

        minVol 1E-13;

        //minTetQuality  1e-15;
        minTetQuality -1e+30;

        minArea -1;

        minTwist 0.02;

        minDeterminant 0.001;

        minFaceWeight 0.05;

        minVolRatio 0.01;

        minTriangleTwist -1;

        minFlatness 0.5;

        nSmoothScale 4;

        errorReduction 0.75;
}
```

Note 1

Note 2

Mesh quality controls section



Note 1:
Maximum non-orthogonality angle.

Note 2:
Maximum skewness angle.

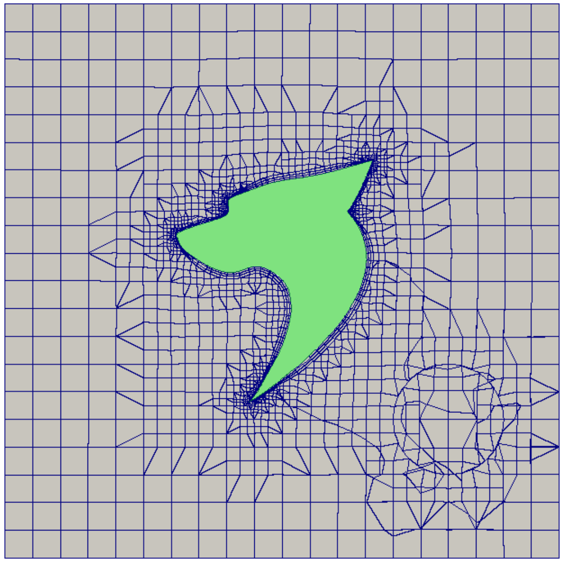
- During the mesh generation process, the mesh quality is continuously monitored.
- The mesher `snappyHexMesh` will try to generate a mesh using the mesh quality parameters defined by the user.
- If a mesh motion or topology change introduces a poor quality cell or face the motion or topology change is undone to revert the mesh back to a previously valid error free state.

Mesh generation using snappyHexMesh

Let us explore the snappyHexMeshDict dictionary.

```
debugFlags
(
        // write intermediate meshes
        mesh

        // write current mesh intersections as .obj files
        intersections

        // write information about explicit feature edge
        // refinement
        featureSeeds

        // write attraction as .obj files
        attraction

        // write information about layers
        layerInfo
);


writeFlags
(
        // write volScalarField with cellLevel for
        // postprocessing
        scalarLevels

        // write cellSets, faceSets of faces in layer
        layerSets

        // write volScalarField for layer coverage
        layerFields
);
```

Mesh debug and write controls sections



- At the end of the dictionary you will find the sections: **debugFlags** and **writeFlags**
- By default they are commented. If you uncomment them you will enable debug information.
- **debugFlags** and **writeFlags** will produce a lot of outputs that you can use to post process and troubleshoot the different steps of the meshing process.

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- This tutorial is located in the directory:
  - **`$PTOFC/101SHM/M101_WD`**

- In this case we are going to generate a body fitted mesh with boundary layer. This is an external mesh.
- Before generating the mesh take a look at the dictionaries and files that will be used.
- These are the dictionaries and files that will be used.
  - *`system/snappyHexMeshDict`*
  - *`system/surfaceFeaturesDict`*
  - *`system/meshQualityDict`*
  - *`system/blockMeshDict`*
  - *`constant/triSurface/wolfExtruded.stl`*
  - *`constant/triSurface/wolfExtruded.eMesh`*

- The file *`wolfExtruded.eMesh`* is generated after using the utility `surfaceFeatures`, which reads the dictionary *`surfaceFeaturesDict`*.

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- To generate the mesh, in the terminal window type:

```
$> foamCleanTutorials
$> blockMesh
$> surfaceFeatures
$> snappyHexMesh
$> checkMesh -latestTime
```

- To visualize the mesh, in the terminal window type:
  - `$> paraFoam`

- Remember to use the VCR controls in paraView/paraFoam to visualize the mesh intermediate steps.

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- In the case directory you will find the time folders **`1`**, **`2`**, and **`3`**, which contain the castellated mesh, snapped mesh and boundary layer mesh respectively. In this case, `snappyHexMesh` automatically saved the intermediate steps.
- Before running the simulation, remember to transfer the solution from the latest mesh to the directory **`constant/polyMesh`**, in the terminal type:

```
1. $> cp 3/polyMesh/* constant/polyMesh
2. $> rm -rf 1
3. $> rm -rf 2
4. $> rm -rf 3
5. $> checkMesh -latestTime
```

Mesh generation using snappyHexMesh

Let us generate the mesh of the wolf dynamics logo.

- If you want to avoid the additional steps of transferring the final mesh to the directory **`constant/polyMesh`** by not saving the intermediate steps, you can proceed as follows:

  - `$> snappyHexMesh -overwrite`

- When you proceed in this way, `snappyHexMesh` automatically saves the final mesh in the directory **`constant/polyMesh`**.
- Have in mind that you will not be able to visualize the intermediate steps.
- Also, you will not be able to restart the meshing process from a saved state (castellated or snapped mesh).
- Unless it is strictly necessary, from this point on we will not save the intermediate steps.

The *constant/polyMesh/boundary* file

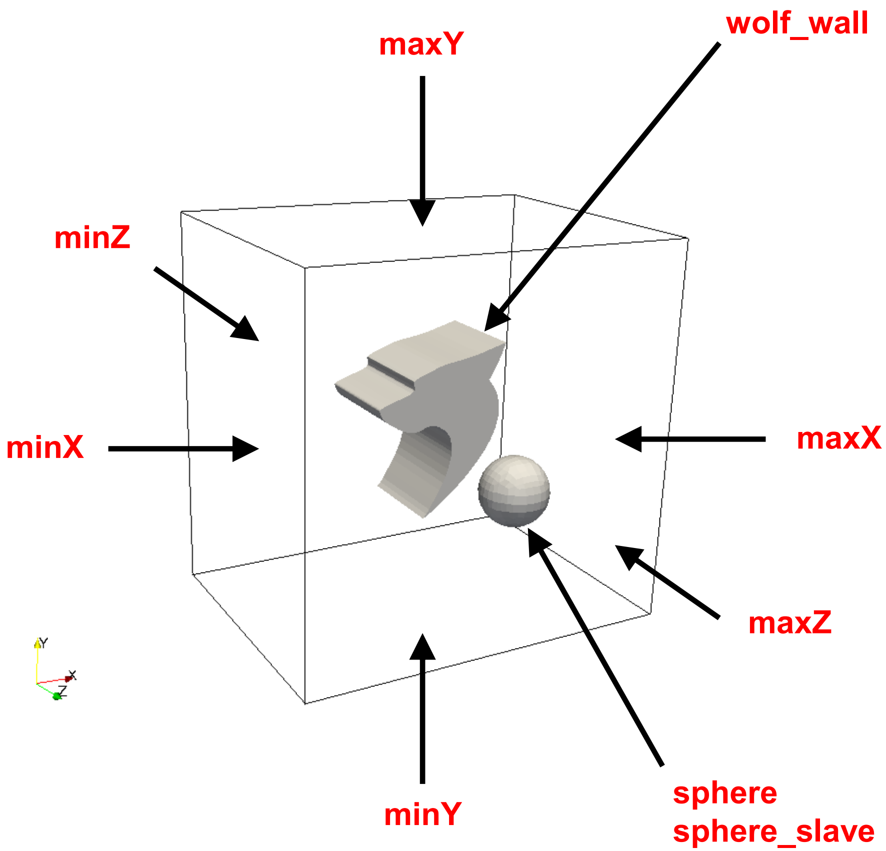
- At this point, we have a valid mesh to run a simulation.
- Have in mind that before running the simulation you will need to set the boundary and initial conditions in the directory **0**.
- Let us talk about the *constant/polyMesh/boundary* file,
  - First of all, this file is automatically generated after you create the mesh or you convert it from a third-party format.
  - In this file, the geometrical information related to the **base type** patch of each boundary of the domain is specified.
  - The **base type** boundary condition is the actual surface patch where we are going to apply a **numerical type** boundary condition.
  - The **numerical type** boundary condition assign a field value to the surface patch (**base type**).
  - You define the **numerical type** patch (or the value of the boundary condition), in the directory **0** or time directories.
  - The **name** and **base type** of the patches was defined in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
  - You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
  - You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch**.

Mesh generation using snappyHexMesh

The *constant/polyMesh/boundary* file

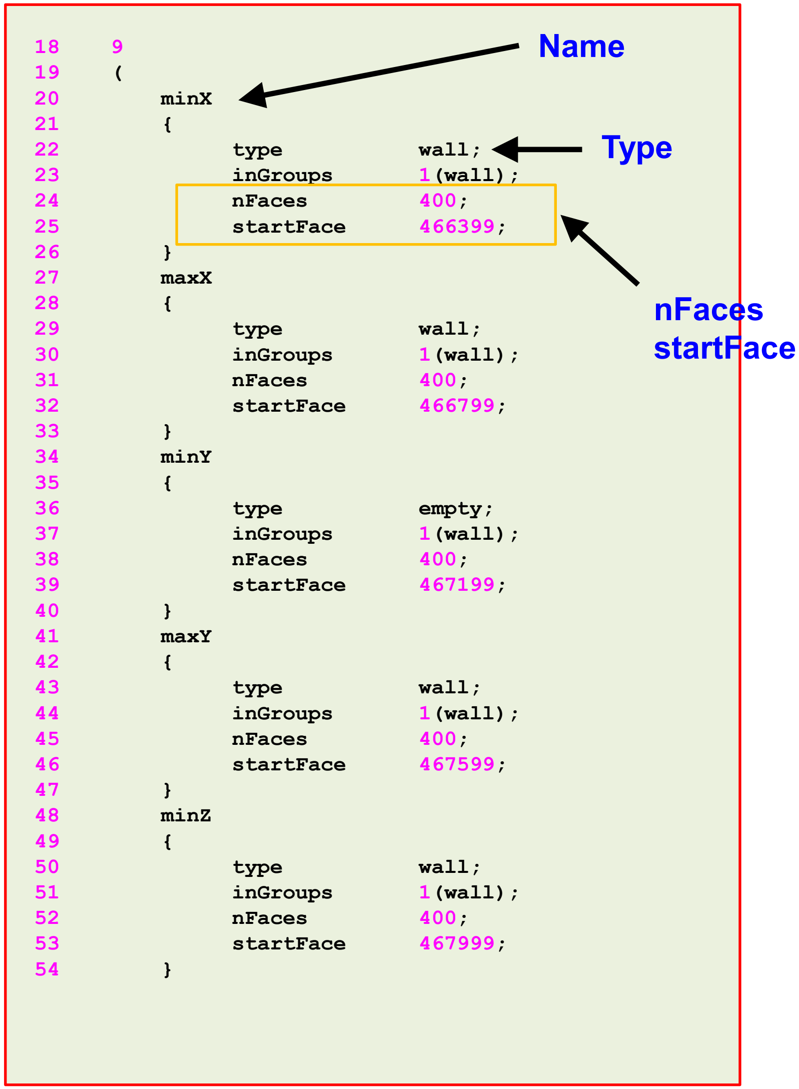
```
9
(
    minX
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       466399;
    }
    maxX
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       466799;
    }
    minY
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       467199;
    }
    maxY
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       467599;
    }
    minZ
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       467999;
    }
```

Number of surface patches
In the list bellow there must be 9 patches definition.



Mesh generation using snappyHexMesh

The *constant/polyMesh/boundary* file



```
9
(
    minX
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       466399;
    }
    maxX
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       466799;
    }
    minY
    {
        type            empty;
        inGroups        1(wall);
        nFaces          400;
        startFace       467199;
    }
    maxY
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       467599;
    }
    minZ
    {
        type            wall;
        inGroups        1(wall);
        nFaces          400;
        startFace       467999;
    }
```

Name and type of the surface patches

- The **name** and **base type** of the patch is given by the user.
- In this case the **name** and **base type** was assigned in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch**.

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information.** ⚠
- Basically, this is telling you the starting face and ending face of the patch.
- This information is created automatically when generating the mesh or converting the mesh.

Mesh generation using snappyHexMesh

The *constant/polyMesh/boundary* file

```
        maxZ
        {
            type            wall;
            inGroups        1(wall);
            nFaces          400;
            startFace       466399;
        }
        wolf_wall
        {
            type            wall;
            inGroups        1(wall);
            nFaces          400;
            startFace       466799;
        }
        sphere
        {
            type            empty;
            inGroups        1(wall);
            nFaces          400;
            startFace       467199;
        }
        sphere_slave
        {
            type            wall;
            inGroups        1(wall);
            nFaces          400;
            startFace       467599;
        }
    )
```

Name

Type

nFaces
startFace

Name and type of the surface patches

- The **name** and **base type** of the patch is given by the user.
- In this case the **name** and **base type** was assigned in the dictionaries *blockMeshDict* and *snappyHexMeshDict*.
- You can change the **name** if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **maxY** from **wall** to **patch**.

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information.**
- Basically, this is telling you the starting face and ending face of the patch.
- This information is created automatically when generating the mesh or converting the mesh.

Cleaning the case directory

- When generating the mesh using OpenFOAM®, it is extremely important to start from a clean case directory.

- To clean all the case directory, in the terminal type:

  - `$> foamCleanTutorials`

- To only erase the mesh information, in the terminal type:

  - `$> foamCleanPolyMesh`

- If you are planning to start the meshing from a previous saved state, you do not need to clean the case directory.

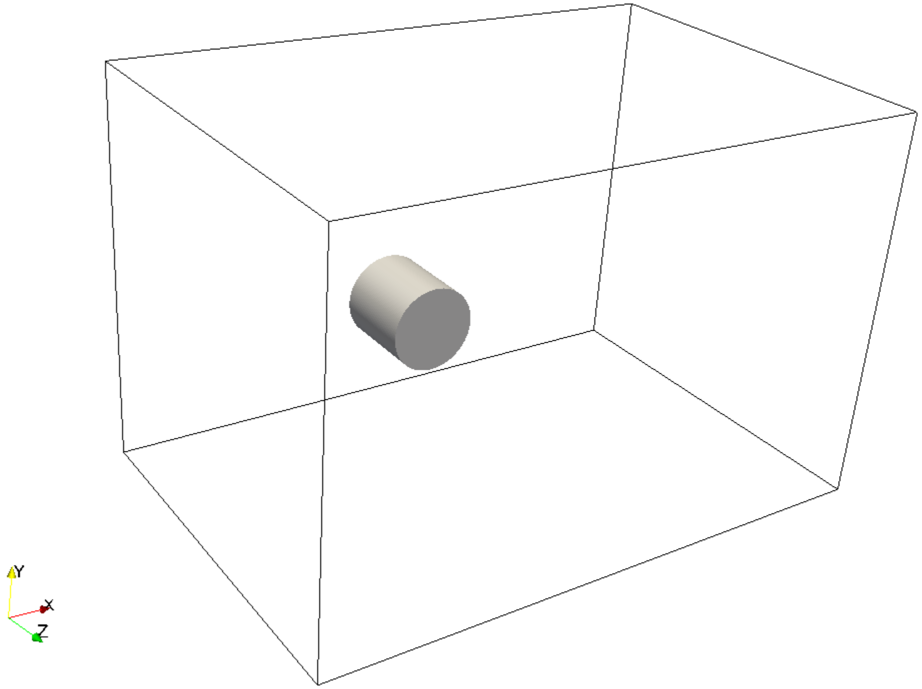
- Before proceeding to compute the solution, remember to always check the quality of the mesh.

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. ~~Mesh generation using snappyHexMesh.~~
6. **snappyHexMesh guided tutorials.**
7. Mesh conversion
8. Geometry and mesh manipulation utilities

snappyHexMesh guided tutorials

- Our first case will be a mesh around a cylinder.
- This is a simple geometry, but we will use it to study all the meshing steps and introduce a few advanced features.
- This case is located in the directory **`$PTOFC/101SHM/M1cyl`**



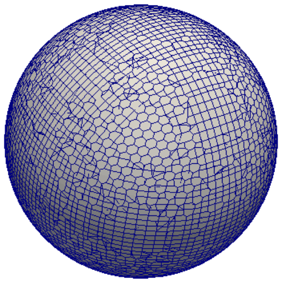
snappyHexMesh guided tutorials

- Meshing with snappyHexMesh – Case 1.
- 3D cylinder with feature edge refinement (external mesh).
- You will find this case in the directory:

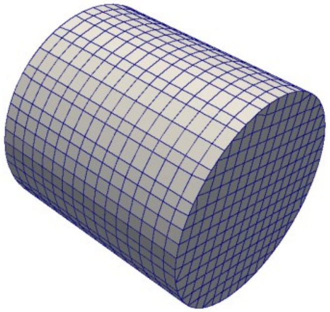
```
$PTOFC/101SHM/M1_cyl/C1
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

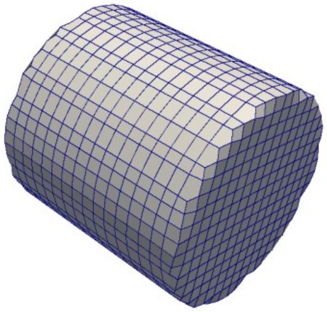
3D Cylinder with edge refinement.



Sphere with no edge refinement



Cylinder with edge refinement



Cylinder with no edge refinement

- If the geometry has sharp angles and you want to resolve those edges, you should use edge refinement.
- In the left figure there is no need to use edge refinement as there are no sharp angles.
- In the mid figure we used edge refinement to resolve the sharp angles.
- In the right figure we did not use edge refinement, therefore we did not resolve well the sharp angles.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

- How do we control curvature refinement and enable edge refinement?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{

          ...
          ...
          ...

          //Local curvature and
          //feature angle refinement
          resolveFeatureAngle 30;

          ...
          ...
          ...

          //Explicit feature edge refinement
          features
          (
                    {
                              file “surfacemesh.eMesh";
                              level 0;
                    }
          );

          ...
          ...
          ...

}
```

To control curvature refinement

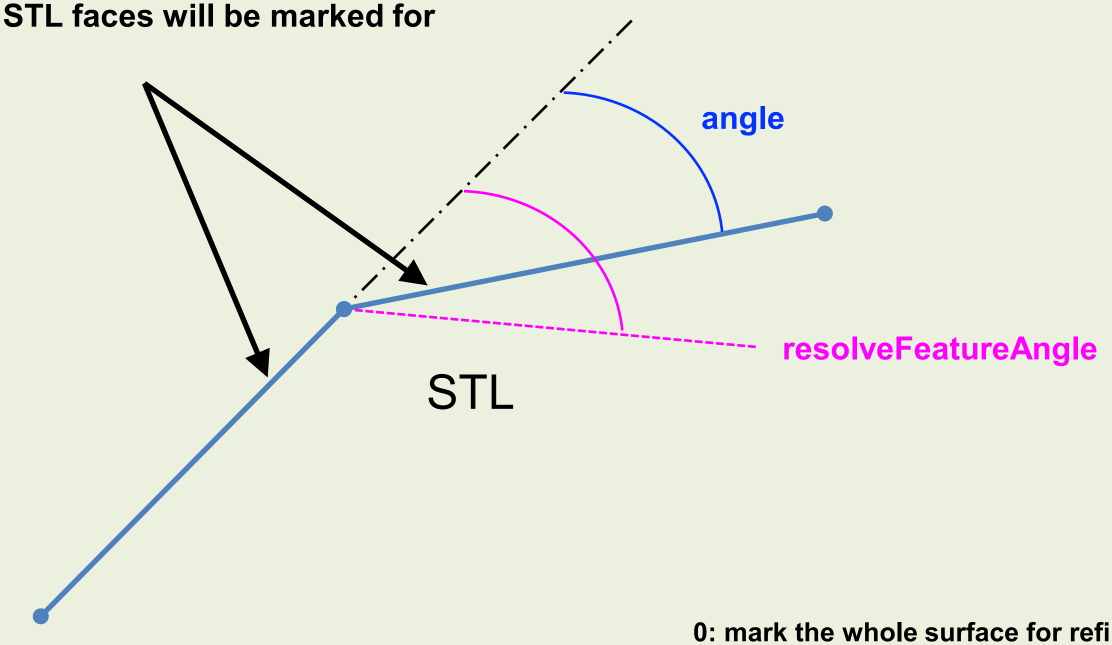
To enable and control edge refinement level

3D Cylinder with edge refinement.

How **resolveFeatureAngle** works?

angle < resolveFeatureAngle
No curvature refinement

If angle is more than resolveFeatureAngle the adjacent STL faces will be marked for refinement



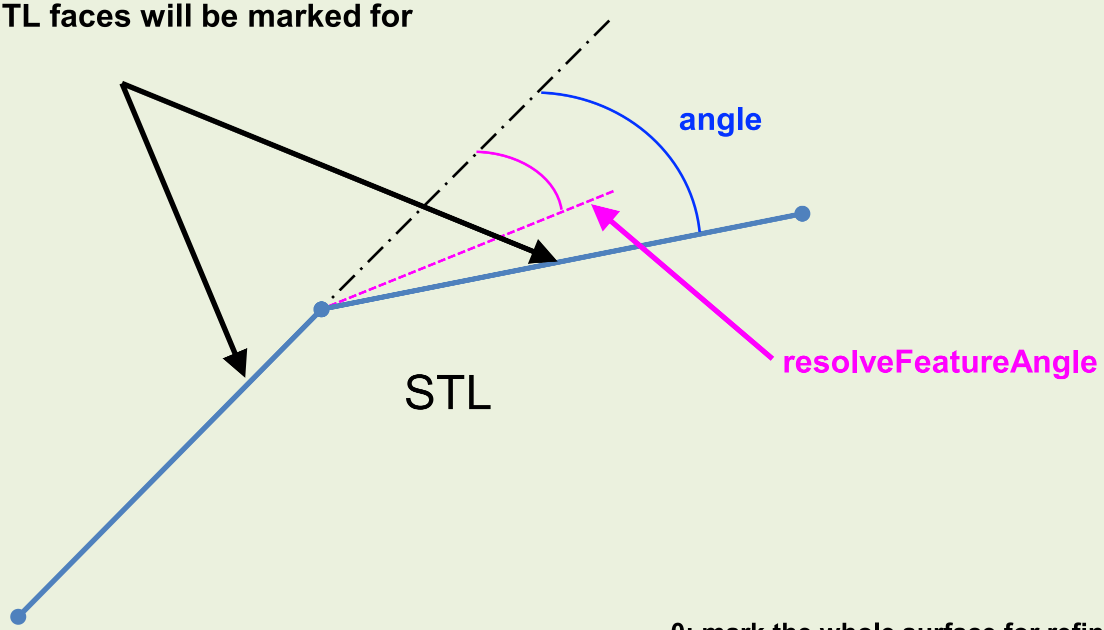
0: mark the whole surface for refinement
180: do not mark any STL face for refinement

3D Cylinder with edge refinement.

How **resolveFeatureAngle** works?

angle > resolveFeatureAngle
Curvature refinement

If angle is more than resolveFeatureAngle the adjacent STL faces will be marked for refinement



0: mark the whole surface for refinement
180: do not mark any STL face for refinement

3D Cylinder with edge refinement.

- How do we control surface refinement?
- In the file *snappyHexMeshDict*, look for the following entry:

```
castellatedMeshControls
{
                    ...
                    ...
                    ...

          //Surface based refinement
          refinementSurfaces
          {
                    banana_stlSurface
                     {
                              level (2 4);
                     }
          }

                    ...
                    ...
                    ...

}
```

To control surface refinement.
The first digit controls the global surface refinement level and the second digit controls the curvature refinement level, according to the angle set in the entry resolveFeatureAngle

3D Cylinder with edge refinement.

- How do we create refinement regions?
- In the file *snappyHexMeshDict*, look for the following entry:

```
geometry
{
        ...
        ...
        ...

        refinementBox
        {
                type searchableBox;
                min  ( -2 -2 -2);
                max (  2   2  2);
        }

        ...
        ...
        ...
};
```

Name of refinement region

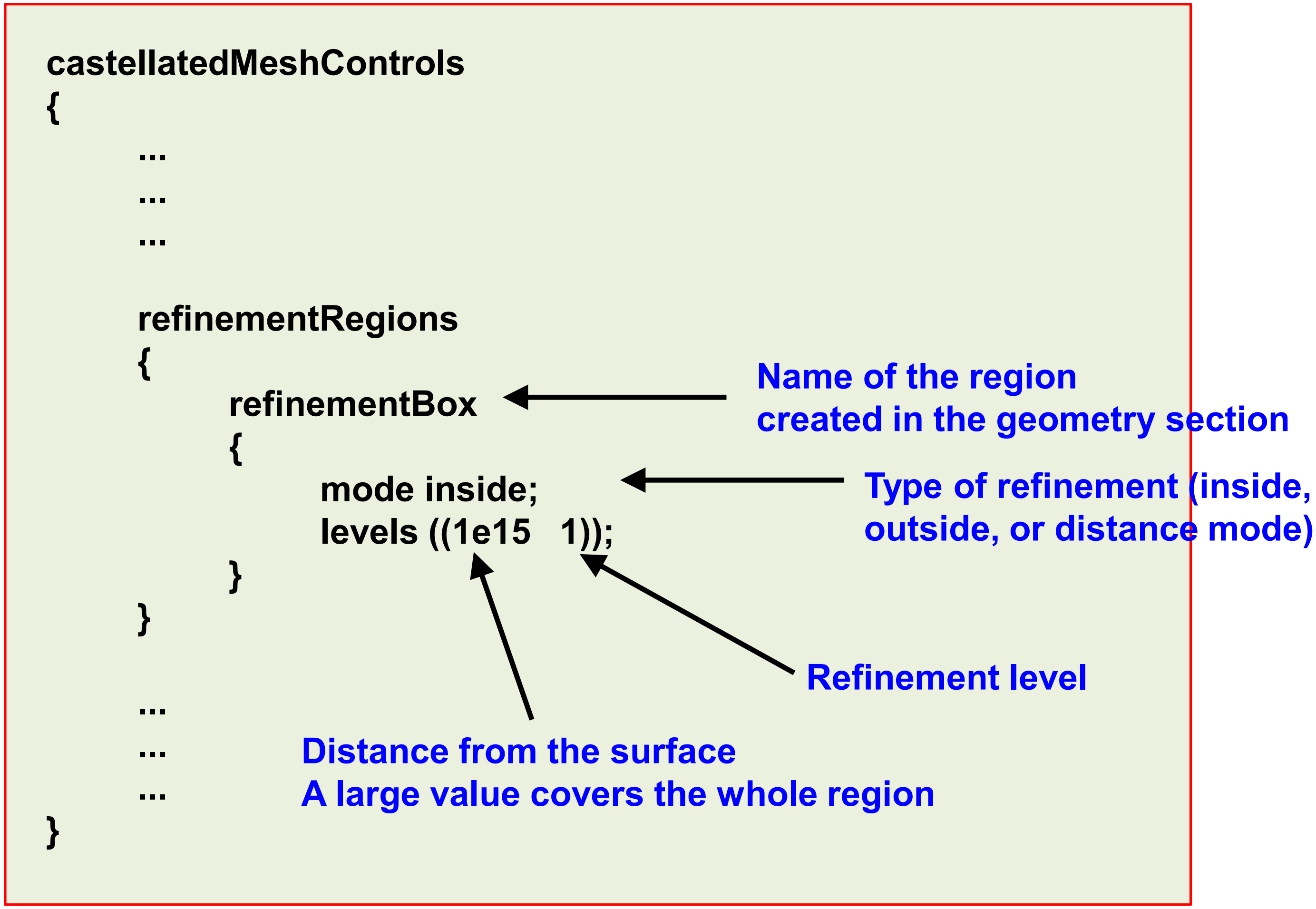
Geometrical entity type.
This is the zone where we
want to apply the refinement

Dimensions of geometrical entity

snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

- How do we create refinement regions?
- In the file *snappyHexMeshDict*, look for the following entry:



```
castellatedMeshControls
{
        ...
        ...
        ...

        refinementRegions
        {
                refinementBox
                {
                        mode inside;
                        levels ((1e15   1));
                }
        }

        ...
        ...
        ...
}
```

Name of the region created in the geometry section

Type of refinement (inside, outside, or distance mode)

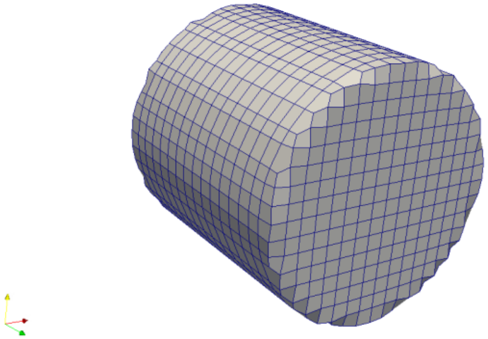
Refinement level

Distance from the surface
A large value covers the whole region

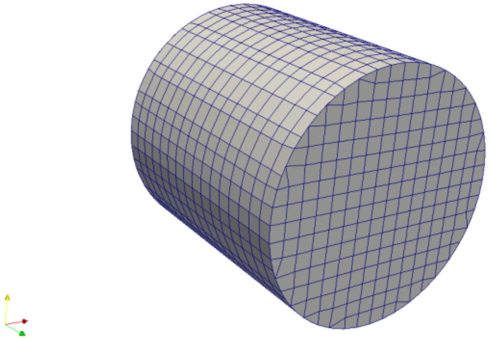
snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 110
Surface based refinement level (2 2)

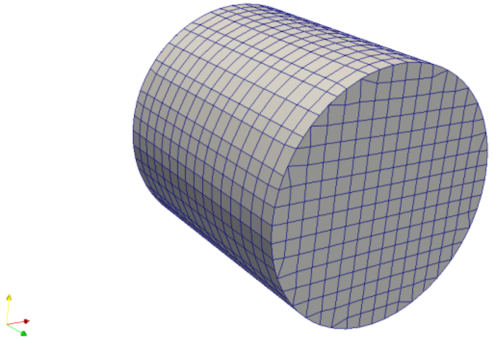


Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 2)

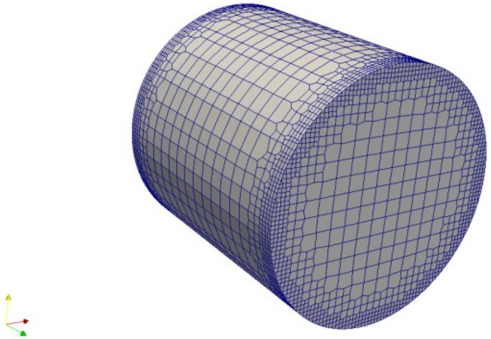
- To control edges capturing you can decrease the value of **resolveFeatureAngle**.
- Be careful, this parameter also controls curvature refinement, so if you choose a low value you also will be adding a lot of refinement on the surface.

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 2)



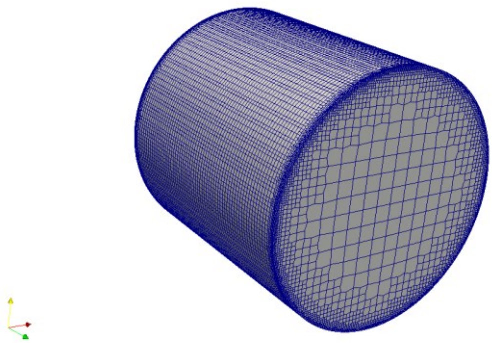
Explicit feature edge refinement level 4
resolveFeatureAngle 60
Surface based refinement level (2 2)

- To control edges refinement level, you can change the value of the explicit feature edge refinement level.

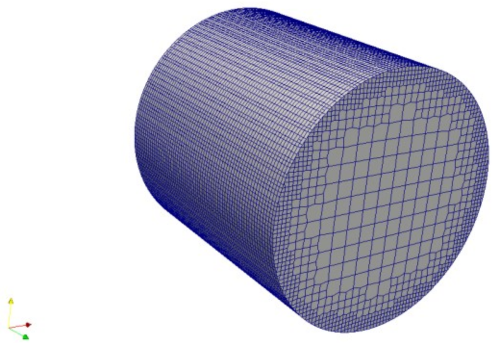
snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 6
resolveFeatureAngle 5
Surface based refinement level (2 4)

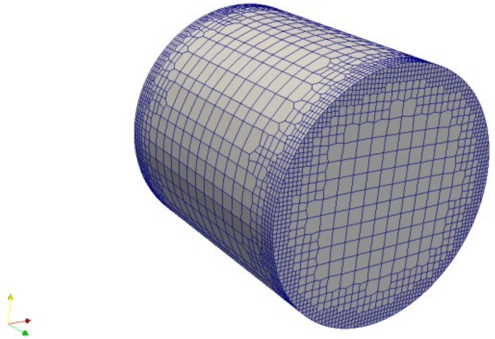


Explicit feature edge refinement level 0
resolveFeatureAngle 5
Surface based refinement level (2 4)

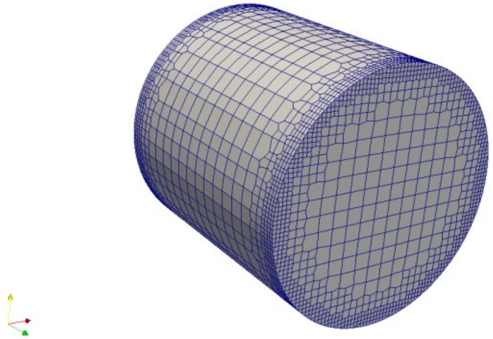
- To control edges refinement level, you can change the value of the explicit feature edge refinement level.

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 4)



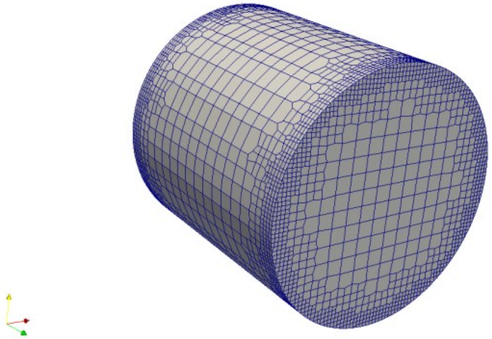
Explicit feature edge refinement level 4
resolveFeatureAngle 60
Surface based refinement level (2 2)

- To control surface refinement level, you can change the value of the surface based refinement level.
- The first digit controls the global surface refinement level and the second digit controls the curvature refinement level.

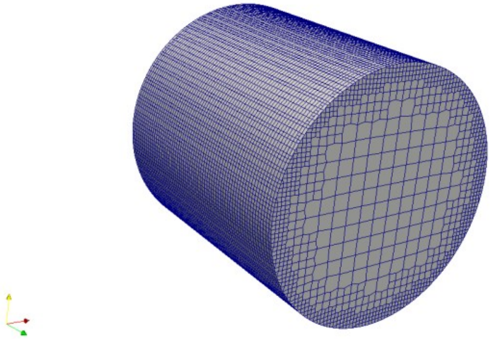
snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

Effect of various parameters on edge capturing and surface refinement



Explicit feature edge refinement level 0
resolveFeatureAngle 60
Surface based refinement level (2 4)



Explicit feature edge refinement level 0
resolveFeatureAngle 5
Surface based refinement level (2 4)

- To control surface refinement due to curvature together with control based surface refinement level, you can change the value of **resolveFeatureAngle**, and surface based refinement level

3D Cylinder with edge refinement.

- Let us explore the dictionary *surfaceFeaturesDict* used by the utility `surfaceFeatures`.
- This utility will extract surface features (sharp angles) according to an angle criterion (**includedAngle**).

```
surfaces ("surfacemesh.stl")

includedAngle          150;

subsetFeatures
{
        nonManifoldEdges     yes;

        openEdges            yes;
}

writeObj        yes;
```

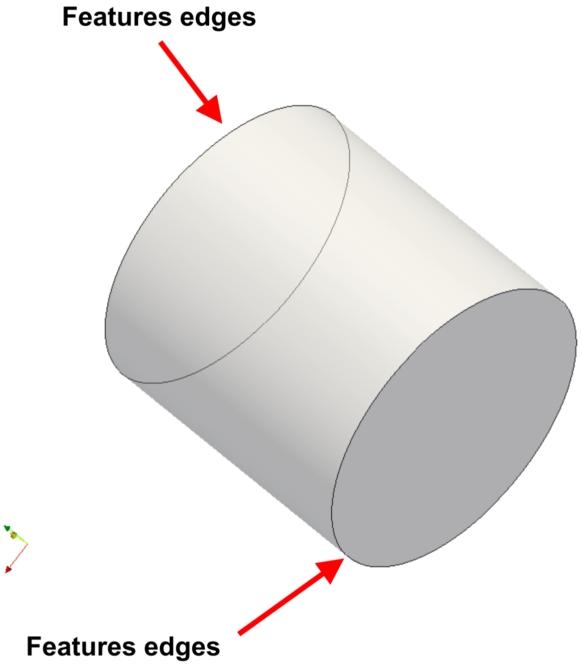
Name of the STL. The STL file is located in constant/triSurface

Angle criterion to extract features

Keep non-manifold edges (edges with more that 2 connected faces)

Keep open edges (edges with 1 connected face)

If you want to save the .obj files



snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

- Let us explore the dictionary *`surfaceFeaturesDict`* used by the utility `surfaceFeatures`.
- This utility will extract surface features (sharp angles) according to an angle criterion (**includedAngle**).

```
surfaces ("surfacemesh.stl")

includedAngle            150;

subsetFeatures
{
          nonManifoldEdges      yes;

          openEdges             yes;
}

writeObj         yes;
```

surfaces ("surfacemesh.stl"): Name of the STL. The STL file is located in constant/triSurface

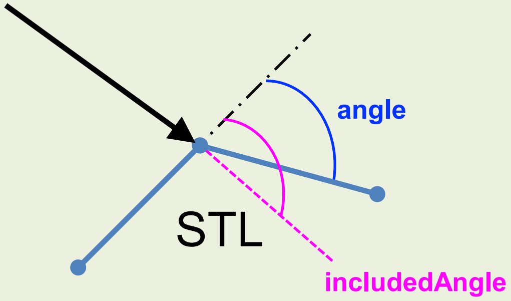
includedAngle 150;: Angle criterion to extract features

nonManifoldEdges yes;: Keep non-manifold edges (edges with more that 2 connected faces)

openEdges yes;: Keep open edges (edges with 1 connected face)

writeObj yes;: If you want to save the .obj files

If angle is less than includedAngle this feature will be marked

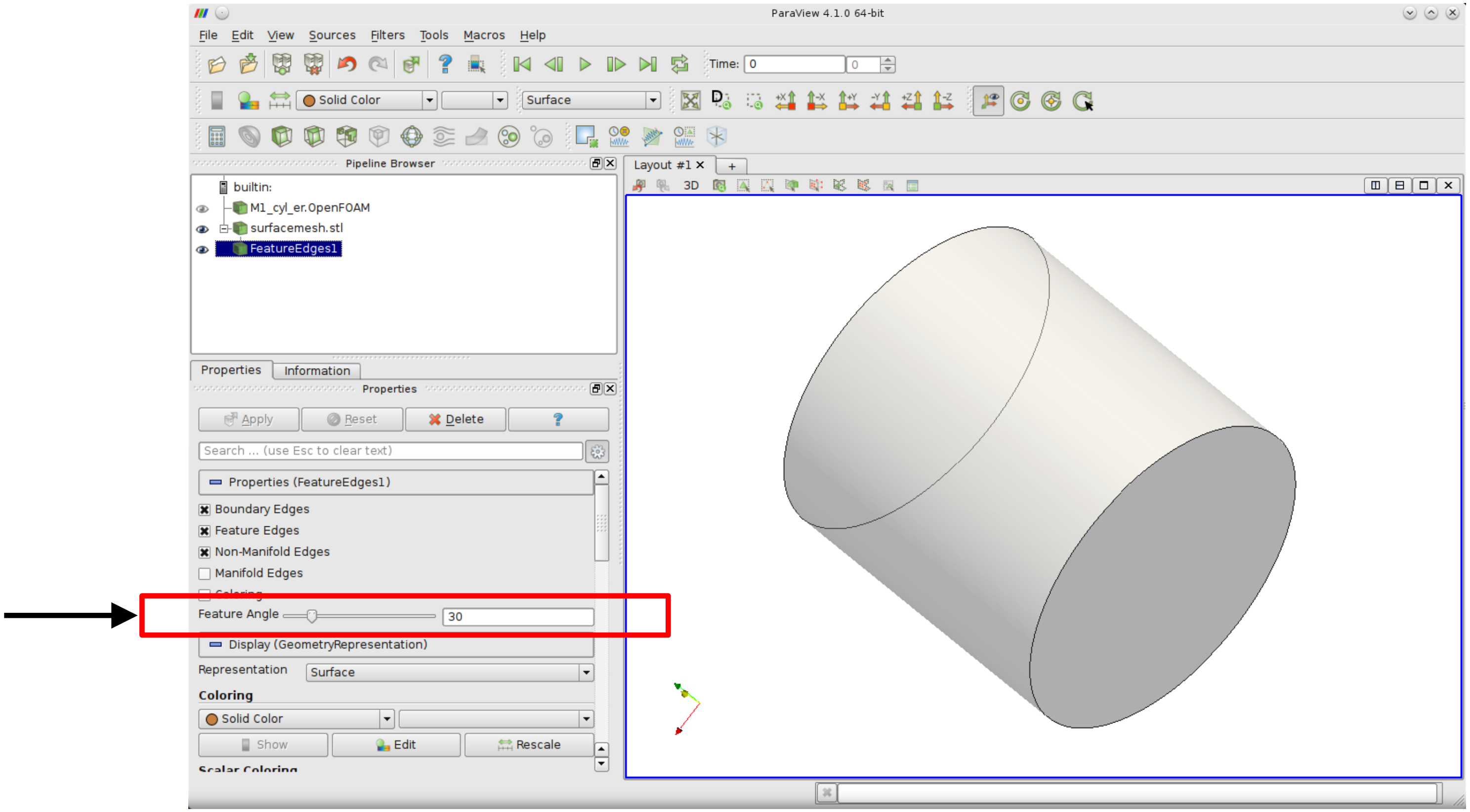


Mark edges whose adjacent surface normals are at an angle less than includedAngle

0: selects no edges
180: selects all edge

3D Cylinder with edge refinement.

- If you want to have a visual representation of the feature edges, you can use paraview/paraFoam.
- Just look for the filter `Feature Edges`.
- Have in mind that the angle you need to define in paraview/paraFoam is the complement of the angle you define in the dictionary *`surfaceFeaturesDict`*



builtin:
M1_cyl_er.OpenFOAM
surfacemesh.stl
FeatureEdges1

Properties (FeatureEdges1)

☒ Boundary Edges
☒ Feature Edges
☒ Non-Manifold Edges
□ Manifold Edges
□ Coloring

Feature Angle 30

Display (GeometryRepresentation)

Representation Surface

Coloring

Solid Color

3D Cylinder with edge refinement.

- In this case we are going to generate a body fitted mesh with edge refinement. This is an external mesh.
- These are the dictionaries and files that will be used.
  - *system/snappyHexMeshDict*
  - *system/surfaceFeaturesDict*
  - *system/meshQualityDict*
  - *system/blockMeshDict*
  - *constant/triSurface/surfacemesh.stl*
  - *constant/triSurface/surfacemesh.eMesh*

- The file *surfacemesh.eMesh* is generated after using the utility `surfaceFeatures`, which reads the dictionary *surfaceFeaturesDict*.
- The utility `surfaceFeatures`, will save a set of *.obj files with the captured edges. These files are located in the directory **`constant/extendedFeatureEdgeMesh`**. You can use paraview to visualize the *.obj files.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement.

- Let us generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> surfaceFeatures
3.  $> blockMesh
4.  $> snappyHexMesh -overwrite
5.  $> checkMesh -latestTime
6.  $> paraFoam
```

- In step 2 we extract the sharp angles from the geometry.
- In step 3 we generate the background mesh.
- In step 4 we generate the body fitted mesh. Have in mind that as we use the option `-overwrite`, we are not saving the intermediate steps.
- In step 5 we check the mesh quality.

snappyHexMesh guided tutorials

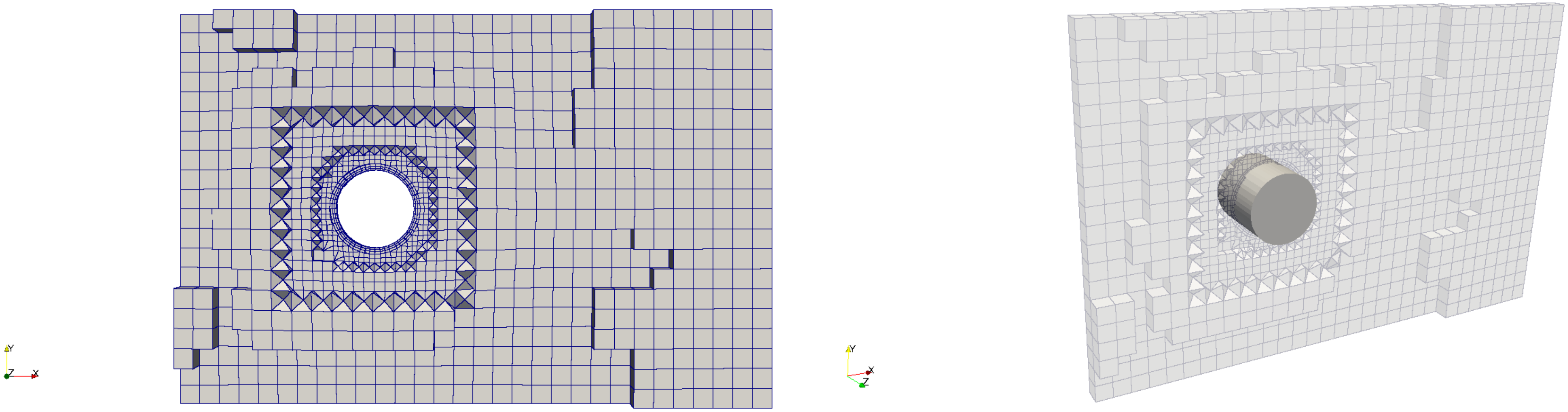
- Meshing with snappyHexMesh – Case 2.
- 3D cylinder with feature edge refinement and boundary layer (external mesh).
- You will find this case in the directory:

```
$PTOFC/101SHM/M1_cyl/C2
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.



Your final mesh should looks like this one

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- How do we enable boundary layer?
- In the file `snappyHexMeshDict`, look for the following entry:

```
castellatedMesh true;          //or false
snap true;                     //or false
addLayers true;                //or false

        ...
        ...
        ...
```

Set this parameter to true if you want to enable boundary layer meshing →

3D Cylinder with edge refinement and boundary layer.

- How do we enable boundary layer?
- In the file `snappyHexMeshDict`, look for the section **addLayersControls**:

```
addLayersControls
{

        //Global parameters
        relativeSizes true;
        expansionRatio 1.2;
        finalLayerThickness 0.5;
        minThickness 0.1;

        layers
        {
                banana_stlSurface
                {
                        nSurfaceLayers 3;
                }
        }

        // Advanced settings

        …
        …
        …

}
```

Name of the surface or user-defined patch where you want to add the boundary layer mesh.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

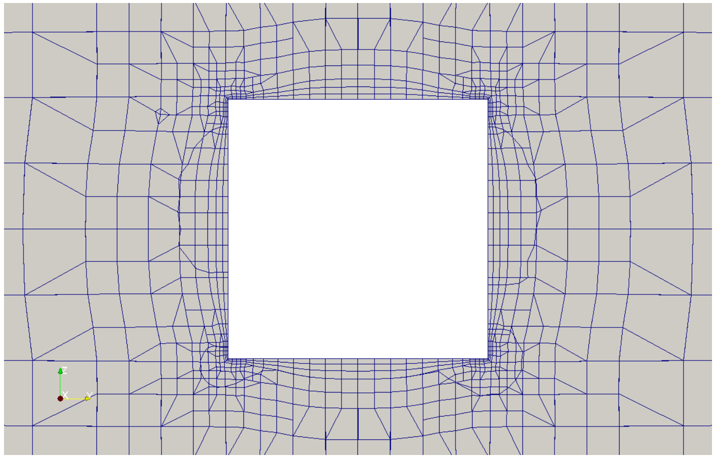
- How do we control boundary layer collapsing?
- In the file `snappyHexMeshDict`, look for the section **addLayersControls**:

```
addLayersControls
{

          ...
          ...
          ...

          // Advanced settings
          nGrow 0;
          featureAngle 130;

          ...
          ...
          ...

}
```

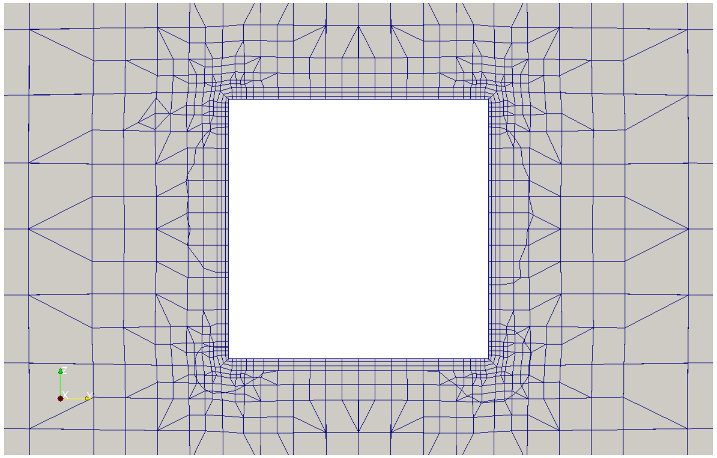
Increase this value to avoid BL collapsing

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



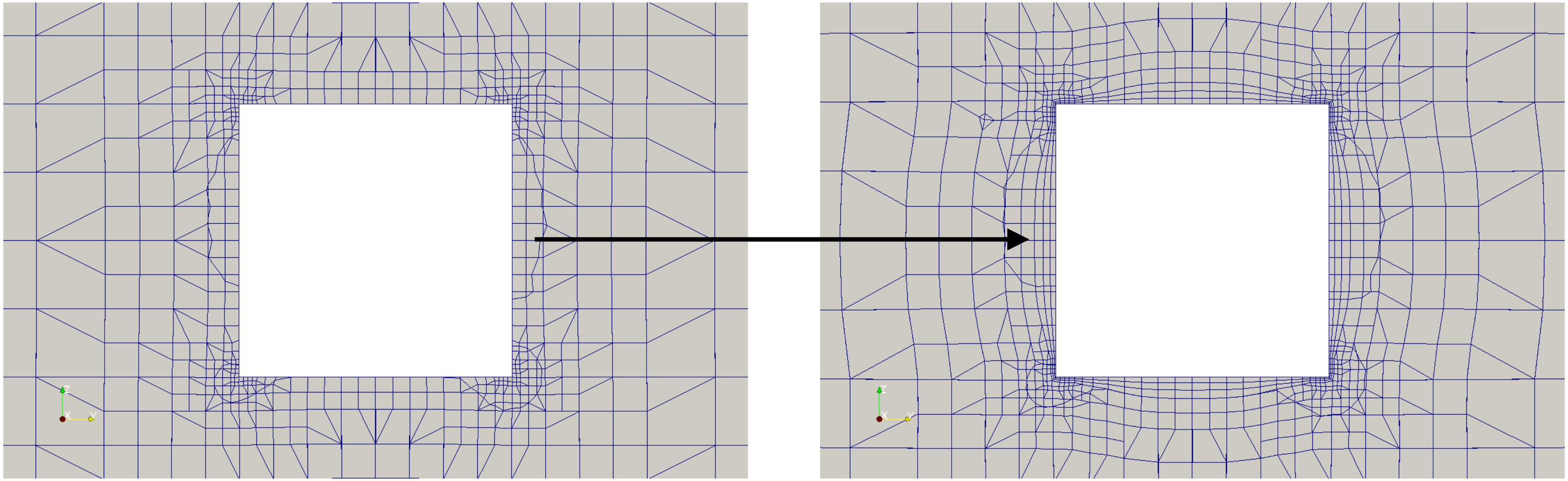
relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 4)



relativeSizes false
expansionRatio 1.2
firstLayerThickness 0.025
minThickness 0.01
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 4)

3D Cylinder with edge refinement and boundary layer.

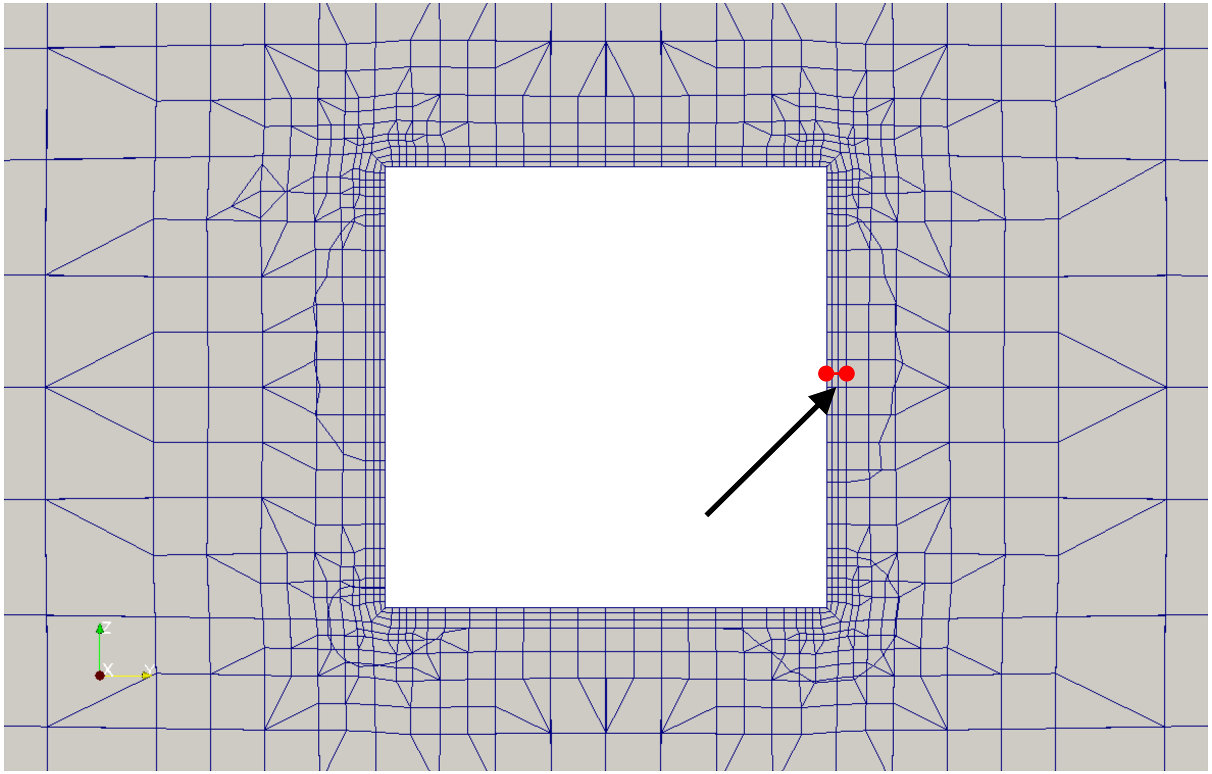
Effect of different parameters on the boundary layer meshing



- When the option **relativeSizes** is true, the boundary layer meshing is done relative to the size of the cells next to the surface.
- This option requires less user intervention but can not guarantee a uniform boundary layer.
- Also, it is quite difficult to set a desired thickness of the first layer.

3D Cylinder with edge refinement and boundary layer.

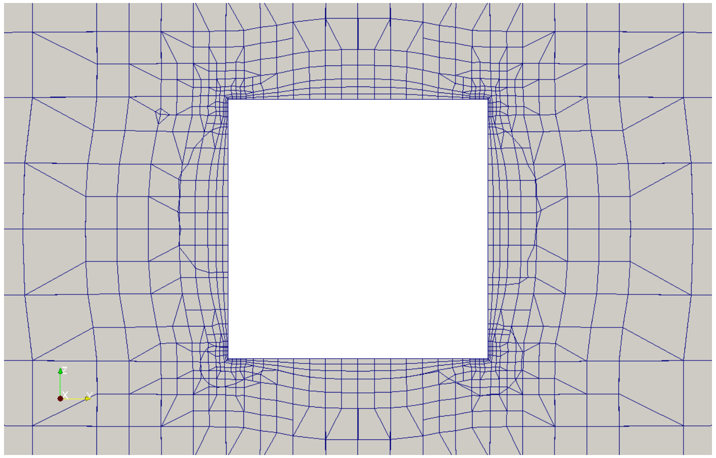
Effect of different parameters on the boundary layer meshing



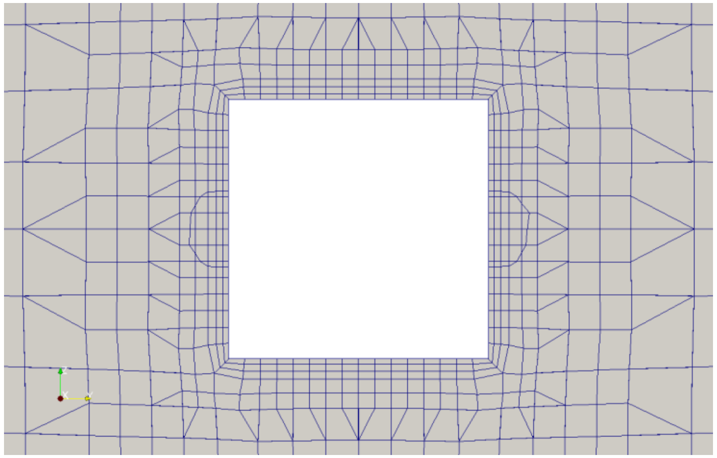
- When the option **relativeSizes** is false, we give the actual thickness of the layers.
- This option requires a lot user intervention but it guarantees a uniform boundary layer and the desired layer thickness.

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 4)

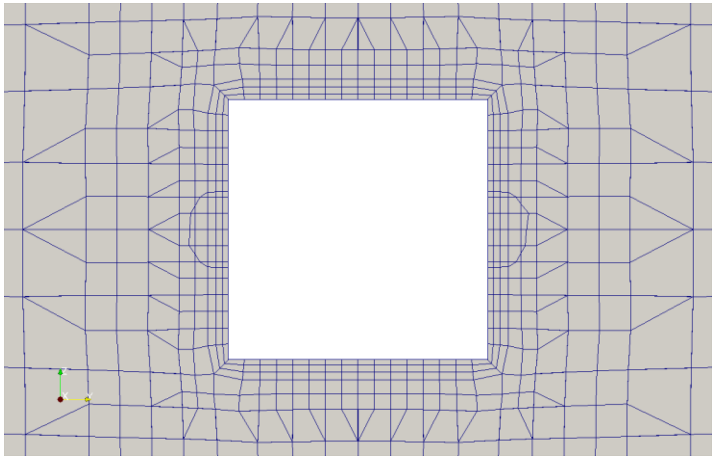


relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 2)

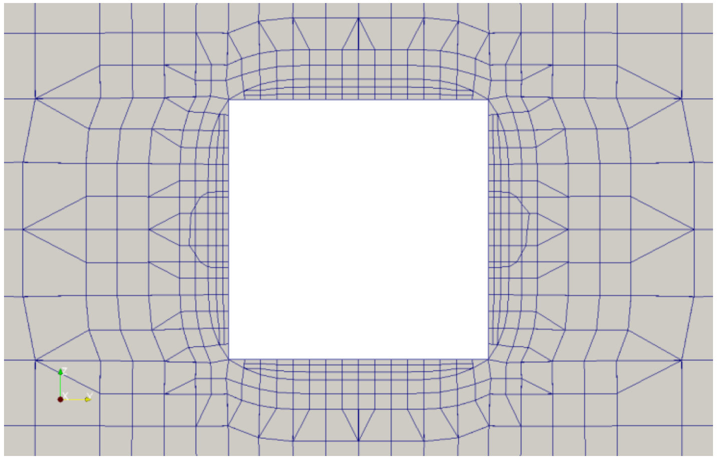
- When the option **relativeSizes** is true and in order to have a uniform boundary layer, we need to have a uniform surface refinement.
- Nevertheless, we still do not have control on the desired thickness of the first layer.

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 130
nSurfaceLayers 3
Surface based refinement level (2 2)



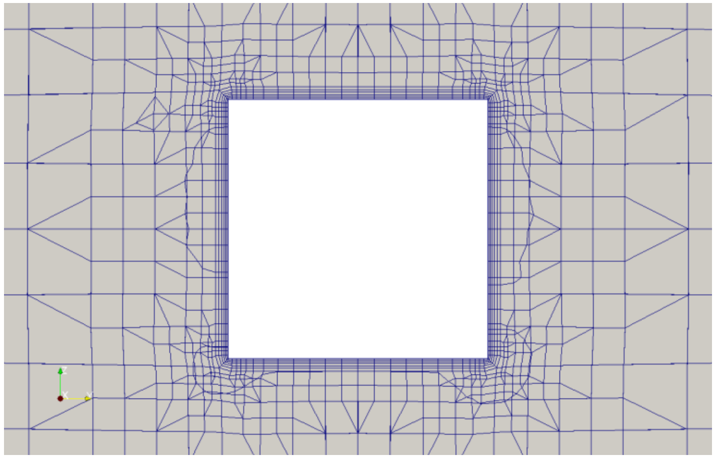
relativeSizes true
expansionRatio 1.2
finalLayerThickness 0.5
minThickness 0.1
featureAngle 30
nSurfaceLayers 3
Surface based refinement level (2 2)

- To avoid boundary layer collapsing close to the corners, we can increase the value of the boundary layer parameter **featureAngle**.

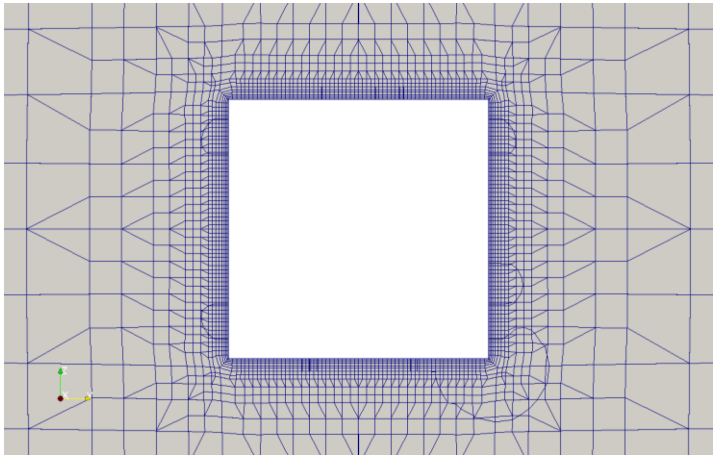
snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

Effect of different parameters on the boundary layer meshing



relativeSizes false
nSurfaceLayers 6



relativeSizes false
nSurfaceLayers 6
Refinement region at the stl surface:
mode distance;
levels ((0.05 4))

- The disadvantage of setting **relativeSizes** to false, is that it is difficult to control the expansion ratio from the boundary layer meshing to the far mesh.
- To control this transition, we can add a refinement region at the surface with distance mode.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- To generate the mesh, in the terminal window type:

1. `$> foamCleanTutorials`
2. `$> surfaceFeatures`
3. `$> blockMesh`
4. `$> snappyHexMesh -overwrite`
5. `$> checkMesh –latestTime`
6. `$> paraFoam`

3D Cylinder with edge refinement and boundary layer.

- At the end of the meshing process you will get the following information regarding the boundary layer meshing:

```
patch                  faces      layers      overall     thickness
                                               [m]         [%]
-----                  -----      ------      ---         ---
banana_stlSurface      4696       3           0.0569      95.9

Layer mesh : cells:48577     faces:157942     points:61552
```

- This is a general summary of the boundary layer meshing.
- Pay particular attention to the overall and thickness information.
- Overall is roughly speaking the thickness of the whole boundary layer.
- Thickness is the percentage of the patch that has been covered with the boundary layer mesh. A thickness of 100% means that the whole patch has been covered (a perfect BL mesh).

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

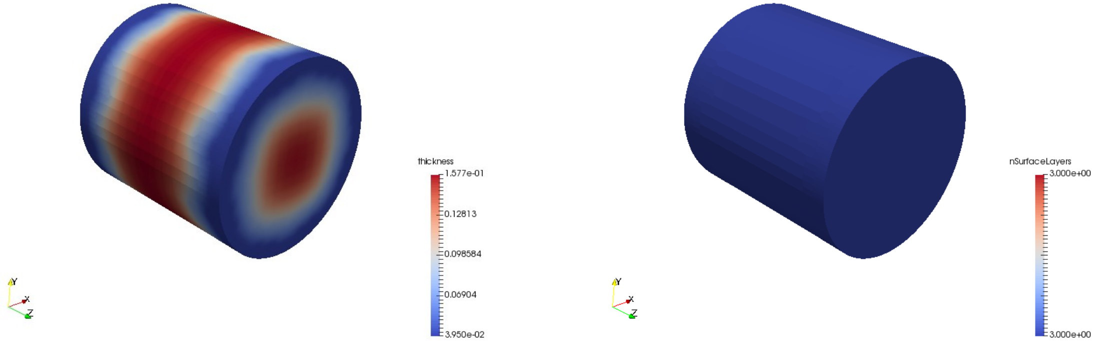
- If you want to visualize the boundary layer thickness, you can enable **writeFlags** in the *snappyhexMeshDict* dictionary,

```
...
...
...

writeFlags
(
        scalarLevels;     // write volScalarField with cellLevel for postprocessing
        layerSets;        // write cellSets, faceSets of faces in layer
        layerFields;      // write volScalarField for layer coverage
);

...
...
...
```

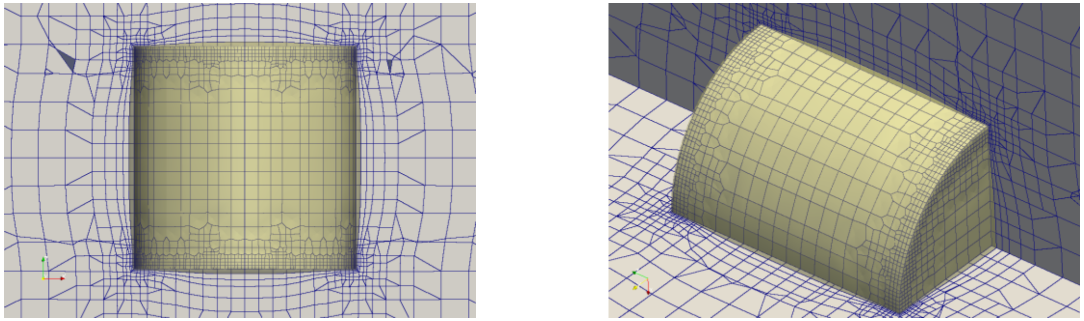
snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- Then you can use paraview/paraFoam to visualize the boundary layer coverage.



Boundary layer thickness and number of layers



The yellow surface represent the BL coverage

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- After creating the mesh and if you do not like the inflation layer or you want to try different layer parameters, you do not need to start the meshing process from scratch.
- To restart the meshing process from a saved state you need to save the intermediate steps (castellation and snapping), and then create the inflation layers starting from the snapped mesh.
- That is, do not use the option `snappyHexMesh -overwrite`.
- Also, in the dictionary *`controlDict`* remember to set the entry `startFrom` to `latestTime` or the time directory where the snapped mesh is saved (in this case 2).
- Before restarting the meshing, you will need to turn off the castellation and snapping options and turn on the boundary layer options in the *`snappyHexMeshDict`* dictionary.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer.

- Remember, before restarting the meshing you will need to modify the *snappyHexMeshDict* dictionary as follows:

```
castellatedMesh        false;
snap                   false;
addLayers              true;
```

- At this point, you can restart the meshing process by typing in the terminal,
  - `$> snappyHexMesh`

- By the way, you can restart the boundary layer mesh from a previous mesh with a boundary layer.
- So in theory, you an add one layer at a time, this will give you more control but it will require more manual work and some scripting.

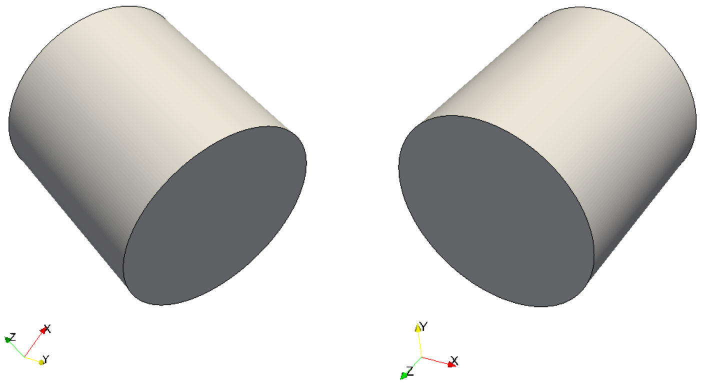
snappyHexMesh guided tutorials

- Meshing with snappyHexMesh – Case 3.
- 3D cylinder with feature edge refinement and boundary layer using a STL with multiple surfaces (external mesh).
- You will find this case in the directory:

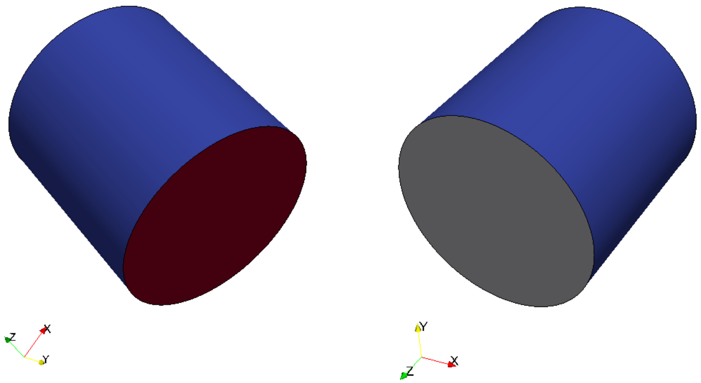
```
$PTOFC/101SHM/M1_cyl/C3
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.



STL visualization with a single surface using paraview (the single surface in represented with a single color)

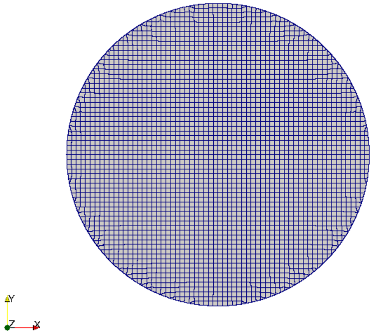


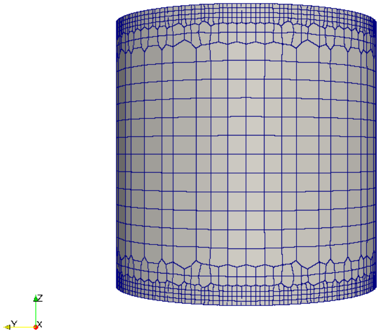
STL visualization with multiple surfaces using paraview (each color corresponds to a different surface)

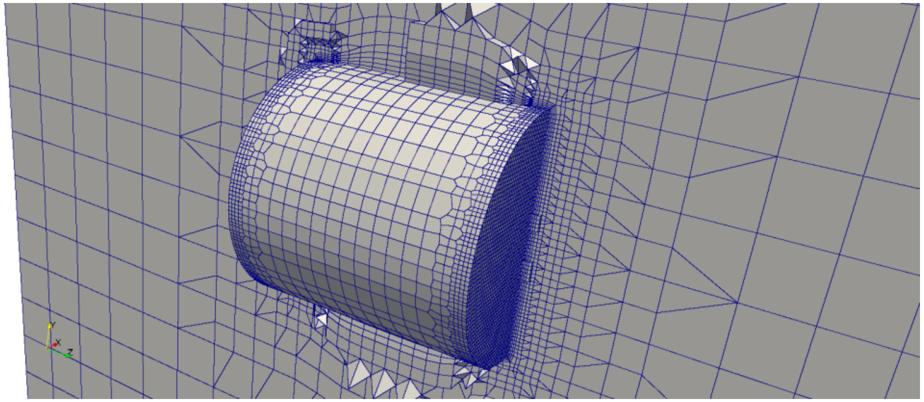
- When you use a STL with multiple surfaces, you have more control over the meshing process.
- By default, STL files are made up of one single surface.
- If you want to create the multiple surfaces you will need to do it in the solid modeler.
- Alternatively, you can split the STL manually or using the utility `surfaceAutoPatch`.
- Loading multiple STLs is equivalent to using a STL with multiple surfaces.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.







- When you use a STL with multiple surfaces, you have more control over the meshing process.
- In this case, we were able to use different refinement parameters in the lateral and central surface patches of the cylinder.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we assign different names to different surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:

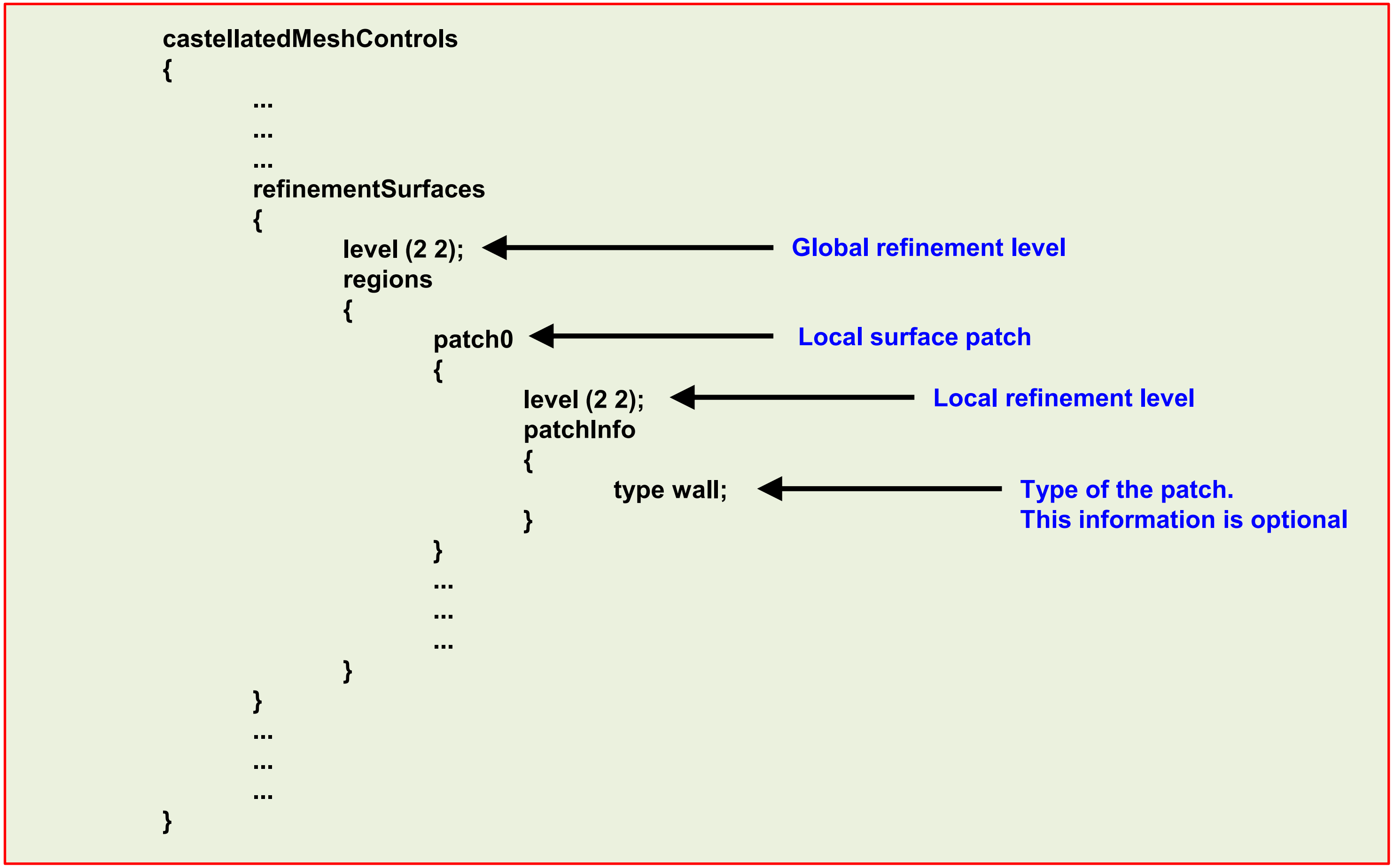
```
geometry
{
        surfacemesh.stl
        {
                type triSurfaceMesh;
                name stlSurface;

                regions
                {
                        patch0
                        {
                                name surface0;
                        }
                        patch1
                        {
                                name surface1;
                        }
                        patch2
                        {
                                name surface2;
                        }
                }
        }
        ...
        ...
        ...
}
```

Named region in the STL file

User-defined patch name
This is the name you need to use when setting the boundary layer meshing

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we refine user defined surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:



```
castellatedMeshControls
{
          ...
          ...
          ...
          refinementSurfaces
          {
                    level (2 2);
                    regions
                    {
                              patch0
                              {
                                        level (2 2);
                                        patchInfo
                                        {
                                                  type wall;
                                        }
                              }
                              ...
                              ...
                              ...
                    }
          }
          ...
          ...
          ...
}
```

Global refinement level

Local surface patch

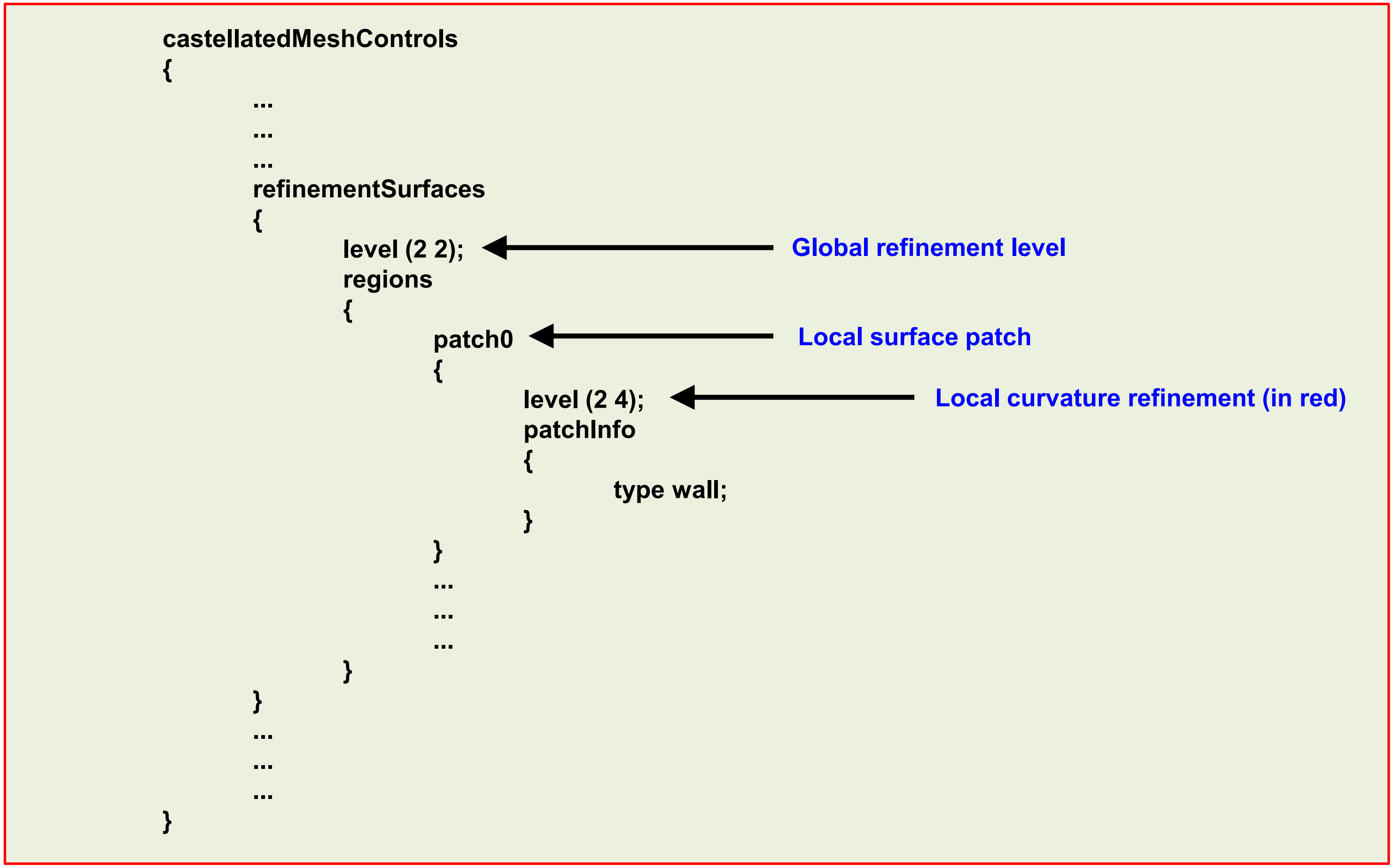
Local refinement level

Type of the patch.
This information is optional

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we control curvature refinement on surface patches?
- In the file *snappyHexMeshDict*, look for the following entry:



```
castellatedMeshControls
{
          ...
          ...
          ...
          refinementSurfaces
          {
                    level (2 2);
                    regions
                    {
                              patch0
                              {
                                        level (2 4);
                                        patchInfo
                                        {
                                                  type wall;
                                        }
                              }
                              ...
                              ...
                              ...
                    }
          }
          ...
          ...
          ...
}
```

Global refinement level

Local surface patch

Local curvature refinement (in red)

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we control curvature refinement on surface patches?
- In the file `snappyHexMeshDict`, look for the following entry:

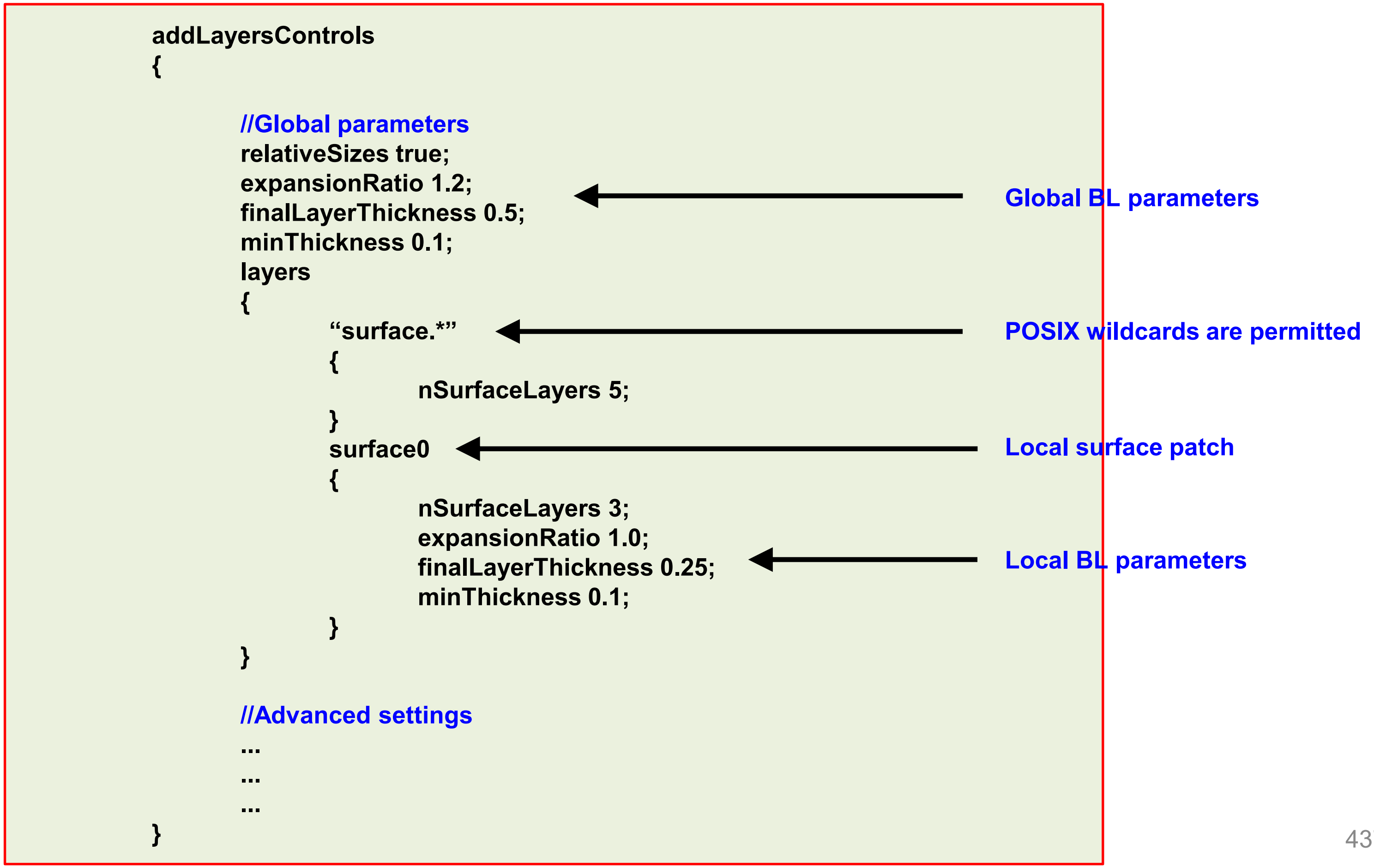
```
castellatedMeshControls
{

          ...
          ...
          ...

          //Local curvature and
          //feature angle refinement
          resolveFeatureAngle 60;

          ...
          ...
          ...

}
```

The default value is 30.
Using a higher value will capture less features.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- How do we control boundary layer meshing on the surface patches?
- In the file `snappyHexMeshDict`, look for the following entry:



```
addLayersControls
{

        //Global parameters
        relativeSizes true;
        expansionRatio 1.2;
        finalLayerThickness 0.5;
        minThickness 0.1;
        layers
        {
                “surface.*”
                {
                        nSurfaceLayers 5;
                }
                surface0
                {
                        nSurfaceLayers 3;
                        expansionRatio 1.0;
                        finalLayerThickness 0.25;
                        minThickness 0.1;
                }
        }

        //Advanced settings
        …
        …
        …
}
```

Global BL parameters

POSIX wildcards are permitted

Local surface patch

Local BL parameters

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- Let us first create the STL file with multiple surfaces.
- In the directory **geo**, you will find the original STL file.
- In the terminal type:

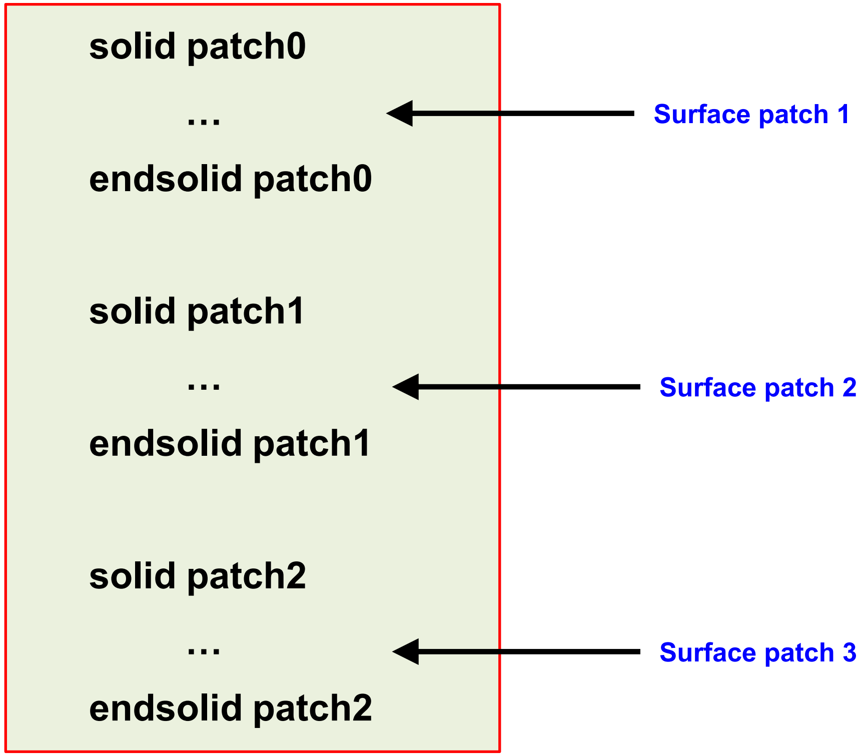
```
1.  $> cd geo
2.  $> surfaceAutoPatch geo.stl output.stl 130
3.  $> cp output.stl ../constant/triSurface/surfacemesh.stl
4.  $> cd ..
5.  $> paraview
```

- The utility `surfaceAutoPatch` will read the original STL file (*`geo.stl`*), and it will find the patches using an angle criterion of 130 (similar to the angle criterion used with the utility `surfaceFeatures`). It writes the new STL geometry in the file *`output.stl`*.
- By the way, it is better to create the STL file with multiple surfaces directly in the solid modeler.
- FYI, there is an equivalent utility for meshes, `autoPatch`. So if you forgot to define the patches, this utility will automatically find the patches according to an angle criterion.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- If you open the file `output.stl`, you will notice that there are three surfaces defined in the STL file. The different surfaces are defined in by the following sections:



```
solid patch0
    …
endsolid patch0

solid patch1
    …
endsolid patch1

solid patch2
    …
endsolid patch2
```

- The name of the solid sections are automatically given by the utility `surfaceAutoPatch`.

- The convention is as follows: patch0, patch1, patch2, … patchN.

- If you do not like the names, you can change them directly in the STL file.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- The new STL file is already in the **`constant/triSurface`** directory.
- To generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> surfaceFeatures
3.  $> blockMesh
4.  $> snappyHexMesh -overwrite
5.  $> checkMesh -latestTime
```

- To visualize the mesh, in the terminal window type:

```
6.  $> paraFoam
```

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- This case is ready to run using the solver `simpleFoam`. But before running, you will need to set the boundary and initial conditions.
- You will need to manually modify the file *`constant/polyMesh/boundary`*
- Remember:
  - **Base type** boundary conditions are defined in the file *`boundary`* located in the directory **`constant/polyMesh`**.
  - **Numerical type** boundary conditions are defined in the field variables files located in the directory **`0`** or the time directory from which you want to start the simulation (e.g. *`U`*, *`p`*).
  - The name of the base type boundary conditions and numerical type boundary conditions needs to be the same.
  - Also, the base type boundary condition needs to be compatible with the numerical type boundary condition.

snappyHexMesh guided tutorials

3D Cylinder with edge refinement and boundary layer, using a STL file with multiple surfaces.

- This case is ready to run with `simpleFoam`.
- To run the case (mesh and simulation), type in the terminal,

1.
```
$> sh run_all.sh
```

- Feel free to open the files `run_mesh.sh` (meshing steps) and `run_solver.sh` (simulation steps) to get an idea of all steps used.
- The most critical step is to give the right name and type to the boundary patches, this is done in the file `boundary` and the input files located in the directory **0** (boundary conditions and initial conditions).

snappyHexMesh guided tutorials

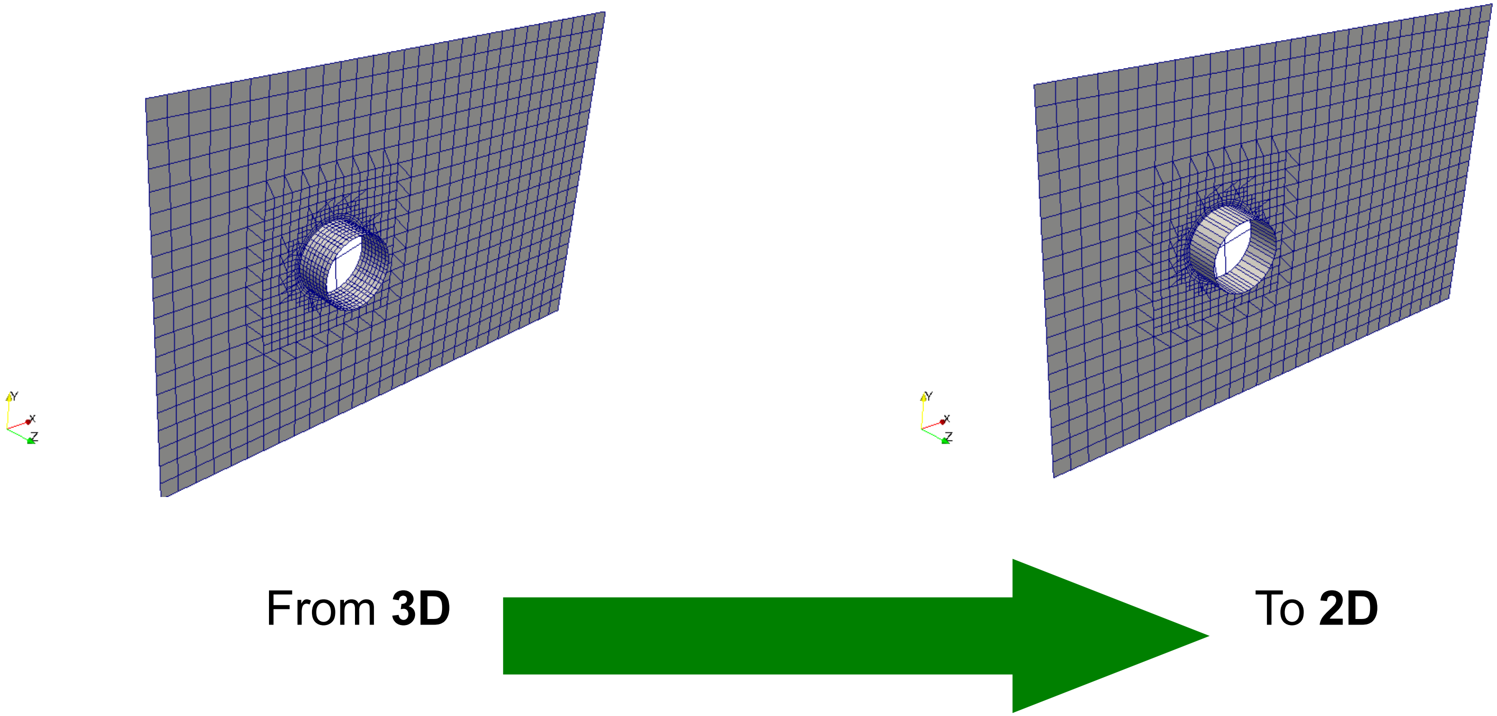
- Meshing with snappyHexMesh – Case 4.
- 2D cylinder (external mesh)
- You will find this case in the directory:

```
$PTOFC/101SHM/M1_cyl/C4
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

snappyHexMesh guided tutorials

2D Cylinder

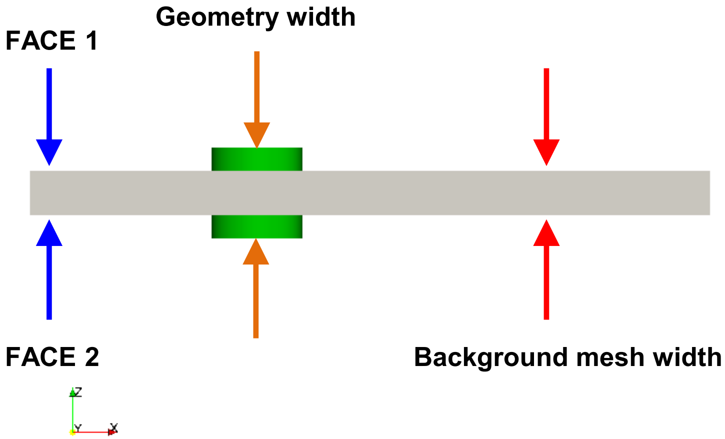


From **3D** → To **2D**

- To generate a 2D mesh using `snappyHexMesh`, we need to start from a 3D. After all, `snappyHexMesh` is a 3D mesher.
- To generate a 2D mesh (and after generating the 3D mesh), we use the utility `extrudeMesh`.
- The utility `extrudeMesh` works by projecting a face into a mirror face. Therefore, the faces need to parallel.

2D Cylinder

The utility `extrudeMesh` **works by projecting FACE 1 into FACE 2. Therefore, the faces need to be parallel.**



- At most, the input geometry and the background mesh need to have the same width.
- If the input geometry is larger than the background mesh, it will be automatically cut by the faces of the background mesh.
- In this case, the input geometry will be cut by the two lateral patches of the background mesh.
- If you want to take advantage of symmetry in 3D, you can cut the geometry in half using one of the faces of the background mesh.
- When dealing with 2D
- Extracting the features edges is optional for the 2D geometry extremes, but it is recommended if there are internal edges that you want to resolve.

snappyHexMesh guided tutorials

2D Cylinder

- How do we create the 2D mesh?
- After generating the 3D mesh, we use the utility `extrudeMesh`.
- This utility reads the *`extrudeMeshDict`*,

```
constructFrom patch;

sourceCase “.”
sourcePatches (minZ);

exposedPatchName maxZ;

extrudeModel linearNormal

nLayers 1;

linearNormalCoeffs
{
        thickness 1;
}

mergeFaces false;
```

sourcePatches (minZ); ← Name of source patch

exposedPatchName maxZ; ← Name of the mirror patch

nLayers 1; ← Number of layers to use in the linear extrusion. As this is a 2D case we must use 1 layer

thickness 1; ← Thickness of the extrusion. It is highly recommended to use a value of 1

snappyHexMesh guided tutorials

2D Cylinder

- To generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> snappyHexMesh -overwrite
4.  $> extrudeMesh
5.  $> checkMesh -latestTime
6.  $> paraFoam
```

- Remember, the utility `extrudeMesh` (step 4) reads the dictionary *`extrudeMeshDict`*, which is located in the directory **`system`**.
- Also remember to set the empty patches in the dictionary *`boundary`* and in the boundary conditions.

Exercises

- To get a feeling of the surface refinement, try to change the value of the surface refinement in the dictionary *snappyHexMeshDict*.
- In the dictionary *snappyHexMeshDict*, change the value of **nCellsBetweenLevels** and **resolveFeatureAngle**. What difference do you see in the output?
- Use paraview to get a visual representation of the feature angles.
- In the dictionary *snappyHexMeshDict*, try to add curvature based refinement.
- In the dictionary *snappyHexMeshDict*, in the section **addLayersControls** change the value of **featureAngle**. Use a value of 60 and 160 and compare the boundary layer meshing.
- To control the boundary layer collapsing, try to use a uniform surface refinement. For this you have two options, set surface level refinement to a uniform value or adding distance region refinement at the wall.
- To control the boundary layer collapsing, try to use absolute sizes when creating the boundary layer mesh.
- To get a feeling of region refinement, try to change the value of the local refinement in the dictionary *snappyHexMeshDict*. What difference do you see in the output?
- Try to use local inflation layers in the regions defined.
- In the dictionary *extrudeMeshDict*, change the value of **nLayers** and **thickness**.
- In the dictionary *extrudeMeshDict*, try to change the **extrudeModel**.

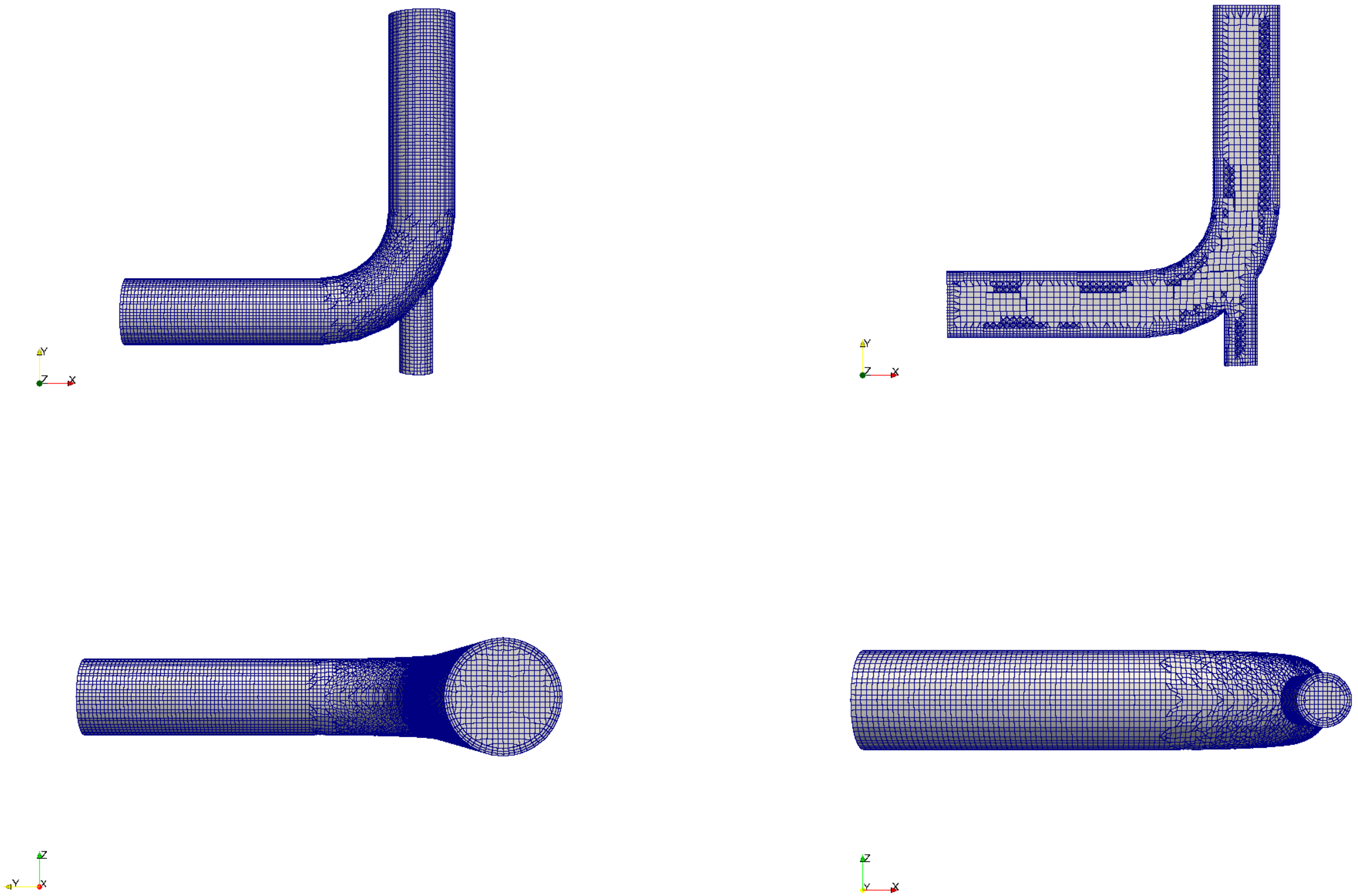
snappyHexMesh guided tutorials

- Meshing with snappyHexMesh – Case 5.
- Mixing elbow (internal mesh)
- You will find this case in the directory:

```
$PTOFC/101SHM/M2_mixing_elbow
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

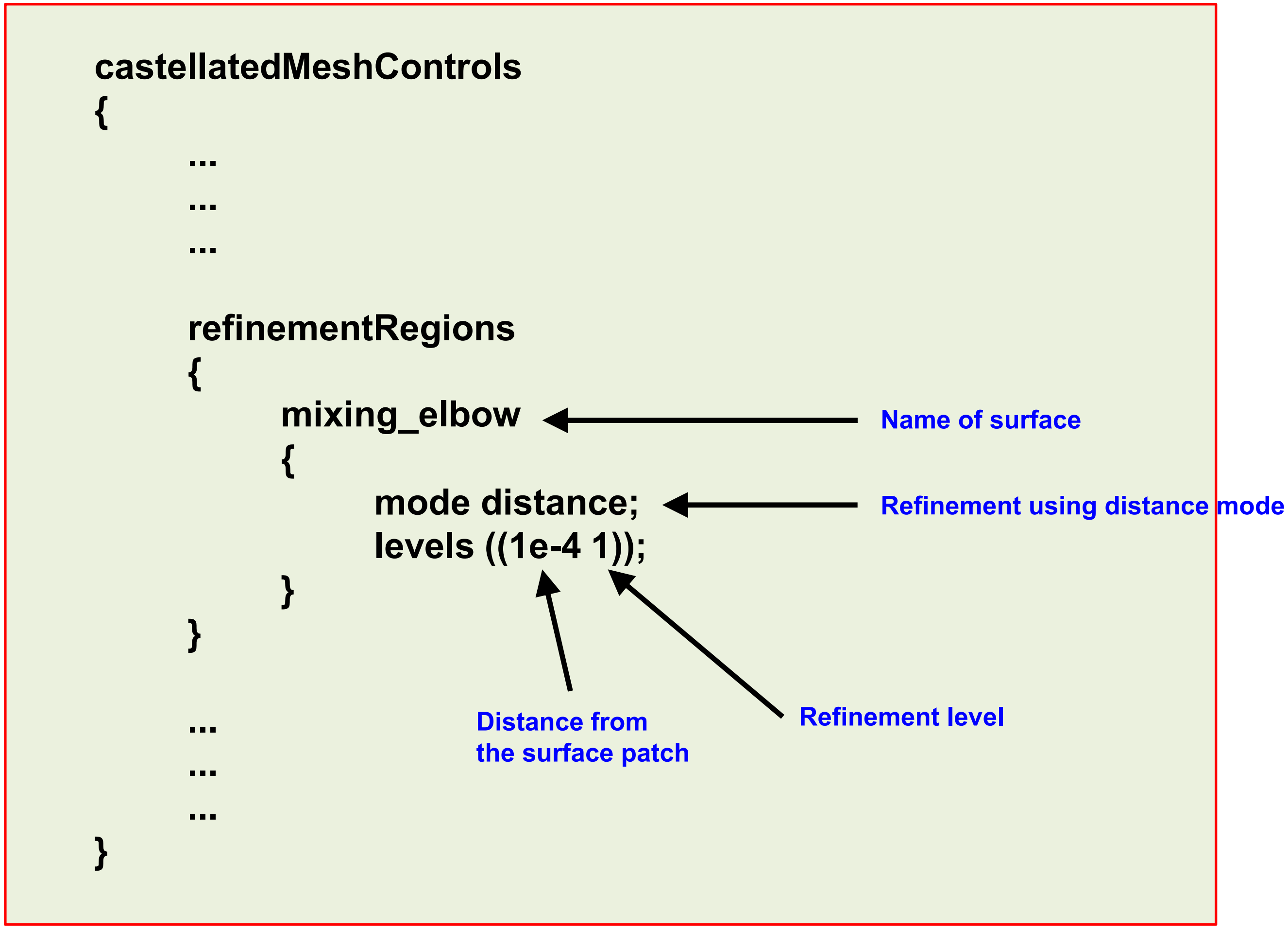
Mixing elbow.



Your final mesh should looks like this one

Mixing elbow.

- How do we control surface refinement using region refinement?
- In the file *snappyHexMeshDict*, look for the following entry:



```
castellatedMeshControls
{
        ...
        ...
        ...

        refinementRegions
        {
                mixing_elbow
                {
                        mode distance;
                        levels ((1e-4 1));
                }
        }

        ...
        ...
        ...
}
```

Name of surface

Refinement using distance mode

Distance from the surface patch

Refinement level

snappyHexMesh guided tutorials

Mixing elbow.

- In this case we are going to generate a body fitted mesh with edge refinement and boundary layer meshing.
- This is an internal mesh.
- These are the dictionaries and files that will be used.
  - *`system/snappyHexMeshDict`*
  - *`system/surfaceFeaturesDict`*
  - *`system/meshQualityDict`*
  - *`system/blockMeshDict`*
  - *`constant/triSurface/surfacemesh_multi.stl`*
  - *`constant/triSurface/surfacemesh_multi.eMesh`*

- The file *`surfacemesh_multi.eMesh`* is generated after using the utility `surfaceFeatures`, which reads the dictionary *`surfaceFeaturesDict`*.

snappyHexMesh guided tutorials

Mixing elbow.

- At this point, we are going to work in parallel (but you can work in serial as well).
- To generate the mesh, in the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> surfaceFeatures
3.  $> blockMesh
4.  $> decomposePar
5.  $> mpirun -np 4 snappyHexMesh –parallel –overwrite
6.  $> mpirun -np 4 checkMesh –parallel –latestTime
7.  $> reconstructParMesh -constant
8.  $> paraFoam
```

Mixing elbow.

- So what did we do?
  - Step 4: we distribute the mesh among the processors we want to use.
  - Step 5 and 6: we run in parallel.
  - Step 7: we put back together the decomposed mesh.
  - Step 8: we visualize the reconstructed mesh.

- Notice that the utility `blockMesh` does not run in parallel.
- Remember to set the keyword **numberOfSubdomains** in the dictionary *decomposeParDict* equal to the number of processors you want to use.
- In this case, as we are using 4 processors with mpirun, **numberOfSubdomains** needs to be equal to 4.
- To run the simulation and after reconstructing the mesh, you will need to transfer the boundary and initial conditions information to the decomposed mesh,
  - `$> decomposePar –fields`
- Or you can force to decompose everything as follows,
  - `$> decomposePar –force`

snappyHexMesh guided tutorials

Mixing elbow.

- After running `checkMesh`, you will get the following information regarding the patch names:

```
                 Patch    Faces   Points                  Surface topology
  mixing_elbow_inlet1     1264     1297  ok (non-closed singly connected)
                  pipe    38884    41118  ok (non-closed singly connected)
  mixing_elbow_inlet2      314      337  ok (non-closed singly connected)
   mixing_elbow_outlet     1264     1297  ok (non-closed singly connected)
```

- Sometimes you can get empty patches.

```
                 Patch    Faces   Points                  Surface topology
                  minX        0        0                        ok (empty)
                  maxX        0        0                        ok (empty)
                  minY        0        0                        ok (empty)
                  maxY        0        0                        ok (empty)
                  minZ        0        0                        ok (empty)
                  maxZ        0        0                        ok (empty)
  mixing_elbow_inlet1     1264     1297  ok (non-closed singly connected)
                  pipe    38884    41118  ok (non-closed singly connected)
  mixing_elbow_inlet2      314      337  ok (non-closed singly connected)
   mixing_elbow_outlet     1264     1297  ok (non-closed singly connected)
```

snappyHexMesh guided tutorials

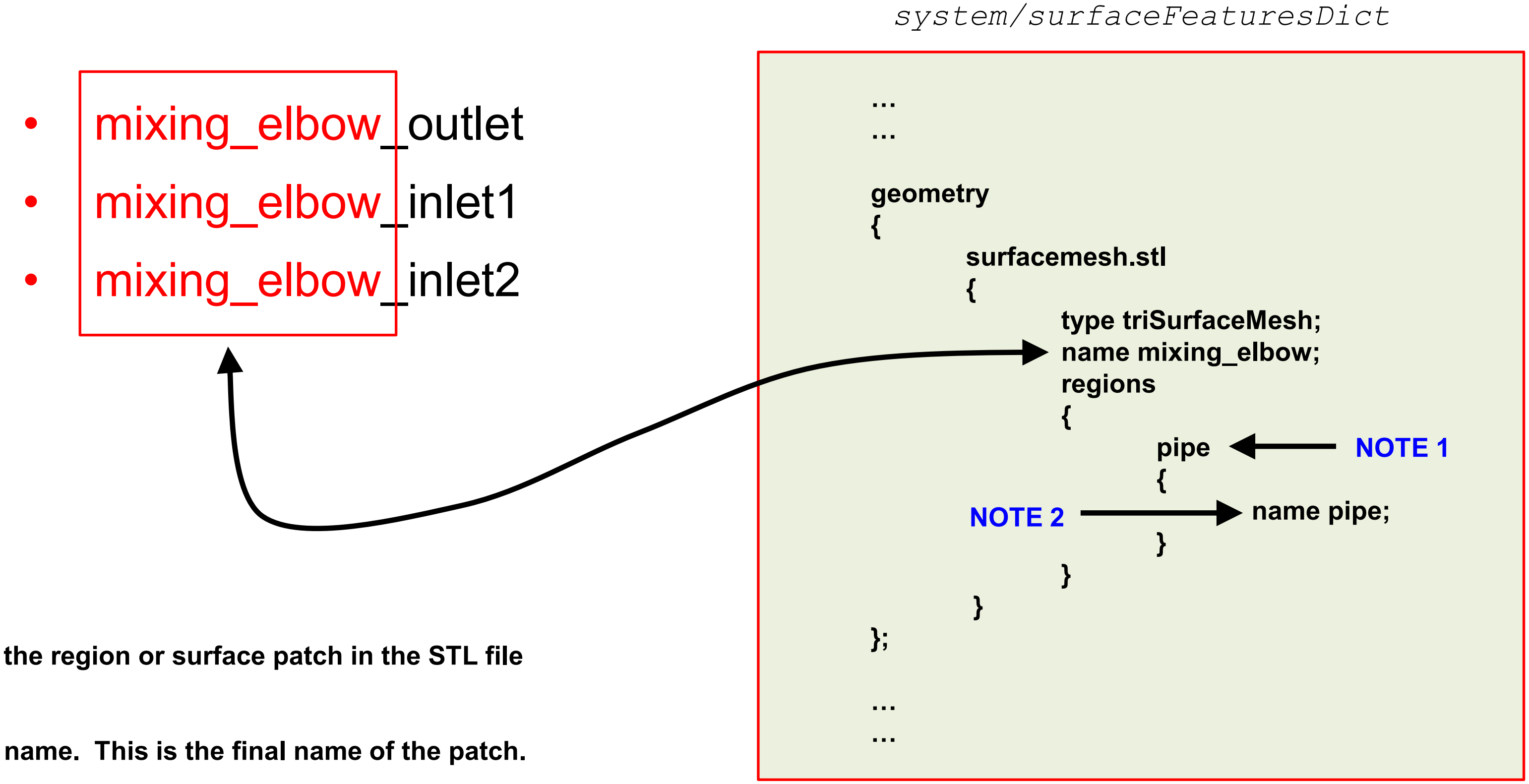
Mixing elbow.

- Empty patches are no problem, they remain from the background mesh.
- To erase the empty patches, you can do it manually (you will need to modify the file *boundary*), or you can use the utility `createPatch` as follows (the utility runs in parallel):
    - `$> createPatch -overwrite`
- The surface patch **pipe** was created in the geometry section of the dictionary *snappyHexMeshDict*.
- The patches **mixing_elbow_outlet**, **mixing_elbow_inlet1** and **mixing_elbow_inlet2** were created automatically by `snappyHexMesh`.
- You have the choice of giving the names of the patches yourself or letting `snappyHexMesh` assign the names automatically.
- Remember, when creating the boundary layer mesh, these are the names you need to use to assign the layers.

snappyHexMesh guided tutorials

Mixing elbow.

- The mesh used in the previous case was a STL with multiple surfaces.
- In you do not create the regions in the geometry section of the dictionary *snappyHexMeshDict*, `snappyHexMesh` will automatically assign the names of the surface patches as follows:



- mixing_elbow_outlet
- mixing_elbow_inlet1
- mixing_elbow_inlet2

system/surfaceFeaturesDict

```
...
...

geometry
{
          surfacemesh.stl
          {
                    type triSurfaceMesh;
                    name mixing_elbow;
                    regions
                    {
                              pipe
                              {
                                        name pipe;
                              }
                    }
          }
};

...
...
```

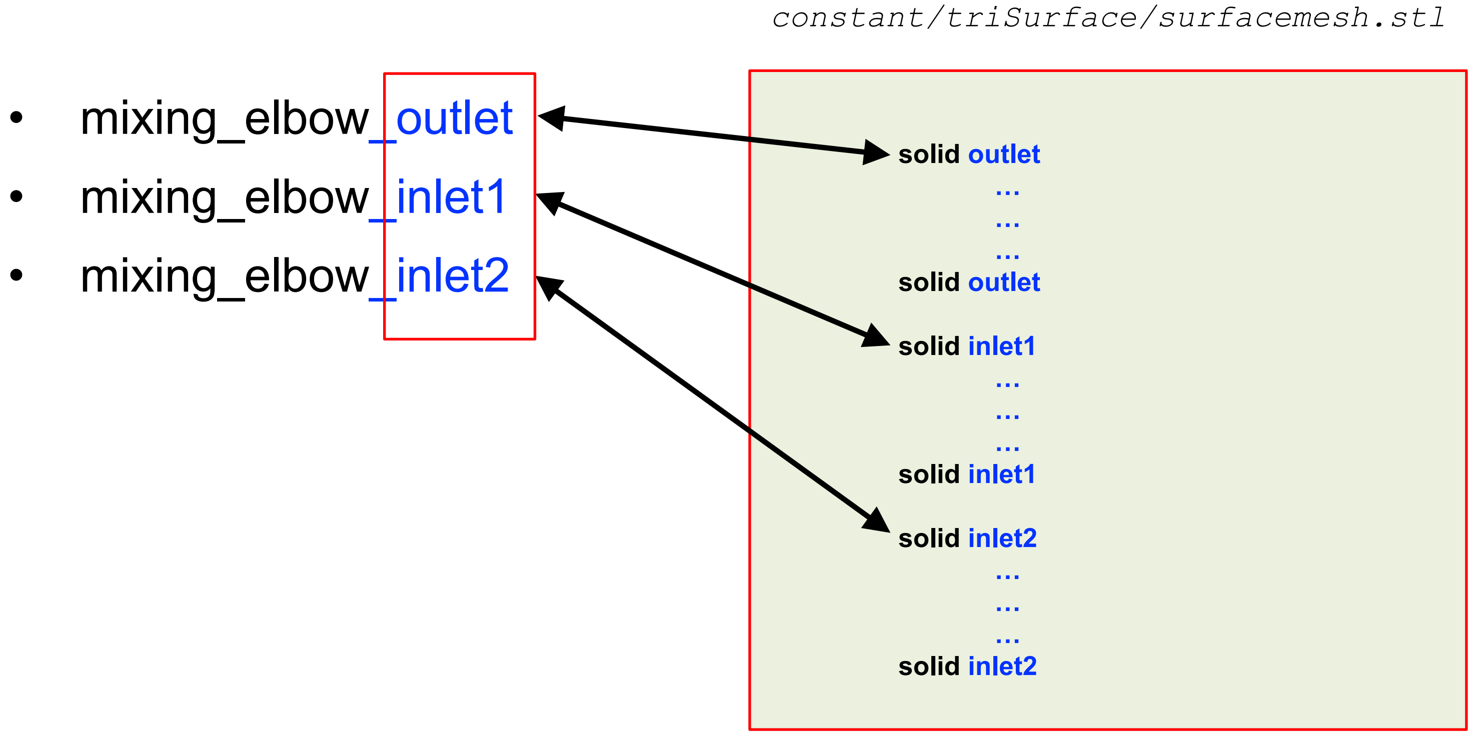
NOTE 1:
This is the name of the region or surface patch in the STL file

NOTE 2:
User-defined patch name. This is the final name of the patch.

snappyHexMesh guided tutorials

Mixing elbow.

- The mesh used in the previous case was a STL with multiple surfaces.
- In you do not create the regions in the geometry section of the dictionary *snappyHexMeshDict*, snappyHexMesh will automatically assign the names of the surface patches as follows:



constant/triSurface/surfacemesh.stl

- mixing_elbow_outlet
- mixing_elbow_inlet1
- mixing_elbow_inlet2

```
solid outlet
…
…
…
solid outlet
solid inlet1
…
…
…
solid inlet1
solid inlet2
…
…
…
solid inlet2
```

snappyHexMesh guided tutorials

Mixing elbow.

- The mesh used in the previous case was a STL with multiple surfaces.
- In the directory **`geometry`**, you fill find the file *`allss.stl`*, this STL has one single surface.
- Try to use this STL file to generate the mesh.
- You will notice that the final mesh has only one patch, namely **mixing_elbow** (or whatever name you chose).
- Also, it is not possible to have local control on the mesh refinement and boundary layer meshing.
- You will also face the conundrum that as there is only one surface patch, it is not possible to assign boundary conditions.

snappyHexMesh guided tutorials

Mixing elbow.

- To solve the problem of the single surface patch, you can use the utility `autoPatch`. To do so, you can proceed as follows:
  - `$> autoPatch 60 -overwrite`
- The option `-overwrite`, will copy the new mesh in the directory **`constant/polyMesh`**.
- The utility `autoPatch` will use an angle criterion to find the patches, and will assign the name **auto0**, **auto1**, **auto2** and **auto3** to the new patches.
- The angle criterion is similar to that of the utility `surfaceFeatures`.
- The only difference is that it uses the complement of the angle. So, the smaller the angle the more patches it will find.
- The naming convention is **autoN**, where N is the patch number.
- Remember, `autoPatch` will manipulate the mesh located in the directory **`constant/polyMesh`**.
- FYI, `autoPatch` does not un in parallel.

Exercises

- To get a feeling of the **includedAngle** value, try to change the value in the dictionary *surfaceFeaturesDict*.
- Remember the higher the **includedAngle** value, the more features you will capture.
- In the dictionary *snappyHexMeshDict*, change the value of **resolveFeatureAngle** (try to use a lower value), and check the mesh quality in the intersection between both pipes.
- In the **castellatedMeshControls** section, try to disable or modify the distance refinement of the **mixing_elbow** region (**refinementRegions**).
- What difference do you see in the output?
- Starting from the body fitted mesh, add 3 inflation layers at the walls (save the intermediate step).
- Try to add local surface refinement at the surface patch inlet2 (look at the STL file *constant/triSurface/surfacemesh_multi.stl*).
- Using paraview, extract the feature edge at the joint section of the pipes. Then, try to add local refinement at this feature edge.
- Try to add curvature refinement at the feature edge extracted from the surface patch inlet1.

snappyHexMesh guided tutorials

Exercises

- Use the STL file with a single surface (*surfacemesh_single.stl*) and generate the same mesh, do not add inflation layers.
  - Use the utility `autopatch` to split the mesh in different surface patches. To get a feeling on how to use this utility, use different angle values. Try to get four surface patches.
  - After splitting the mesh in four surface patches, rename the boundary patches using the utility `createPatch`.
  - After renaming the boundary patches, change the type of each one using the utility `foamDictionary`.
  - Starting from the body fitted mesh, add 3 inflation layers at the walls (do not save the intermediate step).
  - **Hints: if you do not know how to use the utilities createPatch and foamDictionary, look at the script file run_mesh_single_surface.sh**

- After generating the mesh, setup a simple incompressible simulation (with no turbulence model).
  - Set the inlet velocity to 1 at both inlet patches and use a dynamic viscosity value equal to 0.01. Run the simulation in steady and unsteady mode.

snappyHexMesh guided tutorials

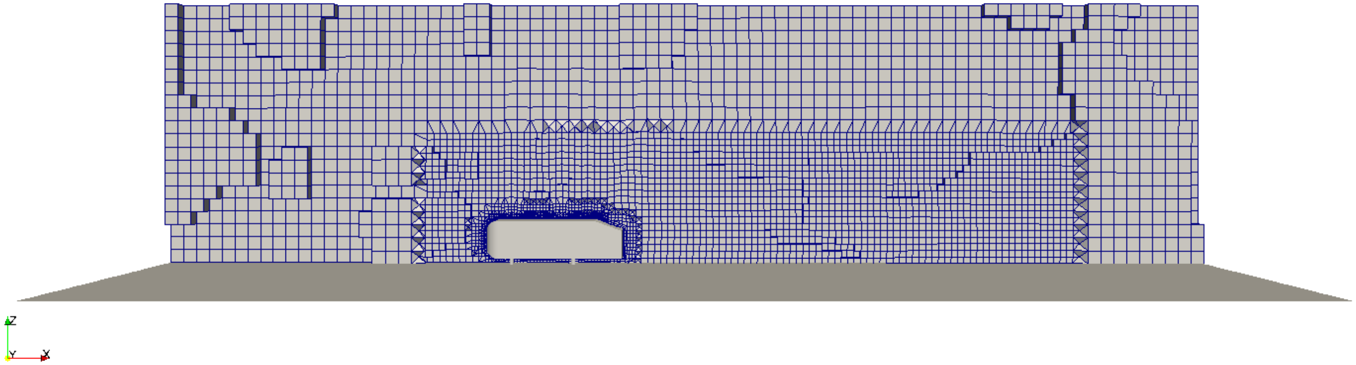
- Meshing with snappyHexMesh – Case 6.
- Ahmed body (external mesh).
- You will find this case in the directory:

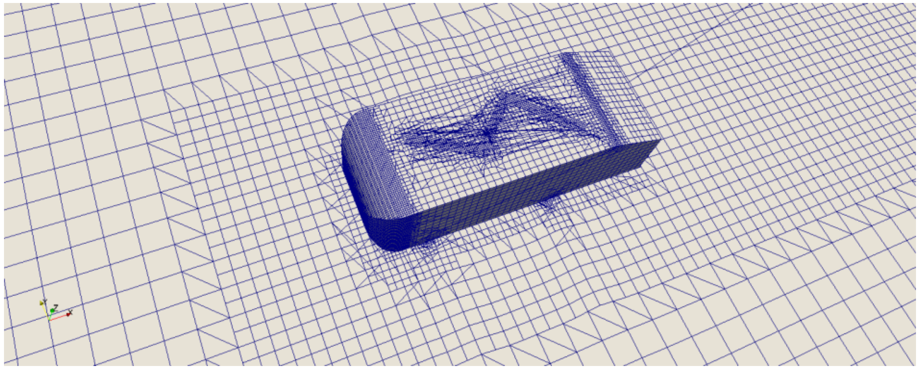
```
$PTOFC/101SHM/M4_ahmed
```

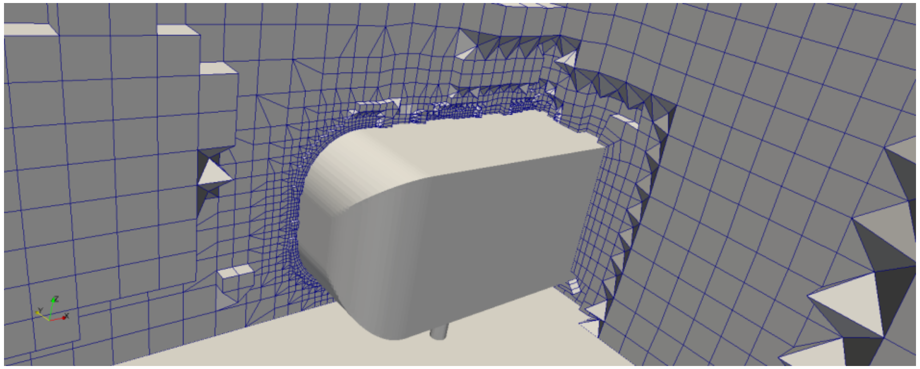
- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

snappyHexMesh guided tutorials

Ahmed body







- At this point, we all have a clear idea of how `snappyHexMesh` works.
- If not, please raise your hand.
- So let us go free styling and let us play around with this case.

snappyHexMesh guided tutorials

Ahmed body

- In our YouTube channel you will find a playlist with many videos for this case. The playlist is titled: CFD workflow tutorial using open-source tools.
- You can find our YouTube channel in the following link: https://www.youtube.com/channel/UCNNBm3KxVS1rGeCVUU1p61g
- In these videos, we show a few extra features and some tips and tricks to take the most out of `snappyHexMesh`.
- If you get lost, read the *`REAME.FIRST`* file that you will find in the working directory.
- The dictionaries *`snappyHexMeshDict`* and *`blockMeshDict`* used in this case are very clean and ready to use. So feel free to use them as your templates.
- Our best advice is not to get lost in all the options available in the dictionary *`snappyHexMeshDict`*. Most of the times the default options will work fine.
- That being said, you only need to read in the geometries, set the feature edges and surface refinement levels, choose in which surfaces you want to add the boundary layers, and choose how many layers you want to add.
- Final advices:
  - If you are working with a complicated geometry, add one layer at a time.
  - Use paraFoam/paraview to get visual references.
  - Always check the quality of your mesh.

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. ~~Mesh generation using snappyHexMesh.~~
6. ~~snappyHexMesh guided tutorials.~~
7. **Mesh conversion**
8. Geometry and mesh manipulation utilities

Mesh conversion

- OpenFOAM® gives users a lot of flexibility when it comes to meshing.
- You are not constrained to use OpenFOAM® meshing tools.
- To convert a mesh generated with a third-party software to OpenFOAM® format, you can use the OpenFOAM® mesh conversion utilities.
- If your format is not supported, you can write your own conversion tool.
- By the way, many of the commercially available meshers can save the mesh in OpenFOAM® format or in a compatible format.
- In the directory **$PTOFC/mesh_conversion_sandbox** you will find a few meshes generated using the most popular third-party mesh generation applications.
- Feel free to play with these meshes.
- In the *README.FIRST* file of each case, you will find the instructions of how to convert the mesh.

- Remember to always check the file *boundary* after converting the mesh. You will need to change the **name** and **type** of the surface patches according to what you would intent to do.
- Also, to convert the mesh you need to be in the top level of the case directory, and you need to give to the conversion utility the path (absolute or relative) of the mesh to be converted.

Mesh conversion

- In the directory `$FOAM_UTILITIES/mesh/conversion` you will find the source code for the mesh conversion utilities:

  - **ansysToFoam**
  - **cfx4ToFoam**
  - **datToFoam**
  - **fluent3DMeshToFoam**
  - **fluentMeshToFoam**
  - **foamMeshToFluent**
  - **foamToStarMesh**
  - **foamToSurface**
  - **gambitToFoam**
  - **gmshToFoam**
  - **ideasUnvToFoam**
  - **kivaToFoam**
  - **mshToFoam**
  - **netgenNeutralToFoam**
  - **Optional/ccm26ToFoam**
  - **plot3dToFoam**
  - **sammToFoam**
  - **star3ToFoam**
  - **star4ToFoam**
  - **tetgenToFoam**
  - **vtkUnstructuredToFoam**
  - **writeMeshObj**

- Take your time and read the instructions/comments contained in the source code of the mesh conversion utilities so you can understand how to use these powerful tools.

Mesh conversion

- Let us convert to OpenFOAM® format a mesh generated using Salome.
- You will find this case in the directory:

```
$PTOFC/mesh_conversion_sandbox/M1_mixing_elbow_salome
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- Remember to export the mesh in UNV format in Salome. ⚠
- Then use the utility `ideasUnvToFoam` to convert the mesh to OpenFOAM® native format.
- In the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> foamCleanPolyMesh
3.  $> ideasUnvToFoam ../../meshes_and_geometries/salome_elbow3d/Mesh_1.unv
4.  $> checkMesh
5.  $> paraFoam
```

- Remember to always check the file *boundary* after converting the mesh. ⚠
- To convert the mesh, you need to be in the top level of the case directory, and you need to give the path (absolute or relative) of the mesh to be converted.

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- `ideasUnvToFoam` output.

```
Create time

Processing tag:2411
Starting reading points at line 3.
Read 31136 points.

Processing tag:2412
Starting reading cells at line 62278.
First occurrence of element type 11 for cell 1 at line 62279
First occurrence of element type 41 for cell 361 at line 63359
First occurrence of element type 111 for cell 20933 at line 104503
Read 151064 cells and 20572 boundary faces.

Processing tag:2467
Starting reading patches at line 406633.
For group 1 named pipe trying to read 19778 patch face indices.
For group 2 named inlet1 trying to read 358 patch face indices.
For group 3 named inlet2 trying to read 78 patch face indices.
For group 4 named outlet trying to read 358 patch face indices.

Sorting boundary faces according to group (patch)
0: pipe is patch
1: inlet1 is patch
2: inlet2 is patch
3: outlet is patch

Constructing mesh with non-default patches of size:
    pipe        19778
    inlet1      358
    inlet2      78
    outlet      358

End
```

Internal cells and boundary faces read

Boundary patches detected

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- `checkMesh` output.

```
Mesh stats
    points:             31136
    faces:              312414
    internal faces:     291842
    cells:              151064
    faces per cell:     4
    boundary patches:   4
    point zones:        0
    face zones:         0
    cell zones:         0

Overall number of cells of each type:
    hexahedra:     0
    prisms:        0
    wedges:        0
    pyramids:      0
    tet wedges:    0
    tetrahedra:    151064
    polyhedra:     0

Checking topology...
    Boundary definition OK.
    Cell to face addressing OK.
    Point usage OK.
    Upper triangular ordering OK.
    Face vertices OK.
    Number of regions: 1 (OK).
```

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- `checkMesh` output.

```
 Checking patch topology for multiply connected surfaces...
    Patch                Faces    Points   Surface topology
    pipe                 19778    9938     ok (non-closed singly connected)
    inlet1               358      200      ok (non-closed singly connected)
    inlet2               78       50       ok (non-closed singly connected)
    outlet               358      200      ok (non-closed singly connected)

Checking geometry...
    Overall domain bounding box (0 -0.414214 -0.5) (5 5 0.5)
    Mesh has 3 geometric (non-empty/wedge) directions (1 1 1)
    Mesh has 3 solution (non-empty) directions (1 1 1)
    Boundary openness (-1.0302e-17 -6.17232e-17 -1.77089e-16) OK.
    Max cell openness = 2.32045e-16 OK.
    Max aspect ratio = 4.67245 OK.
    Minimum face area = 0.000286852. Maximum face area = 0.010949.  Face area magnitudes OK.
    Min volume = 2.74496e-06. Max volume = 0.00035228.  Total volume = 6.75221.  Cell volumes OK.
    Mesh non-orthogonality Max: 54.2178 average: 15.1295
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 0.649359 OK.
    Coupled point location match (average 0) OK.

Mesh OK.

End
```

Everything is OK

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- The *boundary* file.

```
4
(
    pipe
    {
        type            patch;
        nFaces          19778;
        startFace       291842;
    }
    inlet1
    {
        type            patch;
        nFaces          358;
        startFace       311620;
    }
    inlet2
    {
        type            patch;
        nFaces          78;
        startFace       311978;
    }
    outlet
    {
        type            patch;
        nFaces          358;
        startFace       312056;
    }
)
```

Number of boundary patches

Name of the boundary patches

- In this case, the utility recognized the name of the boundary patches.
- If you do not like the names feel free to change them.
- Remember, do not use spaces of strange symbols.

Mesh conversion

Case 1. Mixing elbow (internal mesh).

- The *boundary* file.

```
4
(
    pipe
    {
        type            patch;
        nFaces          19778;
        startFace       291842;
    }
    inlet1
    {
        type            patch;
        nFaces          358;
        startFace       311620;
    }
    inlet2
    {
        type            patch;
        nFaces          78;
        startFace       311978;
    }
    outlet
    {
        type            patch;
        nFaces          358;
        startFace       312056;
    }
)
```

Base type of the boundary patches

- In this case, the utility automatically assigned the **base type patch** to all boundary patches.
- Feel free to change the **base type** according to your needs.
- In this case, it will be wise to change the **base type** of patch **pipe** to **wall**.

Mesh conversion

Exercises

- Remember, you can change the name and type of the boundary patches manually, but as we want to do things automatically, we will use the utilities `createPatch` and `foamDictionary`
  - After converting the mesh to OpenFOAM® format, rename the boundary patches using the utility `createPatch`.
  - After converting the mesh to OpenFOAM® format, change the type of each boundary patch using the utility `foamDictionary`.
- After converting the mesh to OpenFOAM® format, add 5 inflation layers at the walls (do not save the intermediate step).
- Check the mesh quality before and after adding the inflation layers.
- After generating the mesh, setup a simple incompressible simulation (with no turbulence model).
  - Set the inlet velocity to 1 at both inlet patches and use a dynamic viscosity value equal to 0.01. Run the simulation in steady and unsteady mode.

Roadmap

1. ~~Meshing preliminaries~~
2. ~~What is a good mesh?~~
3. ~~Mesh quality assessment in OpenFOAM®~~
4. ~~Mesh generation using blockMesh.~~
5. ~~Mesh generation using snappyHexMesh.~~
6. ~~snappyHexMesh guided tutorials.~~
7. ~~Mesh conversion~~
8. Geometry and mesh manipulation utilities

Geometry and mesh manipulation utilities

- First of all, by mesh manipulation we mean modifying a valid OpenFOAM® mesh.
- These modifications can be scaling, rotation, translation, mirroring, topological changes, mesh refinement and so on.
- In the directory **`$FOAM_UTILITIES/mesh/manipulation`** you will find the mesh manipulation utilities. Just to name a few:

  - **autoPatch**
  - **checkMesh**
  - **createBaffles**
  - **mergeMeshes**
  - **mergerOrSplitBaffles**
  - **mirrorMesh**
  - **polyDualMesh**
  - **refineMesh**
  - **renumberMesh**
  - **rotateMesh**
  - **setSet**
  - **splitMesh**
  - **splitMeshRegions**
  - **stitchMesh**
  - **subsetMesh**
  - **topoSet**
  - **transformPoints**

Geometry and mesh manipulation utilities

- In the directory **$FOAM_UTILITIES/mesh/manipulation** you will find the following mesh manipulation utilities.
- Inside each utility directory you will find a **.C* file with the same name as the directory. This is the main file, where you will find the top-level source code and a short description of the utility.
- For instance, in the directory **checkMesh**, you will find the file *checkMesh.C*, which is the source code of the utility checkMesh. In the source code you will find the following description:

```
Checks validity of a mesh.


Usage
    - checkMesh [OPTION]

    \param -allGeometry \n
    Checks all (including non finite-volume specific) geometry

    \param -allTopology \n
    Checks all (including non finite-volume specific) addressing

    \param -meshQuality \n
    Checks against user defined (in \a system/meshQualityDict) quality settings

    \param -region \<name\> \n
    Specify an alternative mesh region.
```

Geometry and mesh manipulation utilities

- In OpenFOAM® it is also possible to manipulate the geometries in STL format.
- These modifications can be scaling, rotation, translation, mirroring, topological changes, normal orientation, and so on.
- In the directory **`$FOAM_UTILITIES/surface`** you will find the mesh manipulation utilities. Just to name a few:

  - **surfaceAdd**
  - **surfaceAutoPatch**
  - **surfaceBooleanFeatures**
  - **surfaceCheck**
  - **surfaceConvert**
  - **surfaceFeatureConvert**
  - **surfaceFeatures**
  - **surfaceInertia**
  - **surfaceMeshConvert**
  - **surfaceMeshExport**
  - **surfaceMeshTriangulate**
  - **surfaceOrient**
  - **surfaceSplitByPatch**
  - **surfaceSubset**
  - **surfaceToPatch**
  - **surfaceTransformPoints**

Geometry and mesh manipulation utilities

- In the directory **`$FOAM_UTILITIES/surface`** you will find the following surface manipulation utilities.
- Inside each utility directory you will find a *`*.C`* file with the same name as the directory. This is the main file, where you will find the top-level source code and a short description of the utility.
- For instance, in the directory **`surfaceTransformPoints`**, you will find the file *`surfaceTransformPoints.C`*, which is the source code of the utility `surfaceTransformPoints`. In the source code you will find the following description:

> **Transform (scale/rotate) a surface.**
>
> **Like transformPoints but for surfaces.**
>
> **The rollPitchYaw option takes three angles (degrees):**
>
> **- roll (rotation about x) followed by**
>
> **- pitch (rotation about y) followed by**
>
> **- yaw (rotation about z)**
>
> **The yawPitchRoll does yaw followed by pitch followed by roll.**

Geometry and mesh manipulation utilities

- Let us do some surface manipulation.
- For this we will use the ahmed body tutorial.
- You will find this case in the directory:

```
$PTOFC/mesh_quality_manipulation/M5_ahmed_body_transform
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Geometry and mesh manipulation utilities

Geometry manipulation in OpenFOAM®

- We will now manipulate a STL geometry. In the terminal type:

1.
```
$> foamCleanTutorials
```
2.
```
$> surfaceMeshInfo ./constant/triSurface/ahmed_body.stl
```
3.
```
$> surfaceCheck ./constant/triSurface/ahmed_body.stl
```
4.
```
$> surfaceTransformPoints -rollPitchYaw '(0 0 15)’
   ./constant/triSurface/ahmed_body.stl rotated.stl
```
5.
```
$> surfaceTransformPoints -translate '(0 0.12 0)'
   ./constant/triSurface/ahmed_body.stl   translated.stl
```
6.
```
$> surfaceTransformPoints -scale '(0.9 1.1 1.3)'
    ./constant/triSurface/ahmed_body.stl   scaled.stl
```
7.
```
$> surfaceInertia -density 2700 –noFunctionObjects
   ./constant/triSurface/ahmed_body.stl
```
8.
```
$> surfaceOrient ./constant/triSurface/ahmed_body_wrong_normals.stl
   out.stl ‘(1e10 1e10 1e10)’
```

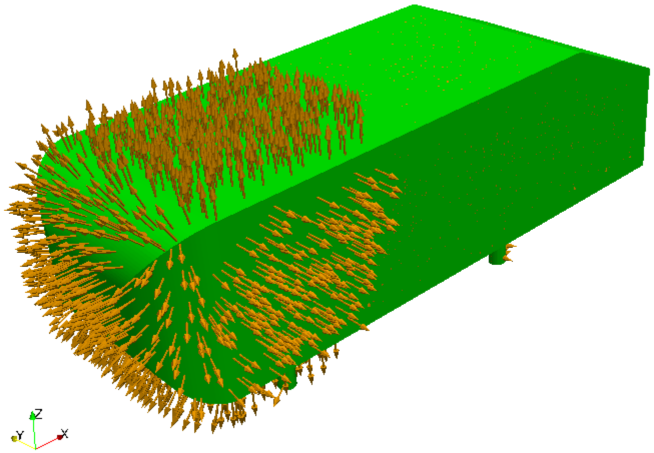
Geometry and mesh manipulation utilities

Geometry manipulation in OpenFOAM®

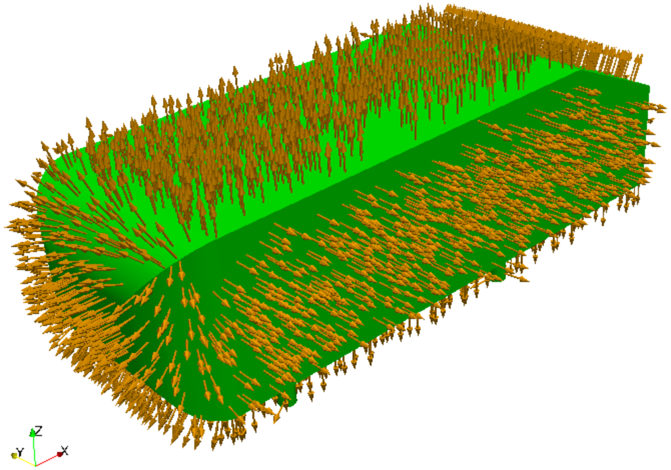
- In step 2 we use the utility `surfaceMeshInfo` to get general information about the STL (such as number of faces and so on).
- In step 3 we use the utility `surfaceCheck` to check the STL file.
- In step 4 we use the utility `surfaceTransformPoints` to rotate the STL. We read in the STL *./constant/triSurface/ahmed_body.stl* and we write out the STL *rotated.stl*
- In step 5 we use the utility `surfaceTransformPoints` to translate the STL. We read in the STL *./constant/triSurface/ahmed_body.stl* and we write out the STL *translated.stl*
- In step 6 we use the utility `surfaceTransformPoints` to scale the STL. We read in the STL *./constant/triSurface/ahmed_body.stl* and we write out the STL *scaled.stl*
- In step 7 we use the utility `surfaceInertia` to compute the inertia of the STL. We read in the STL *./constant/triSurface/ahmed_body.stl*. Notice that we need to give a reference density value.
- In step 8 we use the utility `surfaceOrient` to orient the normals of the STL in the same way. We read in the STL `./constant/triSurface/ahmed_body_wrong_normals.stl` and we write out the STL *out.stl*. Notice that we give an outside point or `‘(1e10 1e10 1e10)’`, if this point is outside the STL all normals will be oriented outwards, if the point is inside the STL all normals will be oriented inwards.

Geometry manipulation in OpenFOAM®

- Pay particular attention to step 8.
- We already have seen that snappyHexMesh computes surface angles using the surface normals as a reference, so it is extremely important to have the normals oriented in the same way and preferably outwards.



`ahmed_body_wrong_normals.stl`



STL after orienting all normals in the same direction.

Geometry manipulation in OpenFOAM®

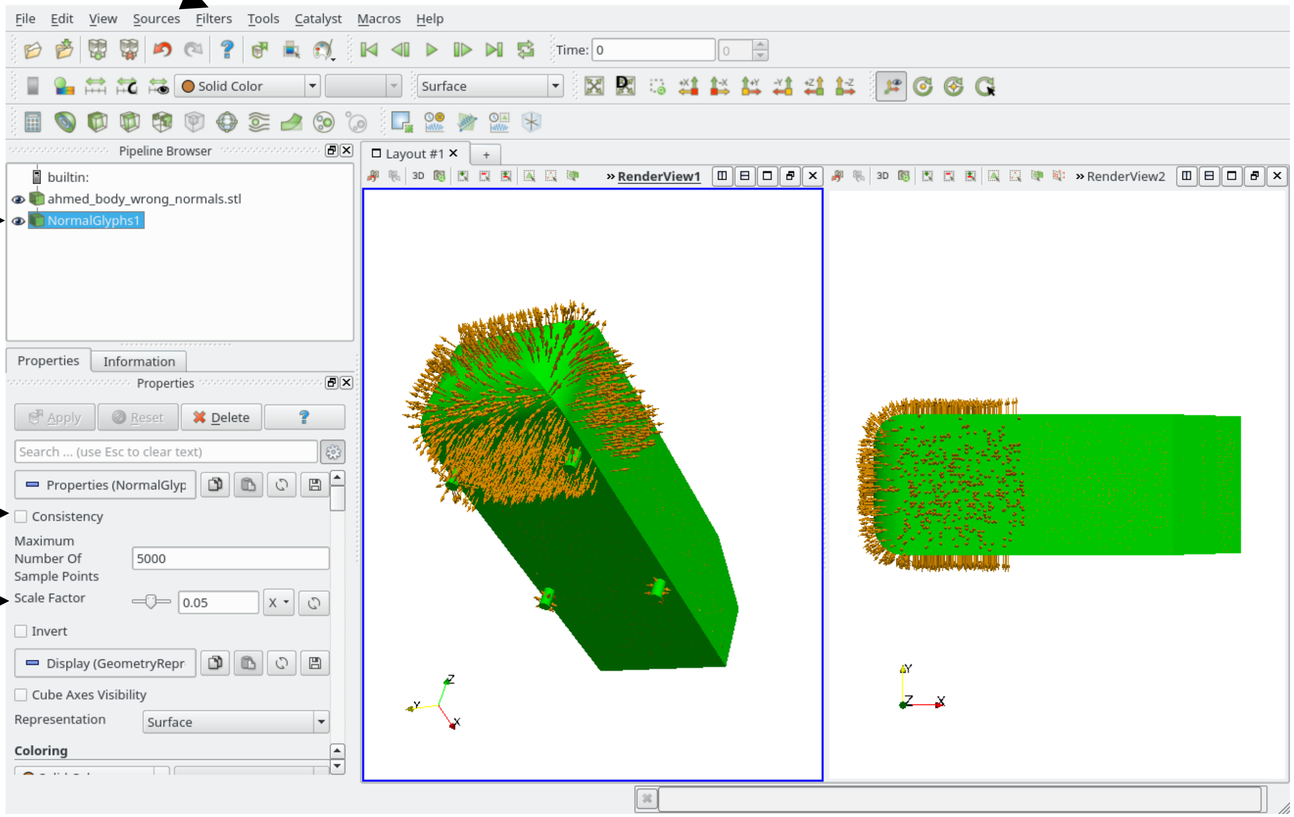
- To plot the normals in paraview/paraFoam you can use the filter `Normal Glyphs`

Select the `Normal Glyphs` from the filter menu

Apply the `Normal Glyphs` filter to the STL

Uncheck this option

Scale the vectors to fit the screen



Time: 0

builtin:
ahmed_body_wrong_normals.stl
NormalGlyphs1

☐ Consistency

Maximum Number Of Sample Points: 5000

Scale Factor: 0.05

☐ Invert

☐ Cube Axes Visibility

Representation: Surface

Coloring

Geometry and mesh manipulation utilities

- Let us do some mesh manipulation.
- For this we will use the 2D cylinder tutorial.
- You will find this case in the directory:

```
$PTOFC/mesh_quality_manipulation/M7_cylinder_transform
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Mesh manipulation in OpenFOAM®

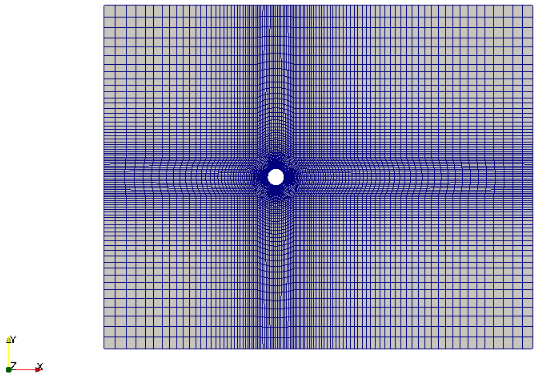
- We will now manipulate a mesh. In the terminal type:

```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> transformPoints -rollPitchYaw '(0 0 90)'
4.  $> transformPoints -scale '(0.01 0.01 0.01)'
5.  $> transformPoints -translate '(0 0 1)'
6.  $> createPatch -noFunctionObjects –overwrite
7.  $> checkMesh
8.  $> paraFoam
```

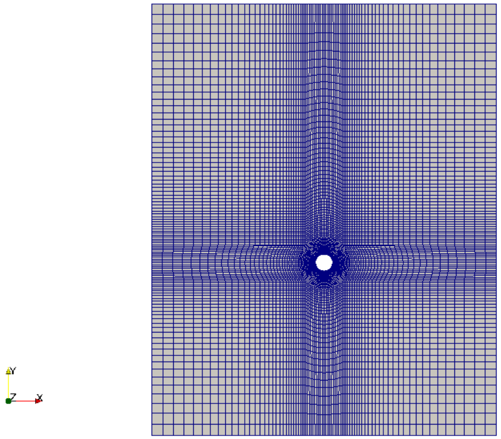
- In step 3 we use the utility `transformPoints` to rotate the mesh. We rotate the mesh by 90° about the **Z** axis.
- In step 4 we use the utility `transformPoints` to scale the mesh. We scale the mesh by a factor of `'(0.01 0.01 0.01)'`.
- In step 5 we use the utility `transformPoints` to translate the mesh. We translate the mesh by the vector `'(0 0 1)'`.
- In step 6 we use the utility `createPatch` to rename the patches of the mesh. This utility reads the dictionary `system/createPatchDict`. Instead of using the utility `createPatch` we could have modified the *`boundary`* file directly.
- This case is ready to run using the solver `buoyantBoussinesqPimpleFoam`.

Geometry and mesh manipulation utilities

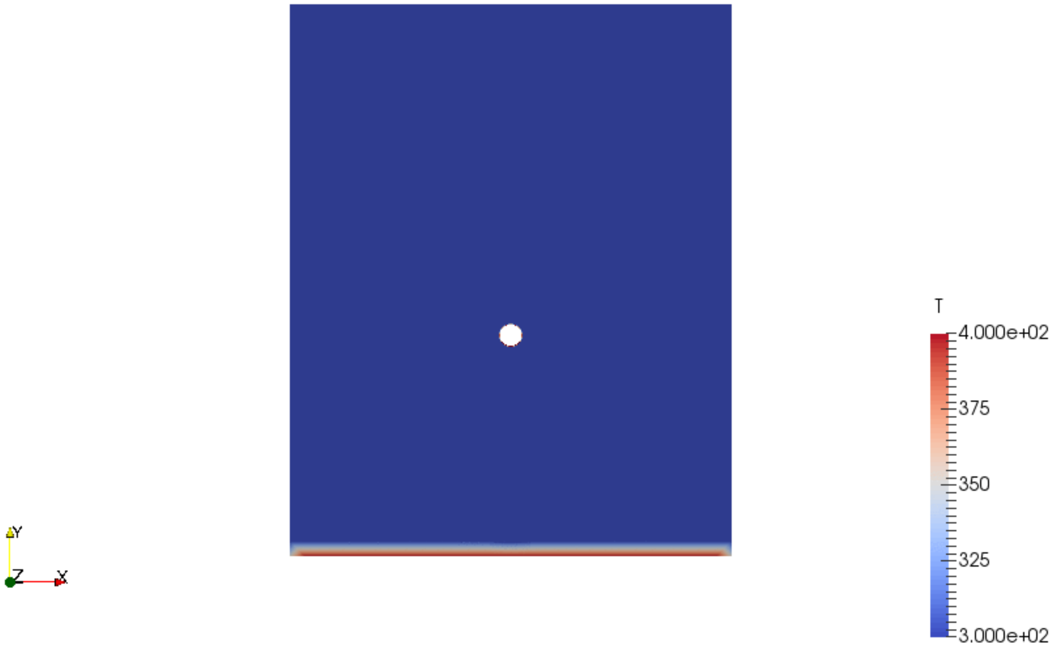
Mesh manipulation in OpenFOAM®



Original mesh



Transformed mesh



After renaming the patches and transforming the mesh, we can use it to conduct this buoyant flow simulation

www.wolfdynamics.com/wiki/heated_cyl/ani1.gif

Module 4

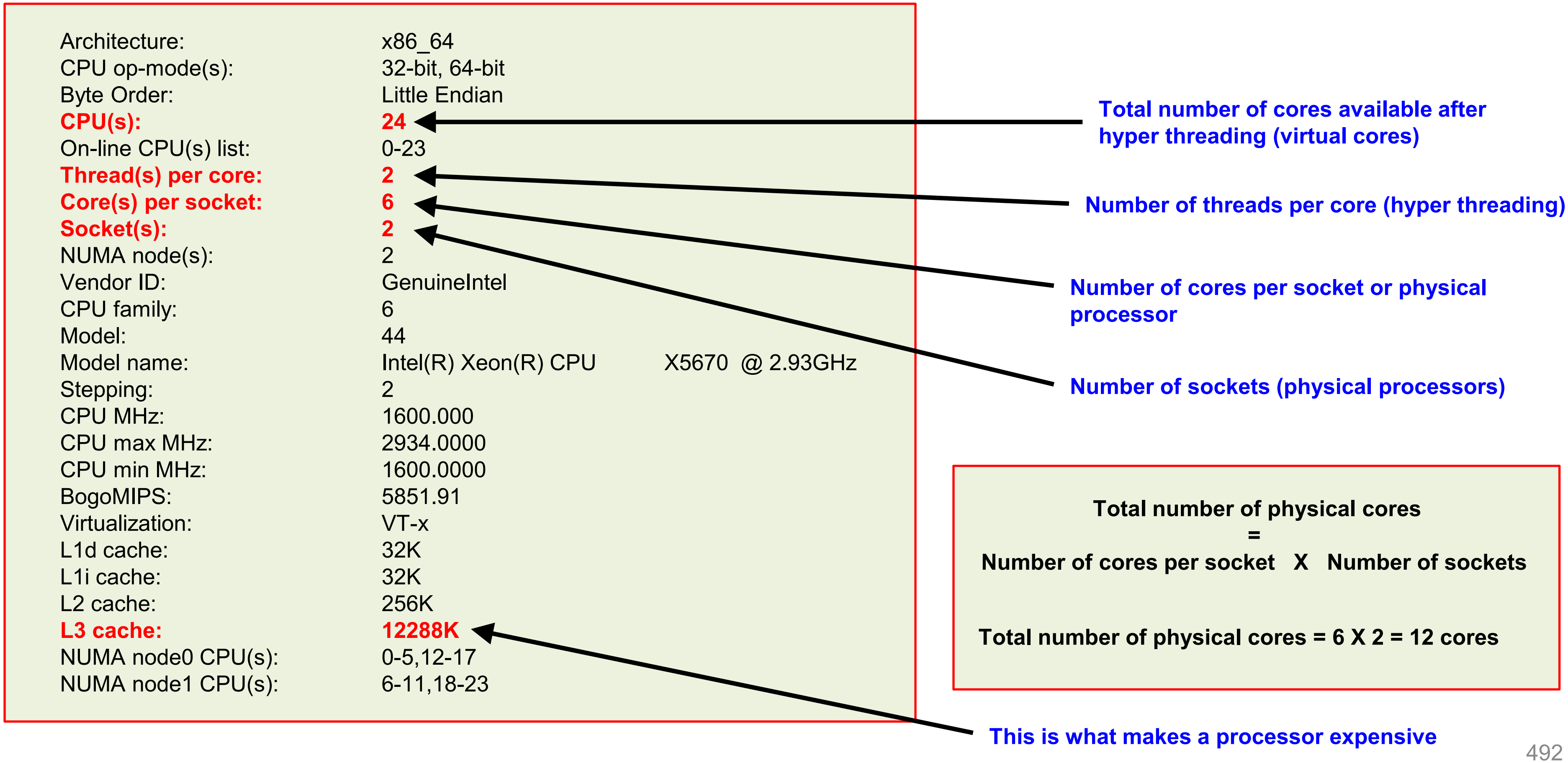
Running in parallel

Roadmap

1. **Running in parallel**

Running in parallel

- First of all, to know how many processors/cores you have available in your computer, type in the terminal:
  - `$> lscpu`
- The output for this particular workstation is the following:



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU         X5670  @ 2.93GHz
Stepping:              2
CPU MHz:               1600.000
CPU max MHz:           2934.0000
CPU min MHz:           1600.0000
BogoMIPS:              5851.91
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
```

Total number of cores available after hyper threading (virtual cores)

Number of threads per core (hyper threading)

Number of cores per socket or physical processor

Number of sockets (physical processors)

Total number of physical cores
=
Number of cores per socket X Number of sockets

Total number of physical cores = 6 X 2 = 12 cores

This is what makes a processor expensive

Running in parallel

- OpenFOAM® does not take advantage of hyper threading technology (HT).
- HT is basically used by the OS to improve multitasking performance.
- This is what we have in the workstation of the previous example:
    - 24 virtual cores (hyper threaded)
    - 12 physical cores
- To take full advantage of the hardware, we use the maximum number of physical cores (12 physical cores in this case) when running in parallel.
- If you use the maximum number of virtual cores, OpenFOAM® will run but it will be slower in comparison to running with the maximum number of physical cores (or even less cores).
- Same rule applies when running in clusters/super computers, so always read the hardware specifications to know the limitations.

Running in parallel

Why use parallel computing?

- **Solve larger and more complex problems (scale-up):**

  Thanks to parallel computing we can solve bigger problems (scalability). A single computer has limited physical memory, many computers interconnected have access to more memory (distributed memory).

- **Provide concurrency (scale-out):**

  A single computer or processor can only do one thing at a time. Multiple processors or computing resources can do many things simultaneously.

- **Save time (speed-up):**

  Run faster (speed-up) and increase your productivity, with the potential of saving money in the design process.

- **Save money:**

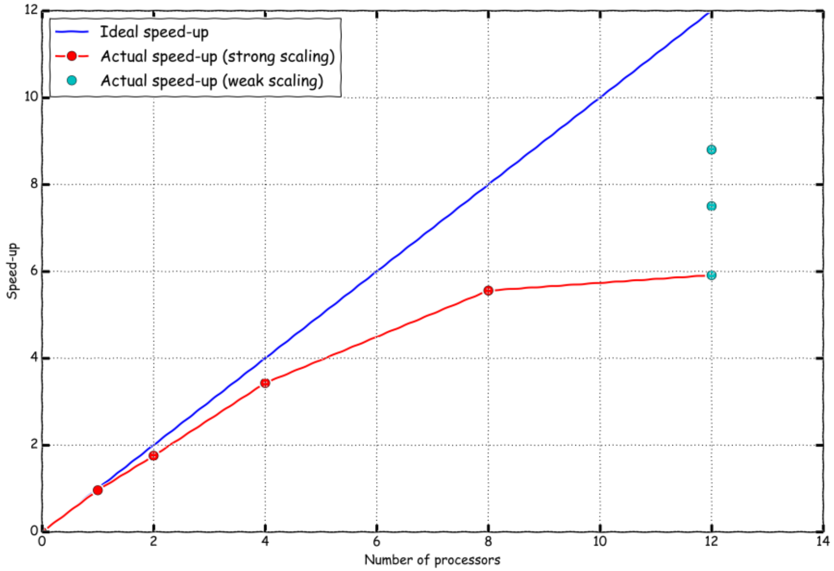
  In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components.

- **Limits to serial computing:**

  Both physical and practical reasons pose significant constraints to simply building ever faster serial computers (*e.g*, transmission speed, CPU clock rate, limits to miniaturization, hardware cooling).

Running in parallel

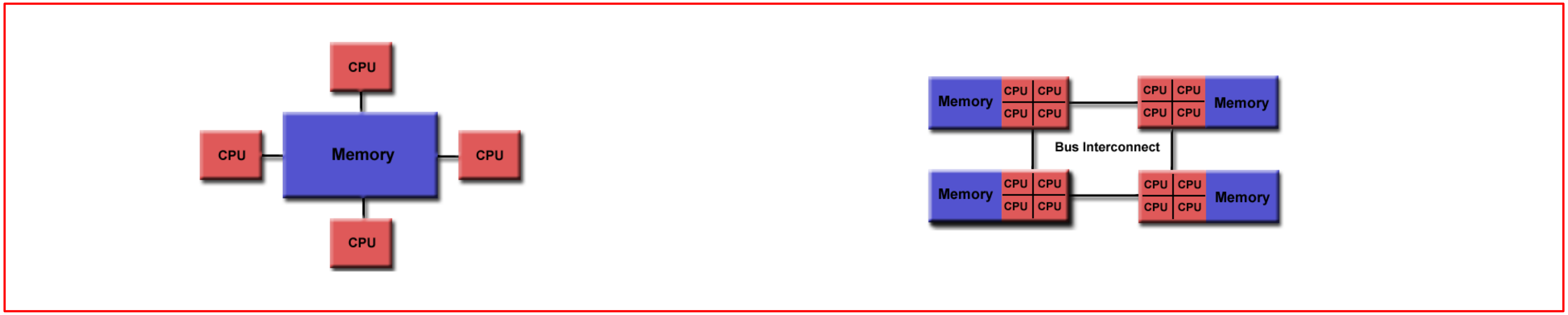
Speed-up and scalability example



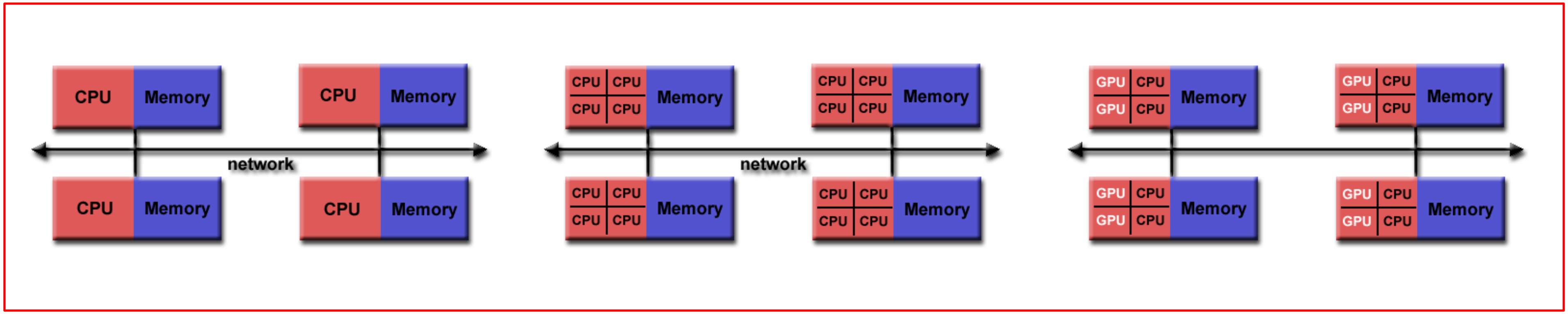
- In the context of high-performance computing (HPC), there are two common metrics that measure the scalability of the application:
  - **Strong scaling (Amdahl's law)**: which is defined as how the solution time varies with the number of processors for a fixed problem size (number of cells in CFD)
  - **Weak scaling (Gustafson's law)**: which is defined as how the solution time varies with the number of processors for a fixed problem size per processor (or increasing the problem size with a fix number of processors).
- In this example, when we reach 12 cores inter-processor communication slow-downs the computation. But if we increase the problem size for a fix number of processors, we will increase the speed-up.
- The parallel case with 1 processor runs slower than the serial case due to the extra overhead when calling the MPI library.

Running in parallel

- The method of parallel computing used by OpenFOAM® is known as domain decomposition, in which the geometry and associated fields are broken into pieces and distributed among different processors.



Shared memory architectures – Workstations and portable computers



Distributed memory architectures – Clusters and super computers

Running in parallel

Some facts about running OpenFOAM® in parallel:

- Applications generally do not require parallel-specific coding. The parallel programming implementation is hidden from the user.
- In order to run in parallel you will need an MPI library installation in your system.
- Most of the applications and utilities run in parallel.
- If you write a new solver, it will be in parallel (most of the times).
- We have been able to run in parallel up to 15000 processors.
- We have been able to run OpenFOAM® using single GPU and multiple GPUs.
- Do not ask about scalability, that is problem/hardware specific.
- If you want to learn more about MPI and GPU programming, do not look in my direction.
- And of course, to run in parallel you need the hardware.

Running in parallel

To run OpenFOAM® in parallel you will need to:

- **Decompose the domain.**

  To do so we use the `decomposePar` utility. You also need the dictionary *`decomposeParDict`* which is located in the **`system`** directory.

- **Distribute the jobs among the processors or computing nodes.**

  To do so, OpenFOAM® uses the standard message passing interface (MPI). By using MPI, each processor runs a copy of the solver on a separate part of the decomposed domain.

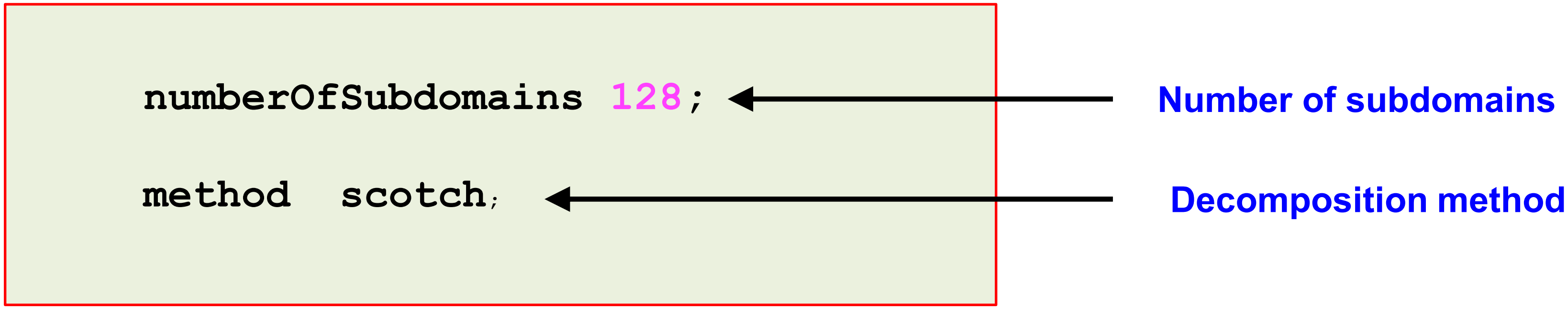
- **Additionally, you might want to reconstruct (put back together) the decomposed domain.**

  This is done by using the `reconstrucPar` utility. You do not need a dictionary to use this utility.

Running in parallel

Domain Decomposition in OpenFOAM®

- The mesh and fields are decomposed using the `decomposePar` utility.
- They are broken up according to a set of parameters specified in a dictionary named *decomposeParDict* that is located in the **system** directory of the case.
- In the *decomposeParDict* dictionary the user must set the number of domains in which the case should be decomposed (using the keyword **numberOfSubdomains**). The value used should correspond to the number of physical cores available.



```
numberOfSubdomains 128;

method  scotch;
```

Number of subdomains

Decomposition method

- In this example, we are subdividing the domain in 128 subdomains, therefore we should have 128 physical cores available.
- The main goal of domain decomposition is to minimize the inter-processors communication and the processor workload.

Running in parallel

Domain Decomposition Methods

- These are the decomposition methods available in OpenFOAM® 8. To name a few:
  - hierarchical
  - manual
  - metis
  - multiLevel
  - none
  - scotch
  - simple
  - structured

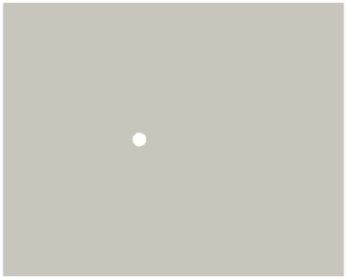
We highly recommend you to use this method. The only input that requires from the user is the number of subdomains/cores. This method attempts to minimize the number of processor boundaries.

- If you want more information about each decomposition method, just read the source code:
  - **`$WM_PROJECT_DIR/src/parallel/decompose/`**

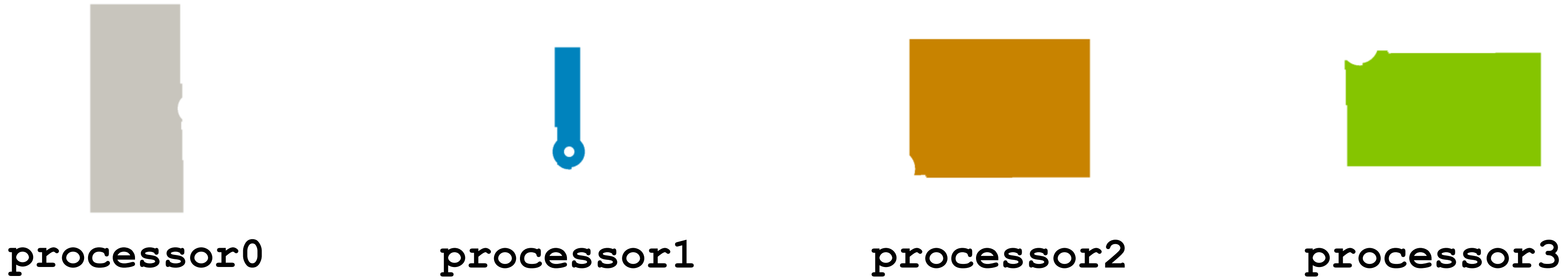
Running in parallel

Running in parallel – Gathering all together

The information inside the directories **`polyMesh/`** and **`0/`** is decomposed using the utility `decomposePar`



`decomposePar`

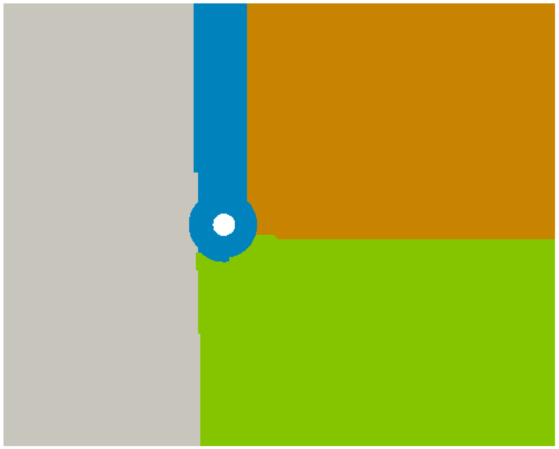


- Inside each **`processorN`** directory you will have the mesh information, boundary conditions, initial conditions, and the solution for that processor.

Running in parallel

Running in parallel – Gathering all together

- After decomposing the mesh, we can run in parallel using MPI.



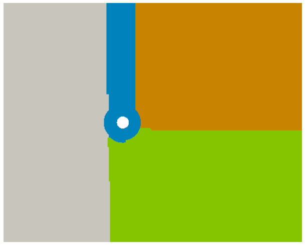
- The interface between each region, is know as halo zone.
- The inter-processor communication in the halo zone is managed by the MPI library.
- Remember, the main goal is to minimize the halo zone, therefore, the inter-processor communication.

```
$> mpirun -np <NPROCS> <application/utility> -parallel
```

- The number of processors to use or **<NPROCS>**, needs to be the same as the number of partitions (**numberOfSubdomains**).
- Do not forget to use the flag **`-parallel`**.

Running in parallel

Running in parallel – Gathering all together



- In the decomposed case, you will find the mesh information, boundary conditions, initial conditions, and the solution for every processor.
- The information is inside the directory **`processorN`** (where **`N`** is the processor number).

```
reconstructPar
```

- When you reconstruct the case, you glue together all the information contained in the decomposed case.
- All the information (mesh, boundary conditions, initial conditions, and the solution), is transfer to the original case folder (**`polyMesh`** and time solution directories).

Running in parallel

Running in parallel – Gathering all together

- Summarizing, to run in parallel we proceed in the following way:

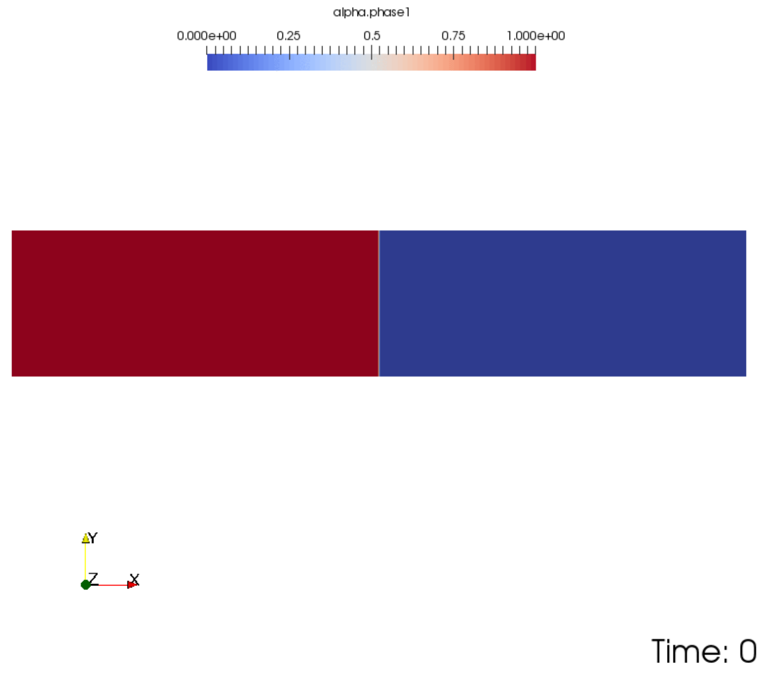
1. `$> decomposePar`
2. `$> mpirun -np <NPROCS> <application/utility> -parallel`
3. `$> reconstructPar`

- When running in parallel, do not forget to add this flag.
- If you do not add this flag, the application will run, but it will execute <NPROCS> duplicates of the same job (it will be slower).

- You can do the post-processing and visualization on the decomposed case or reconstructed case. We are going to address this later on.
- If you are dealing with moving bodies where the mesh topology is changed or if you are using AMR, you will need to use `reconstructParMesh` before `reconstrucPar`.

Running in parallel

Kelvin Helmholtz instability in a coarse mesh



Volume fraction

www.wolfdynamics.com/wiki/kelvin_helmholtz/ani1.gif

Processors	Clock time (seconds)	Mesh size in x, y, and z directions
1	955	800 X 160 X 1
2	564	800 X 160 X 1
4	333	800 X 160 X 1
8	234	800 X 160 X 1
12	244	800 X 160 X 1

You will find this case in the directory: **`$PTOFC/parallel/kelvin_helmholtz`**

Running in parallel

Visualization of a parallel case

- The traditional way is to first reconstruct the case and then do the post-processing and visualization on the reconstructed case.
- To do so, we type in the terminal:

```
1. | $> reconstructPar
2. | $> paraFoam
```

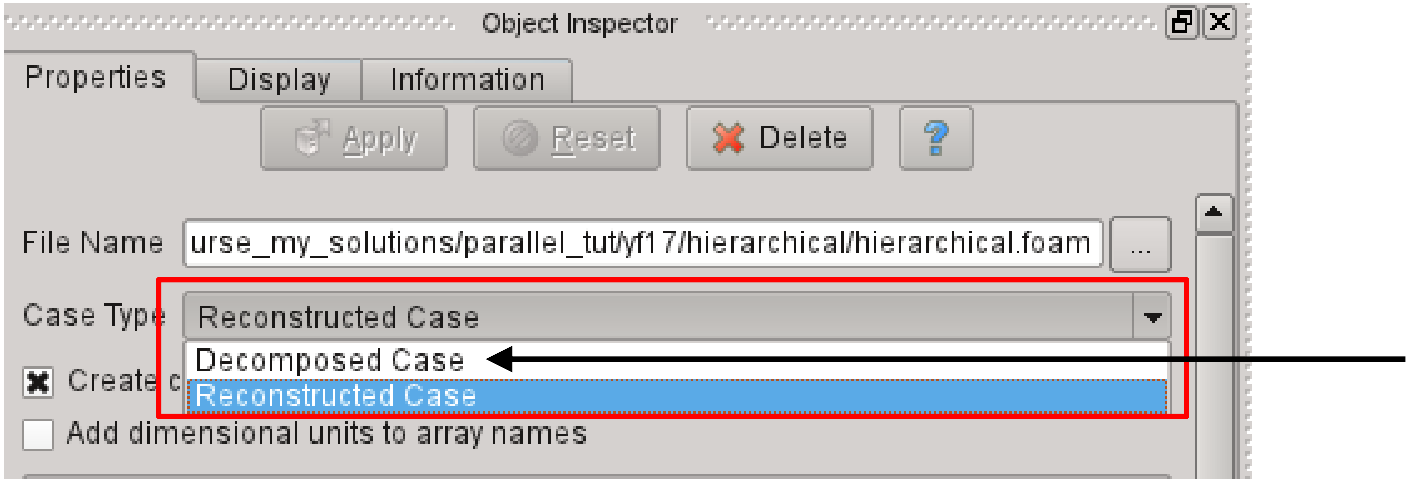
- Step 1 reconstruct the case. Remember, you can choose to reconstruct all the time steps, the last time step or a range of time steps.
- In step 2, we use `paraFoam` to visualize the reconstructed case.

Running in parallel

Visualization of a parallel case

- An alternative way to visualize the solution, is by proceeding in the following way
  - `$> paraFoam -builtin`

- The option `-builtin` let us post-process the decomposed case directly.
- Remember, you will need to select on the object inspector the `Decomposed Case` option.



File Name urse_my_solutions/parallel_tut/yf17/hierarchical/hierarchical.foam

Case Type Reconstructed Case

Decomposed Case

Reconstructed Case

☒ Create c

☐ Add dimensional units to array names

Running in parallel

Visualization of a parallel case

- Both of the previous methods are valid.
- When we use the option `–builtin` with `paraFoam`, we have the option to work on the decomposed case directly. In other words, we do not need to reconstruct the case.
- This option is also faster than running `paraFoam` with no flags.
- But wait, there is a third option.
- The third option consist in post-processing each decomposed domain individually.
- To load all processor directories, you will need to manually create the file *`processorN.OpenFOAM`* (where **N** is the processor number) in each processor folder.
- After creating all *`processorN.OpenFOAM`* files, you can launch `paraFoam` and load each file (the *`processorN.OpenFOAM`* files).
- As you can see, this option requires more input from the user.

Running in parallel

Decomposing big meshes

- One final word, the utility `decomposePar` does not run in parallel.
- So, it is not possible to distribute the mesh among different computing nodes to do the partitioning in parallel.
- If you need to partition big meshes, you will need a computing node with enough memory to handle the mesh.
- We have been able to decompose meshes with up to 500 000 000 elements, but we used a computing node with 512 gigs of memory.
- For example, in a computing node with 16 gigs of memory, it is not possible to decompose a mesh with 30 000 000. You will need to use a computing node with at least 32 gigs of memory.
- Same applies for the utility `reconstructPar`.

Running in parallel

Do all utilities run in parallel?

- At this point, you might be wondering if all solvers/utilities run in parallel.
- To know what solvers/utilities do not run in parallel, in the terminal type:
  - `$> find $WM_PROJECT_DIR -type f | xargs grep –sl 'noParallel'`
- Paradoxically, the utilities used to decompose the domain and reconstruct the domain do not run in parallel.
- Another important utility that does not run in parallel is `blockMesh`. So to generate big meshes with `blockMesh` you need to use a big fat computing node.
- Another important utility that does not run in parallel by default is `paraFoam`.
- To compile `paraFoam` with MPI support, in the file *makeParaView* (located in the directory `$WM_THIRD_PARTY_DIR)`, set the option **`withMPI`** to true,
  - **`withMPI = true`**
- While you are working with the file *makeParaView*, you might consider enabling Python support,
  - **`withPYTHON = true`**

Running in parallel

Exercises

- Choose any tutorial or design your own case and do a scalability test. Scale your case with two different meshes (a coarse and a fine mesh).
- Run the same case using different partitioning methods. Which method scales better? Do you get the same results?
- Do you think that the best partitioning method is problem dependent?
- Compare the wall time of a test case using the maximum number of cores and the maximum number of virtual cores. Which scenario is faster and why?
- Run a parallel case without using the `–parallel` option. Does it run? Is it faster of slower? How many outputs do you see on the screen?
- Do you get any speed-up by using `renumberMesh`?
- What applications do not run in parallel?

Module 5

The postprocess utility – Sampling – Probing – On-the-fly postprocessing – Field manipulation – Data conversion

Roadmap

1. **On-the-fly postprocessing – functionObjects and the postProcess utility**
2. Sampling with the postProcess utility
3. Field manipulation
4. Data conversion

On-the-fly postprocessing – functionObjects

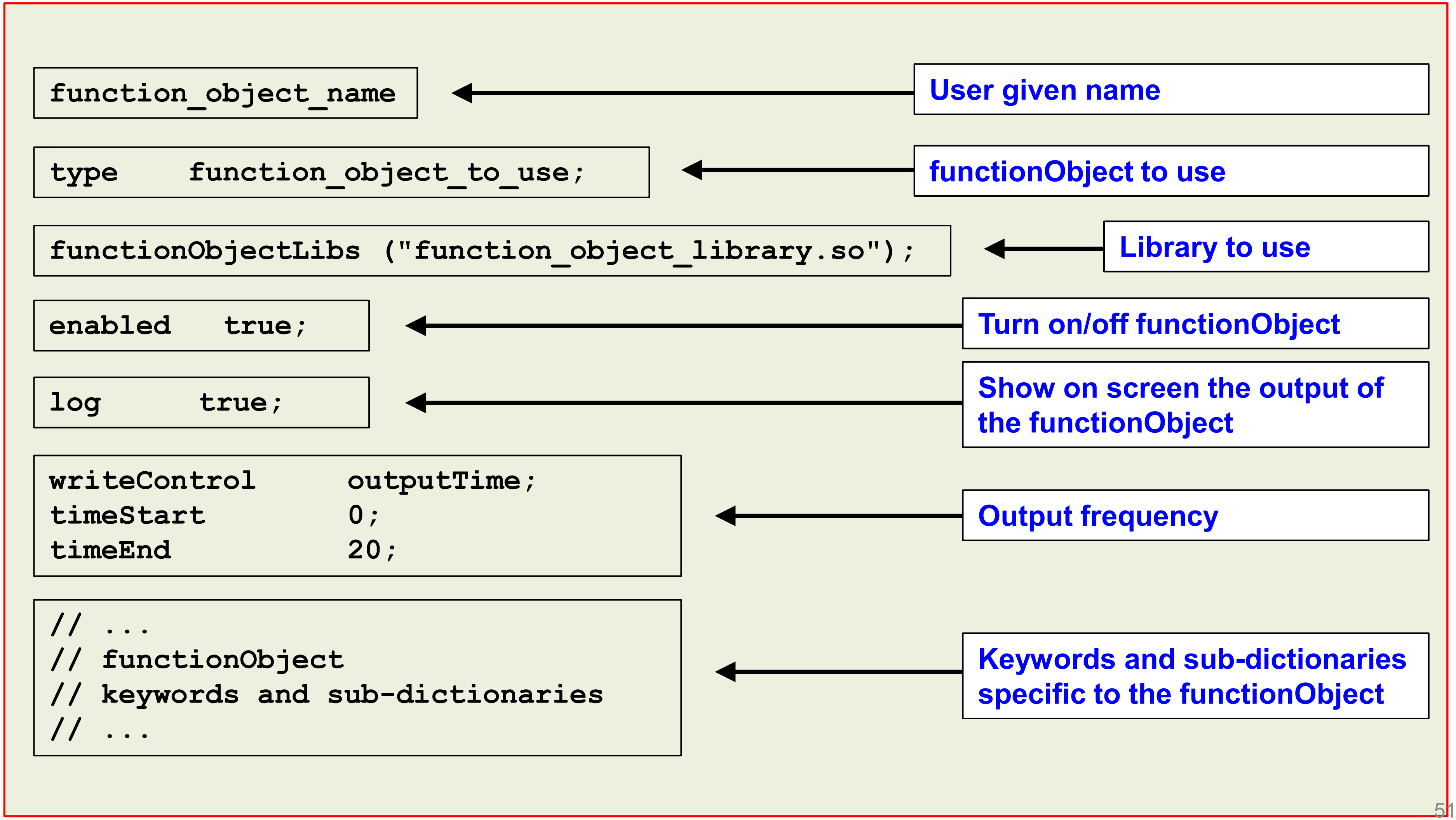
- It is possible to perform data extraction/manipulation operations while the simulation is running by using **functionObjects**.
- **functionObjects** are small pieces of code executed at a regular interval without explicitly being linked to the application.
- When using **functionObjects**, files of sampled data can be written for plotting and post processing.
- **functionObjects** are specified in the **controlDict** dictionary and executed at pre-defined intervals.
- All **functionObjects** are runtime modifiable.
- Depending of the **functionObject** you are using, its output is saved in the directory **`postProcessing`** or in the solution directory (time directories).
- It is also possible to execute **functionObject** after the simulation is over, we will call this running **functionObject** a-posteriori.
- For example, you can use **functionObjects** to compute the Mach number, the vorticity field, and to sample the velocity at given points or along a line, and everything while the simulation is running.

On-the-fly postprocessing – functionObjects

- In the directory `$FOAM_SRC/functionObjects` you will find the source code for the **functionObjects**.
- There are many **functionObjects**, and according to what they do, they are located in different sub-directories, namely, `field`, `forces`, `lagrangian`, `solvers`, and `utilities`. Just to name a few **functionObjects**:
    - **courantNo**
    - **div**
    - **fieldAverage**
    - **fieldMinMax**
    - **grad**
    - **MachNo**
    - **Q**
    - **vorticity**
    - **yPlus**
    - **forceCoeffs**
    - **forces**
    - **icoUncoupledKinematicCloud**
    - **scalarTransport**
    - **codedFunctionObject**
    - **residuals**
    - **systemCall**
    - **timeActivatedFileUpdate**
    - **writeObjects**
- In addition to the **functionObjects** located in the directory `$FOAM_SRC/functionObjects`, you can also run the sampling and co-processing utilities on-the-fly.
- You will find the source code for the sampling and co-processing utilities in the directory `$FOAM_SRC/sampling`.

On-the-fly postprocessing – functionObjects

- **functionObjects** are defined in the *controlDict* dictionary.
- To execute a **functionObject** you need to at least define the following entries:



```
function_object_name

type     function_object_to_use;

functionObjectLibs ("function_object_library.so");

enabled    true;

log      true;

writeControl      outputTime;
timeStart         0;
timeEnd           20;

// ...
// functionObject
// keywords and sub-dictionaries
// ...
```

User given name

functionObject to use

Library to use

Turn on/off functionObject

Show on screen the output of the functionObject

Output frequency

Keywords and sub-dictionaries specific to the functionObject

On-the-fly postprocessing – functionObjects

- There are many **functionObjects** implemented in OpenFOAM®, and sometimes is not very straightforward how to use a specific **functionObject**.
- Also, **functionObjects** can have many options and some limitations.
- Our best advice is to read the doxygen documentation or the source code to learn how to use **functionObjects**.
- Remember, the source code of the **functionObjects** is located in the directory:

```
$WM_PROJECT_DIR/src/postProcessing/functionObjects
```

- The source code of the sampling and co-processing utilities is located in the directory:

```
$WM_PROJECT_DIR/src/sampling
```

- The source code of the database entries required for the **functionObjects** is located in the directory:

```
$FOAM_SRC/OpenFOAM/db/functionObjects
```

- Here after we are going to study a few commonly used **functionObjects**.

On-the-fly postprocessing – functionObjects

- Let us do some on-the-fly postprocessing.
- For this we will use the multi-element airfoil 2D case.
- You will find this case in the directory:

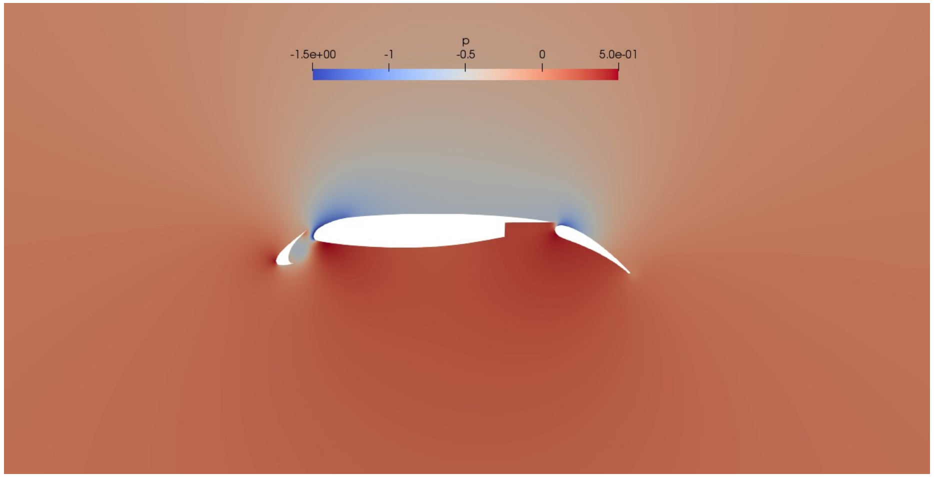
```
$PTOFC/101postprocessing/MDA_30P30N
```

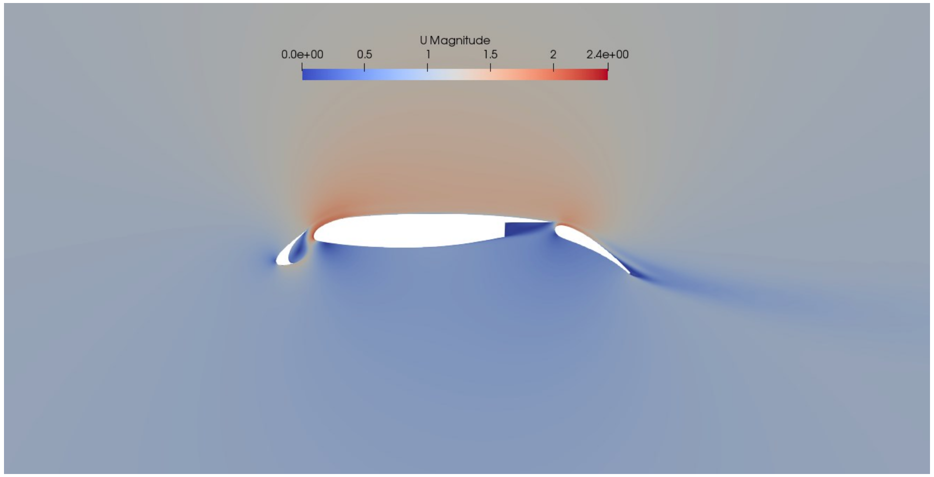
- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

On-the-fly postprocessing – functionObjects

At the end of the day, you should get something like this

Qualitative post-processing





Quantitative post-processing

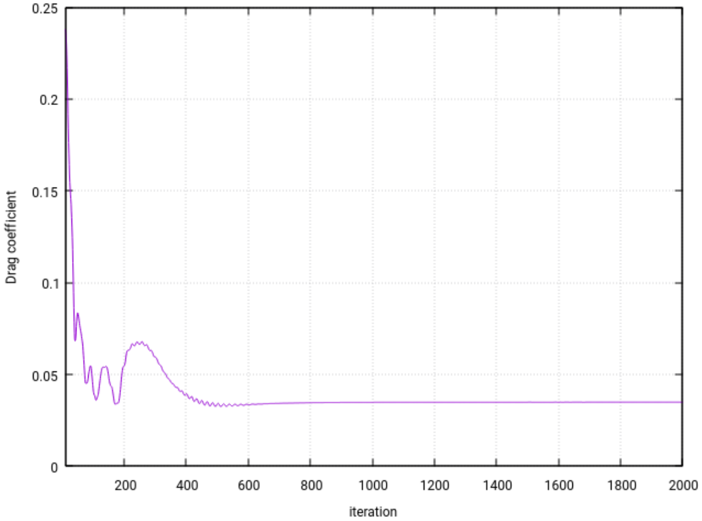
	c_d	c_l
Experimental values	0.0332	2.167
Numerical values	0.0346	2.238

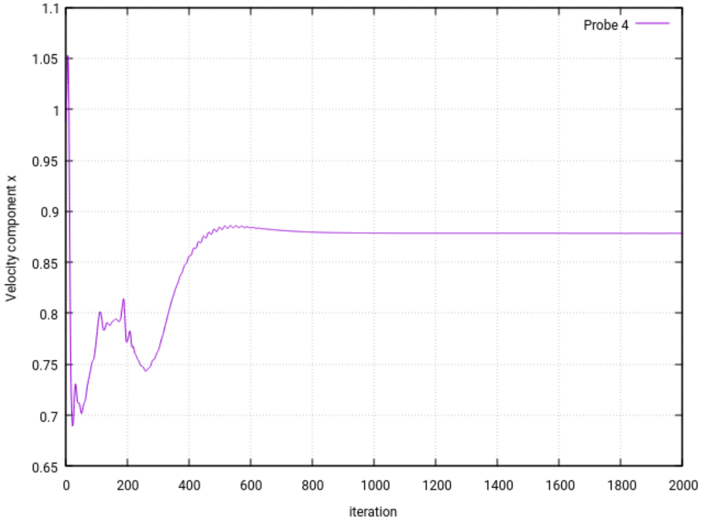
Additionally, by using **functionObjects** we will compute many derived quantities, such as,

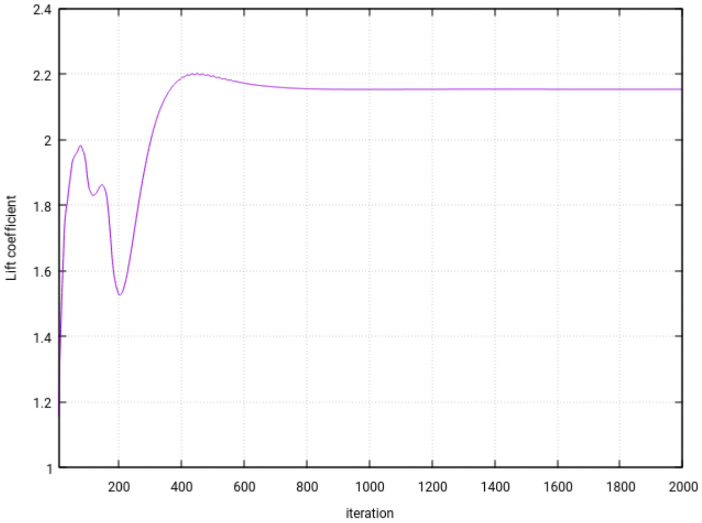
- yPlus.
- Voriticity.
- Mean values of the field variables (notice that we will compute the average of a steady solution).
- Forces.
- Force coefficients.
- Minimum and maximum values of the field variables.
- Sampling at given points.
- Mass flow at inlets and outlets.

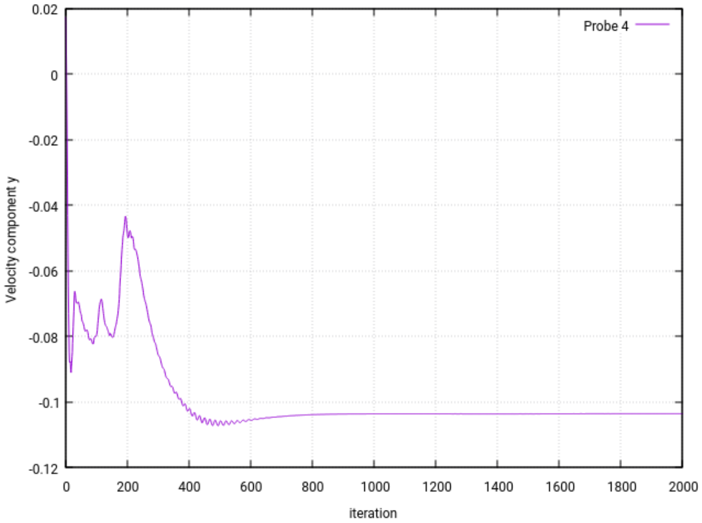
On-the-fly postprocessing – functionObjects

At the end of the day, you should get something like this





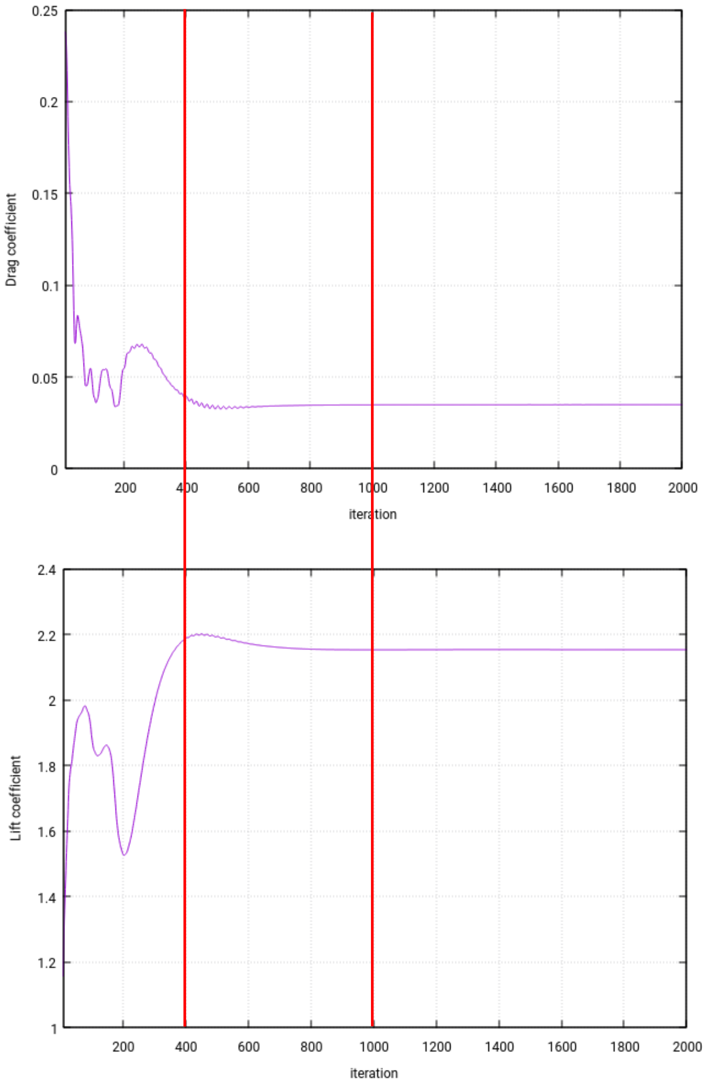


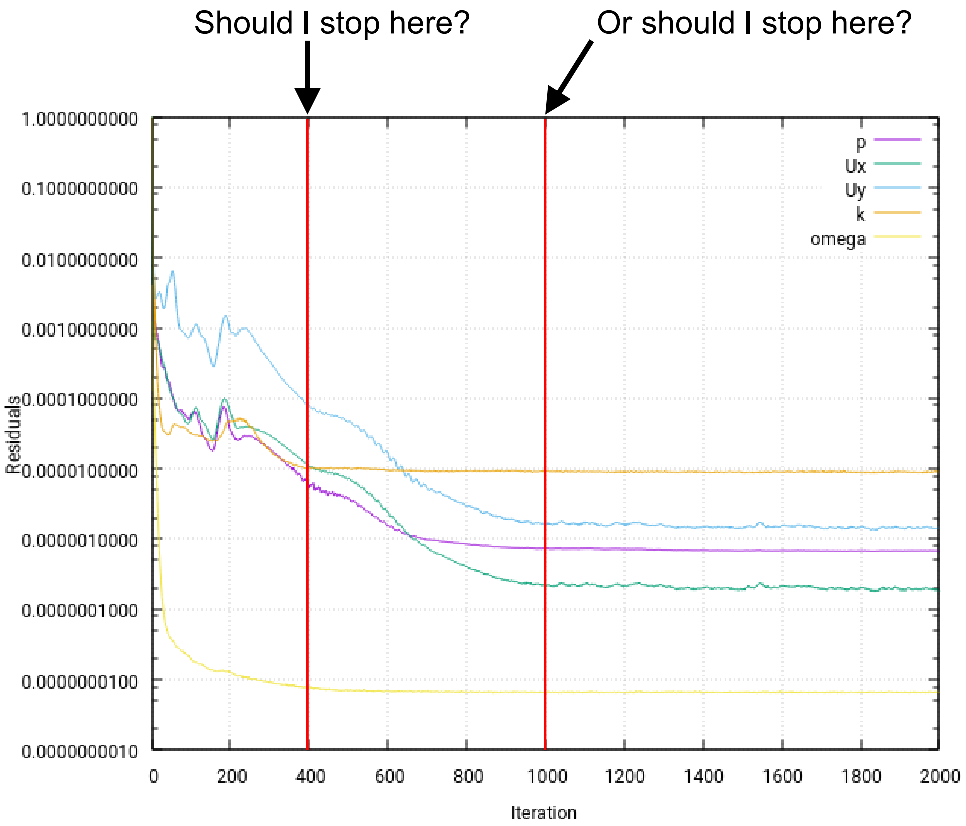


Quantitative post-processing

On-the-fly postprocessing – functionObjects

At the end of the day, you should get something like this




Should I stop here?

Or should I stop here?

Quantitative post-processing – Assessing residuals

On-the-fly postprocessing – functionObjects

Running the case

- Let us run this case using the automatic scripts distributed with the tutorial. In the terminal type:

```
$> sh run_all.sh
```

- After the simulation is finish, you will find the decomposed directories (**`processor0`**, **`processor1`**, **`processor2`** and **`processor3`**), the **`postProcessing`** directory, and the **`2000`** directory. The solution, and output of the **functionObjects**, is saved in these directories.

- Remember, to visualize the decomposed solution you will need to launch `paraFoam` as follows,

```
$> paraFoam -builtin
```

- Do not erase the solution as we are going to use it in the next section.

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
51      functions
52      {

            name_of_the_functionObject_dictionary
            {
                Sub-dictionary with functionObject entries
            }

206     #include "externalFunctionObject"

211     }
```

- Let us take a look at the bottom of the *controlDict* dictionary file. In this dictionary is where we define all **functionObjects.**
- Within this dictionary, **functionObjects** are defined in the sub-dictionary **functions**, *i.e.*,

```
functions
{
        functionObjects definition
};
```

- In this case, the **functionObjects** are defined in lines 51-211 (the sub-dictionary **functions**).
- Each defined **functionObject** has its own name and its compulsory keywords and entries.
- Notice that in line 206 we use the directive include to call an external dictionary with the **functionObjects** definition.
- If you do not give the path of the external dictionary, the solver will look for it in the directory **`system`**.
- If you use the include directive, you will need to update the *controlDict* dictionary in order to read any modification done in the included dictionary files.

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
51      functions
52      {

56       forces_object
57       {
58               type forces;


59               functionObjectLibs ("libforces.so");
60
61
62               writeControl   timeStep;
63               writetInterval  1;
64
65               enabled true;
66
67               //// Patches to sample
68               patches ("wall_slat" "wall_airfoil" "wall_flap");


70               //// Name of fields
71               pName p;
72               Uname U;



74               //only for incompressible flows
75               rho rhoInf;
76               rhoInf 1.0;



78               //// Centre of rotation
79               CofR (0 0 0);
80       }

211      }
```

functionObject identifier (user given)

- Let us explain in detail how to setup a **functionObject**.
- As the names implies, this **functionObject** is used to compute the forces on a given body or set of bodies (line 56).
- You can add as many forces **functionObjects** (or any other one) as you like, but you should assign them different identifiers (line 56). Remember not to use white spaces when naming **functionObjects**.
- The output of this **functionObject** is saved in the directory **`postProcessing/forces_object`**, where the directory name is taken from line 56.
- Inside this directory, you will find the subdirectory `0`, which means that you started to sample data from time `0`. If you start from a different time, you will find a different subdirectory, *e.g.*, **`86.05`**
- Remember, different **functionObjects** will have different entries, to know the entries just refer to the online documentation or skim the source code, which is located in the directory,
  - **`$WM_PROJECT_DIR/src/postProcessing/functionObjects`**

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
functions
{

  forces_object
  {
        type forces;


        functionObjectLibs ("libforces.so");


        writeControl   timeStep;
        writetInterval  1;

        enabled true;

        //// Patches to sample
        patches ("wall_slat" "wall_airfoil" "wall_flap");


        //// Name of fields
        pName p;
        Uname U;


        //only for incompressible flows
        rho rhoInf;
        rhoInf 1.0;


        //// Centre of rotation
        CofR (0 0 0);
  }

  }
```

- Let us study all entries of the forces **functionObject**

functionObject identifier (user given)

functionObject to use

functionObject library to use

Controls for saving frequency

Turn on/off **functionObject**

Compute the forces on these patches

Name of the velocity and pressure fields. If you use different fields, *e.g.*, **pMean** and **Umean**, they need to be computed before this **functionObject**

Reference density value. It only needs to be defined for incompressible flows. For compressible flows, the computed density is used instead (you will need to define a dummy value, though)

Reference center of rotation to compute moments

Note:

- The source code of this functionObject is located in the directory **`$FOAM_SRC/functionObjects/forces/forces`**
- Use the banana method to know all the options available for each entry.

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
functions
{

 forceCoeffs_object
 {
         type forceCoeffs;
         functionObjectLibs ("libforces.so");

         enabled true;

         patches ("wall_slat" "wall_airfoil" "wall_flap");

         pName p;
         Uname U;

         rho rhoInf;
         rhoInf 1.0;

         log true;

         CofR (0.0 0 0);

         pitchAxis (0 0 1);
         magUInf 1.0;
         lRef 1;
         Aref 1;

         writeControl   timeStep;
         writeInterval  1;

         liftDir     (0 1 0);
         dragDir     (1 0 0);

 }

 }
```

functionObject identifier (user given)

- Let us study now the **functionObject** used to compute the force coefficients.

- This **functionObject** computes the force coefficients.
- These entries are similar to those of the force **functionObject**

This option will output the values to a text file located in the directory **`postProcessing/forceCoeffs_object`**

Reference center of rotation to compute moments

Reference values used to compute coefficients

Controls for saving frequency

Reference axes to compute the lift and drag coefficients.

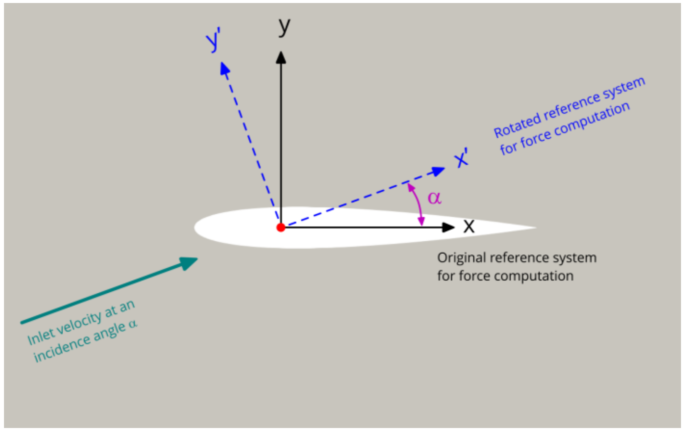
On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
functions
{

 forceCoeffs_object
 {
         type forceCoeffs;
         functionObjectLibs ("libforces.so");

         enabled true;

         patches ("wall_slat" "wall_airfoil" "wall_flap");

         pName p;
         Uname U;

         rho rhoInf;
         rhoInf 1.0;

         log true;

         CofR (0.0 0 0);

         pitchAxis (0 0 1);
         magUInf 1.0;
         lRef 1;
         Aref 1;

         writeControl   timeStep;
         writeInterval  1;

         liftDir     (0 1 0);
         dragDir     (1 0 0);

 }

 }
```

functionObject identifier (user given)

- Let us study now the **functionObject** used to compute the force coefficients.



$$\text{liftDir}\ (-\sin(\alpha), \cos(\alpha), 0)$$

$$\text{dragDir}\ (\cos(\alpha), \sin(\alpha), 0)$$

- Reference axes to compute the lift and drag coefficients.
- Remember, lift and drag are perpendicular and parallel to the incoming flow, respectively.
- So if the inlet velocity is entering at a given angle, you should adjust the vectors **liftDir** and **dragDir** so they are aligned with the incoming flow (rotation matrix).

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
51     functions
52     {

131     minmaxdomain
132     {
133             type fieldMinMax;
134
135             functionObjectLibs ("libfieldFunctionObjects.so");
136
137             enabled true;
138
139             mode component;
140
141             writeControl timeStep;
142             writeInterval 1;
143
144             log true;
145
146             fields (p U nuTilda nut k omega);
147     }
148
149
150
151
152
153     yplus
154     {
155             type yPlus;
156             functionObjectLibs ("libfieldFunctionObjects.so");
157             enabled true;
158             log     true;
159             writeControl outputTime;
160     }

211     }
```

- **fieldMinMax functionObject**
  - This **functionObject** is used to compute the minimum and maximum values of the field variables.
  - The output of this **functionObject** is saved in ascii format in the file *fieldMinMax.dat* located in the directory

    `postProcessing/minmaxdomain/0`
  - Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 131).

- **yPlus functionObject**
  - This **functionObject** is used to compute the yPlus field.
  - This **functionObject** has two outputs, one output saved in the solution directories (**`1`**, **`2`**, **`3`**, and so on). You can visualize this output using paraview/paraFoam.
  - The second output is located in the directory

    `postProcessing/yplus/0`.
  - In this file you will find the minimum, maximum and average values of yPlus in all patches defined as walls.
  - Remember, the name of the directory where the output data (descriptive statistics) is saved is the same as the name of the **functionObject** (line 153).

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
51     functions
52     {

166     fieldAverage1
167     {
168     type              fieldAverage;
169     libs ( "libfieldFunctionObjects.so" );
170     writeControl     writeTime;

177     fields
178     (
179             U
180             {
181                     mean        on;
182                     prime2Mean  on;
183                     base        time;
184             }
185
186             p
187             {
188                     mean        on;
189                     prime2Mean  on;
190                     base        time;
191             }
192
193             nut
194             {
195                     mean        on;
196                     prime2Mean  on;
197                     base        time;
198             }
199     );
200     }

206    #include "externalFunctionObject"

211     }
```

User given file name

- **fieldAverage functionObject**
  - This **functionObject** is used to compute the average values of the field variables.
  - The output of this **functionObject** is saved in the time solution directories.
  - In this case, we are computing the field averages of velocity (**U**), pressure (**p**), and turbulent viscosity (**nut**).
  - In this **functionObject**, prime2Mean is the average of the product of the fluctuations of the variable,

$$\overline{\phi'\phi'}$$

- In line 206 we add a **functionObject** definition using an external file.
- In this case, the **functionObject** is located in the directory **`system`**
  - If you want to run this **functionObject** online, do not add lines 51, 52, and 211 in the file *externalFunctionObject*.
  - To run this functionObject a-posteriori (after the simulation is over by using the saved solution); add lines 51, 52, and 211 to the file *externalFunctionObject*. We explain how to run **functionObjects** a posteriori later.

On-the-fly postprocessing – functionObjects

The `externalFunctionObject` dictionary

```
24      probes_online
25      {
26              type            probes;
27              functionObjectLibs ("libfieldFunctionObjects.so");
28              enabled         true;
29              writeControl timeStep;
30              writeInterval 1;
31
32              probeLocations
33              (
34                      (1   0    0)
35                      (2   0    0)
36                      (2   0.25        0)
37                      (2  -0.25        0)
38              );
39
40              fields
41              (
42                      U
43                      p
44              );
45
46      }

52      vorticity
53      {
54              type vorticity;
55              functionObjectLibs ("libfieldFunctionObjects.so");
56              enabled         true;
57              log             true;
58              writeControl    outputTime;
59      }
```

- **probes functionObject**
  - This **functionObject** is used to probe field data at the given locations.
  - In this case, we are sampling the fields **U** and **p** (lines 35-39)
  - The output of this **functionObject** is saved in ascii format in the files *p* and *U* located in the directory

    `postProcessing/probes_online/0`

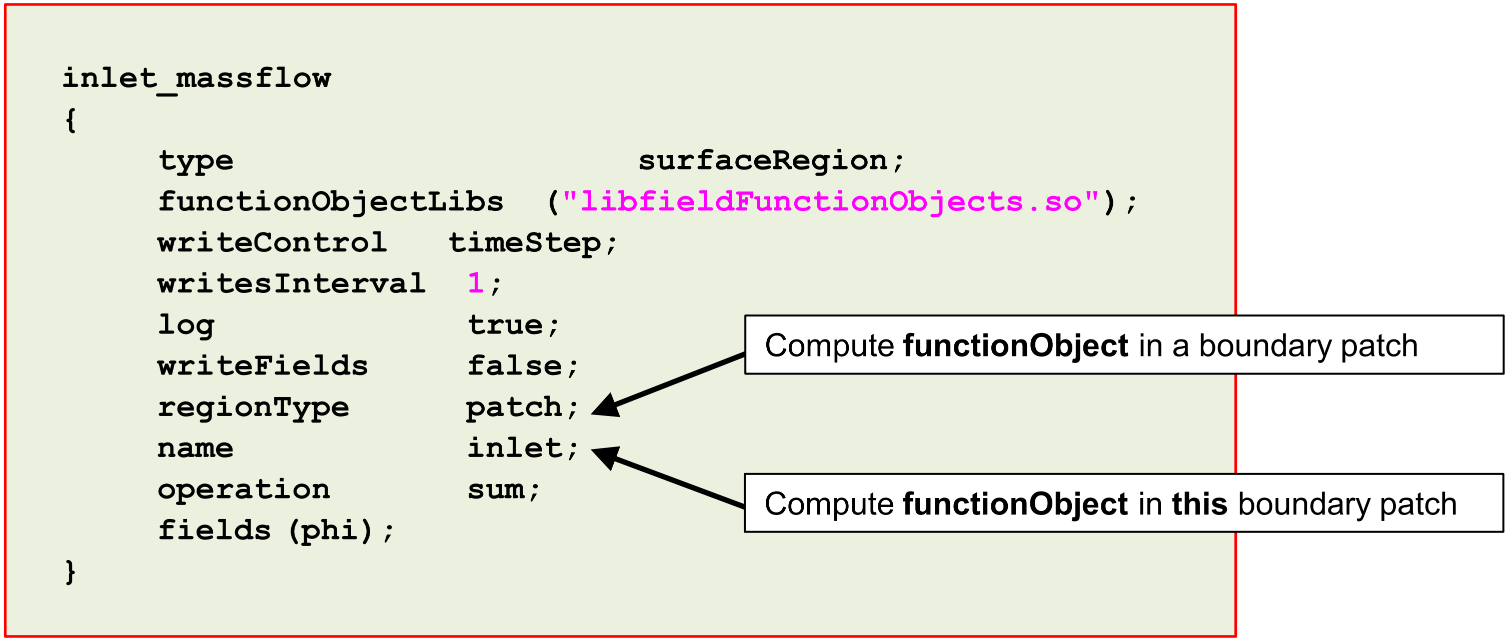
  - Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 19).

- **vorticity functionObject**
  - This **functionObject** is used to compute the vorticity field.
  - The output of this **functionObject** is saved in the solution directories (`1`, `2`, `3`, and so on). You can visualize this output using paraview/paraFoam.

On-the-fly postprocessing – functionObjects

Final remarks on functionObjects

- A **functionObject** that is very useful, but we did not use in this case:



```
inlet_massflow
{
     type                          surfaceRegion;
     functionObjectLibs  ("libfieldFunctionObjects.so");
     writeControl   timeStep;
     writesInterval  1;
     log             true;
     writeFields     false;
     regionType      patch;
     name            inlet;
     operation       sum;
     fields (phi);
}
```

Compute **functionObject** in a boundary patch

Compute **functionObject** in **this** boundary patch

- This **functionObject** is used to computed the mass flow across a boundary patch.
- Remember, the method is conservative so what is going in, is going out (unless you have source terms).
- So if you want to measure the mass imbalance, setup this function object for each boundary patch where you have flow entering or going out of the domain.

On-the-fly postprocessing – functionObjects

Final remarks on functionObjects

- As you can see, there are many functionObjects implemented in OpenFOAM®.
- We just explained the most common **functionObjects**.
- You can use the banana method to know all the options available for each entry, search in the documentation, or read the source code located in the directory **`$FOAM_SRC/functionObjects`**
- In the supplement slides you will find more examples of more complex **functionObjects**.
- You will also find a deck of slides with a detailed explanation of advanced paraview features and some basic instructions for data plotting and analysis using gnuplot.
- Remember, you can also do the same postprocessing using paraview/paraFoam, but you will only work on the saved fields.
- A great advice before running your simulation, setup all your **functionObjects** and gather as much as possible quantitative data.

On-the-fly postprocessing – functionObjects

Running functionObjects a-posteriori

- Sometimes it can happen that you forget to use a **functionObject** or you want to execute a **functionObject** a-posteriori (when the simulation is over).
- The solution to this problem is to use the solver with the option `-postProcess`. This will only compute the new **functionObject**, it will not rerun the simulation.
- For instance, let us say that you forgot to use a given **functionObject**. Open the dictionary *`controlDict`*, add the new **functionObject**, and type in the terminal,
  - `$> name_of_the_solver -postProcess -dict dictionary_location`
- You also have the option of adding the new **functionObject** in an external file. If you chose this option, do not forget to add the **functionOption** within the **function** sub-dictionary block:

```
function
{
        //functionObject definitions here
};
```

- By proceeding in this way you do not need to rerun the simulation, you just compute the new **functionObject**.

On-the-fly postprocessing – functionObjects

Running functionObjects a-posteriori

- In the directory **`system`**, you will find the following **functionObject** external dictionaries: *`externalFunctionObject`*
- To run this **functionObject** a-posteriori, type in the terminal:

```
1. $> simpleFoam -postProcess -dict system/externalFunctionObject -noZero

2. $> simpleFoam -postProcess -dict system/externalFunctionObject -time 500:2000

3. $> simpleFoam -postProcess -dict system/functionObject3 -latestTime
```

- In step 1, we are reading the dictionary `system/externalFunctionObject` and we are doing the computation for all the saved solutions, except time zero.
- In step 2, we are reading the dictionary `system/externalFunctionObject` and we are doing the computation for the time range 500 to 2000
- In step 3, we are reading the dictionary *`functionObject3`* and we are doing the computation only for he latest saved solution.
- If you do not give any time manipulator, the computation will be carried out on every saved solution.

On-the-fly postprocessing – functionObjects

Exercises

- Where is located the source code of the **functionObjects**?
- Try to run in parallel? Do all **functionObjects** work properly?
- Compute the Courant number using **functionObjects**.
- Compute the total pressure and velocity gradient using **functionObjects** (on-the-fly and a-posteriori).
- Sample data (points, lines and surfaces) using **functionObjects** (a-posteriori).
- Is it possible to do system calls using **functionObjects**? If so what **functionObject** will you use and how do you use it? Setup a sample case.
- Is it possible to update dictionaries using **functionObjects**? If so what **functionObjects** will you use and how do you use it? Setup a sample case.
- What are the compulsory entries of the **functionObjects**?

Roadmap

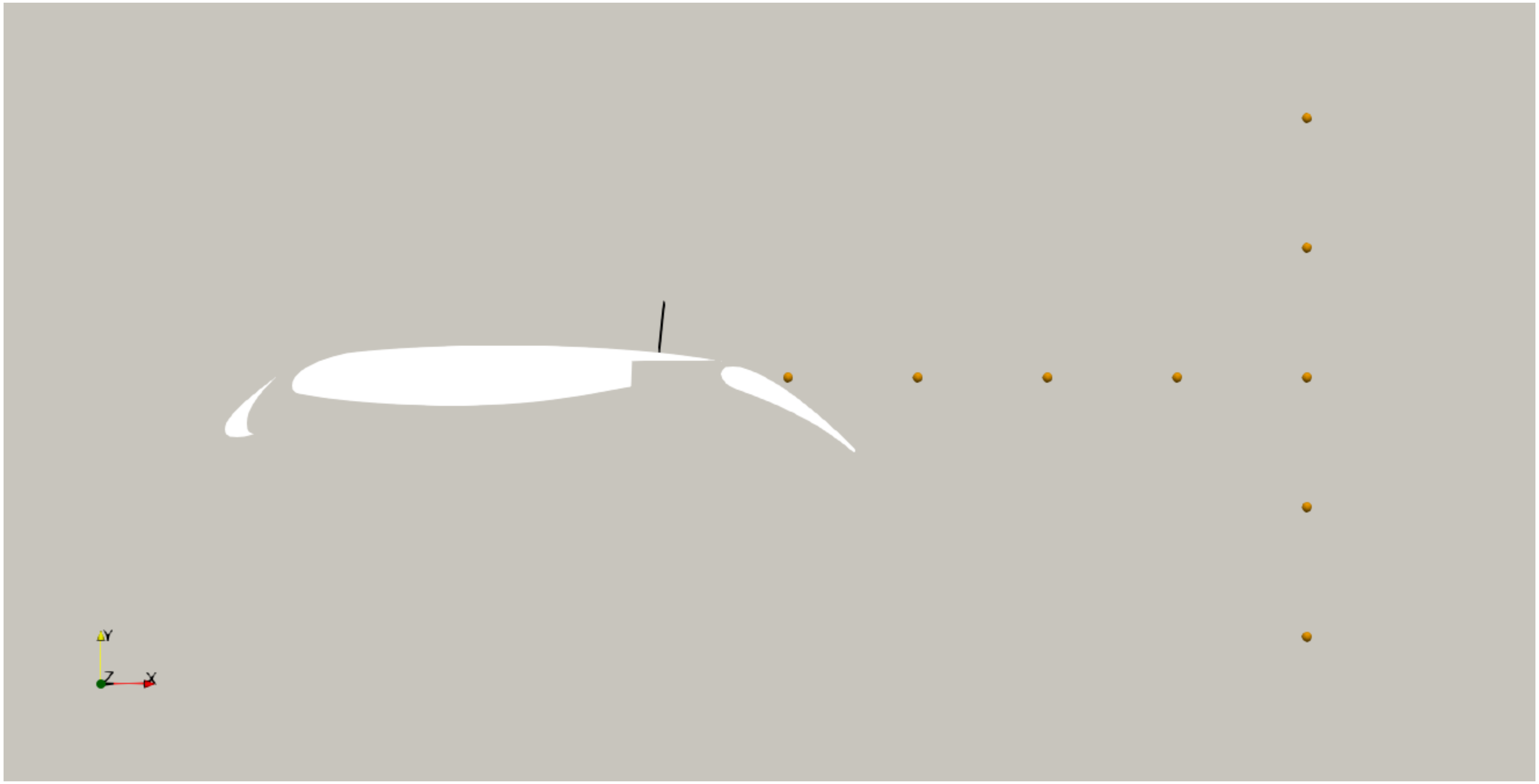
1. ~~On-the-fly postprocessing – functionObjects and the postProcess utility~~
2. **Sampling with the postProcess utility**
3. Field manipulation
4. Data conversion

Sampling with the postProcess utility

- OpenFOAM® provides the `postProcess` utility to sample field data for plotting.
- The sampling parameters are specified in a dictionary located in the case **`system`** directory.
- You can give any name to the input dictionary, hereafter we are going to name them *`sampleDict`* (to sample along a line) and *`probesDict`* (to sample in a set of probes).
- During the sampling, and inside the case directory, a new directory named **`postProcessing`** will be created. In this directory, the sampled values are stored in a sub-directory with the name of the input dictionary, in this case, **`sampleDict`** and **`probesDict`**.
- This utility can sample points, lines, and surfaces.
- Data can be written in a range of formats including well-known plotting packages such as: grace/xmgr, gnuplot and jPlot.
- The sampling can be executed by running the utility `postProcess` in the case directory and according to the application syntax.
- A final word, this utility does not do the sampling while the solver is running. It does the sampling after you finish the simulation.

Sampling with the postProcess utility

- To do sampling, we will use the solution from the previous case.
- If you do not have the solution, follow the instructions given in the previous slides.
- Hereafter, we will sample along a line and in a few probe locations, as illustrated in the figure below.



Sampling with the postProcess utility

Running the case

- Let us do the sampling,

```
1.  $> postProcess -func sampleDict -time 2000
2.  $> postProcess -func probesDict -time 2000
```

- In step 1, we do some sampling using the dictionary *`sampleDict`*. We also do the sampling only for time 2000
- In step 2, we do some sampling using the dictionary *`probesDict`*. We also do the sampling only for time 2000.
- Remember, you can use different time manipulators.
- If you do not give any time manipulator option, the sampling will be computed for all saved solutions (including time directory 0).

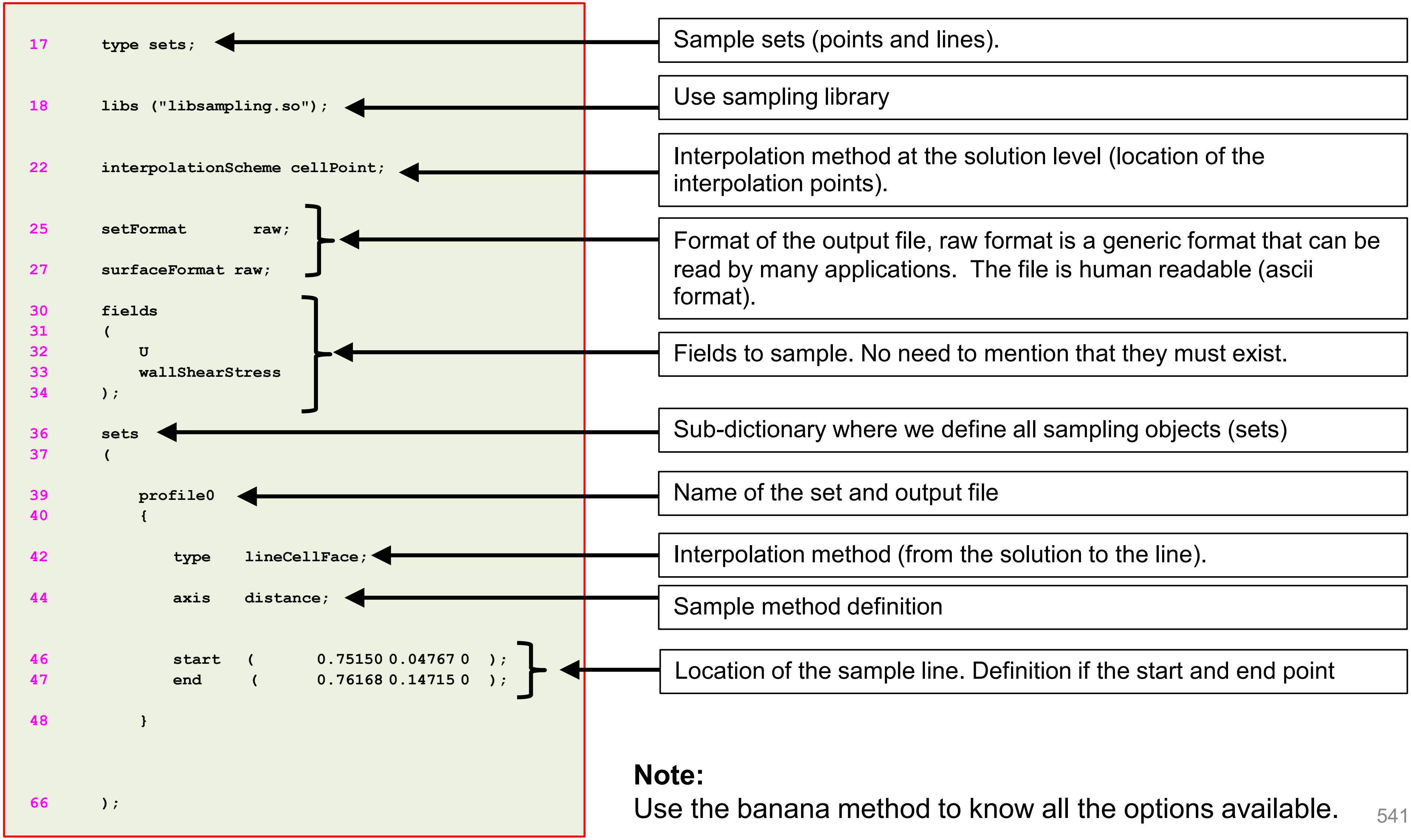
Sampling with the postProcess utility

The *sampleDict* and *probesDict* dictionaries

- These dictionaries are located in the directory **system**.
- In this case, the *sampleDict* dictionary is used to sample along a line. This file contains several entries to be set according to the user needs. The following entries can be set,
    - The choice of the interpolationScheme.
    - The format of the line data output.
    - The format of the surface data output.
    - The fields to be sample.
    - The sub-dictionaries that controls each sampling operation.
    - In these sub-dictionaries you can set the name, type and geometrical information of the sampling operation.
- In this case, the *probesDict* is used to sample in a set of points. This file contains several entries to be set according to the user needs. The following entries,
    - The fields to be sample.
    - Location of the probes.
- The following **functionObjects** type can be used to do sampling: **patchProbes**, **probes**, **sets**, or **surfaces**.

Sampling with the postProcess utility

The *sampleDict* dictionary



```
type sets;

libs ("libsampling.so");

interpolationScheme cellPoint;

setFormat       raw;

surfaceFormat raw;

fields
(
    U
    wallShearStress
);

sets
(

    profile0
    {

        type    lineCellFace;

        axis    distance;

        start   (       0.75150 0.04767 0   );
        end     (       0.76168 0.14715 0   );

    }

);
```

Sample sets (points and lines).

Use sampling library

Interpolation method at the solution level (location of the interpolation points).

Format of the output file, raw format is a generic format that can be read by many applications. The file is human readable (ascii format).

Fields to sample. No need to mention that they must exist.

Sub-dictionary where we define all sampling objects (sets)

Name of the set and output file

Interpolation method (from the solution to the line).

Sample method definition

Location of the sample line. Definition if the start and end point

Note:
Use the banana method to know all the options available.

Sampling with the postProcess utility

The *sampleDict* dictionary

```
type sets;

libs ("libsampling.so");

interpolationScheme cellPoint;

setFormat       raw;

surfaceFormat raw;

fields
(
    U
    wallShearStress
);

sets
(

    profile0
    {

        type    lineCellFace;

        axis    distance;


        start   (       0.75150 0.04767 0   );
        end      (       0.76168 0.14715 0   );

    }



);
```

Name of sampled set

- Remember, the sampled data is always saved in the directory **`postProcessing`**
- Then, in the sub-directory **`sampleDict`** (whose name corresponds to the name of the input file), you will find the data sampled in a directory corresponding to the sampled time.
- For example, in this case you fill find the data in the directory **`postProcessing/sampleDict/2000`**
- Then, in the file *`profile0_U_wallShearStress.xy`* you will find the data.
- The name of the output file corresponds to the name of the sampled set, appended by the name of the sampled fields.
- Different files will be created for tensor, vector and scalar fields.
- Feel free to open the output files using your favorite text editor.

Sampling with the postProcess utility

The *probesDict* dictionary

```
type probes;



(
    p
    U
);



probeLocations
(
    (1.0    0 0)
    (1.25   0 0)
    (1.5   0 0)
    (1.75   0 0)
    (2.0    0 0)
    (2.0 -.25 0)
    (2.0 -.5 0)
    (2.0  .25 0)
    (2.0  .5 0)
);
```

Sample points.

Fields to sample. No need to mention that they must exist.

Location of the points.

Note:
Use the banana method to know all the options available.

Sampling with the postProcess utility

The *probesDict* dictionary

```
type probes;


(
    p
    U
);


probeLocations
(
    (1.0    0 0)
    (1.25   0 0)
    (1.5   0 0)
    (1.75   0 0)
    (2.0    0 0)
    (2.0 -.25 0)
    (2.0 -.5 0)
    (2.0   .25 0)
    (2.0   .5 0)
);
```

- Remember, the sampled data is always saved in the directory **`postProcessing`**
- Then, in the sub-directory **`probesDict`** (whose name corresponds to the name of the input file), you will find the data sampled in a directory corresponding to the sampled time.
- For example, in this case you fill find the data in the directory **`postProcessing/probesDict/2000`**
- Then, inside this directory, you will find several files containing the sampled data.
- The name of the output file corresponds to the name of the sampled fields, in this case, *U* and *p*.
- Feel free to open the output files using your favorite text editor.

Sampling with the postProcess utility

The output files – **functionObject** type **sets** or **surfaces**

- The output format of the point sampling (cloud) is as follows:

Scalars

```
#POINT_COORDINATES (X Y Z)                SCALAR_VALUE
0       0       0.05                      13.310995
0       0       0.1                       19.293817
…
```

Vectors

```
#POINT_COORDINATES (X Y Z)                VECTOR_COMPONENTS (X Y Z)
0       0       0.05                      0       0       2.807395
0       0       0.1                       0       0       2.826176
…
```

Sampling with the postProcess utility

The output files – **functionObject** type **sets** or **surfaces**

- The output format of the line sampling is as follows:

Scalars

```
#AXIS_COORDINATE                 SCALAR_VALUE
0                                18.594038
0.0015                           18.249091
…
```

Vectors

```
#AXIS_COORDINATE                 VECTOR_COMPONENTS (X Y Z)
0                                0      0      1.6152966
0.0015                           0      0      1.8067536
…
```

Sampling with the postProcess utility

The output files – **functionObject** type **sets** or **surfaces**

- The output format of the surface sampling is as follows:

Scalars

```
#POINT_COORDINATES (X Y Z)            SCALAR_VALUE
0       0       0.05                  13.310995
0       0       0.1                   19.293817
…
```

Vectors

```
#POINT_COORDINATES (X Y Z)            VECTOR_COMPONENTS (X Y Z)
0       0       0.05                  0       0       2.807395
0       0       0.1                   0       0       2.826176
…
```

Sampling with the postProcess utility

The output files – **functionObject** type **probes**

- The output format of the probing is as follows:

Scalars

```
# Probe 0 (0 0 0.025)
# Probe 1 (0 0 0.05)
# Probe 2 (0 0 0.075)
# Probe 3 (0 0 0.1)
#       Probe           0               1               2               3
#       Time
        0               0               0               0               0
        0.005           19.1928         16.9497         14.2011         11.7580
        0.01            16.6152         14.5294         12.1733         10.0789
        …
        …
        …
```

Sampling with the postProcess utility

The output files – **functionObject** type **probes**

- The output format of the probing is as follows:

Vectors

```
# Probe 0 (0 0 0.025)
# Probe 1 (0 0 0.05)
# Probe 2 (0 0 0.075)
# Probe 3 (0 0 0.1)
#       Probe           0                1                2                3
#       Time
        0               (0 0 0)          (0 0 0)          (0 0 0)          (0 0 0)
        0.005           (0 0 2.1927)     (0 0 2.1927)     (0 0 2.1927)     (0 0 2.1927)
        0.01            (0 0 2.5334)     (0 0 2.5334)     (0 0 2.5334)     (0 0 2.5334)
        …
        …
        …
```

Sampling with the postProcess utility

Exercises

- Where is located the source code of the utility `postProcess`?
- Try to do the sampling in parallel? Does it run? What about the output file?
- How many options are there available to do sampling in a line?
- Do point, line, and surface sampling using paraFoam/ParaView and compare with the output of the `postProcess` utility. Do you get the same results?
- Compute the descriptive statistics of each column of the output files using gnuplot. Be careful with the parentheses of the vector files.

  (Hint: you can use sed within gnuplot)

Roadmap

1. ~~On-the-fly postprocessing – functionObjects and the postProcess utility~~
2. ~~Sampling with the postProcess utility~~
3. **Field manipulation**
4. Data conversion

Field manipulation

- Hereafter we are going to deal with field manipulation
- Field manipulation means modifying a field variable or deriving a new field variable using the primitive variables computed during the solution stage.
- We will do the post-processing using the command line interface (CLI), or non-GUI mode.
- The utility `postProcess` can be used as a single application, *e.g.*,
  - `$> postProcess –func vorticity`
- Or it can be used with a solver using the option `–postprocess`, *e.g.*,
  - `$> simpleFoam -postprocess –func vorticity`
- Running the solver with the option `–postprocess` will only execute the post-processing and it will let you access data available on the database for the particular solver (such as physical properties or turbulence model).

Field manipulation

- To get a list of what can be computed using the `postProcess` utility, type in the terminal:
  - `$> postProcess -list`

- The utility `postProcess` can take many options. To get more information on how to use the utility, type in the terminal:
  - `$> postProcess -help`
  - `$> simpleFoam -postProcess -help`

- The options of the solver using the `-postProcess` flag are the same as the options of the utility `postProcess`.
- In the sub-directory **`$FOAM_UTILITIES/postProcessing/postProcess`** you will find the utility `postProcess`.
- In the directory **`$FOAM_SRC/functionObjects`**, you will find the source code of the objects that can be used to compute a new field.

Field manipulation

- We will now do some field manipulation using the cylinder case.
- For this we will use the supersonic wedge tutorial located in the directory:

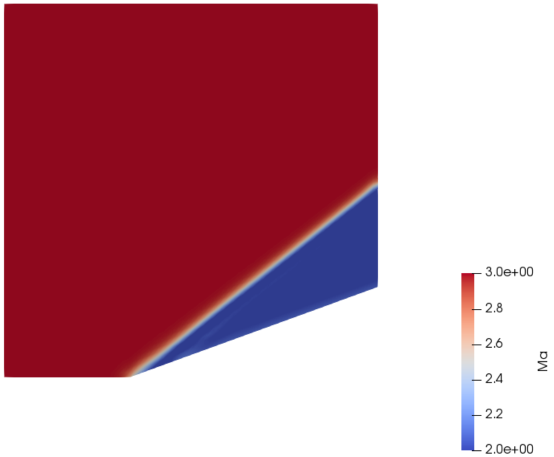
```
$PTOFC/101postprocessing/supersonic_wedge/
```

- In the case directory, you will find the README.FIRST file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, run_all.sh, run_mesh.sh, run_sampling.sh, run_solver.sh, and so on. These files can be used to run the case automatically by typing in the terminal, for example, sh run_solver.
- We highly recommend you to open the README.FIRST file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

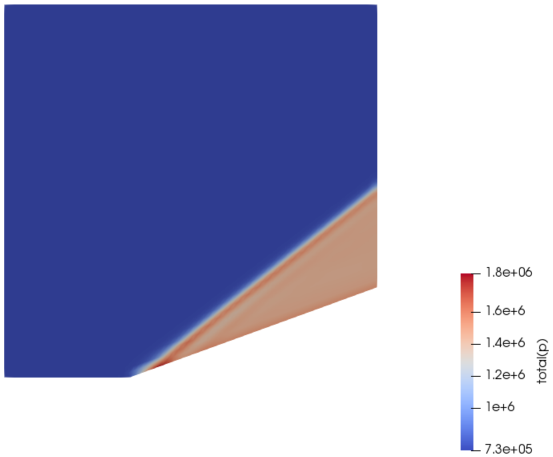
Field manipulation

After computing the solution, we can compute derived fields (*e.g.*, Mach number, density, Courant number, vorticity, and so on), using the primitive fields (U, p, T)

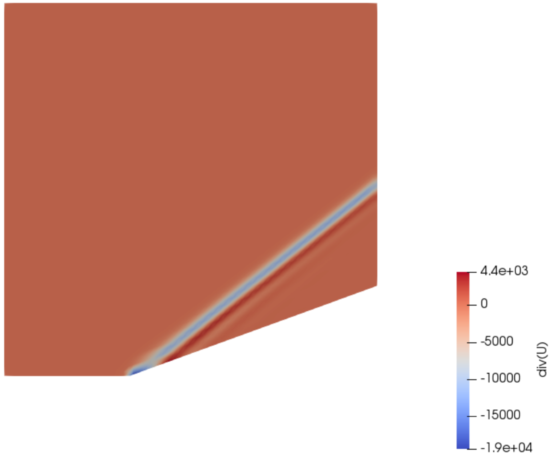
Mach number



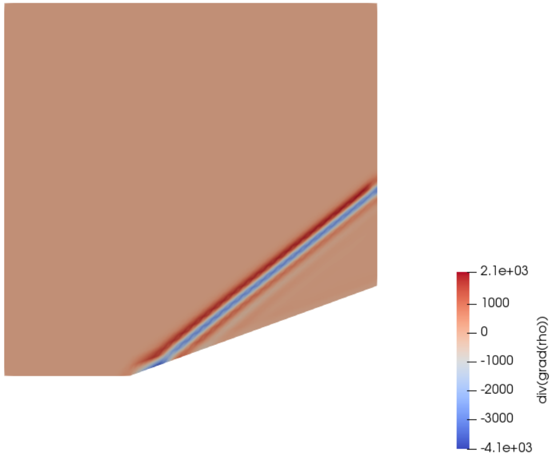
Total pressure



Divergence of U



Divergence of density gradient (numerical shadowgraph)



Field manipulation

What are we going to do?

- We will use this case to introduce the `postProcess` utility for field manipulation.
- We will also show how to run the solver with the option `-postProcess`. This will let us do only the post-processing after the solution has been computed, and it will let us access the database of the solver.
- To find the numerical solution we will use the solver `rhoPimpleFoam`.
- `rhoPimpleFoam` is a transient solver for laminar or turbulent flow of compressible gas.

Running the case

- Let us run this case using the automatic script, in the terminal type,

```
1.    $> sh run_solver.sh
2.    $> paraFoam
```

- Fell free to open the file *`run_solver.sh`* to know all the steps.

Field manipulation

- After finding the solution, we can compute the new field variables using the primitive variables computed during the solution stage. In the terminal type:

```
1.  $> rhoPimpleFoam -postProcess -func MachNo
2.  $> rhoPimpleFoam -postProcess -func CourantNo
3.  $> rhoPimpleFoam -postProcess -func wallShearStress
4.  $> rhoPimpleFoam -postProcess -func 'writeObjects(rho)' -time 0
5.  $> rhoPimpleFoam -postProcess -func vorticity
6.  $> postProcess -func vorticity
7.  $> rhoPimpleFoam -postProcess -dict system/externalFunctionObject -latestTime
```

- If the new field variables require information of the simulation database (fluxes, turbulence properties, transport properties), you will need to process as in steps 1-5.

- If the new field variable only requires to use a variable that already exist in the solution folder, you can proceed as in step 6.

Field manipulation

- In step 1 we compute the Mach number. To compute this value, the `postProcess` utility needs to access the `thermophysicalProperties` dictionary.
- In step 2 we compute the Courant number. To compute this value, the `postProcess` utility needs to access the face fluxes (phi).
- In step 3 we compute the wall shear stress. To compute this value, the `postProcess` utility needs to access the transport and turbulence properties.
- In step 4 we compute the density (**rho**) for the initial time (time = 0). To compute this value, the `postProcess` utility needs to access the simulation database.
- In steps 5 and 6 we compute the vorticity field, this field is derived from the velocity field. The `postProcess` utility does not need to access any particular solver information. Both options will give the same output.
- In step 7 we use an external file to compute the derived fields. In this case we are computing the density gradient (**grad(rho)**) and the divergence of the density gradient (**div(grad(rho)**)).
- Remember, in order to compute the derived field **div(grad(rho)**, you need to compute first **grad(rho)**.

Field manipulation

- After finding the solution, we can compute new field variables using the primitive variables computed during the solution stage. In the terminal type:

```
1.   $> postProcess -func 'grad(U)'
2.   $> postProcess -func 'components(U)'
3.   $> postProcess -func 'mag(U)'
4.   $> postProcess -func 'magSqr(U)'
5.   $> postProcess -func 'totalPressureCompressible(rho,U,p)' -noZero
6.   $> postProcess -func 'div(U)' -time 500:1000
7.   $> postProcess -func 'mag(grad(U))' -latestTime
```

- We can also use the utility `postProcess` to compute the average and integral of a specified field over a patch. In the terminal type:

```
8.   $> postProcess -func 'patchAverage(name=inlet,p)' -latestTime
9.   $> postProcess -func 'patchAverage(name=outlet,U)' -latestTime
10.  $> postProcess -func 'patchIntegrate(name=inlet,p)' -latestTime
11.  $> postProcess -func 'patchIntegrate(name=outlet,U)' -latestTime
```

Field manipulation

- In steps 1-11, all the fields are derived from pre-existing fields. The `postProcess` utility does not need to access any particular solver information.
- In step 1 we compute the gradient of the velocity vector **U**. The field is saved as **grad(U)**.
- In step 2 we compute the components of the velocity vector **U**. The components are saved as **Ux**, **Uy** and **Uz**.
- In step 3 we compute the magnitude of the velocity vector **U**. The output is saved as **mag(U)**.
- In step 4 we compute the magnitude squared of the velocity vector **U**. The output is saved as **magSqr(U)**.
- In step 5 we compute the total pressure. The output is saved as **total(p)**. The option **–noZero** means do not compute the value for time zero.
- In step 6 we compute the divergence of the velocity vector **U**. The output is saved as **div(U)**. You will need to define how to interpolate **div(U)** in the *`fvSchemes`* dictionary. The option **–time 500:1000** means save the values between the given range (500-1000).
- In step 7 we compute the magnitude of the gradient of the velocity vector **U**. The output is saved as **mag(Grad(U))**. The option **–latestTime** will compute the value only for the latest saved solution.
- In step 8 we compute the average of **p** over the patch **inlet**.
- In step 9 we compute the average of **U** over the patch **outlet**.
- In step 10 we compute the integral of **p** over the patch **inlet**.
- In step 11 we compute the integral of **U** over the patch **outlet**.

Roadmap

1. ~~On-the-fly postprocessing – functionObjects and the postProcess utility~~
2. ~~Sampling and probing with the postProcess utility~~
3. ~~Field manipulation~~
4. **Data conversion**

Data conversion

- OpenFOAM® gives users a lot of flexibility when it comes to scientific visualization.
- You are not obliged to use OpenFOAM® visualization tools (paraFoam or paraview).
- You can convert the solution obtained with OpenFOAM® to many third-party formats by using OpenFOAM® data conversion utilities.
- If you are looking for a specific format and it is not supported, you can write your own conversion tool.
- In the directory **`$FOAM_UTILITIES/postProcessing/dataConversion`**, you will find the source code of the following data conversion utilities:

  - **foamDataToFluent**
  - **foamToEnsight**
  - **foamToEnsightParts**
  - **foamToGMV**
  - **foamToTecplot360**
  - **foamToTetDualMesh**
  - **foamToVTK**
  - **smapToFoam**

- To get more information on how to use a data conversion utility, you can read the source code or type in the terminal:
  - `$> name_of_data_conversion_utility -help`

Data conversion

ASCII ↔ Binary conversion

```
application     icoFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         50;

deltaT          0.01;

writeControl    runTime;

writeInterval   1;

purgeWrite      0;

writeFormat     binary;

writePrecision  8;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

- Another utility that might come in handy, specially when dealing with large meshes is `foamFormatConvert`.
- This utility converts the mesh and field variables into ascii or binary format.
- In order to manually edit the *`boundary`* file and the field variables dictionaries (initial and boundary conditions), they must be in ascii format.
- After editing these files, we can convert them into binary format.
- Working in binary format can significantly reduce data parsing and dimension of the files (specially for large meshes).
- The drawback is that the files are not human readable anymore.
- To convert ascii files into binary files, just type in the terminal:
  - `$> foamFormatConvert`
- Remember you will need to set the keyword **writeFormat** to binary in the `controlDict` dictionary.
- In the same way, if you want to convert from binary to ascii, set the keyword **writeFormat** to ascii in the `controlDict` dictionary and type in the terminal:
  - `$> foamFormatConvert`

Module 6

Finite volume method overview

Roadmap

1. **Finite Volume Method: A Crash Introduction**
2. On the CFL number
3. Linear solvers in OpenFOAM®
4. Pressure-Velocity coupling in OpenFOAM®
5. Unsteady and steady simulations
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

Finite Volume Method: A Crash introduction

- This a brief introduction to the FVM to illustrate some basic concepts.
- There is much more under the hood.
- We will use the general transport equation as the starting point to explain the FVM,

$$\int_{V_P} \underbrace{\frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)\, dV}_{\text{convective term}} - \int_{V_P} \underbrace{\nabla \cdot (\rho \Gamma_\phi \nabla \phi)\, dV}_{\text{diffusion term}} = \int_{V_P} \underbrace{S_\phi(\phi)\, dV}_{\text{source term}}$$

- Starting from this equation, we can write down the Navier-Stokes equations (NSE).
- So everything we are going to address also applies to the NSE or any set of equations that can be derived form the general transport equation.

Finite Volume Method: A Crash introduction

- This a brief introduction to the FVM to illustrate some basic concepts.
- There is much more under the hood.
- We will use the general transport equation as the starting point to explain the FVM,

$$\int_{V_P} \underbrace{\frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)\, dV}_{\text{convective term}} - \int_{V_P} \underbrace{\nabla \cdot (\rho \Gamma_\phi \nabla \phi)\, dV}_{\text{diffusion term}} = \int_{V_P} \underbrace{S_\phi(\phi)\, dV}_{\text{source term}}$$

Problem statement

- Find the approximate solution to the general transport equation for the transported quantity ϕ in a given domain, with given boundary conditions (BC) and initial conditions (IC).
- It is an initial boundary value problem (IBVP).
- This is a second order equation. Therefore, for good accuracy, it is necessary that the order of the discretization is equal or higher than the order of the equation that is being discretized (in space and time).

Finite Volume Method: A Crash introduction

- Let us use the general transport equation as the starting point to explain the FVM,

$$\int_{V_P} \underbrace{\frac{\partial \rho\phi}{\partial t} dV}_{\text{temporal derivative}} + \int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)\, dV}_{\text{convective term}} - \int_{V_P} \underbrace{\nabla \cdot (\rho \Gamma_\phi \nabla \phi)\, dV}_{\text{diffusion term}} = \int_{V_P} \underbrace{S_\phi(\phi)\, dV}_{\text{source term}}$$

- Hereafter we are going to assume that the discretization practice is at least second order accurate in space and time.
- As consequence of the previous requirement, all dependent variables are assumed to vary linearly around a point P in space and instant t in time,

$$\phi(\mathbf{x}) = \phi_P + (\mathbf{x} - \mathbf{x}_P) \cdot (\nabla \phi)_P \qquad \text{where} \qquad \phi_P = \phi(\mathbf{x}_P)$$

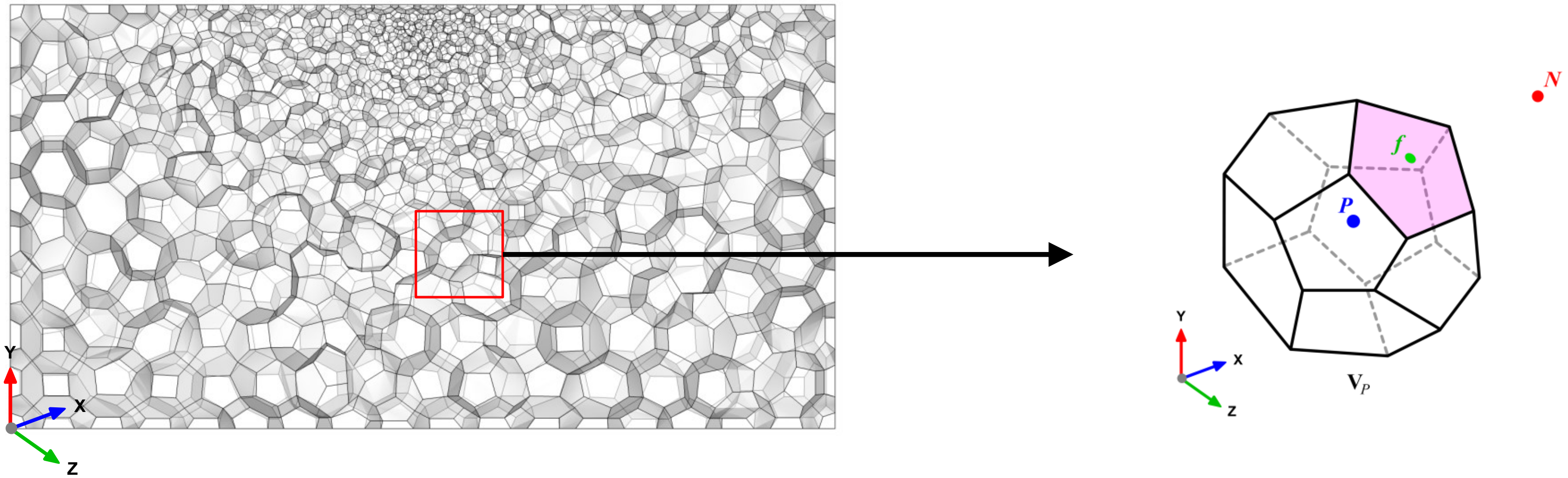
$$\phi(t + \delta t) = \phi^t + \delta t \left(\frac{\partial \phi}{\partial t} \right)^t \qquad \text{where} \qquad \phi^t = \phi(t)$$

Profile assumptions using Taylor expansions around point *P* (in space) and point *t* (in time)

Finite Volume Method: A Crash introduction

Domain discretization – Mesh information and variable arrangement

- Domain discretization (or mesh generation), consist in dividing the solution domain into a finite number of arbitrary control volumes or cells, such as the one illustrated below.
- Inside each control volume the solution is sought.
- The control volumes can be of any shape (*e.g.*, tetrahedrons, hexes, prisms, pyramids, dodecahedrons, and so on). The only requirement is that the faces that made up the control volume need to be planar.
- We also know which control volumes are internal and which control volumes lie on the boundaries.



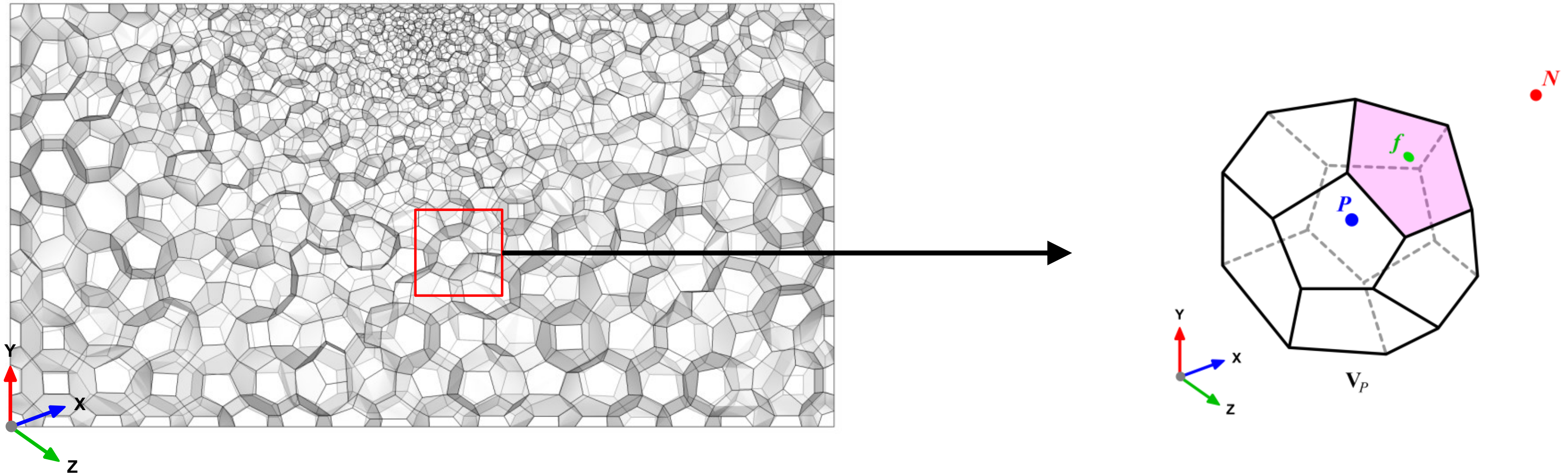
Finite Volume Method: A Crash introduction

Domain discretization – Mesh information and variable arrangement

- In the control volume illustrated, the centroid ***P*** and face center ***f*** are known.
- We also assume that the values of all variables are computed and stored in the centroid of the control volume ***V_p*** and that they are represented by a piecewise constant profile (the mean value),

$$\phi_P = \overline{\phi} = \frac{1}{V_P}\int_{V_P} \phi(\mathbf{x})dV$$

- This is known as the cell centered collocated arrangement.
- <u>All approximations used so far are at least second order accurate.</u>



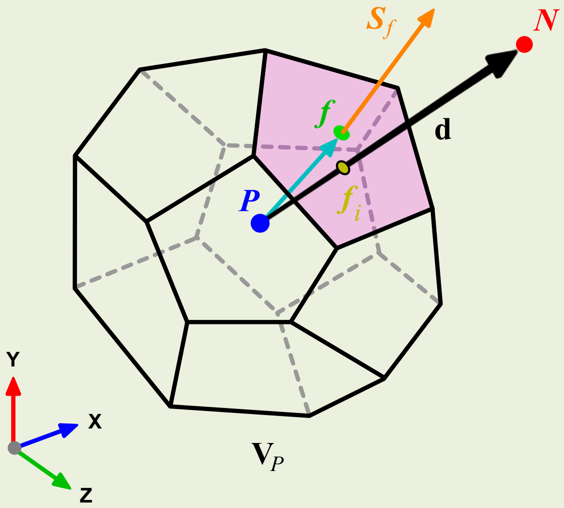
Finite Volume Method: A Crash introduction

Domain discretization – Mesh information and variable arrangement

- Putting all together, it is a lot geometrical information that we need to track.
- A lot of overhead goes into the data book-keeping.
- At the end of the day, the FVM simply consist in conservation of the transported quantities and interpolating information from cell centers to face centers.

Summary:

- The control volume V_P has a volume V and is constructed around point P, which is the centroid of the control volume. Therefore the notation V_P .
- The vector from the centroid P of V_P to the centroid N of the neighboring control volume V_N is named $\mathbf{d}$.
- We also know all neighbors V_N of the control volume V_P
- The control volume faces are labeled f, which also denotes the face center.
- The location where the vector $\mathbf{d}$ intersects a face is f_i .
- The face area vector S_f point outwards from the control volume, is located at the face centroid, is normal to the face and has a magnitude equal to the area of the face.
- The vector from the centroid P to the face center f is named Pf.



Finite Volume Method: A Crash introduction

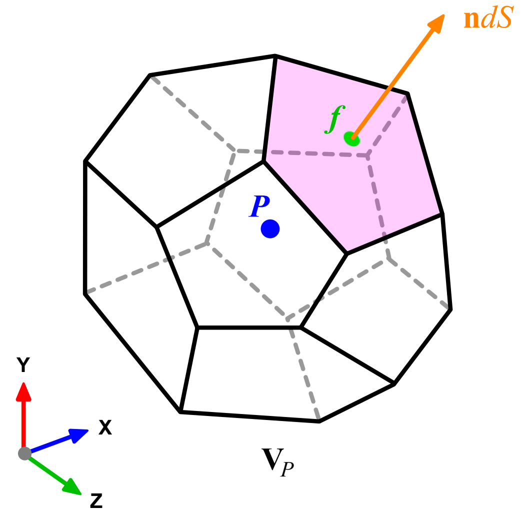
Gauss theorem and face fluxes computation

- Let us recall the Gauss or Divergence theorem,

$$\int_V \nabla \cdot \mathbf{a} dV = \oint_{\partial V} d\mathbf{S} \cdot \mathbf{a}$$

where ∂V_P is a closed surface bounding the control volume V_P and dS represents an infinitesimal surface element are with associated normal $\mathbf{n}$ pointing outwards of the surface ∂V_P , and $\mathbf{n} dS = d\mathbf{S}$

- The Gauss or Divergence theorem simply states that the outward flux of a vector field through a closed surface is equal to the volume integral of the divergence over the region inside the surface.
- This theorem is fundamental in the FVM.
- It is used to convert the volume integrals appearing in the governing equations into surface integrals.



Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- Let us use the Gauss theorem to convert the volume integrals into surface integrals,

$$\int_{V_P} \underbrace{\frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)\, dV}_{\text{convective term}} - \int_{V_P} \underbrace{\nabla \cdot (\rho \Gamma_\phi \nabla \phi)\, dV}_{\text{diffusion term}} = \int_{V_P} \underbrace{S_\phi (\phi)\, dV}_{\text{source term}}$$

$$\int_V \nabla \cdot \mathbf{a} dV = \oint_{\partial V} d\mathbf{S} \cdot \mathbf{a}$$

$$\frac{\partial}{\partial t} \int_{V_P} (\rho \phi)\, dV + \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \mathbf{u} \phi)}_{\text{convective flux}} - \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)}_{\text{diffusive flux}} = \int_{V_P} S_\phi (\phi)\, dV$$

- At this point the problem reduces to interpolating somehow the cell centered values (known quantities) to the face centers.
- That is, we need to compute the gradient terms, source terms, and convective and diffusive fluxes across the faces.

Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Convective term:

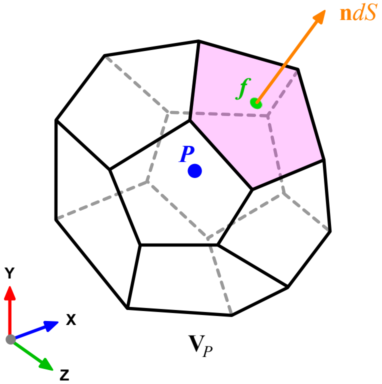
$$\int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)\, dV}_{\text{convective term}} = \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \mathbf{u} \phi)}_{\text{convective flux}} = \sum_f \int_f d\mathbf{S} \cdot (\rho \mathbf{u} \phi)_f \approx \underbrace{\sum_f \mathbf{S}_f \cdot \left(\overline{\rho \mathbf{u} \phi}\right)_f = \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f}$$

By using Gauss theorem we convert volume integrals into surface integrals

where we have approximated the integrant by means of the mid point rule, which is second order accurate

Gauss theorem:

$$\int_V \nabla \cdot \mathbf{a} dV = \oint_{\partial V} d\mathbf{S} \cdot \mathbf{a}$$



Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Diffusive term:

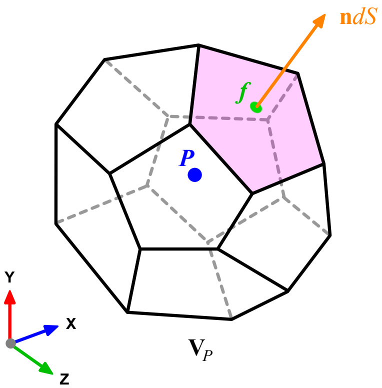
$$\int_{V_P} \underbrace{\nabla \cdot (\rho \Gamma_\phi \nabla \phi)\, dV}_{\text{diffusion term}} = \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)}_{\text{diffusive flux}} = \sum_f \int_f d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)_f \approx \underbrace{\sum_f \mathbf{S}_f \cdot \left(\overline{\rho \Gamma_\phi \nabla \phi}\right)_f}_{} = \sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f$$

By using Gauss theorem we convert volume integrals into surface integrals

where we have approximated the integrant by means of the mid point rule, which is second order accurate

Gauss theorem:

$$\int_V \nabla \cdot \mathbf{a} dV = \oint_{\partial V} d\mathbf{S} \cdot \mathbf{a}$$



Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Gradient term:

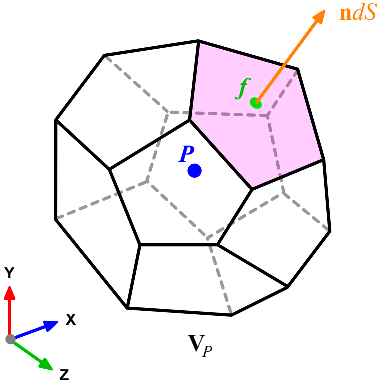
$$(\nabla \phi)_P = \frac{1}{V_P} \sum_f (\mathbf{S}_f \phi_f)$$

where we have approximated the centroid gradients by using the Gauss theorem.

This method is second order accurate and is known as Gauss cell-based.

Gauss theorem:

$$\int_V \nabla \cdot \mathbf{a} dV = \oint_{\partial V} d\mathbf{S} \cdot \mathbf{a}$$



Note:

- There are more methods for gradients computation, *e.g.*, least squares, node-based reconstruction, and so on.
- As there is some algebra involved, we do not provide the demonstration.

Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Source term:

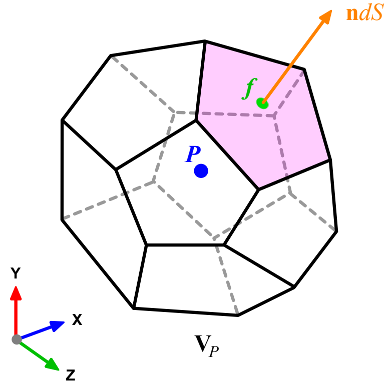
$$\int_{V_P} S_\phi\left(\phi\right) dV = S_c V_P + S_p V_P \phi_P$$

This approximation is exact if S_ϕ is either constant or varies linearly within the control volume; otherwise is second order accurate.

Sc is the constant part of the source term and ***Sp*** is the non-linear part

Gauss theorem:

$$\int_V \nabla \cdot \mathbf{a} dV = \oint_{\partial V} d\mathbf{S} \cdot \mathbf{a}$$



Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- Using the previous equations to evaluate the general transport equation over all the control volumes, we obtain the following semi-discrete equation

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \sum_f \underbrace{\mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f}_{\text{convective flux}} - \sum_f \underbrace{\mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f}_{\text{difussive flux}} = \underbrace{(S_c V_P + S_p V_P \phi_P)}_{\text{source term}}$$

where $\mathbf{S} \cdot (\rho \mathbf{u} \phi) = F^C$ is the convective flux and $\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi) = F^D$ is the diffusive flux.

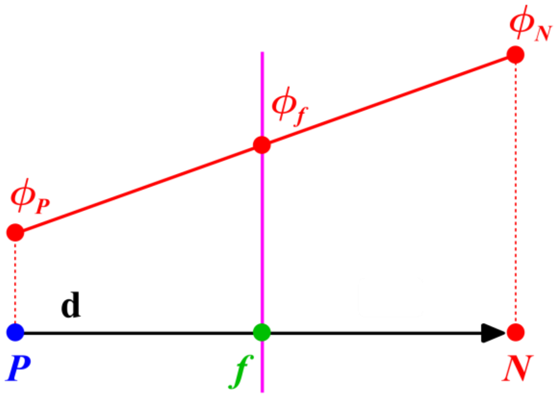
- And recall that all variables are computed and stored at the centroid of the control volumes.

- The face values appearing in the convective and diffusive fluxes have to be computed by some form of interpolation from the centroid values of the control volumes at both sides of face f.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- By looking the figure below, the face values appearing in the convective flux can be computed as follows,



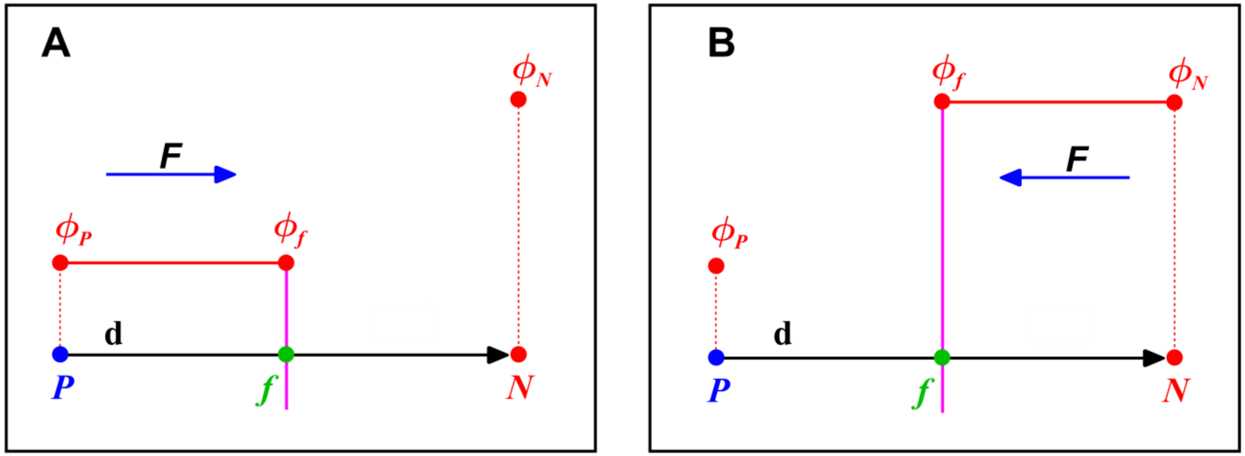
$$\phi_f = f_x \phi_P + (1 - f_x)\,\phi_N$$

$$f_x = \frac{fN}{PN} = \frac{|\,\mathbf{x}_f - \mathbf{x}_N\,|}{|\,\mathbf{d}\,|}$$

- This type of interpolation scheme is known as linear interpolation or central differencing and it is second order accurate.
- However, it may generate oscillatory solutions (unbounded solutions).

Interpolation of the convective fluxes

- By looking the figure below, the face values appearing in the convective flux can be computed as follows,



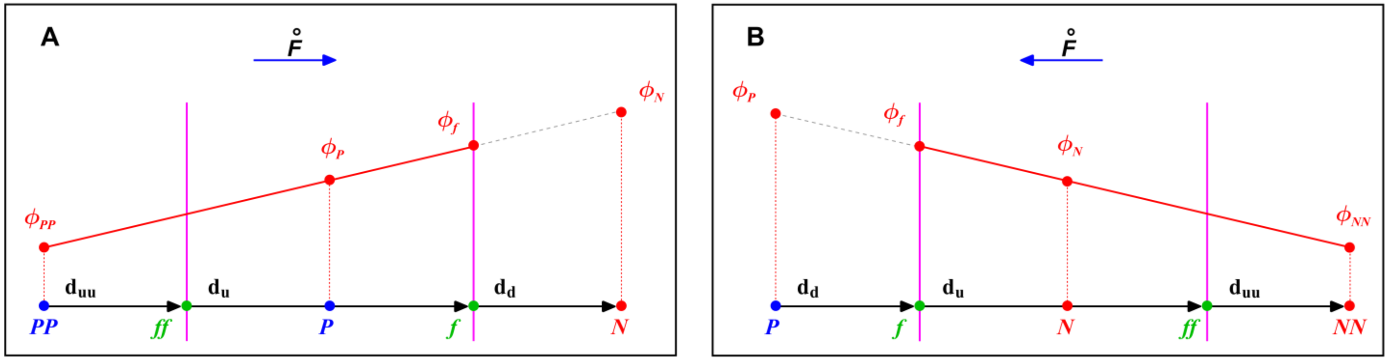
$$\phi_f = \begin{cases} \phi_f = \phi_P & \text{for} \quad \overset{\circ}{F} \geq 0, \\ \phi_f = \phi_N & \text{for} \quad \overset{\circ}{F} < 0. \end{cases}$$

- This type of interpolation scheme is known as upwind differencing and it is first order accurate.
- This scheme is bounded (non-oscillatory) and diffusive.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- By looking the figure below, the face values appearing in the convective flux can be computed as follows,



$$\phi_f = \begin{cases} \phi_P + \dfrac{1}{2}(\phi_P - \phi_{PP}) = \dfrac{2}{3}\phi_P - \dfrac{1}{2}\phi_{PP} & \text{for} \quad \overset{\circ}{F} \geq 0, \\[2ex] \phi_N + \dfrac{1}{2}(\phi_N - \phi_{NN}) = \dfrac{2}{3}\phi_N - \dfrac{1}{2}\phi_{NN} & \text{for} \quad \overset{\circ}{F} < 0. \end{cases}$$

- This type of interpolation scheme is known as second order upwind differencing (SOU), linear upwind differencing (LUD) or Beam-Warming (BW), and it is second order accurate.
- For highly convective flows or in the presence of strong gradients, this scheme is oscillatory (unbounded).

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- To prevent oscillations in the SOU, we add a gradient or slope limiter function $\psi(r)$.

$$\phi_f = \begin{cases} \phi_P + \frac{1}{2}\psi_P^-(\phi_P - \phi_{PP}) & \text{for} \quad \overset{\circ}{F} \geq 0, \\ \phi_N - \frac{1}{2}\psi_P^+(\phi_{NN} - \phi_N) & \text{for} \quad \overset{\circ}{F} < 0. \end{cases}$$

- When the limiter detects strong gradients or changes in slope, it switches locally to low resolution (upwind).
- The concept of the limiter function $\psi(r)$ is based on monitoring the ratio of successive gradients, *e.g.*,

$$r_P^- = \frac{\phi_N - \phi_P}{\phi_P - \phi_{PP}} \quad \text{and} \quad r_P^+ = \frac{\phi_P - \phi_N}{\phi_N - \phi_{NN}}$$

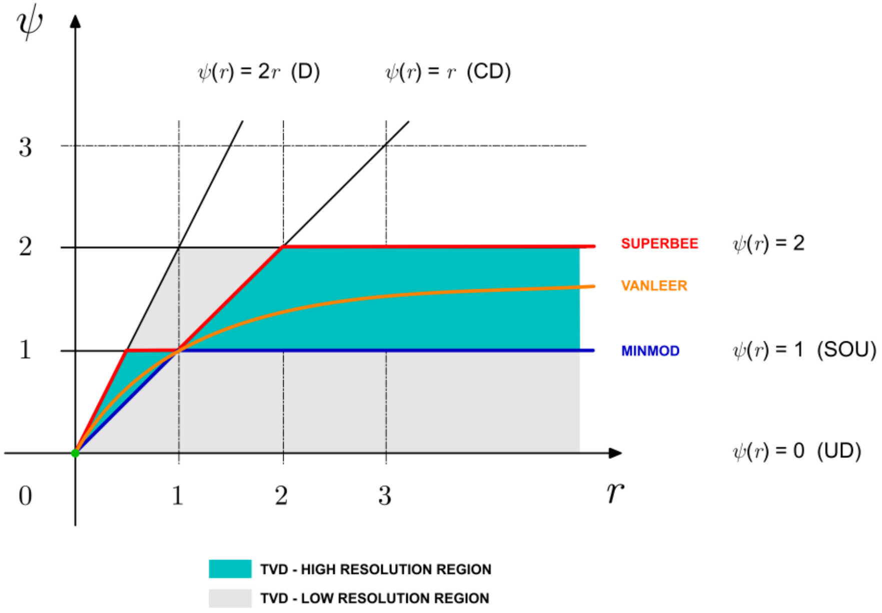
- By adding a well-designed limiter function $\psi(r)$, we get a high resolution (second order accurate) and bounded scheme (HR). This is a TVD scheme.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

- A TVD scheme, is a scheme that does not create new local undershoots and/or overshoots in the solution or amplify existing extremes.
- In CFD we want stable, non-oscillatory, bounded, high order schemes.
- The Sweby diagram (Sweby, 1984), gives the necessary and sufficient conditions for a scheme to be TVD.
- In the figure, the shaded area represents the admissible TVD region. However, not all limiter functions are second order.
- High resolution schemes falls in the blue area and low resolution schemes falls in the grey area.
- The drawback of the limiters is that they reduce the accuracy of the scheme **locally** to first order (low resolution scheme), when *r* < 0 (sharp gradient, opposite slopes or zero gradient). However, this is justified when it serves to suppress oscillations.
- No particular limiter has been found to work well for all problems, and a particular choice is usually made on a trial and error basis.

$$\phi_f = \begin{cases} \phi_P + \dfrac{1}{2}\psi_P^-(\phi_P - \phi_{PP}) & \text{for} \quad \overset{\circ}{F} \geq 0, \\ \phi_N - \dfrac{1}{2}\psi_P^+(\phi_{NN} - \phi_N) & \text{for} \quad \overset{\circ}{F} < 0. \end{cases}$$

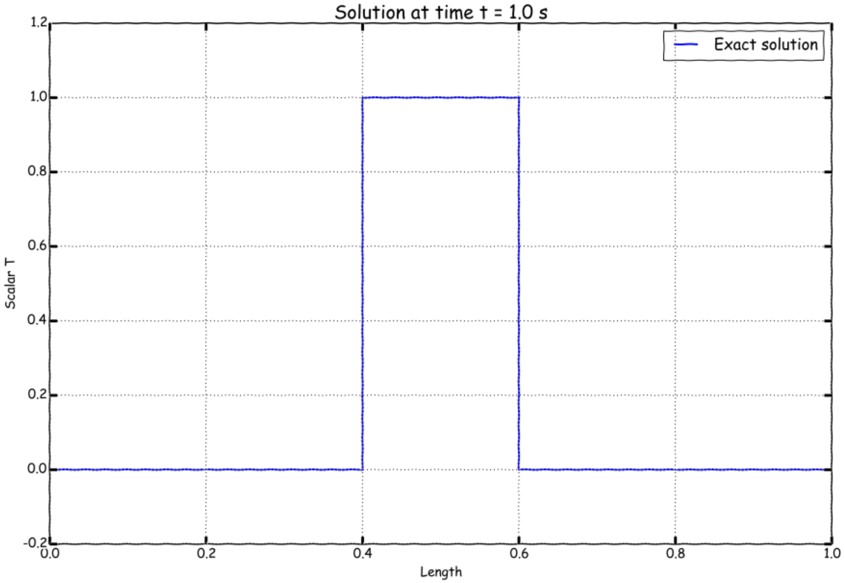


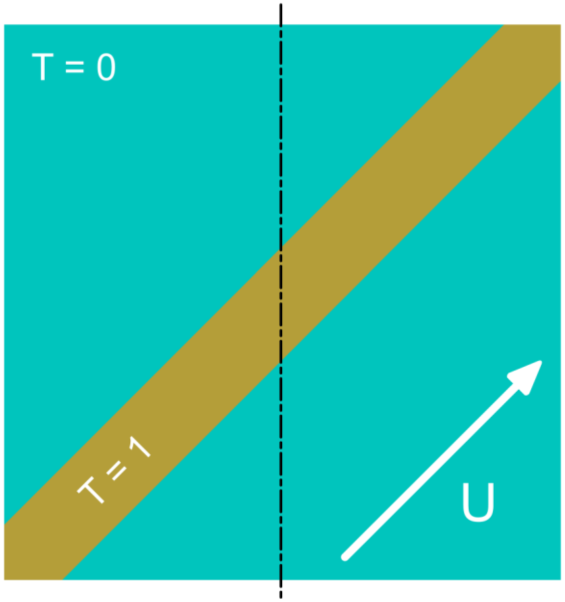
UD = upwind
SOU = second order upwind
CD = central differencing
D = downwind

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

- Let us see how the upwind, linear upwind, linear, and Minmod TVD schemes behave in a numerical schemes killer test case:
  - The oblique double step profile in a uniform vector field (pure convection).
- Even if this problem seems to be easy, from the numerical point of view is difficult to resolve due to the strong discontinuities.
- This problem has an exact solution.



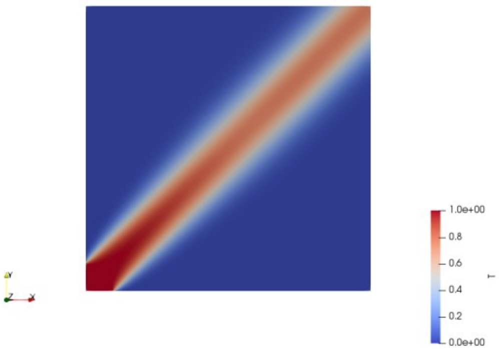


Finite Volume Method: A Crash introduction

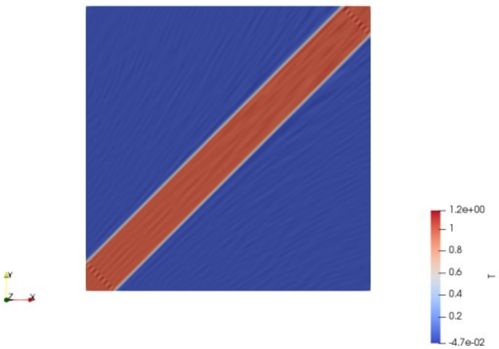
Interpolation of the convective fluxes – TVD schemes

- Qualitative comparison of the upwind, linear upwind, linear, and Minmod TVD schemes

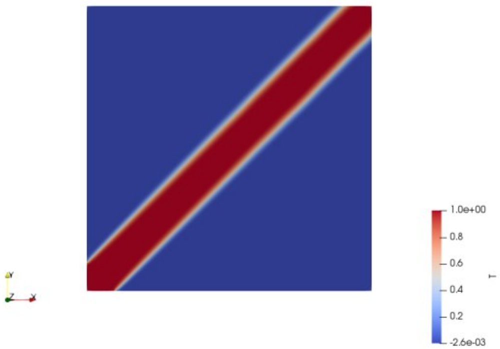
Upwind – 1st order
Very bounded but too Diffusive



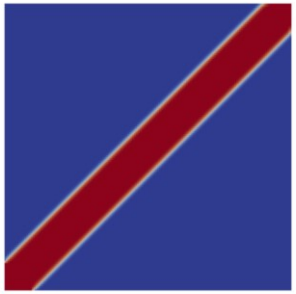
Linear – 2nd order
Very accurate but too oscillatory



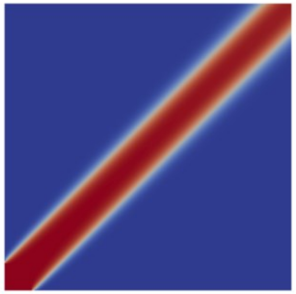
Linear Upwind – 2nd order
Bounded and accurate



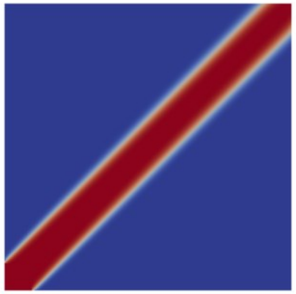
SuperBee – TVD high resolution
Compressive



Minmod – TVD high resolution
Diffusive

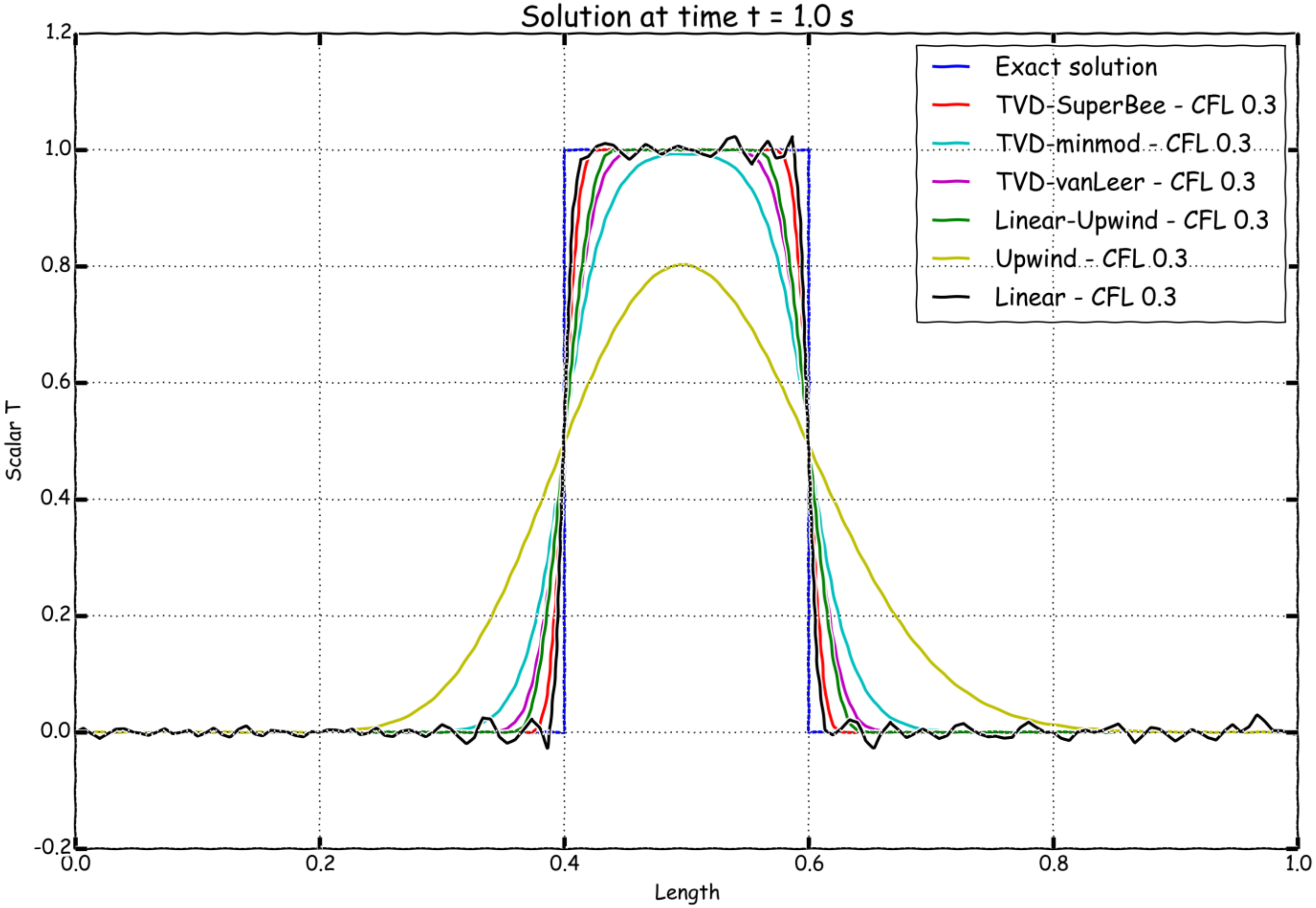


vanLeer – TVD high resolution
Smooth



Interpolation of the convective fluxes – TVD schemes

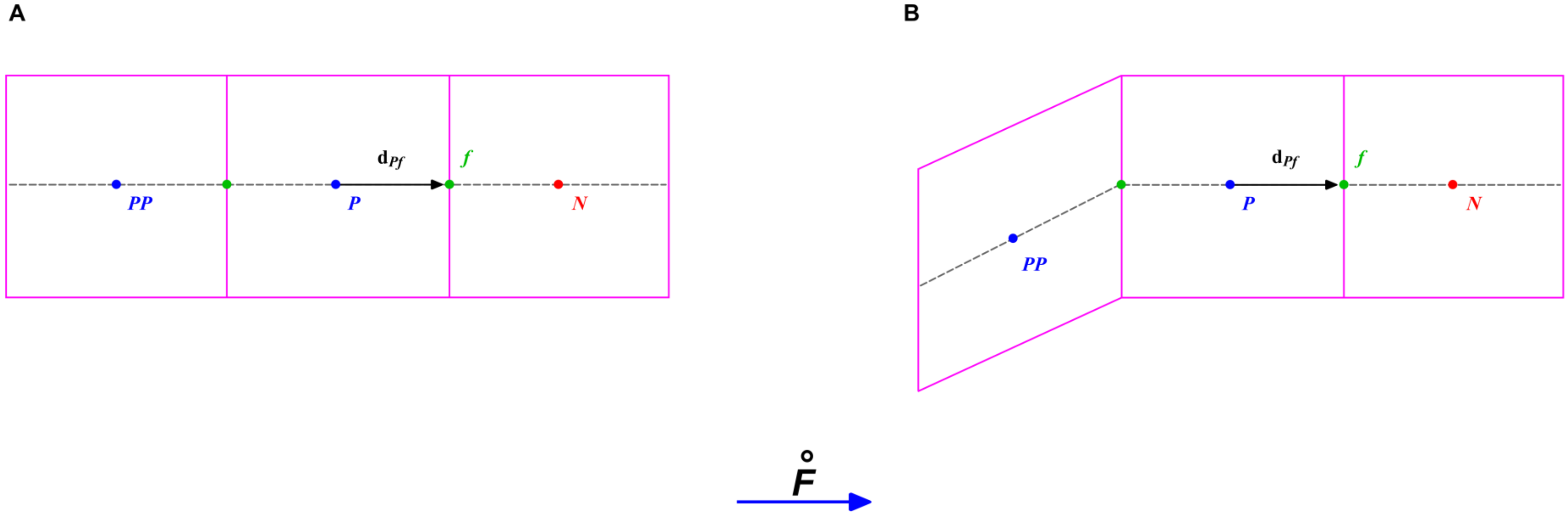
- Quantitative comparison of the upwind, linear upwind, linear, and Minmod TVD schemes



Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Unstructured meshes

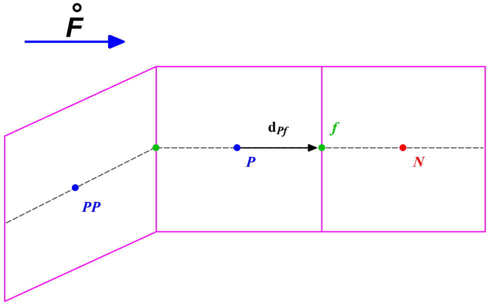
- In the previous explanation, we assumed a line structure (figure A). That is, the cell centers PP, P, and N are all aligned.
- In unstructured meshes (which are often used in industrial cases), most of the times the cell center PP is not aligned with the vector connecting cells P and N (figure B). Therefore, extending the previous formulations to these meshes is not very straightforward.
- Higher-order schemes for unstructured meshes are an area of active research, and new ideas continue to emerge.



Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Unstructured meshes

- A simple way around this problem is to redefine higher-order schemes in terms of gradients at the control volume P.
- For example, using the gradient of the cells, we can compute the face values as follows,



Upwind → $\phi_f = \phi_P$

Central difference → $\phi_f = \phi_P + \nabla \phi_f \cdot \mathbf{d}_{Pf}$

Second order upwind differencing → $\phi_f = \phi_P + (2\nabla \phi_P - \nabla \phi_f) \cdot \mathbf{d}_{Pf}$

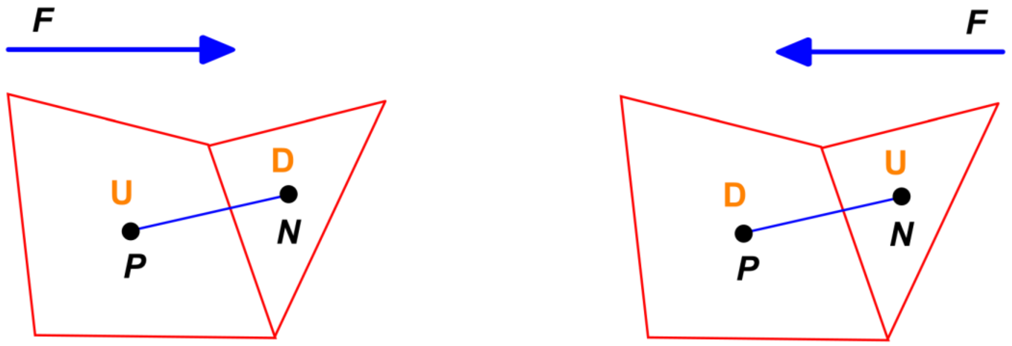
- Notice that in this new formulation the cell PP does not appear anymore.
- The problem now turns in the accurate evaluation of the gradients at the cell and face centers.
- For example, the gradients at the cell centers can be computed using the Gauss method, and then interpolated to the face centers.
- At this point, we are only missing the reconstruction of the cell center gradients at the face centers, this is explained latter.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Unstructured meshes

- In unstructured meshes, as often the value of the node PP (of NN) is not available or straightforward to compute, the ratio of successive gradients *r* can be computed as follows [1],

$$r = \left[\frac{(2\nabla\phi_P \cdot \mathbf{d}_{PN})}{\phi_D - \phi_U} - 1 \right]$$



U → Upwind

D → Downwind

- As you can see, the value of *r* depends on the flow direction.
- There are many ways to compute *r*. This is an area of active research

Reference:

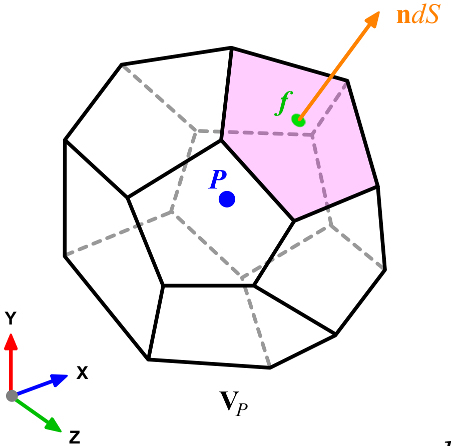
[1] Darwish, M. S., Moukalled, F., “TVD schemes for unstructured grids”

Finite Volume Method: A Crash introduction

Gradients computation at cell centers

- There are many methods for the computation of the cell centered gradients, *e.g.*, least squares, Gauss cell-based, Gauss node-based, and so on.
- Using the Gauss cell-based method, the cell centered gradients can be computed as follows,

$$(\nabla \phi)_P = \frac{1}{V_P} \sum_f (\mathbf{S}_f \phi_f)$$



$$\mathbf{n}dS = d\mathbf{S}$$

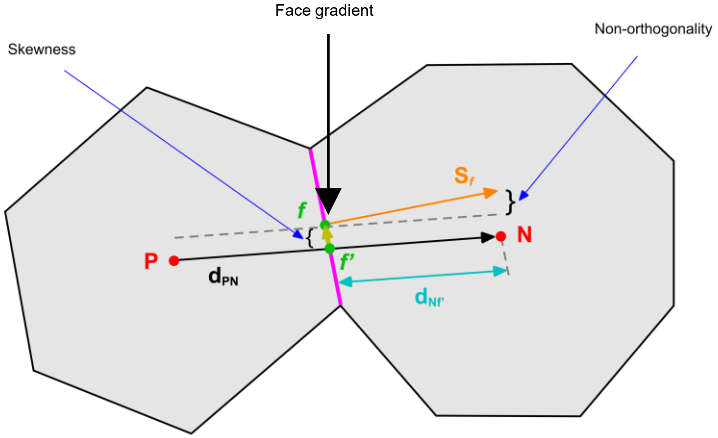
- This approximation is second order accurate given that the mesh quality is acceptable, and the volume of the cell is finite.
- In general, the least squares method tends to be more accurate.

Finite Volume Method: A Crash introduction

Gradients reconstruction at face centers

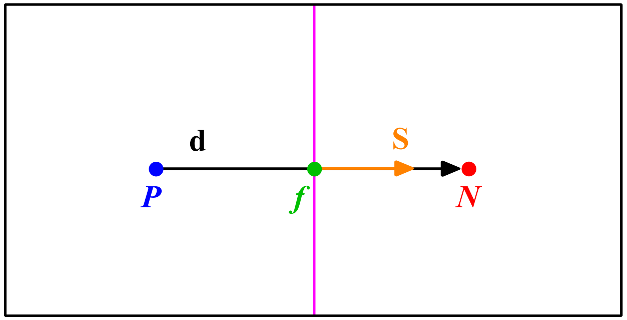
- Face gradients $\nabla \phi_f$ arise from the discretization process of the convective and diffusive terms.
- One way to reconstruct the face gradient $\nabla \phi_f$, is by using weighted interpolation of the cell centered quantities $\nabla \phi_P$ and $\nabla \phi_N$.
- Mesh non-orthogonality and skewness introduce errors when approximating the face gradients, so corrections need to be added.
- This is an iterative process, where we compute successively better approximations to the gradients starting from an initial approximation.

$$\nabla \phi_f = f_x \nabla \phi_P + (1 - f_x) \nabla \phi_N \qquad \text{where} \qquad f_x = fN/PN$$



Interpolation of diffusive fluxes in an orthogonal mesh

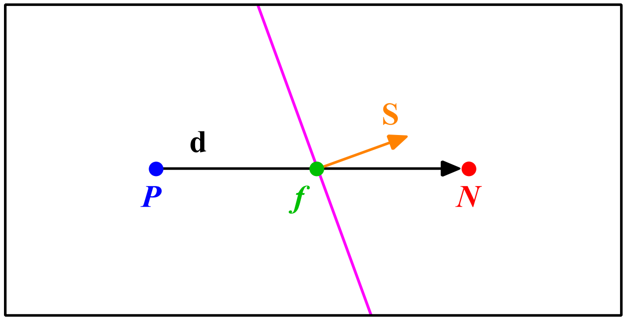
- By looking the figure below, the face values appearing in the diffusive flux in an orthogonal mesh can be computed as follows,



$$\mathbf{S} \cdot (\nabla \phi)_f = |\mathbf{S}| \frac{\phi_N - \phi_P}{|\mathbf{d}|}.$$

- This is a central difference approximation of the first order derivative. This type of approximation is second order accurate.

- By looking the figure below, the face values appearing in the diffusive flux in a non-orthogonal mesh (20°) can be computed as follows,



$$\mathbf{S} \cdot (\nabla \phi)_f = \underbrace{|\Delta_{\perp}| \frac{\phi_N - \phi_P}{|\mathbf{d}|}}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k} \cdot (\nabla \phi)_f}_{\text{non-orthogonal contribution}}.$$

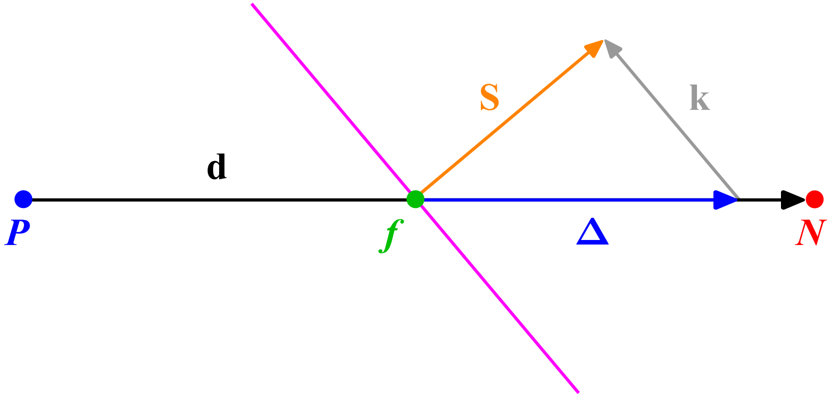
- This type of approximation is second order accurate but involves a larger truncation error. It also uses a larger numerical stencil, which make it less stable.

- Remember, the non-orthogonal angle is the angle between the vector **S** and the vector **d**

Finite Volume Method: A Crash introduction

Correction of diffusive fluxes in a non-orthogonal mesh

- By looking the figures below, the face values appearing in the diffusive flux in a non-orthogonal mesh (40°) can be computed as follows.
- Using the over-relaxed approach, the diffusive fluxes can be corrected as follow,



Over-relaxed approach

$$\Delta_\perp = \frac{\mathbf{d}}{\mathbf{d}\cdot\mathbf{S}}|\mathbf{S}|^2.$$

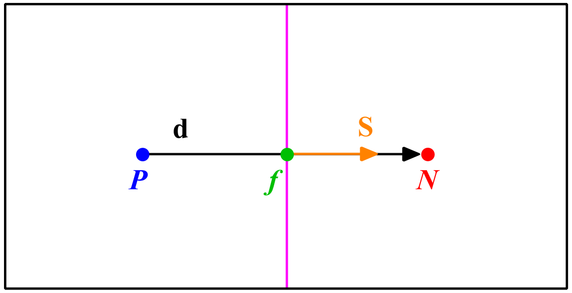
$$\mathbf{S} = \Delta_\perp + \mathbf{k}.$$

$$\mathbf{S}\cdot(\nabla\phi)_f = \underbrace{|\Delta_\perp|\frac{\phi_N-\phi_P}{|\mathbf{d}|}}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k}\cdot(\nabla\phi)_f}_{\text{non-orthogonal contribution}}.$$

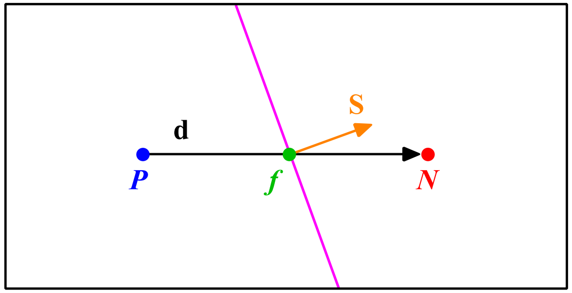
Finite Volume Method: A Crash introduction

Mesh induced errors

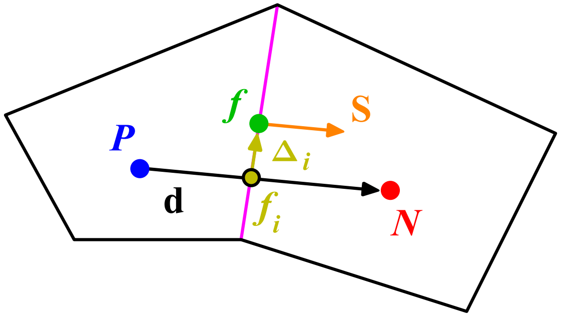
- In order to maintain second order accuracy, and to avoid unboundedness, we need to correct non-orthogonality and skewness errors.
- The ideal case is to have an orthogonal and non skew mesh, but this is the exception rather than the rule.



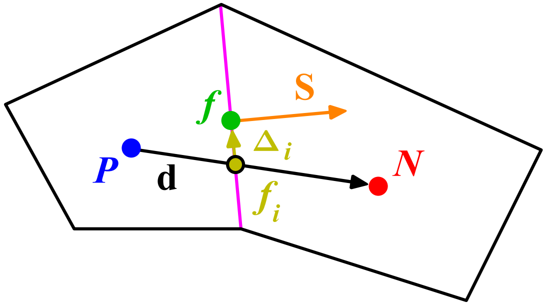
Orthogonal and non skew mesh



Non-orthogonal and non skew mesh



Orthogonal and skew mesh



Non-orthogonal and skew mesh

Finite Volume Method: A Crash introduction

Temporal discretization

- Using the previous equations to evaluate the general transport equation over all the control volumes, we obtain the following semi-discrete equation,

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \sum_f \underbrace{\mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f}_{\text{convective flux}} - \sum_f \underbrace{\mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f}_{\text{difussive flux}} = \underbrace{(S_c V_P + S_p V_P \phi_P)}_{\text{source term}}$$

where $\mathbf{S} \cdot (\rho \mathbf{u} \phi) = F^C$ is the convective flux and $\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi) = F^D$ is the diffusive flux.

- After spatial discretization, we can proceed with the temporal discretization. By proceeding in this way we are using the Method of Lines (MOL).
- The main advantage of the MOL method, is that it allows us to select numerical approximations of different accuracy for the spatial and temporal terms. Each term can be treated differently to yield to different accuracies.

Finite Volume Method: A Crash introduction

Temporal discretization

- Now, we evaluate in time the semi-discrete general transport equation

$$\int_t^{t+\Delta t} \left[\left(\frac{\partial \rho \phi}{\partial t} \right)_P V_P + \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f - \sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f \right] dt$$

$$= \int_t^{t+\Delta t} (S_c V_P + S_p V_P \phi_P) \, dt.$$

- At this stage, we can use any time discretization scheme, *e.g.*, Crank-Nicolson, euler implicit, forward euler, backward differencing, adams-bashforth, adams-moulton.
- It should be noted that the order of the temporal discretization of the transient term does not need to be the same as the order of the discretization of the spatial terms.
- Each term can be treated differently to yield different accuracies. As long as the individual terms are at least second order accurate, the overall accuracy will also be second order.

Finite Volume Method: A Crash introduction

Linear system solution

- After spatial and temporal discretization and by using equation

$$\int_t^{t+\Delta t}\left[\left(\frac{\partial \rho\phi}{\partial t}\right)_P V_P + \sum_f \mathbf{S}_f\cdot(\rho\mathbf{u}\phi)_f - \sum_f \mathbf{S}_f\cdot(\rho\Gamma_\phi\nabla\phi)_f\right]dt = \int_t^{t+\Delta t}(S_c V_P + S_p V_P\phi_P)dt$$

in every control volume V_P of the domain, a system of linear algebraic equations for the transported quantity ϕ is assembled,

$$\begin{pmatrix}
a_{11} & a_{12} & & \ddots & & & & \\
a_{21} & a_{22} & a_{23} & & \ddots & & & \\
& \ddots & \ddots & \ddots & & \ddots & & \\
\ddots & & \ddots & \ddots & \ddots & & \ddots & \\
& a_S & & a_W & a_P & a_E & & a_N \\
& & \ddots & & \ddots & \ddots & \ddots & \\
& & & \ddots & & \ddots & \ddots & \ddots \\
& & & & \ddots & & \ddots & a_{PP}
\end{pmatrix}
\times
\begin{pmatrix}
\phi_S \\ \\ \phi_W \\ \phi_P \\ \phi_E \\ \\ \phi_N
\end{pmatrix}
=
\begin{pmatrix}
b_S \\ \\ b_W \\ b_P \\ b_E \\ \\ b_N
\end{pmatrix}$$

- This system can be solved by using any iterative or direct method.

Finite Volume Method: A Crash introduction

So, what does OpenFOAM® do?

- It simply discretize in space and time the governing equations in arbitrary polyhedral control volumes over the whole domain.
- Assembling in this way a large set of linear discrete algebraic equations (DAE), and then it solves this system of DAE to find the solution of the transported quantities.
- Therefore, we need to give to OpenFOAM® the following information:
  - Discretization of the solution domain or the mesh.
    - This information is contained in the directory `constant/polyMesh`
  - Boundary conditions and initials conditions.
    - This information is contained in the directory `0`
  - Physical properties such as density, gravity, diffusion coefficient, viscosity, etc.
    - This information is contained in the directory `constant`
  - Physics involve, such as turbulence modeling, mass transfer, source terms, dynamic meshes, multiphase models, combustion models, etc.
    - This information is contained in the directories `constant` and/or `system`

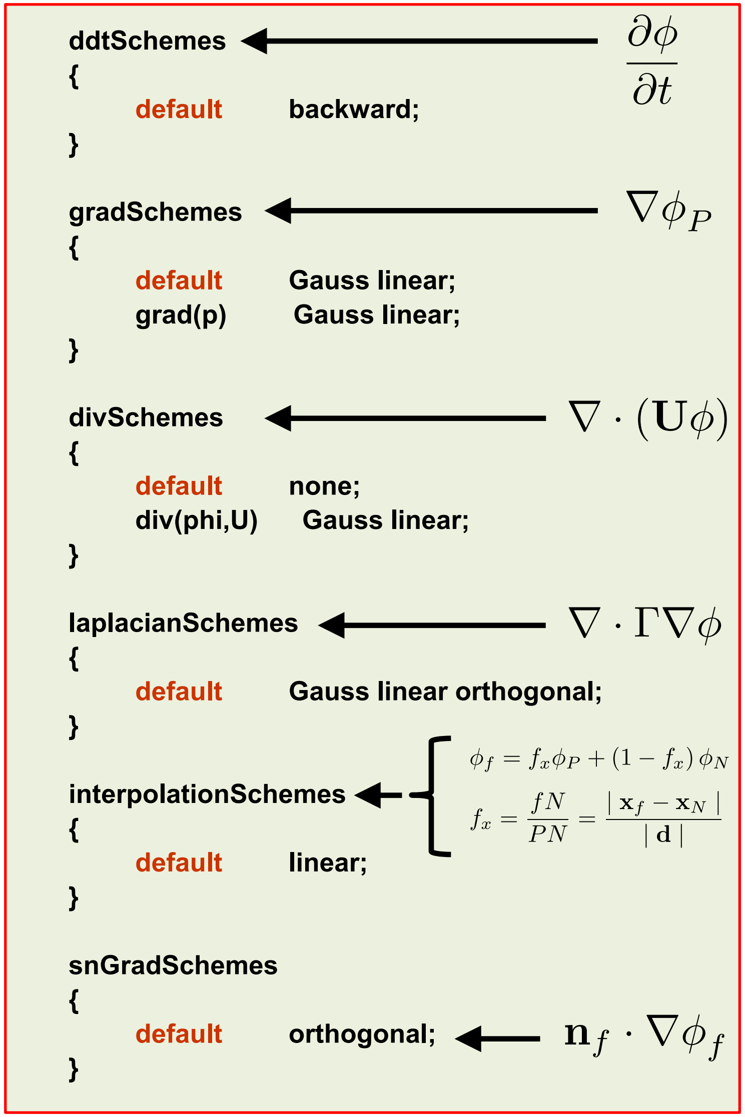
Finite Volume Method: A Crash introduction

So, what does OpenFOAM® do?

- Therefore, we need to give to OpenFOAM® the following information:
  - How to discretize in space each term of the governing equations (diffusive, convective, gradient and source terms).
    - This information is set in the *`system/fvSchemes`* dictionary.
  - How to discretize in time the obtained semi-discrete governing equations.
    - This information is set in the *`system/fvSchemes`* dictionary.
  - How to solve the linear system of discrete algebraic equations (crunch numbers).
    - This information is set in the *`system/fvSolution`* dictionary.
  - Set runtime parameters and general instructions on how to run the case (such as time step, maximum CFL number, solution saving frequency, and so on).
    - This information is set in the *`system/controlDict`* dictionary.
  - Additionally, we may set sampling and monitors for post-processing (**functionObjects**).
    - This information is set in the *`system/fvSchemes`* dictionary or in the specific sampling dictionaries located in the directory **`system/`**

Finite Volume Method: A Crash introduction

Where do we set all the discretization schemes in OpenFOAM®?



```
ddtSchemes
{
        default       backward;
}

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

divSchemes
{
        default       none;
        div(phi,U)    Gauss linear;
}

laplacianSchemes
{
        default       Gauss linear orthogonal;
}

interpolationSchemes
{
        default       linear;
}

snGradSchemes
{
        default       orthogonal;
}
```

$$\frac{\partial \phi}{\partial t}$$

$$\nabla \phi_P$$

$$\nabla \cdot (\mathbf{U}\phi)$$

$$\nabla \cdot \Gamma \nabla \phi$$

$$\left\{ \begin{array}{l} \phi_f = f_x \phi_P + (1 - f_x)\, \phi_N \\ f_x = \dfrac{fN}{PN} = \dfrac{\mid \mathbf{x}_f - \mathbf{x}_N \mid}{\mid \mathbf{d} \mid} \end{array} \right.$$

$$\mathbf{n}_f \cdot \nabla \phi_f$$

- The *fvSchemes* dictionary contains the information related to the discretization schemes for the different terms appearing in the governing equations.
- The discretization schemes can be chosen in a term-by-term basis.
- The keyword **ddtSchemes** refers to the time discretization.
- The keyword **gradSchemes** refers to the gradient term discretization.
- The keyword **divSchemes** refers to the convective term discretization.
- The keyword **laplacianSchemes** refers to the Laplacian term discretization.
- The keyword **interpolationSchemes** refers to the method used to interpolate values from cell centers to face centers. It is unlikely that you will need to use something different from linear.
- The keyword **snGradSchemes** refers to the discretization of the surface normal gradients evaluated at the faces.
- Remember, if you want to know the options available for each keyword you can use the banana method.

Time discretization schemes

- There are many time discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
  - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/ddtSchemes`

- These are the time discretization schemes that you will use most of the times:
  - **steadyState:** for steady state simulations (implicit/explicit).
  - **Euler:** time dependent first order (implicit/explicit), bounded.
  - **backward:** time dependent second order (implicit), bounded/unbounded.
  - **CrankNicolson:** time dependent second order (implicit), bounded/unbounded.

- First order methods are bounded and stable, but diffusive.
- Second order methods are accurate, but they might become oscillatory.
- At the end of the day, we always want a second order accurate solution.
- If you keep the CFL less than one when using the Euler method, numerical diffusion is not that much (however, we advise you to do your own benchmarking).

Finite Volume Method: A Crash introduction

Time discretization schemes

- The **Crank-Nicolson** method as it is implemented in OpenFOAM®, uses a blending factor.

```
ddtSchemes
{
         default       CrankNicolson  ψ ;
}
```

- Setting ψ to 0 is equivalent to running a pure **Euler** scheme (robust but first order accurate).
- By setting the blending factor equal to 1 you use a pure **Crank-Nicolson** (accurate but oscillatory, formally second order accurate).
- If you set the blending factor to 0.5, you get something in between first order accuracy and second order accuracy, or in other words, you get the best of both worlds.
- A blending factor of 0.7-0.9 is safe to use for most applications (stable and accurate).

Finite Volume Method: A Crash introduction

Convective terms discretization schemes

- There are many convective terms discretization schemes available in OpenFOAM® (more than 50 last time we checked).
- You will find the source code in the following directory:
  - `$WM_PROJECT_DIR/src/finiteVolume/interpolation/surfaceInterpolation`

- These are the convective discretization schemes that you will use most of the times:
  - **upwind**: first order accurate.
  - **linearUpwind**: second order accurate, bounded.
  - **linear:** second order accurate, unbounded.
  - A good TVD scheme (**vanLeer** or **Minmod**): TVD, second order accurate, bounded.
  - **limitedLinear**: second order accurate, unbounded, but more stable than pure linear. Recommended for LES simulations (kind of similar to the Fromm method).
  - **LUST**: blended 75% **linear** and 25% **linearUpwind** scheme

- First order methods are bounded and stable but diffusive.
- Second order methods are accurate, but they might become oscillatory.
- At the end of the day, we always want a second order accurate solution.

Finite Volume Method: A Crash introduction

Convective terms discretization schemes

- When you use **linearUpwind** for **div(phi,U)**, you need to tell OpenFOAM® how to compute the velocity gradient or **grad(U):**

```
gradSchemes
{
        grad(U)         cellMDLimited Gauss linear 1.0;
}

divSchemes
{
        div(phi,U)      Gauss linearUpwind                grad(U);
}
```

- Same applies for every transported quantity (e.g. **k**, **epsilon**, **omega**, **T**)

Gradient terms discretization schemes

- There are many gradient discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
  - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/gradSchemes`
- These are the gradient discretization schemes that you will use most of the times:
  - **Gauss linear** (cell-based method)
  - **Gauss pointLinear** (node-based method; more accurate than the cell-based method)
  - **leastSquares**

- To avoid overshoots or undershoots when computing the gradients, you can use gradient limiters.
- Gradient limiters increase the stability of the method but add diffusion due to clipping.
- You will find the source code in the following directory:
  - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/gradSchemes/limitedGradSchemes`
- These are the most important gradient limiter schemes available in OpenFOAM®:
  - **cellLimited, cellMDLimited, faceLimited, faceMDLimited**
- All of the gradient discretization schemes are at least second order accurate.

Finite Volume Method: A Crash introduction

Gradient terms discretization schemes

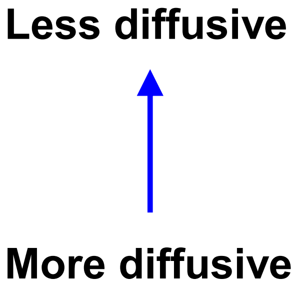
- These are the gradient limiter schemes available in OpenFOAM®:

cellMDLimited

cellLimited

faceMDLimited

faceLimited

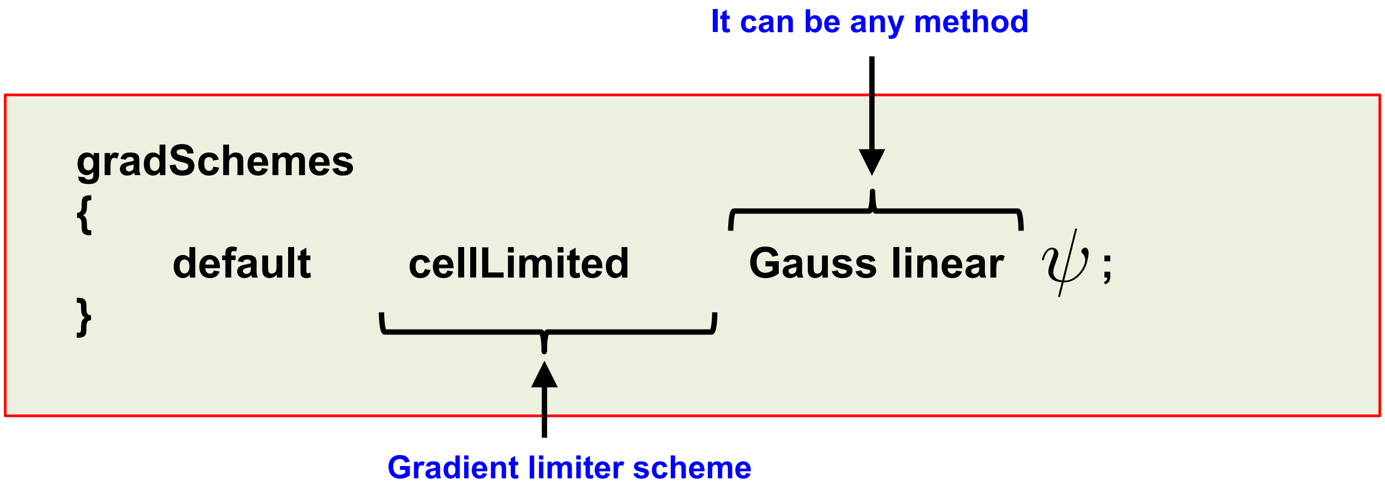


Note: for smooth field variation, cell limiting may provide less numerical dissipation on meshes with skewed cells

- Cell limiters will limit cell-to-cell values.
- Face limiters will limit face-to-cell values.
- The multi-directional (dimensional) limiters (**cellMDLimited** and **faceMDLimited**), will apply the limiter in each face direction separately.
- The standard limiters (**cellLimited** and **faceLimited**), will apply the limiter to all components of the gradient.
- The default method is the Minmod.

Gradient terms discretization schemes

- The gradient limiter implementation in OpenFOAM®, uses a blending factor ψ.



It can be any method

```
gradSchemes
{
        default      cellLimited      Gauss linear  ψ;
}
```

Gradient limiter scheme

- Setting ψ to 0 is equivalent to turning off the gradient limiter. You gain accuracy but the solution might become unbounded.
- By setting the blending factor equal to 1 the limiter is set to be very aggressive (kind of saying that it is always on). You gain stability but you give up accuracy (due to gradient clipping).
- If you set the blending factor to 0.5, you get the best of both worlds.
- You can use limiters with all gradient discretization schemes.

Finite Volume Method: A Crash introduction

Laplacian terms discretization schemes

- There are many Laplacian terms discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
  - **`$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/snGradSchemes`**

- These are the Laplacian terms discretization schemes that you will use most of the times:
  - **orthogonal:** mainly limited for hexahedral meshes with no grading (a perfect mesh). Second order accurate, bounded on perfect meshes, without non-orthogonal corrections.
  - **corrected:** for meshes with grading and non-orthogonality. Second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
  - **limited:** for meshes with grading and non-orthogonality. Second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
  - **uncorrected:** usually limited to hexahedral meshes with very low non-orthogonality. Second order accurate, without non-orthogonal corrections. Stable but more diffusive than the limited and corrected methods.

$$\mathbf{S}\cdot(\nabla\phi)_f = |\mathbf{S}|\frac{\phi_N-\phi_P}{|\mathbf{d}|}.$$

Can be computed using the over-relaxed approach

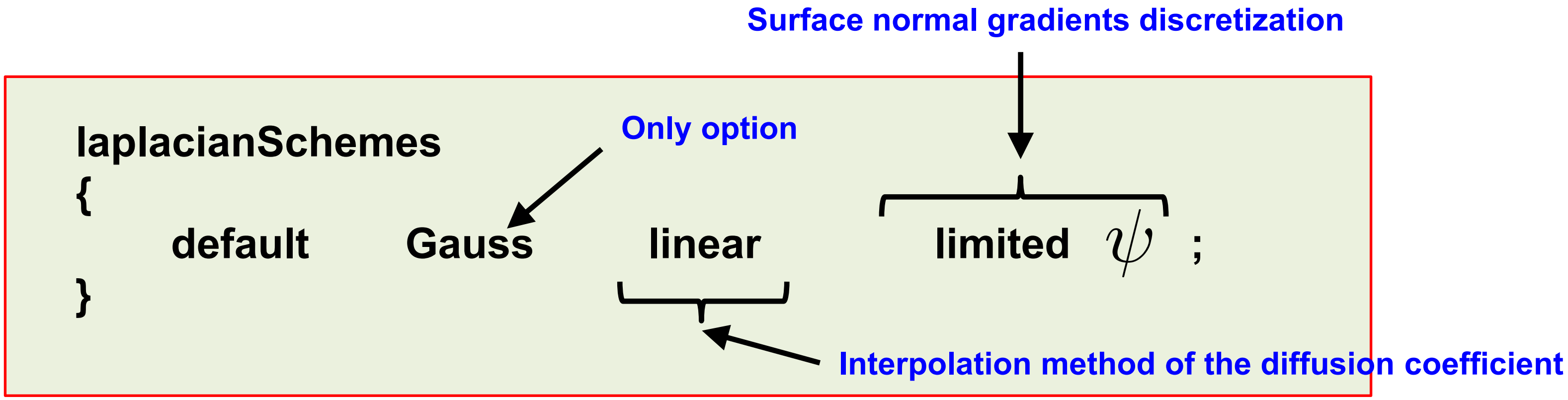
$$\mathbf{S}\cdot(\nabla\phi)_f = \underbrace{|\Delta_\perp|\frac{\phi_N-\phi_P}{|\mathbf{d}|}}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k}\cdot(\nabla\phi)_f}_{\text{non-orthogonal contribution}}.$$

$$\mathbf{S}\cdot(\nabla\phi)_f = \underbrace{|\Delta_\perp|\frac{\phi_N-\phi_P}{|\mathbf{d}|}}_{\text{orthogonal contribution}} + \cancel{\underbrace{\mathbf{k}\cdot(\nabla\phi)_f}_{\text{non-orthogonal contribution}}}.$$

Can be computed using the over-relaxed approach

Laplacian terms discretization schemes

- The limited method uses a blending factor ψ .



Surface normal gradients discretization

Only option

```
laplacianSchemes
{
    default      Gauss      linear       limited  ψ  ;
}
```

Interpolation method of the diffusion coefficient

- Setting ψ to 1 is equivalent to using the **corrected** method. You gain accuracy, but the solution might become unbounded.
- By setting the blending factor equal to 0 is equivalent to using the **uncorrected** method. You give up accuracy but gain stability.
- If you set the blending factor to 0.5, you get the best of both worlds. In this case, the non-orthogonal contribution does not exceed the orthogonal part. You give up accuracy but gain stability.
- For meshes with non-orthogonality less than 70, you can set the blending factor to 1.
- For meshes with non-orthogonality between 70 and 85, you can set the blending factor to 0.5
- For meshes with non-orthogonality more than 85, it is better to get a better mesh. But if you want to use that mesh, you can set the blending factor to 0.333-0.5, and increase the number of non-orthogonal corrections.
- If you are doing LES or DES simulations, use a blending factor of 1 (this means that you need good meshes).

Finite Volume Method: A Crash introduction

Laplacian terms discretization schemes

- Just to make it clear, the blending factor ψ is used to avoid the non-orthogonal contribution exceeding the orthogonal part.
- That is, non-orthogonal contribution ≤ orthogonal contribution.

The blending factor works as a limiter acting on this term (non-orthogonal contribution)

$$\mathbf{S} \cdot (\nabla \phi)_f = \underbrace{|\Delta_{\perp}| \frac{\phi_N - \phi_P}{|\mathbf{d}|}}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k} \cdot (\nabla \phi)_f}_{\text{non-orthogonal contribution}} .$$

Implicit part

Explicit part

- In meshes with large non-orthogonality, the explicit term can lead to unboundedness and eventually divergence.
- This limiting is local, similar to the treatment done for the connective terms when using slope limiters and TVD schemes.
- The explicit contribution is added to the RHS of the linear system (source term), so if this term becomes too large it will lead to convergence problems.
- It becomes harder to guarantee diagonal dominance of the matrix of coefficient.

Finite Volume Method: A Crash introduction

Laplacian terms discretization schemes

- The surface normal gradients terms usually use the same method as the one chosen for the Laplacian terms.
- For instance, if you are using the **limited 1** method for the Laplacian terms, you can use the same method for **snGradSchemes**:

```
laplacianSchemes
{
        default                 Gauss linear        limited 1;
}

snGradSchemes
{
        default                 limited 1;
}
```

What method should I use?

```
ddtSchemes
{
        default             CrankNicolson 0;
}
gradSchemes
{
        default             cellLimited Gauss linear 0.5;
        grad(U)             cellLimited Gauss linear 1;
}
divSchemes
{
        default                         none;
        div(phi,U)                      Gauss linearUpwindV grad(U);
        div(phi,omega)                  Gauss linearUpwind   default;
        div(phi,k)                      Gauss linearUpwind   default;
        div((nuEff*dev(T(grad(U)))))        Gauss linear;
}
laplacianSchemes
{
        default             Gauss linear limited 1;
}
interpolationSchemes
{
        default             linear;
}
snGradSchemes
{
        default             limited 1;
}
```

- **<u>This setup is recommended for most of the cases.</u>** 👍
- It is equivalent to the default method you will find in commercial solvers.
- In overall, this setup is second order accurate and fully bounded.
- According to the quality of your mesh, you will need to change the blending factor of the **laplacianSchemes** and **snGradSchemes** keywords.
- To keep temporal diffusion to a minimum, use a CFL number less than 2, and preferably below 1.
- If during the simulation the turbulence quantities become unbounded, you can safely change the discretization scheme to upwind. After all, turbulence is diffusion.
- For gradient discretization the **leastSquares** method is more accurate. But we have found that it is a little bit oscillatory in tetrahedral meshes.

Finite Volume Method: A Crash introduction

A very accurate but oscillatory numerics

```
ddtSchemes
{
    default         backward;
}
gradSchemes
{
    default         Gauss leastSquares;
}
divSchemes
{
    default                     none;
    div(phi,U)                  Gauss linear;
    div(phi,omega)              Gauss limitedlinear 1;
    div(phi,k)                  Gauss limitedLinear 1;
    div((nuEff*dev(T(grad(U)))))    Gauss linear;
}
laplacianSchemes
{
    default         Gauss linear limited 1;
}
interpolationSchemes
{
    default         linear;
}
snGradSchemes
{
    default         limited 1;
}
```

- If you are looking for more accuracy, you can use this method.
- In overall, this setup is second order accurate but oscillatory.
- Use this setup with LES simulations or laminar flows with no complex physics and meshes with overall good quality.
- Use this method with good quality meshes.
- According to the quality of your mesh, you will need to change the blending factor of the **laplacianSchemes** and **snGradSchemes** keywords.

Finite Volume Method: A Crash introduction

A very stable but too diffusive numerics

```
ddtSchemes
{
    default          Euler;
}
gradSchemes
{
    default          cellLimited Gauss linear 1;
    grad(U)          cellLimited Gauss linear 1;
}
divSchemes
{
    default                    none;
    div(phi,U)                 Gauss upwind;
    div(phi,omega)             Gauss upwind;
    div(phi,k)                 Gauss upwind;
    div((nuEff*dev(T(grad(U)))))     Gauss linear;
}
laplacianSchemes
{
    default          Gauss linear limited 0.5;
}
interpolationSchemes
{
    default          linear;
}
snGradSchemes
{
    default          limited 0.5;
}
```

- If you are looking for extra stability, you can use this method.
- This setup is very stable but too diffusive.
- This setup is first order in space and time.
- You can use this setup to start the solution in the presence of bad quality meshes or strong discontinuities.
- Remember, you can start using a first order method and then switch to a second order method.
- According to the quality of your mesh, you will need to change the blending factor of the **laplacianSchemes** and **snGradSchemes** keywords.
- **Start robustly, end with accuracy.**
- You can use this method for troubleshooting. If the solution diverges, you better check boundary conditions, physical properties, and so on.

Roadmap

1. ~~Finite Volume Method: A Crash Introduction~~
2. **On the CFL number**
3. Linear solvers in OpenFOAM®
4. Pressure-Velocity coupling in OpenFOAM®
5. Unsteady and steady simulations
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

On the CFL number

- First of all, what is the CFL or Courant number?
- In one dimension, the CFL number is defined as,

$$CFL = \frac{u\ \Delta t}{\Delta x}$$

- The CFL number is a measure of how much information (u) traverses a computational grid cell (Δx) in a given time-step (Δt).
- The CFL number is not a magical number.
- The CFL number is a necessary condition to guarantee the stability of the numerical scheme.
- But not all numerical schemes have the same stability requirements.
- By doing a linear stability study, we can find the stability requirements of each numerical scheme (but this is out of the scope of this lecture).

On the CFL number

- Let us now talk about the **CFL number condition**. The **CFL number condition** is the maximum allowable CFL number a solver can use.
- For the **N** dimensional case, the CFL number condition becomes,

$$CFL = \Delta t \sum_{i=1}^{n} \frac{u_i}{\Delta x_i} \leq CFL_{max}$$

- CFD solvers can be explicit and implicit.
- Explicit and implicit solvers have different stability requirements.
- Implicit numerical methods are **unconditionally stable**.
- In other words, they are not constrained to the **CFL number condition**.
- However, the fact that you are using a numerical method that is unconditionally stable, **does not mean that you can choose a time step of any size**.
- The time-step must be chosen in such a way that it resolves the time-dependent features, and it maintains the solver stability.
- When we use implicit solvers, we need to assemble a large system of equations.
- The memory requirements of implicit methods are much higher than those of explicit methods.
- In OpenFOAM®, most of the solvers are implicit.
- In our personal experience, we have been able to go up to a CFL = 5.0 while maintaining the accuracy and without increasing too much the computational cost.
- But as we are often interested in the unsteadiness of the solution, we usually use a CFL number in the order of 1.0

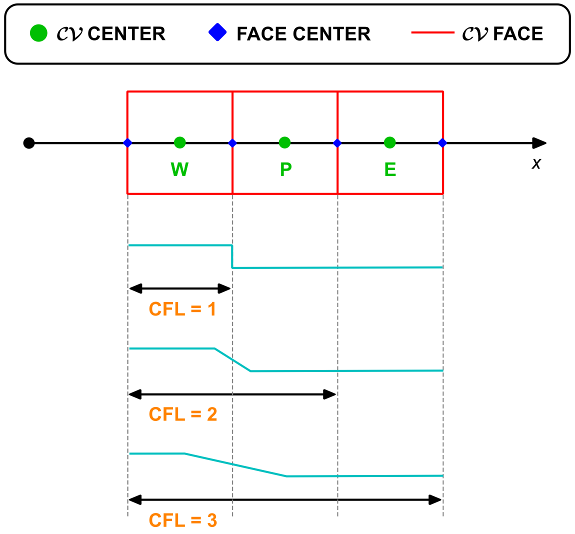
On the CFL number

The CFL number for dummies

- I like to see the CFL number as follows,

$$CFL = \frac{u\,\Delta t}{\Delta x} = \frac{u}{\Delta x/\Delta t} = \frac{\text{speed of the PDE}}{\text{speed of the mesh}}$$

- It is an indication of the amount of information that propagates through one cell (or many cells), in one time-step.



- By the way, and this is extremely important, the CFL condition is a necessary condition for stability (and hence convergence).
- But it is not always sufficient to guarantee stability.
- Other properties of the discretization schemes that you should observe are: conservationess, boundedness, transportiveness, and accuracy.

On the CFL number

How to control the CFL number

```
application         pimpleFoam;

startFrom           latestTime;

startTime           0;

stopAt              endTime;

endTime             10;

deltaT              0.0001;

writeControl        runTime;

writeInterval       0.1;

purgeWrite          0;

writeFormat         ascii;

writePrecision      8;

writeCompression off;

timeFormat          general;

timePrecision       6;

runTimeModifiable yes;

adjustTimeStep      yes;

maxCo               2.0;
maxDeltaT           0.001;
```

- You can control the CFL number by changing the mesh cell size or changing the time-step size.
- The time step size is set in the `controlDict` dictionary.
- The easiest way is by changing the time-step size.
- If you refine the mesh, and you would like to have the same CFL number as the base mesh, you will need to decrease the time-step size.
- On the other side, if you coarse the mesh and you would like to have the same CFL number as the base mesh, you will need to increase the time-step size.
- The keyword **deltaT** controls the time-step size of the simulation (0.0001 seconds in this generic case).
- If you use a solver that supports adjustable time-step (**adjustTimeStep**), you can set the maximum CFL number and maximum allowable time-step using the keywords **maxCo** and **maxDeltaT**, respectively.

On the CFL number

How to control the CFL number

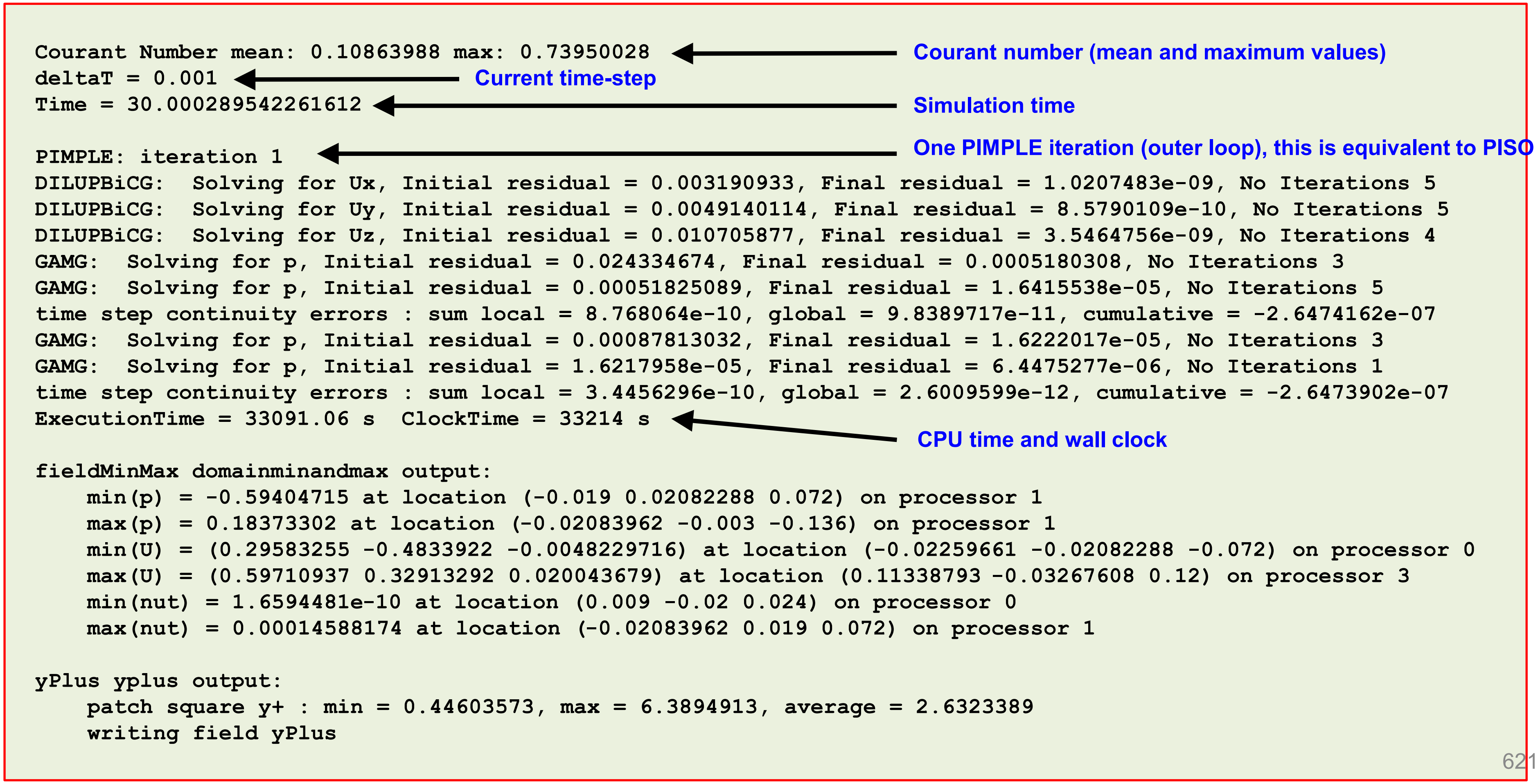
```
application         pimpleFoam;

startFrom           latestTime;

startTime           0;

stopAt              endTime;

endTime             10;

deltaT              0.0001;

writeControl        runTime;

writeInterval       0.1;

purgeWrite          0;

writeFormat         ascii;

writePrecision      8;

writeCompression off;

timeFormat          general;

timePrecision       6;

runTimeModifiable yes;

adjustTimeStep      yes;

maxCo               2.0;
maxDeltaT           0.001;
```

- The option **adjustTimeStep** will automatically adjust the time step to achieve the maximum desired courant number (**maxCo**) or time-step size (**maxDeltaT**).
- When any of these conditions is reached, the solver will stop scaling the time-step size.
- To use these features, you need to turn-on the option **adjustTimeStep**.
- Remember, the first time-step of the simulation is done using the value defined with the keyword **deltaT** and then it is automatically scaled (up or down), to achieve the desired maximum values (**maxCo** and **maxDeltaT**).
- It is recommended to start the simulation with a low time-step in order to let the solver scale-up the time-step size.
- If you want to change the values on-the-fly, you need to turn-on the option **runTimeModifiable**.
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.

On the CFL number

The output screen

- This is the output screen of a solver supporting the option **adjustTimeStep**.
- In this case **maxCo** is equal 2 and **maxDeltaT** is equal to 0.001.
- Notice that the solver reached the maximum allowable **maxDeltaT**.



```
Courant Number mean: 0.10863988 max: 0.73950028
deltaT = 0.001
Time = 30.000289542261612

PIMPLE: iteration 1
DILUPBiCG:  Solving for Ux, Initial residual = 0.003190933, Final residual = 1.0207483e-09, No Iterations 5
DILUPBiCG:  Solving for Uy, Initial residual = 0.0049140114, Final residual = 8.5790109e-10, No Iterations 5
DILUPBiCG:  Solving for Uz, Initial residual = 0.010705877, Final residual = 3.5464756e-09, No Iterations 4
GAMG:  Solving for p, Initial residual = 0.024334674, Final residual = 0.0005180308, No Iterations 3
GAMG:  Solving for p, Initial residual = 0.00051825089, Final residual = 1.6415538e-05, No Iterations 5
time step continuity errors : sum local = 8.768064e-10, global = 9.8389717e-11, cumulative = -2.6474162e-07
GAMG:  Solving for p, Initial residual = 0.00087813032, Final residual = 1.6222017e-05, No Iterations 3
GAMG:  Solving for p, Initial residual = 1.6217958e-05, Final residual = 6.4475277e-06, No Iterations 1
time step continuity errors : sum local = 3.4456296e-10, global = 2.6009599e-12, cumulative = -2.6473902e-07
ExecutionTime = 33091.06 s  ClockTime = 33214 s

fieldMinMax domainminandmax output:
    min(p) = -0.59404715 at location (-0.019 0.02082288 0.072) on processor 1
    max(p) = 0.18373302 at location (-0.02083962 -0.003 -0.136) on processor 1
    min(U) = (0.29583255 -0.4833922 -0.0048229716) at location (-0.02259661 -0.02082288 -0.072) on processor 0
    max(U) = (0.59710937 0.32913292 0.020043679) at location (0.11338793 -0.03267608 0.12) on processor 3
    min(nut) = 1.6594481e-10 at location (0.009 -0.02 0.024) on processor 0
    max(nut) = 0.00014588174 at location (-0.02083962 0.019 0.072) on processor 1

yPlus yplus output:
    patch square y+ : min = 0.44603573, max = 6.3894913, average = 2.6323389
    writing field yPlus
```

Courant number (mean and maximum values)

Current time-step

Simulation time

One PIMPLE iteration (outer loop), this is equivalent to PISO

CPU time and wall clock

Roadmap

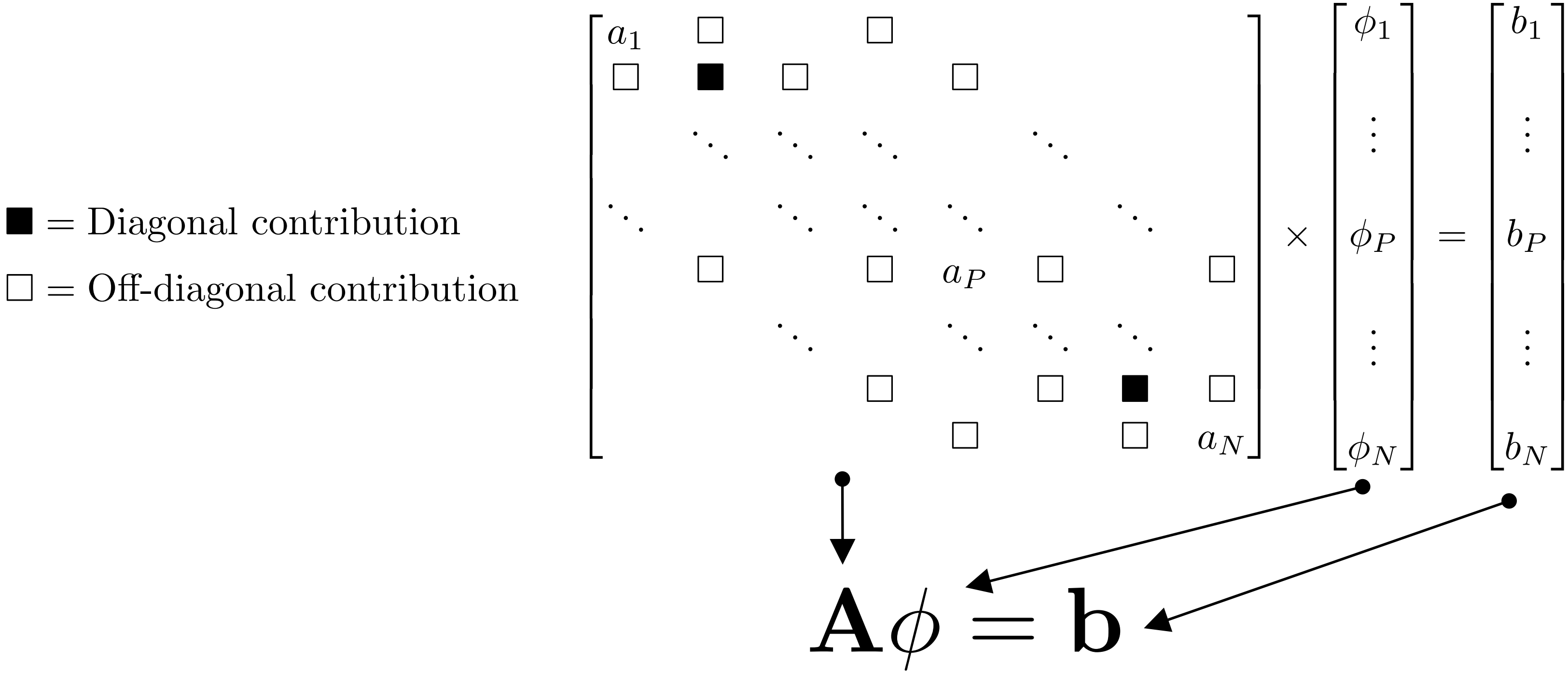
1. ~~Finite Volume Method: A Crash Introduction~~
2. ~~On the CFL number~~
3. **Linear solvers in OpenFOAM®**
4. Pressure-Velocity coupling in OpenFOAM®
5. Unsteady and steady simulations
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

Linear solvers in OpenFOAM®

- After spatial and temporal discretization and by using equation

$$\int_t^{t+\Delta t}\left[\left(\frac{\partial \rho\phi}{\partial t}\right)_P V_P + \sum_f \mathbf{S}_f \cdot (\rho\mathbf{u}\phi)_f - \sum_f \mathbf{S}_f \cdot (\rho\Gamma_\phi \nabla\phi)_f\right] dt = \int_t^{t+\Delta t} (S_c V_P + S_p V_P \phi_P) dt$$

in every control volume V_P of the domain, a system of linear algebraic equations for the transported quantity ϕ is assembled



■ $=$ Diagonal contribution

□ $=$ Off-diagonal contribution

$$\mathbf{A}\phi = \mathbf{b}$$

- This system can be solved by using any iterative or direct method.

Linear solvers in OpenFOAM®

Linear solvers – *fvSolution* dictionary

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
    pFinal
    {
        $p;
        relTol    0;
    }
    U
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors    2;
    nNonOrthogonalCorrectors    1;
}
```

- The equation solvers, tolerances, and algorithms are controlled from the sub-dictionary **solvers** located in the `fvSolution` dictionary file.
- In the dictionary file `fvSolution` and depending on the solver you are using you will find the additional sub-dictionaries **PISO, PIMPLE,** and **SIMPLE**, which will be described later.
- In this dictionary is where we tell OpenFOAM® how to crunch numbers.
- The **solvers** sub-dictionary specifies each linear solver that is used for each equation being solved.
- The linear solvers distinguish between symmetric matrices and asymmetric matrices.
- If you forget to define a linear-solver or use the wrong one, OpenFOAM® will let you know.
- The syntax for each entry within the **solvers** sub-dictionary uses a keyword that is the word relating to the variable being solved in the particular equation and the options related to the linear solver.

Linear solvers in OpenFOAM®

Linear solvers – *fvSolution* dictionary

```
solvers
{
    p
    {
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
    }
    pFinal
    {
         $p;
        relTol    0;
    }
    U
    {
        solver              PBiCGStab;
        preconditioner      DILU;
        tolerance           1e-08;
        relTol              0;
    }
}

PISO
{
    nCorrectors    2;
    nNonOrthogonalCorrectors    1;
}
```

- In this generic case, to solve the pressure (**p**) we are using the **PCG** method with the **DIC** preconditioner, an absolute **tolerance** equal to 1e-06 and a relative tolerance **relTol** equal to 0.
- The entry **pFinal** refers to the final pressure correction (notice that we are using macro syntax), and we are using a relative tolerance **relTol** equal to 0 (disabled).
- To solve the velocity field (**U**) we are using the **PBiCGStab** method with the **DILU** preconditoner, an absolute **tolerance** equal to 1e-08 and a relative tolerance **relTol** equal to 0.
- The linear solvers will iterative until reaching any of the tolerance values set by the user or reaching a maximum value of iterations (optional entry).
- FYI, solving for the velocity is relatively inexpensive, whereas solving for the pressure is expensive.
- The pressure equation is particularly important as it governs mass conservation.
- If you do not solve the equations accurately enough (tolerance), the physics might be wrong.
- Selection of the tolerance is of paramount importance and it might be problem dependent.

Linear solvers in OpenFOAM®

Linear solvers – *fvSolution* dictionary

```
solvers
{
    p
    {
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
    }
    pFinal
    {
        $p;
        relTol     0;
    }
    U
    {
        solver              PBiCGStab;
        preconditioner      DILU;
        tolerance           1e-08;
        relTol              0;
    }
}

PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors     1;
}
```

- The linear solvers are iterative, *i.e.*, they are based on reducing the equation residual over a succession of solutions.
- The residual is a measure of the error in the solution so that the smaller it is, the more accurate the solution.
- More precisely, the residual is evaluated by substituting the current solution into the equation and taking the magnitude of the difference between the left- and right-hand sides (L2-norm).

$$\left|\mathbf{A}\phi^{\mathrm{k}} - \mathbf{b}\right| = \left|\mathbf{r}^{\mathrm{k}}\right|$$

- It is also normalized to make it independent of the scale of the problem being analyzed.

$$\text{Residual} = \frac{|\mathbf{r}|}{\text{Normalization factor}} < \text{Tolerance}$$

Linear solvers in OpenFOAM®

Linear solvers – *fvSolution* dictionary

```
solvers
{
    p
    {
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
    }
    pFinal
    {
        $p;
        relTol    0;
    }
    U
    {
        solver              PBiCGStab;
        preconditioner      DILU;
        tolerance           1e-08;
        relTol              0;
        minIter             3;
        maxIter             100;
    }
}

PISO
{
    nCorrectors    2;
    nNonOrthogonalCorrectors    1;
}
```

- Before solving an equation for a particular field, the initial residual is evaluated based on the current values of the field.
- After each solver iteration the residual is re-evaluated. The solver stops if either of the following conditions are reached:
- The residual falls below the solver tolerance, **tolerance**.
- The ratio of current to initial residuals falls below the solver relative tolerance, **relTol**.
- The number of iterations exceeds a maximum number of iterations, **maxIter**.
- The solver tolerance should represent the level at which the residual is small enough that the solution can be deemed sufficiently accurate.
- The keyword **maxIter** is optional and the default value is 1000.
- The user can also define the minimum number of iterations using the keyword **minIter**. This keyword is optional, and the default value is 0.

Linear solvers in OpenFOAM®

Linear solvers

- These are the linear solvers (segregated) available in OpenFOAM®:
  - **GAMG** → Multigrid solver
  - **PBiCG** → Newton-Krylov solver
  - **PBiCGStab** → Newton-Krylov solver
  - **PCG** → Newton-Krylov solver
  - **smoothSolver** → Smooth solver
  - **diagonalSolver**
- You will find the source code of the linear solvers in the following directory:
  - **$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/solvers**
- When using Newton-Krylov solvers, you need to define preconditoners.
- These are the preconditioners available in OpenFOAM®:
  - **DIC**
  - **DILU**
  - **FDIC**
  - **GAMG**
  - **diagonal**
  - **noPreconditioner**
- You will find the source code in the following directory:
  - **$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/preconditioners**
- The **smoothSolver** solver requires the specification of a smoother.
- These are the smoothers available in OpenFOAM®:
  - **DIC**
  - **DICGaussSeidel**
  - **DILU**
  - **DILUGaussSeidel**
  - **FDIC**
  - **GaussSeidel**
  - **nonBlockingGaussSeidel**
  - **symGaussSeidel**
- You will find the source code in the following directory:
  - **$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/smoothers**

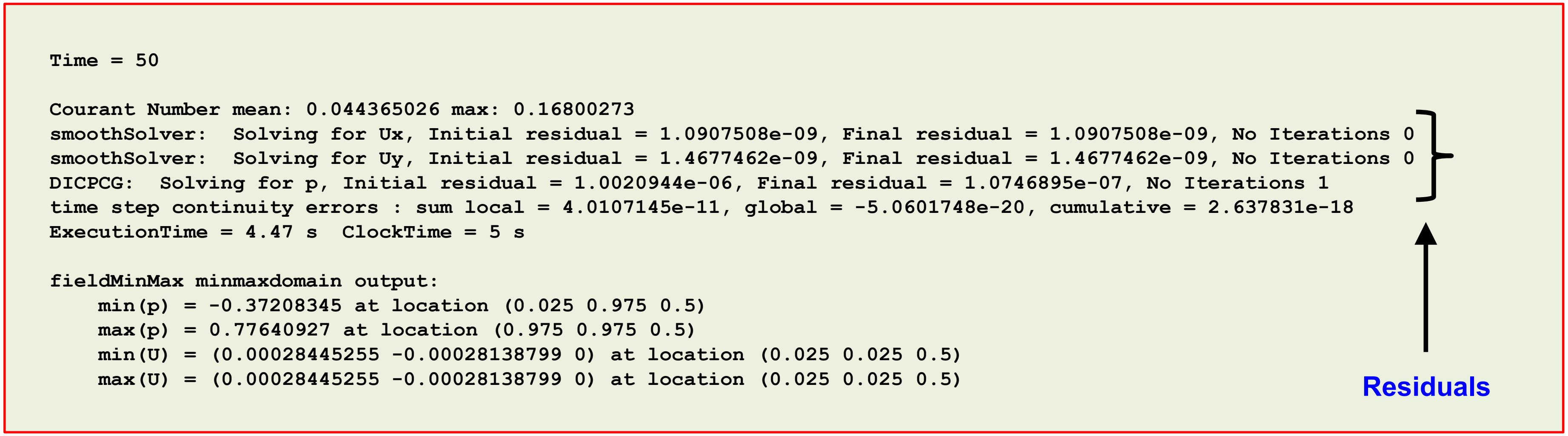
Linear solvers – General remarks

- As you can see, when it comes to linear solvers there are many options and combinations available in OpenFOAM®.
- When it comes to choosing the linear solver, there is no written theory.
- It is problem and hardware dependent (type of the mesh, physics involved, processor cache memory, network connectivity, partitioning method, and so on).
- Most of the times using the **GAMG** method (geometric-algebraic multi-grid), is the best choice for symmetric matrices (*e.g.*, pressure).
- The **GAMG** method should converge fast (less than 20 iterations). If it's taking more iterations, try to change the smoother.
- And if it is taking too long or it is unstable, use the **PCG** solver.
- When running with many cores (more than 1000), using the **PCG** might be a better choice.
- For asymmetric matrices, the **PBiCGStab** method with **DILU** preconditioner is a good choice.
- The **smoothSolver** solver with smoother **GaussSeidel**, also performs very well.
- If the **PBiCGStab** method with **DILU** preconditioner mysteriously crashed with an error related to the preconditioner, use the **smoothSolver** or change the preconditioner.
- But in general the **PBiCGStab** solver should be faster than the **smoothSolver** solver.
- Remember, asymmetric matrices are assembled from the velocity (**U**), and the transported quantities (**k**, **omega**, **epsilon**, **T**, and so on).
- Usually, computing the velocity and the transported quantities is inexpensive and fast, so it is a good idea to use a tight tolerance (1e-8) for these fields.
- The diagonal solver is used for back-substitution, for instance, when computing density using the equation of state (we know **p** and **T**).

Linear solvers in OpenFOAM®

Linear solvers – General remarks

- A few comments on the linear solvers residuals (we will talk about monitoring the residuals later on).
  - Residuals are not a direct indication that you are converging to the right solution.
  - The first time-steps the solution might not converge, this is acceptable.
  - Also, you might need to use a smaller time-step during the first iterations to maintain solver stability.
  - If the solution is not converging after a while, try to reduce the time-step size.



```
Time = 50

Courant Number mean: 0.044365026 max: 0.16800273
smoothSolver:  Solving for Ux, Initial residual = 1.0907508e-09, Final residual = 1.0907508e-09, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.4677462e-09, Final residual = 1.4677462e-09, No Iterations 0
DICPCG:  Solving for p, Initial residual = 1.0020944e-06, Final residual = 1.0746895e-07, No Iterations 1
time step continuity errors : sum local = 4.0107145e-11, global = -5.0601748e-20, cumulative = 2.637831e-18
ExecutionTime = 4.47 s  ClockTime = 5 s

fieldMinMax minmaxdomain output:
    min(p) = -0.37208345 at location (0.025 0.975 0.5)
    max(p) = 0.77640927 at location (0.975 0.975 0.5)
    min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
    max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
```

Linear solvers in OpenFOAM®

Linear solvers tolerances

- So how do we set the tolerances?
- The pressure equation is particularly important, so we should resolve it accurately. Solving the pressure equation is the expensive part of the whole iterative process.
- For the pressure equation (symmetric matrix), you can start the simulation with a **tolerance** equal to **1e-6** and **relTol** equal to **0.01**.
- And after a while, you change these values to **1e-6** and **0.0**, respectively.
- If the linear solver is taking too much time, you can change the convergence criterion to **1e-4** and **relTol** equal to **0.05**. You usually will do this during the first iterations.

Loose tolerance

```
p
{
          solver                  PCG;
          preconditioner          DIC;
          tolerance               1e-6;
          relTol                   0.01;
}
```

Tight tolerance

```
p
{
          solver                  PCG;
          preconditioner          DIC;
          tolerance               1e-6;
          relTol                   0.0;
}
```

Linear solvers in OpenFOAM®

Linear solvers tolerances

- For the velocity field (**U**) and the transported quantities (asymmetric matrices), you can use the following criterion.
- Solving for these variables is relatively inexpensive, so you can start right away with a tight tolerance.
- As a side note, the relative tolerance (**relTol**) is the difference between the initial residuals and the current final residuals.

Loose tolerance

```
U
{
        solver                      PBiCGStab;
        preconditioner              DILU;
        tolerance                   1e-8;
        relTol                      0.001;
 }
```

Tight tolerance

```
U
{
        solver                      PBiCGStab;
        preconditioner              DILU;
        tolerance                   1e-8;
        relTol                      0.0;
 }
```

Linear solvers in OpenFOAM®

Linear solvers tolerances

- It is also a good idea to set the minimum number of iterations (**minIter**), we recommend using a value of 3.
- If your solver is doing too many iterations, you can set the maximum number of iterations (**maxIter**).
- But be careful, if the solver reach the maximum number of iterations it will stop, we are talking about unconverged time-steps or outer-iterations.
- Setting the maximum number of iterations is especially useful during the first time-steps where the linear solver takes longer to converge.
- You can set **minIter** and **maxIter** in all symmetric and asymmetric linear solvers.

```
p
{
        solver                          PCG;
        preconditioner                  DIC;
        tolerance                       1e-6;
        relTol                          0.01;
        minIter                         3;
        maxIter                         100;
}
```

Linear solvers in OpenFOAM®

Linear solvers tolerances

- When you use the **PISO** or **PIMPLE** method with the **momentumPredictor** option (which is enabled by default), you also have the option to set the tolerance for the final pressure corrector step (**pFinal**).
- By proceeding in this way, you can put all the computational effort only in the last corrector step (**pFinal**).
- For all the intermediate corrector steps, you can use a more relaxed convergence criterion.
- For example, you can use the following solver and tolerance criterion for all the intermediate corrector steps (**p**), then in the final corrector step (**pFinal**) you tight the solver tolerance.

Loose tolerance for p

```
p
{
        solver                  PCG;
        preconditioner          DIC;
        tolerance               1e-4;
        relTol                   0.05;
}
```

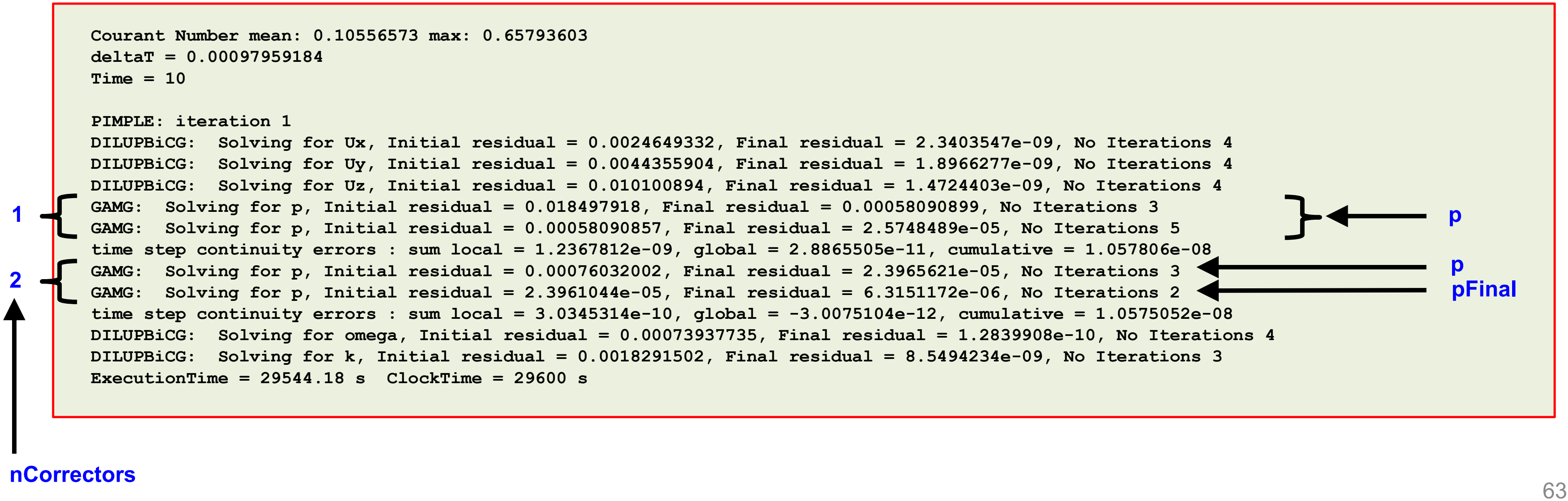
Tight tolerance for pFinal

```
pFinal
{
        solver                  PCG;
        preconditioner          DIC;
        tolerance               1e-6;
        relTol                   0.0;
}
```

Linear solvers in OpenFOAM®

Linear solvers tolerances

- When you use the **PISO** or **PIMPLE** method with the **momentumPredictor** option (which is enabled by default), you also have the option to set the tolerance for the final pressure corrector step (**pFinal**).
- By proceeding in this way, you can put all the computational effort only in the last corrector step (**pFinal** in this case).
- For all the intermediate corrector steps (**p**), you can use a more relaxed convergence criterion.
- If you proceed in this way, it is recommended to do at least 2 corrector steps (**nCorrectors**).

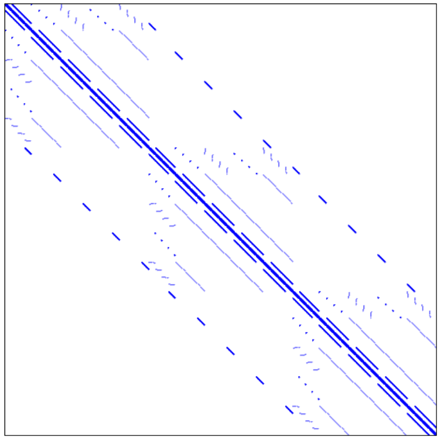


```
Courant Number mean: 0.10556573 max: 0.65793603
deltaT = 0.00097959184
Time = 10

PIMPLE: iteration 1
DILUPBiCG:  Solving for Ux, Initial residual = 0.0024649332, Final residual = 2.3403547e-09, No Iterations 4
DILUPBiCG:  Solving for Uy, Initial residual = 0.0044355904, Final residual = 1.8966277e-09, No Iterations 4
DILUPBiCG:  Solving for Uz, Initial residual = 0.010100894, Final residual = 1.4724403e-09, No Iterations 4
GAMG:  Solving for p, Initial residual = 0.018497918, Final residual = 0.00058090899, No Iterations 3
GAMG:  Solving for p, Initial residual = 0.00058090857, Final residual = 2.5748489e-05, No Iterations 5
time step continuity errors : sum local = 1.2367812e-09, global = 2.8865505e-11, cumulative = 1.057806e-08
GAMG:  Solving for p, Initial residual = 0.00076032002, Final residual = 2.3965621e-05, No Iterations 3
GAMG:  Solving for p, Initial residual = 2.3961044e-05, Final residual = 6.3151172e-06, No Iterations 2
time step continuity errors : sum local = 3.0345314e-10, global = -3.0075104e-12, cumulative = 1.0575052e-08
DILUPBiCG:  Solving for omega, Initial residual = 0.00073937735, Final residual = 1.2839908e-10, No Iterations 4
DILUPBiCG:  Solving for k, Initial residual = 0.0018291502, Final residual = 8.5494234e-09, No Iterations 3
ExecutionTime = 29544.18 s  ClockTime = 29600 s
```

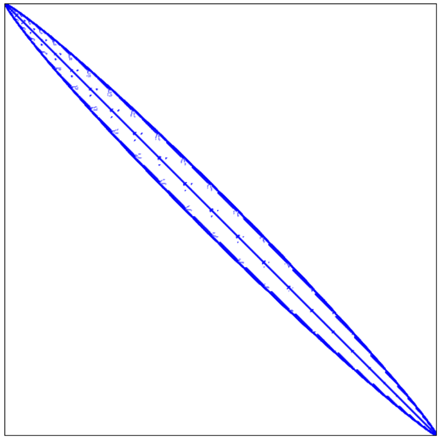
Linear solvers in OpenFOAM®

Linear solvers – Matrix reordering

- As we are solving a sparse matrix, the more diagonal the matrix is, the best the convergence rate will be.
- So it is highly advisable to use the utility `renumberMesh` before running the simulation.
  - `$> renumberMesh -overwrite`
- The utility `renumberMesh` can dramatically increase the speed of the linear solvers, specially during the first iterations.
- The idea behind reordering is to make the matrix more diagonally dominant, therefore, speeding up the iterative solver.



Matrix structure plot before reordering



Matrix structure plot after reordering

Note: this is the actual pressure matrix from an OpenFOAM® model case

Linear solvers in OpenFOAM®

On the multigrid solvers

- The development of multigrid solvers (**GAMG** in OpenFOAM®), together with the development of high-resolution TVD schemes and parallel computing, are among the most remarkable achievements of the history of CFD.
- Most of the time using the **GAMG** linear solver is fine.
- However, if you see that the **GAMG** linear solver is taking too long to converge or is converging in more than 100 iterations, it is better to use the **PCG** linear solver.
- Particularly, we have found that the **GAMG** linear solver in OpenFOAM® does not perform very well when you scale your computations to more than 500 processors.
- Also, we have found that for some multiphase cases the **PCG** method outperforms the **GAMG**.
- But again, this is problem and hardware dependent.
- As you can see, you need to always monitor your simulations (stick to the screen for a while.
- Otherwise, you might end-up using a solver that is performing poorly, and this translate in increased computational time and costs.

Linear solvers in OpenFOAM®

On the multigrid solvers tolerances

- If you go for the **GAMG** linear solver for symmetric matrices (*e.g.*, pressure), the following tolerances are acceptable for most of the cases.

Loose tolerance for p

```
p
{
    solver                     GAMG;
    tolerance                  1e-6;
    relTol                     0.01;
    smoother                   GaussSeidel;
    nPreSweeps                 0;
    nPostSweeps                2;
    cacheAgglomeration         on;
    agglomerator               faceAreaPair;
    nCellsInCoarsestLevel      100;
    mergeLevels                1;
    minIter                    3;
}
```

Tight tolerance for pFinal

```
pFinal
{
    solver                     GAMG;
    tolerance                  1e-6;
    relTol                     0;
    smoother                   GaussSeidel;
    nPreSweeps                 0;
    nPostSweeps                2;
    cacheAgglomeration         on;
    agglomerator               faceAreaPair;
    nCellsInCoarsestLevel      100;
    mergeLevels                1;
    minIter                    3;
}
```

NOTE:
The GAMG parameters are not optimized, that is up to you.
Most of the times is safe to use the proposed parameters.

Linear solvers in OpenFOAM®

Linear solvers tolerances – Steady simulations

- The previous tolerances are fine for unsteady solver.
- For extremely coupled problems you might need to have tighter tolerances.
- You can use the same tolerances for steady solvers. However, it is acceptable to use a looser criterion.
- For steady simulations using the **SIMPLE** method, you can set the convergence controls based on residuals of fields.
- The controls are specified in the **residualControls** sub-dictionary of the dictionary file *fvSolution*.

```
SIMPLE
{
        nNonOrthogonalCorrectors    2;

        residualControl
        {
                p    1e-4;
                U    1e-4;
        }
}
```

Residual control for every field variable you are solving

Linear solvers in OpenFOAM®

Linear solvers benchmarking of a model case

Case	Linear solver for P	Preconditioner or smoother	MR	Time	QOI
IC1	PCG	FDIC	NO	278	2.8265539
IC2	smoothSolver	symGaussSeidel	NO	2070	2.8271198
IC3	ICCG	GAMG	NO	255	2.8265538
IC4	GAMG	GaussSeidel	NO	1471	2.8265538
IC5	PCG	GAMG-GaussSeidel	NO	302	2.8265538
IC6	GAMG	GaussSeidel	YES	438	2.8265539
IC7	PCG	FDIC	YES	213	2.8265535
IC8	PCG	GAMG-GaussSeidel	YES	283	2.8265538
IC9	ICCG	GAMG	YES	261	2.8265538
IC10	PCG	DIC	NO	244	2.8265539

Solver used = `icoFoam` – Incompressible case
MR = matrix reordering (`renumberMesh`)
QOI = quantity of interest. In this case the maximum velocity at the outlet (m/s)
TIME = clock time (seconds)

Linear solvers in OpenFOAM®

Exercises

- Choose any tutorial or a case of your own and do a benchmarking of the linear solvers.
- Using your benchmarking case, conduct the following numerical experiments:
  - Find the optimal parameters for the **GAMG** solver.
  - Use different linear solvers for **p** and **pFinal** (symmetric matrices). Do you see any advantage?
  - Do a benchmarking of the different reordering methods available

    (Hint: look for the dictionary renumberMeshDict)
  - Compare the performance of the asymmetric solvers **PBiCG**, **PBiCGStab**, and **smoothSolver**. Do you see any significant difference between both solvers?

- Is it possible to switch between segregated and coupled linear solvers on-the-fly?
- In what files are located the controls of the **SIMPLE**, **PISO**, and **PIMPLE** methods?

  (Hint: for example, using grep look for the keyword nCorrectors in the directory src/finiteVolume)

Roadmap

1. ~~Finite Volume Method: A Crash Introduction~~
2. ~~On the CFL number~~
3. ~~Linear solvers in OpenFOAM®~~
4. **Pressure-Velocity coupling in OpenFOAM®**
5. Unsteady and steady simulations
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

Pressure-Velocity coupling in OpenFOAM®

- To solve the Navier-Stokes equations we need to use a solution approach able to deal with the nonlinearities of the governing equations and with the coupled set of equations.

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

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

$$\frac{\partial (\rho e_t)}{\partial t} + \nabla \cdot (\rho e_t \mathbf{u}) = \nabla \cdot q - \nabla \cdot (p \mathbf{u}) + \tau \mathbf{:} \nabla \mathbf{u},$$

$$+$$

Additional equations deriving from models, such as, volume fraction, chemical reactions, turbulence modeling, combustion, multi-species, etc.

Pressure-Velocity coupling in OpenFOAM®

- Many numerical methods exist to solve the Navier-Stokes equations, just to name a few:
    - Pressure-correction methods (Predictor-Corrector type).
        - SIMPLE, SIMPLEC, SIMPLER, PISO.
    - Projection methods.
        - Fractional step (operator splitting), MAC, SOLA.
    - Density-based methods and preconditioned solvers.
        - Riemann solvers, ROE, HLLC, AUSM+, ENO, WENO.
    - Artificial compressibility methods.
    - Artificial viscosity methods.

- The most widely used approaches for solving the NSE are:
    - Pressure-based approach (predictor-corrector).
    - Density-based approach.

Pressure-Velocity coupling in OpenFOAM®

- Historically speaking, the pressure-based approach was developed for low-speed incompressible flows, while the density-based approach was mainly developed for high-speed compressible flows.
- However, both methods have been extended and reformulated to solve and operate for a wide range of flow conditions beyond their original intent.
- In OpenFOAM®, you will find segregated pressure-based solvers.
- The segregated pressure-based solvers in OpenFOAM®, solve a modified pressure equation (pressure-Poisson equation).
- The following methods are available:
    - **SIMPLE** (Semi-Implicit Method for Pressure-Linked Equations)
    - **SIMPLEC** (SIMPLE Corrected/Consistent)
    - **PISO** (Pressure Implicit with Splitting Operators)
- You will find the solvers in the following directory:
    - `$WM_PROJECT_DIR/applications/solvers`
- Additionally, you will find something called **PIMPLE**, which is a hybrid between **SIMPLE** and **PISO** (known as iterative **PISO** outside OpenFOAM® jargon).
    - This formulation can give you more accuracy and stability when using very large time-steps or in pseudo-transient simulations.

Pressure-Velocity coupling in OpenFOAM®

- In OpenFOAM®, the **PISO** and **PIMPLE** methods are formulated for unsteady simulations.
- Whereas, the **SIMPLE** and **SIMPLEC** methods are formulated for steady simulations.
- If conserving time is not a priority, you can use the **PIMPLE** method in pseudo transient mode.
- The pseudo transient **PIMPLE** method is more stable than the **SIMPLE** method, but it has a higher computational cost.
- Also, the pseudo transient **PIMPLE** method tends to be faster than the fully transient **PIMPLE** when reaching steady states.
- Depending on the method and solver you are using, you will need to define a specific sub-dictionary in the dictionary file *fvSolution*.
- For instance, if you are using the **PISO** method, you will need to specify the **PISO** sub-dictionary.
- And depending on the method, each sub-dictionary will have different entries.

Pressure-Velocity coupling in OpenFOAM®

On the origins of the methods

- **SIMPLE**
    - S. V. Patankar and D. B. Spalding, “A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows”, Int. J. Heat Mass Transfer, 15, 1787-1806 (1972).

- **SIMPLE-C**
    - J. P. Van Doormaal and G. D. Raithby, “Enhancements of the SIMPLE method for predicting incompressible fluid flows”, Numerical Heat Transfer, 7, 147-163 (1984).

- **PISO**
    - R. I. Issa, “Solution of the implicitly discretized fluid flow equations by operator-splitting”, J. Comput. Phys., 62, 40-65 (1985).

- **PIMPLE**
    - Unknown origins outside OpenFOAM® ecosystem (we are referring to the semantics).
    - It is equivalent to **PISO** with outer iterations (iterative time-advancement of the solution).
    - Useful reference (besides **PISO** reference):
        - I. E. Barton, “Comparison of SIMPLE and PISO-type algorithms for transient flows, Int. J. Numerical methods in fluids, 26,459-483 (1998).
        - P. Oliveira and R. I. Issa, “An improved piso algorithm for the computation of buoyancy-driven flows”, Numerical Heat Transfer, 40, 473-493 (2001).

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- This sub-dictionary is located in the dictionary file *fvSolution*.
- It controls the options related to the **SIMPLE** pressure-velocity coupling method.
- The **SIMPLE** method only makes one correction.
- An additional correction to account for mesh non-orthogonality is available when using the **SIMPLE** method. The number of non-orthogonal correctors is specified by the **nNonOrthogonalCorrectors** keyword.
- The number of non-orthogonal correctors is chosen according to the mesh quality.
- For orthogonal meshes you can use 0 non-orthogonal corrections. However, it is strongly recommended to do at least 1 non-orthogonal correction (this helps stabilizing the solution).
- For non-orthogonal meshes, it is recommended to do at least 1 correction.

```
SIMPLE
{
    nNonOrthogonalCorrectors    1;
}
```

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- You can use the optional keyword **consistent** to enable or disable the **SIMPLEC** method.
- This option is disable by default.
- In the **SIMPLEC** method, the cost per iteration is marginally higher but the convergence rate is better, so the number of iterations is reduced.
- The **SIMPLEC** method relaxes the pressure in a consistent manner and additional relaxation of the pressure is not generally necessary (but it is recommended).
- In addition, convergence of the **p-U** system is better and still is reliable with less aggressive relaxation factors of the momentum equation.

```
SIMPLE
{
    consistent      yes;
    nNonOrthogonalCorrectors    1;
}
```

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- These are the typical (or industry standard) under-relaxation factors for the **SIMPLE** and **SIMPLEC** methods.
- Remember the under-relaxation factors are problem dependent.

SIMPLE

```
relaxationFactors
{
    fields
    {
        p        0.3;
    }
    equations
    {
        U        0.7;
        k        0.7;
        omega    0.7;
    }
}
```

SIMPLEC

```
relaxationFactors
{
    fields
    {
        p        1.0;
    }
    equations
    {
        p        1.0;
        U        0.9;
        k        0.9;
        omega    0.9;
    }
}
```

Usually there is no need to under-relax pressure; however, it is advisable.

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- If you are planning to use the **SIMPLEC** method, we recommend you to use under-relaxation factors that are little bit more smaller that the industry standard values.
- If during the simulation you still have some stability problems, try to reduce all the values to 0.5.
- Remember the under-relaxation factors are problem dependent.
- If you are having convergence problems, it is recommended to start the simulation with low values (about 0.3), and then increase the values slowly up to 0.7 or 0.9 (for faster convergence).

SIMPLEC

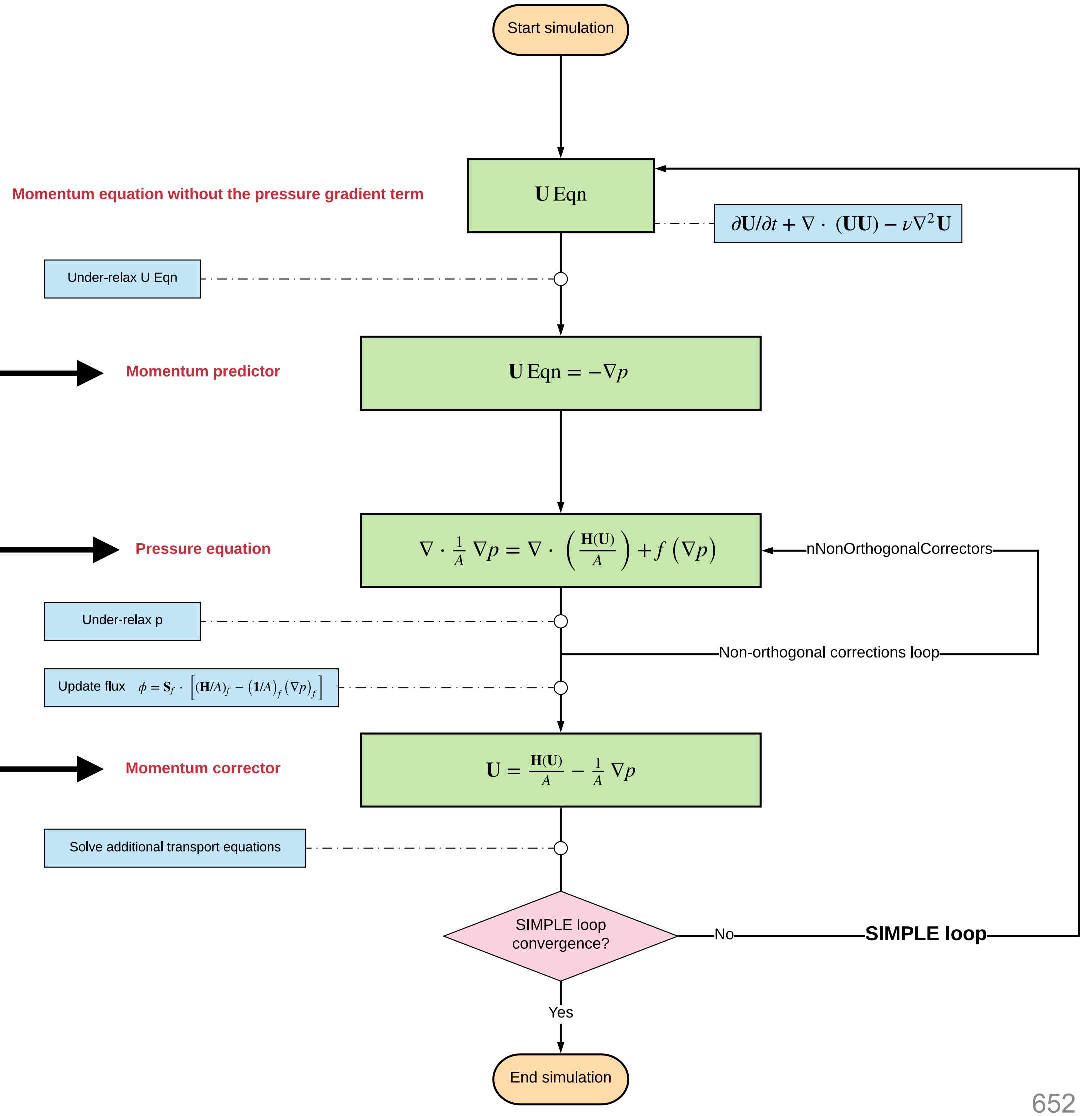
```
relaxationFactors
{
        fields
        {
                p               0.7;
        }
        equations
        {
                p               0.7;
                U               0.7;
                k               0.7;
                omega           0.7;
        }
}
```

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE loop in OpenFOAM®

```
fvVectorMatrix UEqn
(
    fvm::ddt(U) + fvm::div(phi, U) - fvm::laplacian(nu, U)
);


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


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


U = HbyA – rAU*fvc::grad(p);
```

This is an excerpt of the actual source code of the solver



Start simulation
U Eqn
Momentum equation without the pressure gradient term
$\partial \mathbf{U}/\partial t + \nabla \cdot (\mathbf{U}\mathbf{U}) - \nu \nabla^2 \mathbf{U}$
Under-relax U Eqn
Momentum predictor
$\mathbf{U}\,\mathrm{Eqn} = -\nabla p$
Pressure equation
$\nabla \cdot \frac{1}{A} \nabla p = \nabla \cdot \left(\frac{\mathbf{H}(\mathbf{U})}{A} \right) + f\left(\nabla p\right)$
nNonOrthogonalCorrectors
Under-relax p
Non-orthogonal corrections loop
Update flux $\phi = \mathbf{S}_f \cdot \left[(\mathbf{H}/A)_f - \left(\mathbf{1}/A\right)_f \left(\nabla p\right)_f \right]$
Momentum corrector
$\mathbf{U} = \frac{\mathbf{H}(\mathbf{U})}{A} - \frac{1}{A} \nabla p$
Solve additional transport equations
SIMPLE loop convergence?
No
SIMPLE loop
Yes
End simulation

Pressure-Velocity coupling in OpenFOAM®

The PISO sub-dictionary

- This sub-dictionary is located in the dictionary file *fvSolution*.
- It controls the options related to the **PISO** pressure-velocity coupling method.
- The **PISO** method requires at least one correction (**nCorrectors**).
- For good accuracy and stability (specially in unstructured meshes), it is recommended to use at least 2 **nCorrectors**.
- An additional correction to account for mesh non-orthogonality is available when using the **PISO** method. The number of non-orthogonal correctors is specified by the **nNonOrthogonalCorrectors** keyword.
- The number of non-orthogonal correctors is chosen according to the mesh quality.
- For orthogonal meshes you can use 0 non-orthogonal corrections. However, it is strongly recommended to do at least 1 non-orthogonal correction (this helps stabilizing the solution).
- For non-orthogonal meshes, it is recommended to do at least 1 correction.

```
PISO
{
    nCorrectors    2;
    nNonOrthogonalCorrectors    1;
}
```

Pressure-Velocity coupling in OpenFOAM®

The PISO sub-dictionary

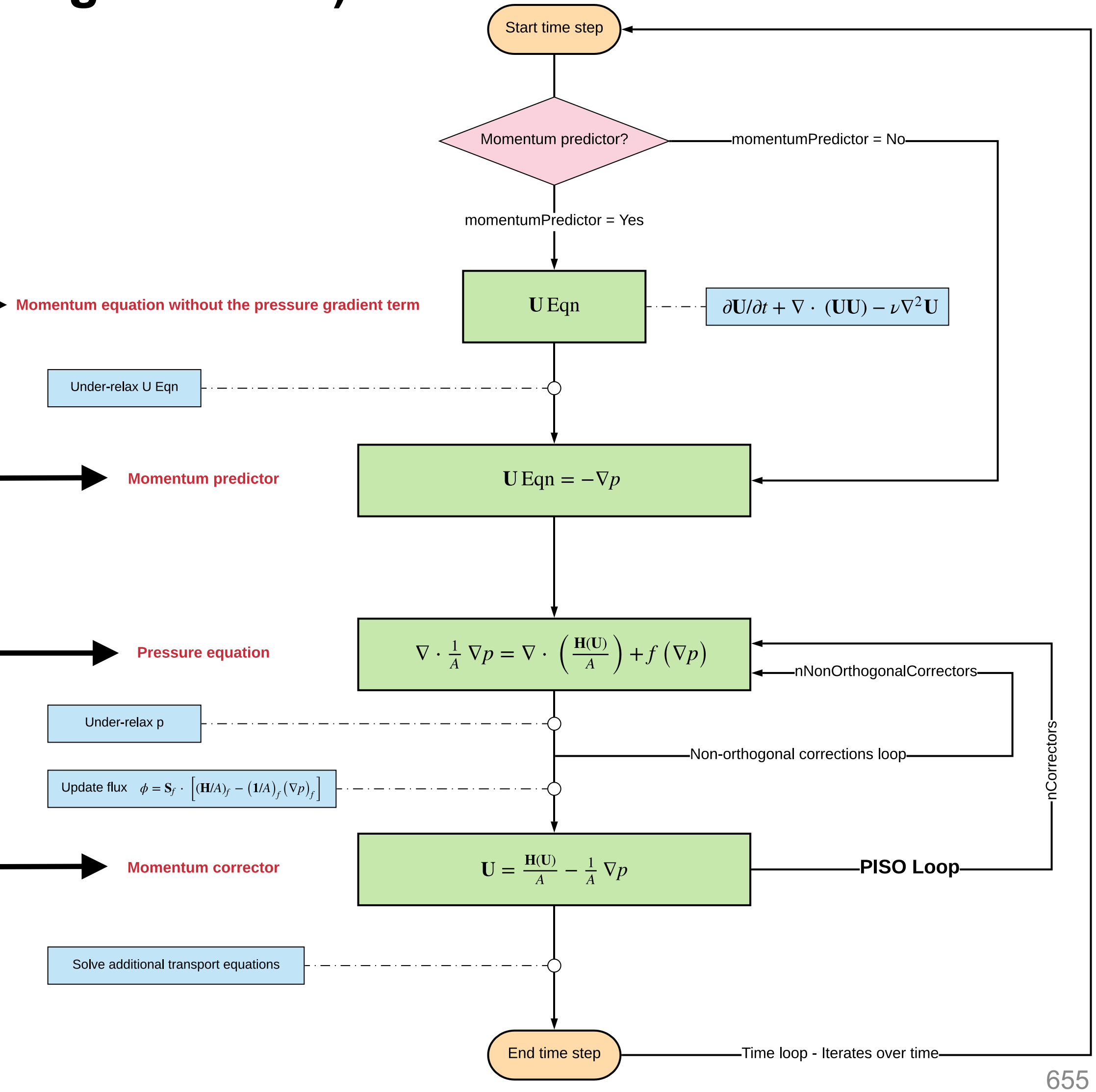
- You can use the optional keyword **momentumPredictor** to enable or disable the momentum predictor step.
- The momentum predictor helps in stabilizing the solution as we are computing better approximations for the velocity.
- It is clear that this will add an extra computational cost, which most of the times is negligible.
- In most of the solvers, this option is enabled by default.
- It is recommended to use this option for highly convective flows (high Reynolds number). If you are working with low Reynolds flow or creeping flows it is recommended to turn it off.
- Note that when you enable the option **momentumPredictor**, you will need to define the linear solvers for the variables **.*Final** (we are using regex notation).
- Also, if you want to use URF you will need to apply then to all field variables (including **.*Final**).

```
PISO
{
    momentumPredictor     yes;
    nCorrectors     2;
    nNonOrthogonalCorrectors     1;
}
```

Pressure-Velocity coupling in OpenFOAM®

The PISO loop in OpenFOAM®
(PISO with non-iterative marching – NITA –)

```
fvVectorMatrix UEqn
(
    fvm::ddt(U) + fvm::div(phi, U) - fvm::laplacian(nu, U)
);



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



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



U = HbyA – rAU*fvc::grad(p);
```


Start time step
Momentum predictor?
momentumPredictor = No
momentumPredictor = Yes
Momentum equation without the pressure gradient term
U Eqn
$\partial \mathbf{U}/\partial t + \nabla \cdot (\mathbf{U}\mathbf{U}) - \nu \nabla^2 \mathbf{U}$
Under-relax U Eqn
Momentum predictor
$\mathbf{U}\,\mathrm{Eqn} = -\nabla p$
Pressure equation
$\nabla \cdot \frac{1}{A} \nabla p = \nabla \cdot \left(\frac{\mathbf{H}(\mathbf{U})}{A} \right) + f\left(\nabla p\right)$
nNonOrthogonalCorrectors
Under-relax p
Non-orthogonal corrections loop
nCorrectors
Update flux $\phi = \mathbf{S}_f \cdot \left[(\mathbf{H}/A)_f - \left(\mathbf{1}/A\right)_f \left(\nabla p\right)_f \right]$
Momentum corrector
$\mathbf{U} = \frac{\mathbf{H}(\mathbf{U})}{A} - \frac{1}{A} \nabla p$
PISO Loop
Solve additional transport equations
End time step
Time loop - Iterates over time

This is an excerpt of the actual source code of the solver

Pressure-Velocity coupling in OpenFOAM®

The PIMPLE sub-dictionary

- This sub-dictionary is located in the dictionary file *fvSolution*. It controls the options related to the **PIMPLE** pressure-velocity coupling method.
- The **PIMPLE** method works very similar to the **PISO** method.
- In fact, setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- The keyword **nOuterCorrectors** controls a loop outside the **PISO** loop.
- To gain more stability, especially when using large time-steps or when dealing with complex physics (combustion, chemical reactions, shock waves, and so on), you can use more outer correctors (**nOuterCorrectors**).
    - Usually between 2 and 5 corrections for computational efficiency.
- Have in mind that increasing the number of **nOterCorrectors** will highly increase the computational cost.

```
PIMPLE
{
        momentumPredictor  yes;
        nOuterCorrectors        1;
        nCorrectors    2;
        nNonOrthogonalCorrectors    1;
}
```

The PIMPLE sub-dictionary

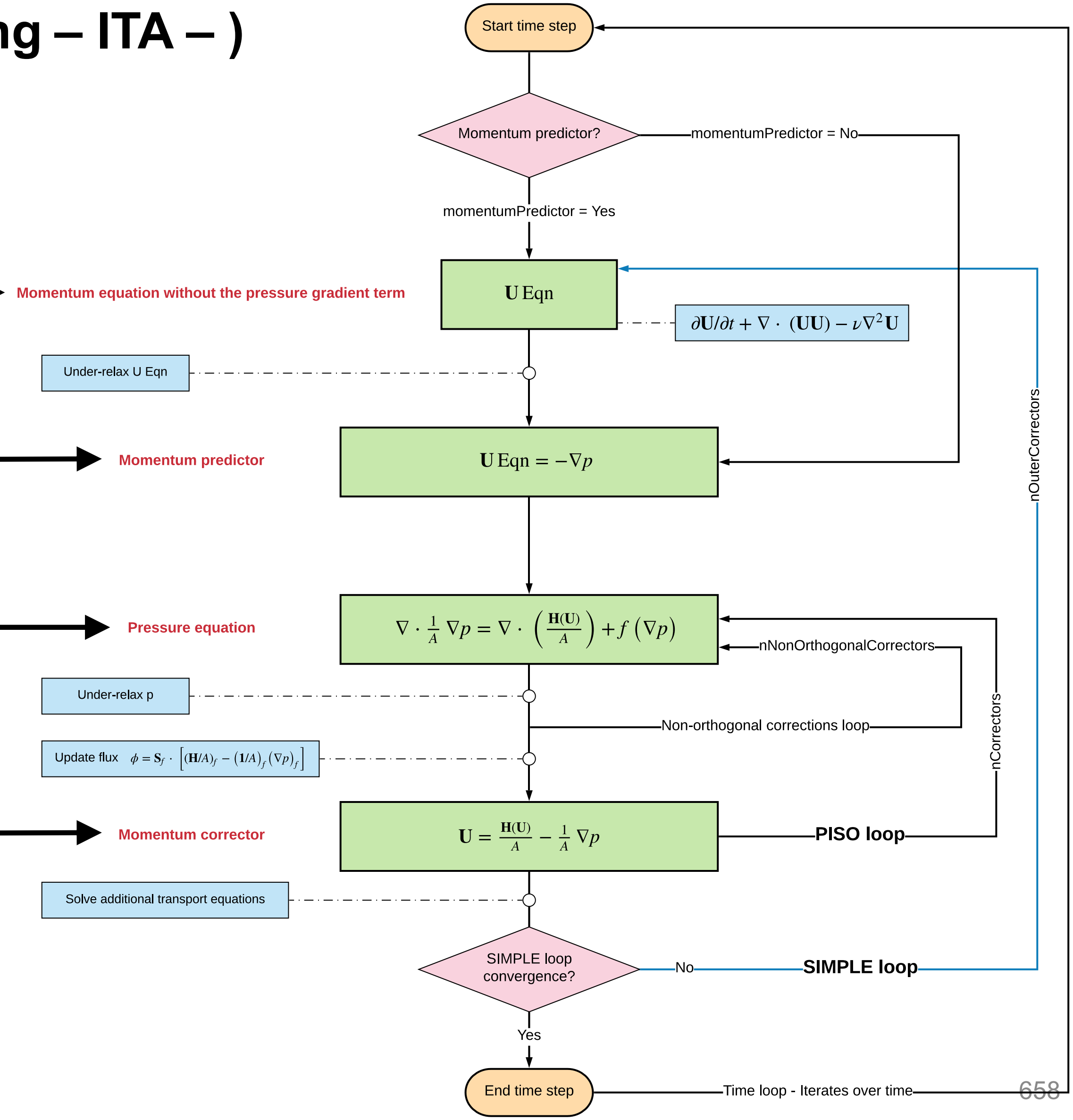
- You can use under-relaxation factors (URF) with the **PIMPLE** solvers.
- By using URF, you will gain more stability in time dependent solutions (as they control the amount of change of field variables within the time-step).
- However, if you use too low URF values, your solution might not be time-accurate anymore.
- You can use the same or larger URF values as those for steady simulation.
- Note that when you enable the option **momentumPredictor**, you will need to define the linear solvers for the variables **.*Final** (we are using regex notation).
- You can assign URF to all variables (including **.*Final**), to only the intermediate field variables (**U**, **p**, **k**, and so on), or to only the **.*Final** variables (**UFinal**, **pFinal**, **kFinal**, and so on).
- We recommend to use URF in all variables.

```
PIMPLE
{
        momentumPredictor  yes;
        nOuterCorrectors        1;
        nCorrectors     2;
        nNonOrthogonalCorrectors    1;
}
```

Pressure-Velocity coupling in OpenFOAM®

The PIMPLE loop in OpenFOAM®
(PISO with iterative marching – ITA –)

```
fvVectorMatrix UEqn
(
    fvm::ddt(U) + fvm::div(phi, U) - fvm::laplacian(nu, U)
);


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


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


U = HbyA – rAU*fvc::grad(p);
```

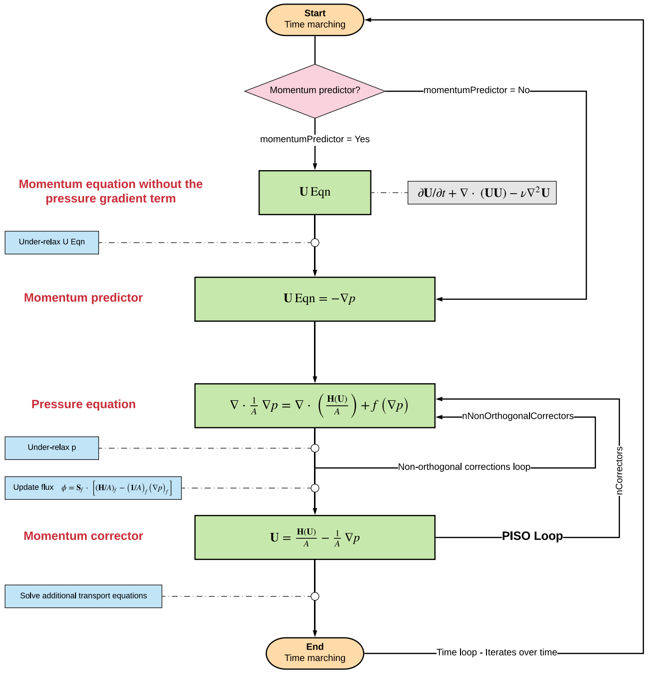

Momentum equation without the pressure gradient term
Momentum predictor
Pressure equation
Momentum corrector
Start time step
Momentum predictor?
momentumPredictor = No
momentumPredictor = Yes
U Eqn
$\partial \mathbf{U}/\partial t + \nabla \cdot (\mathbf{UU}) - \nu \nabla^2 \mathbf{U}$
Under-relax U Eqn
$\mathbf{U}\,\mathrm{Eqn} = -\nabla p$
$\nabla \cdot \frac{1}{A} \nabla p = \nabla \cdot \left(\frac{\mathbf{H(U)}}{A} \right) + f\left(\nabla p\right)$
nNonOrthogonalCorrectors
Under-relax p
Non-orthogonal corrections loop
Update flux $\phi = \mathbf{S}_f \cdot \left[(\mathbf{H}/A)_f - (1/A)_f (\nabla p)_f \right]$
$\mathbf{U} = \frac{\mathbf{H(U)}}{A} - \frac{1}{A} \nabla p$
PISO loop
nCorrectors
nOuterCorrectors
Solve additional transport equations
SIMPLE loop convergence?
No
SIMPLE loop
Yes
End time step
Time loop - Iterates over time

This is an excerpt of the actual source code of the solver

Pressure-Velocity coupling in OpenFOAM®

Comparison of PISO with non-iterative time-advancement (PISO-NITA) against PISO with Iterative time-advancement (PISO-ITA)

- The main difference between both methods is the outer loop present in the **PISO-ITA**.
- This outer loop gives more stability and allow the use of very large time-steps (CFL numbers).
- The recommended CFL number of the **PISO-NITA** is below 2 (for good accuracy and stability).



Start
Time marching

Momentum predictor?

momentumPredictor = No

momentumPredictor = Yes

Momentum equation without the pressure gradient term

U Eqn

$\partial \mathbf{U}/\partial t + \nabla \cdot (\mathbf{UU}) - \nu \nabla^2 \mathbf{U}$

Under-relax U Eqn

Momentum predictor

$\mathbf{U}\,\mathrm{Eqn} = -\nabla p$

Pressure equation

$\nabla \cdot \frac{1}{A} \nabla p = \nabla \cdot \left(\frac{\mathbf{H}(\mathbf{U})}{A}\right) + f(\nabla p)$

nNonOrthogonalCorrectors

Under-relax p

Non-orthogonal corrections loop

Update flux $\phi = \mathbf{S}_f \cdot \left[(\mathbf{H}/A)_f - (\mathbf{1}/A)_f (\nabla p)_f\right]$

nCorrectors

Momentum corrector

$\mathbf{U} = \frac{\mathbf{H}(\mathbf{U})}{A} - \frac{1}{A}\nabla p$

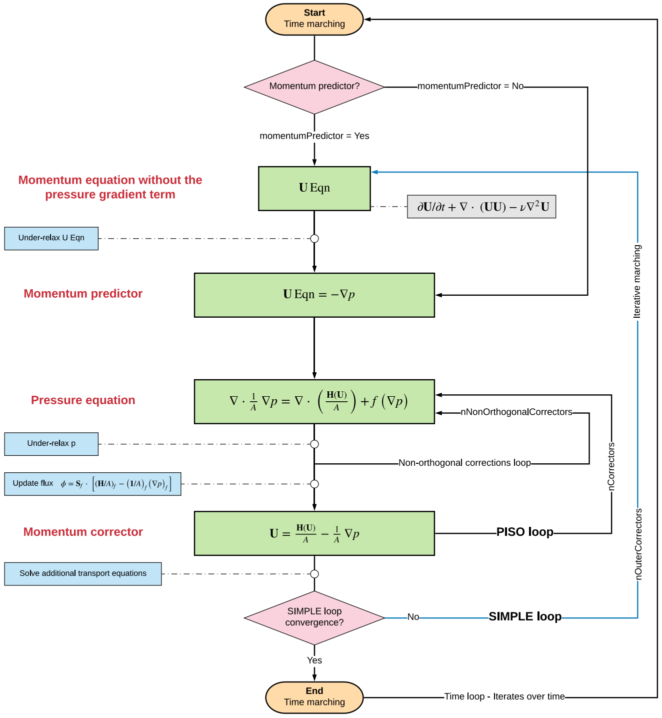
PISO Loop

Solve additional transport equations

End
Time marching

Time loop - Iterates over time

PISO-NITA



Start
Time marching

Momentum predictor?

momentumPredictor = No

momentumPredictor = Yes

Momentum equation without the pressure gradient term

U Eqn

$\partial \mathbf{U}/\partial t + \nabla \cdot (\mathbf{UU}) - \nu \nabla^2 \mathbf{U}$

Under-relax U Eqn

Iterative marching

Momentum predictor

$\mathbf{U}\,\mathrm{Eqn} = -\nabla p$

Pressure equation

$\nabla \cdot \frac{1}{A} \nabla p = \nabla \cdot \left(\frac{\mathbf{H}(\mathbf{U})}{A}\right) + f(\nabla p)$

nNonOrthogonalCorrectors

Under-relax p

Non-orthogonal corrections loop

Update flux $\phi = \mathbf{S}_f \cdot \left[(\mathbf{H}/A)_f - (\mathbf{1}/A)_f (\nabla p)_f\right]$

nCorrectors

Momentum corrector

$\mathbf{U} = \frac{\mathbf{H}(\mathbf{U})}{A} - \frac{1}{A}\nabla p$

PISO loop

nOuterCorrectors

Solve additional transport equations

SIMPLE loop
convergence?

No

SIMPLE loop

Yes

End
Time marching

Time loop - Iterates over time

PISO-ITA (PIMPLE in OpenFOAM®)

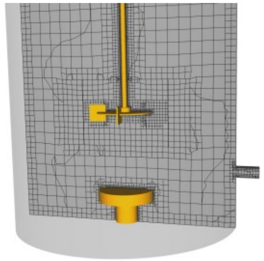
Roadmap

1. ~~Finite Volume Method: A Crash Introduction~~
2. ~~On the CFL number~~
3. ~~Linear solvers in OpenFOAM®~~
4. ~~Pressure-Velocity coupling in OpenFOAM®~~
5. **Unsteady and steady simulations**
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

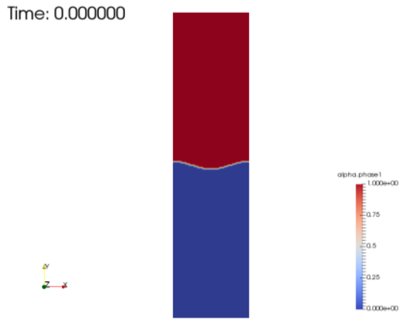
Unsteady and steady simulations

- Nearly all flows in nature and industrial applications are unsteady (also known as transient or time-dependent).
- Unsteadiness can be due to:
  - Instabilities.
  - Non-equilibrium initial conditions.
  - Time-dependent boundary conditions.
  - Source terms.
  - Chemical reactions and finite rate chemistry.
  - Phase change.
  - Moving or deforming bodies.
  - Turbulence.
  - Buoyancy and heat transfer.
  - Discontinuities.
  - Multiple phases.
  - Fluid structure interaction.
  - Combustion.
  - And much more.

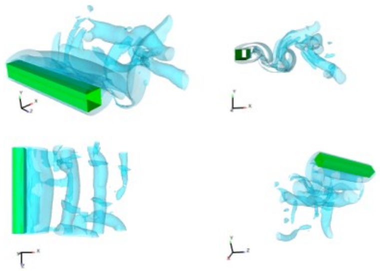
Sliding grids – Continuous stirred tank reactor
www.wolfdynamics.com/wiki/FVM_uns/ani5.gif



Multiphase flow
www.wolfdynamics.com/wiki/FVM_uns/ani3.gif



Turbulent flows - SRS
www.wolfdynamics.com/wiki/FVM_uns/ani4.gif



Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

- Select the time step. The time-step must be chosen in such a way that it resolves the time-dependent features and maintains solver stability.
- Select the temporal discretization scheme.
- Set the tolerance (absolute and/or relative) of the linear solvers.
- Monitor the CFL number.
- Monitor the stability and boundedness of the solution.
- Monitor a quantity of interest.
- And of course, you need to save the solution with a given frequency.
- Have in mind that unsteady simulations generate a lot of data.
- End time of the simulation?, it is up to you.
- In the *`controlDict`* dictionary you need to set runtime parameters and general instructions on how to run the case (such as time step and maximum CFL number). You also set the saving frequency.
- In the *`fvSchemes`* dictionary you need to set the temporal discretization scheme.
- In the *`fvSolution`* dictionary you need to set the linear solvers.
- Also, you will need to set the number of corrections of the velocity-pressure coupling method used (e.g. **PISO** or **PIMPLE**), this is done in the *`fvSolution`* dictionary.
- Additionally, you may set **functionObjects** in the *`controlDict`* dictionary. The **functionObjects** are used to do sampling, probing and co-processing while the simulation is running.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

$$\frac{\partial \phi}{\partial t}$$

```
ddtSchemes
{
        default        backward;
}

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

divSchemes
{
        default        none;
        div(phi,U)      Gauss linear;
}

laplacianSchemes
{
        default        Gauss linear orthogonal;
}

interpolationSchemes
{
        default        linear;
}

snGradSchemes
{
        default        orthogonal;
}
```

- The *fvSchemes* dictionary contains the information related to time discretization and spatial discretization schemes.
- In this generic case we are using the **backward** method for time discretization (**ddtSchemes**).
- This scheme is second order accurate but oscillatory.
- The parameters can be changed on-the-fly.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

```
startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         10;

deltaT          0.0001;

writeControl    runTime;

writeInterval   0.1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  8;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable       yes;

adjustTimeStep          yes;
maxCo                   2.0;
maxDeltaT               0.001;
```

- The `controlDict` dictionary contains runtime simulation controls, such as, start time, end time, time step, saving frequency and so on.
- Most of the entries are self-explanatory.
- This generic case starts from time 0 (**startTime**), and it will run up to 10 seconds (**endTime**).
- It will write the solution every 0.1 seconds (**writeInterval**) of simulation time (**runTime**).
- The time step of the simulation is 0.0001 seconds (**deltaT**).
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**) with a precision of 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on (**yes**), we can modify all these entries while we are running the simulation.
- To reduce parsing time and file size, it is recommended to use binary format to write the solution.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

```
startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         10;

deltaT          0.0001;

writeControl    runTime;

writeInterval   0.1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  8;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable       yes;

adjustTimeStep          yes;
maxCo                   2.0;
maxDeltaT               0.001;
```

- In this generic case, the solver supports adjustable time-step (**adjustTimeStep**).
- The option **adjustTimeStep** will automatically adjust the time step to achieve the maximum desired courant number (**maxCo**) or time-step size (**maxDeltaT**).
- When any of these conditions is reached, the solver will stop scaling the time-step size.
- Remember, the first time-step of the simulation is done using the value defined with the keyword **deltaT** and then it is automatically scaled (up or down), to achieve the desired maximum values (**maxCo** and **maxDeltaT**).
- It is recommended to start the simulation with a low time-step in order to let the solver scale-up the time-step size.
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.
- If you are planning to use large time steps (CFL much higher than 1), it is recommended to do at least 3 correctors steps (**nCorrectors**) in **PISO/PIMPLE** loop, and at least 2 outer correctors in the **PIMPLE** loop.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

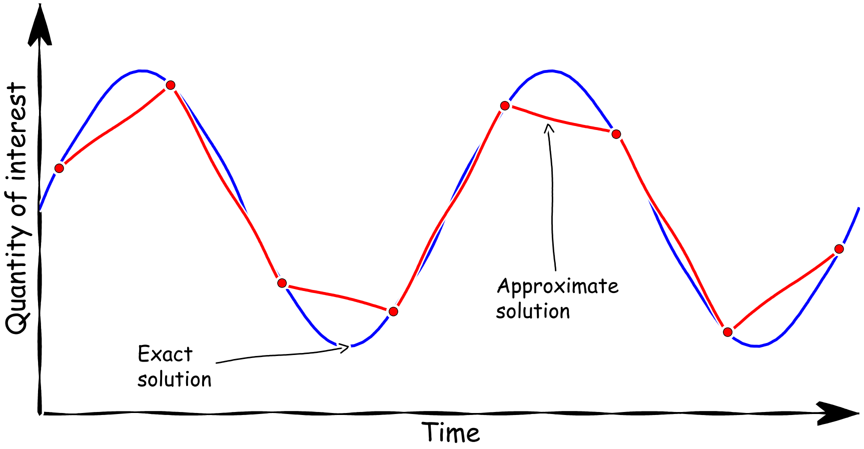
```
solvers
{
    p
    {
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
    }

    pFinal
    {
        $p;
        relTol    0;
    }

    “U.*”
    {
        solver              smoothSolver;
        smoother            symGaussSeidel;
        tolerance           1e-08;
        relTol              0;
    }
}
PIMPLE
{
    nOuterCorrectors 1;
    nCorrectors     2;
    nNonOrthogonalCorrectors     1;
}
```

- The *fvSolution* dictionary contains the instructions of how to solve each discretized linear equation system.
- As for the *controlDict and fvSchemes* dictionaries, the parameters can be changed on-the-fly.
- To set these parameters, follow the guidelines given in the previous section.
- Depending on the solver you are using, you will need to define the sub-dictionary **PISO** or **PIMPLE**.
- Setting the keyword **nOuterCorrectors** to 1 in **PIMPLE** solvers is equivalent to running using the **PISO** method.
- To gain more stability, especially when using large time-steps, you can use more outer correctors (**nOuterCorrectors**).
- If you are using large time steps (CFL much higher than 1), it is recommended to do at least 3 correctors steps (**nCorrectors**) in **PISO/PIMPLE** loop.
- Remember, in both **PISO** and **PIMPLE** method you need to do at least one correction (**nCorrectors**).
- Adding corrections increase the computational cost (**nOuterCorrectors** and **nCorrectors**).

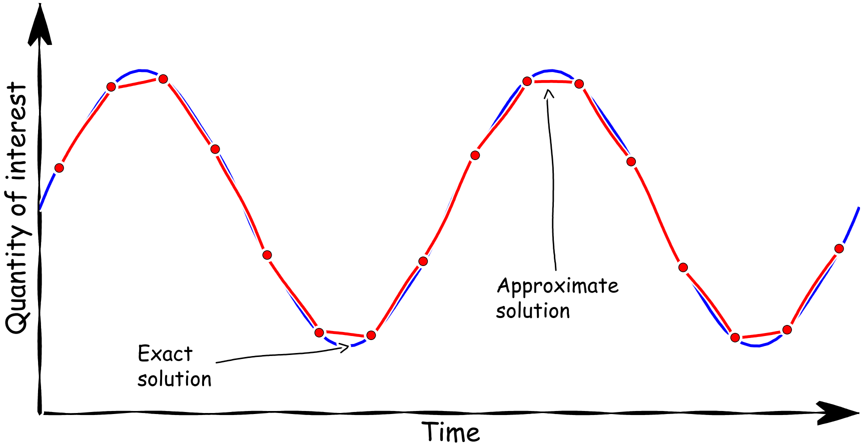
Unsteady and steady simulations

How to choose the time-step in unsteady simulations and monitor the solution

- Remember, when running unsteady simulations the time-step must be chosen in such a way that it resolves the time-dependent features and maintains solver stability.



When you use large time steps you do not resolve well the physics

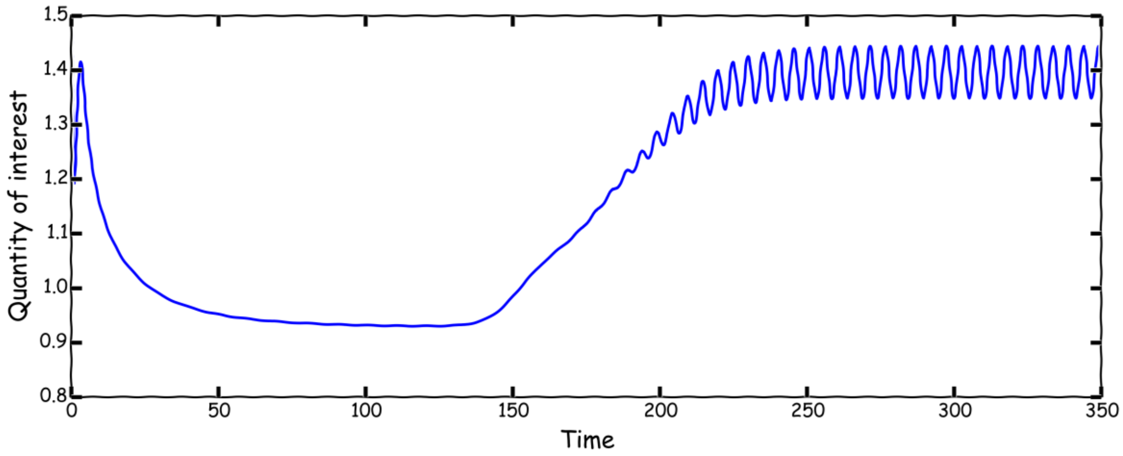


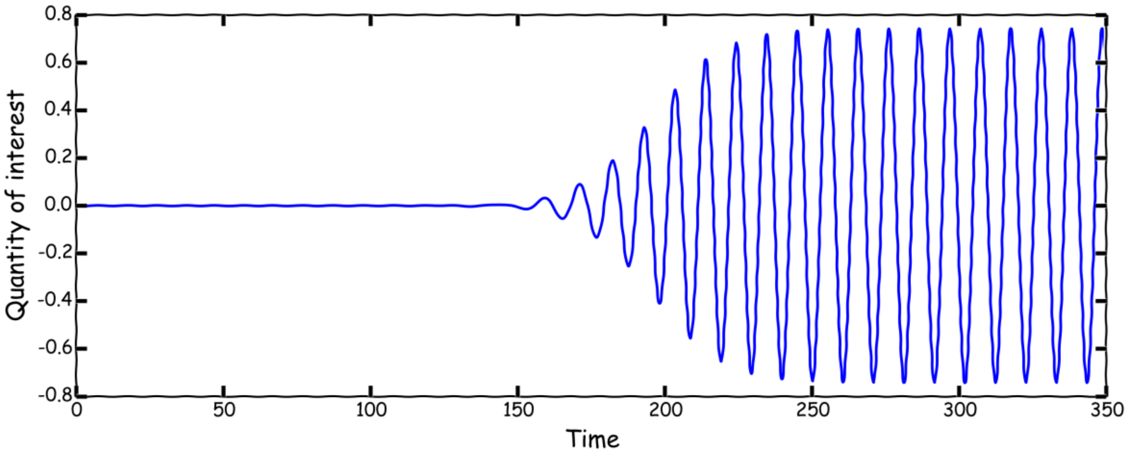
By using a smaller time step you resolve better the physics and you gain stability

Unsteady and steady simulations

Monitoring unsteady simulations

- When running unsteady simulations, it is highly advisable to monitor a quantity of interest.
- The quantity of interest can fluctuate in time, this is an indication of unsteadiness.





Unsteady and steady simulations

What about steady simulations?

- First of all, steady simulations are a big simplification of reality.
- Steady simulations is a trick used by CFDers to get fast outcomes with results that might be even more questionable.
- Remember, most of the flows you will encounter are unsteady so be careful of this hypothesis.
- In steady simulations, we made two assumptions:
  - We ignore unsteady fluctuations. That is, we neglect the time derivative in the governing equations.
  - We perform time averaging when dealing with stationary turbulence (RANS modeling)
- The advantage of steady simulations is that they require low computational resources, give fast outputs, and are easier to post-process and analyze.
- To do so, you need to use the appropriate solver and use the right discretization scheme.
- As you are not solving the time derivative, you do not need to set the time step. However, you need to tell OpenFOAM® how many iterations you would like to run.
- You can also set the residual controls (**residualControl**), in the `fvSolution` dictionary file. You set the **residualControl** in the **SIMPLE** sub-dictionary.
- If you do not set the residual controls, OpenFOAM® will run until reaching the maximum number of iterations (**endTime**).

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- In the *controlDict* dictionary you need to set runtime parameters and general instructions on how to run the case (such as the number of iterations to run).
- Remember to set also the saving frequency.
- In the *fvSchemes* dictionary you need to set the time discretization scheme, for steady simulations it must be **steadyState**.
- In the *fvSolution* dictionary you need to set the linear solvers, under-relaxation factors, and residual controls.
- Also, you will need to set the number of corrections of the velocity-pressure coupling method used (e.g. **SIMPLE** or **SIMPLEC**), this is done in the *fvSolution* dictionary.
- Additionally, you may set **functionObjects** in the *controlDict* dictionary.
- The **functionObjects** are used to do sampling, probing and co-processing while the simulation is running.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- The under-relaxation factors (URF) control the change of the variable ϕ.

$$\phi_P^n = \phi_P^{n-1} + \alpha(\phi_P^{n^*} - \phi_P^{n-1})$$

- Under-relaxation is a feature typical of steady solvers using the **SIMPLE** family of methods.
- These are the URF commonly used with **SIMPLE** and **SIMPLEC** (industry standard),

SIMPLE		SIMPLEC	
p	**0.3;**	**p**	**1;**
U	**0.7;**	**U**	**0.9;**
k	**0.7;**	**k**	**0.9;**
omega	**0.7;**	**omega**	**0.9;**

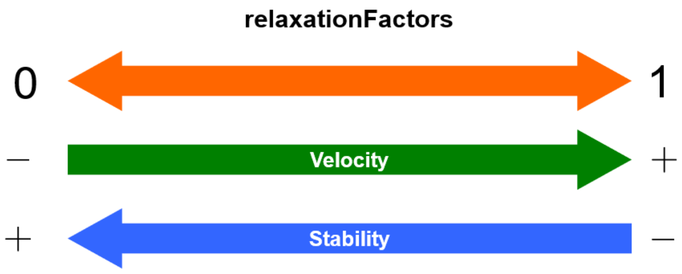
Pressure Usually does not require under-relaxing

- According to the physics involved you will need to add more under-relaxation factors.
- Finding the right URF involved experience and some trial and error.
- Selecting the URF it is kind of equivalent to selecting the right time step.
- Many times, steady simulations diverge because of wrongly chosen URF.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- The URF are bounded between 0 and 1.
- If you set the URF close to one you increase the convergence rate but loose solution stability.
- On the other hand, if you set the URF close to zero you gain stability but reduce convergence rate.



- An optimum choice of under-relaxation factors is one that is small enough to ensure stable computation but large enough to move the iterative process forward quickly.
- Under-relaxation can be implicit (equation in OpenFOAM) or explicit (field in OpenFOAM).

$$\frac{a_P \phi}{\alpha} = \sum_N a_N \phi_N + b + \frac{1-\alpha}{\alpha} a_P \phi_{n-1}$$

Implicit URF

$$\phi = \phi_{n-1} + \alpha \Delta \phi$$

Explicit URF

- You can relate URF to the CFL number as follows,

$$CFL = \frac{\alpha}{1-\alpha} \qquad \alpha = \frac{CFL}{1+CFL}$$

- A small CFL number is equivalent to small URF.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

```
ddtSchemes
{
        default         steadyState;   ←
}

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

divSchemes
{
        default         none;
        div(phi,U)     bounded Gauss linear;
}

laplacianSchemes
{
        default         Gauss linear orthogonal;
}

interpolationSchemes
{
        default         linear;
}

snGradSchemes
{
        default         orthogonal;
}
```

$$\cancel{\frac{\partial \phi}{\partial t}}$$

- The *fvSchemes* dictionary contains the information related to time discretization and spatial discretization schemes.
- In this generic case and as we are interested in using a steady solver, we are using the **steadyState** method for time discretization (**ddtSchemes**).
- It is not a good idea to switch between steady and unsteady schemes on-the-fly.
- For steady state cases, the bounded form can be applied to the divSchemes, in this case, div(phi,U) **bounded** Gauss linear**.**
- This adds a linearized, implicit source contribution to the transport equation of the form,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) - \boxed{(\nabla \cdot \mathbf{u})\mathbf{u}} = \nabla \cdot (\Gamma \nabla \mathbf{u}) + S$$

- This term removes a component proportional to the continuity error. This acts as a convergence aid to tend towards a bounded solution as the calculation proceeds.
- At convergence, this term becomes zero and does not contribute to the final solution.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

```
startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         10000;

deltaT          1;

writeControl    runTime;

writeInterval   100;

purgeWrite      10;

writeFormat     ascii;

writePrecision  8;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable       yes;
```

- The `controlDict` dictionary contains runtime simulation controls, such as, start time, end time, time step, saving frequency and so on.
- Most of the entries are self-explanatory.
- As we are doing a steady simulation, let us talk about iterations instead of time (seconds).
- This generic case starts from iteration 0 (**startTime**), and it will run up to 10000 iterations (**endTime**).
- It will write the solution every 100 iterations (**writeInterval**) of simulation time (**runTime**).
- It will advance the solution one iteration at a time (**deltaT**).
- It will keep the last 10 saved solutions (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**) with a precision of 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on (**true**), we can modify all these entries while we are running the simulation.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

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

   U
   {
      solver            smoothSolver;
      smoother          symGaussSeidel;
      tolerance         1e-08;
      relTol            0;
   }
}

SIMPLE
{
   nNonOrthogonalCorrectors    2;

   residualControl
   {
      p   1e-4;
      U    1e-4;
   }
}
```

- The *fvSolution* dictionary contains the instructions of how to solve each discretized linear equation system.
- As for the *controlDict* and *fvSchemes* dictionaries, the parameters can be changed on-the-fly.
- To set these parameters, follow the guidelines given in the previous section.
- Increasing the number of **nNonOrthogonalCorrectors** corrections will add more stability but at a higher computational cost.
- Remember, **nNonOrthogonalCorrectors** is used to improve the gradient computation due to mesh quality.
- The **SIMPLE** sub-dictionary also contains convergence controls based on residuals of fields. The controls are specified in the **residualControls** sub-dictionary.
- The user needs to specify a tolerance for one or more solved fields and when the residual for every field falls below the corresponding residual, the simulation terminates.
- If you do not set the **residualControls**, the solver will iterate until reaching the maximum number of iterations set in the *controlDict* dictionary.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

```
relaxationFactors
{
    fields
    {
        p   0.3;
    }
    equations
    {
        U    0.7;
    }
}
```

- The *fvSolution* dictionary also contains the **relaxationFactors** sub-dictionary.
- The **relaxationFactors** sub-dictionary which controls under-relaxation, is a technique used for improving stability when using steady solvers.
- Under-relaxation works by limiting the amount which a variable changes from one iteration to the next, either by modifying the solution matrix and source prior to solving for a field (**equations** keyword) or by modifying the field directly (**fields** keyword).
- Under-relaxing the equations is also known as implicit under-relaxation.
- Whereas, under-relaxing the fields is also known as explicit under-relaxation.
- An optimum choice of under-relaxation factors is one that is small enough to ensure stable computation but large enough to move the iterative process forward quickly.
- In this case we are using the industry standard URF.
- Remember, URF are problem dependent.
- If you do not define URF, the solver will not under-relax.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- To enable the consistent formulation of the **SIMPLE** method, you need to add the following keywork to the **SIMPLE** sub-dictionary,

```
SIMPLE
{
          consistent    yes;
          nNonOrthogonalCorrectors    3;
}
```

Enabled/disabled consistent formulation of the SIMPLE loop

- The following URF are recommended,

SIMPLE

```
relaxationFactors
{
          fields
          {
                    p                   0.3;
          }
          equations
          {
                    U                   0.7;
                    k                   0.6;
                    omega               0.6;
          }
}
```

SIMPLEC

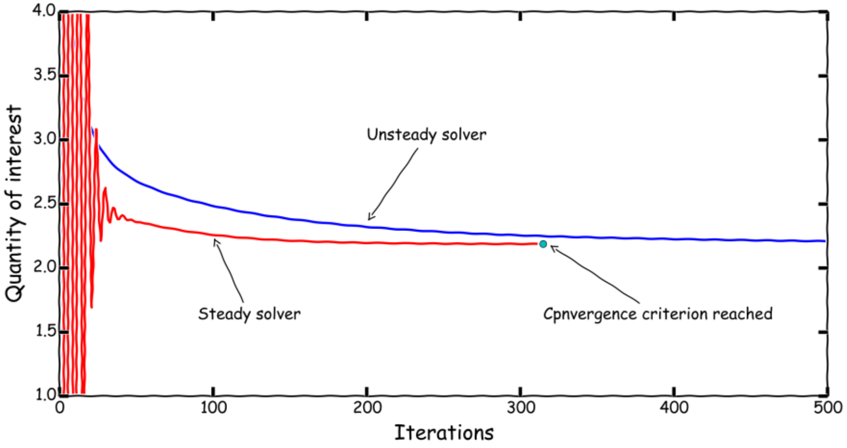
```
relaxationFactors
{
          fields
          {
                    p                   0.7;
          }
          equations
          {
                    U                   0.7;
                    k                   0.7;
                    omega               0.7;
          }
}
```

Unsteady and steady simulations

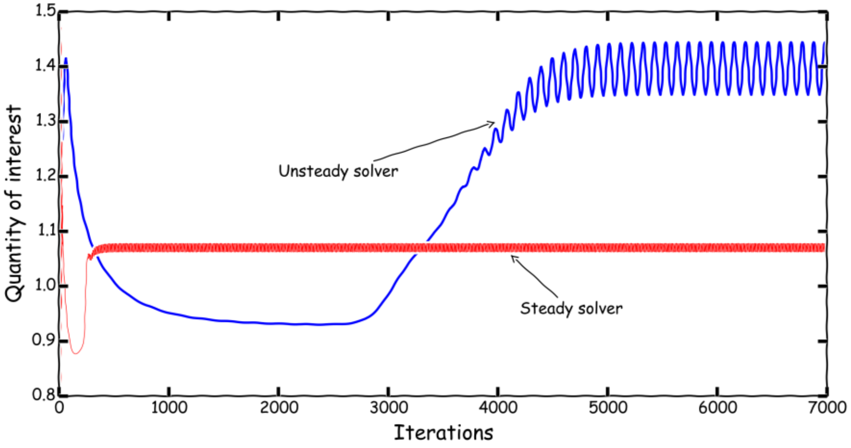
Steady simulations vs. Unsteady simulations

- Steady simulations require less computational power than unsteady simulations.
- They are also much faster than unsteady simulations.
- But sometimes they do not converge to the right solution.
- They are easier to post-process and analyze (you just need to take a look at the last saved solution).
- You can use the solution of an unconverged steady simulation as initial conditions for an unsteady simulation.
- Remember, steady simulations are not time accurate. Therefore is not a good idea to compute a dominant frequency using steady simulations, *e.g.*, vortex shedding frequency.

Steady solution QOI



unsteady solution QOI

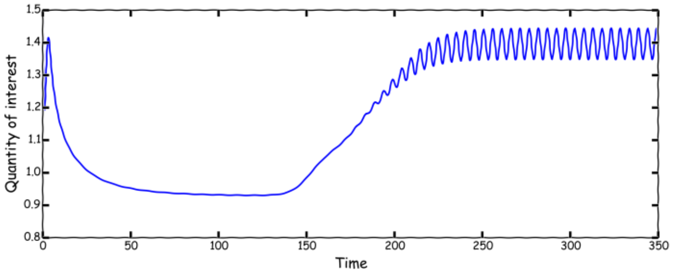


Roadmap

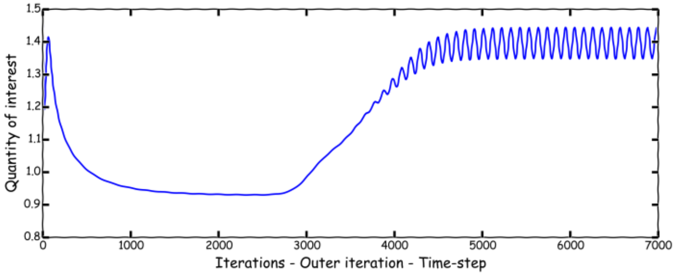
1. ~~Finite Volume Method: A Crash Introduction~~
2. ~~On the CFL number~~
3. ~~Linear solvers in OpenFOAM®~~
4. ~~Pressure-Velocity coupling in OpenFOAM®~~
5. ~~Unsteady and steady simulations~~
6. **Understanding residuals**
7. Boundary and initial conditions
8. Numerical playground

Understanding residuals

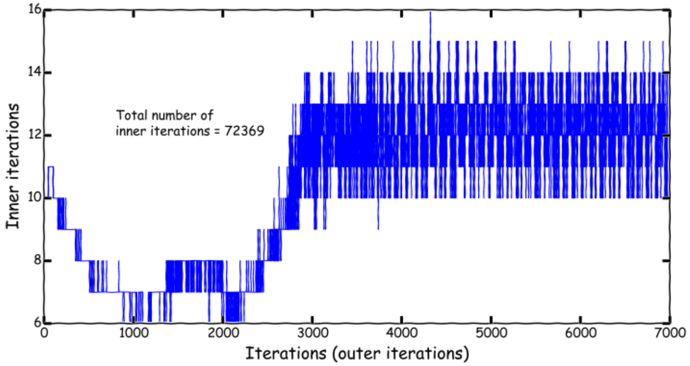
- Before talking about residuals, let us clarify something.
- When we talk about iterations in unsteady simulations, we are talking about the time-step or outer-iterations.



1. To arrive to this physical time of the monitored QOI



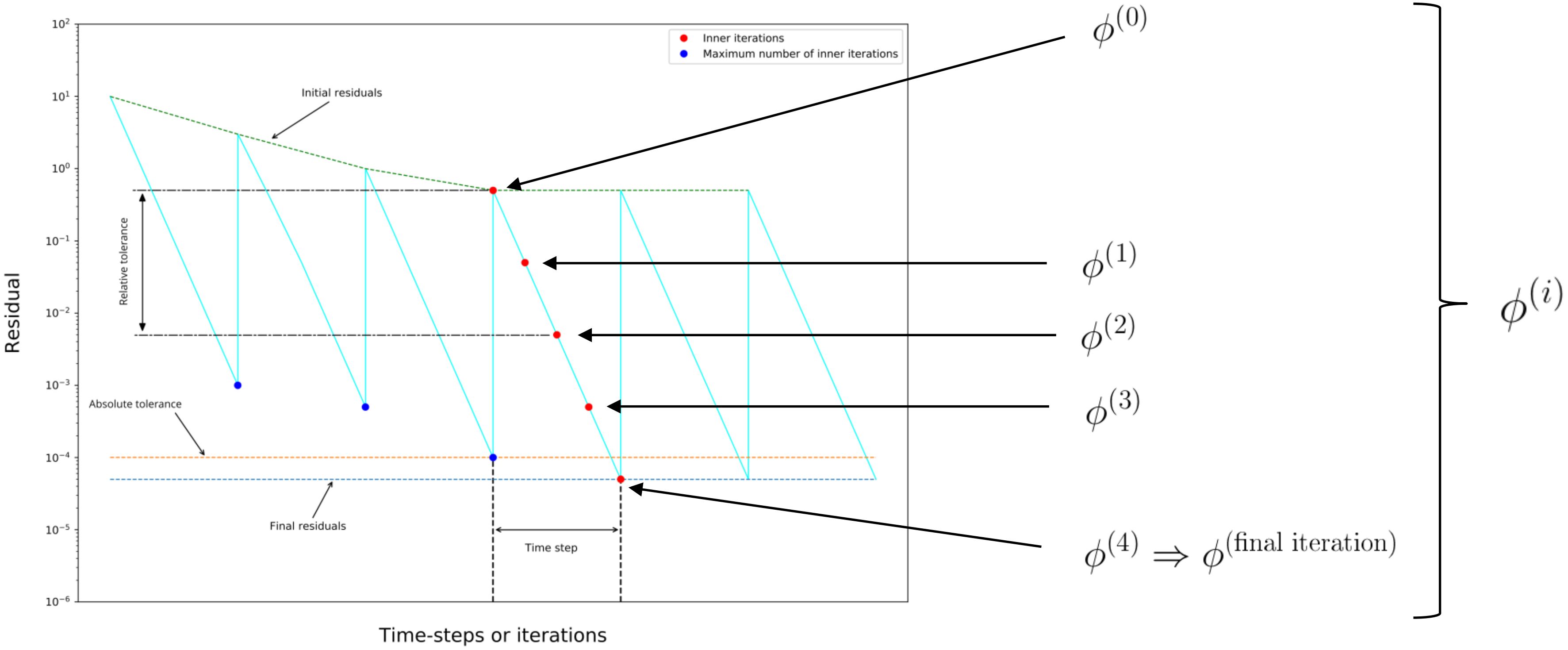
2. We iterate this many times



3. And we iterate inside each time-step (or outer-iteration), until reaching the linear solver tolerance or maximum number of iterations.

Understanding residuals

- To get a better idea of how iterative methods work, and what are initial residuals and final residuals, let us take another look at a residual plot.

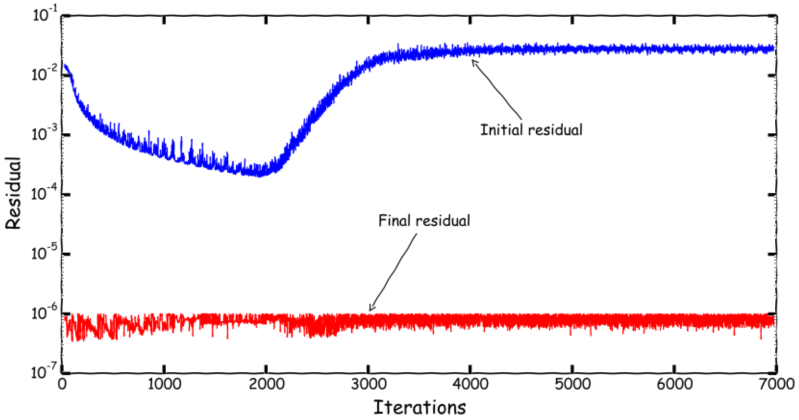


- $\phi^{(0)}$ is the initial guess used to start the iterative solver.
- You can use any value at iteration 0, but usually is a good choice to take the previous solution vector.
- If the following condition is fulfilled $\left|\mathbf{A}\phi^{i} - \mathbf{b}\right| \leq |\mathbf{r}|$ (where **r** is the convergence criterion or tolerance), the linear solver will stop iterating and will advance to the next time-step.
- By working in an iterative way, every single iteration $\phi^{(i)}$ is a better approximation of the previous iteration $\phi^{(i-1)}$.
- Sometimes the linear solver might stop before reaching the predefined convergence criterion because it has reached the maximum number of iterations, you should be careful of this because we are talking about unconverged iterations.

Understanding residuals

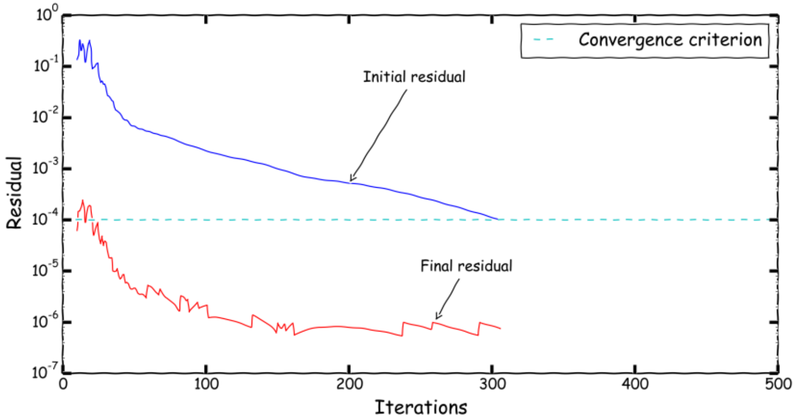
- This is a typical residual plot for an unsteady simulation.
- Ideally, the solution should converge at every time-step (final residuals tolerance).
- If the solution is not converging, that is, the residuals are not reaching the predefined final residual tolerance, try to reduce the time-step size.
- The first time-steps the solution might not converge, this is acceptable.
- Also, you might need to use a smaller time-step during the first iterations to maintain solver stability.
- You can also increase the number of maximum inner iterations.
- If the initial residuals fall bellow the convergence criterion, you might say that you have arrived to a steady solution (the exception rather than the rule).

Unsteady solution residuals



- This is a typical residual plot for a steady simulation.
- In this case, the initial residuals are falling below the convergence criterion (monotonic convergence), hence we have reached a steady-state.
- In the solver does not reach the convergence criteria or the residuals get stalled, it does not mean that the solution is diverging, it is just an indication of unsteadiness and it might be better to run using an unsteady solver.
- In comparison to unsteady solvers, steady solvers require less iterations to arrive to a converge solution, if they arrive.

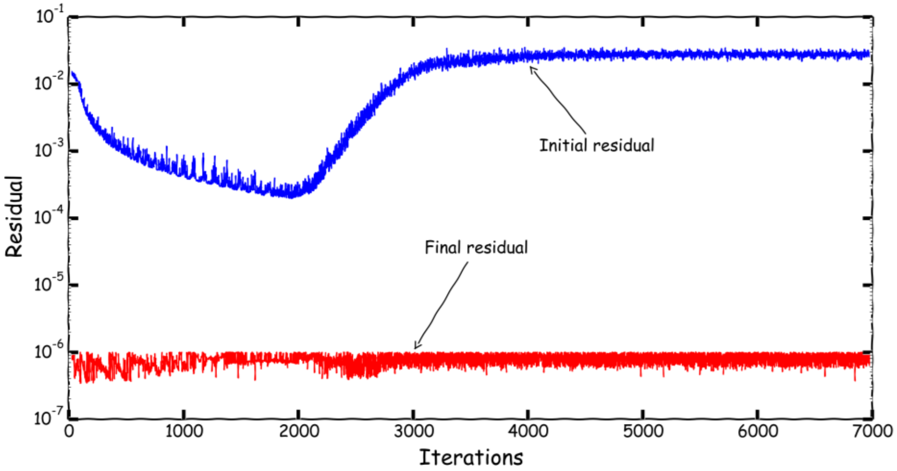
Steady solution residuals



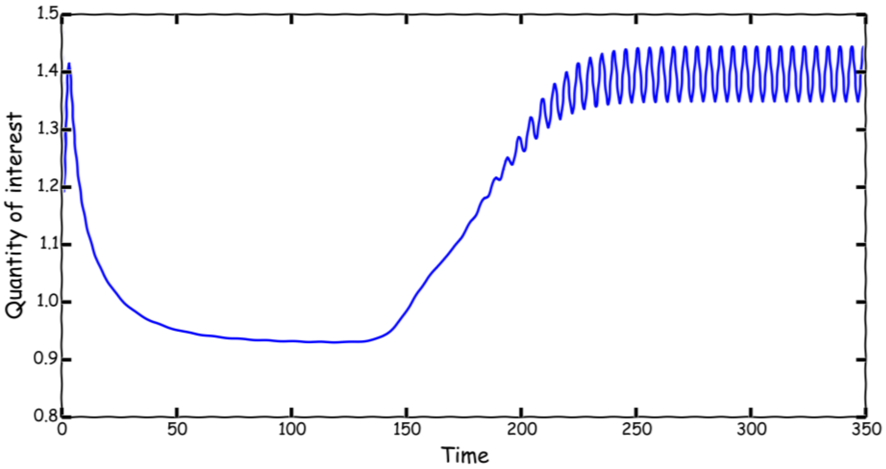
Understanding residuals

- Remember, residuals are not a direct indication that you are converging to the right solution.
- It is better to monitor a quantity of interest (QOI).
- And by the way, you should get physically realistic values.
- In this case, if you monitor the residuals you might get the impression that the simulation is diverging.
- Instead, if you monitor a QOI you will realize that there is an initial transient (long one by the way), then the onset of an instability, and then a periodic behavior of the phenomenon.
- You should assess the convergence of the solution and compute the unsteady statistics in the time window where the behavior of the QOI is periodic.
- To monitor the stability, you can check the minimum and maximum values of the field variables.
- If you have bounded quantities, check that you do not have over-shoots or under-shoots.

Residuals

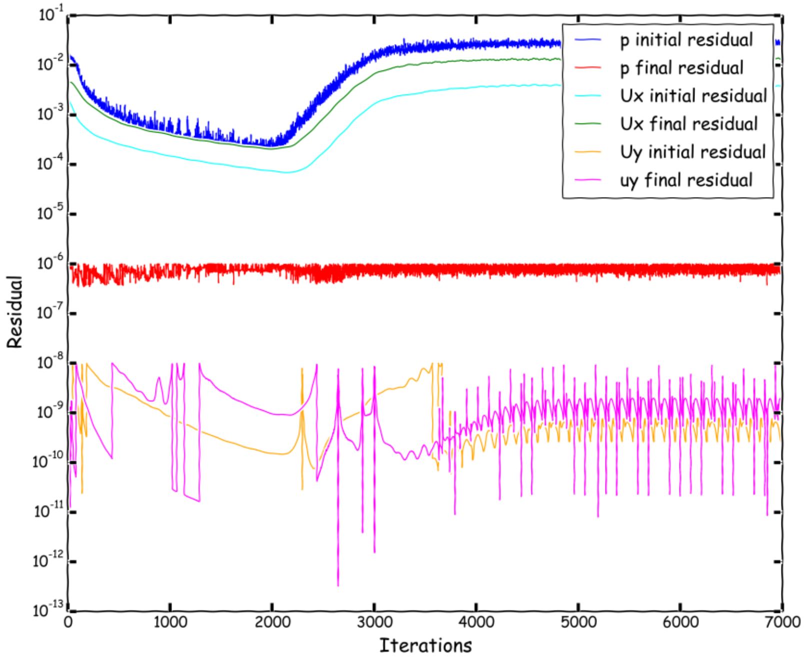


QOI

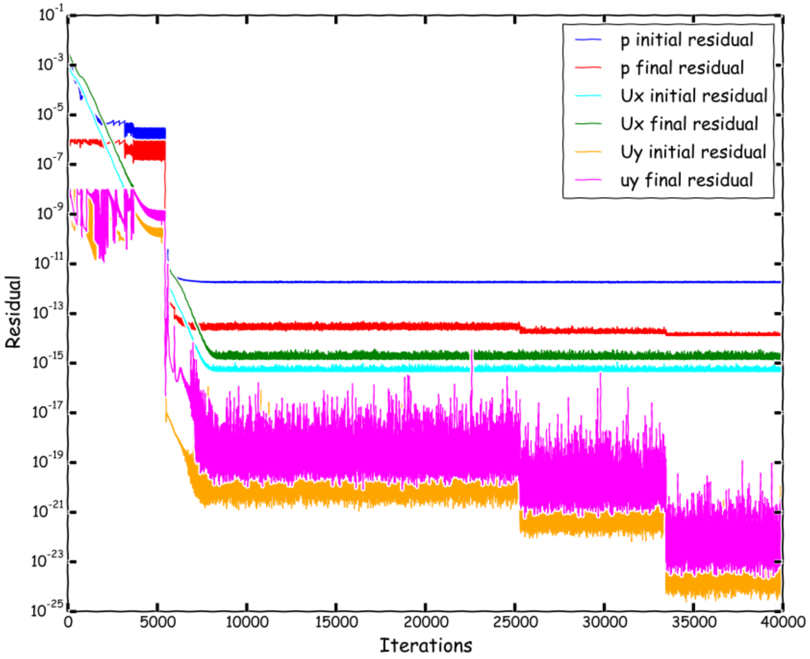


Understanding residuals

- This is the output of the residuals for all field variables of an unsteady case.
- Notice that at the beginning the residuals show a monotonic behavior.
- Then, after a while the convergence rate changes.
- This not necessarily means that the solution is diverging, it might be an indication of unsteadiness.

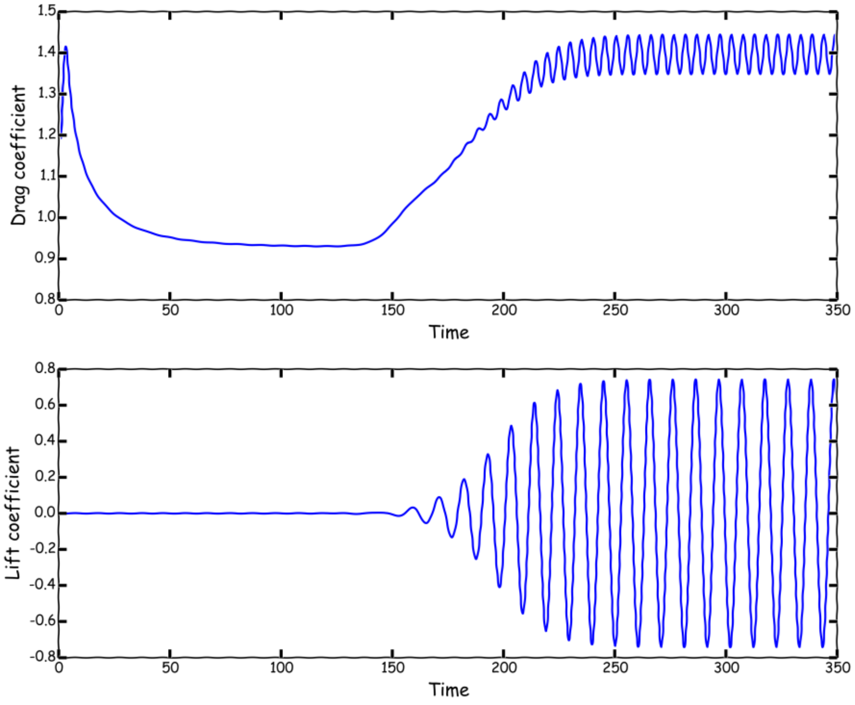


- This is the output of the residuals for all field variables of a steady case.
- The jumps are due to the changes in tolerance introduced while running the simulation.
- As you can see, the residuals are falling in a monotonic way.

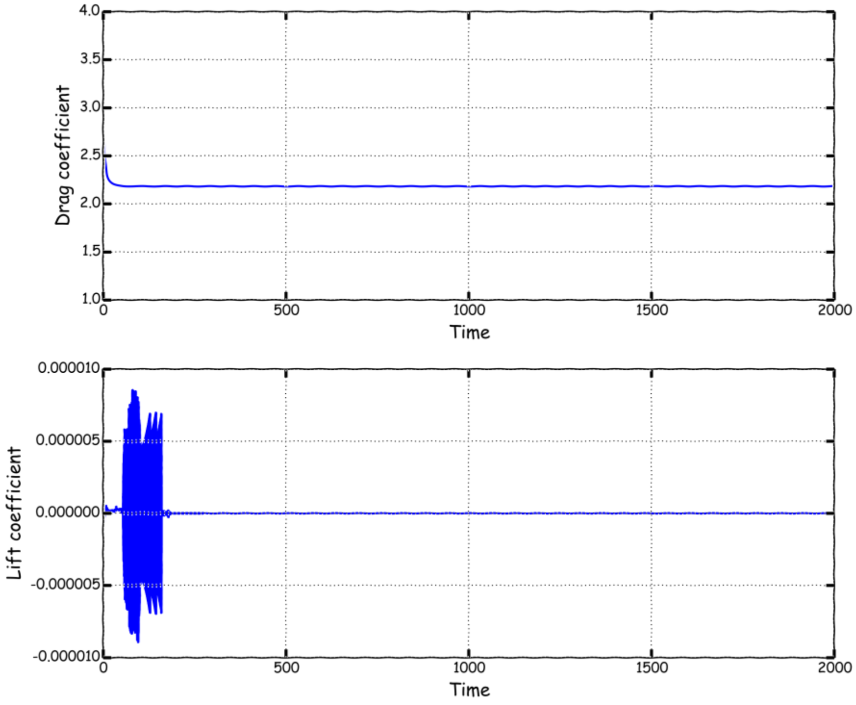


Understanding residuals

- This is the output of the aerodynamic coefficients for an unsteady case.



- This is the output of the aerodynamic coefficients for a steady case.



Roadmap

1. ~~Finite Volume Method: A Crash Introduction~~
2. ~~On the CFL number~~
3. ~~Linear solvers in OpenFOAM®~~
4. ~~Pressure-Velocity coupling in OpenFOAM®~~
5. ~~Unsteady and steady simulations~~
6. ~~Understanding residuals~~
7. **Boundary and initial conditions**
8. Numerical playground

Boundary conditions and initial conditions

On the initial boundary value problem (IBVP)

- First of all, when we use a CFD solver to find the approximate solution of the governing equations, we are solving an Initial Boundary Value Problem (IBVP).
- In an IBVP, we need to impose appropriate boundary conditions and initial conditions.
- No need to say that the boundary conditions and initial conditions need to be physically realistic.
- Boundary conditions are a required component of the numerical method, they tell the solver what is going on at the boundaries of the domain.
- You can think of boundary conditions as source terms.
- Initial conditions are also a required component of the numerical method, they define the initial state of the problem.

Boundary conditions and initial conditions

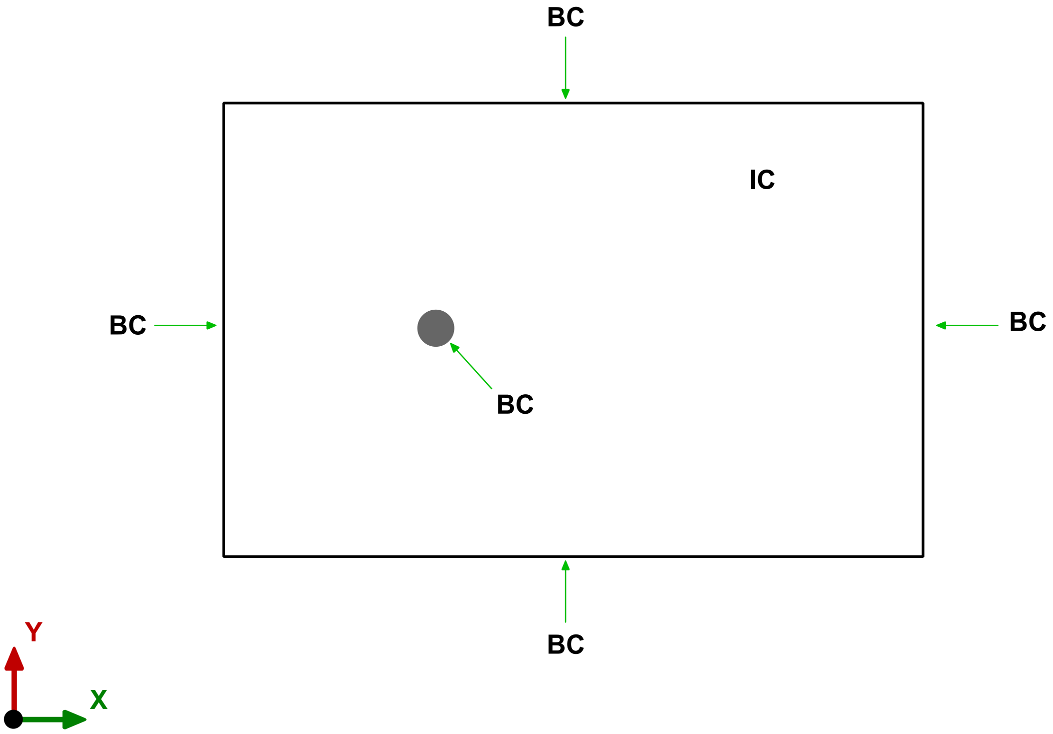
A few words about boundary conditions

- Boundary conditions (BC) can be divided into three fundamental mathematical types:
  - **Dirichlet boundary conditions**: when we use this BC, we prescribe the value of a variable at the boundary.
  - **Neumann boundary conditions**: when we use this BC, we prescribe the gradient normal to the boundary.
  - **Robin Boundary conditions**: this BC is a mixed of Dirichlet boundary conditions and Neumann boundary
- You can use any of these three boundary conditions in OpenFOAM®.
- During this discussion, the semantics is not important, that depends of how you want to call the BCs or how they are named in the solver, *i.e.*, in, inlet, inflow, velocity inlet, incoming flow and so on.
- Defining boundary conditions involves:
  - Finding the location of the boundary condition in the domain.
  - Determining the boundary condition type.
  - Giving the required physical information.
- The choice of the boundary conditions depend on:
  - Geometrical considerations.
  - Physics involved.
  - Information available at the boundary condition location.
  - Numerical considerations.
- And most important, you need to understand the physics involved.

Boundary conditions and initial conditions

A few words about boundary conditions

- To define boundary conditions you need to know the location of the boundaries (where they are in your mesh).
- You also need to supply the information at the boundaries.
- Last but not least important, you must know the physics involved.



Boundary conditions and initial conditions

A few words about initial conditions

- Initial conditions (IC) can be divided into two groups:
  - **Uniform initial conditions**.
  - **Non-uniform initial conditions**.

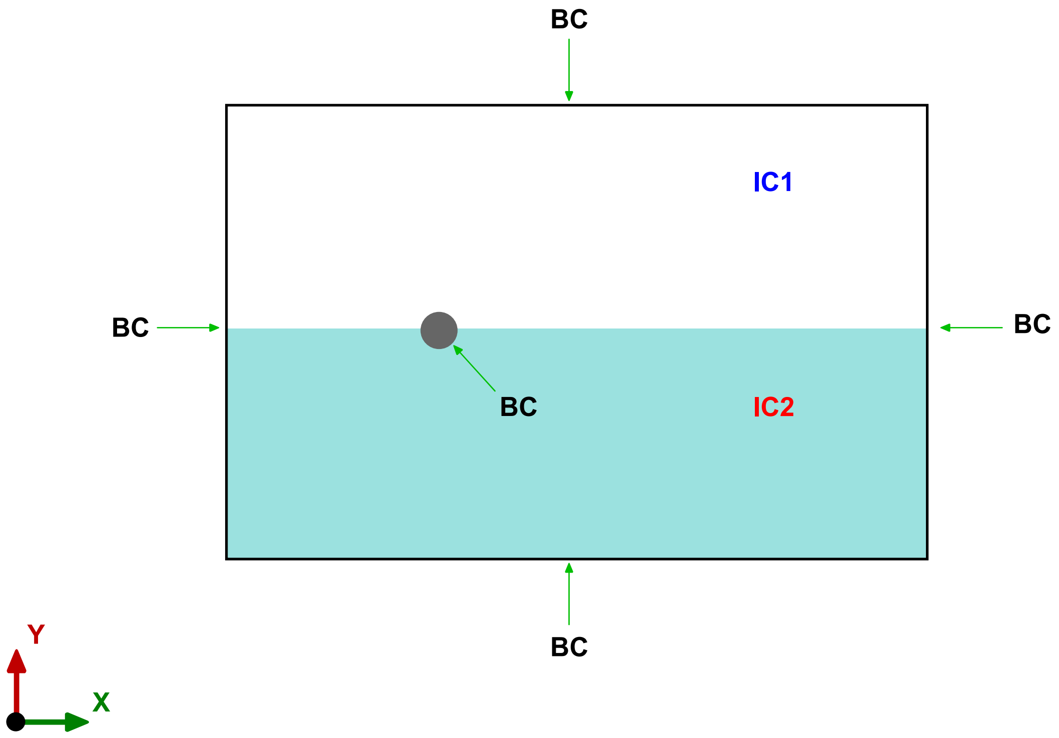
- For non-uniform IC, the value used can be obtained from:
  - Another simulation (including a solution with different grid resolution).
  - A potential solver.
  - Experimental results.
  - A mathematical function
  - Reduced order models.

- Defining initial conditions involves:
  - Finding the location of the initial condition in the domain.
  - Determining the initial condition type.
  - Giving the required physical information.
- The choice of the initial conditions depend on:
  - Geometrical considerations.
  - Physics involved.
  - Information available.
  - Numerical considerations.
- And most important, you need to understand the physics involved.

Boundary conditions and initial conditions

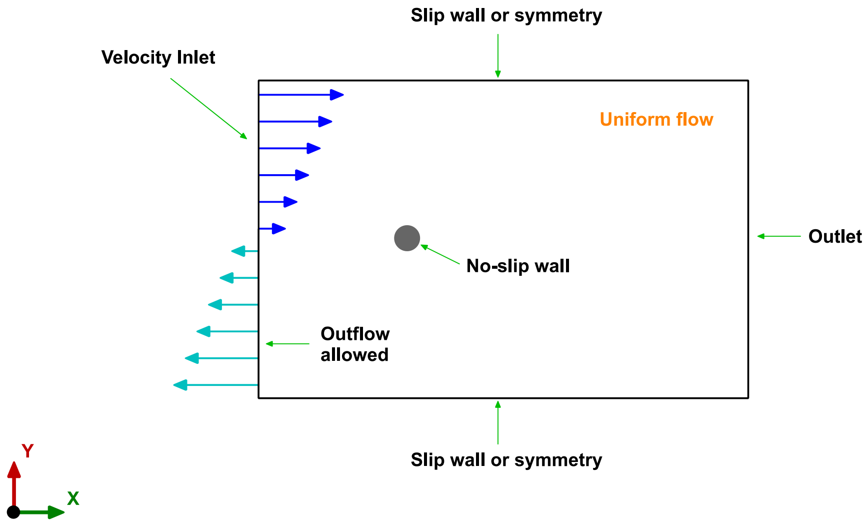
A few words about initial conditions

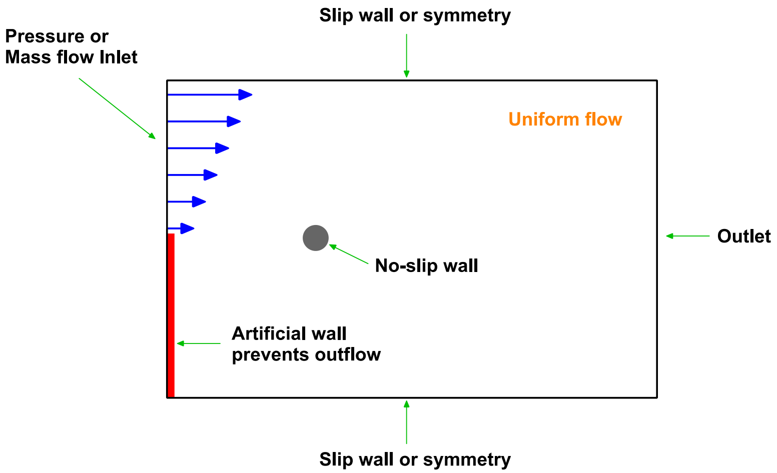
- For initial conditions, you need to supply the initial information or initial state of your problem.
- This information can be a uniform value or a non-uniform value.
- You can apply the initial conditions to the whole domain or separated zones of the domain.
- Last but not least important, you must know the physics involved.



Boundary conditions and initial conditions

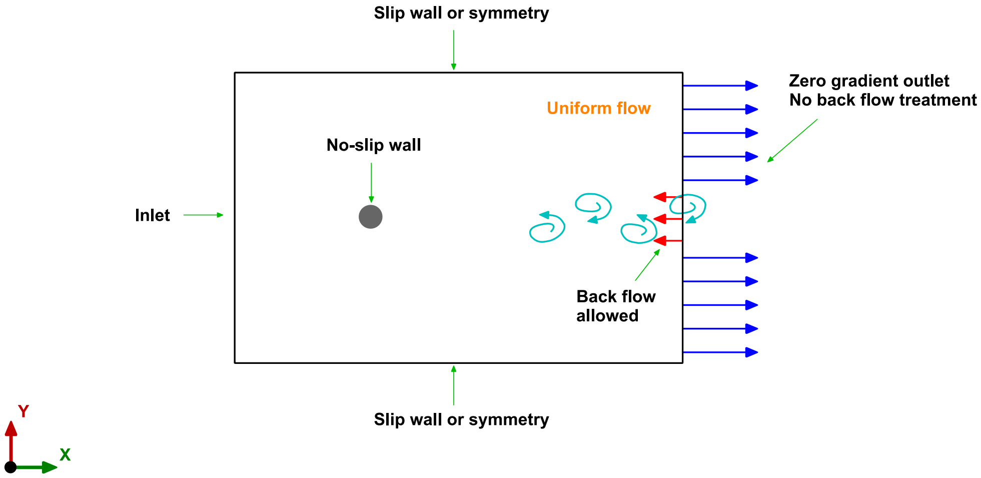
- **Inlets and outlets boundary conditions:**
  - Inlets are for regions where inflow is expected; however, inlets might support outflow when a velocity profile is specified.
  - Pressure boundary conditions do not allow outflow at the inlets.
  - Velocity specified inlets are intended for incompressible flows.
  - Pressure and mass flow inlets are suitable for compressible and incompressible flows.
  - Same concepts apply to outlets, which are regions where outflow is expected.

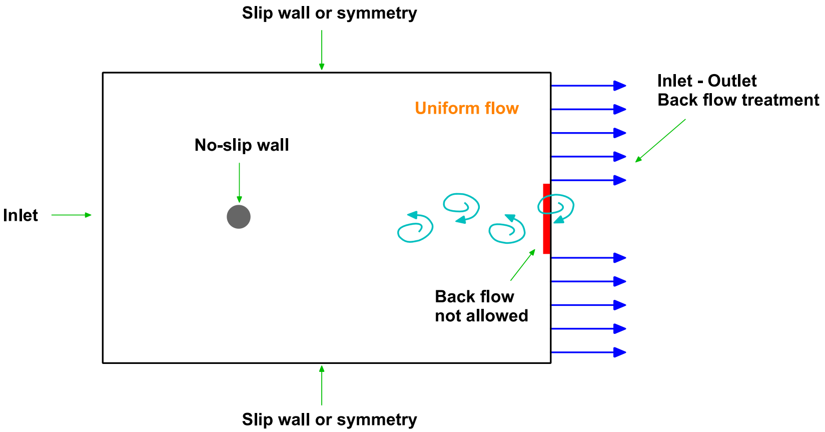




Boundary conditions and initial conditions

- **Zero gradient (Neumann) and backflow boundary conditions:**
  - Zero gradient boundary conditions extrapolates the values from the domain. They require no information.
  - Zero gradient boundary conditions can be used at inlets, outlets, and walls.
  - Backflow boundary conditions provide a generic outflow/inflow condition, with specified inflow/outflow for the case of backflow.
  - In the case of a backflow outlet, when the flux is positive (out of domain) it applies a Neumann boundary condition (zero gradient), and when the flux is negative (into of domain), it applies a Dirichlet boundary condition (fixed value).
  - Same concept applies to backflow inlets.

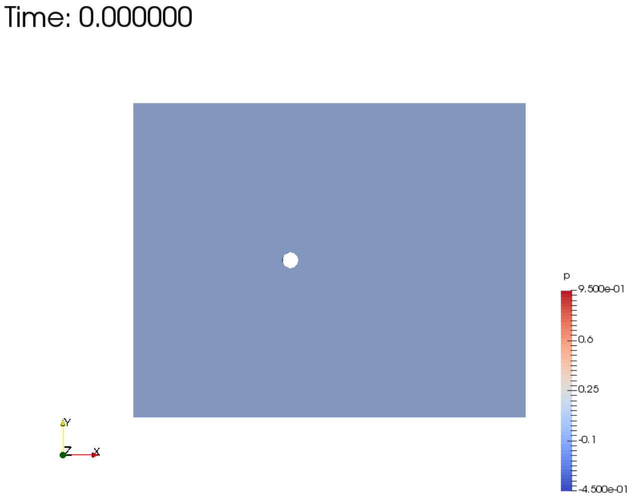




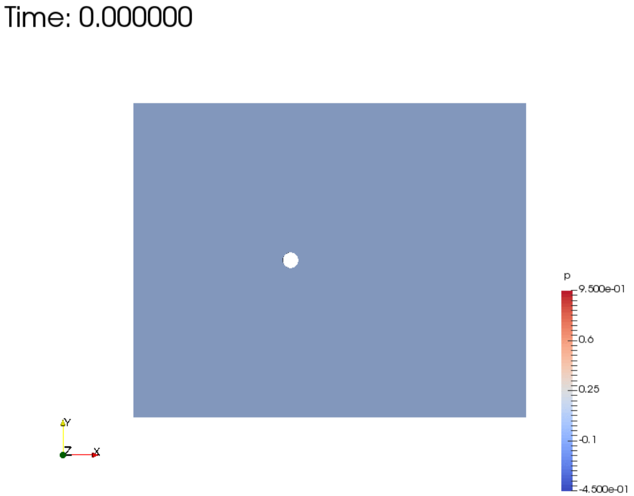
Boundary conditions and initial conditions

- **On the outlet pressure boundary condition**
  - Some combinations of boundary conditions are very stable, and some are less reliable.
  - And some configurations are unreliable.
    - Inlet velocity at the inlet and pressure zero gradient at the outlet. This combination should be avoided because the static pressure level is not fixed.
  - Qualitatively speaking, the results are very different.
  - This simulation will eventually crash.

BCs 1. Inlet velocity and fixed outlet pressure
www.wolfdynamics.com/wiki/BC/aniBC1.gif



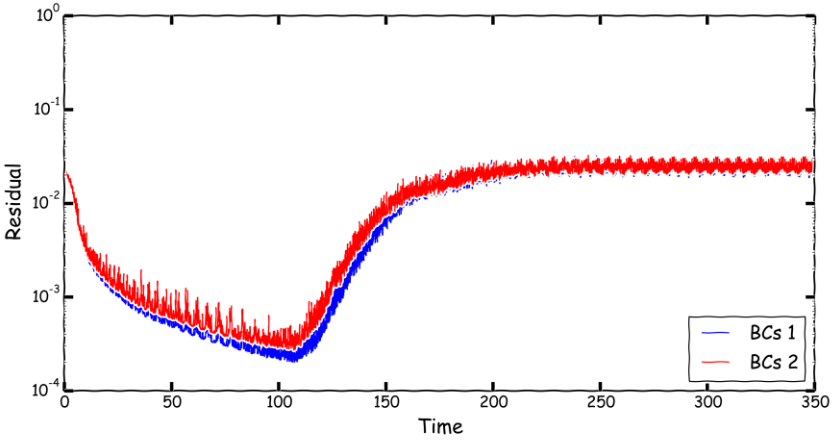
BCs 2. Inlet velocity and zero gradient outlet pressure
www.wolfdynamics.com/wiki/BC/aniBC2.gif



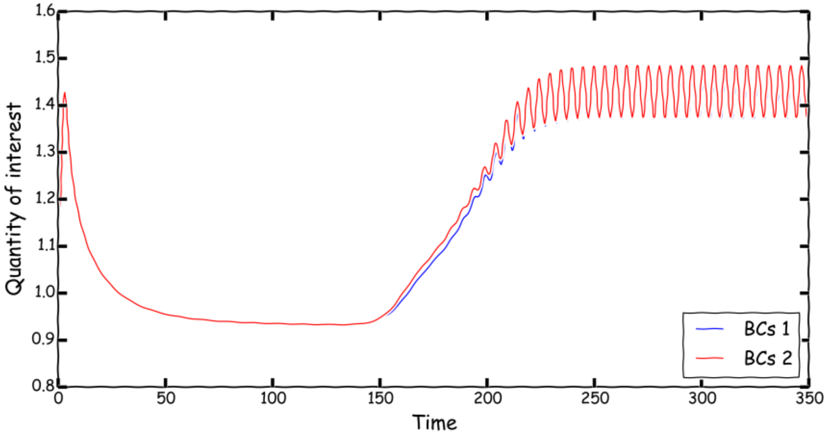
Boundary conditions and initial conditions

- **On the outlet pressure boundary condition**
  - If you only rely on a QOI and the residuals, you will not see any major difference between the two cases with different outlet pressure boundary condition.
  - This is very misleading.
  - However, when you visualize the solution you will realize that something is wrong. This is a case where pretty pictures can be used to troubleshoot the solution.
  - Quantitative speaking, the results are very similar.
  - However, this simulation will eventually crash.

Residual plot for pressure

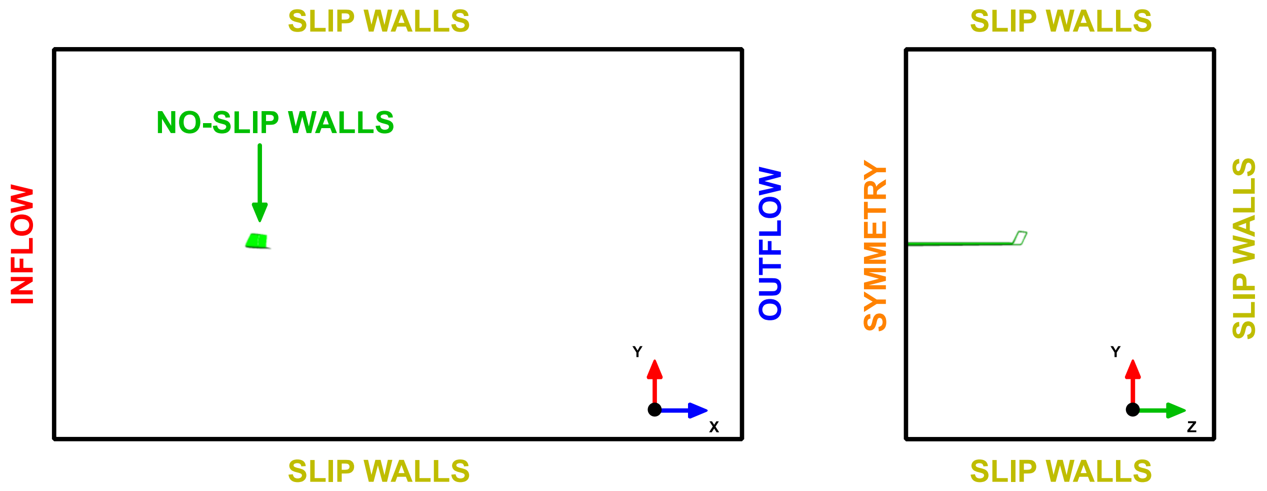


Quantity of interest – Force coefficient on the body



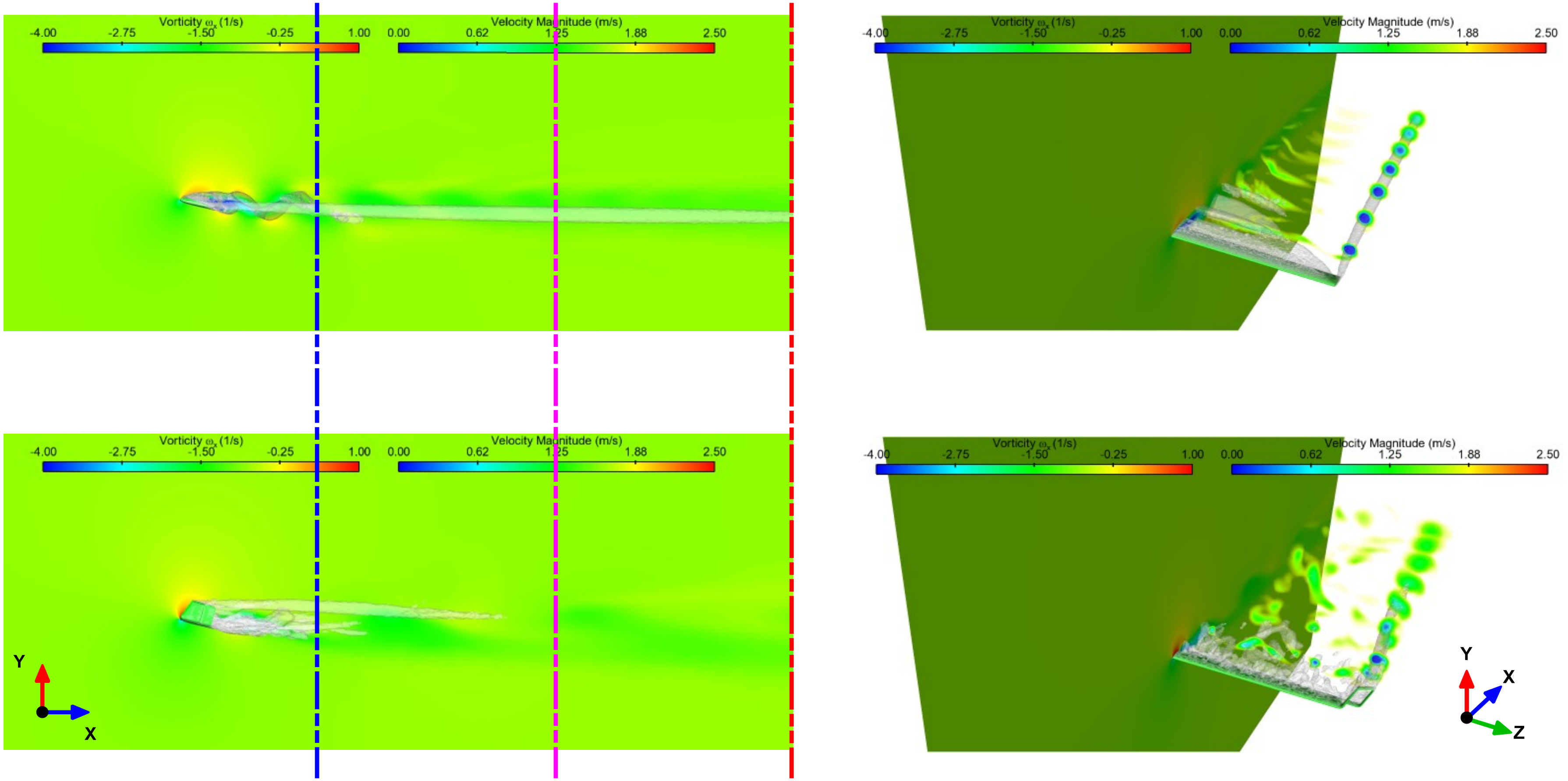
Boundary conditions and initial conditions

- **Symmetry boundary conditions:**
  - Symmetry boundary conditions are a big simplification of the problem. However, they help to reduce mesh cell count.
  - Have in mind that symmetry boundary conditions only apply to **planar faces**.
  - To use symmetry boundary conditions, both the geometry and the flow field must be symmetric.
  - Mathematically speaking, setting a symmetry boundary condition is equivalent to zero normal velocity at the symmetry plane, and zero normal gradients of all variables at the symmetry plane.
  - Physically speaking, they are equivalent to slip walls.



Boundary conditions and initial conditions

- **Location of the outlet boundary condition:**
  - Place outlet boundary conditions as far as possible from recirculation zones or backflow conditions, by doing this you increase the stability.
  - Remember, backflow conditions requires special treatment.



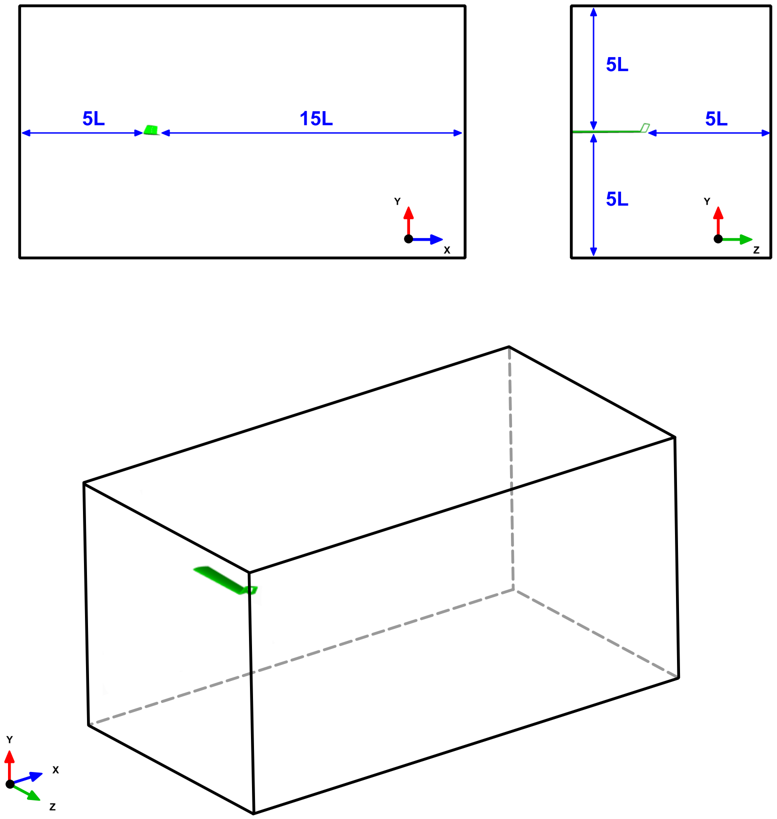
Possible backflow

Might be OK

Far enough so the flow can be considered fully developed

Boundary conditions and initial conditions

- **Domain dimensions (when the dimensions are not known):**
  - If you do not have any constrain in the domain dimensions, you can use as a general guideline the dimensions illustrated in the figure below, where **L** is a reference length (in this case, **L** is the wing chord).
  - The values illustrated in the figure are on the conservative side, nut if you want to play safe, multiply the values by two or more.
  - Always verify that there are no significant gradients normal to any of the boundaries patches. If there are, you should consider increasing the domain dimensions.



Boundary conditions and initial conditions

A few considerations and guidelines

- Boundary conditions and initial conditions need to be physically realistic.
- Poorly defined boundary conditions can have a significant impact on your solution.
- Initial conditions are as important as the boundary conditions.
- A good initial condition can improve the stability and convergence rate.
- On the other hand, unphysical initial conditions can slow down the convergence rate or can cause divergence.
- You need to define boundary conditions and initials conditions for every single variable you are solving.
- Setting the right boundary conditions is extremely important, but you need to understand the physics.
- You need to understand the physics in order to set the right boundary conditions.
- Do not force the flow at the outlet, use a zero normal gradient for all flow variables except pressure. The solver extrapolates the required information from the interior.
- Be careful with backward flow at the outlets (flow coming back to the domain) and backward flow at inlets (reflection waves), they required special treatment.
- If possible, select inflow and outflow boundary conditions such that the flow either goes in or out normal to the boundaries.
- At outlets, use zero gradient boundary conditions only with incompressible flows and when you are sure that the flow is fully developed.
- Outlets that discharge to the atmosphere can use a static pressure boundary condition. This is interpreted as the static pressure of the environment into which the flow exhausts.

Boundary conditions and initial conditions

A few considerations and guidelines

- Inlets that take flow into the domain from the atmosphere can use a total pressure boundary condition (e.g. open window).
- Mass flow inlets produce a uniform velocity profile at the inlet.
- Pressure specified boundary conditions allow a natural velocity profile to develop.
- The required values of the boundary conditions and initial conditions depend on the equations you are solving, and physical models used, *e.g.*,
    - For incompressible and laminar flows you will need to set only the velocity and pressure.
    - If you are solving a turbulent compressible flow you will need to set velocity, pressure, temperature and the turbulent variables.
    - For multiphase flows you will need to set the primitives variables for each phase. You will also need to initialize the phases.
    - If you are doing turbulent combustion or chemical reactions, you will need to define the species, reactions and turbulent variables.
- Minimize grid skewness, non-orthogonality, growth rate, and aspect ratio near the boundaries. You do not want to introduce diffusion errors early in the simulation, especially close to the inlets.
- Try to avoid large gradients in the direction normal to the boundaries and near inlets and outlets.
    - That is to say, put your boundaries far away from where things are happening.

Boundary conditions and initial conditions

- OpenFOAM® distinguish between **base type** boundary conditions and **numerical type** boundary conditions.

Base type boundary conditions	Numerical type boundary conditions
• Base type boundary conditions are based on geometry information (surface patches) or on inter-processor communication link (halo boundaries). • Base type boundary conditions are defined in the file *boundary* located in the directory **constant/polyMesh** • The file *boundary* is automatically created when you generate or convert the mesh. • When you convert a mesh to OpenFOAM® format, you might need to manually modify the file *boundary*. This is because the conversion utilities do not recognize the boundary type of the original mesh. • Remember, if a base type boundary condition is missing, OpenFOAM® will complain and will tell you where and what is the error. • Also, if you misspelled something OpenFOAM® will complain and will tell you where and what is the error	• Numerical type boundary condition assigns the value to the field variables in the given surface patch. • Numerical type boundary conditions are defined in the field variables dictionaries located in the directory **0** (*e.g.* *U*, *p*). • When we talk about numerical type boundary conditions, we are referring to Dirichlet, Neumann or Robin boundary conditions. • You need to manually create the field variables dictionaries (*e.g.* *0/U*, *0/p*, *0/T*, *0/k*, *0/omega*). • Remember, if you forget to define a numerical boundary condition, OpenFOAM® will complain and will tell you where and what is the error. • Also, if you misspelled something OpenFOAM® will complain and will tell you where and what is the error.

Boundary conditions and initial conditions

- The following base type and numerical type boundary conditions are constrained or paired.
- That is, the type needs to be same in the *boundary* dictionary and field variables dictionaries (*e.g.* *0/U*, *0/p*, *0/T*, *0/k*, *0/omega*).

Base type	Numerical type
constant/polyMesh/boundary	*0/U* - *0/p* - *0/T* - *0/k* - *0/omega* (IC/BC)
cyclic **cyclicAMI** **empty** **processor** **symmetry** **symmetryPlane** **wedge**	**cyclic** **cyclicAMI** **empty** **processor** **symmetry** **symmetryPlane** **wedge**

- These are known as **constraint** patches in OpenFOAM.
- To find a complete list and the source code location of these patches, go to the directory **`$WM_PROJECT_DIR`** and type in the terminal:
  - **`$> find . -type d -iname *constraint*`**

Boundary conditions and initial conditions

- The base type **patch** can be any of the boundary conditions available in OpenFOAM®.
- Mathematically speaking; they can be Dirichlet, Neumann or Robin boundary conditions.

Base type	**Numerical type**
constant/polyMesh/boundary	*0/U - 0/p - 0/T - 0/k - 0/omega* (IC/BC)
patch	**advective** **calculated** **codedFixedValue** **epsilonWallFunction** **fixedValue** **inletOutlet** **movingWallVelocity** **rotatingWallVelocity** **slip** **supersonicFreeStream** **totalPressure** **zeroGradient** … and so on Refer to the doxygen documentation or the source code for a list of all numerical boundary conditions available.

Boundary conditions and initial conditions

- The **wall** base type boundary condition is defined as follows:

Base type	**Numerical type**	
constant/polyMesh/boundary	*0/U*	*0/p*

wall	**type fixedValue;**
	value uniform (0 0 0);
		zeroGradient

- This boundary condition is not contained in the patch base type boundary conditions group, because specialize modeling options can be used on this boundary condition.
- An example is turbulence modeling, where turbulence can be generated or dissipated at the walls.

Boundary conditions and initial conditions

- To deal with **backflow** at outlets, you can use the following boundary condition:

Base type	Numerical type	
constant/polyMesh/boundary	*0/U*	*0/p*

patch	**type inletOutlet;**
	inletValue uniform (0 0 0);
	value uniform (0 0 0);
		type fixedValue;
		value uniform 0;

- The **inletValue** keyword is used for the reverse flow.
- In this case, if flow is coming back into the domain it will use the value set using the keyword **inletValue**. Otherwise it will use a **zeroGradient** boundary condition.
- For the turbulent variables (**k**, **omega**, **epsilon**, and so on), you can use **inletOutlet** type (pay attention that these quantities are scalars).

Boundary conditions and initial conditions

- Typical boundary conditions are as follows (external aerodynamics),

Boundary type description	**Pressure**	**Velocity**	**Turbulence fields**
Inlet face	**zeroGradient**	**fixedValue**	**fixedValue**
Outlet face	**fixedValue**	**inletOutlet**	**inletOutlet**
Wall face	**zeroGradient**	**fixedValue**	**Wall functions***
Symmetry face	**symmetry**	**symmetry**	**symmetry**
Periodic face	**cyclic**	**cyclic**	**cyclic**
Empty face (2D)	**empty**	**empty**	**empty**
Slip wall	**slip**	**slip**	**slip**

* Wall functions can be: **kqWallFunction**, **omegaWallFunction**, **nutkWallFunction**, and so on (next slide).

Boundary conditions and initial conditions

- Typical wall functions boundary conditions are as follows,

Field variable	**Wall functions – High RE**	**Resolved BL – Low RE**
nut	**nut(–)WallFunction***	**fixedValue 0** or a small number
k, q, R	**kqRWallFunction**	**fixedValue 0** or a small number
epsilon	**epsilonWallFunction**	**zeroGradient** or **fixedValue 0** or a small number
omega	**omegaWallFunction**	**omegaWallFunction** or **fixedValue** with a big number
zeta	**–**	**fixedValue 0** or a small number
nuTilda	**–**	**fixedValue 0** or a small number

* **nutkAtmRoughWallFunction**, **nutkRoughWallFunction**, **nutkWallFunction**, **nutLowReWallFunction**, **nutURoughWallFunction**, **nutUSpaldingWallFunction**, **nutUTabulatedWallFunction**, **nutUWallFunction**, **nutWallFunction**.

Boundary conditions and initial conditions

- Finally, remember that the name of the base type boundary condition and the name of the numerical type boundary condition needs to be the same, if not, OpenFOAM® will complain.
- Pay attention to this, specially if you are converting the mesh from another format.
- Also, do not use spaces of funny characters when assigning the names to the boundary patches.
- The following names are consistent among all dictionary files,

Base type	**Numerical type**	
constant/polyMesh/boundary	*0/U*	*0/p*
inlet	**inlet**	**inlet**
top	**top**	**top**
cylinder	**cylinder**	**cylinder**
sym	**sym**	**sym**

Boundary conditions and initial conditions

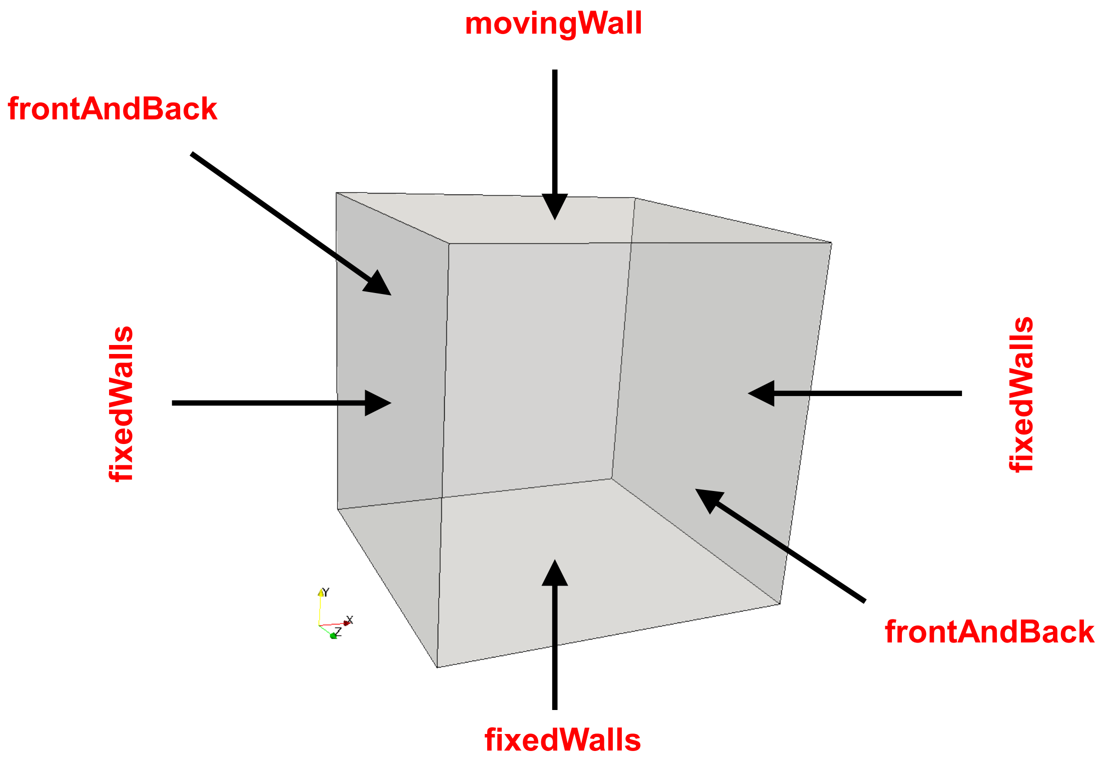
- There is a plethora of boundary conditions implemented in OpenFOAM®.
- You can find the source code of the main numerical boundary conditions in the following directory:
  - **`$WM_PROJECT_DIR/src/finiteVolume/fields/`**
- The wall boundary conditions for the turbulence models (wall functions), are located in the following directory:
  - **`$WM_PROJECT_DIR/src/MomentumTransportModels/momentumTransportModels/derivedFvPatchFields/wallFunctions`**
- To find all the boundary conditions implemented in OpenFOAM, go to the directory **`$WM_PROJECT_DIR`** and type in the terminal,
  - `$> find . -type d -iname *fvPatch*`
  - `$> find . -type d -iname *derivedFv*`
  - `$> find . -type d -iname *pointPatch*`
- To get more information about all the boundary conditions available in OpenFOAM® you can read the Doxygen documentation.
- You can access the documentation online at this link http://cpp.openfoam.org/v8/

Boundary conditions and initial conditions

The *constant/polyMesh/boundary* dictionary

- For a generic case, the file *boundary* is divided as follows

```
3
(


    movingWall
    {
        type            patch;
        nFaces          20;
        startFace       760;
    }

    fixedWalls
    {
        type            wall;
        nFaces          60;
        startFace       780;
    }

    frontAndBack
    {
        type            empty;
        nFaces          800;
        startFace       840;
    }


)
```



Boundary conditions and initial conditions

The *constant/polyMesh/boundary* dictionary

- For a generic case, the file *boundary* is divided as follows

```
3
(


    movingWall
    {
        type            patch;
        nFaces          20;
        startFace       760;
    }

    fixedWalls
    {
        type            wall;
        nFaces          60;
        startFace       780;
    }

    frontAndBack
    {
        type            empty;
        nFaces          800;
        startFace       840;
    }


)
```

Number of surface patches
There must be 3 patches definition.

Name and type of the surface patches

- The name and type of the patch is given by the user.
- You can change the name if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **movingWall** from **patch** to **wall**.
- When converting the mesh from a third-party format, OpenFOAM® will try to recover the information from the original format. But it might happen that it does not recognize the base type and name of the original format. If that is your case, you will need to modify the file manually or using any of the mesh manipulation utilities distributed with OpenFOAM®.

nFaces and startFace keywords

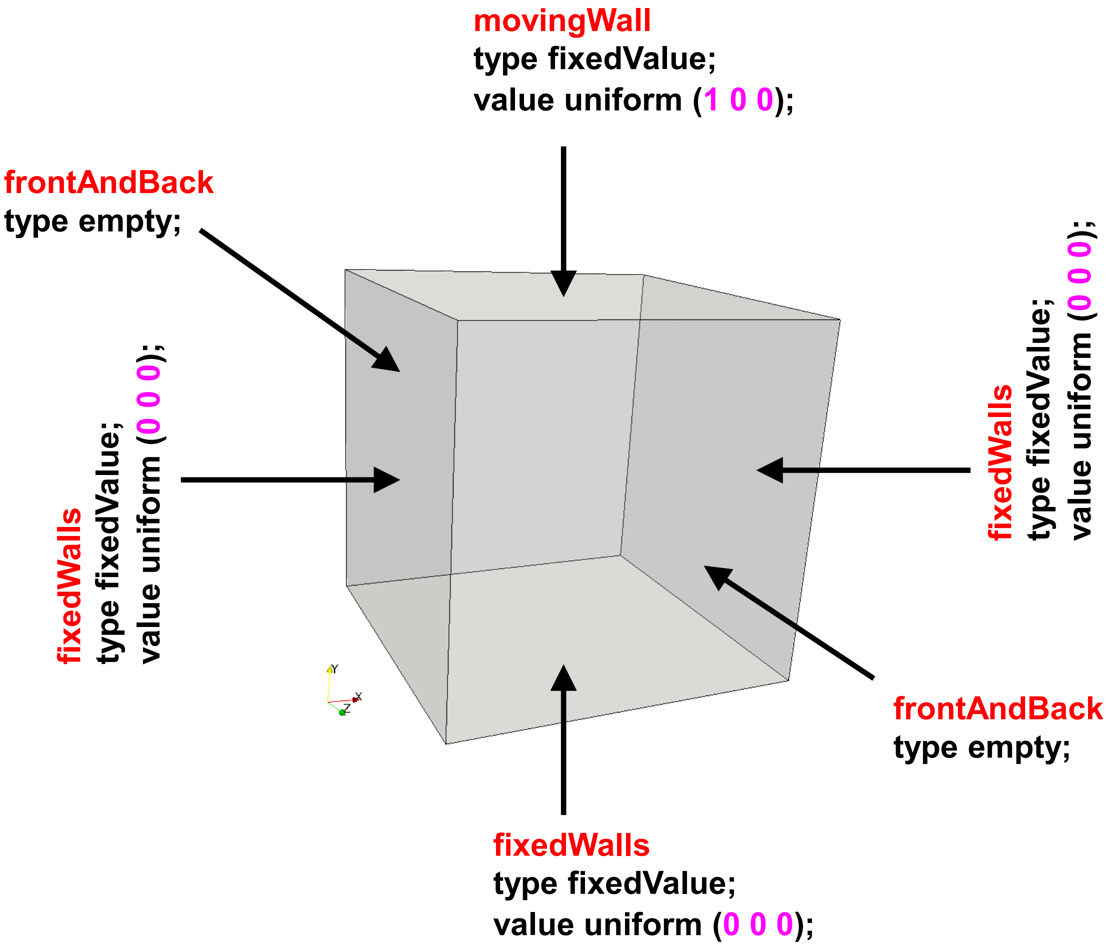
- Unless you know what are you doing, **you do not need to change this information.**

Boundary conditions and initial conditions

The *0/U* dictionary

- For a generic case, the **numerical** type BC are assigned as follows (**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;
    }
}
```



movingWall
type fixedValue;
value uniform (1 0 0);

frontAndBack
type empty;

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

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

frontAndBack
type empty;

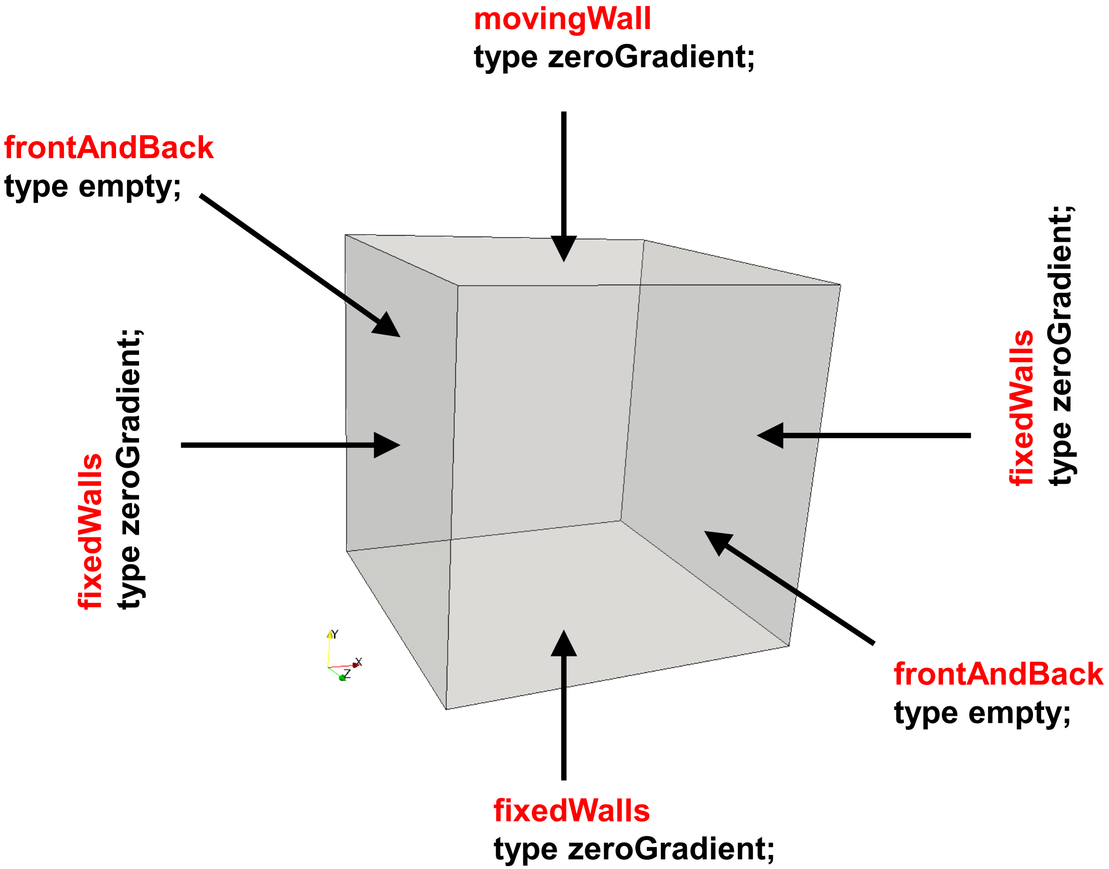
fixedWalls
type fixedValue;
value uniform (0 0 0);

Boundary conditions and initial conditions

The *0/p* dictionary

- For a generic case, the **numerical** type BC are assigned as follows (**p**),

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

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type                zeroGradient;
    }

    fixedWalls
    {
        type                zeroGradient;
    }

    frontAndBack
    {
        type                empty;
    }
}
```



movingWall
type zeroGradient;

frontAndBack
type empty;

fixedWalls
type zeroGradient;

fixedWalls
type zeroGradient;

frontAndBack
type empty;

fixedWalls
type zeroGradient;

Roadmap

1. ~~Finite Volume Method: A Crash Introduction~~
2. ~~On the CFL number~~
3. ~~Linear solvers in OpenFOAM®~~
4. ~~Pressure-Velocity coupling in OpenFOAM®~~
5. ~~Unsteady and steady simulations~~
6. ~~Understanding residuals~~
7. ~~Boundary and initial conditions~~
8. **Numerical playground**

Numerical playground

Merry-go-round:

Pure convection of a passive scalar in a vector field – One dimensional tube.

Numerical playground

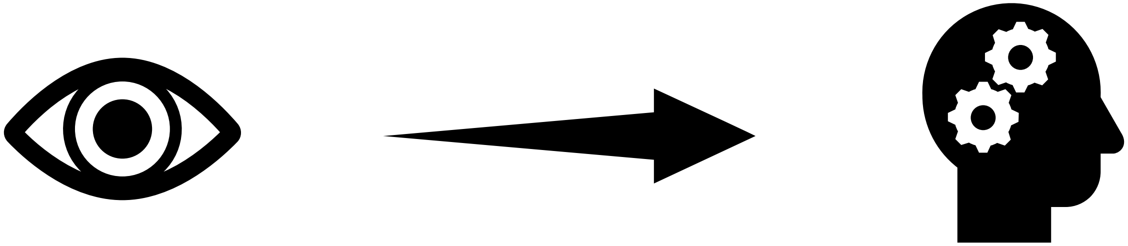
- This is a visual and mental exercise only.

- You will find this case in the directory

`$PTOFC/101FVM/pureConvection/orthogonal_1d`

- In this directory, you will also find the *`README.FIRST`* file with the instructions of how to run the case.

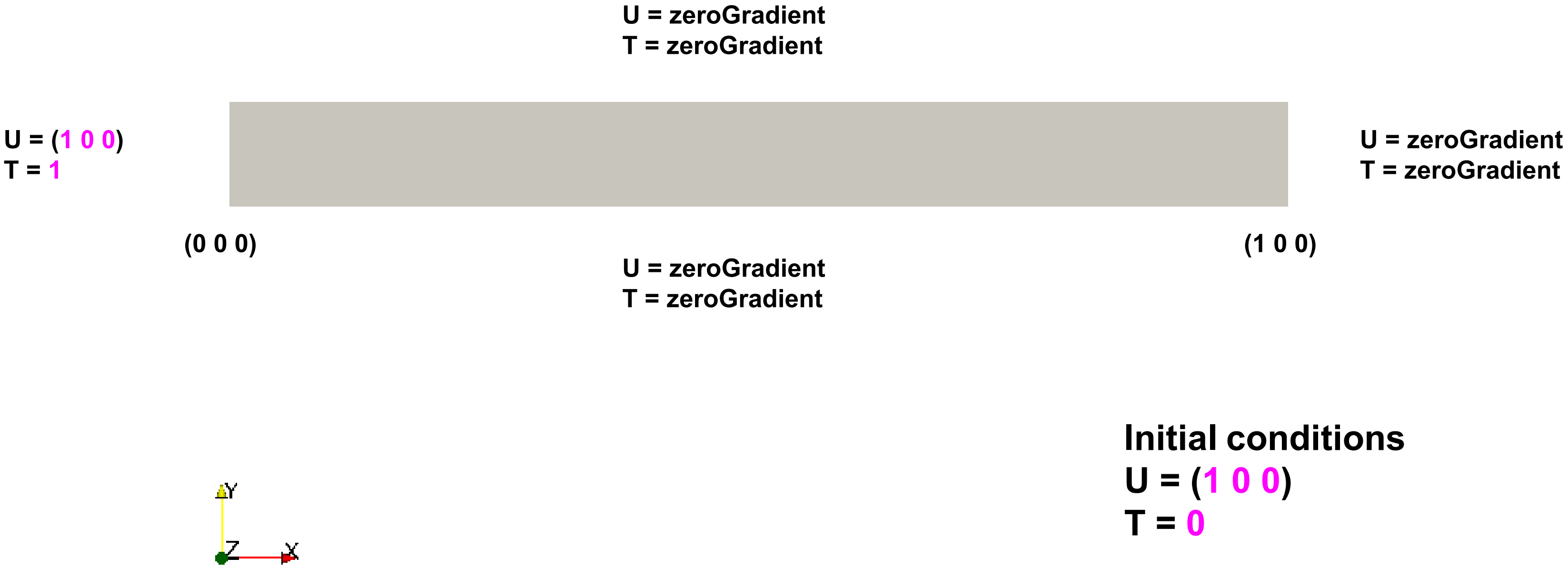
- Hereafter, we will focus our eyes to train our brain.



Numerical playground

Pure convection of a scalar in a vector field – One dimensional tube.

$$\frac{\partial T}{\partial t} + \nabla \cdot (\phi T) - \cancelto{0}{\nabla \cdot (\Gamma \nabla T)} = 0$$



U = zeroGradient
T = zeroGradient

U = (1 0 0)
T = 1

U = zeroGradient
T = zeroGradient

(0 0 0)

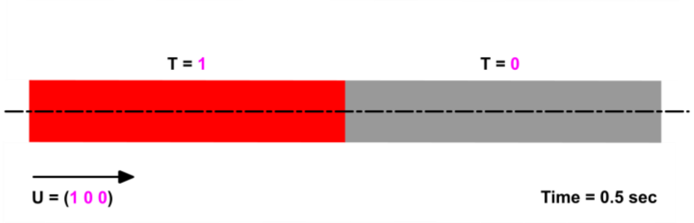
(1 0 0)

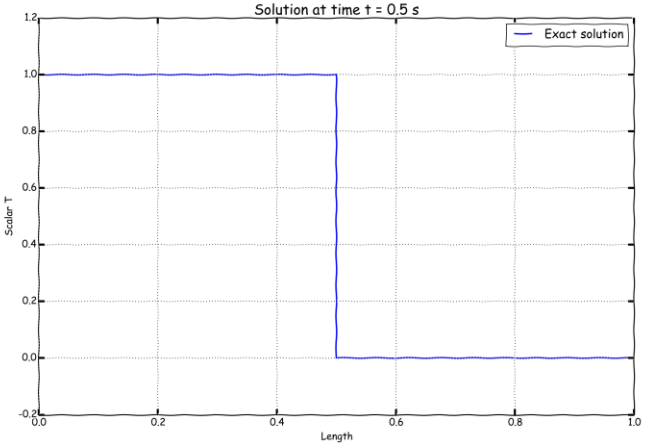
U = zeroGradient
T = zeroGradient

Initial conditions
U = (1 0 0)
T = 0

Numerical playground

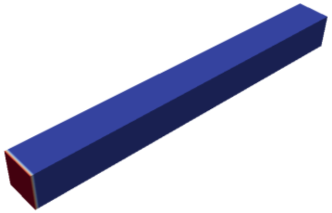
- This problem has an exact solution in the form of a traveling wave.
- We will use this case to study the different discretization schemes implemented in OpenFOAM®.
- In the figure, we show the solution for time = 0.5 s





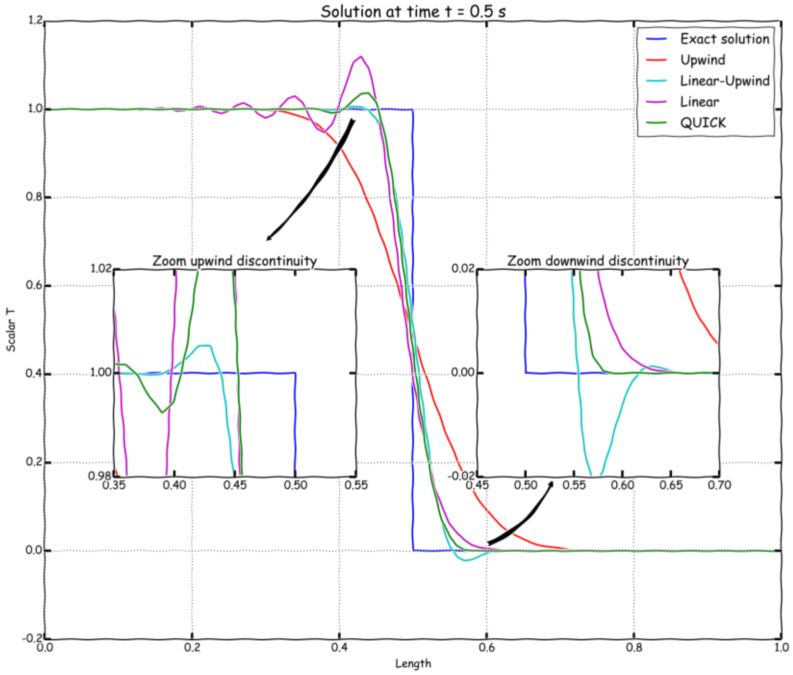
www.wolfdynamics.com/wiki/pureconvection/xani1.gif

www.wolfdynamics.com/wiki/pureconvection/xani2.gif

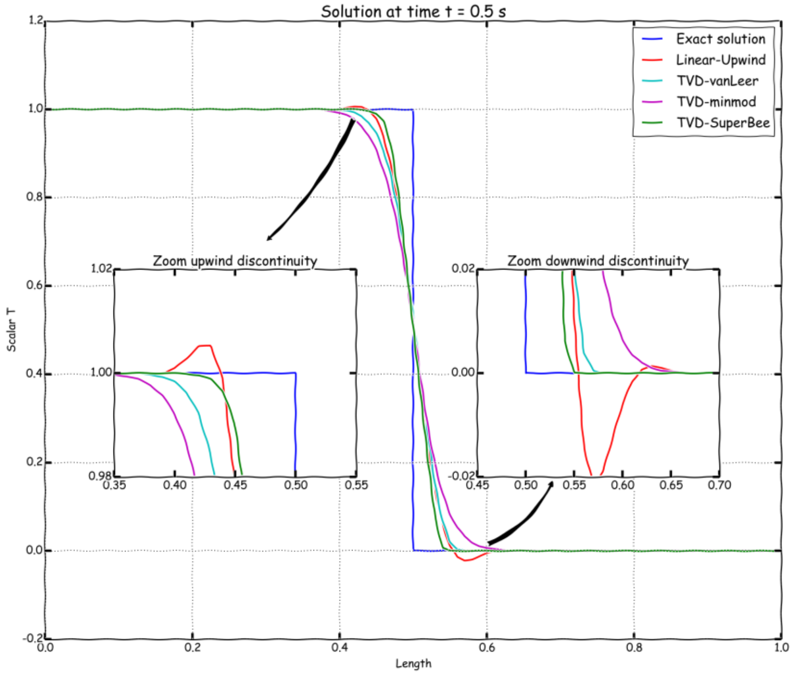


Numerical playground

Comparison of different spatial discretization schemes.
Euler in time – 100 cells – CFL = 0.1
Linear limiter functions on the Sweby diagram.

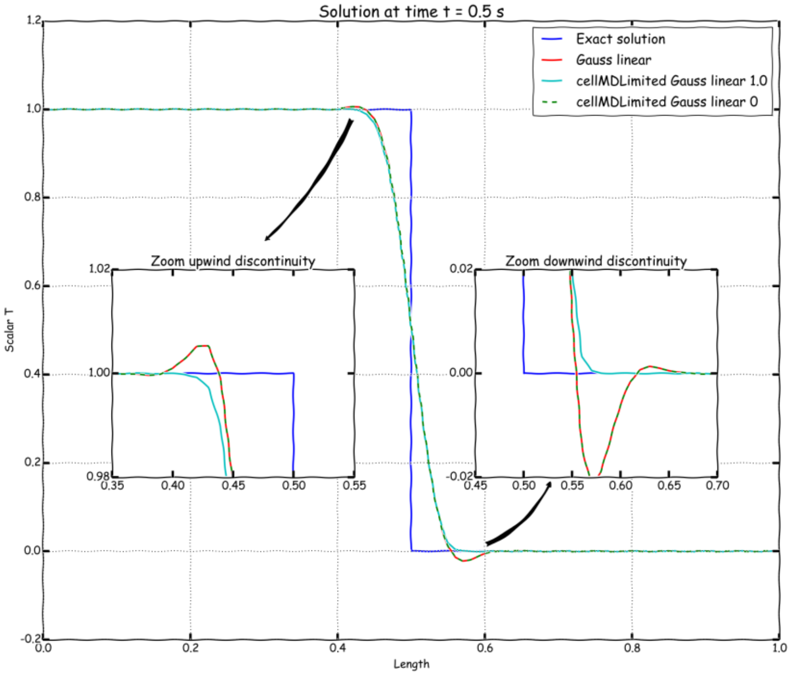


Comparison of different spatial discretization schemes.
Euler in time – 100 cells – CFL = 0.1
Non-linear limiter functions on the Sweby diagram.

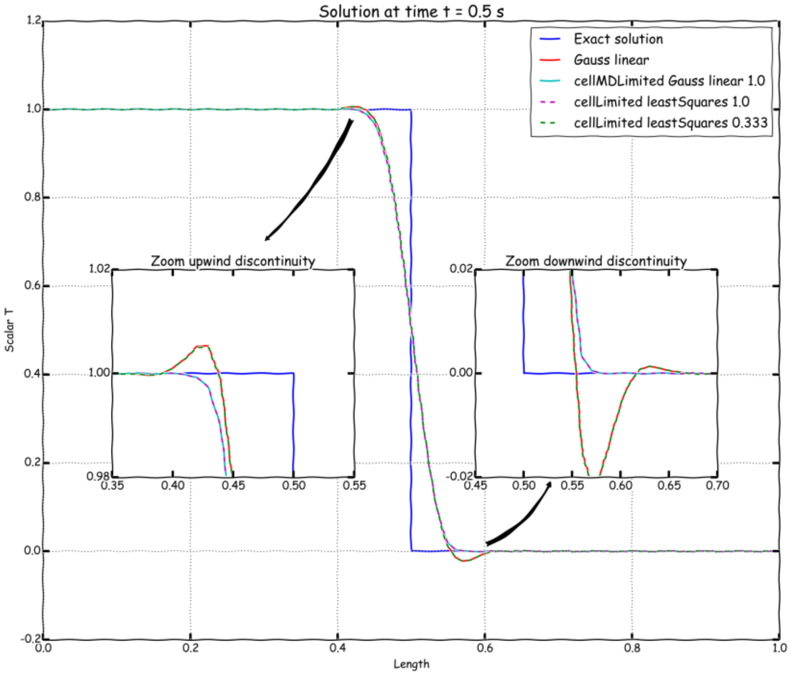


Numerical playground

Comparison of different gradient limiters.
Linear upwind in space – Euler in time – 100 cells – CFL 0.1

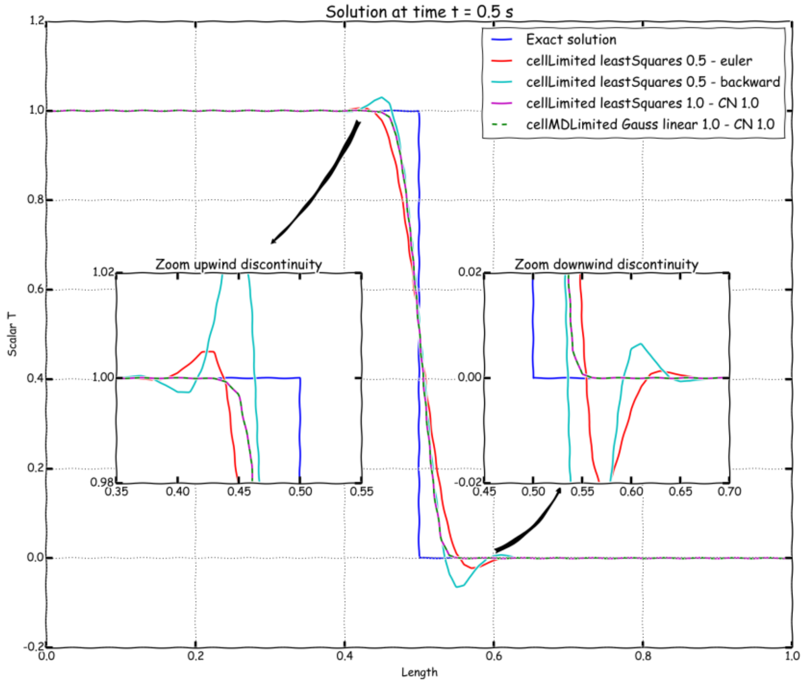


Comparison of different gradient limiters.
Linear upwind in space – Euler in time – 100 cells – CFL 0.1

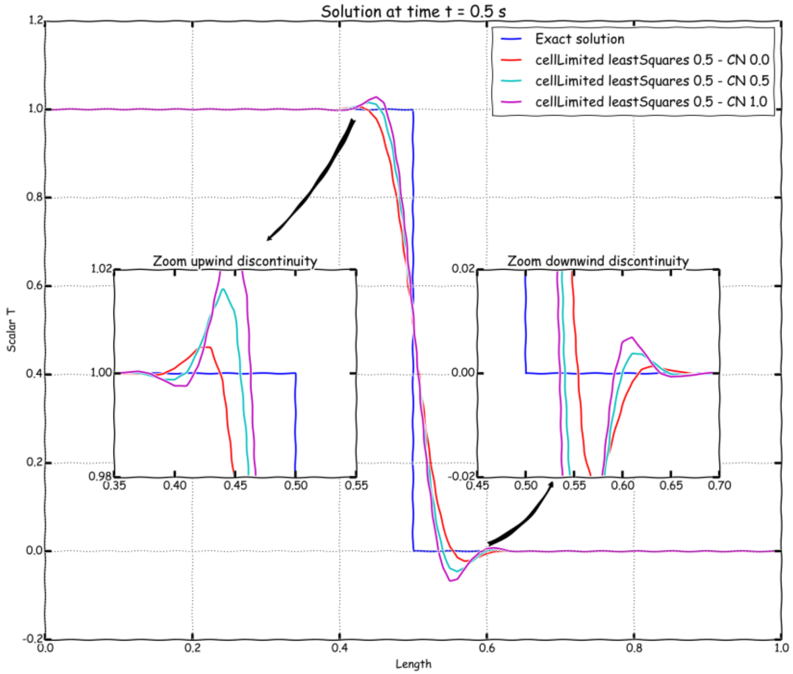


Numerical playground

Comparison of different time discretization schemes and gradient limiters.
Linear upwind in space – 100 cells – CFL 0.1

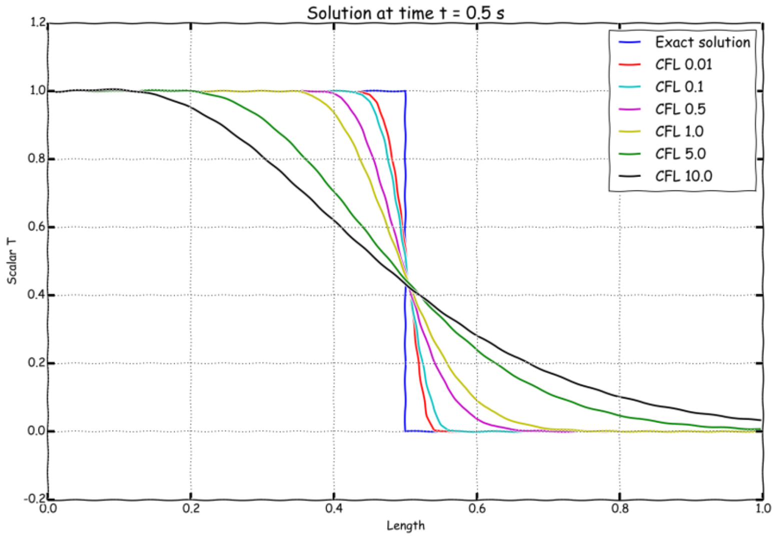


Comparison of Crank Nicolson blending factor using cellLimited leastSquares 0.5 gradient limiter.
Linear upwind in space – 100 cells – CFL 0.1



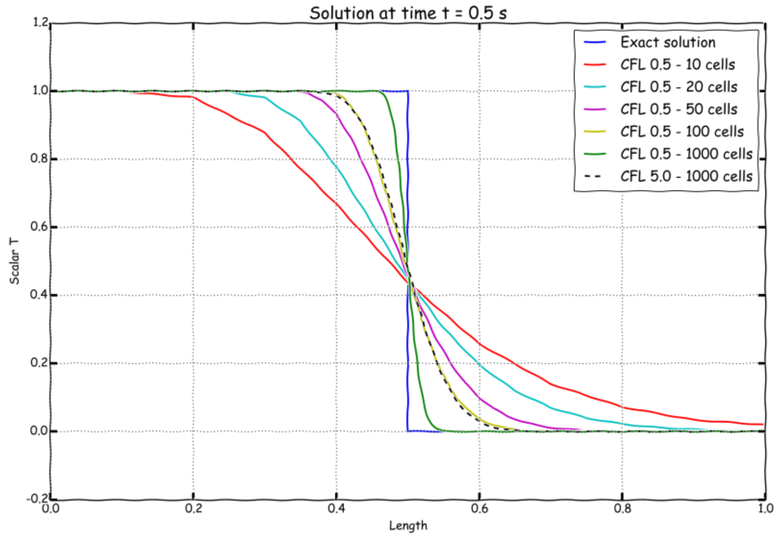
Numerical playground

Comparison of different time-step size (different CFL number).
Linear upwind in space – Euler in time – 100 cells



$$CFL = \frac{u\Delta t}{\Delta x}$$

Comparison of different mesh sizes.
Linear upwind in space – Euler in time



Numerical playground

- This case was for your eyes and brain only, but we encourage you to reproduce all the previous results,
    - Use all the time discretization schemes.
    - Use all the spatial discretization schemes.
    - Use all the gradient discretization schemes.
    - Use gradient limiters.
    - Use different mesh resolution.
    - Use different time-steps.

- Sample the solution and compare the results.
- Try to find the best combination of numerical schemes.
- Remember, in the *README.FIRST* file you will find the instructions of how to run the case.

Numerical playground

Exercises

- Which one of the following schemes is useless: **upwind, downwind, or linear**
- Compare the solution obtained with the following schemes: **upwind**, **linearUpwind**, **MUSCL**, **QUICK**, **cubic**, **UMIST**, **OSHER**, **Minmod**, **vanAlbada.** Are all of them bounded? Are they second order accurate?
- Use the **linearUpwind** method with **Gauss linear**, **Gauss pointLinear** and **leastSquares** for gradient computations, which method is more accurate?
- Imagine that you are using the **linearUpwind** method with no gradient limiters. How will you stabilize the solution if it becomes unbounded?
- When using gradient limiters, what is clipping?
- Use the **linearUpwind** with different gradient limiters. Which method is more unbounded?
- Use the **vanLeer** method with a CFL number of 0.1, 0.9 and 2, did all solutions converge? Are both solutions bounded?
- In the directory **`tri_mesh`**, you will find the same case setup using a triangular mesh.
    - Run the case and compare the solution with the equivalent setup using the orthogonal mesh.
    - Repeat the same experiments as before and draw your conclusions about which method is better for unstructured meshes.
    - With unstructured meshes, is it possible to get the same accuracy level as for orthogonal meshes?

Numerical playground

Exercises

- The solver `scalarTransportFoam` does not report the CFL number on the screen. How will you compute the CFL number in this case?

  (Hint: you can take a look at the post-processing slides or the utilities directory)
- Which one is more diffusive, spatial discretization or time discretization?
- Are all time discretization schemes bounded?
- If you are using the Crank-Nicolson scheme, how will you avoid oscillations?
- Does the solution improve if you reduce the time-step?
- Use the **upwind** scheme and a really fine mesh. Does the accuracy of the solution improve?
- From a numerical point of view, what is the Peclet number? Can it be compared to the Reynolds number?

$$Pe = \frac{LU}{D} = \frac{\text{convection effects}}{\text{diffusion effects}}$$

- If the Peclet number is more than 2, what will happen with your solution if you were using a **linear** scheme?

  (Hint: to change the Peclet number you will need to change the diffusion coefficient)
- Pure convection problems have analytical solutions. You are asked to design your own tutorial with an analytical solution in 2D or 3D.
- Try to break the solver using a time step less than 0.005 seconds. You are allowed to modify the original mesh and use any combination of discretization schemes.

Numerical playground

Slide:

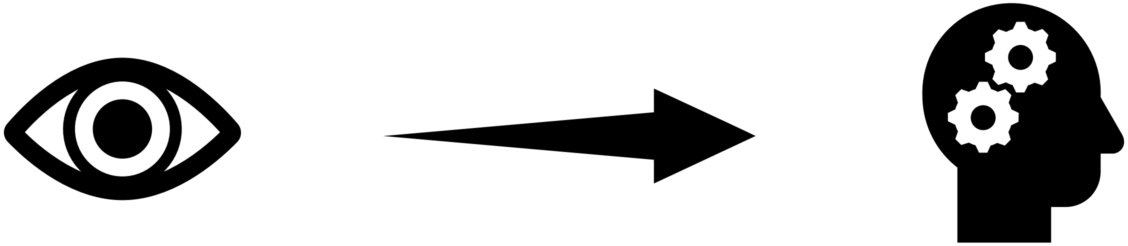
2D Laplace equation in a square domain.

Numerical playground

- This is a visual and mental exercise only.
- You will find this case in the directory

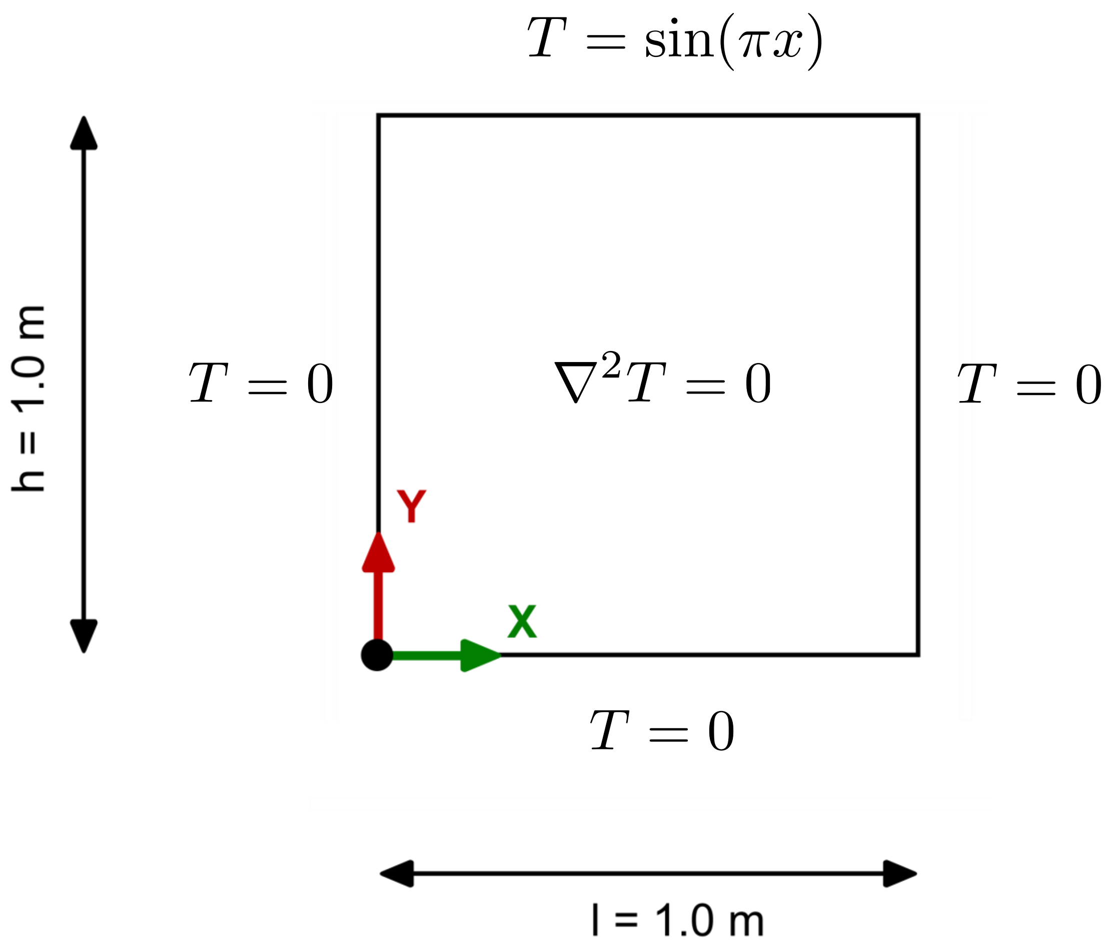
`$PTOFC/101FVM/laplace`

- In this directory, you will also find the *`README.FIRST`* file with the instructions of how to run the case.
- Hereafter, we will focus our eyes to train our brain.



Numerical playground

2D Laplace equation in a square domain

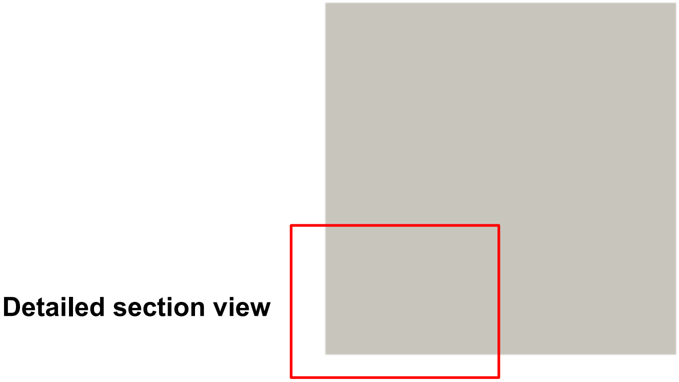


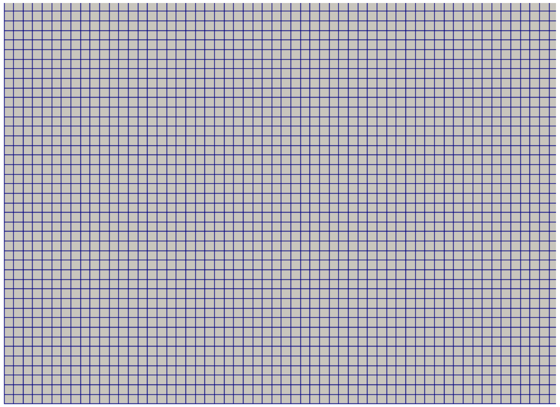
Numerical playground

2D Laplace equation in a square domain

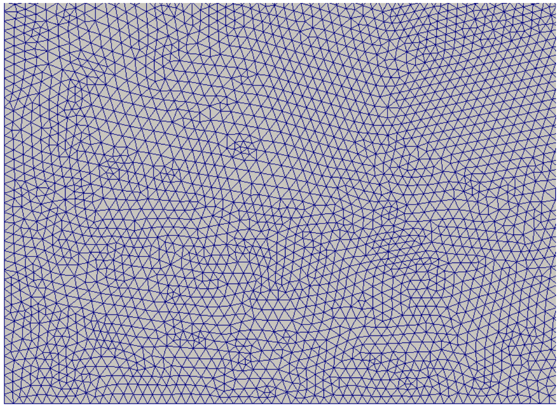
- This case consist of one domain and three different element types.

Domain

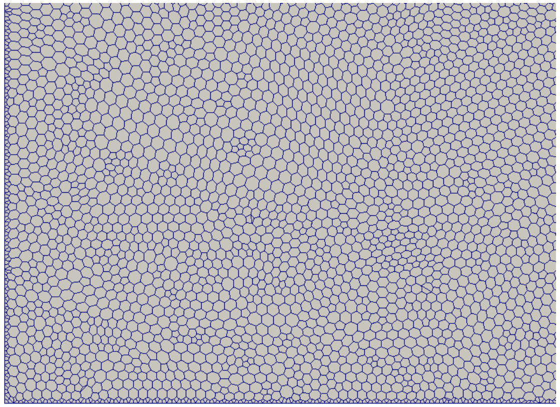




Hexahedral mesh



Triangular mesh

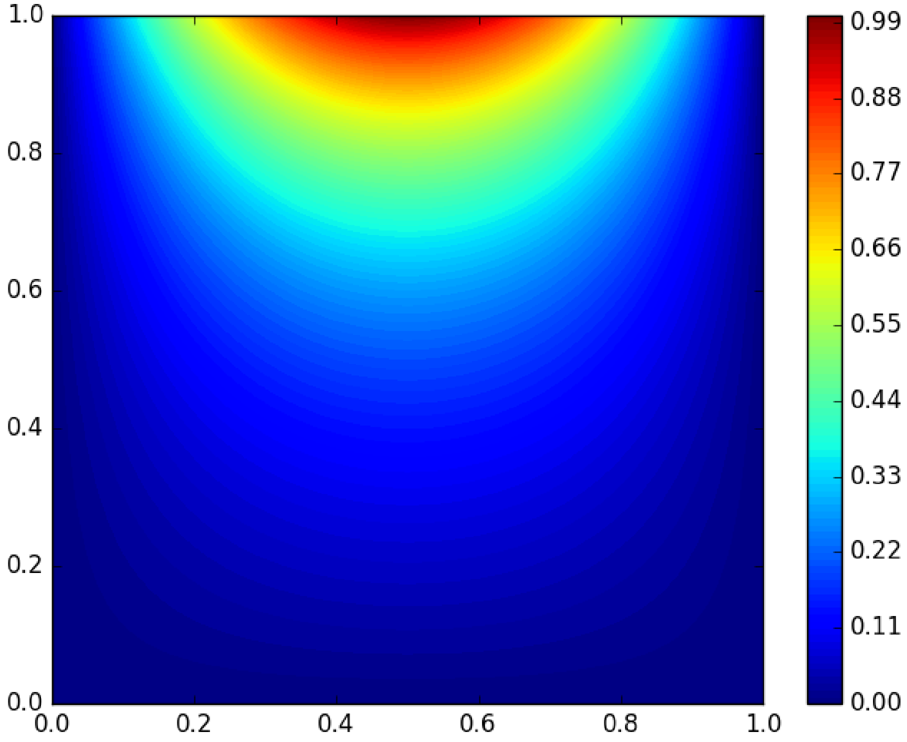


Polyhedral mesh

Numerical playground

2D Laplace equation in a square domain

- We will study the influence of the element type on the gradients computation.
- We will also study the influence of the **gradSchemes** method and **laplacianSchemes** method on the solution.

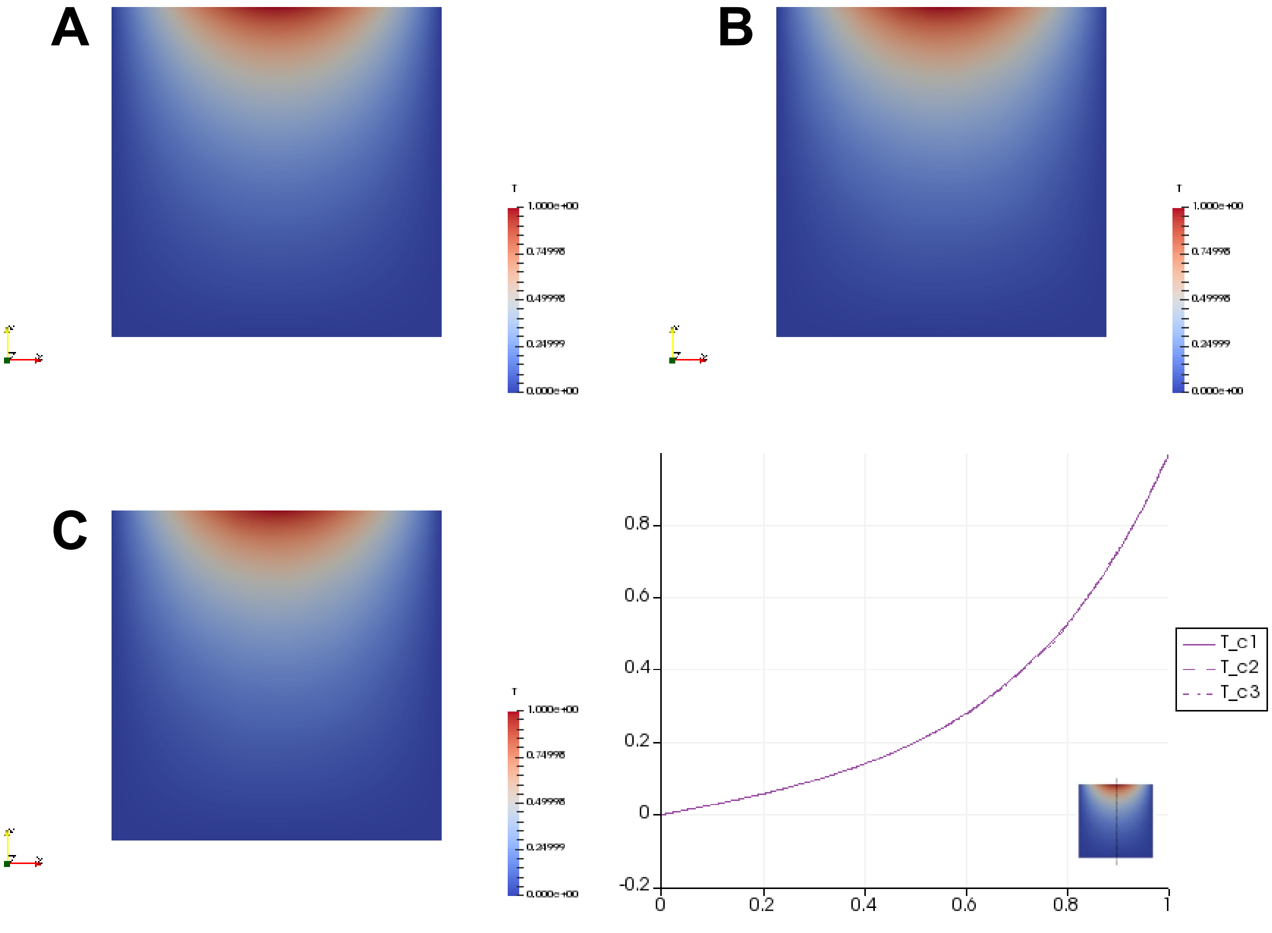


This problem has the following analytical solution:

$$T(x,y) = \frac{\sin(\pi x) \times \sinh(\pi y)}{\sinh(\pi)}$$

Numerical playground

2D Laplace equation in a square domain



T field

gradSchemes:
Gauss linear

laplacianSchemes:
Gauss linear orthogonal

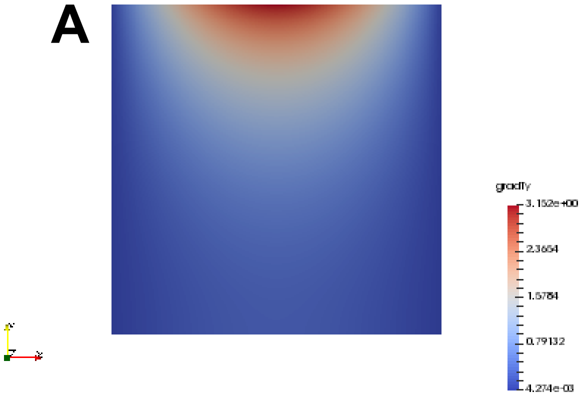
A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

- This is the actual solution.
- Each mesh gives basically the same solution.
- However, when we look at the information behind the field T, we will see a different outcome.
- Precisely, we will take a look at the gradients.

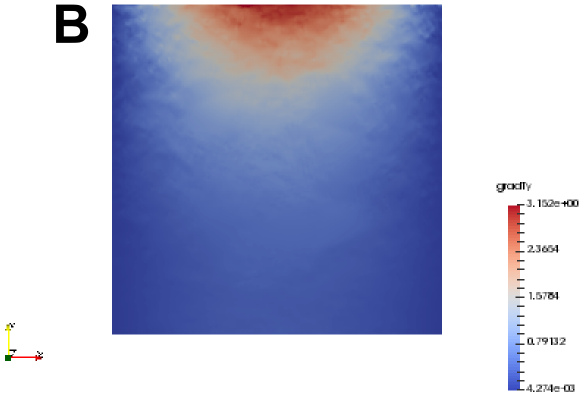
Numerical playground

2D Laplace equation in a square domain

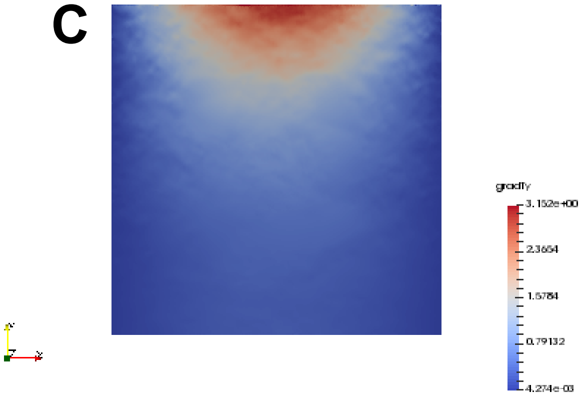
A

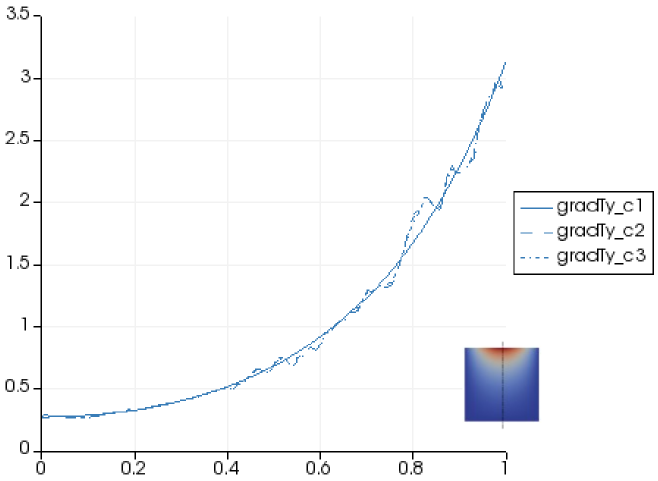


B



C





$\mathrm{grad}_y(T)$ **field**

gradSchemes:
Gauss linear

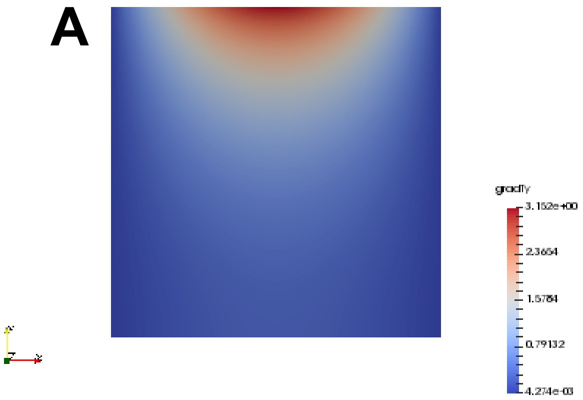
laplacianSchemes:
Gauss linear orthogonal

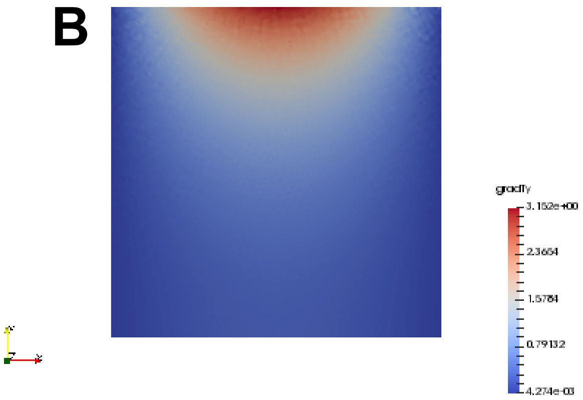
A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

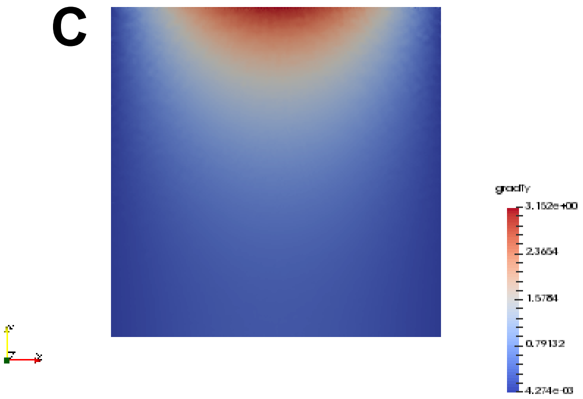
- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- For the method used in this case, the gradients on the unstructured meshes are noisy.

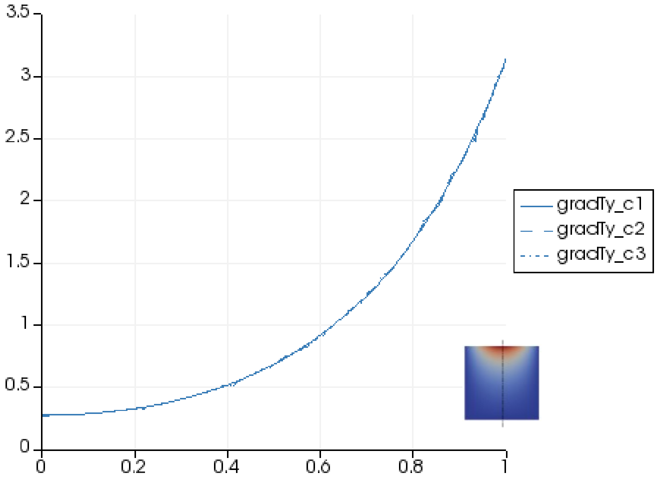
Numerical playground

2D Laplace equation in a square domain









$\mathrm{grad}_y(T)$ **field**

gradSchemes:
Gauss linear

laplacianSchemes:
Gauss linear limited 1

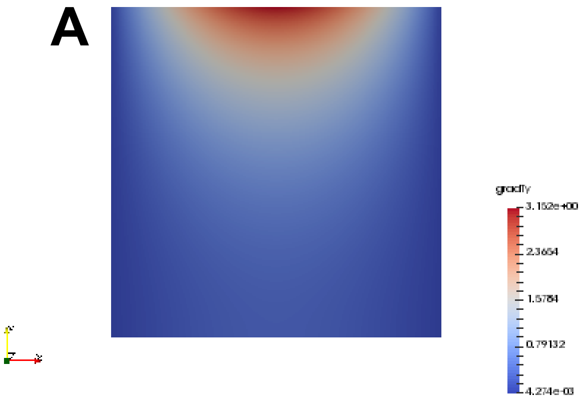
A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- By adjusting the numerics, we can smooth the gradients.
- All meshes show similar gradients.

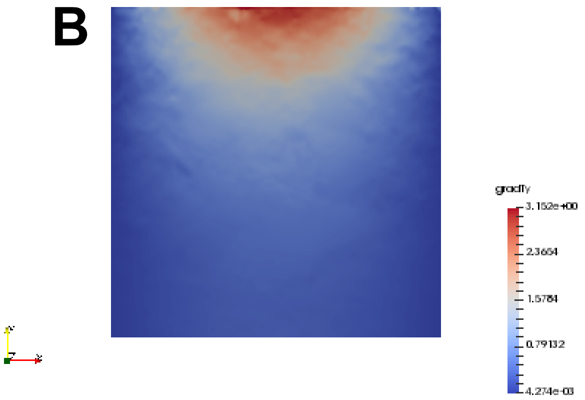
Numerical playground

2D Laplace equation in a square domain

A



B

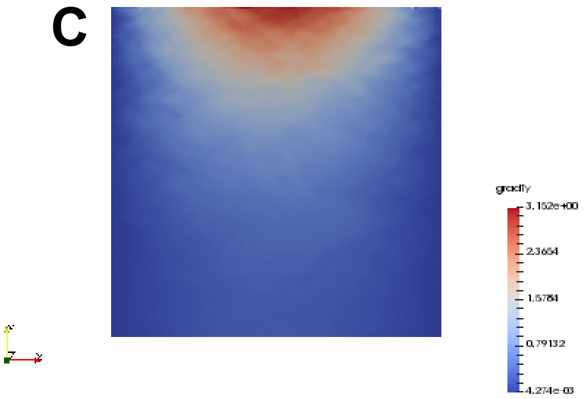


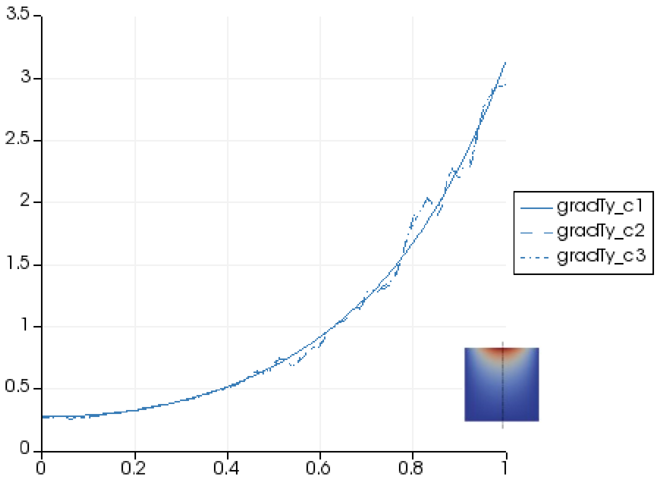
gradSchemes:
Gauss leastSquares

laplacianSchemes:
Gauss linear orthogonal

A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

C





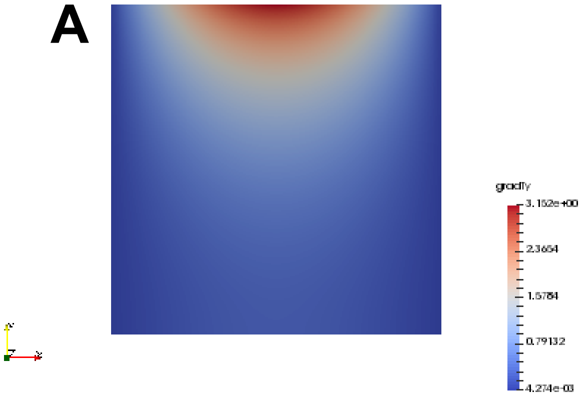
$\mathrm{grad}_y(T)$ **field**

- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- For the method used in this case, the gradients on the unstructured meshes are noisy.

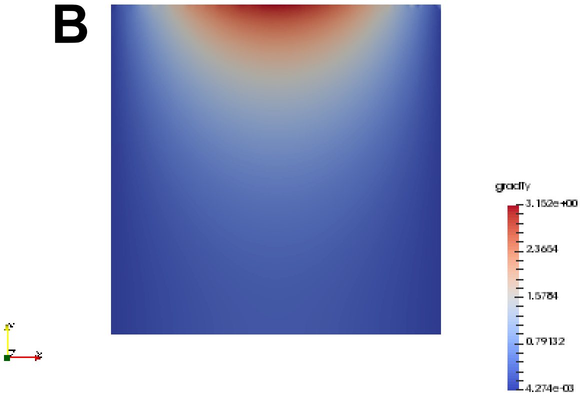
Numerical playground

2D Laplace equation in a square domain

A



B

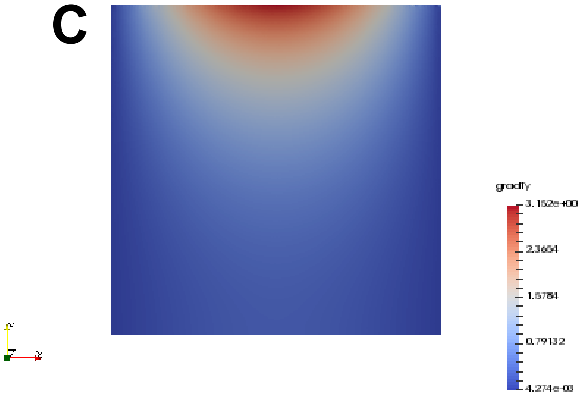


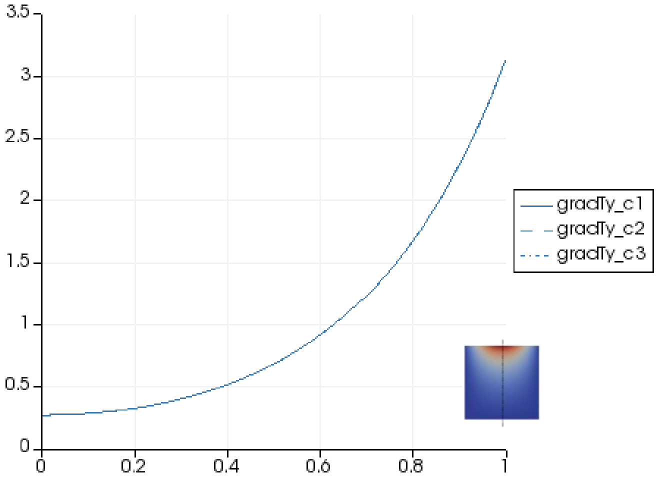
gradSchemes:
Gauss leastSquares

laplacianSchemes:
Gauss linear limited 1

A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

C





$\mathrm{grad}_y(T)$ **field**

- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- By adjusting the numerics, we can smooth the gradients.
- All meshes show similar gradients.

Numerical playground

- This case was for your eyes and brain only, but we encourage you to reproduce all the previous results.
- In the subdirectory **`c1`** you will find the hexahedral mesh, in the subdirectory **`c2`** you will find the triangular mesh, and in the subdirectory **`c3`** you will find the polyhedral mesh.
- Use the script `runallcases.sh` to run all the cases automatically.
- When launching `paraFoam` it will give you a warning, accept the default option (yes).
- In `paraFoam`, go to the `File` menu and select `Load State`. Load the state located in the directory **`paraview`** (*`state1.pvsm`*).
- In the window that pops out, give the location of the *`*.foam`* files inside each subdirectory (*`c1/c1.foam`*, *`c2/c2.foam`*, and *`c3/c3.foam`*).
- The file *`state1.pvsm`* will load a preconfigured state with all the solutions.
- If you are interested in running the cases individually, enter the subdirectory and follow the instructions in the *`README.FIRST`* file.

Numerical playground

Exercises

- Run the case using all gradient discretization schemes available. Which scheme gives the best results?
- According to the previous results, which element type is the best one? Do you think that the choice of the element type is problem dependent (e.g., direction of the flow)?
- Use the **leastSquares** method for gradient discretization, and the **corrected** and **uncorrected** method for Laplacian discretization. Do you get the same results in all the meshes? How can you improve the results?

  (Hint: look at the corrections)
- Does it make sense to do more non-orthogonal corrections using the **uncorrected** method?
- Run a case only 1 iteration. Do you get a converged solution? Is there a difference between 1 and 100 iterations? Compare the solutions.
- Use a different interpolation method for the diffusion coefficient. Do you get the same results?
- Try to break the solver (this is a difficult task in this case). You are allowed to modify the original mesh and use any combination of discretization schemes.

Numerical playground

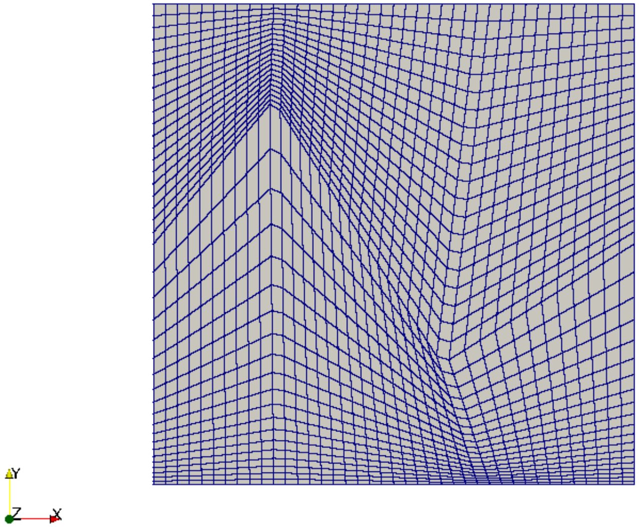
Swing:

Flow in a lid-driven square cavity – Re = 100

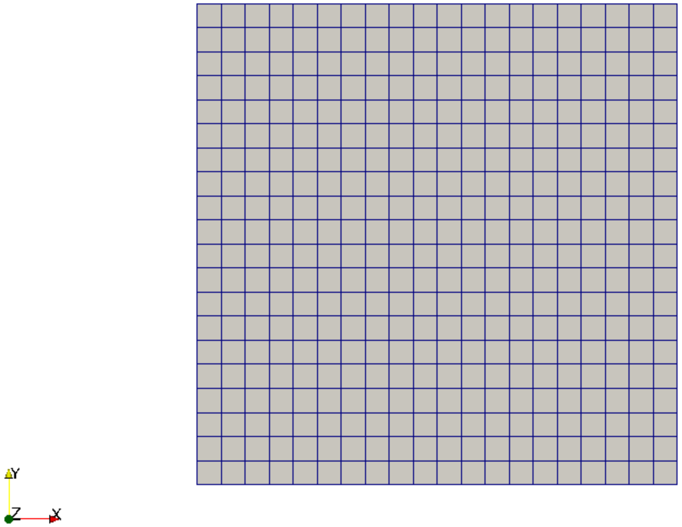
Effect of grading and non-orthogonality on the accuracy of the solution

Numerical playground

Flow in a lid-driven square cavity – Re = 100
Non-orthogonal mesh vs. orthogonal mesh



Non-orthogonal mesh
The overall quality of this mesh is good (in terms of non-orthogonality and skewness), <u>but by no standard this is a good mesh.</u>

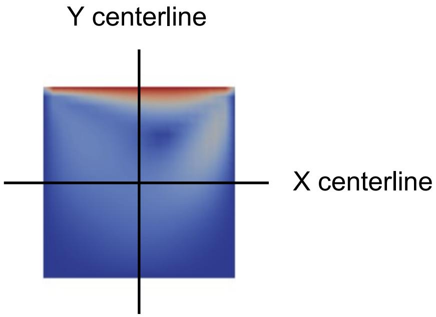


Orthogonal mesh
This is a perfect mesh

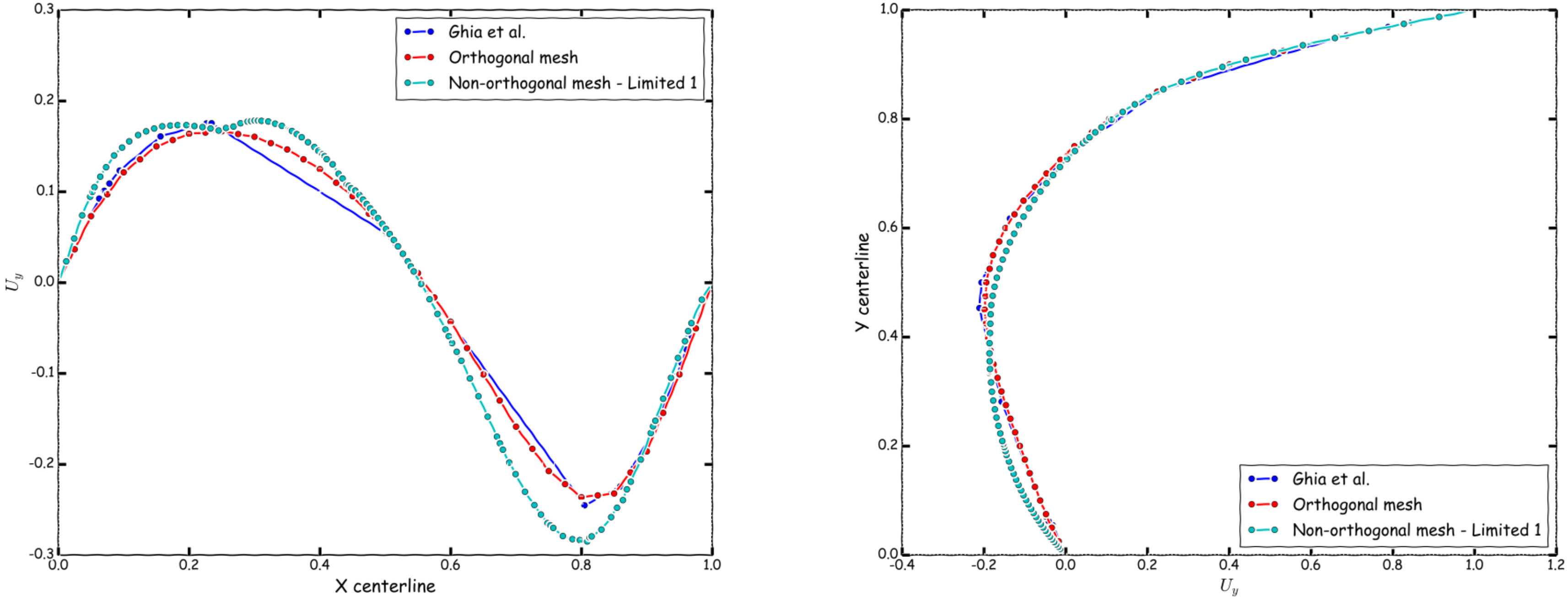
- Often people refer to these non-orthogonal meshes as Kershaw distorted meshes.
- We will use this case to learn how to adjust the numerical schemes according to mesh non-orthogonality and grading.

Numerical playground

LaplacianSchemes orthogonal – Non-orthogonal corrections disabled



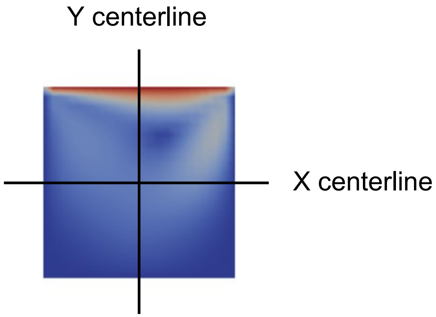
- And as CFD is not only about pretty colors, we should also validate the results



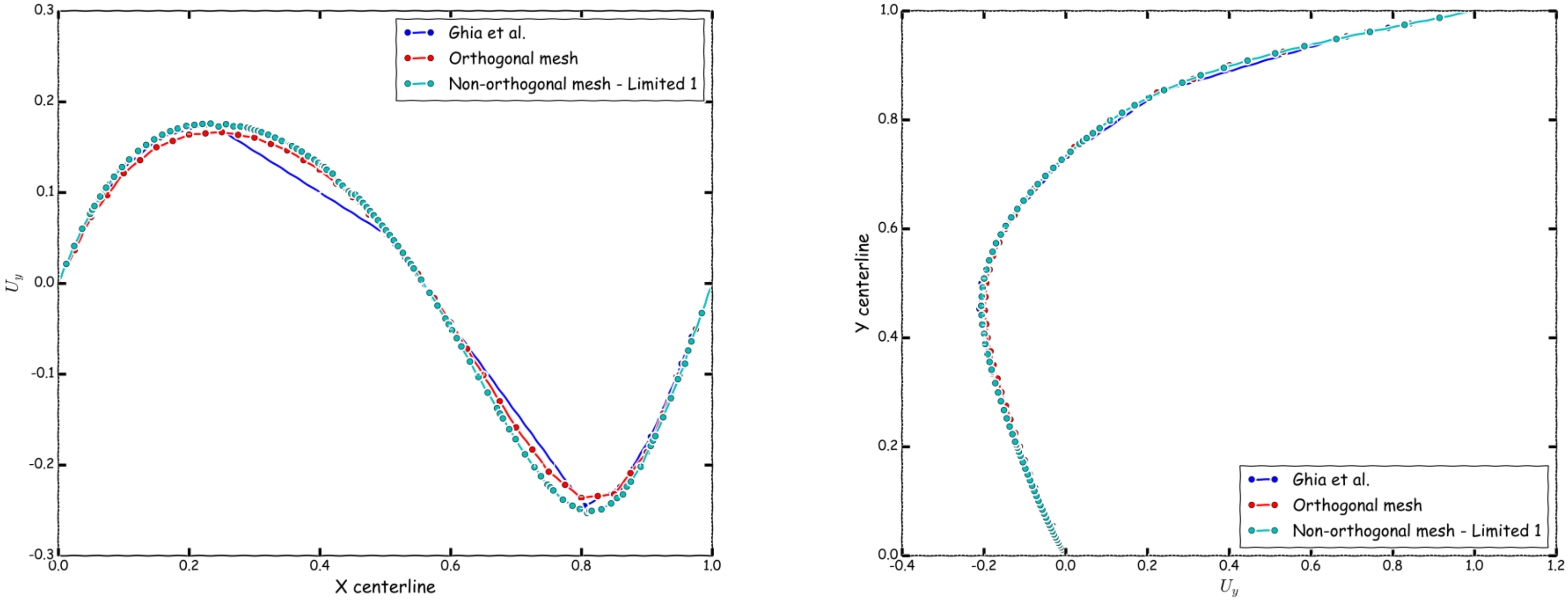
High-Re Solutions for incompressible flow using the navier-stokes equations and a multigrid method
U. Ghia, K. N. Ghia, C. T. Shin.
Journal of computational physics, 48, 387-411 (1982)

Numerical playground

LaplacianSchemes orthogonal – Non-orthogonal corrections enabled



- And as CFD is not only about pretty colors, we should also validate the results



High-Re Solutions for incompressible flow using the navier-stokes equations and a multigrid method
U. Ghia, K. N. Ghia, C. T. Shin.
Journal of computational physics, 48, 387-411 (1982)

Numerical playground

How to adjust the numerical method to deal with non-orthogonality

```
ddtSchemes
{
    default          backward;
}

gradSchemes
{
    default          Gauss linear;
    //default          Gauss skewCorrected linear;
    //default          cellMDLimited Gauss linear 1;
    grad(p)          Gauss linear;
}

divSchemes
{
    default          none;
    //div(phi,U)       Gauss linearUpwind default;
    div(phi,U)       Gauss linear;
}

laplacianSchemes
{
    default          Gauss linear orthogonal;
    //default          Gauss linear limited 1;
    //default          Gauss skewCorrected linear limited 1;
}

interpolationSchemes
{
    //default           skewCorrected linear;
    default          linear;
}

snGradSchemes
{
    default          orthogonal;
    //default          limited 1;
}
```

- In the dictionary *fvSchemes* we can enable non-orthogonal corrections.
- Non-orthogonal corrections are chosen using the keywords **laplacianSchemes** and **snGradSchemes**.
- These are the **laplacianSchemes** and **snGradSchemes** schemes that you will use most of the times:
  - **orthogonal:** second order accurate, bounded on perfect meshes, without non-orthogonal corrections.
  - **corrected:** second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
  - **limited** ψ**:** second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
  - **uncorrected:** second order accurate, without non-orthogonal corrections. Stable but more diffusive than limited and corrected.

Numerical playground

How to adjust the numerical method to deal with non-orthogonality

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

    pFinal
    {
        $p;
        relTol           0;
    }

    U
    {
        solver           smoothSolver;
        smoother         symGaussSeidel;
        tolerance        1e-08;
        relTol           0;
    }
}

PISO
{
    nCorrectors     1;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}
```

- Additionally, in the dictionary *fvSolution* we need to define the number of **PISO** corrections (**nCorrectors**) and non-orthogonal corrections (**nNonOrthogonalCorrectors**).
- You need to do at least one **PISO** correction. Increasing the number of **PISO** correctors will improve the stability and accuracy of the solution at a higher computational cost.
- For orthogonal meshes, 1 **PISO** correction is ok. But as most of the time you will deal with non-orthogonal meshes, doing 2 **PISO** corrections is a good choice.
- If you are using a method with non-orthogonal corrections (**corrected** or **limited 1-0.5**), you need to define the number of non-orthogonal corrections (**nNonOrthogonalCorrectors**).
- If you use 0 **nNonOrthogonalCorrectors**, you are computing the initial approximation using central differences (accurate but unstable), with no explicit correction.
- To take into account the non-orthogonality of the mesh, you will need to increase the number of corrections (you get better approximations using the previous correction).
- Usually 2 **nNonOrthogonalCorrectors** is ok.

Numerical playground

- We will now illustrate a few of the discretization schemes available in OpenFOAM® using a model case.
- We will use the lid-driven square cavity case to study the effect of grading and non-orthogonality on the accuracy of the solution
- This case is located in the directory:

```
$PTOFC/101FVM/nonorthoCavity/
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Numerical playground

What are we going to do?

- This is the same case as the one we used during the first tutorial session.
- The only difference is that we have modified the mesh a little bit in order to add grading and non-orthogonality.
- After generating the mesh, we will use the utility `checkMesh` to control the quality of the mesh. Is it a good mesh?
- We will use this case to learn how to adjust the numerical schemes according to mesh non-orthogonality and grading.
- After finding the numerical solution we will do some sampling and plotting.

Running the case

- You will find this tutorial in the directory **`$PTOFC/101FVM/nonorthoCavity`**
- In the terminal window type:

```
1.  $> foamCleanTutorials
2.  $> blockMesh -dict system/blockMeshDict.0
3.  $> checkMesh
4.  $> pisoFoam | log.solver
5.  $> postProcess -func sampleDict -latestTime
6.  $> gnuplot gnuplot/gnuplot_script
7.  $> paraFoam
```

Numerical playground

To run the case, follow these steps

- First run the case using the original dictionaries. Did it crash right?
- Now change the **laplacianSchemes** and **snGradSchemes** to **limited 1**. It crashed again but this time it ran a few more time-steps, right?
- Now increase the number of **nNonOrthogonalCorrectors** to 2. It crashed again but it is running more time-steps, right?
- Now increase the number of **PISO** corrections to 2 (**nCorrectors**). Did it run?
- Basically we enabled non-orthogonal corrections, we computed better approximations of the gradients, and we increased the number of **PISO** corrections to get better predictions of the field variables (**U** and **p**).
- Now set the number of **nNonOrthogonalCorrectors** to 0. Did it crash right? This is telling us that the mesh is sensitive to the gradients.
- Now change the **laplacianSchemes** and **snGradSchemes** to **limited 0** (uncorrected). In this case we are not using non-orthogonal corrections, therefore there is no need to increase the value of **nNonOrthogonalCorrectors**.
- We are using a method that uses a wider stencil to compute the Laplacian, this method is more stable but a little bit more diffusive. Did it run?
- At this point, compare the solution obtained with corrected and uncorrected schemes. Which one is more diffusive?

Numerical playground

- When it comes to **laplacianSchemes** and **snGradSchemes** this is how we proceed most of the times (a robust setup),

```
laplacianSchemes
{
        default         Gauss linear limited 1;
}

snGradSchemes
{
        default         limited 1;
}
```

```
PISO
{
        nCorrectors     2;
        nNonOrthogonalCorrectors 1;
}
```

- This method works fine for meshes with non-orthogonality less than 75.
- If the non-orthogonality is more than 75, you should consider using **limited 0.5**, and increasing **nCorrectors** and **nNonOrthogonalCorrectors**.
- When the non-orthogonality is more than 85, the best solution is to redo the mesh.

Numerical playground

Exercises

- Using the non-orthogonal mesh and the original dictionaries, try to run the solver reducing the time-step. Do you get a solution at all?
- Try to get a solution using the method **limited 1** and two **nNonOrthogonalCorrectors** (leave **nCorrectors** equal to 1).

  (Hint: try to reduce the time-step)
- If you managed to get a solution using the previous numerical scheme. How long did it take to get the solution? Use the robust setup, clock the time and compare with the previous case. Which one is faster? Do you get the same solution?
- Instead of using the non-orthogonal mesh, use a mesh with grading toward all edges. How will you stabilize the solution?

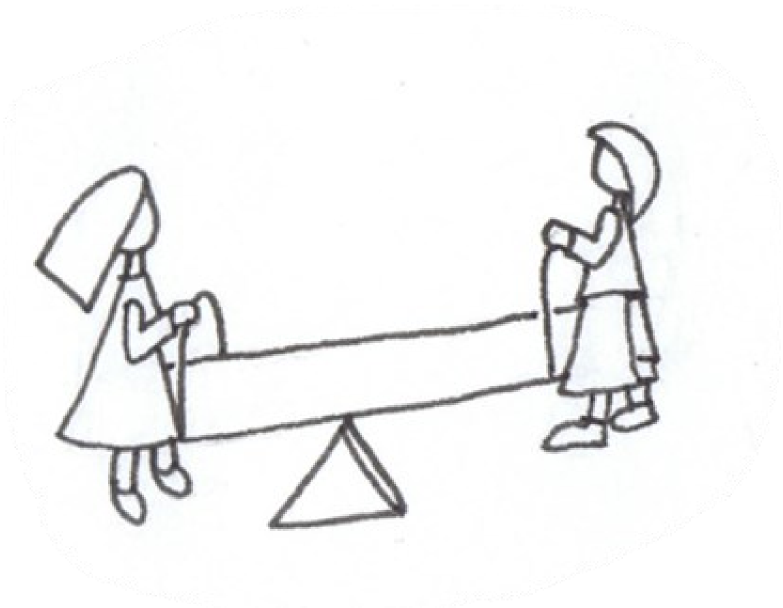
  (Hint: take a look at the blockMesh slides in order to add grading to the mesh)
- Try to get a solution using a time-step of 0.05 seconds. Use the original discretization schemes for the gradient and convective terms.

  (Hint: increase nCorrectors and nNonOrthogonalCorrectors)
- Using the uniform orthogonal mesh and a robust numerics, determined the largest CFL you can use. Is the solution still accurate? What about the clock-speed?
- Try to break the solver and interpret the output screen. You are allowed to modify the original mesh and use any combination of discretization schemes.

Numerical playground

Seesaw:

Sod's shock tube.



Numerical playground

Sod’s shock tube

- This case has an analytical solution and plenty of experimental data.
- This is an extreme test case used to test solvers.
- Every single commercial and open source solver use this case for validation of the numerical schemes.
- The governing equation of this test case are the Euler equations.

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

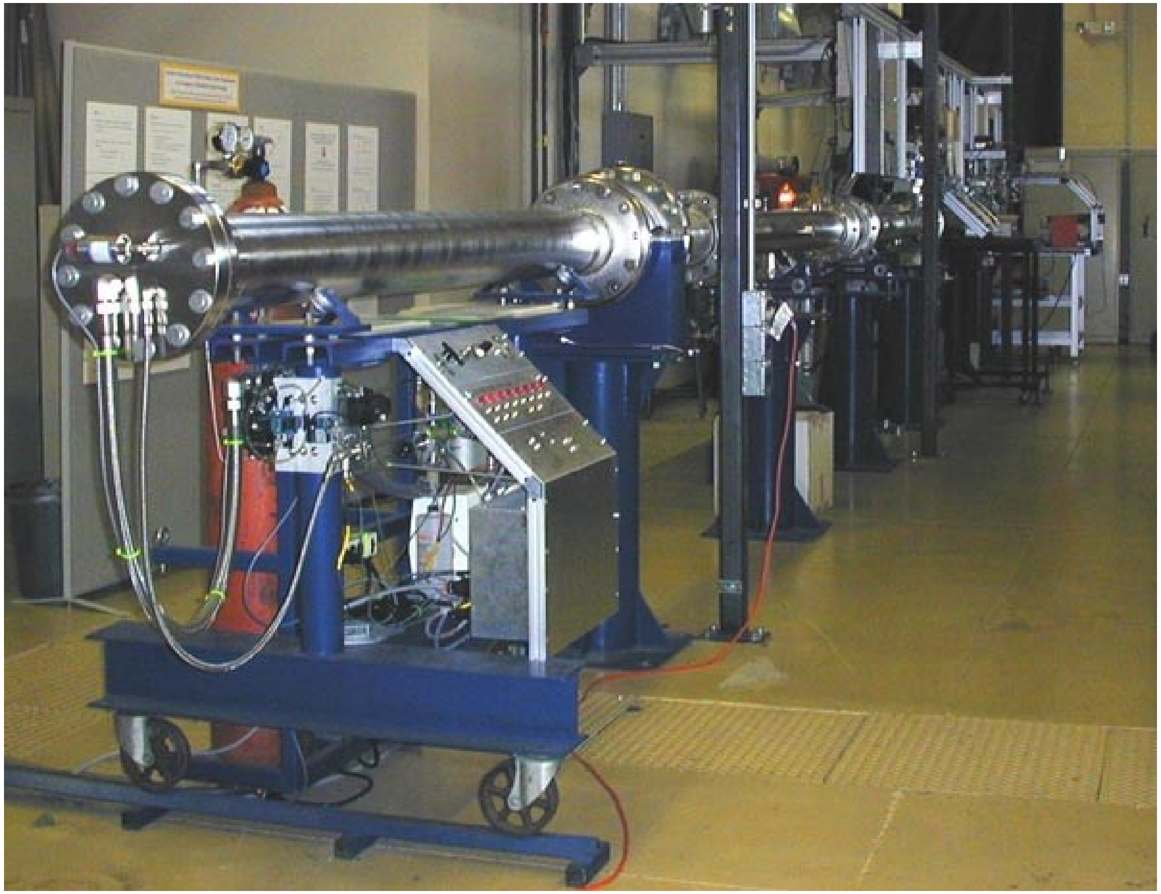
$$\frac{\partial (\rho \mathbf{U})}{\partial t} + \nabla \cdot (\rho \mathbf{U}\mathbf{U}) + \nabla p = 0$$

$$\frac{\partial (\rho e_t)}{\partial t} + \nabla \cdot (\rho e_t \mathbf{U}) + \nabla \cdot (p \mathbf{U}) = 0$$

$$p = \rho R_g T$$

Numerical playground

High Purity Photolysis Shock Tube (NASA Tube)

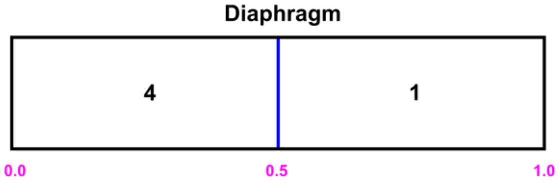


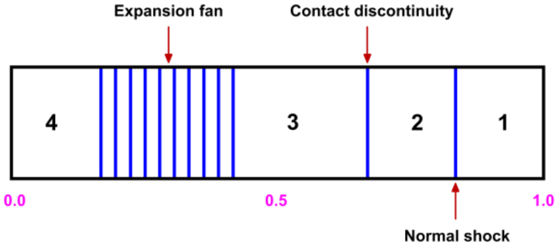
Shock tube. The driver section, including vacuum pumps, controls, and helium driver gas.

Photo credit: Stanford University. http://hanson.stanford.edu/index.php?loc=facilities_nasa

Numerical playground

Sod’s shock tube



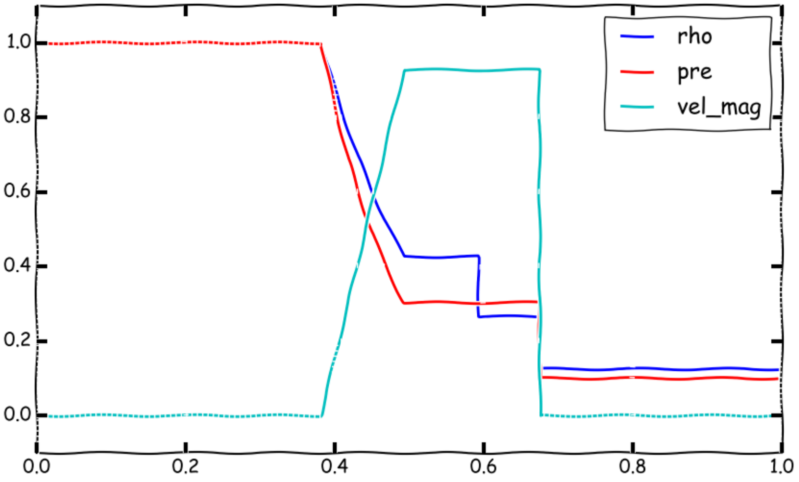


All walls are slip

$$\mathbf{U_4} = \mathbf{U_1} = 0$$

$$p_4 = 1, \;\; p_1 = 0.1$$

$$T_4 = 0.00348, \;\; T_1 = 0.00278$$

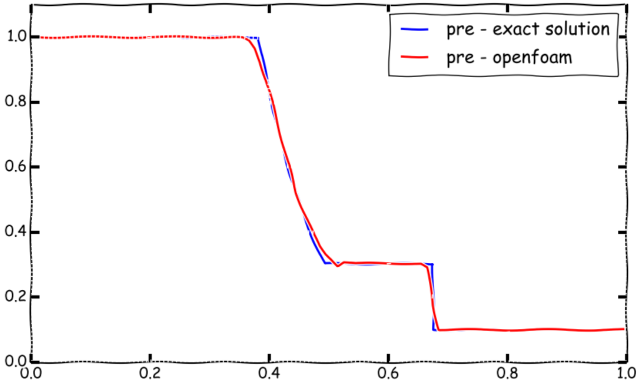


Boundary conditions and initial conditions

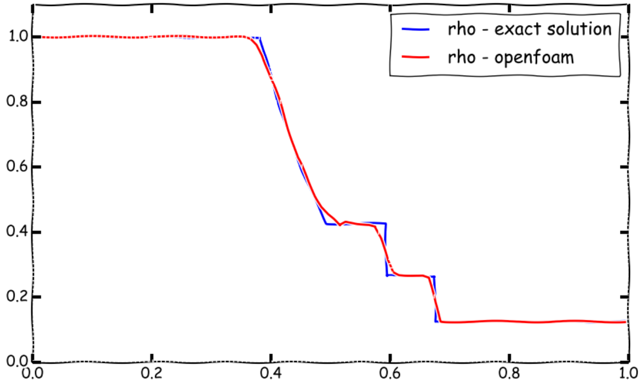
Analytical solution

Numerical playground

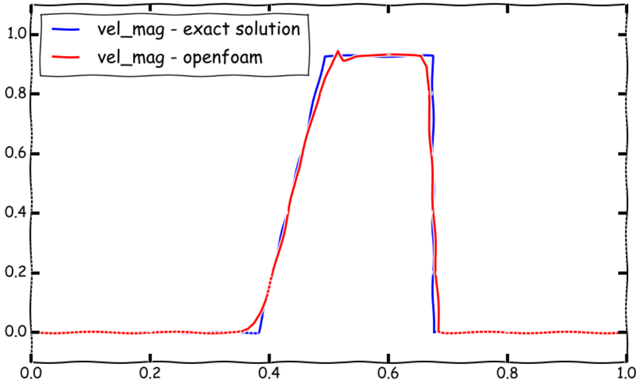
Sod's shock tube



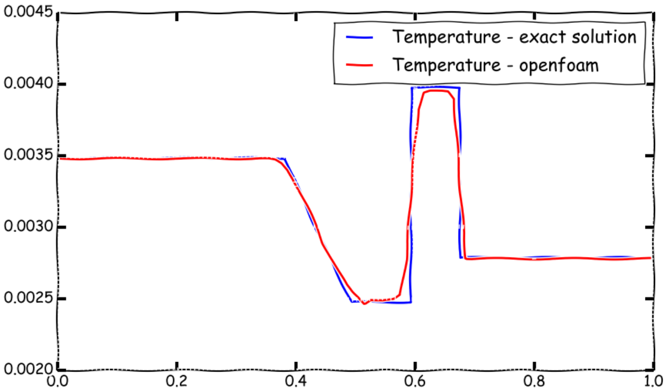
Pressure field



Density field



Velocity magnitude field



Temperature field

Numerical playground

- We will now illustrate a few of the discretization schemes available in OpenFOAM® using a severe model case.
- We will use the Sod’s shock tube case.
- This case is located in the directory:

```
$PTOFC/101FVM/shockTube/
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Numerical playground

What are we going to do?

- Now is your turn.
- You are asked to select the best discretization scheme for the physics involve.
- Remember the following concepts: accuracy, stability and boundedness.
- We will compare your numerical solution with the analytical solution.
- At this point, we are very familiar with the numerical schemes. It is up to you to choose the best setup.
- You can start using the original dictionaries.
- To find the numerical solution we will use the solver `rhoPimpleFoam`.
- `rhoPimpleFoam` is a transient solver for laminar or turbulent flow of a compressible gas.
- After finding the numerical solution we will do some sampling.
- At the end, we will do some plotting (using gnuplot or Python) and scientific visualization.

Numerical playground

Running the case

- You will find this tutorial in the directory **`$PTOFC/101FVM/schockTube`**
- In the terminal window type:

```
1.   $> foamCleanTutorials
2.   $> blockMesh
3.   $> checkMesh
4.   $> rm -rf 0
5.   $> cp -r 0_org 0
6.   $> setFields
7.   $> rhoPimpleFoam | tee log.solver
8.   $> postProcess -func sampleDict -latestTime
9.   $> paraFoam
```

- To plot the analytical solution against the numerical solution, go to the directory **`python`** and run the Python script.
- In the terminal window type:

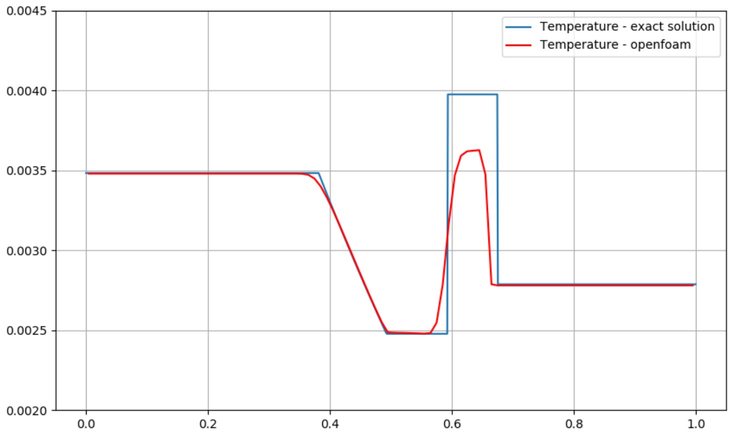
```
1.   $> python3 python/sodshocktube.py
```

- The Python script will save four .png files with the solution. Feel free to explore and adapt the Python script to your needs.
- Python (version 3) must be installed in order to use the script

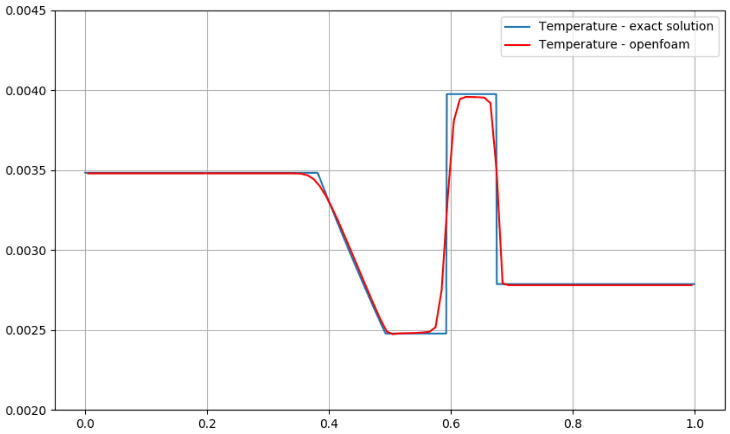
Numerical playground

Running the case

- If you used the values proposed in the dictionaries, the solution diverged, right? Try to get the case working.
  Hint: look at the gradient limiters.
- By adjusting the gradient limiters, the case will run, but the final solution is not very accurate. How can you increase the accuracy of the solution?
  Hint: look at the PIMPLE corrections.



Not so accurate solution



Accurate solution

Numerical playground

Exercises

- Using the proposed case setup, try to get an accurate solution by reducing the time-step or refining the mesh. Did you succeed in getting an accurate solution?
- Run the case using different time discretization schemes.
- Run the case using different gradient discretization schemes.
- Run the case using different convective discretization schemes for the term **div(phi,U)**.
- Run the case using different convective discretization schemes for the terms **div(phi,e)** and **div(phi,K)**. What are the variables **e** and **K**?
- Extend the case to 2D and 3D. Do you get the same solution?
- Try to run a 2D case using a triangular mesh and adjust the numerical scheme to get an accurate and stable solution.
- Try to run the 1D case using an explicit solver. For the same CFL number, do you have the same time step size as for the implicit solver?

  (Hint: look for the solver with the word Central)
- Try to break the solver (this is extremely easy in this case). You are allowed to modify the original mesh and use any combination of discretization schemes.

Module 7

Highlights – Implementing boundary conditions and initial conditions using codeStream

Roadmap

1. **codeStream – Highlights**
2. Implementing boundary conditions using codeStream
3. Solution initialization using codeStream

codeStream – Highlights

codeStream – Boundary conditions

- There are many boundary conditions available in OpenFOAM®.
- But from time to time it may happen that you do not find what you are looking for.
- It is possible to implement your own boundary conditions, so in theory you can do whatever you want.
- Remember, you have the source code.
- To implement your own boundary conditions, you have three options:
    - Use **codeStream**.
    - Use high level programing.
    - Use an external library (*e.g.*, **swak4foam**).
- **codeStream** is the simplest way to implement boundary conditions, and most of the times you will be able to code boundary conditions with no problem.
- If you can not implement your boundary conditions using **codeStream**, you can use high level programming. However, this requires some knowledge on C++ and OpenFOAM® API.
- Hereafter, we are going to work with **codeStream** and basic high-level programming.
- We are not going to work with **swak4Foam** because it is an external library that is not officially supported by the OpenFOAM® foundation. However, it works very well and is relatively easy to use.

codeStream – Highlights

codeStream – Initial conditions

- When it comes to initial conditions, you can use the utility `setFields`.
- This utility is very flexible, you can even read STL files and use them to initialize fields.
- But again, it may happen that you can not get the desired results.
- As for boundary conditions, to implement your own initials conditions you have three options:
  - Use **codeStream**.
  - Use high level programing.
  - Use an external library (e.g., **swak4foam**).
- **codeStream** is the simplest way to implement initial conditions, and most of the times you will be able to code initial conditions with no problem.
- If you can not implement your initial conditions using **codeStream**, you can use high level programming. However, this requires some knowledge on C++ and OpenFOAM® API.
- Hereafter, we are going to work only with **codeStream**.
- Using high level programming is a little bit trickier, and we guarantee you that 99.9% of the times **codeStream** will work.
- We are not going to work with **swak4Foam** because it is an external library that is not officially supported by the OpenFOAM® foundation. However, it works very well and is relatively easy to use.

codeStream – Highlights

- Hereafter we will work with **codeStream**, which will let us program directly in the input dictionaries.
- With **codeStream**, we will implement our own boundary conditions and initial conditions without going thru the hustle and bustle of high-level programming.
- If you are interested in high level programming, refer to the supplements.
- In the supplemental slides, we address the following topics:
    - Building blocks, implementing boundary conditions using high level programming, modifying applications, implementing an application from scratch, and adding the scalar transport equation to icoFoam.
- High level programming requires some knowledge on C++ and OpenFOAM® API library.
- This is the hard part of programming in OpenFOAM®.
- Before doing high level programming, we highly recommend you try with **codeStream**, most of the time it will work.
- Also, before modifying solvers or trying to implement your own solvers, understand the theory behind the FVM.
- Remember, you can access the API documentation in the following link, https://cpp.openfoam.org/v8 ℹ

Roadmap

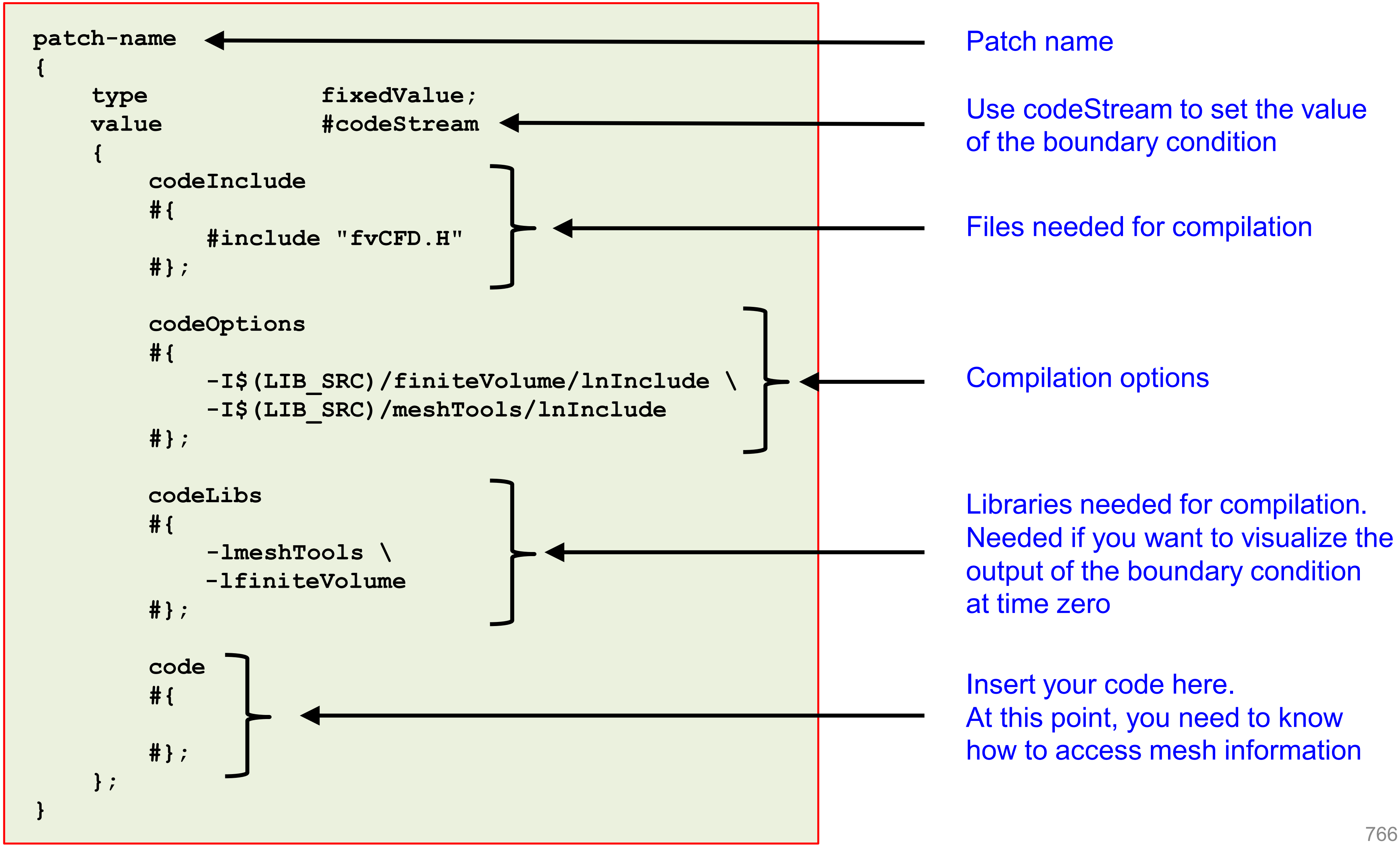
1. ~~codeStream – Highlights~~
2. **Implementing boundary conditions using codeStream**
3. Solution initialization using codeStream

Implementing boundary conditions using codeStream

- OpenFOAM® includes the capability to compile, load and execute C++ code at run-time.
- This capability is supported via the directive **#codeStream**, that can be used in any input file for run-time compilation.
- This directive reads the entries **code** (compulsory), **codeInclude** (optional), **codeOptions** (optional), and **codeLibs** (optional), and uses them to generate the dynamic code.
- The source code and binaries are automatically generated and copied in the directory **`dynamicCode`** of the current case.
- The source code is compiled automatically at run-time.
- The use of **codeStream** is a very good alternative to avoid high level programming of boundary conditions or the use of external libraries.
- Hereafter we will use **codeStream** to implement new boundary conditions.
- Have in mind that **codeStream** can be used in any dictionary.

Implementing boundary conditions using codeStream

Body of the codeStream directive for boundary conditions



```
patch-name
{
    type            fixedValue;
    value           #codeStream
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Patch name

Use codeStream to set the value of the boundary condition

Files needed for compilation

Compilation options

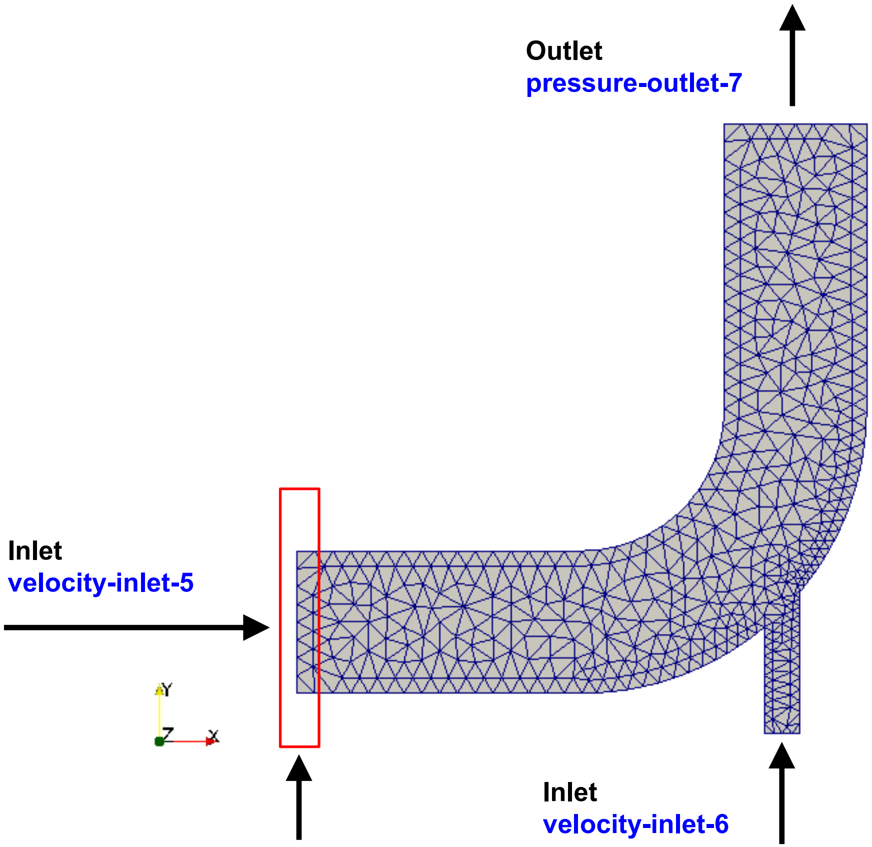
Libraries needed for compilation.
Needed if you want to visualize the output of the boundary condition at time zero

Insert your code here.
At this point, you need to know how to access mesh information

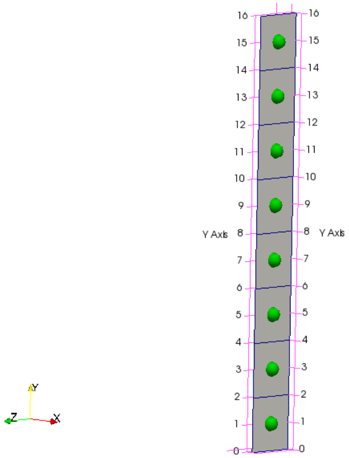
Implementing boundary conditions using codeStream

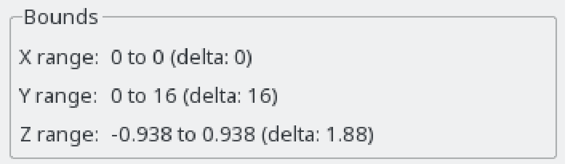
Implementation of a parabolic inlet profile using codeStream

- Let us implement a parabolic inlet profile.
- The firs step is identifying the patch, its location and the dimensions.
- You can use paraview to get all visual references.



Parabolic inlet profile





Bounds

X range: 0 to 0 (delta: 0)

Y range: 0 to 16 (delta: 16)

Z range: -0.938 to 0.938 (delta: 1.88)

Bounds of velocity-inlet-5 boundary patch

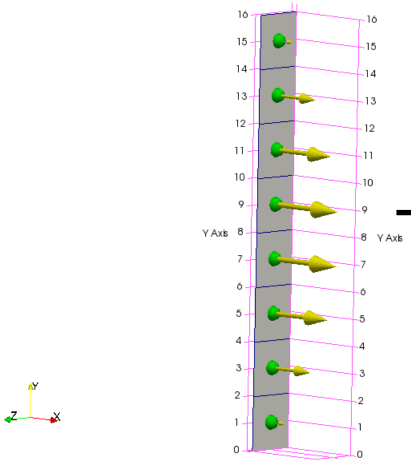
Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using codeStream

- We will use the following formula to implement the parabolic inlet profile

$$U_{max}\left(1.0-\frac{(y-c)^2}{r^2}\right)$$

- For this specific case *c* is the patch midpoint in the y direction (8), *r* is the patch semi-height or radius (8) and *Umax* is the maximum velocity.
- We should get a parabolic profile similar to this one,



$$\longrightarrow \quad U_{max}\left(1.0-\frac{(y-8)^2}{64}\right)$$

Implementing boundary conditions using codeStream

- The **codeStream** BC in the body of the file *U* is as follows,

```
velocity-inlet-5
{
    type                fixedValue;
    value               #codeStream
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Patch name

Depending of what are you trying to do, you will need to add new files, options and libraries.

For most of the cases, this part is always the same.

Insert your code here.
At this point, you need to know how to access mesh information

Implementing boundary conditions using codeStream

- The **code** section of the **codeStream** BC in the body of the file *U* is as follows,

```
1     code
2     #{
3         const IOdictionary& d = static_cast<const IOdictionary&>
4         (
5             dict.parent().parent()
6          );
7
8          const fvMesh& mesh = refCast<const fvMesh>(d.db());
9          const label id = mesh.boundary().findPatchID("velocity-inlet-5");
10         const fvPatch& patch = mesh.boundary()[id];
11
12         vectorField U(patch.size(), vector(0, 0, 0));
13
14         ...
15         ...
16         ...
17     #};
```

To access boundary mesh information

Remember to update this value with the actual name of the patch

- Lines 3-11, are always standard, they are used to access boundary mesh information.
- In lines 3-6 we access the current dictionary.
- In line 8 we access the mesh database.
- In line 9 we get the label id (an integer) of the patch **velocity-inlet-5** (notice that you need to give the name of the patch).
- In line 10 using the label id of the patch, we access the boundary mesh information.
- In line 12 we initialize the vector field. The statement **patch.size()** gets the number of faces in the patch, and the statement **vector(0, 0, 0)** initializes a zero-vector field in the patch.

Implementing boundary conditions using codeStream

- The **code** section of the **codeStream** BC in the body of the file *U* is as follows,

```
1     code
2     #{
3         ...
4         ...
5         ...
6         const scalar pi = constant::mathematical::pi;
7         const scalar U_0    = 2.;  //maximum velocity
8         const scalar p_ctr = 8.;  //patch center
9         const scalar p_r    = 8.;  //patch radius
10
11        forAll(U, i)          //equivalent to for (int i=0; patch.size()<i; i++)
12        {
13            const scalar y = patch.Cf()[i][1];
14            U[i] = vector(U_0*(1-(pow(y - p_ctr,2))/(p_r*p_r)), 0., 0.);
15        }
16
17        writeEntry(os, "", U);
18    #};
```

Index used to access the y coordinate

0 → x

1 → y

2 → z

Assign input profile to vector field U (component x)

$$U_{max}\left(1.0 - \frac{(y-8)^2}{64}\right)$$

- In lines 6-17 we implement the new boundary condition.
- In lines 6-9 we declare a few constants needed in our implementation.
- In lines 11-15 we use a forAll loop to access the boundary patch face centers and to assign the velocity profile values. Notice the U was previously initialized.
- In line 13 we get the y coordinates of the patch faces center.
- In line 14 we assign the velocity value to the patch faces center.
- In line 17 we write the U values to the dictionary.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using codeStream

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_BC/2Delbow_UparabolicInlet`**
- To run the case, type in the terminal,

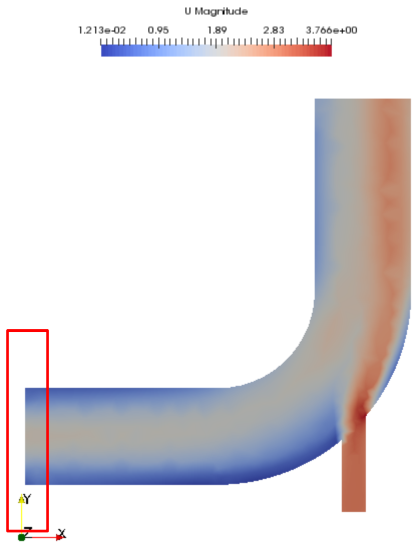
```
1.   $> foamCleanTutorials
2.   $> fluentMeshToFoam ../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh
3    $> checkMesh
4.   $> rm -rf 0
5.   $> cp -r 0_org 0
6.   $> pisoFoam | tee log.solver
7.   $> paraFoam
```

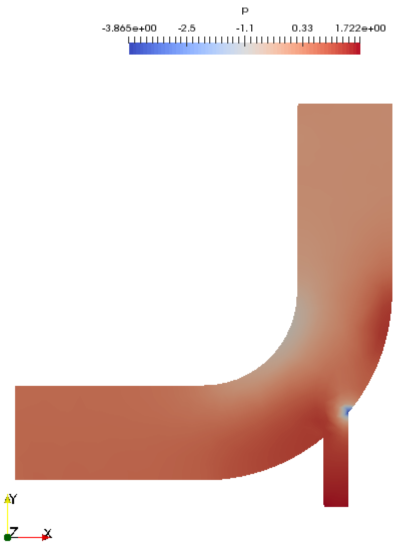
- The **codeStream** boundary condition is implemented in the file *0/U*.
- FYI, you can run in parallel with no problem.

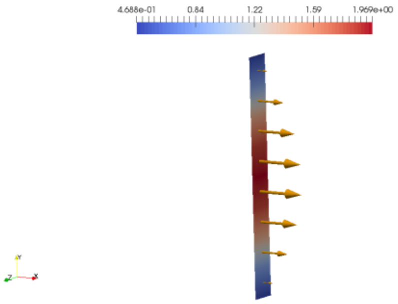
Implementing boundary conditions using codeStream

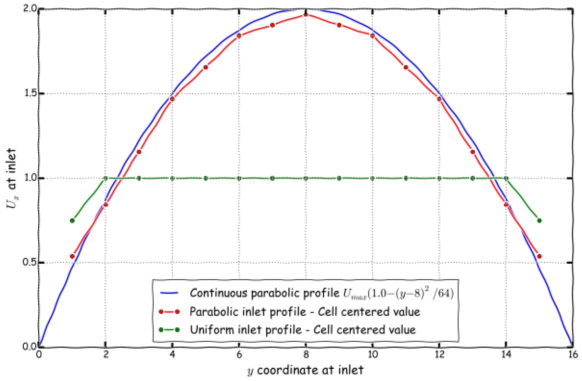
Implementation of a parabolic inlet profile using codeStream

- If everything went fine, you should get something like this,









Implementing boundary conditions using codeStream

codeStream works with scalar and vector fields

- We just implemented the input parabolic profile using a vector field.
- You can do the same using a scalar field, just proceed in a similar way.
- Remember, now we need to use scalars instead of vectors.
- And you will also use an input dictionary holding a scalar field.

```
code
#{
    ...
    ...
    ...
    scalarField S(patch.size(), scalar(0) );

    forAll(S, i)
    {
        const scalar y = patch.Cf()[i][1];
        S[i] = scalar( 2.0*sin(3.14159*y/8.) );
    }

    writeEntry(os, "", S);
#};
```

Initialize scalar field

Loop using scalar field size

Write profile values in scalar field

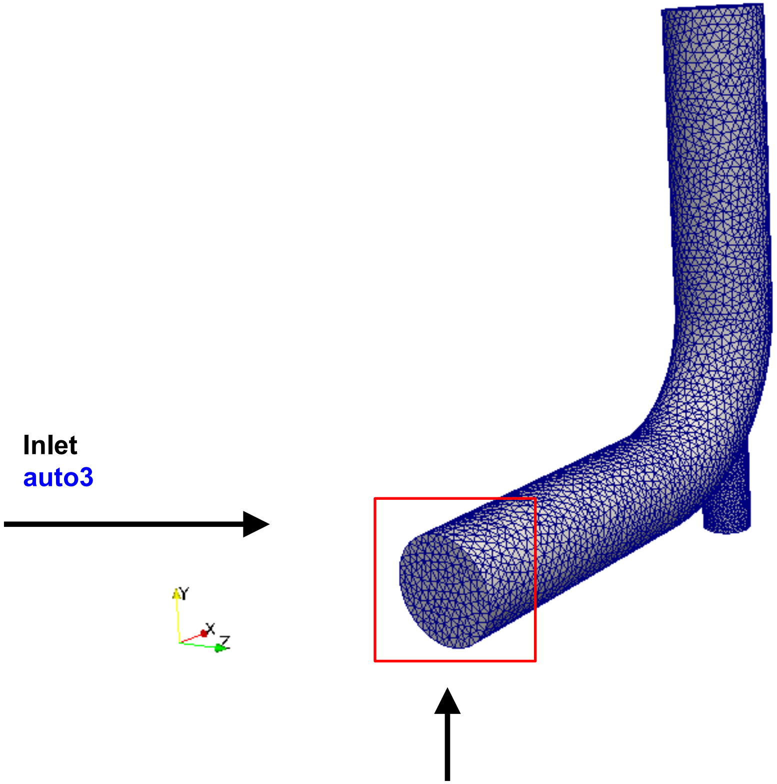
Write output to input dictionary

Notice that the name of the field does not need to be the same as the name of the input dictionary

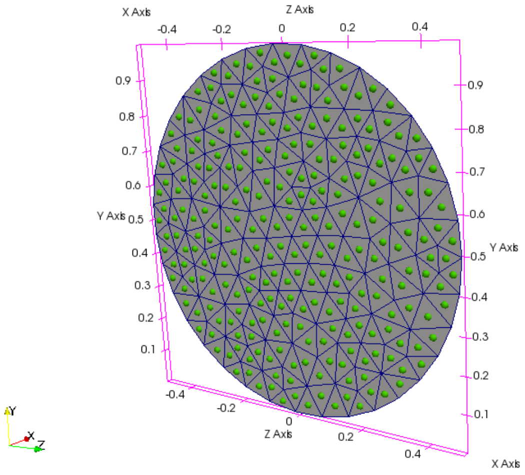
Implementing boundary conditions using codeStream

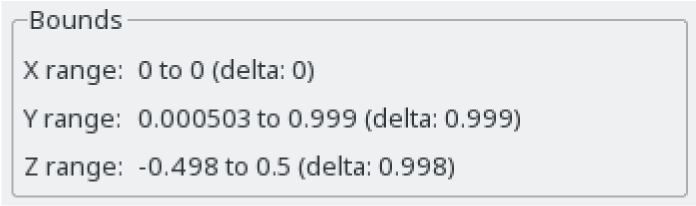
Implementation of a paraboloid inlet profile using codeStream

- Let us work in a case a little bit more complicated, a paraboloid input profile.
- As usual, the first step is to get all the spatial references.



Paraboloid inlet profile





Bounds

X range: 0 to 0 (delta: 0)

Y range: 0.000503 to 0.999 (delta: 0.999)

Z range: -0.498 to 0.5 (delta: 0.998)

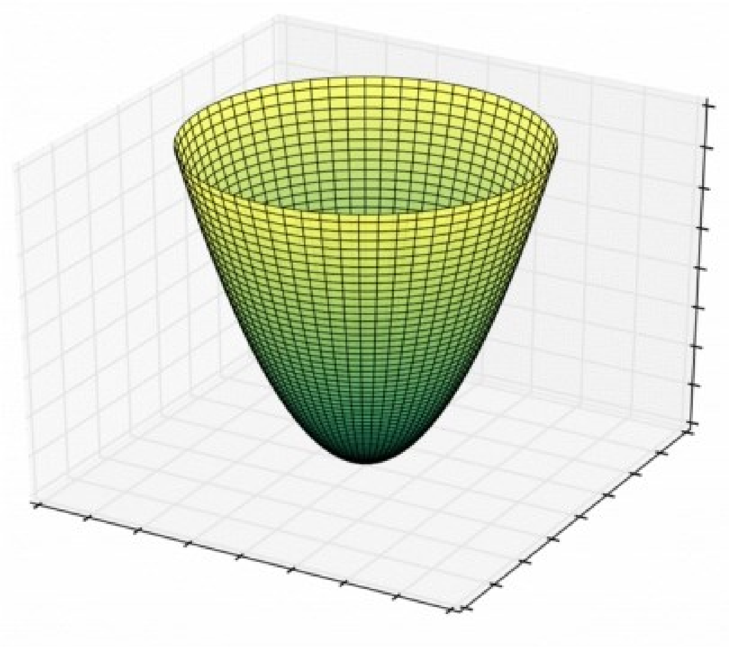
Bounds of auto3 boundary patch

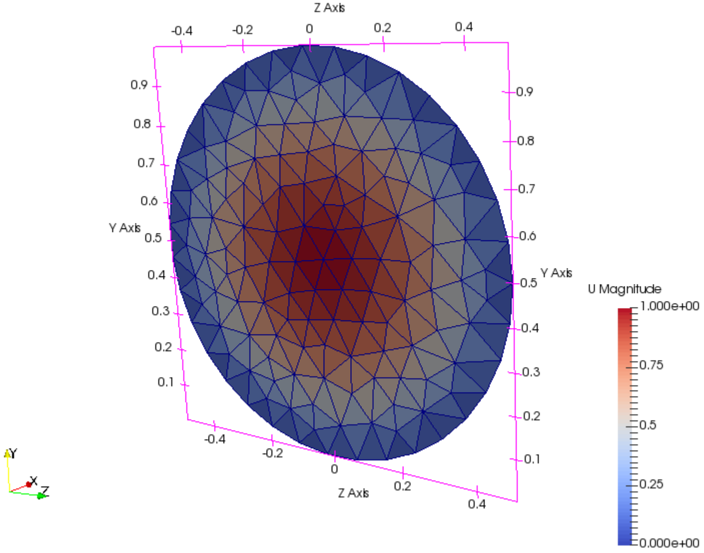
Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using codeStream

- We will implement the following equation in the boundary patch **auto3**.

$$U = \left(\frac{z}{0.5}\right)^2 + \left(\frac{y-0.5}{0.5}\right)^2 - 1$$





Implementing boundary conditions using codeStream

- The **codeStream** BC in the body of the file *U* is as follows,

```
auto3
{
    type            fixedValue;
    value           #codeStream
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Patch name

For most of the cases, this part is always the same. But depending of what are you trying to do, you will need to add more files, options and libraries.

Insert your code here.
We will implement the following equation

$$U = \left(\frac{z}{0.5}\right)^2 + \left(\frac{y-0.5}{0.5}\right)^2 - 1$$

Implementing boundary conditions using codeStream

- Hereafter, we only show the actual implementation of the **codeStream** boundary condition.
- The rest of the body is a template that you can always reuse. Including the section of how to access the dictionary and mesh information.
- Remember, is you are working with a vector, you need to use vector fields. Whereas, if you are working with scalars, you need to use scalars fields.

```
code
#{
    ...
    ...
    ...
    vectorField U(patch.size(), vector(0, 0, 0) );

    const scalar s  = 0.5;

    forAll(U, i)
    {
        const scalar x = patch.Cf()[i][0];
        const scalar y = patch.Cf()[i][1];
        const scalar z = patch.Cf()[i][2];

        U[i] = vector(-1.*(pow(z/s, 2) + pow((y-s)/s,2) - 1.0), 0, 0);
    }

    writeEntry(os, "", U);
#};
```

Initialize vector field

Initialize scalar

Access faces center coordinates (x, y, and z)

$$U = \left(\frac{z}{0.5}\right)^2 + \left(\frac{y-0.5}{0.5}\right)^2 - 1$$

Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using codeStream

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_BC/3Delbow_Uparaboloid/`**
- To run the case, type in the terminal,

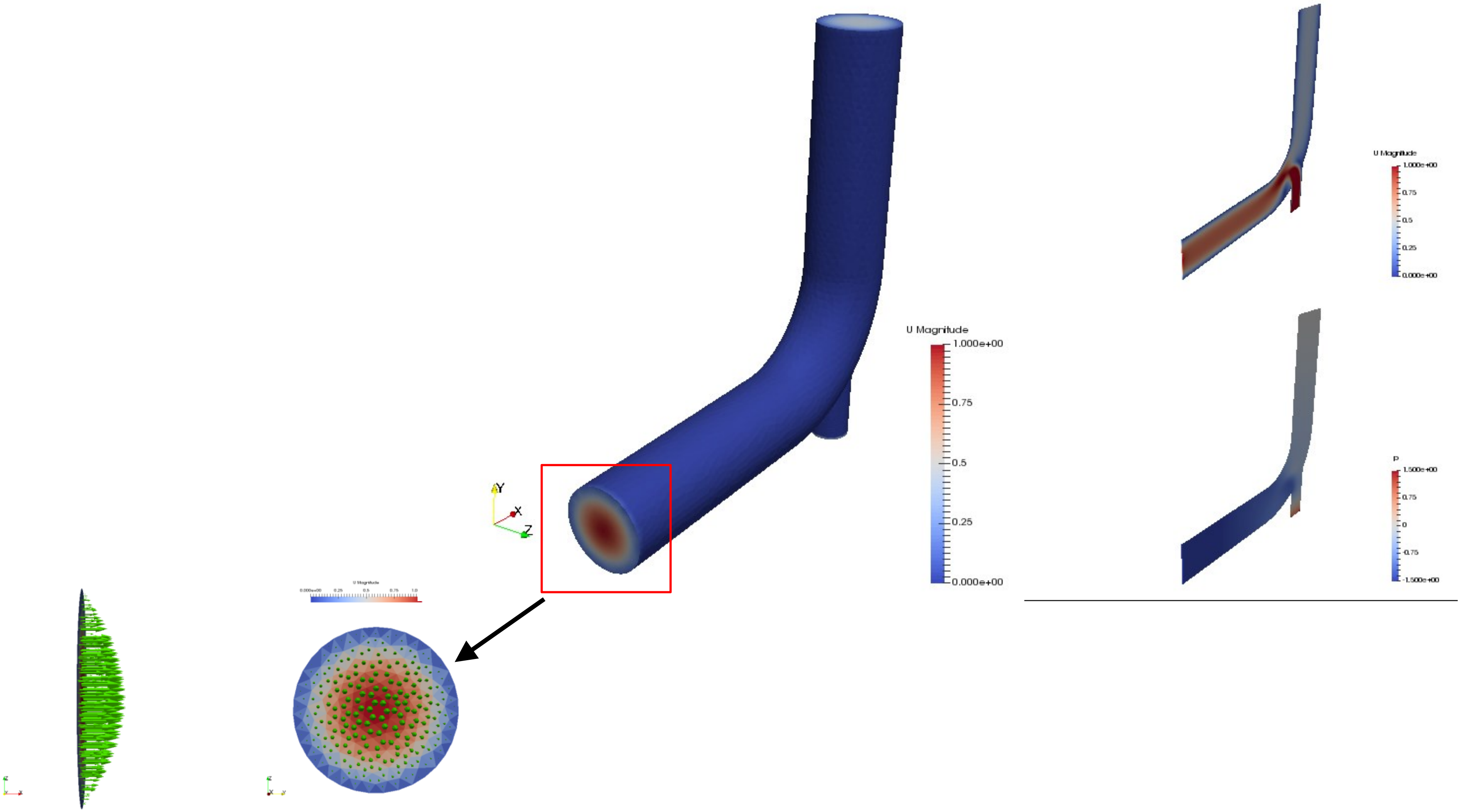
```
1.  $> foamCleanTutorials
2.  $> gmshToFoam ../../../meshes_and_geometries/gmsh_elbow3d/geo.msh
3.  $> autoPatch 75 -overwrite
4.  $> createPatch -overwrite
5.  $> renumberMesh -overwrite
6.  $> rm -rf 0
7.  $> cp -r 0_org 0
8.  $> pisoFoam | tee log.solver
9.  $> paraFoam
```

- The **codeStream** boundary condition is implemented in the file *0/U*.
- FYI, you can run in parallel with no problem.

Implementing boundary conditions using codeStream

Implementation of a paraboloid inlet profile using codeStream

- If everything went fine, you should get something like this,



Implementing boundary conditions using codeStream

codedFixedValue and codedMixed boundary conditions

- OpenFOAM® also includes the boundary conditions **codedFixedValue** and **codedMixed**.
- These boundary conditions are derived from **codeStream** and work in a similar way.
- They use a friendlier notation and let you access more information of the simulation database (e.g. time).
- The source code and binaries are automatically generated and copied in the directory **`dynamicCode`** of the current case.
- Another feature of these boundary conditions, is that the **code** section can be read from an external dictionary (*`system/codeDict`*), which is run-time modifiable.
- The boundary condition **codedMixed** works in similar way. This boundary condition gives you access to fixed values (Dirichlet BC) and gradients (Neumann BC).
- Let us implement the parabolic profile using **codedFixedValue**.

Implementing boundary conditions using codeStream

Body of the codedFixedValue boundary conditions

```
patch-name
{
    type                codedFixedValue;
    value               uniform (0 0 0);


    name  name_of_BC;

/*
    codeOptions
    #{
        -I$(LIB_SRC)/finiteVolume/lnInclude \
        -I$(LIB_SRC)/meshTools/lnInclude
    #};

    codeInclude
    #{
        #include "fvCFD.H"
        #include <cmath>
        #include <iostream>
    #};
*/

    code
    #{

    #};
}
```

Patch name

Use codedFixedValue and initializations

Unique name of the new boundary condition.
If you have more codedFixedValue BC, the names must be different

Optional compilation options.
You do not need to add these options unless it is required.
At the following link, you can find a bug related to codedFixedValue (SEP2020)
https://bugs.openfoam.org/view.php?id=3555

In this section we do the actual implementation of the boundary condition.
This is the only part of the body that you will need to change. The rest of the body is a template that you can always reuse.

Implementing boundary conditions using codeStream

- The **code** section of the **codeStream** BC in the body of the file U is as follows,

```
1     code
2     #{
3         const fvPatch& boundaryPatch = patch();
4         const vectorField& Cf = boundaryPatch.Cf();
5         vectorField& field = *this;
6
7         scalar U_0 = 2, p_ctr = 8, p_r = 8;
8
9         forAll(Cf, faceI)
10        {
11            field[faceI] = vector(U_0*(1-(pow(Cf[faceI].y()-p_ctr,2))/(p_r*p_r)),0,0);
12        }
13     #};
```

$$U_{max}\left(1.0 - \frac{(y-8)^2}{64}\right)$$

- Lines 3-5, are always standard, they give us access to mesh and field information in the patch.
- The coordinates of the faces center are stored in the vector field **Cf** (line 4).
- In this case, as we are going to implement a vector profile, we initialize a vector field where we are going to assign the profile (line 5).
- In line 7 we initialize a few constants that will be used in our implementation.
- In lines 9-12 we use a forAll loop to access the boundary patch face centers and to assign the velocity profile values.
- In line 11 we do the actual implementation of the boundary profile (similar to the **codeStream** case). The vector field was initialized in line 5.

Implementing boundary conditions using codeStream

codedFixedValue and codedMixed boundary conditions

- As you can see, the syntax and use of the **codedFixedValue** and **codedMixed** boundary conditions is much simpler than **codeStream**.
- You can use these instructions as a template.
- At the end of the day, you only need to modify the **code** section.
- Depending of what you want to do, you might need to add new headers and compilation options.
- Remember, is you are working with a vector, you need to use vector fields. Whereas, if you are working with scalars, you need to use scalars fields.
- One disadvantage of these boundary conditions, is that you can not visualize the fields at time zero. You will need to run the simulation for at least one iteration.
- On the positive side, accessing time and other values from the simulation database is straightforward.
- Time can be accessed by adding the following statement,

```
this->db().time().value()
```

Implementing boundary conditions using codeStream

- Let us add time dependency to the parabolic profile.

```
1     code
2     #{
3         const fvPatch& boundaryPatch = patch();
4         const vectorField& Cf = boundaryPatch.Cf();
5         vectorField& field = *this;
6
7         scalar U_0 = 2, p_ctr = 8, p_r = 8;
8
9         scalar t = this->db().time().value();
10
11        forAll(Cf, faceI)
12        {
13           field[faceI] = vector(sin(t)*U_0*(1-(pow(Cf[faceI].y()-p_ctr,2))/(p_r*p_r)),0,0);
14        }
15     #};
```

Time

Time dependency

$$sin(t) \times U_{max}\left(1.0 - \frac{(y-c)^2}{r^2}\right)$$

- This implementation is similar to the previous one, we will only address how to deal with time.
- In line 8 we access simulation time.
- In line 13 we do the actual implementation of the boundary profile (similar to the **codeStream** case). The vector field was initialized in line 5 and time is accessed in line 9.
- In this case, we added time dependency by simple multiplying the parabolic profile by the function **sin(t)**.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using codedFixedValue

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_BC/2Delbow_UparabolicInlet_timeDep`**
- To run the case, type in the terminal,

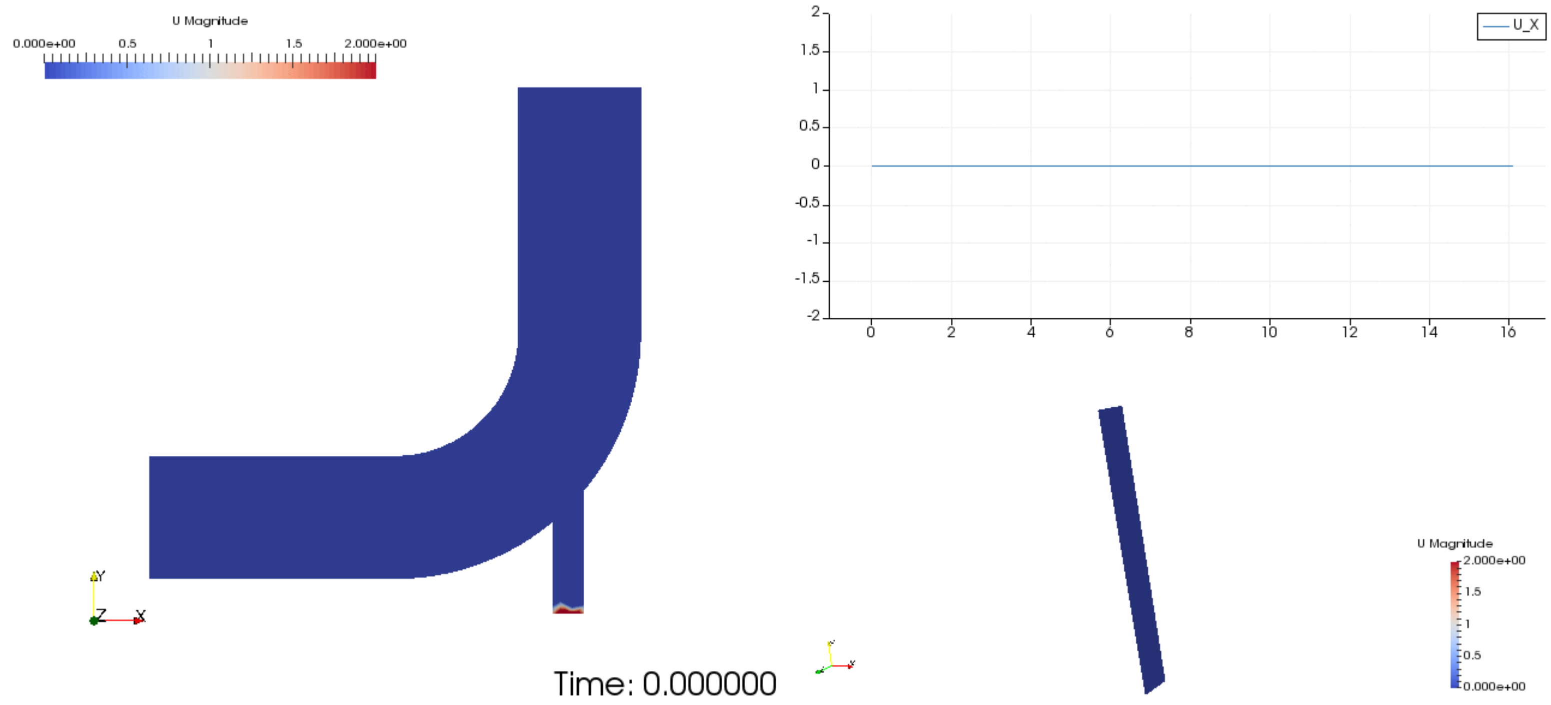
```
1.  $> foamCleanTutorials
2.  $> fluentMeshToFoam ../../../meshes_and_geometries/fluent_elbow2d_1/ascii.msh
3.  $> checkMesh | tee log
4.  $> rm -rf 0
5.  $> rm -rf 0_org 0
6.  $> pisoFoam | tee log.solver
7.  $> paraFoam
```

- The **codeStream** boundary condition is implemented in the file *0/U*.
- FYI, you can run in parallel with no problem.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using codedFixedValue

- If everything went fine, you should get something like this,



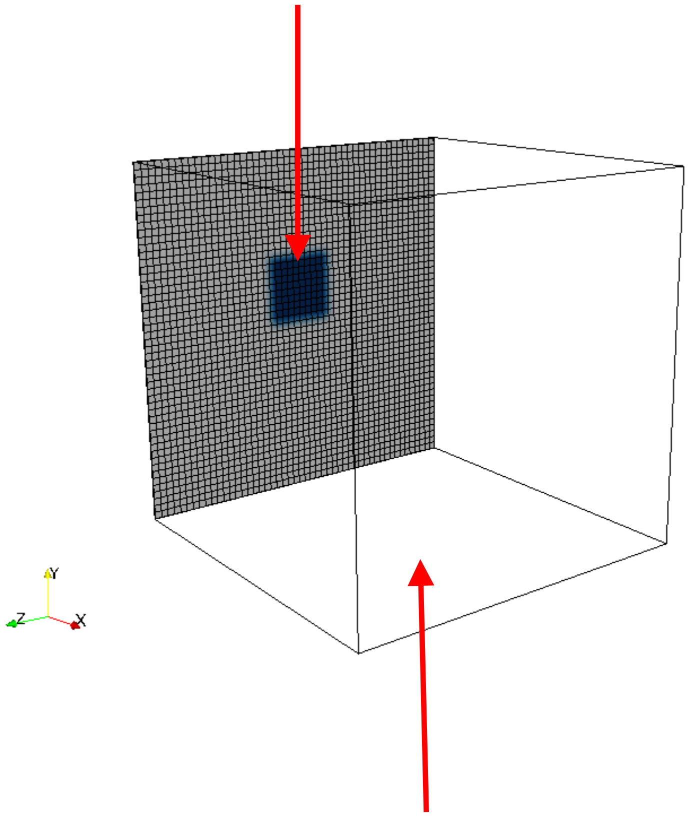
www.wolfdynamics.com/wiki/BCIC/elbow_unsBC1.gif

Implementing boundary conditions using codeStream

Filling a tank using codedFixedValue

- Let us do a final example.
- We will deal with scalar and vector fields at the same time.
- We will use **codedFixedValue**.
- For simplicity, we will only show the **code** section of the input files.
- Remember, the rest of the body can be used as a template.
- And depending of what you want to do, you might need to add new headers, libraries, and compilation options.
- Hereafter we will setup an inlet boundary condition in a portion of an existing patch.
- By using **codedFixedValue** BC, we do not need to modify the actual mesh topology.
- We will assign a velocity field and a scalar field to a set of faces (dark area in the figure).
- We are going to simulate filling a tank with water.
- We will use the solver `interFoam`.

Water enters here
This is a face selection in a single boundary patch



The tank is initially empty

Implementing boundary conditions using codeStream

- Definition of the vector field boundary condition (dictionary file *U*),

```
leftWall
{
  type                codedFixedValue;
  value               uniform (0 0 0);
  name                inletProfile1;

  code
  #{
     const fvPatch& boundaryPatch = patch();
     const vectorField& Cf = boundaryPatch.Cf();
     vectorField& field = *this;

     scalar minz = 0.4;
     scalar maxz = 0.6;
     scalar miny = 0.5;
     scalar maxy = 0.7;

     scalar t = this->db().time().value();
     ...
     ...
     ...
   #};
 }
```

Name of the patch where we want to implement the boundary condition

Use codedFixedValue BC and initialize value. The initialization is only needed for paraview in order to visualize something at time zero.

Unique name of the BC
Do not use the same name in other patches ⚠

Access boundary mesh information and initialize vector field **field**

Initialize variables

Access time

Implementing boundary conditions using codeStream

- Definition of the vector field boundary condition (dictionary file *U*),

```
7     code
8     #{
          ...
          ...
          ...
19
20        forAll(Cf, faceI)
21        {
22
23            if (
24                 (Cf[faceI].z() > minz)  &&
25                 (Cf[faceI].z() < maxz)  &&
26                 (Cf[faceI].y() > miny)  &&
27                 (Cf[faceI].y() < maxy)
28               )
29               {
30                   if ( t < 1.)
31                   {
32                       field[faceI] = vector(1,0,0);
33                   }
34                   else
35                   {
36                       field[faceI] = vector(0,0,0);
37                   }
38               }
39         }
40     #};
41     }
```

Code section. The actual implementation of the BC is done in this section

Loop using size of boundary patch (**Cf**) and iterator **faceI**.
This is equivalent to:
for (int faceI=0; Cf.size()<faceI; faceI++)

Use conditional structure to select faces according to the variables defined in lines 13-16

Use conditional structure to add time dependency and assign values to the selected faces.
The variable **field** was initialize in line 11.

Implementing boundary conditions using codeStream

- Definition of the scalar field boundary condition (dictionary file *alpha.water*),

```
leftWall
{
    type                codedFixedValue;
    value               uniform 0;
    Name                inletProfile2;

    code
    #{
         const fvPatch& boundaryPatch = patch();
         const vectorField& Cf = boundaryPatch.Cf();
         scalarField& field = *this;

         field = patchInternalField();

         scalar minz = 0.4;
         scalar maxz = 0.6;
         scalar miny = 0.5;
         scalar maxy = 0.7;

         scalar t = this->db().time().value();

         ...
         ...
         ...
     #};
 }
```

Name of the patch where we want to implement the boundary condition

Use codedFixedValue BC and initialize value. The initialization is only needed for paraview in order to visualize something at time zero.

Unique name of the BC
Do not use the same name in other patches ⚠

Code section. The actual implementation of the BC is done in this section

Access boundary mesh information and initialize scalar field **field**

Assign value from the internal field to the patch

Initialize variables

Access time

Implementing boundary conditions using codeStream

- Definition of the scalar field boundary condition (dictionary file *alpha.water*),

```
7        code
8        #{
              ...
              ...
              ...
22
23            forAll(Cf, faceI)
24            {
25                if (
26                       (Cf[faceI].z() > minz)  &&
27                       (Cf[faceI].z() < maxz)  &&
28                       (Cf[faceI].y() > miny)  &&
29                       (Cf[faceI].y() < maxy)
30                   )
31                   {
32                       if ( t < 1.)
33                       {
34                           field[faceI] = 1.;
35                       }
36                       else
37                       {
38                           field[faceI] = 0.;
39                       }
40                   }
41             }
42        #};
43    }
```

Code section. The actual implementation of the BC is done in this section

Loop using size of boundary patch (**Cf**) and iterator **faceI**.
This is equivalent to:
for (int faceI=0; Cf.size()<faceI; faceI++)

Use conditional structure to select faces according to the variables defined in lines 13-16

Use conditional structure to add time dependency and assign values to the selected faces.
The variable **field** was initialize in line 11.

Implementing boundary conditions using codeStream

Implementation of a parabolic inlet profile using codedFixedValue

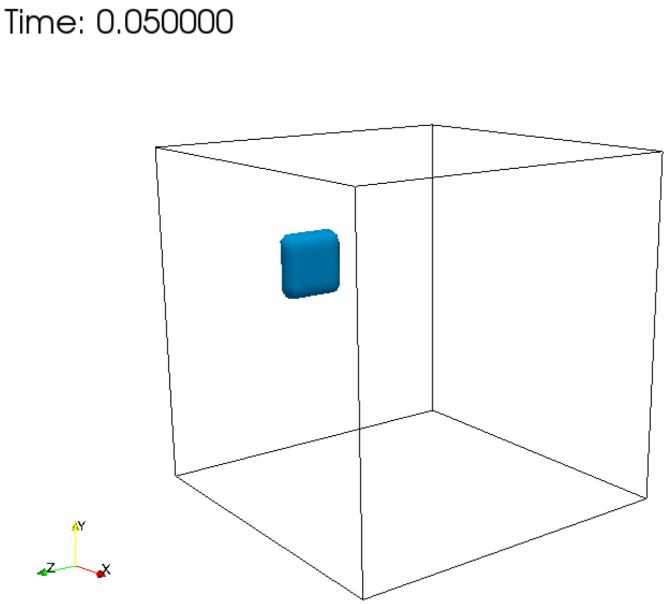
- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_BC/fillBox_BC/`**
- To run the case, type in the terminal,

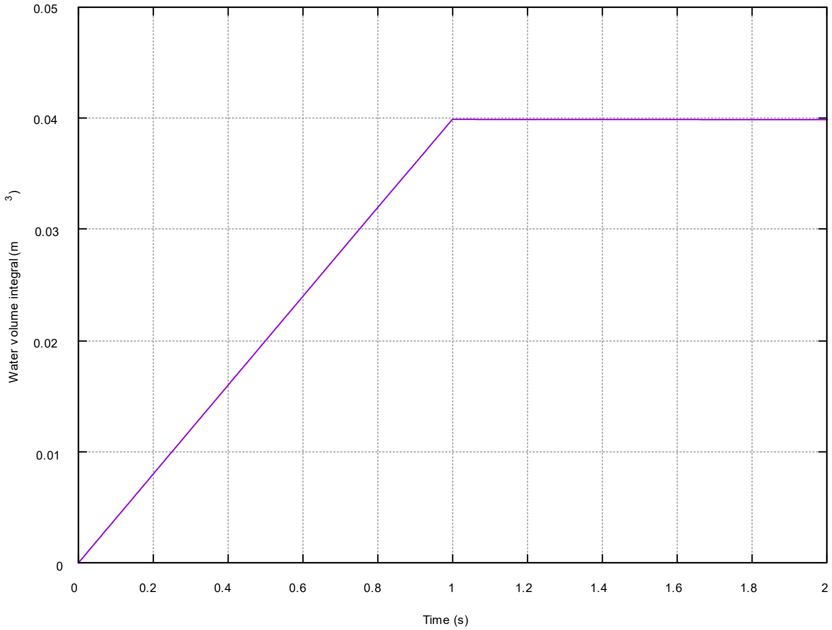
```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> rm -rf 0
4.  $> cp -r 0_org 0
5.  $> decomposePar
6.  $> mpirun -np 4 interFoam -parallel | tee log.solver
7.  $> reconstructPar
8.  $> paraFoam
```

- As you can see, we can also run in parallel with no problem.
- To visualize the parallel results, you will need to use the wrapper `paraFoam` with the option `–builtin`.

Implementation of a parabolic inlet profile using codedFixedValue

- If everything went fine, you should get something like this





Visualization of water phase (alpha.water)

Volume integral of water entering the domain

www.wolfdynamics.com/wiki/BCIC/filltank1.gif

Roadmap

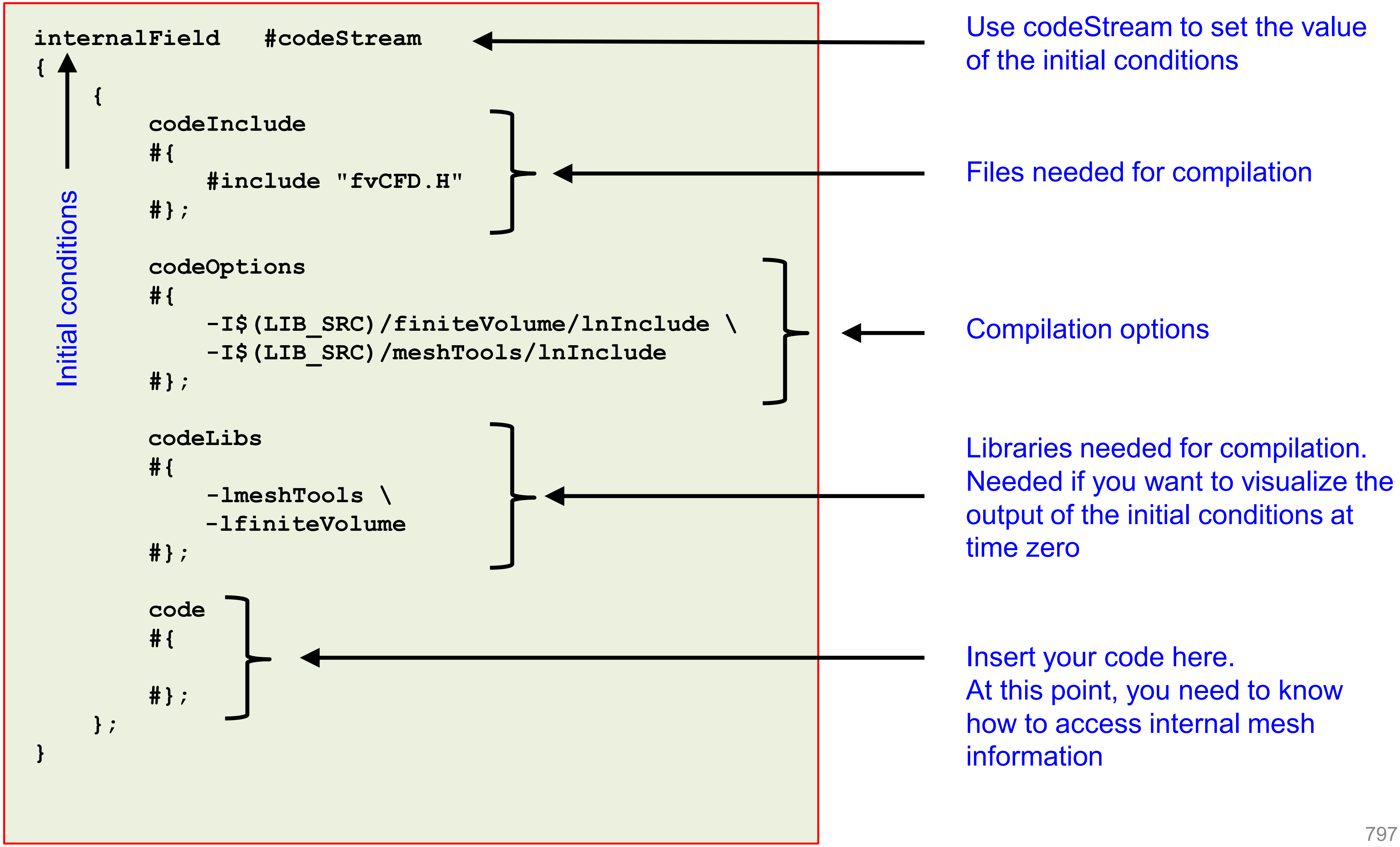
1. ~~codeStream – Highlights~~
2. ~~Implementing boundary conditions using codeStream~~
3. **Solution initialization using codeStream**

Solution initialization using codeStream

- When it comes to initial conditions, you can use the utility `setFields`.
- This utility is very flexible, you can even read STL files and use them to initialize your fields.
- But in case that you can not get the desired results using `setFields`, you can implement your own initial conditions using **codeStream**.
- To implement initial conditions using **codeStream**, we proceed in a similar way as for boundary conditions.
- The source code and binaries are automatically generated and copied in the directory **`dynamicCode`** of the current case.
- The source code is compiled automatically at run-time.
- The use of **codeStream** is a very good alternative to avoid high level programming of initial conditions or the use of external libraries.
- Hereafter we will use **codeStream** to implement new initial conditions.

Solution initialization using codeStream

Body of the codeStream directive for initial conditions



```
internalField   #codeStream
{
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Use codeStream to set the value of the initial conditions

Files needed for compilation

Compilation options

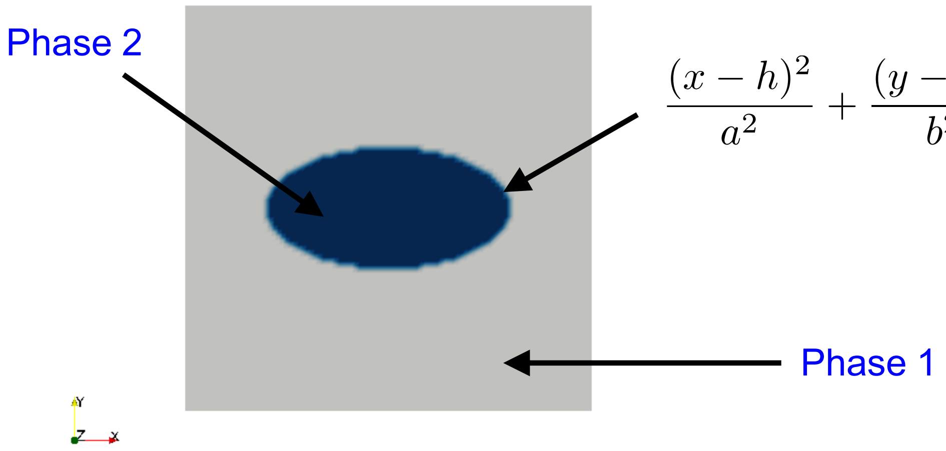
Libraries needed for compilation. Needed if you want to visualize the output of the initial conditions at time zero

Insert your code here.
At this point, you need to know how to access internal mesh information

Solution initialization using codeStream

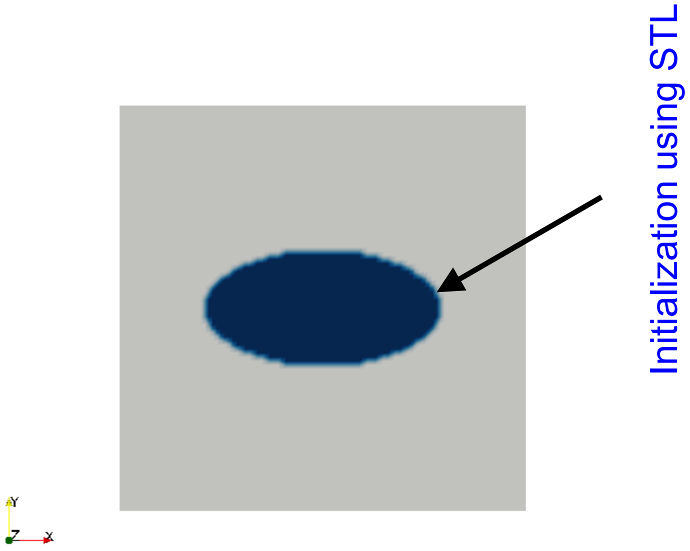
Implementation of an elliptic initialization using codeStream

- Let us implement an elliptic initialization using **codeStream**.
- The firs step is to know your domain and identify the region that you want to initialize.
- Then you will need to do a little bit of math to get the expression for the initialization.
- In this example, we are also going to show you how to do the same initialization by reading a STL file with the utility `setFields`.



$$\frac{(x-h)^2}{a^2}+\frac{(y-k)^2}{b^2}=1$$

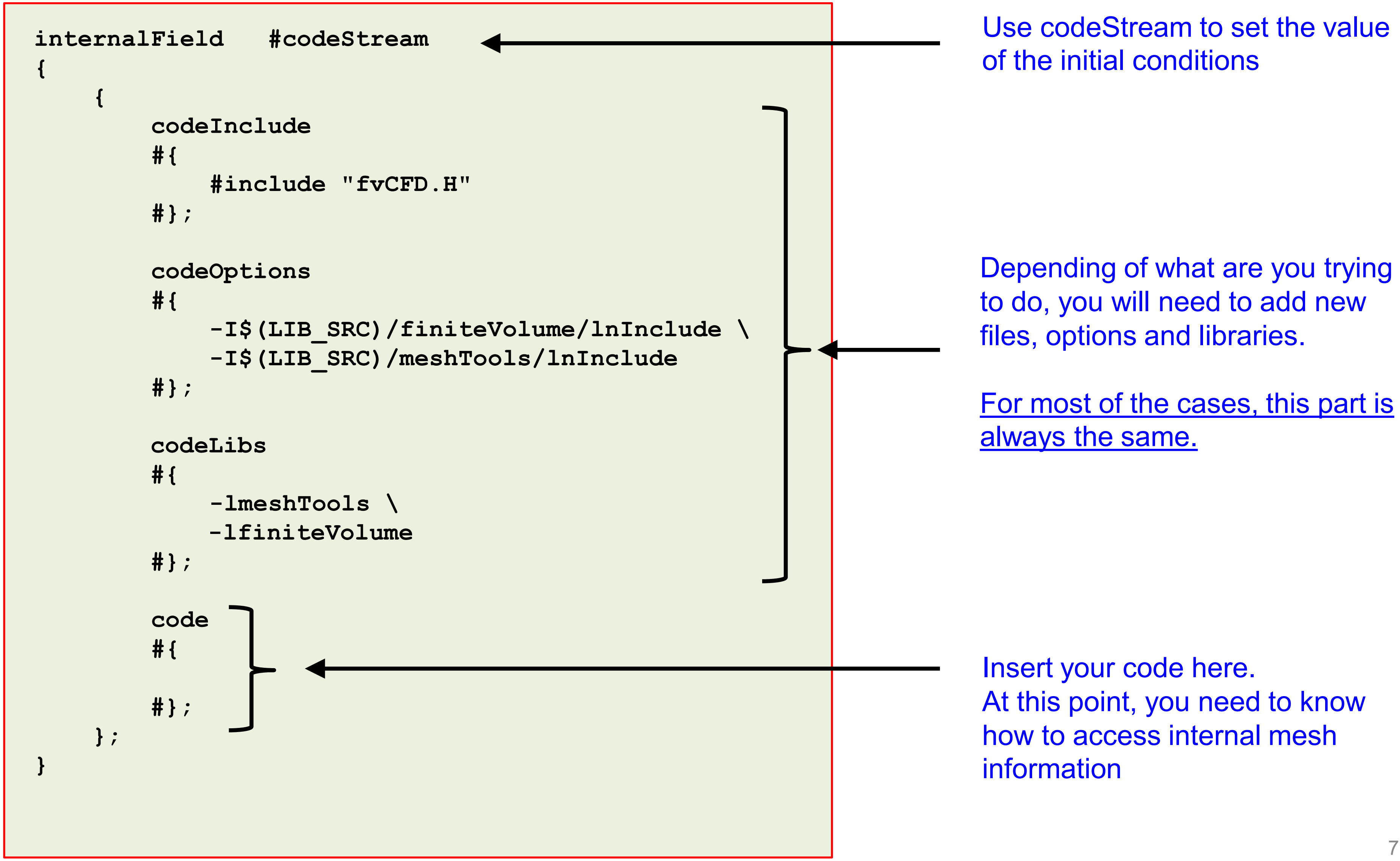
Initialization using **codeStream**



Initialization using a STL with `setFields`

Solution initialization using codeStream

- The **codeStream** IC in the body of the file *alpha.phase1* is as follows,



```
internalField   #codeStream
{
    {
        codeInclude
        #{
            #include "fvCFD.H"
        #};

        codeOptions
        #{
            -I$(LIB_SRC)/finiteVolume/lnInclude \
            -I$(LIB_SRC)/meshTools/lnInclude
        #};

        codeLibs
        #{
            -lmeshTools \
            -lfiniteVolume
        #};

        code
        #{

        #};
    };
}
```

Use codeStream to set the value of the initial conditions

Depending of what are you trying to do, you will need to add new files, options and libraries.

For most of the cases, this part is always the same.

Insert your code here.
At this point, you need to know how to access internal mesh information

Solution initialization using codeStream

- The **code** section of the **codeStream** IC in the body of the file *alpha.phase1* is as follows,

```
code
#{
    const IOdictionary& d = static_cast<const IOdictionary&>(dict);
    const fvMesh& mesh = refCast<const fvMesh>(d.db());

    scalarField alpha(mesh.nCells(), 0.);

    scalar he = 0.5;
    scalar ke = 0.5;
    scalar ae = 0.3;
    scalar be = 0.15;

    forAll(alpha, i)
    {
        const scalar x = mesh.C()[i][0];
        const scalar y = mesh.C()[i][1];
        const scalar z = mesh.C()[i][2];

        if ( pow(y-ke,2) <= ((1 - pow(x-he,2)/pow(ae,2) )*pow(be,2)) )
           {
               alpha[i] = 1.;
           }
     }

     writeEntry(os, "", alpha);
#};
```

Access internal mesh information

Initialize scalar field to zero

Initialize variables

forAll loop to access cell centers and to assign alpha values.
Notice the alpha was previously initialized.
The size of the loop is defined by alpha and the iterator is i.

Access cell centers coordinates

Assign value to alpha

$$(y-k)^2 \leq \left(1 - \frac{(x-h)^2}{a^2}\right) \times b^2$$

If this condition is true, do the following statement

Write output to input dictionary

Solution initialization using codeStream

Implementation of an elliptic initialization using codeStream

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_INIT/elliptical_IC`**
- To run the case, type in the terminal,

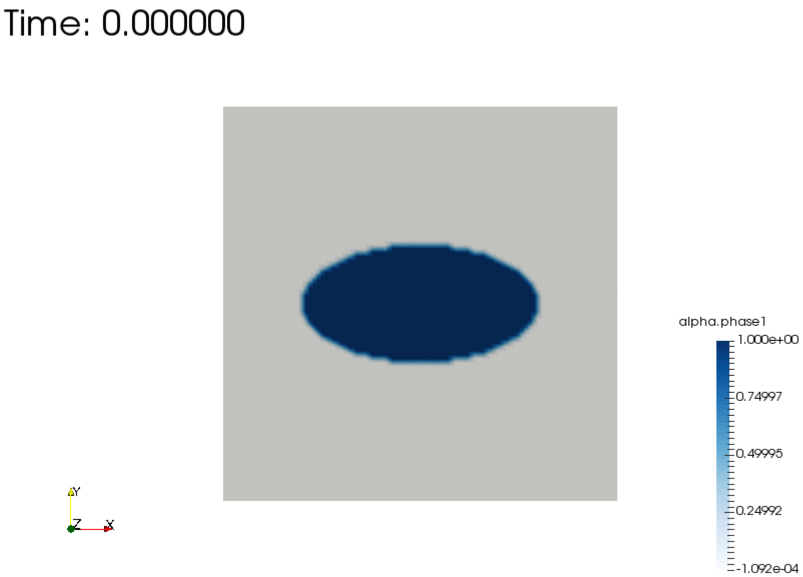
```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> rm -rf 0
4.  $> cp -r 0_org 0
5.  $> paraFoam
6.  $> interFoam | tee log.solver
7.  $> paraFoam
```

- In step 6, we launch `paraFoam` to visualize the initialization.
- FYI, you can run in parallel with no problem.

Solution initialization using codeStream

Implementation of an elliptic initialization using codeStream

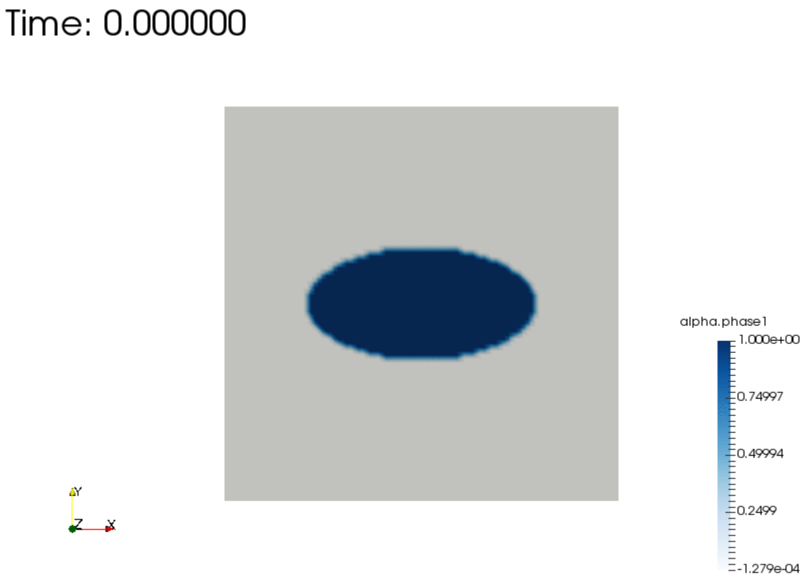
- If everything went fine, you should get something like this,



codeStream initialization

Visualization of volume fraction (alpha.phase1)

www.wolfdynamics.com/wiki/BCIC/bubble_zeroG.gif



setFields initialization

Visualization of volume fraction (alpha.phase1)

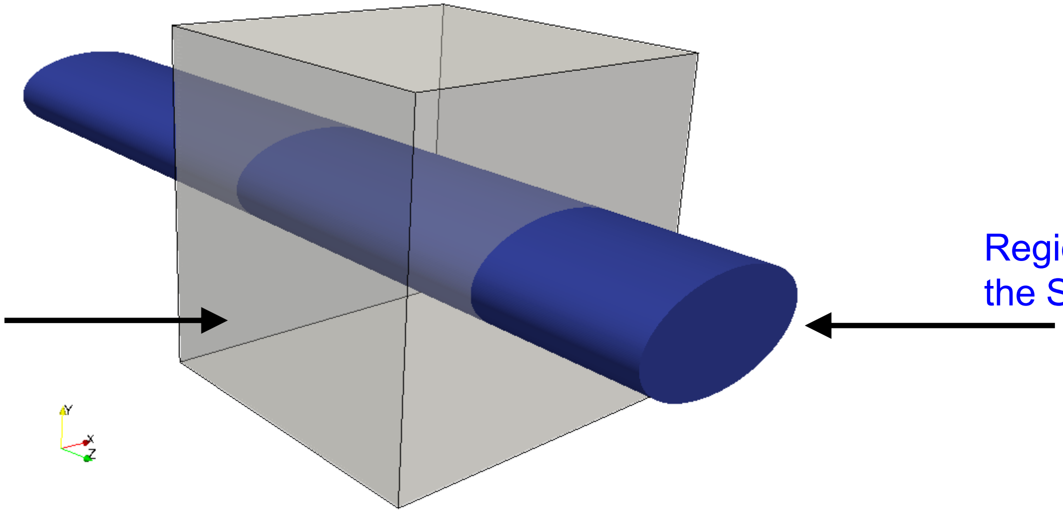
www.wolfdynamics.com/wiki/BCIC/bubble_zeroG_SF.gif

Surface tension driven flow - Bubble in a zero gravity flow using interFoam

Solution initialization using codeStream

Elliptic initialization using setFields

- Let us do the same initialization using a STL file with `setFields`.
- First, you will need to create the solid model that encloses the region you want to initialize.
- For this, you can use your favorite CAD/solid modeling software. Remember to save the geometry is STL format.
- Then you will need to read in the STL file using `setFields`.
- You will need to modify the `setFieldsDict` dictionary.



Computational domain

Region defined by the STL file

Solution initialization using codeStream

The setFieldsDict dictionary

```
defaultFieldValues
(
    volScalarFieldValue alpha.phase1 0
);

regions
(

    surfaceToCell
    {
        file "./geo/ellipse.stl";

        outsidePoints ((0.5 0.85 0));

        includeInside true;

        includeOutside false;

        includeCut false;

        fieldValues
        (
            volScalarFieldValue alpha.phase1 1
        );

    }
);
```

Initialize the whole domain to zero

setFields method to read STL files.
If you want to know all the options available use a word that does not exist in the enumerator list (e.g. banana)

Location of the STL file to read

A point located outside the STL

Use what is inside the STL

Use what is outside the STL

Include cells cut by the STL

Initialize this value.
In this case the initialization will be inside the STL

Solution initialization using codeStream

Elliptic initialization using setFields

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_INIT/elliptical_IC`**
- To run the case, type in the terminal,

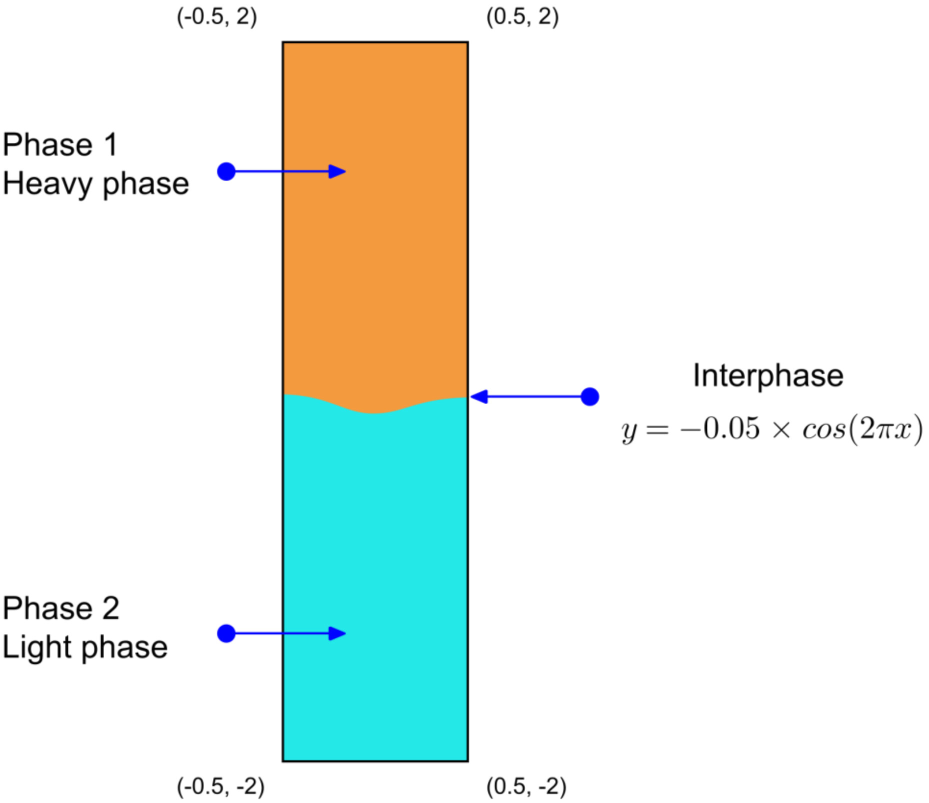
```
1.  $> foamCleanTutorials
2.  $> blockMesh
3.  $> rm -rf 0
4.  $> cp -r 0_org 0
5.  $> setFields
6.  $> paraFoam
```

- At this point, compare this initialization with the previous one.
- Also, feel free to launch the simulation using `interFoam`.

Solution initialization using codeStream

Rayleigh-Taylor instability initialization

- Let us study the Rayleigh-Taylor instability.
- In this case, we have two phases with different physical properties (one phase is heavier).
- To onset this instability, we need to perturbate somehow the interface between the two phases.
- We will use **codeStream** to initialize the two phases.
- For simplicity, we will only show the **code** section of the input files.
- The entries **codeInclude, codeOptions,** and **codeLibs**, are the same most of the times.



Solution initialization using codeStream

- The **code** section of the **codeStream** IC in the body of the file *alpha.phase1* is as follows,

```
code
#{
    const IOdictionary& d = static_cast<const IOdictionary&>(dict);
    const fvMesh& mesh = refCast<const fvMesh>(d.db());

    scalarField alpha(mesh.nCells(), 0.);

    forAll(alpha, i)
    {
        const scalar x = mesh.C()[i][0];
        const scalar y = mesh.C()[i][1];

        if (y >= -0.05*cos(2*constant::mathematical::pi*x))
        {
            alpha[i] = 1.;
        }
    }

    writeEntry(os, "", alpha);
#};
```

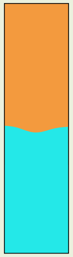
Access internal mesh information

Initialize scalar field to zero

Access cell centers coordinates

Assign value to alpha

$$y = -0.05 \times cos(2\pi x)$$



Write output to input dictionary

- For simplicity, we only show the **code** section.
- The rest of the body of the **codeStream** IC is a template.

Solution initialization using codeStream

Rayleigh-Taylor instability initialization

- This case is ready to run, the input files are located in the directory **`$PTOFC/101programming/codeStream_INIT/rayleigh_taylor`**
- To run the case, type in the terminal,

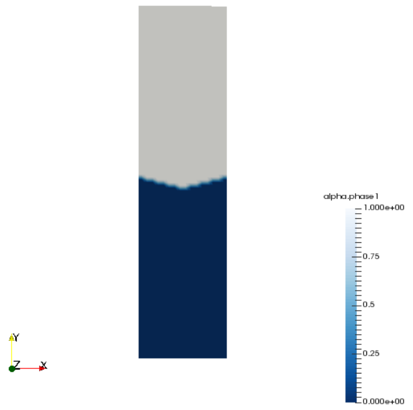
```
1. | $> foamCleanTutorials
2. | $> blockMesh
3. | $> rm -rf 0
4. | $> cp -r 0_org 0
5. | $> interFoam | tee log.solver
6. | $> paraFoam
```

- FYI, you can run in parallel with no problem.

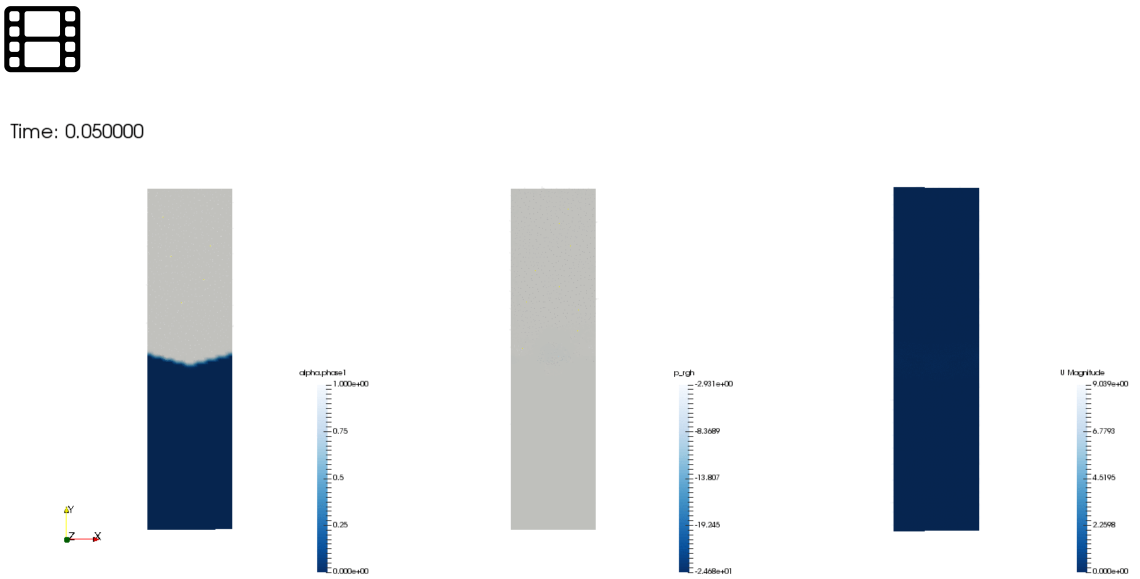
Solution initialization using codeStream

Rayleigh-Taylor instability initialization

- If everything went fine, you should get something like this,



Initial conditions



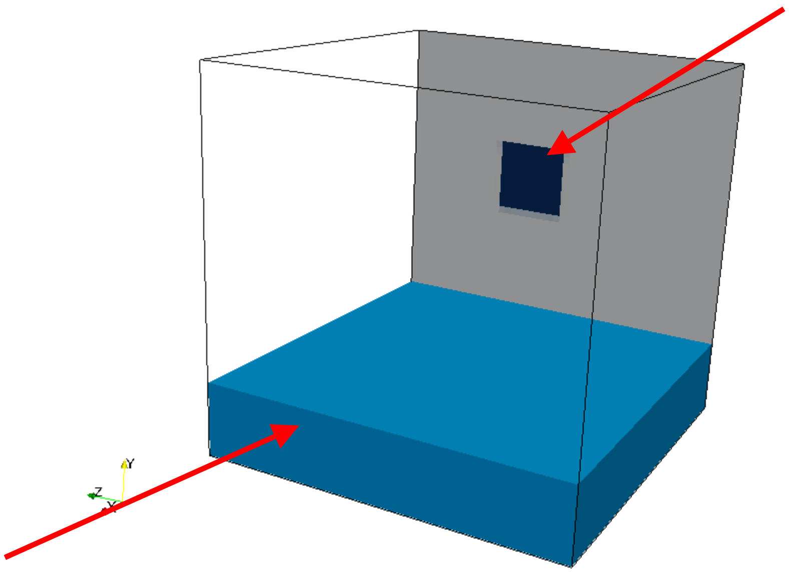
Visualization of volume fraction, static pressure and velocity magnitude

www.wolfdynamics.com/wiki/BCIC/rayleigh_taylor_ins1.gif

Solution initialization using codeStream

Filling a tank using codeStream and codedFixedValue

- Let us do a final example.
- We will implement BCs and ICs at the same.
- For simplicity, we will only show the **code** section of the input files.
- This setup is similar to the last example of the previous section (filling a tank using **codedFixedValue**).



Water enters here
This is a single boundary patch

Initial water level

Solution initialization using codeStream

- The **code** section of the **codeStream** IC in the body of the file *alpha.water* is as follows,

```
internalField   #codeStream
{
    ...
    ...
    ...
    code
    #{
        const IOdictionary& d = static_cast<const IOdictionary&>(dict);
        const fvMesh& mesh = refCast<const fvMesh>(d.db());

        scalarField alpha(mesh.nCells(), 0.);

        forAll(alpha, i)
        {
            const scalar x = mesh.C()[i][0];
            const scalar y = mesh.C()[i][1];
            const scalar z = mesh.C()[i][2];

            if (y <= 0.2)
            {
                alpha[i] = 1.;
            }
        }

        writeEntry(os, "", alpha);
    #};
```

Use codeStream to set the value of the initial conditions

Access internal mesh information

Initialize scalar field to zero

Access cell centers coordinates

Assign value to alpha according to conditional structure

Write output to input dictionary

Solution initialization using codeStream

- The **code** section of the **codeFixedValue** BC in the body of the file *U* is as follows,

Name of the patch where we want to implement the boundary condition

```
leftWall
{
    type            codedFixedValue;
    value           uniform (0 0 0);

    name            inletProfile1;

    code
    #{
        const fvPatch& boundaryPatch = patch();
        const vectorField& Cf = boundaryPatch.Cf();
        vectorField& field = *this;

        scalar min = 0.5;
        scalar max = 0.7;

        scalar t = this->db().time().value();
        ...
        ...
        ...
    #};
}
```

Use codedFixedValue BC and initialize value. The initialization is only needed for paraview in order to visualize something at time zero.

Unique name of the BC
Do not use the same name in other patches ⚠

Code section. The actual implementation of the BC is done here

Access boundary mesh information and initialize vector field **field**

Initialize variables

Access time

Solution initialization using codeStream

- The **code** section of the **codeFixedValue** BC in the body of the file *U* is as follows,

```
code
#{
    ...
    ...
    ...

    forAll(Cf, faceI)
    {

        if (
               (Cf[faceI].z() > min)  &&
               (Cf[faceI].z() < max)  &&
               (Cf[faceI].y() > min)  &&
               (Cf[faceI].y() < max)
           )
           {
               if ( t < 2.)
               {
                   field[faceI] = vector(1,0,0);
               }
               else
               {
                   field[faceI] = vector(0,0,0);
               }
           }
    }
#};
```

Code section. The actual implementation of the BC is done here

Loop using size of boundary patch (**Cf**) and iterator **faceI**.
This is equivalent to:
for (int faceI=0; Cf.size()<faceI; faceI++)

Use conditional structure to select faces.

Use conditional structure to add time dependency and assign values to the selected faces.

Solution initialization using codeStream

- The **code** section of the **codeFixedValue** BC in the body of the file `alpha.water` is as follows,

```
leftWall
{
    type            codedFixedValue;
    value           uniform 0;

    name            inletProfile2;


    code
    #{
        const fvPatch& boundaryPatch = patch();
        const vectorField& Cf = boundaryPatch.Cf();
        scalarField& field = *this;


        field = patchInternalField();

        scalar min = 0.5;
        scalar max = 0.7;

        scalar t = this->db().time().value();
        ...
        ...
        ...
    #};
}
```

Name of the patch where we want to implement the boundary condition

Use codedFixedValue BC and initialize value. The initialization is only needed for paraview in order to visualize something at time zero.

Unique name of the BC
Do not use the same name in other patches ⚠

Code section. The actual implementation of the BC is done here

Access boundary mesh information and initialize scalar field **field**

Assign value from the internal field to the patch

Initialize variables

Access time

Solution initialization using codeStream

- The **code** section of the **codeFixedValue** BC in the body of the file *alpha.water* is as follows,

```
code
#{
        ...
        ...
        ...

    forAll(Cf, faceI)
    {
        if (
                (Cf[faceI].z() > min) &&
                (Cf[faceI].z() < max) &&
                (Cf[faceI].y() > min) &&
                (Cf[faceI].y() < max)
              )
              {
                 if ( t < 2.)
                 {
                     field[faceI] = 1.;
                 }
                 else
                 {
                     field[faceI] = 0.;
                 }
              }
    }
#};
```

Code section. The actual implementation of the BC is done here

Loop using size of boundary patch (**Cf**) and iterator **faceI**.
This is equivalent to:
for (int faceI=0; Cf.size()<faceI; faceI++)

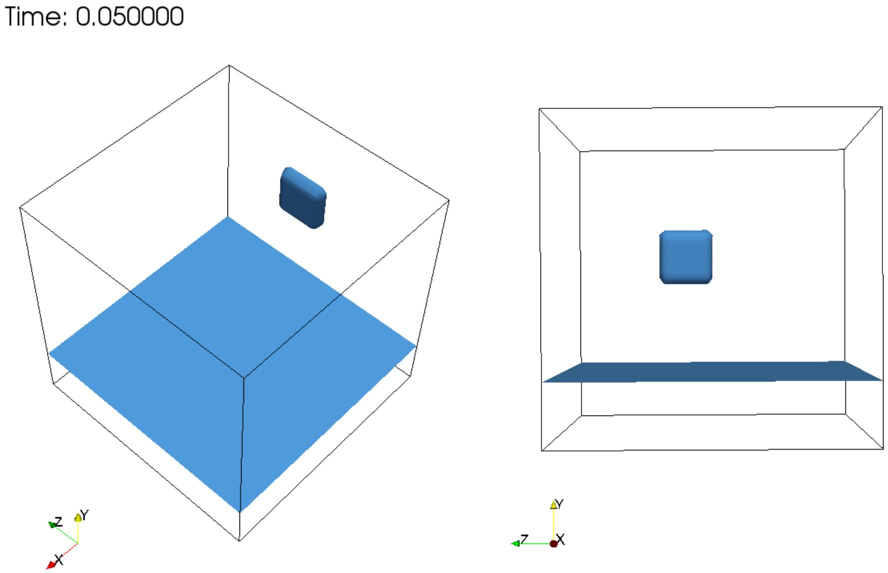
Use conditional structure to select faces

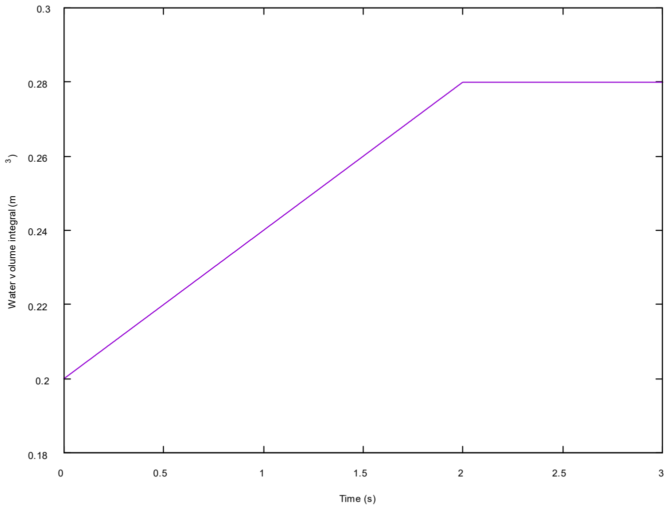
Use conditional structure to add time dependency and assign values to the selected faces.

Solution initialization using codeStream

Filling a tank using codeStream and codedFixedValue

- If everything went fine, you should get something like this,





Visualization of water phase (alpha.water)

www.wolfdynamics.com/wiki/BCIC/filltank2.gif

Volume integral of water entering the domain

Module 8

Advanced physics
Turbulence modeling – Multiphase flows – Compressible flows – Moving bodies – Source terms – Passive scalars

- In this module, we will deal with advanced modeling capabilities.
- Advanced modeling capabilities rely a lot in physical models, such as, turbulence, multiphase flows, porous media, combustion, radiation, heat transfer, phase change, acoustics, cavitation, and so on.
- Therefore, it is extremely important to get familiar with the theory behind these models.

"Essentially, all models are wrong,
but some are useful"

G. E. P. Box

George Edward Pelham Box
18 October 1919 – 28 March 2013. Statistician, who worked in the areas of quality control, time-series analysis, design of experiments, and Bayesian inference. He has been called *"one of the great statistical minds of the 20th century"*.

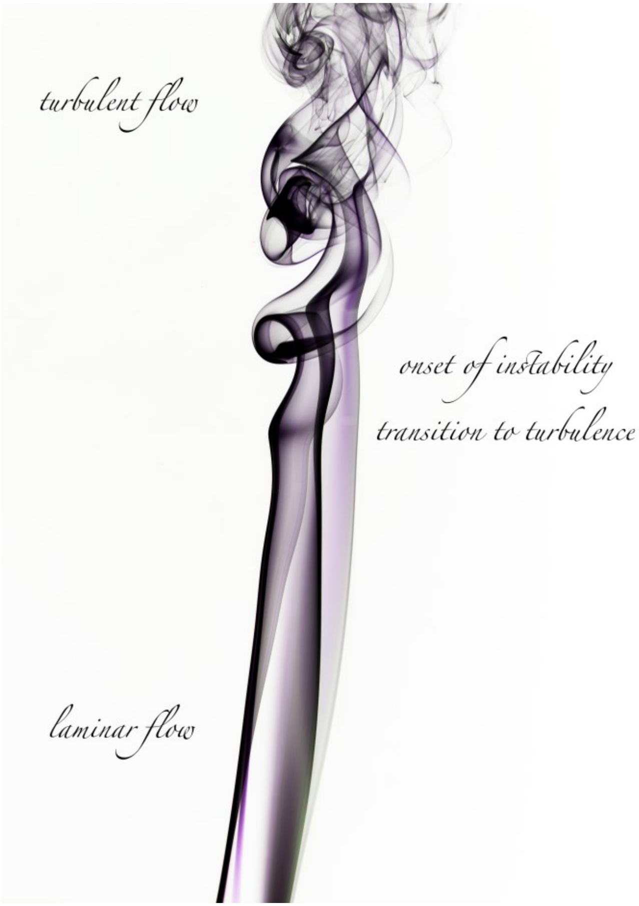
Roadmap

A crash introduction to:

1. **Turbulence modeling in OpenFOAM®**
2. Multiphase flows modeling in OpenFOAM®
3. Compressible flows in OpenFOAM®
4. Moving bodies in OpenFOAM®
5. Source terms in OpenFOAM®
6. Scalar transport pluggable solver

What is turbulence?

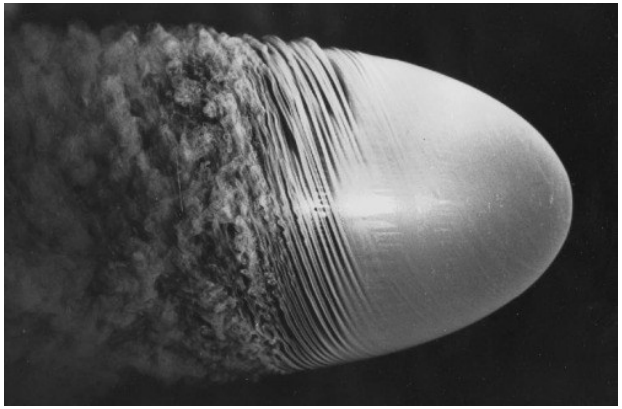
- For the purpose of this training, let us state the following:
  - Turbulence is an unsteady, aperiodic motion in which all three velocity components fluctuate in space and time.
  - Every transported quantity shows similar fluctuations (pressure, temperature, species, concentration, and so on)
  - Turbulent flows contains a wide range of eddy sizes (scales):
    - Large eddies derives their energy from the mean flow. The size and velocity of large eddies are on the order of the mean flow.
    - Large eddies are unstable and they break-up into smaller eddies.
    - The smallest eddies convert kinetic energy into thermal energy via viscous dissipation.
    - The behavior of small eddies is more universal in nature.



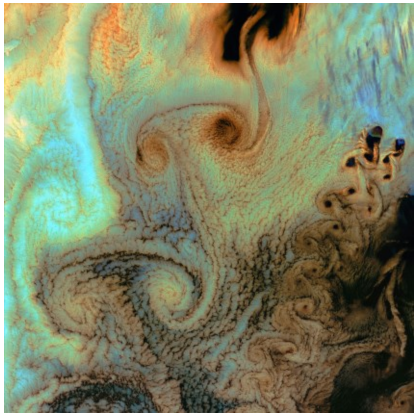
Buoyant plume of smoke rising from a stick of incense
Photo credit: https://www.flickr.com/photos/jlhopgood/
This work is licensed under a Creative Commons License (CC BY-NC-ND 2.0)

A crash introduction to turbulence modeling in OpenFOAM®

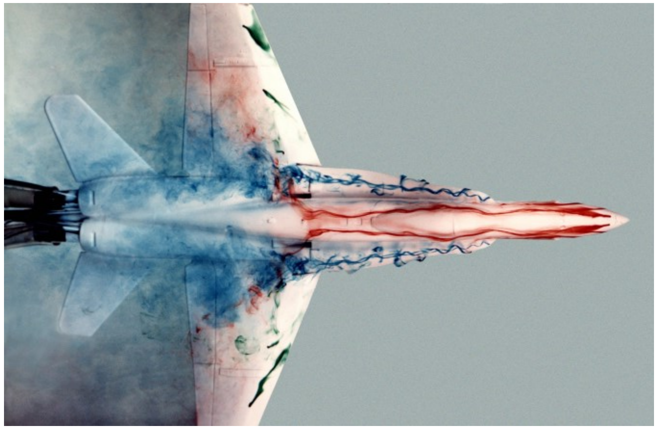
Wake turbulence behind individual wind turbines
Photo credit: NREL's wind energy research group.
Copyright on the images is held by the contributors. Apart from Fair Use, permission must be sought for any other purpose.



Flow visualization over a spinning spheroid
Photo credit: Y. Kohama.
Copyright on the images is held by the contributors. Apart from Fair Use, permission must be sought for any other purpose.

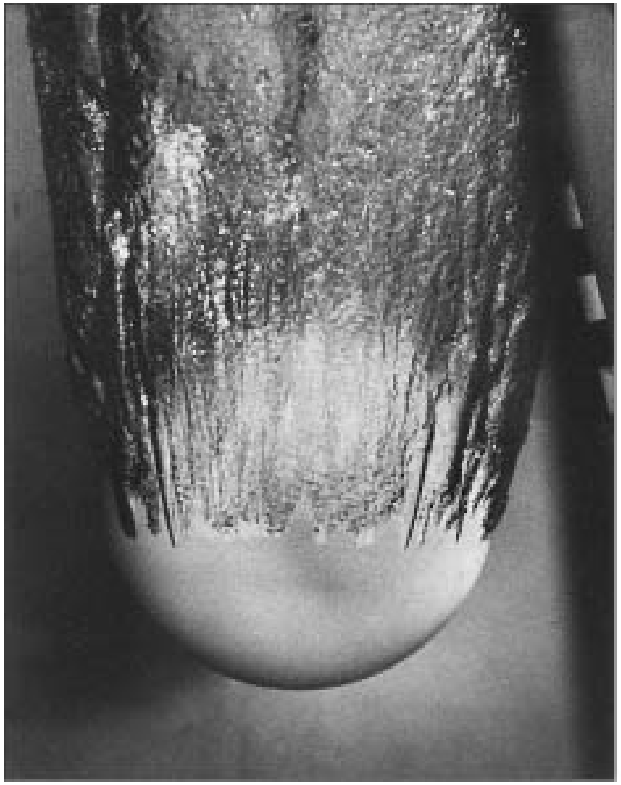


Von Karman vortices created when prevailing winds sweeping east across the northern Pacific Ocean encountered Alaska's Aleutian Islands
Photo credit: USGS EROS Data Center Satellite Systems Branch.
Copyright on the images is held by the contributors. Apart from Fair Use, permission must be sought for any other purpose.

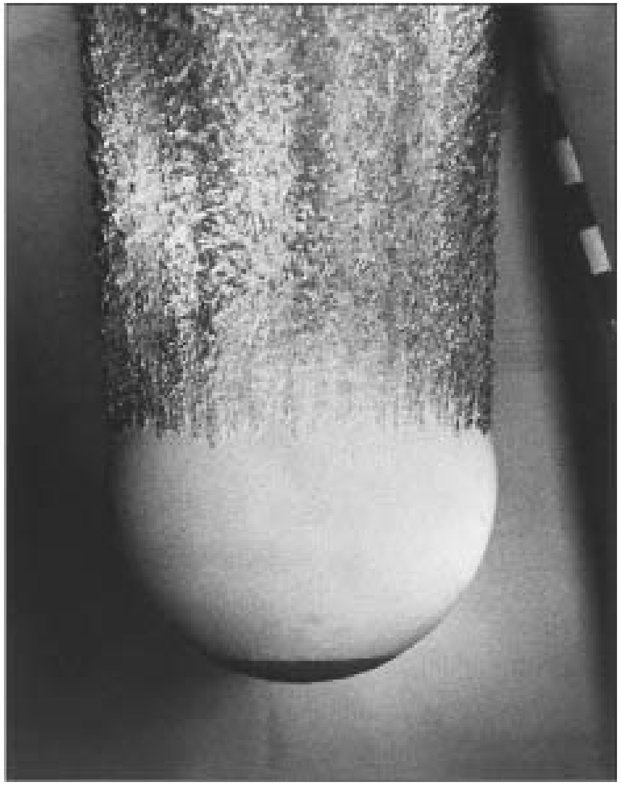


Vortices on a 1/48-scale model of an F/A-18 aircraft inside a Water Tunnel
Photo credit: NASA Dryden Flow Visualization Facility.
Copyright on the images is held by the contributors. Apart from Fair Use, permission must be sought for any other purpose.

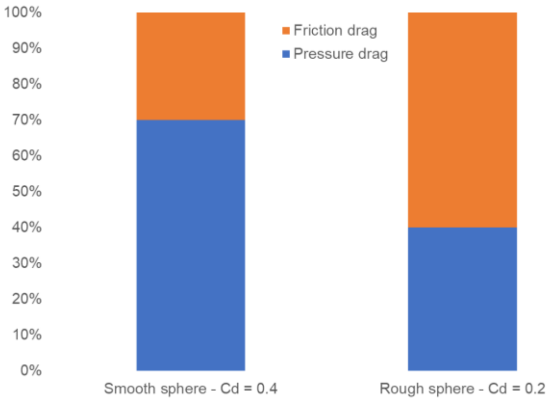
Turbulence, does it matter?



(*a*) $C_D \approx 0.4$



(*b*) $C_D \approx 0.2$



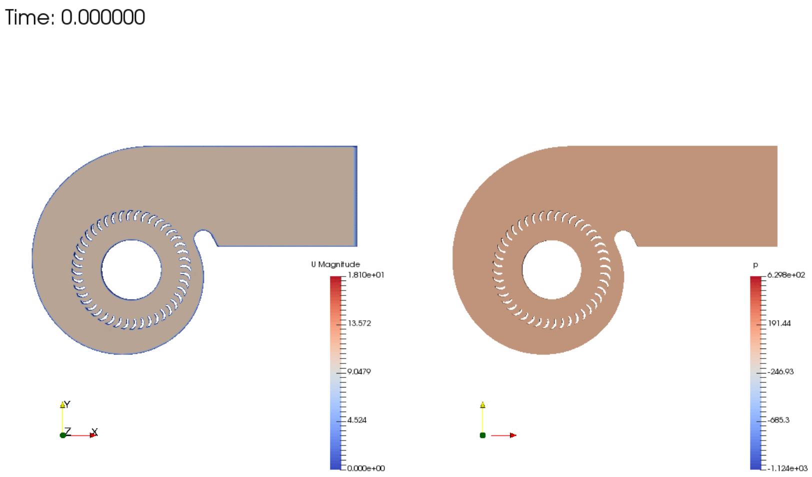
Abstract representation of the drag decomposition

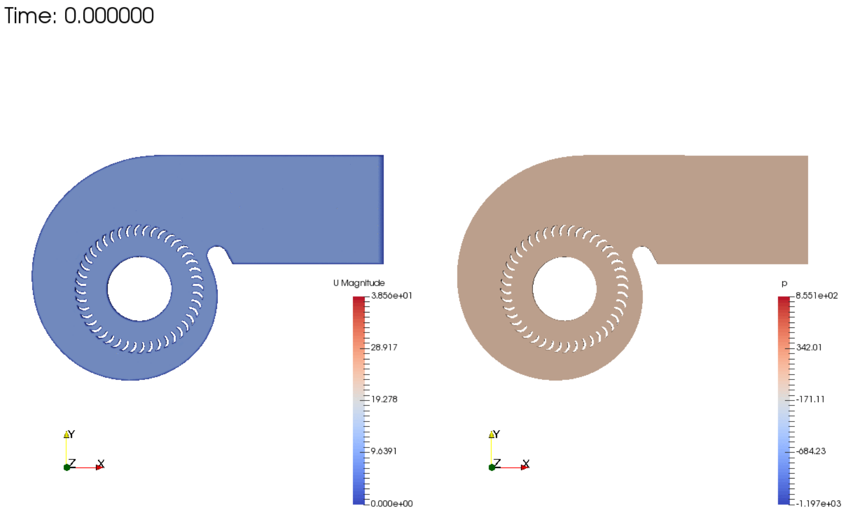
Flow around two spheres. Left image: smooth sphere. Right image: sphere with rough surface at the nose

Photo credit: http://www.mhhe.com/engcs/civil/finnemore/graphics/photos/AuthorRecommendedImages/index.html

Turbulence, does it matter?

Blower simulation using sliding grids



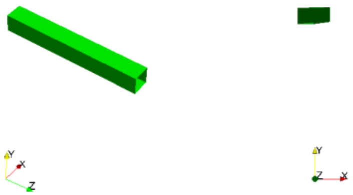


No turbulence model used (laminar, no turbulence modeling, DNS, unresolved DNS, name it as you want)
http://www.wolfdynamics.com/training/turbulence/image1.gif

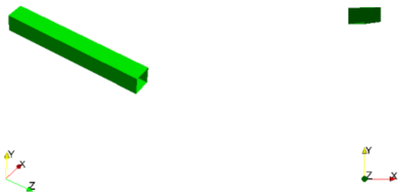
K-epsilon turbulence model
http://www.wolfdynamics.com/training/turbulence/image2.gif

Turbulence, does it matter?

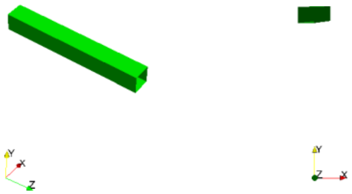
Vortex shedding past square cylinder



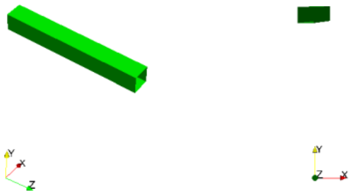
URANS (K-Omega SST with no wall functions) – Vortices visualized by Q-criterion
www.wolfdynamics.com/wiki/squarecil/urans2.gif



LES (Smagorinsky) – Vortices visualized by Q-criterion
www.wolfdynamics.com/wiki/squarecil/les.gif



Laminar (no turbulence model) – Vortices visualized by Q-criterion
www.wolfdynamics.com/wiki/squarecil/laminar.gif



DES (SpalartAllmarasDDES) – Vortices visualized by Q-criterion
www.wolfdynamics.com/wiki/squarecil/des.gif

Turbulence, does it matter?

Vortex shedding past square cylinder

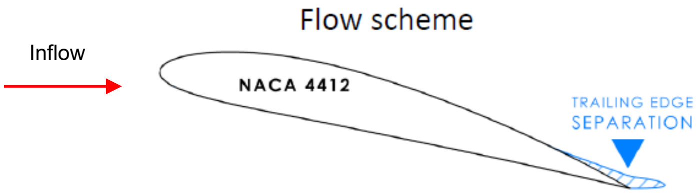
Turbulence model	Drag coefficient	Strouhal number	Computing time (s)
Laminar	2.81	0.179	93489
LES	2.32	0.124	77465
DES	2.08	0.124	70754
URANS (WF)	2.31	0.130	67830
URANS (No WF)	2.28	0.135	64492
RANS	2.20	-	28246 (10000 iter)
Experimental values	2.05-2.25	0.132	-

Note: all simulations were run using 4 cores.

References:
Lyn, D.A. and Rodi, W., The flapping shear layer formed by flow separation from the forward corner of a square cylinder. *J. Fluid Mech., 267, 353, 1994.*
Lyn, D.A., Einav, S., Rodi, W. and Park, J.H., A laser-Doppler velocimetry study of ensemble-averaged characteristics of the turbulent near wake of a square cylinder. *Report. SFB 210 /E/100.*

Turbulence, does it matter?

Separated flow around a NACA-4412 airfoil



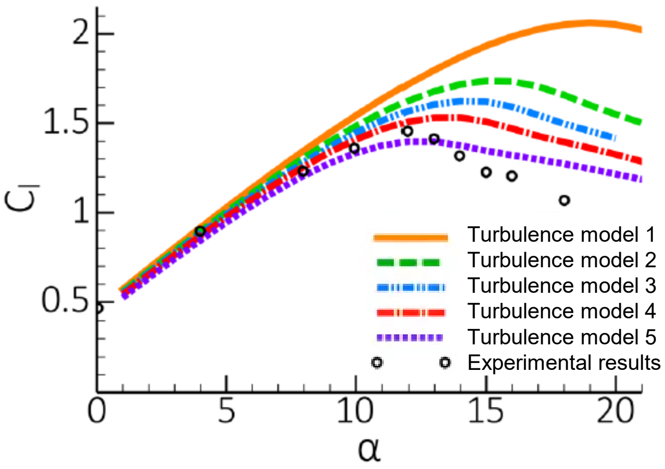
Incompressible flow

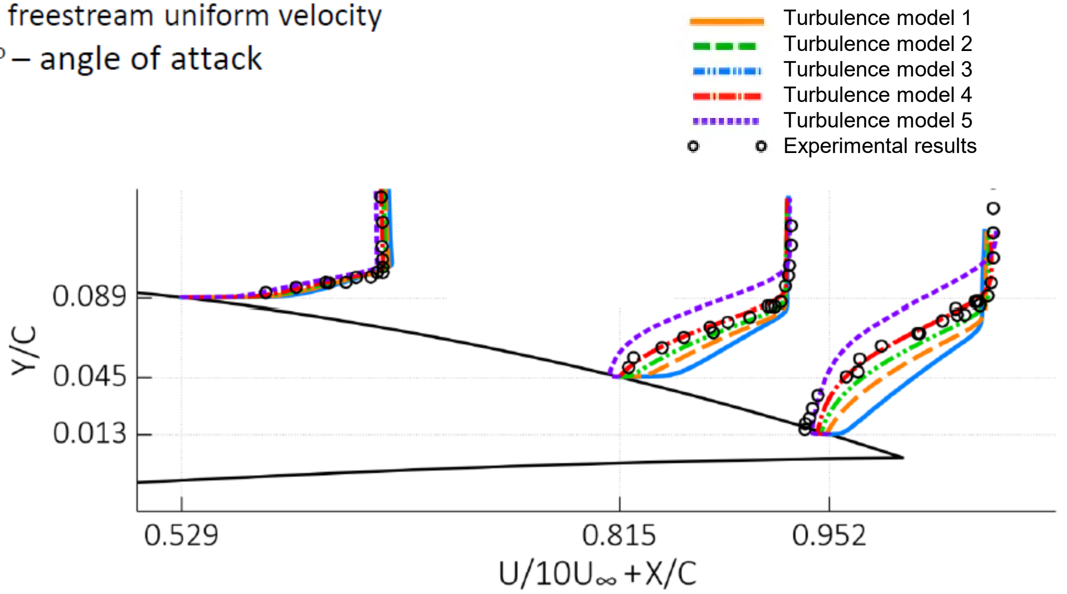
$Re = U_{\infty} \cdot C/\nu = 1.64 \cdot 10^{6}$

C - airfoil chord

U_{∞} - freestream uniform velocity

$\alpha = 12^{\circ}$ – angle of attack





- CFD has been around since the late 1970s, and after all these years is not that easy to compute the flow around 2D airfoils.
- In particular, predicting the maximum lift and stall characteristics is not trivial.

References:

F. Menter. “A New Generalized k-omega model. Putting flexibility into Turbulence models (GEKO)”, Ansys Germany
A. J. Wadcock. “Investigation of Low-Speed Turbulent Separated Flow Around Airfoils”, NASA Contractor Report 177450

A crash introduction to turbulence modeling in OpenFOAM®

Turbulence modeling in engineering

- Most natural and engineering flows are turbulent, hence the necessity of modeling turbulence.
- The goal of turbulence modeling is to develop equations that predict the time averaged velocity, pressure, temperature fields without calculating the complete turbulent flow pattern as a function of time.
- There is no universal turbulence model, hence you need to know the capabilities and limitations of the turbulence models.
- Simulating turbulent flows in any general CFD solver requires selecting a turbulence model, providing initial conditions and boundary conditions for the closure equations of the turbulent model, selecting a near-wall modeling, and choosing runtime parameters and numerics.

Why modeling turbulent flows is challenging?

- Unsteady aperiodic motion.
- All fluid properties and transported quantities exhibit random spatial and temporal variations.
- They are intrinsically three-dimensional due to vortex stretching.
- Strong dependence from initial conditions.
- Contains a wide range of scales (eddies).
- Therefore, in order to accurately model/resolve turbulent flows, the simulations must be three-dimensional, time-accurate, and with fine enough meshes such that all spatial scales are resolved.

Reynolds number and Rayleigh number

- It is well known that the Reynolds number characterizes if the flow is laminar or turbulent.
- So before doing a simulation or experiment, check if the flow is turbulent.
- The Reynolds number is defined as follows,

Convective effects

$$Re_L = \frac{\rho U L}{\mu} \quad \text{where} \quad L = x, d, d_h, \text{ etc}$$

Viscous effects

- Where *U* and *L* are representative velocity and length scales.
- If you are dealing with natural convection, you can use the Rayleigh number, Grashof number, and Prandtl number to characterize the flow.

Specific heat

Thermal expansion coefficient

Buoyancy effects

$$Ra = \frac{g\beta L^3 \Delta T}{\nu\alpha} = \frac{\rho^2 c_p \beta g L^3 \Delta T}{\mu k} = Gr \times Pr$$

Viscous effects

Thermal conductivity

Momentum diffusivity

$$Pr = \frac{\nu}{\alpha} = \frac{\mu c_p}{k} \qquad Gr = \frac{g\beta(T_S - T_\infty)L^3}{\nu^2}$$

Thermal diffusivity

Reynolds number and Rayleigh number

- Turbulent flow occurs at large Reynolds number.
  - For external flows,

$$Re_x \geq 500000$$

Around slender/streamlined bodies (surfaces)

$$Re_d \geq 20000$$

Around an obstacle (bluff body)

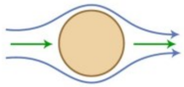
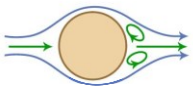
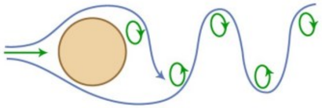
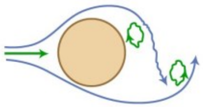
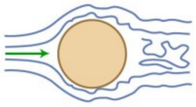
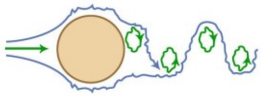
  - For internal flows,

$$Re_{d_h} \geq 2300$$

- Notice that other factors such as free-stream turbulence, surface conditions, blowing, suction, roughness and other disturbances, may cause transition to turbulence at lower Reynolds number.
- If you are dealing with natural convection and buoyancy, turbulent flows occurs when

$$\frac{Ra}{Pr} \geq 10^9$$

What happens when we increase the Reynolds number?

	**Creeping flow (no separation)** **Steady flow**	Re < 5
	**A pair of stable vortices in the wake** **Steady flow**	5 < Re < 40 - 46
	**Laminar vortex street (Von Karman street)** **Unsteady flow**	40 - 46 < Re < 150
	**Laminar boundary layer up to the separation point, turbulent wake** **Unsteady flow**	150 < Re < 300 Transition to turbulence 300 < Re < 3 x 10^5
	**Boundary layer transition to turbulent** **Unsteady flow**	3 x 10^5 < Re < 3 x 10^6
	**Turbulent vortex street, but the wake is narrower than in the laminar case** **Unsteady flow**	3 x 10^6 > Re

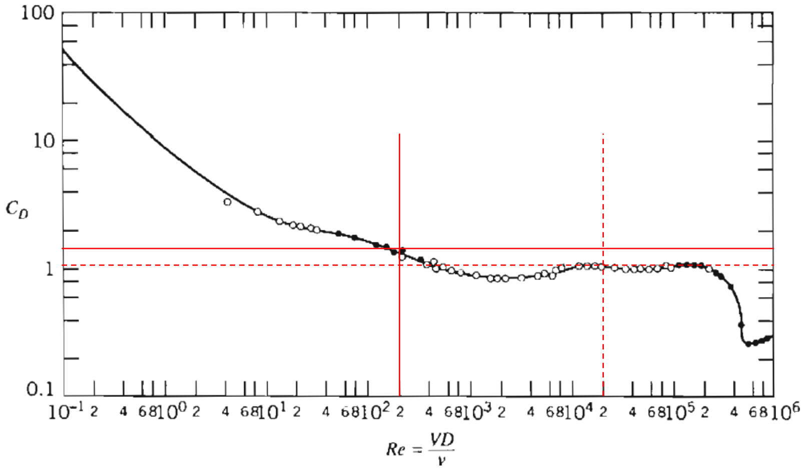
- Easy to simulate
- Steady

- Relatively easy to simulate.
- It becomes more challenging when the boundary layer transition to turbulent
- Unsteady

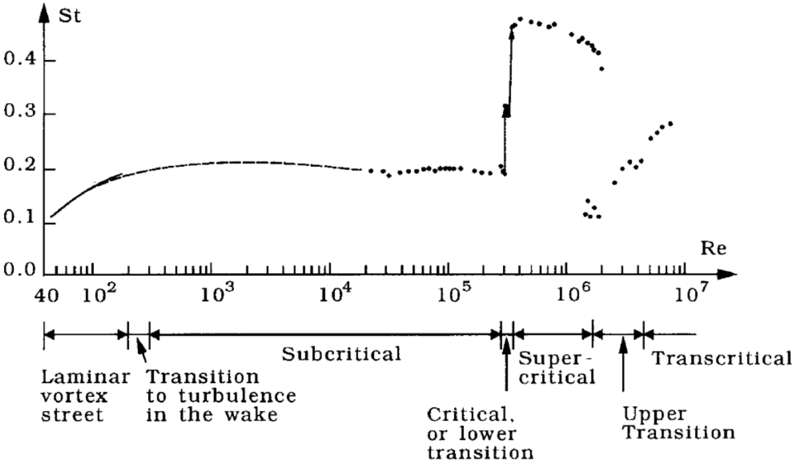
- Challenging to simulate
- Unsteady

Vortex shedding behind a cylinder and Reynolds number

What happens when we increase the Reynolds number?



Drag coefficient as a function of Reynolds number for a smooth cylinder [1]

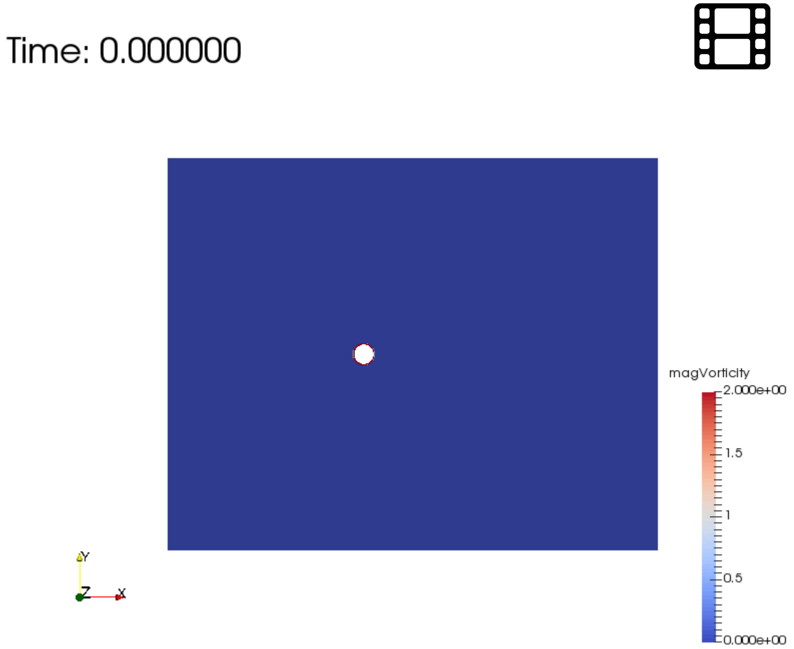


Strouhal number $St = \frac{fL}{U}$ for a smooth cylinder [2]

References:

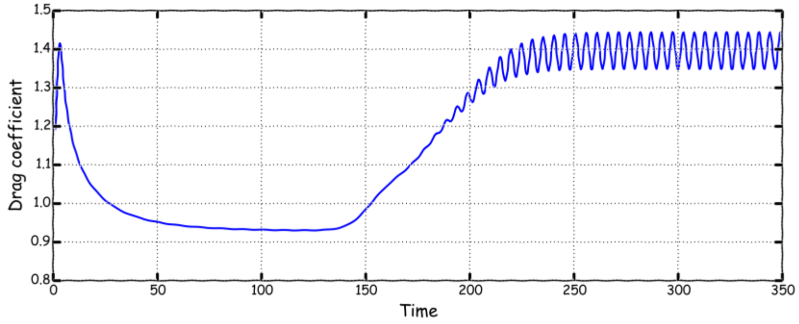
1. Fox, Robert W., et al. Introduction to Fluid Mechanics. Hoboken, NJ, Wiley, 2010
2. Sumer, B. Mutlu, et al. Hydrodynamics Around Cylindrical Structures. Singapore, World Scientific, 2006

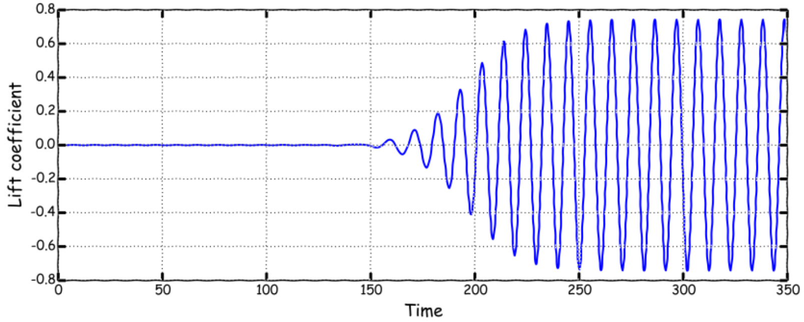
Vorticity does not always mean turbulence



Instantaneous vorticity magnitude field

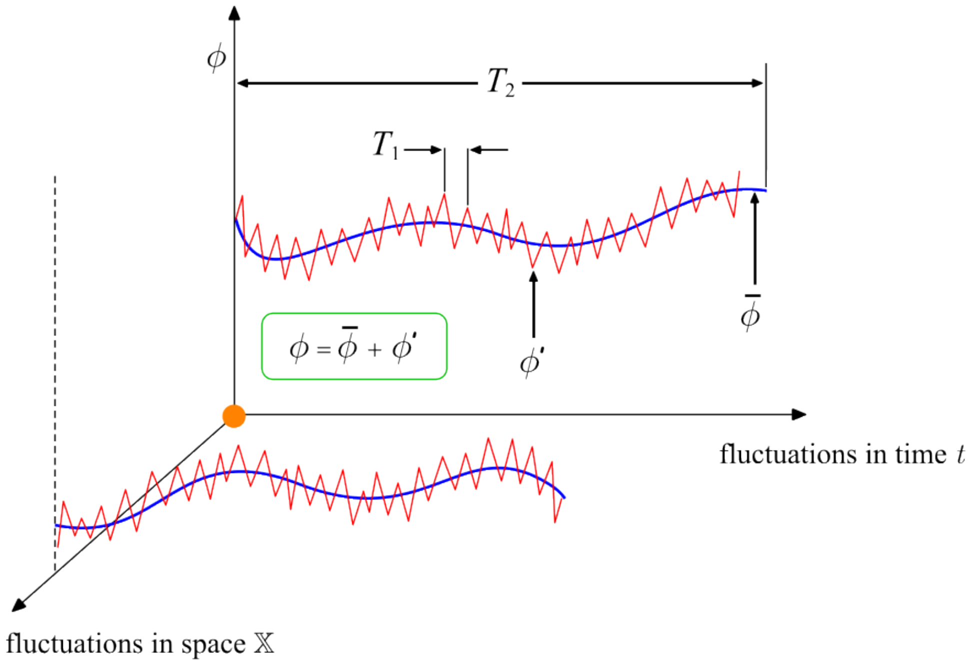
www.wolfdynamics.com/wiki/cylinder_vortex_shedding/movvort.gif





- The Reynold number in this case is 100, for these conditions the flow still is laminar.
- We are in the presence of the Von Karman vortex street, which is the periodic shedding of vortices caused by the unsteady separation of the fluid around blunt bodies.
- Vorticity is not a direct indication of turbulence.
- However turbulent flows are rotational, they exhibit vortical structures.

Turbulence modeling – Fluctuations of transported quantities


ϕ
T_2
T_1
$\phi = \bar{\phi} + \phi'$
ϕ'
$\bar{\phi}$
fluctuations in time t
fluctuations in space $\mathbb{X}$

$$\phi(\mathbf{x},t) = \bar{\phi}(\mathbf{x},t) + \phi'(\mathbf{x},t)$$

In RANS

- The overbar denotes the mean value.
- The prime denotes the fluctuating value.

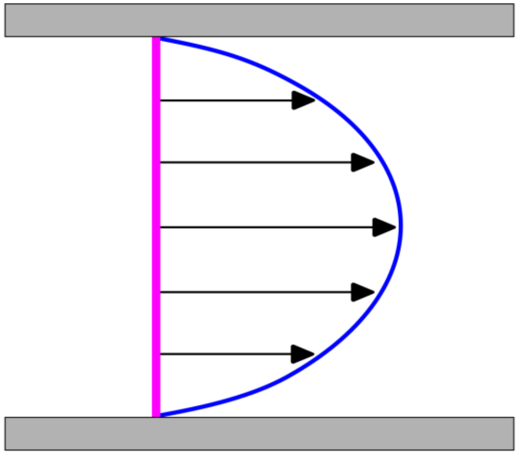
In LES

- The overbar denotes the filtered value.
- The prime denotes the modeled value or residual.

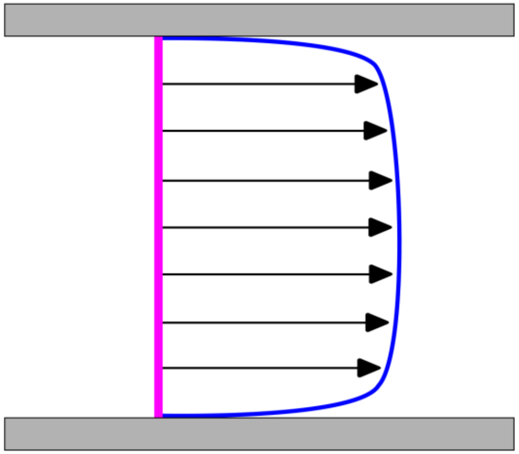
$$\bar{\phi}(\mathbf{x},t) = \frac{1}{T}\int_t^{t+T} \phi(\mathbf{x},t)\mathrm{d}t, \quad T_1 << T << T_2$$

- We have defined turbulence as an unsteady, aperiodic motion in which velocity components and every transported quantity fluctuate in space and time.
- For most engineering application it is impractical to account for all these instantaneous fluctuations.
- Therefore, we need to somehow remove those small scales by using models.
- To remove of filter the instantaneous fluctuations or small scales, two methods can be used: Reynolds averaging and Filtering
- Both methods introduce additional terms that must be modeled for closure.
- We are going to talk about closure methods later.

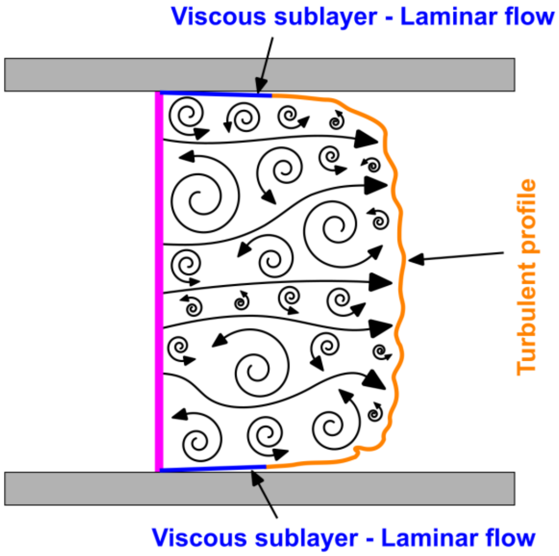
Turbulence modeling – Velocity profile



Laminar flow profile



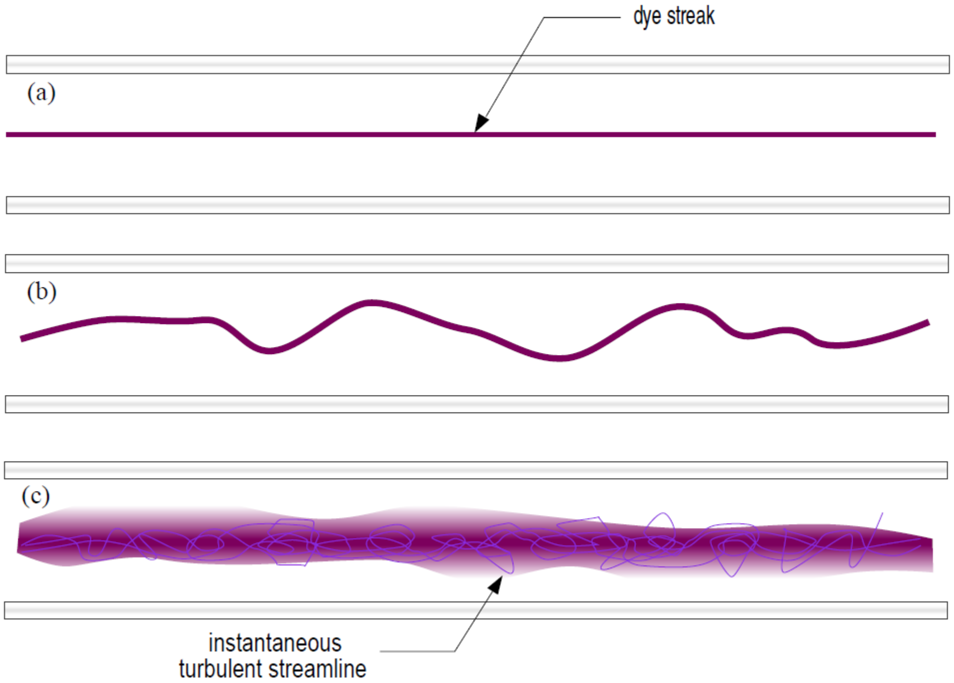
Averaged turbulent flow profile


Viscous sublayer - Laminar flow
Turbulent profile
Viscous sublayer - Laminar flow

Instantaneous turbulent flow profile

- In the laminar flow case, the velocity gradients close to the walls are low and the velocity profile is parabolic.
- Turbulence has a direct effect on the velocity profiles and mixing of transported quantities.
- The turbulent case shows two regions. One thin region close to the walls with very large velocity gradients, and a region far from the wall where the velocity profile is nearly uniform.
- The thin region close to the walls is laminar.
- Far from the flows, the flow becomes turbulent.

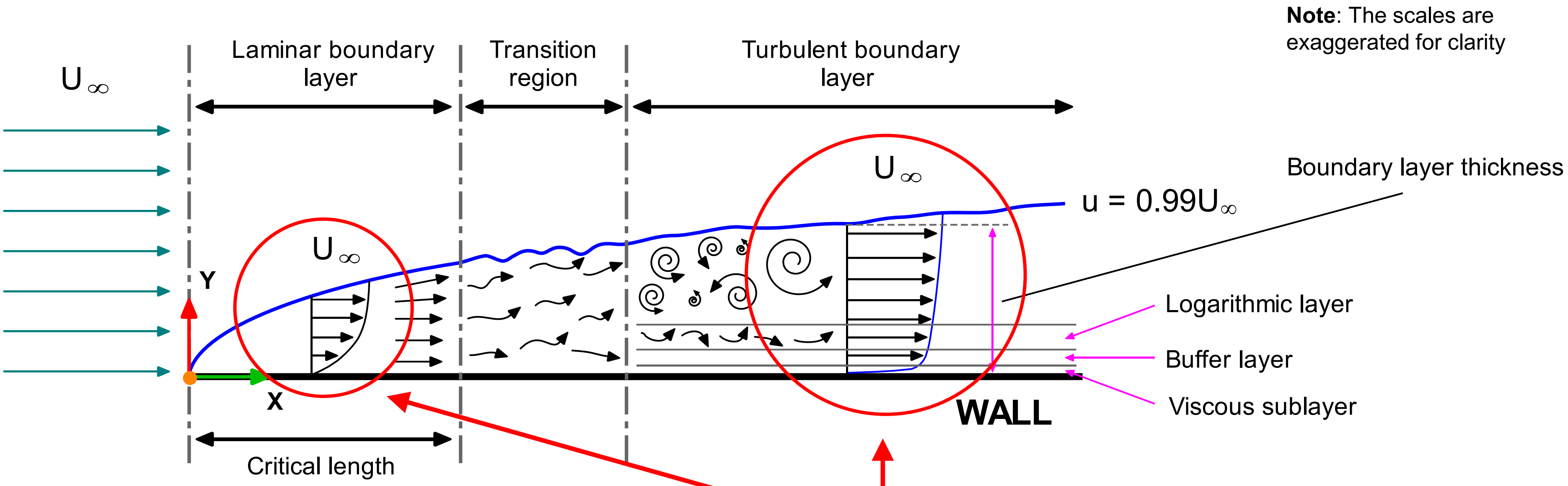
Turbulence modeling – Mixing of transported quantities



Flow in a pipe. (a) Laminar, (b) Transitional, (c) Turbulent

- Turbulence has a direct effect on the velocity profiles and mixing of transported quantities.
- Case (a) correspond to a laminar flow, where the dye can mix with the main flow only via molecular diffusion, this kind of mixing can take very long times.
- Case (b) shows a transitional state where the dye streak becomes wavy, but the main flow still is laminar.
- Case (c) shows the turbulent state, where the dye streak changes direction erratically, and the dye has mixed significantly with the main flow due to the velocity fluctuations.

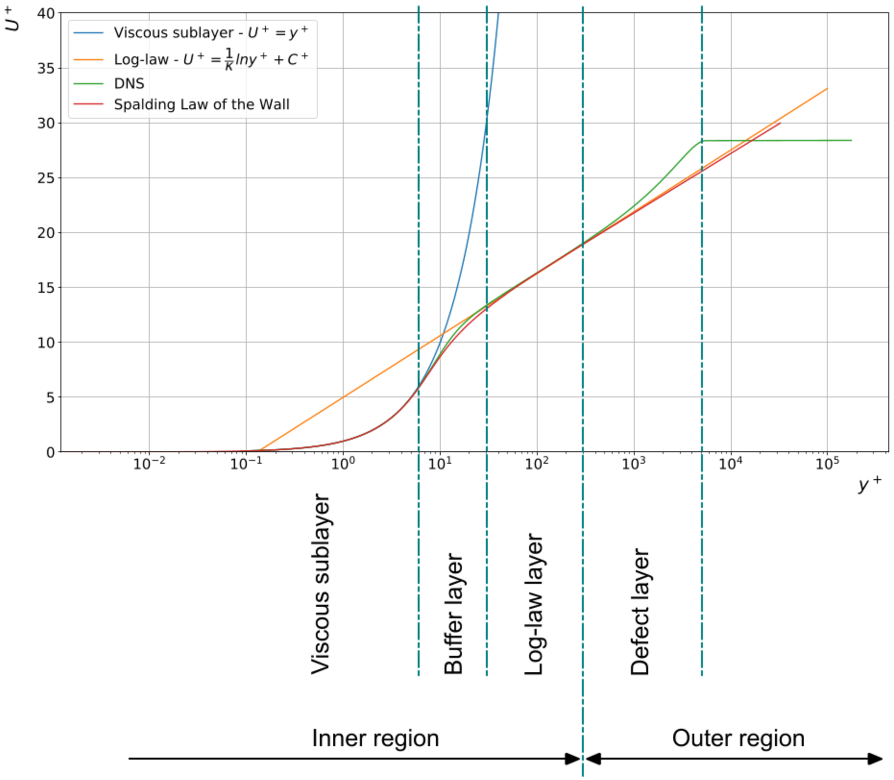
Turbulence near the wall – Boundary layer



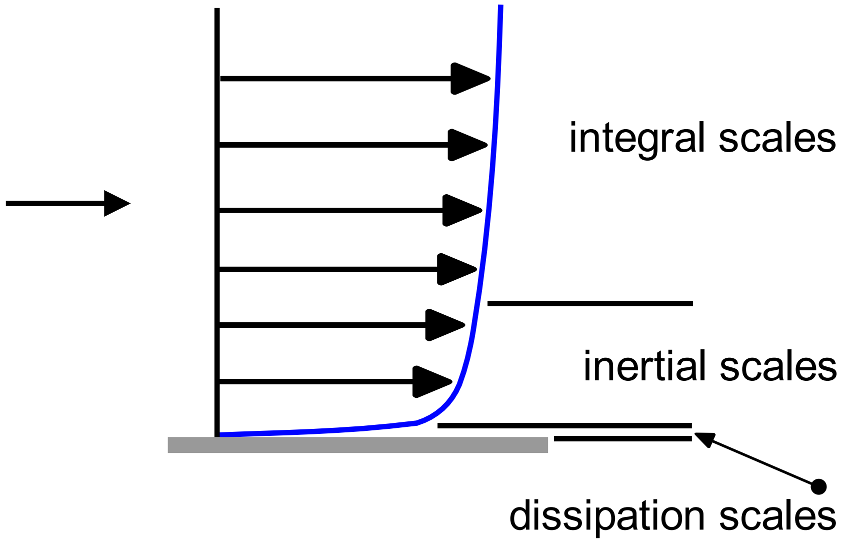
Actual profile - Physical velocity profile

- Near walls, in the boundary layer, the velocity changes rapidly.
- A laminar boundary layer starts to form at the leading edge. As the flow proceeds further downstream, large shear stresses and velocity gradient develop within the boundary layer. At one point, the flow becomes turbulent.
- In CFD, we try to avoid the transition region and the buffer layer. What is happening in this region is not well understood. The flow can become laminar again or can become turbulent.
- The velocity profiles in the laminar and turbulent regions are different.
- Turbulence models require different considerations depending on whether you solve the viscous sublayer, model the log-law layer, or solve the whole boundary layer.

Turbulence near the wall - Law of the wall



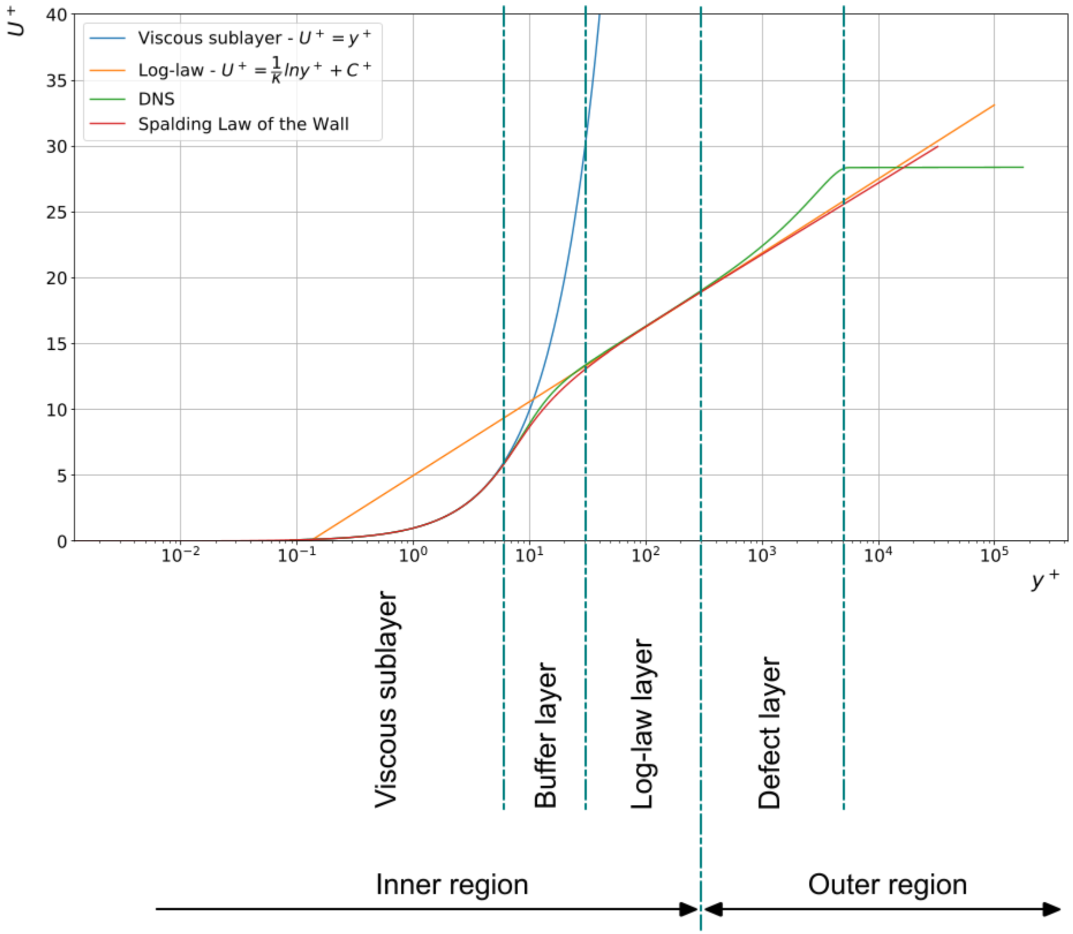
Non-dimensional profile



Actual profile - Physical velocity profile

- The use of the non-dimensional velocity u+ and non-dimensional distance from the wall y+, results in a predictable boundary layer profile for a wide range of flows.
- Turbulence models require different considerations depending on whether you solve the viscous sublayer of model the log-law layer.

Turbulence near the wall - Definition of y^+ and u^+



$$y^+ = \frac{\rho \times U_\tau \times y}{\mu} = \frac{U_\tau \times y}{\nu}$$

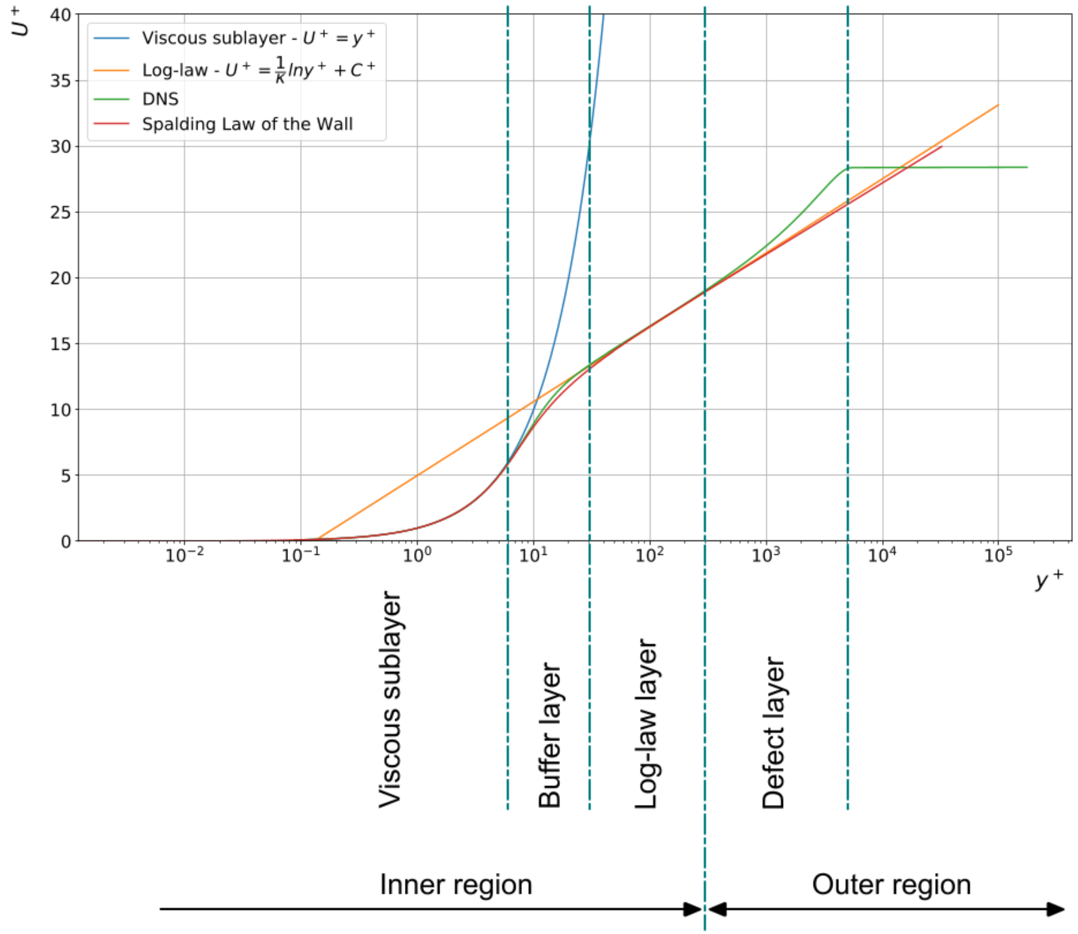
$$U_\tau = \sqrt{\frac{\tau_w}{\rho}}$$

$$u^+ = \frac{U}{U_\tau}$$

Where y is the distance normal to the wall, U_τ is the shear velocity, and u^+ relates the mean velocity to the shear velocity

- y^+ or wall distance units is a very important concept when dealing with turbulence modeling.
- Remember this definition as we are going to use it a lot.

Turbulence near the wall - Relations according to y^+ value



Note 1: the range of y^+ values might change from reference to reference but roughly speaking they are all close to these values.

Note 2: the y^+ upper limit of the buffer layer depends on the Reynolds number. Large Re will have higher y^+ upper limit

Buffer layer

$$30 < y^+ < 300$$

$$u^+ = \frac{1}{\kappa} \ln y^+ + C^+$$

$$\kappa \approx 0.41 \quad C^+ \approx 5.0$$

Viscous sublayer

$$y^+ < 6$$

$$u^+ = y^+$$

Logarithmic layer (log-law)

$$6 < y^+ < 30$$

$$u^+ \neq y^+$$

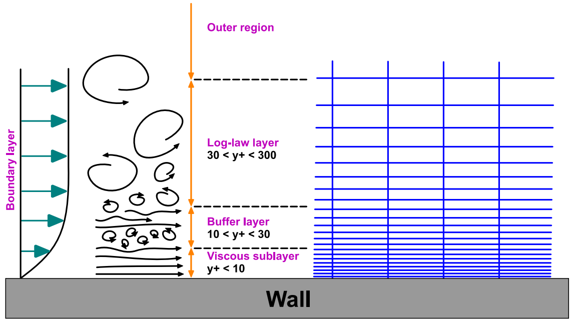
$$u^+ \neq \frac{1}{\kappa} \ln y^+ + C^+$$

Near-wall treatment and wall functions

- When dealing with wall turbulence, we need to choose a near-wall treatment.

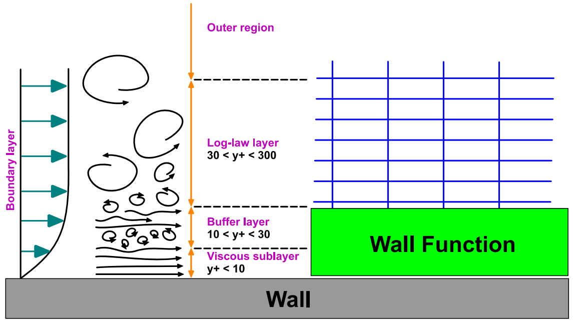
Wall resolving approach

- If you want to resolve the boundary layer up to the viscous sub-layer you need very fine meshes close to the wall.
- In terms of y+, you need to cluster at least 8-10 layers at y+ < 6-10.
- But for good accuracy, usually you will use 15 to 30 layers, with a low growth rate.
- You need to properly resolve the velocity profile.
- This is the most accurate approach, but it is computationally expensive.



Wall resolving approach

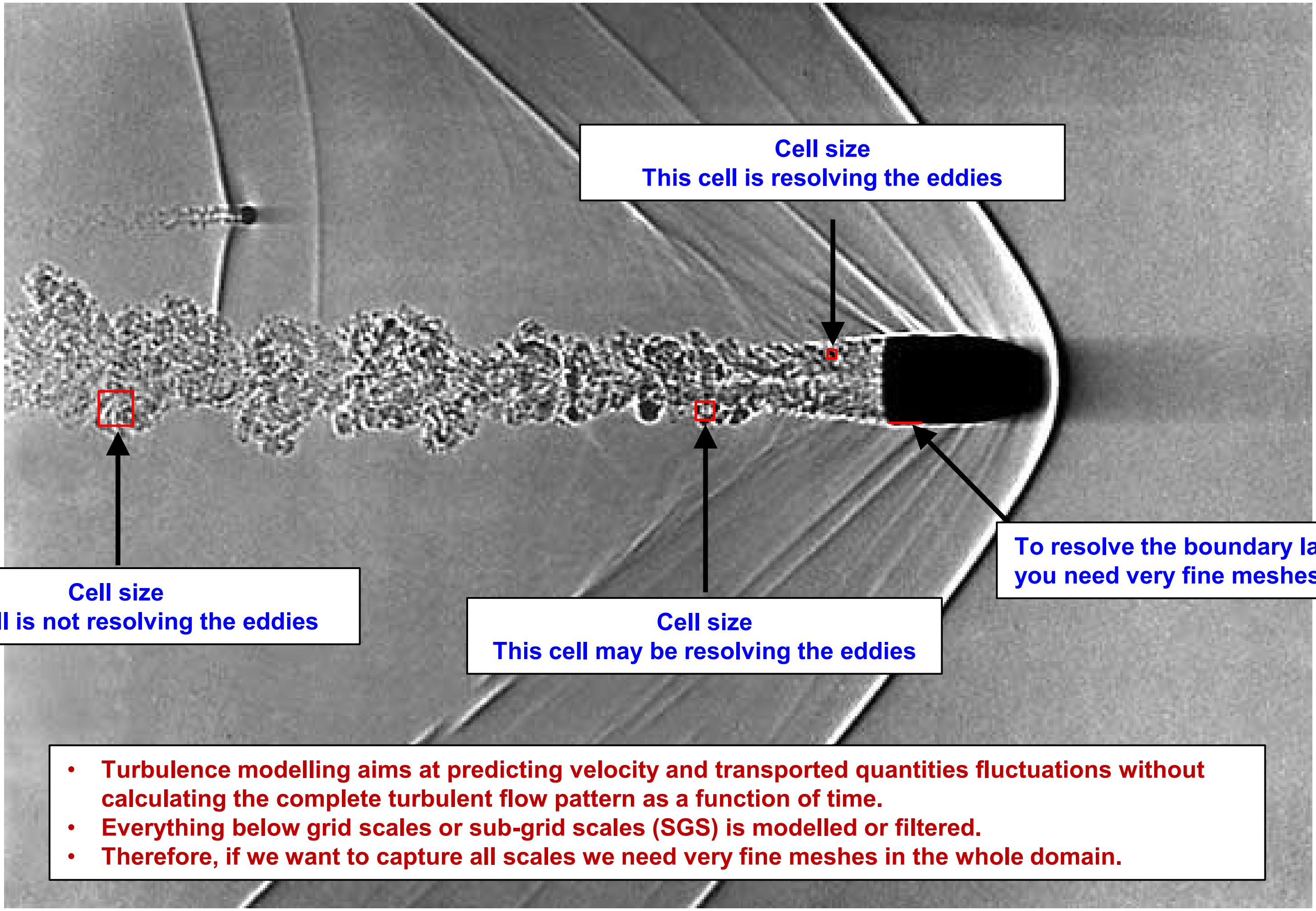
- If you are not interested in resolving boundary layer up to the viscous sub-layer, you can use wall functions.
- In terms of y+, wall functions will model everything for y+ < 30.
- You will need to cluster at least 5 to 10 layers to resolve the profiles (U and k).
- This approach use coarser meshes, but you should be aware of the limitations of the wall functions.



y^+ insensitive

- You can also use the y^+ insensitive wall treatment (sometimes known as continuous wall functions or scalable wall functions).
- This near-wall treatment is valid in the whole boundary layer.
- In terms of y+, you can use this approach for values between 1 < y+ < 300.
- This approach is very flexible as it is independent of the y^+ value but is not available in all turbulence models.
- You should also be aware of its limitations.

Turbulence modeling – Grid scales



Cell size
This cell is resolving the eddies

Cell size
This cell is not resolving the eddies

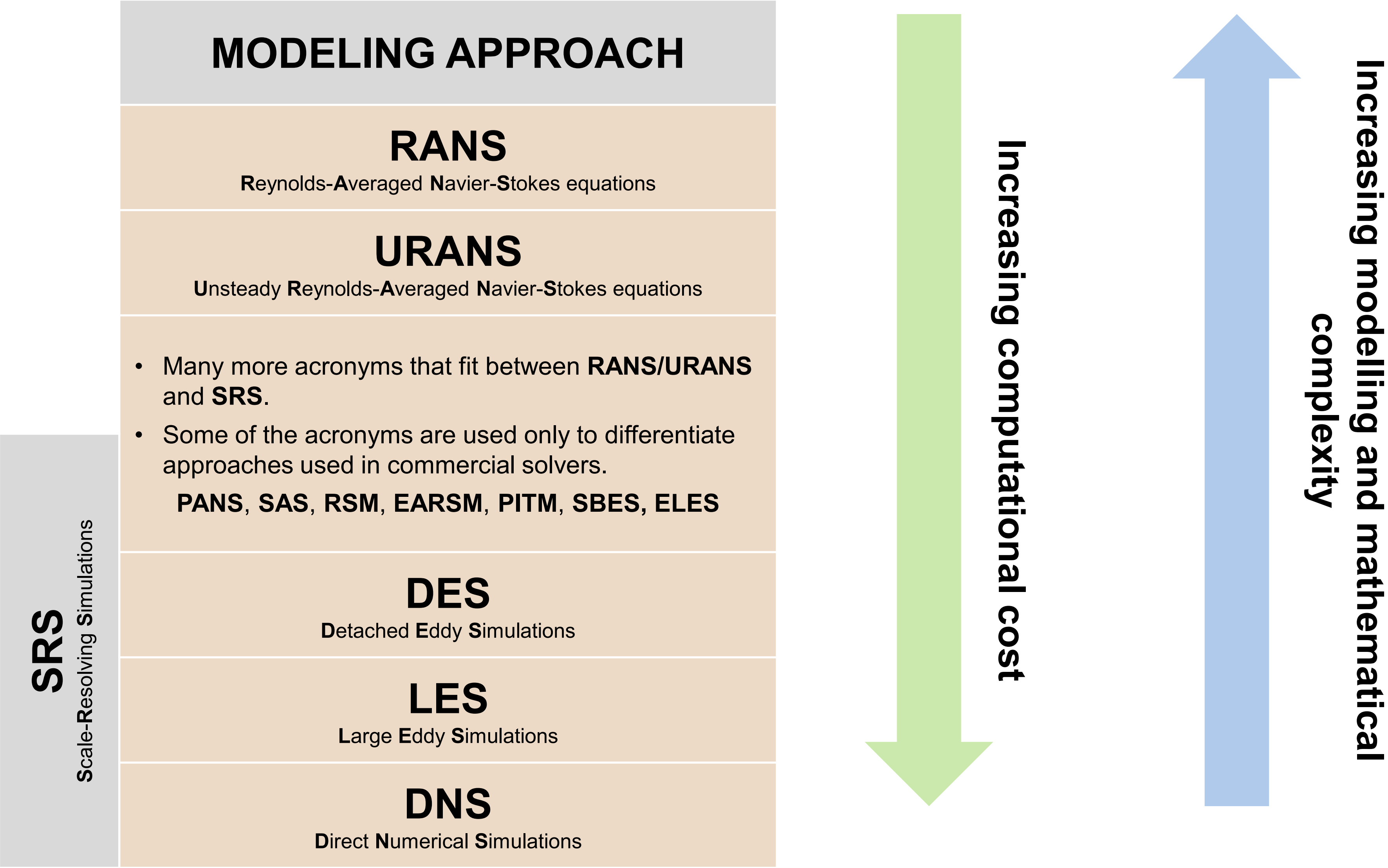
Cell size
This cell may be resolving the eddies

To resolve the boundary layer you need very fine meshes

- Turbulence modelling aims at predicting velocity and transported quantities fluctuations without calculating the complete turbulent flow pattern as a function of time.
- Everything below grid scales or sub-grid scales (SGS) is modelled or filtered.
- Therefore, if we want to capture all scales we need very fine meshes in the whole domain.

Bullet at Mach 1.5
Photo credit: Andrew Davidhazy. Rochester Institute of Technology.

Overview of turbulence modeling approaches



MODELING APPROACH

RANS

Reynolds-**A**veraged **N**avier-**S**tokes equations

URANS

Unsteady **R**eynolds-**A**veraged **N**avier-**S**tokes equations

- Many more acronyms that fit between **RANS/URANS** and **SRS**.
- Some of the acronyms are used only to differentiate approaches used in commercial solvers.

PANS, **SAS**, **RSM**, **EARSM**, **PITM**, **SBES, ELES**

DES

Detached **E**ddy **S**imulations

LES

Large **E**ddy **S**imulations

DNS

Direct **N**umerical **S**imulations

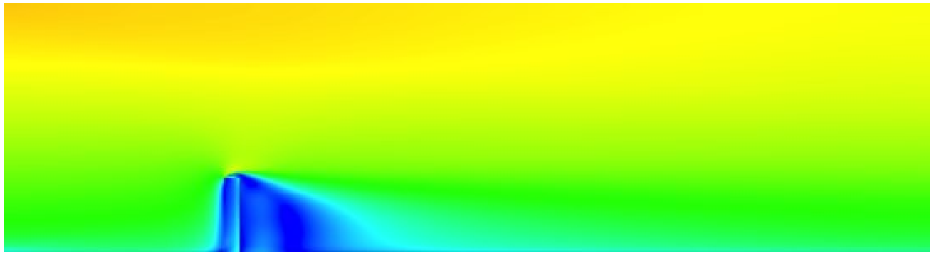
SRS

Scale-**R**esolving **S**imulations

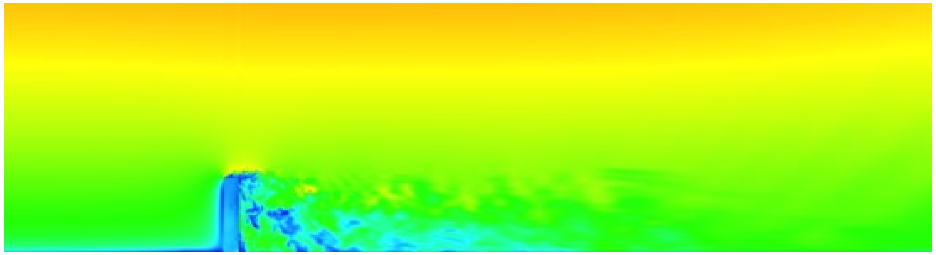
Increasing computational cost

Increasing modelling and mathematical complexity

Overview of turbulence modeling approaches



RANS



LES (Instantaneous field)

RANS/URANS	DES/LES	DNS
• Solve the time-average NSE. • All turbulent spatial scales are modeled. • Many models are available. One equation models, two equation models, Reynolds stress models, transition models, and so on. • This is the most widely approach for industrial flows. • Unsteady RANS (URANS), use the same equations as the RANS but with the transient term retained. • It can be used in 2D and 3D cases.	• Solve the filtered unsteady NSE. • Sub-grid scales (SGS) are filtered, grid scales (GS) are resolved. • Aim at resolving the temporal scales, hence requires small time-steps. • For most industrial applications, it is computational expensive. However, thanks to the current advances in parallel and scientific computing it is becoming affordable. • Many models are available. • It is intrinsically 3D and asymmetric.	• Solves the unsteady laminar NSE. • Solves all spatial and temporal scales; hence, requires extremely fine meshes and small time-steps. • No modeling is required. • It is extremely computational expensive. • Not practical for industrial flows. • It is intrinsically 3D and asymmetric.

Short description of some RANS turbulence models

Model	Short description
Spalart-Allmaras	Suitable for external aerodynamics, tubomachinery and high-speed flows. Good for mildly complex external/internal flows and boundary layer flows under pressure gradient (e.g. airfoils, wings, airplane fuselages, ship hulls). Performs poorly for free shear flows and flows with strong separation.
Standard k–epsilon	Robust. Widely used despite the known limitations of the model. Performs poorly for complex flows involving severe pressure gradient, separation, strong streamline curvature. Suitable for initial iterations, initial screening of alternative designs, and parametric studies.
Realizable k–epsilon	Suitable for complex shear flows involving rapid strain, moderate swirl, vortices, and locally transitional flows (e.g. boundary layer separation, massive separation, and vortex shedding behind bluff bodies, stall in wide-angle diffusers, room ventilation). It overcome the limitations of the standard k-epsilon model.
Standard k–omega	Superior performance for wall-bounded boundary layer, free shear, and low Reynolds number flows compared to models from the k-epsilon family. Suitable for complex boundary layer flows under adverse pressure gradient and separation (external aerodynamics and turbomachinery).
SST k–omega	Offers similar benefits as standard k–omega. Not overly sensitive to inlet boundary conditions like the standard k–omega. Provides more accurate prediction of flow separation than other RANS models. Probably the most widely used RANS model.

Turbulence modeling – Starting equations

$$\textbf{NSE} \quad \left\{ \begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0 \\ \frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) &= -\nabla p + \nabla \cdot \boldsymbol{\tau} + \mathbf{S}_u \\ \frac{\partial (\rho e_t)}{\partial t} + \nabla \cdot (\rho e_t \mathbf{u}) &= \nabla \cdot q - \nabla \cdot (p \mathbf{u}) + \boldsymbol{\tau} \mathbf{:} \nabla \mathbf{u} + \mathbf{S}_e \end{aligned} \right.$$

$$+$$

Additional equations to close the system (thermodynamic variables)

Additionally, relationships to relate the transport properties

Additional closure equations for the turbulence models

- Turbulence models equations cannot be derived from fundamental principles.
- All turbulence models contain some sort of empiricism.
- Some calibration to observed physical solutions is contained in the turbulence models.
- Also, some intelligent guessing is used.
- A lot of uncertainty is involved!

Incompressible RANS equations

- The RANS equations are very similar to the starting equations.

$$\nabla \cdot (\bar{\mathbf{u}}) = 0$$

$$\frac{\partial \bar{\mathbf{u}}}{\partial t} + \nabla \cdot (\bar{\mathbf{u}}\bar{\mathbf{u}}) = -\frac{1}{\rho}(\nabla p) + \nu \nabla^2 \bar{\mathbf{u}} + \frac{1}{\rho}\nabla \cdot \boldsymbol{\tau}^R$$

If we retain this term, we talk about URANS equations and if we drop it we talk about RANS equations

RANS/URANS equations

$$\nabla \cdot (\mathbf{u}) = 0$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = \frac{-\nabla p}{\rho} + \nu \nabla^2 \mathbf{u}$$

NSE with no turbulence models (DNS)

- The differences are that all quantities have been averaged (the overbar over the primitive variables).
- And the appearance of the Reynolds stress tensor τ^R.
- Notice that the Reynolds stress tensor is not actually a stress, it must be multiplied by density in order to have dimensions corresponding to stresses,

$$\tau^R = -\rho\left(\overline{\mathbf{u}'\mathbf{u}'}\right)$$

- To derive the RANS equations we used Reynolds decomposition and a few averaging rules (a lot of algebra is involved),

$$\mathbf{u}(\mathbf{x},t) = \bar{\mathbf{u}}(\mathbf{x}) + \mathbf{u}'(\mathbf{x},t), \qquad p(\mathbf{x},t) = \bar{p}(\mathbf{x}) + p'(\mathbf{x},t)$$

Reynolds decomposition

Incompressible RANS equations

- The RANS approach to turbulence modeling requires the Reynolds stresses to be appropriately modeled.

$$\tau^R = -\rho\left(\overline{\mathbf{u}'\mathbf{u}'}\right) = -\begin{pmatrix} \rho\overline{u'u'} & \rho\overline{u'v'} & \rho\overline{u'w'} \\ \rho\overline{v'u'} & \rho\overline{v'v'} & \rho\overline{v'w'} \\ \rho\overline{w'u'} & \rho\overline{w'v'} & \rho\overline{w'w'} \end{pmatrix}$$

- The Reynolds stress tensor can be modeled using the Boussinesq hypothesis, Reynolds stress models, non-linear eddy viscosity models or algebraic models.
- Let us address the Boussinesq hypothesis which is the most widely used approach to model the Reynolds stress tensor.
- By using this hypothesis we can relate the Reynolds stress tensor to the mean velocity gradient such that,

$$\tau^R = -\rho\left(\overline{\mathbf{u}'\mathbf{u}'}\right) = 2\mu_T\bar{\mathbf{D}}^R - \frac{2}{3}\rho\kappa\mathbf{I} = \mu_T\left[\nabla\bar{\mathbf{u}} + \left(\nabla\bar{\mathbf{u}}\right)^{\mathrm{T}}\right] - \frac{2}{3}\rho\kappa\mathbf{I},$$

- In the previous equation, $\bar{\mathbf{D}}^R = \frac{1}{2}(\nabla\bar{\mathbf{u}} + \nabla\bar{\mathbf{u}}^{\mathrm{T}})$ denotes the strain-rate tensor.
- $\mathbf{I}$ is the identity matrix.
- μ_T is the turbulent eddy viscosity.
- $\kappa = \frac{1}{2}\overline{\mathbf{u}'\cdot\mathbf{u}'}$ is the turbulent eddy viscosity.
- <u>At the end of the day we want to determine the turbulent eddy viscosity.</u>
- The turbulent eddy viscosity is not a fluid property, it is a property needed by the turbulence model.
- Each turbulence model will compute the turbulent eddy viscosity in a different way.

Incompressible RANS equations

- After introducing the Boussinesq approximation and doing some algebra, we obtain the following form of the governing equations,

$$\nabla \cdot (\bar{\mathbf{u}}) = 0$$

$$\frac{\partial \bar{\mathbf{u}}}{\partial t} + \nabla \cdot (\bar{\mathbf{u}}\bar{\mathbf{u}}) = -\frac{1}{\rho}\left(\nabla \bar{p} + \underbrace{\frac{2}{3}\rho \nabla k}_{\text{Normal stresses arising from the Boussinesq approximation}}\right) + \nabla \cdot \left[\frac{1}{\rho}\overbrace{(\mu + \underbrace{\mu_t}_{\text{Turbulent viscosity}})}^{\text{Effective viscosity}} \nabla \bar{\mathbf{u}}\right]$$

- Notice that by introducing the Boussinesq approximation the fluctuating quantities (the prime in the equations) do not appear in the final equations.
- The new equations are expressed entirely n terms of mean values (overbar), which can be computed.
- The problem now reduces to computing the turbulent eddy viscosity μ_T in the momentum equation.
- This is done by adding closure models (one-equation, two-equations, algebraic, transition, Reynolds stress, and so on).

Additional remarks

- We just outlined the incompressible RANS.
- The compressible RANS equations are similar, but when we derive them, we use Favre average (which can be seen as a mass-weighted averaging), instead of Reynolds average.
- Besides RANS, there is also LES and DES turbulence models.
- The idea behind LES/DES models is very similar to RANS, but instead of using averaging we filter the equations in space, and we solve the temporal scales
- At the end of the day, in LES/DES it is also required to model a stress tensor, usually called the SGS stress tensor.
- This stress tensor is related to the scales that cannot be resolved with the mesh; therefore, need to be modelled.
- LES/DES models are intrinsically unsteady and three-dimensional.
- Let us take a look at the governing equations of the $\kappa - \omega$ RANS model (Wilcox 1998 revision).
- Remember, the main goal of the RANS turbulence models is to model the Reynolds stress tensor by computing the turbulent eddy viscosity.

$\kappa - \omega$ Turbulence model overview (Wilcox 1998 revision)

- It is called $\kappa - \omega$ because it solves two additional equations for modeling the turbulent eddy viscosity, namely, the turbulent kinetic energy κ and the specific kinetic energy ω .

$$\nabla_t \rho k + \nabla \cdot (\rho \bar{\mathbf{u}} k) = \overbrace{\tau^R : \nabla \bar{\mathbf{u}}}^{\text{Production}} - \overbrace{\beta^* \rho \omega k}^{\text{Dissipation}} + \overbrace{\nabla \cdot [(\mu + \sigma_k \mu_t) \nabla k]}^{\text{Diffusion}}$$

$$\nabla_t \rho \omega + \nabla \cdot (\rho \bar{\mathbf{u}} \omega) = \underbrace{\alpha \frac{\omega}{k} \tau^R : \nabla \bar{\mathbf{u}}}_{\text{Production}} - \underbrace{\beta \rho \omega^2}_{\text{Dissipation}} + \underbrace{\nabla \cdot [(\mu + \sigma_\omega \mu_t) \nabla \omega]}_{\text{Diffusion}}$$

- These are the closure equations of the turbulence problem using the $\kappa - \omega$ RANS model.
- These are not physical quantities. They kind of represent the generation and destruction of turbulence.
- In the $\kappa - \omega$ model, the turbulent eddy viscosity can be computed as follows,

$$\mu_T = \frac{\rho \kappa}{\omega}$$

- The model has many closure coefficient that we do not show here. These coefficients are calibrated using experimental data, DNS simulations, analytical solutions, or empirical data.
- Note that all quantities are computed in function of mean values. The Reynolds stresses are modeled using the Boussinesq hypothesis. By proceeding in this way, we remove any dependence on the fluctuations.
- It is worth mentioning that different turbulence models will have different ways of computing the turbulent eddy viscosity.

$\kappa - \omega$ Turbulence model free stream initial conditions

- The initial value for the turbulent kinetic energy κ can be computed as follows,

$$\kappa = \frac{3}{2}(UI)^2$$

- The initial value for the specific kinetic energy ω can be computed as follows,

$$\omega = \frac{\rho\kappa}{\mu}\frac{\mu_t}{\mu}^{-1}$$

- Where μ_t/μ is the viscosity ratio and $I = u'/\bar{u}$ is the turbulence intensity.
- If you are totally lost, you can use these reference values. They work most of the times, but it is a good idea to have some experimental data or a better initial estimate.

	Low	Medium	High
I	1.0 %	5.0 %	10.0 %
μ_t/μ	1	10	100

- By the way, use these guidelines for external aerodynamics only.

$\kappa - \omega$ Turbulence model boundary conditions at the walls

- Follow these guidelines to find the boundary conditions for the near-wall treatment.
- We highly recommend you to read the source code and find the references used to implement the model.
- As for the free-stream boundary conditions, you need to give the boundary conditions for the near-wall treatment.
- When it comes to near-wall treatment, you have three options:
  - Use wall functions:

$$30 \leq y^+ \leq 300$$

  - y+ insensitive wall functions, this only applies to the $\kappa - \omega$ family of model:

$$1 \leq y^+ \leq 300$$

  - Resolve the boundary layer (no wall functions):

$$y^+ \leq 6$$

$\kappa - \omega$ Turbulence model boundary conditions at the walls

- Remember, you can only use wall functions if the primitive patch (the patch type defined in the `boundary` dictionary), is of type **wall**.

Field	Wall functions – High RE	Resolved BL – Low RE
nut	**nut(–)WallFunction*** or **nutUSpaldingWallFunction**** (with 0 or a small number)	**nutUSpaldingWallFunction**** or **nutLowReWallFunction** or **fixedValue** (with 0 or a small number)
k, q, R	**kqRWallFunction** $k_{wall} = k$ or $k_{wall} = 1e-10$	**kqRWallFunction** or **kLowReWallFunction** $k_{wall} = k$ or $k_{wall} = 1e-10$
epsilon	**epsilonWallFunction** (with inlet value) $\epsilon_{wall} = \epsilon$	**epsilonWallFunction** (with inlet value) or **zeroGradient** or **fixedValue** (with 0 or a small number)
omega	**omegaWallFunction** $\omega_{wall} = 10 \dfrac{6\nu}{\beta y^2}$	**omegaWallFunction**** or **fixedValue** $\omega_{wall} = 10 \dfrac{6\nu}{\beta y^2}$
nuTilda	**–**	**fixedValue** (with 0 or a small number)

* **`$WM_PROJECT_DIR/src/TurbulenceModels/turbulenceModels/derivedFvPatchFields/wallFunctions/nutWallFunctions`**

** For y^+ insensitive wall functions (continuous wall functions)

Turbulence models available in OpenFOAM®

- There are many turbulence models implemented in OpenFOAM®, from RANS to LES.
- You can also implement yours!
- You can find the turbulence models in the following directories:
  - **`$WM_PROJECT_DIR/src/MomentumTransferModels`**
- The wall functions are in the following directories:
  - **`$WM_PROJECT_DIR/src/MomentumTransferModels/momentumTransferModels/derivedFvPatchFields`**
- If you have absolutely no idea of what model to use, we highly recommend you the k-omega family models or the realizable k-epsilon model.
- Remember, when a turbulent flow enters a domain, turbulent boundary conditions and initial conditions must be specified.
- Also, if you are dealing with wall bounded turbulence you will need to choose the near-wall treatment.
- You can choose to solve the viscous sub-layer (low-Re approach) or use wall functions (high-Re approach).
- You will need to give the appropriate boundary conditions to the near-wall treatment.
- **Our task is to choose the less wrong model !**

y^+ wall distance units normal to the wall

- We never know a priori the y^+ value (because we do not know the friction velocity).
- What we usually do is to run the simulation for a few time-steps or iterations, and then we get an estimate of the y^+ value.
- After determining where we are in the boundary layer (viscous sub-layer, buffer layer or log-law layer), we take the mesh as a good one or we modify it if is deemed necessary.
- It is an iterative process and it can be very time consuming, as it might require remeshing and rerunning the simulation.
- Have in mind that it is quite difficult to get a uniform y^+ value at the walls.
- Try to get a y^+ mean value as close as possible to your target.
- Also, check that you do not get very high maximum values of y^+ (more than a 1000)
- Values up to 300 are fine. Values larger that 300 and up to a 1000 are acceptable is they do not covert a large surface (no more than 10% of the total wall area), or they are not located in critical zones.
- Remember, the upper limit of y^+ also depends on the Reynolds number.
- Use common sense when accessing y^+ value.

Estimating normal wall distance

- To get an initial estimate of the distance from the wall to the first cell center y⁺, without recurring to a precursor simulation, you can proceed as follows,

1. $$Re = \frac{\rho \times U \times L}{\mu}$$

2. $$C_f = 0.058 \times Re^{-0.2}$$ (Skin friction coefficient of a flat plate, there are similar correlations for pipes)

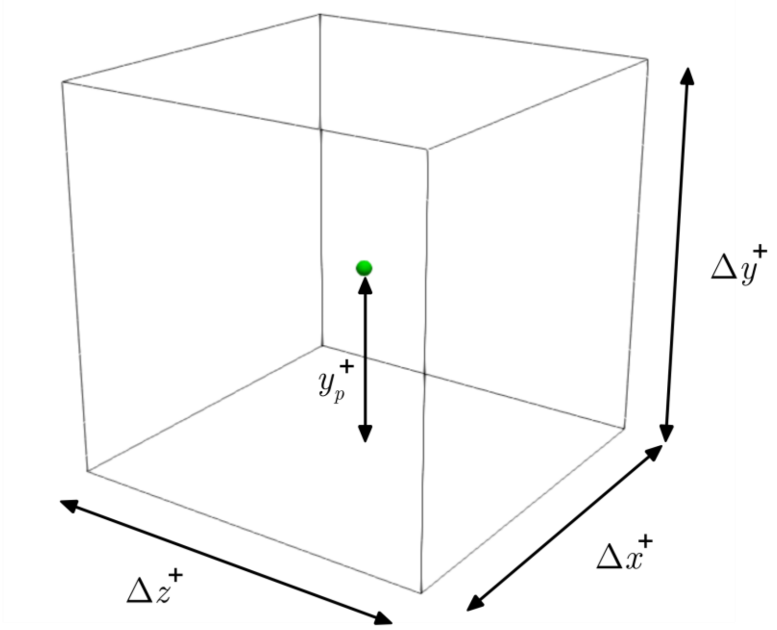
3. $$\tau_w = \frac{1}{2} \times C_f \times \rho \times U_{\infty}^2$$

4. $$U_\tau = \sqrt{\frac{\tau_w}{\rho}}$$

5. $$y = \frac{\mu \times y^+}{\rho \times U_\tau}$$ Your desired value

- You will find a simple calculator for the wall distance estimation in the following link: http://www.wolfdynamics.com/tools.html?id=2

Wall distance units $x^+ – y^+ – z^+$



- Similar to y^+, the wall distance units can be computed in the stream-wise (Δx^+) and span-wise (Δz^+) directions.
- The wall distance units in the stream-wise and span-wise directions can be computed as follows:

$$\Delta x^+ = \frac{U_\tau \Delta x}{\nu} \qquad \Delta z^+ = \frac{U_\tau \Delta z}{\nu}$$

- And recall that y^+ is computed at the cell center, therefore:

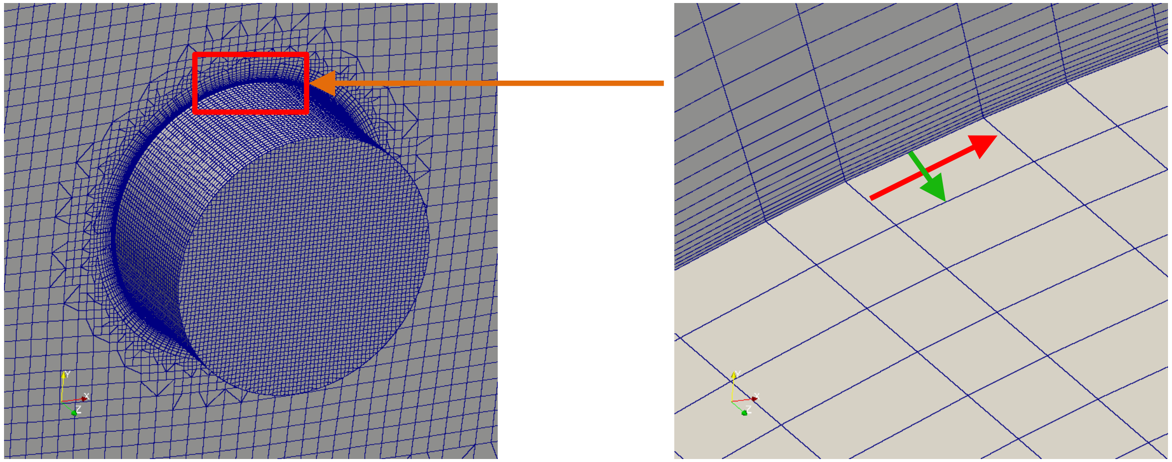
$$\Delta y^+ = 2 \times y^+$$

$$(\Delta x^+, \Delta y^+, \Delta z^+) = \left(\frac{x}{l_\tau}, \frac{y}{l_\tau}, \frac{z}{l_\tau}\right)$$

where $l_\tau = \frac{\nu}{U_\tau}$

Viscous length

Wall distance units and some rough estimates



- Similar to y⁺, the wall distance units can be computed in the stream-wise (Δx^+) and span-wise (Δz^+) directions.
- DES and RANS simulations do not have stream-wise and span-wise wall distance units requirements as in LES simulations. Therefore, they are more affordable.
- Typical requirements for LES are (these are approximations based on different references):

$$\Delta x^+ < 50, \; \Delta z^+ < 50 \qquad \text{for} \qquad y^+ < 6$$

Wall resolving

$$\Delta x^+ < 4\Delta y^+, \; \Delta z^+ < 4\Delta y^+ \qquad \text{for} \qquad 30 \leq y^+ \leq 300$$

Wall modeling

Turbulence modeling guidelines and tips

- Compute Reynolds number and determine whether the flow is turbulent.
- Try to avoid the use of turbulent models with laminar flows.
- Choose the near-wall treatment and estimate y before generating the mesh.
- Run the simulation for a few time steps and get a better prediction of y and correct your initial prediction of y^+.
- The realizable $\kappa - \epsilon$ or $\mathrm{SST}\ \kappa - \omega$ models are good choices for general applications.
- The standard $\kappa - \epsilon$ model is very reliable, you can use it to get initial values. Have in mind that this model use wall functions.
- If you are interested in resolving the large eddies and modeling the smallest eddies, DES or LES are the right choice.
- If you do not have any restriction in the near wall treatment method, use wall functions (even with LES/DES models).
- Be aware of the limitations of the turbulence model chosen, find and read the original references used to implement the model in OpenFOAM®.
- Set reasonable boundary and initial conditions for the turbulence model variables.

Turbulence modeling guidelines and tips

- Always monitor the turbulent variables, some of them are positive bounded.
- Avoid strong oscillations of the turbulent variables.
- If you are doing LES, remember that these models are intrinsically 3D and unsteady. You should choose your time-step in such a way to get a CFL of less than 1 and preferably of about 0.5.
- If you are doing RANS with wall functions, it is perfectly fine to use upwind to discretize the turbulence closure equations. After all, turbulence is a dissipative process. However, some authors may disagree with this, make your own conclusions.
- On the other hand, if you are using a wall resolved approach, it is better to use a high-order discretization scheme to discretize the turbulence closure equations.
- If you are doing unsteady simulations, always remember to compute the average values (ensemble average).
- If you are dealing with external aerodynamics and detached flows, DES simulations are really affordable.
- The work-horse of turbulence modeling in CFD, **RANS**

Turbulence modeling hands-on tutorials

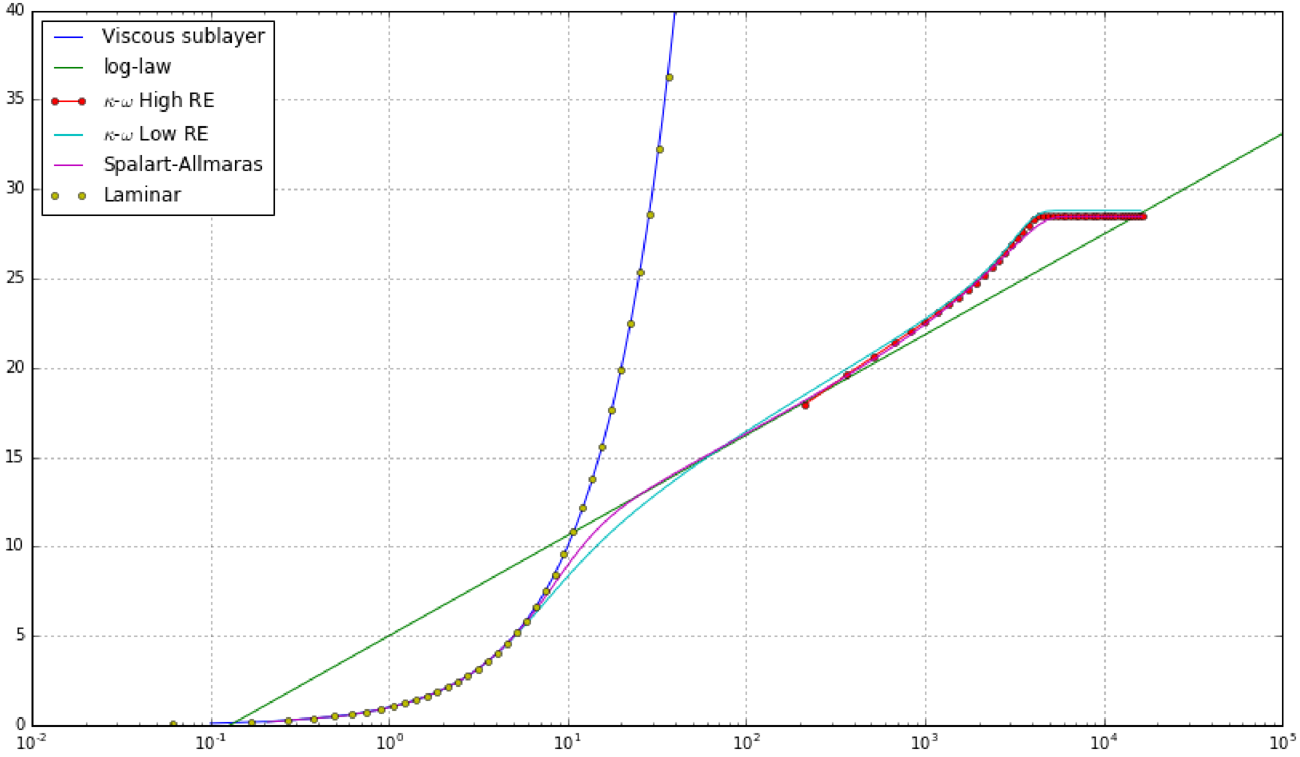
- Laminar-Turbulent flat plate
- Let us run this case. Go to the directory:

```
$PTOFC/advanced_physics/turbulence/flatPlate
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Turbulence modeling hands-on tutorials

Laminar-Turbulent flat plate



- The best way to understand the near the wall treatment and the effect of turbulence near the walls, is by reproducing the law of the wall.
- By plotting the velocity in terms of the non-dimensional variables u+ and y+, we can compare the profiles obtained from the simulations with the theoretical profiles.

Laminar-Turbulent flat plate

- In the directory **`python`** of each case, you will find a jupyter notebook (a python script), that you can use to plot the non-dimensional u^+ and y^+ profiles.
- The notebook uses some precomputed results, but you can adjust it to any case.
- Remember, the u^+ vs. y^+ plot is kind of a universal plot.
- It does not matter your geometry or flow conditions, if you are resolving well the turbulent flow, you should be able to recover this profile.
- To compute this plot, you must sample the wall shear stresses and the velocity along a line normal to the wall.
- Then, you can compute the shear velocity, friction coefficient, and u^+ and y^+ values.

$$y^+ = \frac{\rho \times U_\tau \times y}{\mu} = \frac{U_\tau \times y}{\nu}$$

$$U^+ = \frac{U}{U_\tau}$$

$$U_\tau = \sqrt{\frac{\tau_w}{\rho}}$$

$$c_f = \frac{\tau_w}{0.5 \rho U_\infty^2}$$

Turbulence modeling hands-on tutorials

Laminar-Turbulent flat plate

- We are going to use the following solver: `simpleFoam` (for RANS).
- This case is rather simple, but we will use it to explain many features used in OpenFOAM® when dealing with turbulence, especially when dealing with near the wall treatment.
- We will also show you how to do the post-processing in order to reproduce the law of the wall. For this, we will use a jupyter notebook (a python script).
- Remember, as we are introducing new closure equations for the turbulence problem, we need to define initial and boundary conditions for the new variables.
- We also need to define the discretization schemes and linear solvers to use to solve the new variables.
- It is also a good idea to setup a few functionObjects, such as: y^+, minimum and maximum values, forces, time average, and online sampling.
- You will find the instructions of how to run this case in the file *`README.FIRST`* located in the case directory.

Turbulence modeling hands-on tutorials

Laminar-Turbulent flat plate

- We select the turbulence model in the *momentumTransport* dictionary file.
- This dictionary file is located in the directory **constant**.
- To select the K-Omega SST turbulence model,

```
simulationType  RAS;          ← RANS type simulation

RAS    ← RANS sub-dictionary
{
    RASModel        kOmegaSST;      ← RANS model to use

    turbulence      on;       ← Turn on/off turbulence.  Runtime modifiable

    printCoeffs     on;       ← Print coefficients at the beginning
}
```

- Remember, you need to assign boundary and initial conditions to the new variables (**k**, **omega**, and **nut**).

Turbulence modeling hands-on tutorials

Vortex shedding past square cylinder

- To define the wall functions, follow this table,

Field	Wall functions – High RE	Resolved BL – Low RE
nut	**nutUSpaldingWallFunction**	**fixedValue 0** or a small number
k	**kqRWallFunction**	**fixedValue 0** or a small number
omega	**omegaWallFunction** $\omega_{wall} = 10 \frac{6\nu}{\beta y^2}$	**omegaWallFunction** $\omega_{wall} = 10 \frac{6\nu}{\beta y^2}$

- Run using high-RE and low-RE approaches.
- Compute the initial values of the turbulent quantities using a turbulent intensity value equal to 1% and an eddy viscosity ratio equal to 1.
- After computing the solution with $\kappa - \omega$ model, try to setup the case using the standard $\kappa - \epsilon$ model and the realizable $\kappa - \epsilon$.

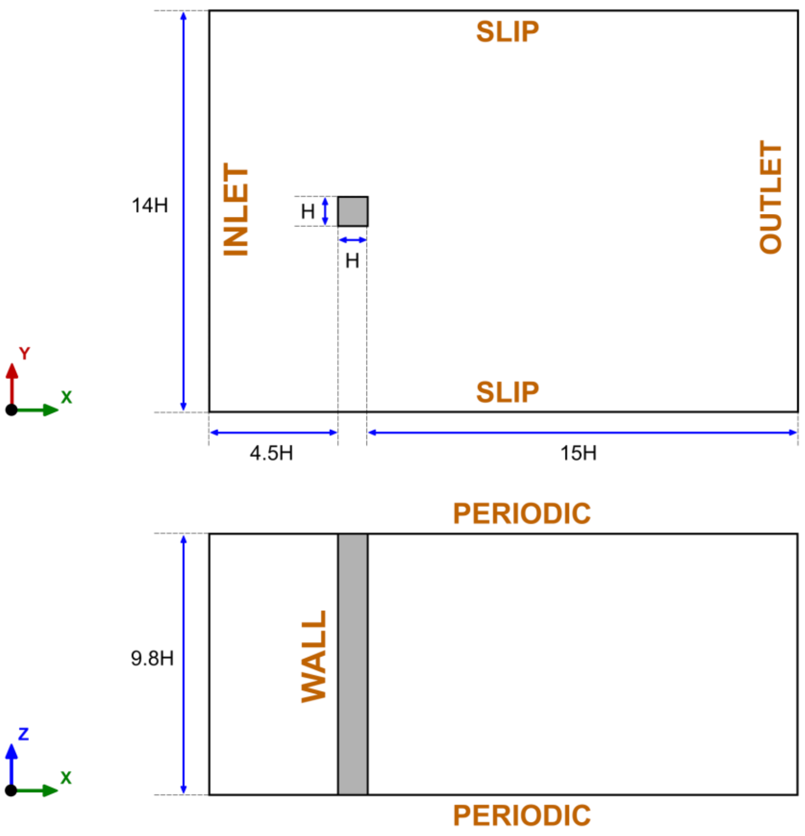
Turbulence modeling hands-on tutorials

- Vortex shedding past square cylinder
- Let us run this case. Go to the directory:

```
$PTOFC/advanced_physics/turbulence/squarecil
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend you to open the `README.FIRST` file and type the commands in the terminal, in this way, you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Vortex shedding past square cylinder



Physical and numerical side of the problem:

- The governing equations of the problem are the incompressible Navier-Stokes equations.
- To model the turbulence, we will use two approaches, LES and RANS.
- We are going to work in a 3D domain with periodic boundary conditions.
- This problem has plenty of experimental data for validation.

$$\mathbf{U}_{in} = 0.535\, m/s$$

$$\mathbf{H} = 0.04\, m$$

$$Re = 21400$$

Working fluid: Water

Turbulence modeling hands-on tutorials

Vortex shedding past square cylinder

Turbulence model	Drag coefficient	Strouhal number	Computing time (s)
Laminar	2.81	0.179	93489
LES	2.36	0.124	77465
DES	2.08	0.124	70754
SAS	2.40	0.164	57690
URANS (WF)	2.31	0.130	67830
URANS (No WF)	2.29	0.135	64492
RANS	2.30	-	28246 (10000 iter)
Experimental values	2.05-2.25	0.132	-

References:
Lyn, D.A. and Rodi, W., The flapping shear layer formed by flow separation from the forward corner of a square cylinder. *J. Fluid Mech., 267, 353, 1994.*
Lyn, D.A., Einav, S., Rodi, W. and Park, J.H., A laser-Doppler velocimetry study of ensemble-averaged characteristics of the turbulent near wake of a square cylinder. *Report. SFB 210 /E/100.*

Turbulence modeling hands-on tutorials

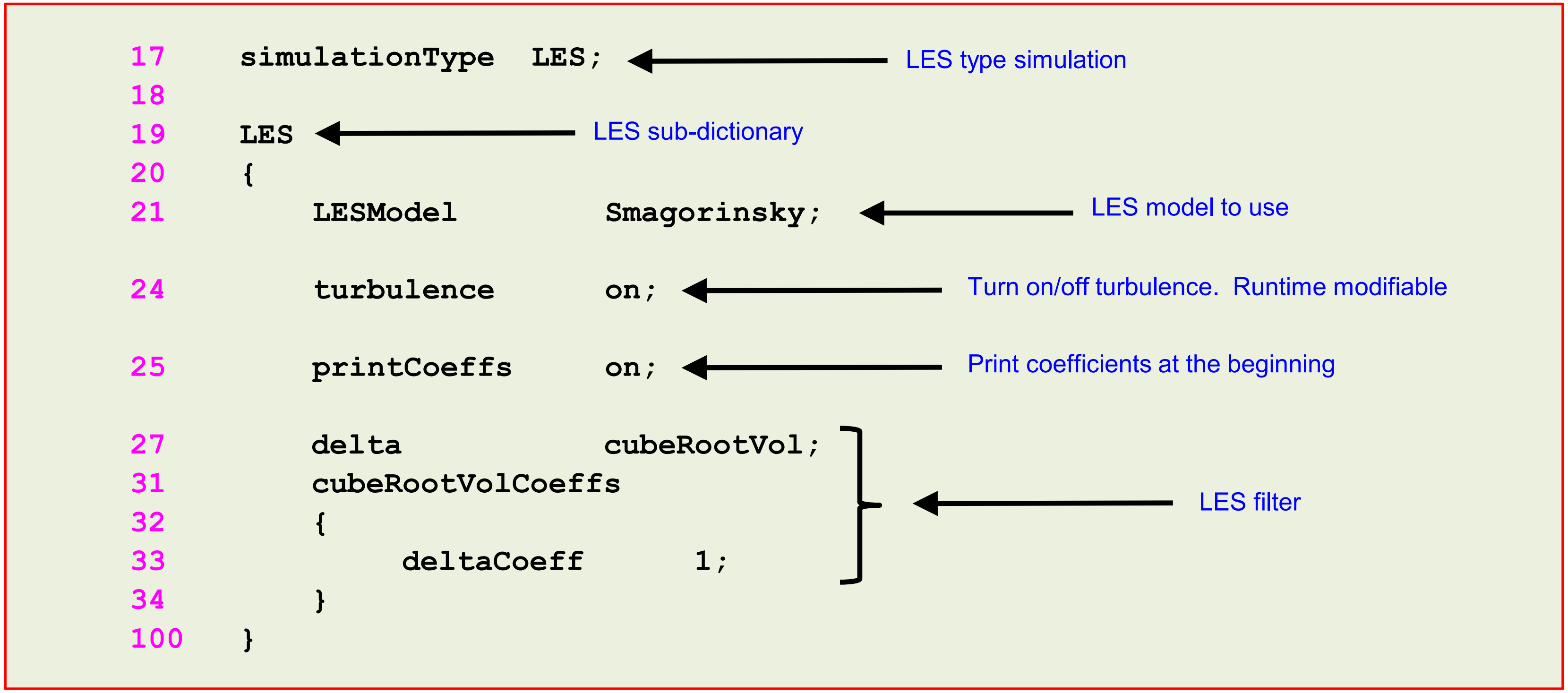
Vortex shedding past square cylinder

- We will use this case to learn how to setup a turbulent case (RANS and LES).
- To run this case we will use the solvers *simpleFoam* (steady solver) and *pimpleFoam* (unsteady solver).
- To get fast outcomes, we will use a coarse mesh. But feel free to refine the mesh, especially close to the walls.
- Remember, as we are introducing new closure equations for the turbulence problem, we need to define initial and boundary conditions for the new variables.
- We will use a few **functionObjects** to compute some additional quantities, such as, Q criterion, y^+, minimum and maximum values, forces, time average, and online sampling.
- After finding the solution, we will visualize the results.
- We will also compare the numerical solution with the experimental results.
- At the end, we will do some plotting and advanced post-processing using gnuplot and Python.
- Have in mind that the unsteady case will generate a lot of data.
- You will find the instructions of how to run this case in the file *README.FIRST* located in the case directory.

Turbulence modeling hands-on tutorials

Vortex shedding past square cylinder

- We select the turbulence model in the *momentumTransport* dictionary file.
- This dictionary file is located in the directory **constant**.
- To select the LES (Smagorinsky) turbulence model,



```
simulationType   LES;

LES
{
    LESModel        Smagorinsky;

    turbulence      on;

    printCoeffs     on;

    delta           cubeRootVol;
    cubeRootVolCoeffs
    {
        deltaCoeff      1;
    }
}
```

LES type simulation

LES sub-dictionary

LES model to use

Turn on/off turbulence. Runtime modifiable

Print coefficients at the beginning

LES filter

- Remember, you need to assign boundary and initial conditions to the new variables (**nut**).

Vortex shedding past square cylinder

- To define the wall functions, follow this table,

Field	**Wall functions – High RE**	**Resolved BL – Low RE**
nut	**nutUSpaldingWallFunction**	**fixedValue 0** or a small number
k	**kqRWallFunction**	**fixedValue 0** or a small number
omega	**omegaWallFunction** $\omega_{wall} = 10\frac{6\nu}{\beta y^2}$	**omegaWallFunction** $\omega_{wall} = 10\frac{6\nu}{\beta y^2}$

- Run using the following combinations of wall functions and compare the outcome.
  - Use High RE for RANS.
  - Use High RE and Low RE for URANS.
  - Use High RE and Low RE for LES.

Vortex shedding past square cylinder

- The initial value for the turbulent kinetic energy κ can be found as follows,

$$\kappa = \frac{3}{2}(UI)^2$$

- The initial value for the specific kinetic energy ω can be found as follows,

$$\omega = \frac{\rho\kappa}{\mu}\frac{\mu_t}{\mu}^{-1}$$

- Use the following initial estimates, $I = 5\%$ and $\frac{\mu_t}{\mu} = 10$

- At this point, we are ready to run. But before running, remember to setup the right numerics in the dictionary files *fvSolution* and *fvSchemes*.

- For the LES simulation, try to keep the CFL number below 0.9. For the URANS simulation, you can go as high as 10.

- Finally, do not forget to setup the **functionObjects** to compute the forces, average values, do the sampling, and compute y^+ on-the-fly.

Roadmap

A crash introduction to:

1. ~~Turbulence modeling in OpenFOAM®~~
2. **Multiphase flows modeling in OpenFOAM®**
3. Compressible flows in OpenFOAM®
4. Moving bodies in OpenFOAM®
5. Source terms in OpenFOAM®
6. Scalar transport pluggable solver

What is a multiphase flow?

- A multiphase flow is a fluid flow consisting of more than one phase component and have some level of phase separation above molecular level.
- Multiphase flows exist in many different forms. For example, two phase flows can be classified according to the state of the different phases:
  - Gas-Liquid mixture.
  - Gas-Solid mixture.
  - Liquid-Solid mixture.
  - Immiscible liquid-liquid.
- Multiphase flows are present in many industrial processes and natural systems.
- Hence the importance of understanding, modeling, and simulating multiphase flows.

Examples of multiphase flows

Municipal and industrial water treatment

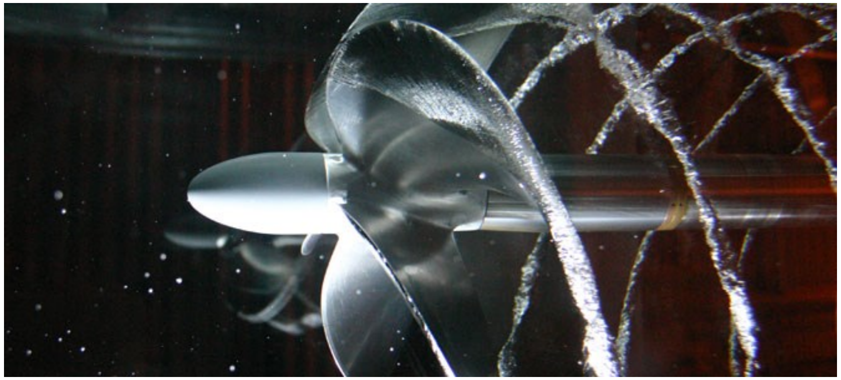
http://www.asiapacific.basf.com/apex/AP/en/upload/Press2010/BASF-Water-Chem-2010-Paper-Chem-2010-Intex-Shanghai

Cargo ship wake

http://developeconomies.com/development-economics/how-to-get-america-back-on-track-free-trade-edition/

Siltation & Sedimentation

http://blackwarriorriver.org/siltation-sedimentation/



Propeller cavitation

http://www.veempropellers.com/features/cavitationresistance

Examples of multiphase flows

Cooling Towers

https://whatiswatertreatment.wordpress.com/what-are-the-systems-associated-with-water-treatment-and-how-are-they-treated/103-2/

Volcano eruption

http://americanpreppersnetwork.com/2014/08/preparing-volcano-eruption.html

Fermentation of beer and spirits

http://www.distillingliquor.com/2015/02/05/how-to-make-alcohol-and-spirits/

Chemical reactor for the pharmaceutical and biotechnology industry

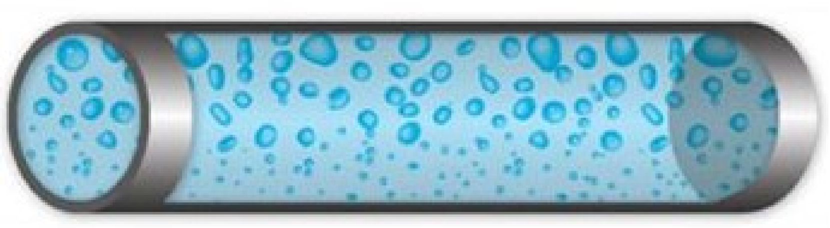
http://www.total-mechanical.com/Industrial/CaseStudies.aspx

Why simulating multiphase flows is challenging?

- Simulating multiphase flows is not an easy task.
- The complex nature of multiphase flows is due to:
    - More than one working fluid.
    - The transient nature of the flows.
    - The existence of dynamically changing interfaces.
    - Significant discontinuities (fluid properties and fluid separation).
    - Complicated flow field near the interface.
    - Interaction of small-scale structures (bubbles and particles).
    - Different spatial-temporal scales.
    - Dispersed phases and particle-particle interactions.
    - Mass transfer and phase change.
    - Turbulence.
    - Many models involved (drag, lift, heat transfer, turbulence dispersion, frictional stresses, collisions, kinetic theory, and so on).

Classifying multiphase flows according to phase morphology

- **Disperse system**: the phase is dispersed as non-contiguous isolated regions within the other phase (the continuous phase) . When we work with a disperse phase, we say that the system is dispersed: disperse-continuous flow.
- **Separated system**: the phase is contiguous throughout the domain and there is one well defined interphase with the other phase. When we work with continuous phases, we say that the system is separated: continuous-continuous flow.

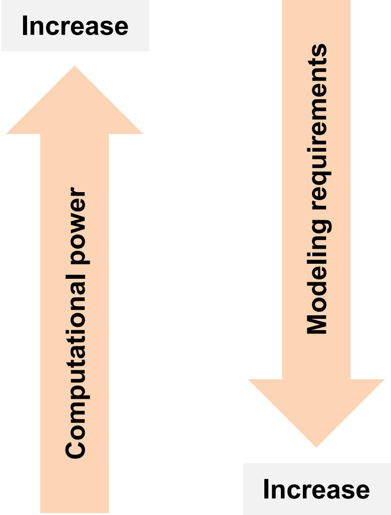


Dispersed system

Separated system

How to treat the wide range of behaviors in multiphase flows

- **Fully resolved:** solves complete physics. All spatial and temporal scales are resolved. Equivalent to DNS in turbulence modelling.
- **Eulerian-Lagrangian:** solves idealized isolated particles that are transported with the flow. One- or two-way coupling is possible. It can account for turbulence, momentum transfer, and mass transfer.
- **Eulerian-eulerians:** solves two or more co-existing fluids. The system can be dispersed or separated, and can account for turbulence, momentum transfer, and mass transfer.

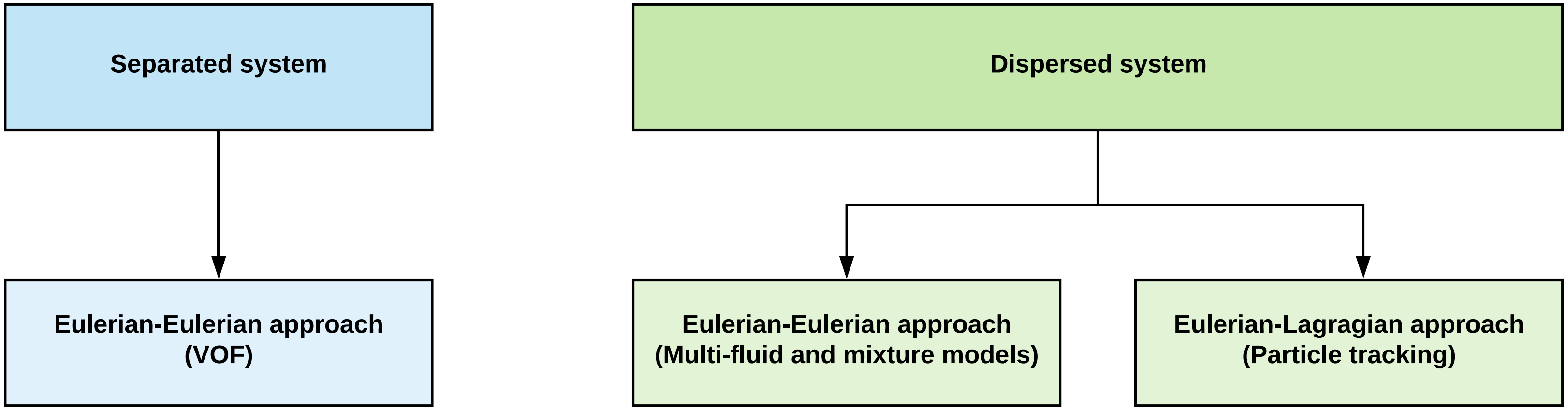


Increase

Computational power

Modeling requirements

Increase



Separated system

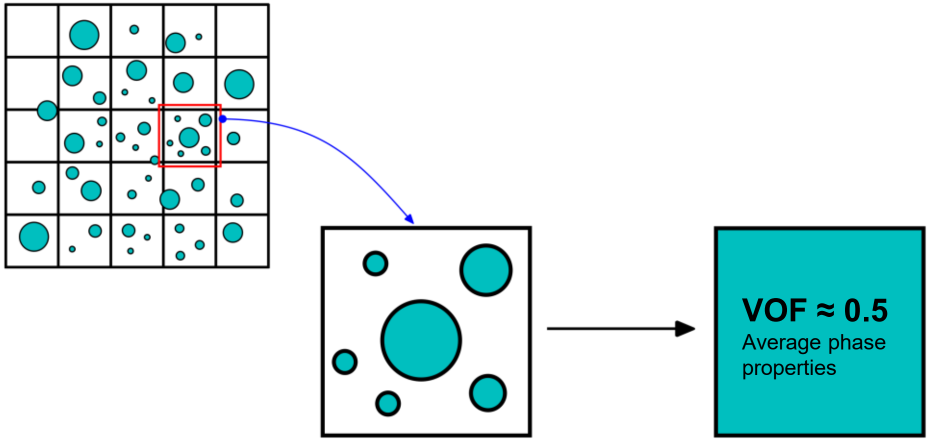
Eulerian-Eulerian approach (VOF)

Dispersed system

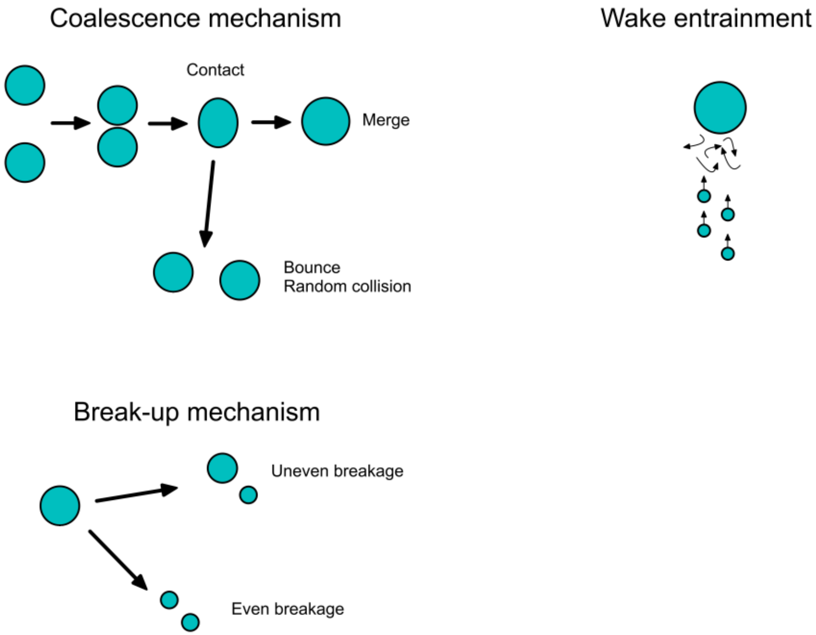
Eulerian-Eulerian approach (Multi-fluid and mixture models)

Eulerian-Lagragian approach (Particle tracking)

How to treat the wide range of behaviors in multiphase flows



- Dispersed phase in a continuous phase.
- In this case, the VOF method is not able to handle bubbles smaller than grid scales.
- Multi-fluid and mixture models are able to model bubbles smaller than grid scales by averaging the phase properties in the discrete domain.

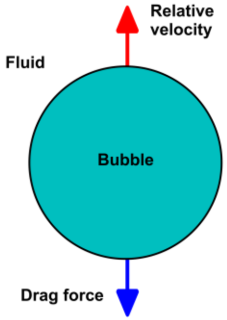


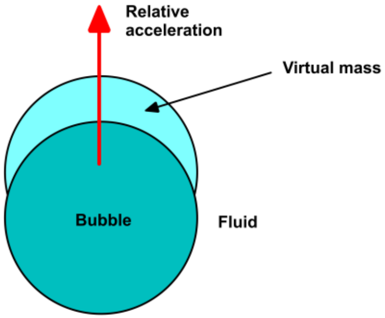
- Multi-fluid and mixture approaches can model bubble coalescence, bubble break-up and wake entrainment in dispersed systems.

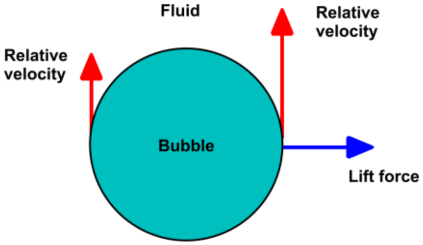
In theory, the VOF method can resolve the smallest bubbles/droplets but the mesh requirements are too prohibitive (equivalent to DNS). In multiphase flows, this is called fully resolved approach.

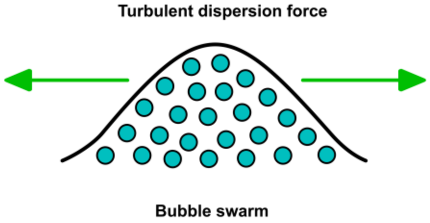
How to treat the wide range of behaviors in multiphase flows

- When using multi-fluid and mixture approaches, interfacial momentum transfer models must be taken into account in order to model mass transfer and phases interaction.
- As for turbulence modeling, there is no universal model.
- It is up to you to choose the model that best fit the problem you are solving.
- Depending on the physics involved, you will find different models and formulations
- You need to know the applicability and limitations of each model, for this, refer to the literature.

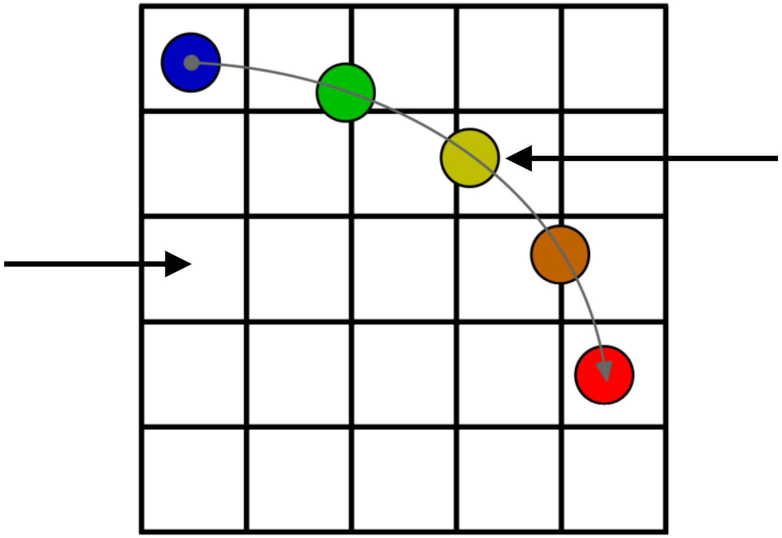








How to treat the wide range of behaviors in multiphase flows

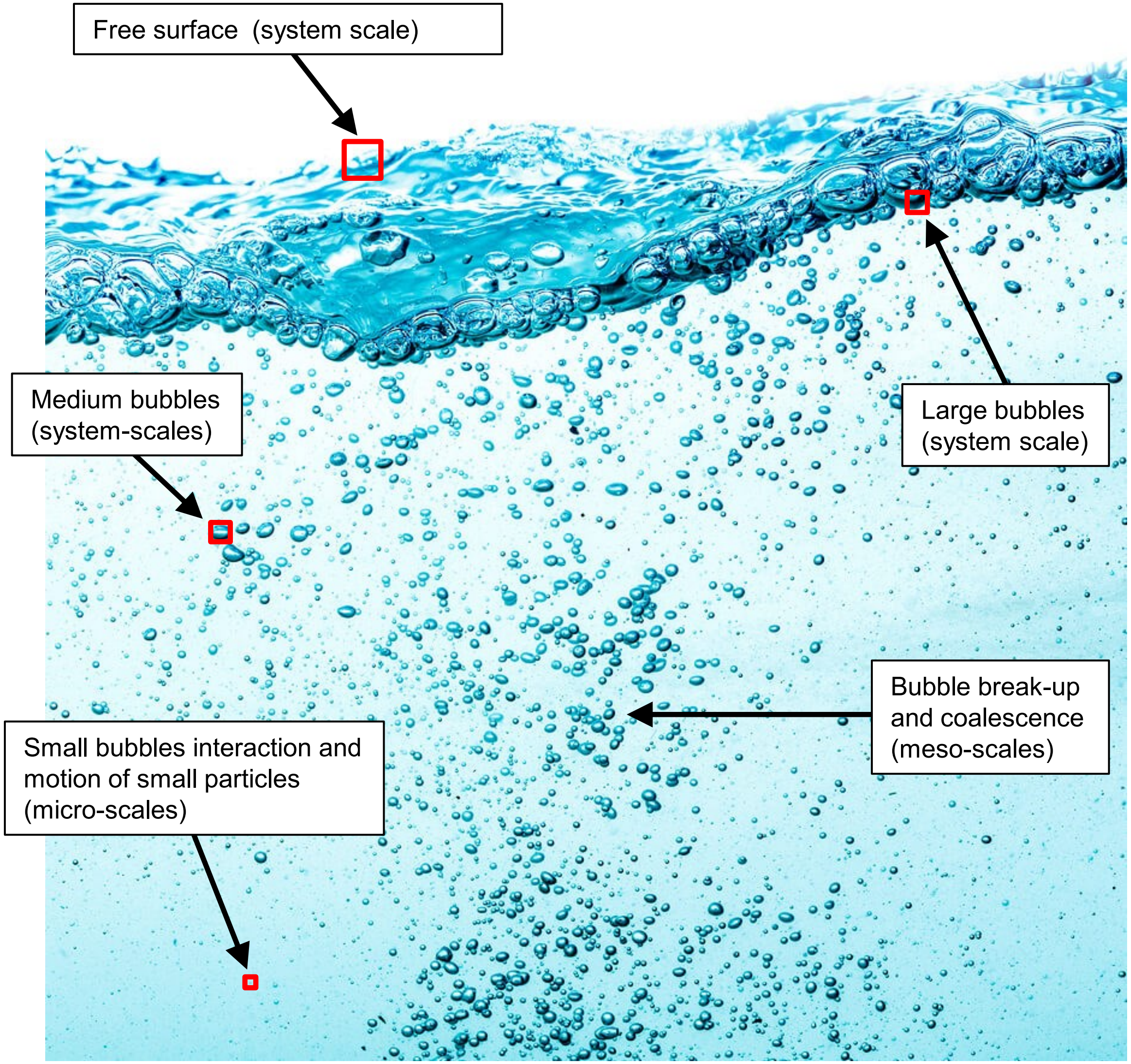


The continuous phase is solved in the mesh

The particles position is tracked by solving an ODE for each particle

- In the Eulerian-Lagrangian framework, the continuous phase is solved in an Eulerian reference system and the particles or dispersed phase is solved in a Lagrangian reference system.
- The particles can be smaller or larger than the grid size.
- The particles can be transported passively, or they can be coupled with the fluid governing equations.
- It accounts for particle interaction and mass transfer.
- The particles can interact with the boundaries and have a fate.

Multiphase flows – Grid scales


Free surface (system scale)
Medium bubbles (system-scales)
Large bubbles (system scale)
Bubble break-up and coalescence (meso-scales)
Small bubbles interaction and motion of small particles (micro-scales)

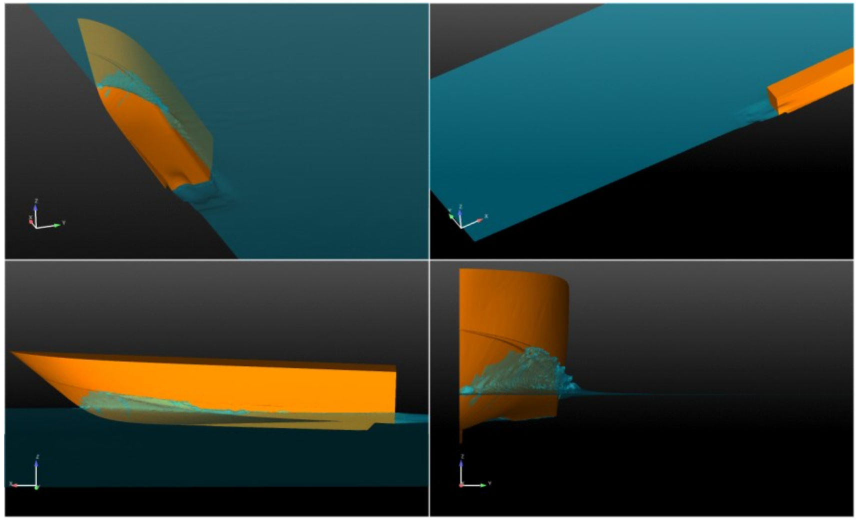
- Applicability of the VOF method to separated systems (non-interpenetrating continua).
- In the figure, the free surface and large bubbles can be track/resolve by the mesh.
- The smaller the features we want to track/resolve, the smaller the cells should be.
- Bubbles, droplets and/or particles larger than grid scales (GS), can be resolved using VOF.
- To resolve a bubble you will need at least two cells in every direction
- Bubbles, droplets and/or particles smaller than grid scales (sub-grid scales or SGS), can not be resolve using the VOF method.
- In such a case, we need to use models.
- Also, bubble break-up, coalescence and entrainment must be modeled, unless the mesh is fine enough so it captures the dynamics and solves the smallest scales.

Multiphase flows are transient and multiscale

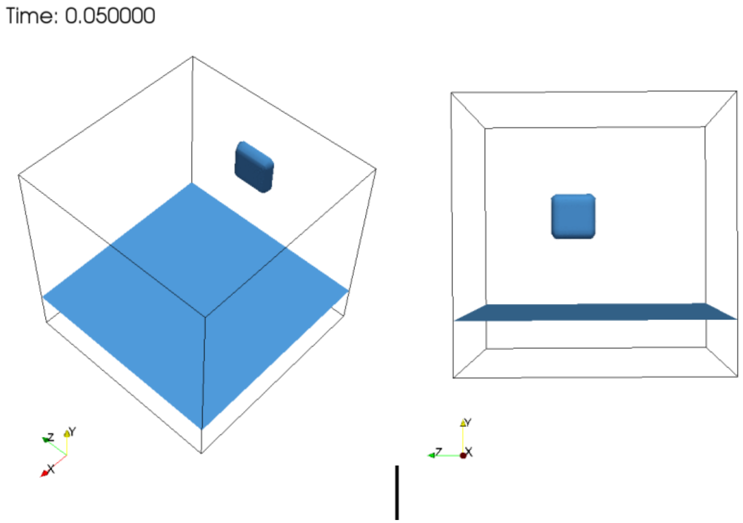
Numerical approaches for multiphase flows

Eulerian-Eulerian (VOF)	Eulerian-Eulerian (Dispersed systems)	Eulerian-Lagrangian
• Non-interpenetrating continua. • Continuous phases: Eulerian. • Fluid properties are written on either side of the interface (no averaging). • Solves one single set of PDEs: mass, momentum, energy.	• Interpenetrating continua. • Continuous phase: Eulerian. • Dispersed phase: Eulerian. • Phase-weighted averages. • Solves PDEs for all phases (including interphase transfer terms): mass, momentum, energy. • It can deal with gas-liquid, gas-solid, and liquid-solid interactions.	• Continuous phase: Eulerian. • Dispersed phase: Lagrangian. • Solves ODEs for particle tracking (for every single particle). • Solves a set of PDEs for the continuous phase: mass, momentum, energy. • Phase interaction terms (including interphase transfer terms).

Numerical approaches for multiphase flows



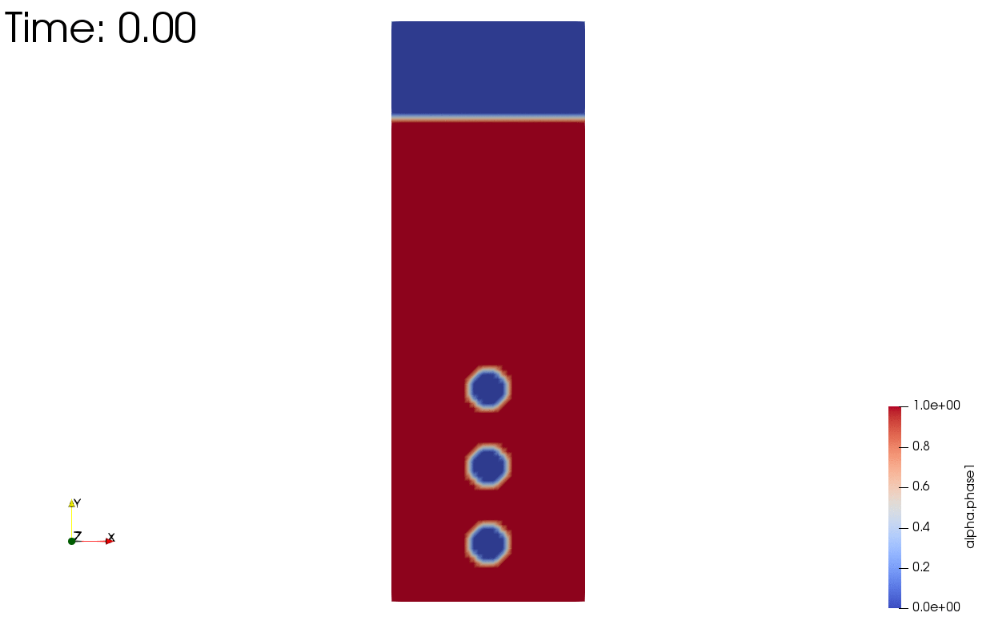
www.wolfdynamics.com/training/mphase/image10.gif

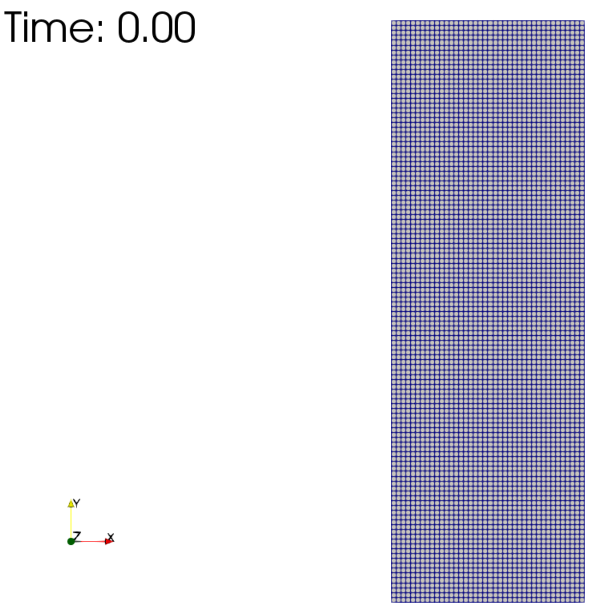


http://www.wolfdynamics.com/training/mphase/image16.gif

- Simulations showing free surface tracking using the VOF approach
- The left image corresponds to a simulation with rigid body motion and accurate surface tracking using the VOF method.

Numerical approaches for multiphase flows



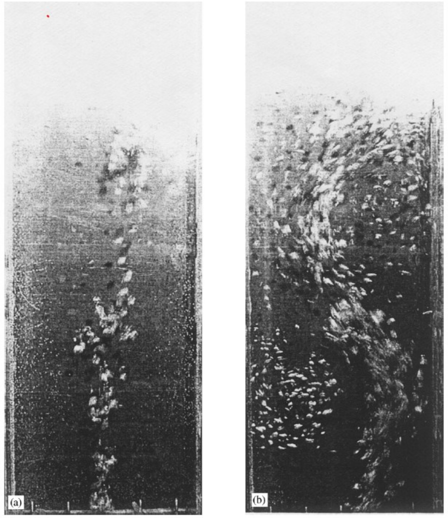


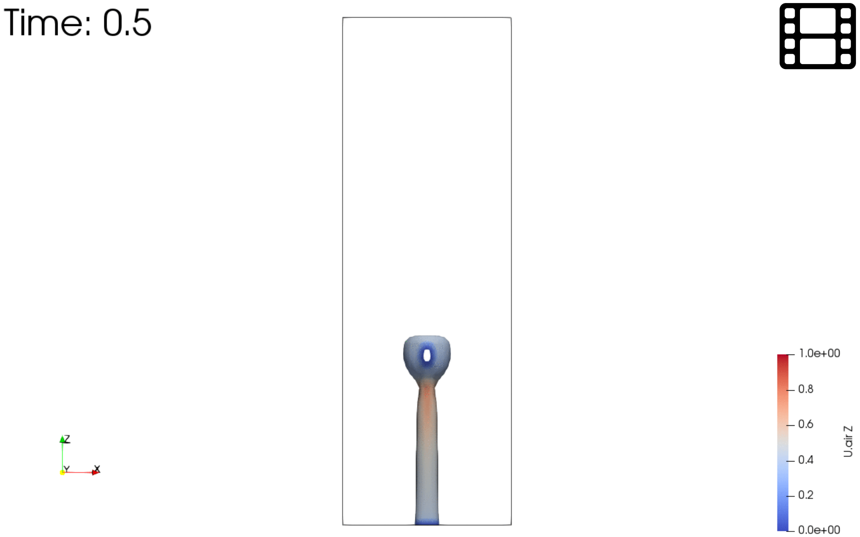
http://www.wolfdynamics.com/training/mphase/image2.gif

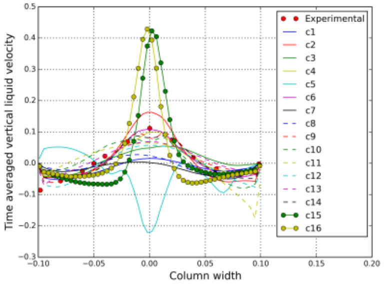
http://www.wolfdynamics.com/training/mphase/image3.gif

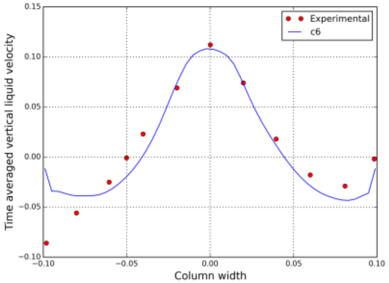
- Simulation showing free surface tracking, bubble tracking, bubble coalescence, bubble break-up and wake entrainment using the VOF method.
- In this simulation the free surface and bubbles are capture by using AMR.
- However, the smallest bubble that can be resolved is at the smallest grid size.

Numerical approaches for multiphase flows









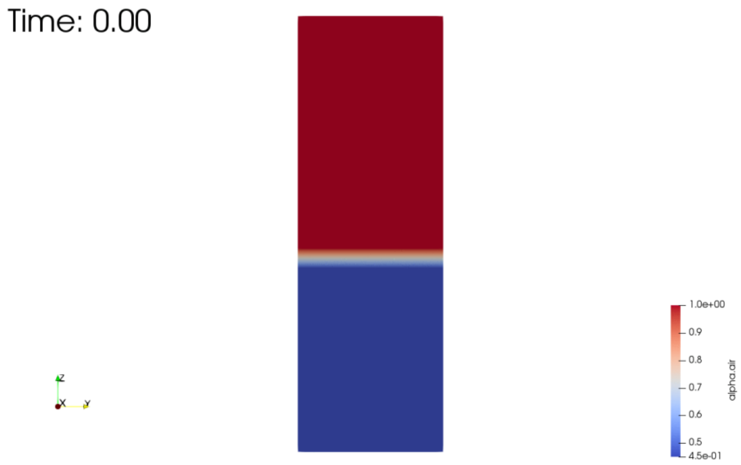
http://www.wolfdynamics.com/training/mphase/image18.gif

- Eulerian-Eulerian simulation (gas-liquid).
- The bubbles are not being solved, instead, the interaction between phase is being averaged.

References:

[1] Vivek V. Buwa, Vivek V. Ranade, Dynamics of gas–liquid flow in a rectangular bubble column: experiments and single/multi-group CFD simulations. Chemical Engineering Science 57 (2002) 4715 – 4736

Numerical approaches for multiphase flows



twoPhaseEulerFoam
Air volume fraction
Turbulent case

http://www.wolfdynamics.com/training/mphase/image42.gif

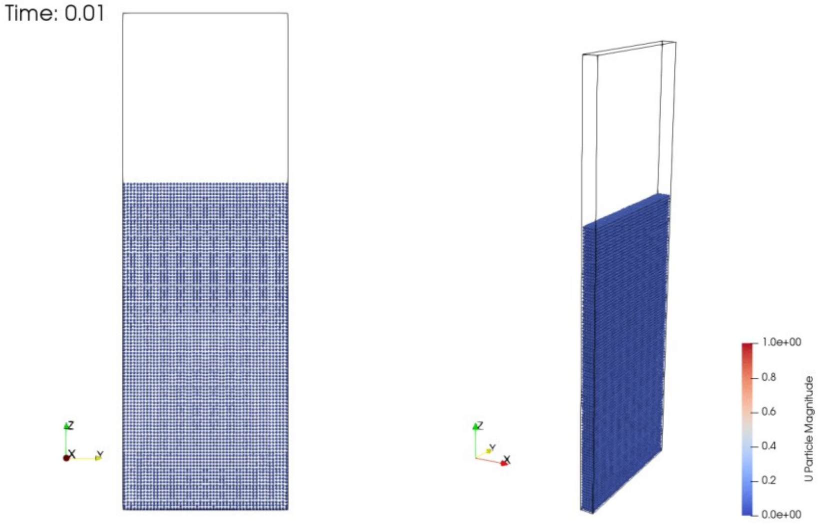
Time: 0.00
Z
X
Y
1.0e+00
0.9
0.8
0.7
0.6
0.5
4.5e-01
alpha.air

twoPhaseEulerFoam
Air volume fraction
Laminar case

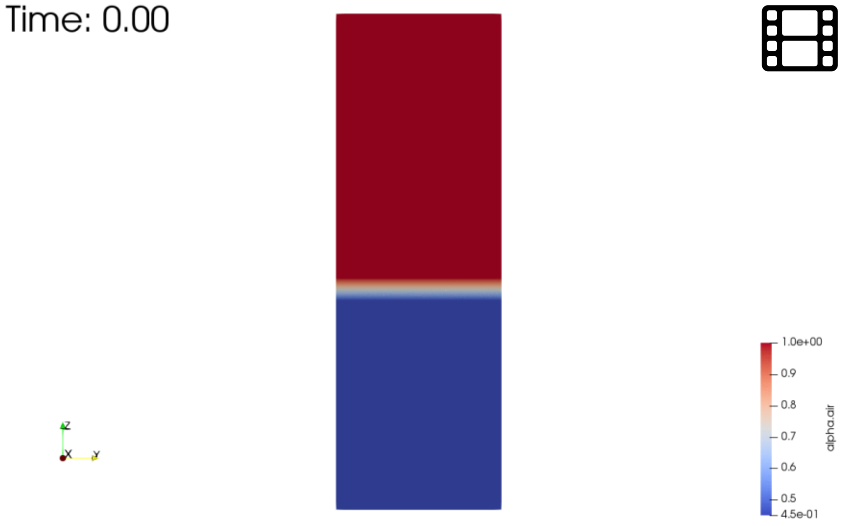
http://www.wolfdynamics.com/training/mphase/image41.gif

- Eulerian-Eulerian simulations using the Eulerian-Granular KTGF approach (solid-gas).
- The granular phase is simulated as continuous phase.
- In these simulations we can observe the influence of turbulence modeling in the solution.

Numerical approaches for multiphase flows



DPMFoam
Particle-particle interactions colored by velocity magnitude (particles not to scale)
http://www.wolfdynamics.com/training/mphase/image43.gif



twoPhaseEulerFoam
Air volume fraction
Turbulent case
http://www.wolfdynamics.com/training/mphase/image42.gif

- Comparison of an Eulerian-Lagrangian simulation and an Eulerian-Eulerian simulation (gas-solid).
- In the Eulerian-Lagrangian approach we track the position of every single particle. We also solve the fate and interaction of all particles.
- In the Eulerian-Eulerian approach we solve the granular phase as a continuous phase.
- The computational requirements of the Eulerian-Eulerian simulation are much lower than those for the Eulerian-Lagrangian simulation.

Volume-of-Fluid (VOF) governing equations for separated systems

- The incompressible, isothermal governing equations can be written as follows,

Source terms:

- Porous media
- Coriolis forces
- Centrifugal forces
- Mass transfer
- and so on …

$$\nabla \cdot \mathbf{U} = 0$$

Surface tension - Continuum surface force (CSF)

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U}\mathbf{U}) = -\nabla p + \nabla \cdot \tau + \rho g + f_\sigma + \rho S$$

$$\frac{\partial \gamma}{\partial t} + \nabla \cdot \mathbf{U}\gamma + \nabla \cdot (\mathbf{U}_r \gamma (1-\gamma)) = 0 \longrightarrow$$

Phase transport equation and interface tracking with surface compression

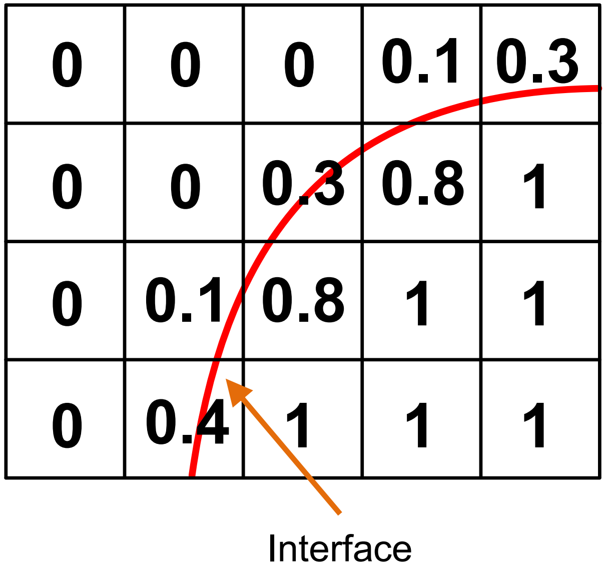
$$0 < \gamma < 1 \longrightarrow$$

Volume fraction (bounded quantity)

- You can see the volume fraction γ as a pointer that indicates what phase (with the corresponding physical properties), is inside each cell of the computational domain.

Volume-of-Fluid (VOF) governing equations for separated systems

- For example, in the case of two phases where phase 1 is represented by $\gamma = 1$ and phase 2 is represented by $\gamma = 0$; a volume fraction value of 1 indicates that the cell is fill with phase 1; a volume fraction of 0.8 indicates that the cell contains 80% of a phase 1; and a volume fraction of 0, indicates that the cell is fill with phase 2.
- The values between 0 and 1 can be seen as the interface between the phases.



0	0	0	0.1	0.3
0	0	0.3	0.8	1
0	0.1	0.8	1	1
0	0.4	1	1	1

- The fluid properties can be written on either side of the interface as follows,

$$\rho = \gamma_1 \rho_1 + (1 - \gamma_1) \rho_2$$

$$\mu = \gamma_1 \mu_1 + (1 - \gamma_1) \mu_2$$

Eulerian-Eulerian governing equations for dispersed systems

- The Eulerian-Eulerian approach solves the governing equations for each phase, it treats the phases as interpenetrating continua.
- The incompressible, isothermal governing equations with interface tracking can be written as follows,

Surface tension - Continuum surface force (CSF)

Interface forces or momentum transfer.
Bubbles interaction models

$$\frac{\partial \alpha_k \rho_k}{\partial t} + \nabla \cdot (\alpha_k \rho_k \mathbf{U}_k) = 0$$

$$\frac{\partial (\alpha_k \rho_k \mathbf{U}_k)}{\partial t} + \nabla \cdot (\alpha_k \rho_k \mathbf{U}_k \mathbf{U}_k) = -\nabla \cdot (\alpha_k \tau_k) - \alpha_k \nabla p + \alpha_k \rho_k \mathbf{g} + \mathbf{M}_k + f_\sigma + \mathbf{S}_k$$

$$\frac{\partial \alpha_k \rho_k}{\partial t} + \nabla \cdot \mathbf{U}_k \alpha_k \rho_k + \nabla \cdot (\mathbf{U}_r \alpha_k \rho_k (1 - \alpha_k)) = 0$$

$$\sum_k \alpha_k = 1.0 \qquad \rho_m = \sum_k \alpha_k \rho_k \qquad \mathbf{U}_m = \frac{\sum_k \alpha_k \rho_k \mathbf{U}_k}{\rho_m}$$

Source terms:

- Porous media
- Coriolis forces
- Centrifugal forces
- Mass transfer
- and so on …

Eulerian-Lagrangian governing equations

- In the Eulerian-Lagrangian framework, the continuous phase is solved in an Eulerian reference system and the particles or dispersed phase is solved in a Lagrangian reference system.
- The particles can be transported passively, or they can be coupled with the fluid governing equations (they can modify the fluid field).
- The particles motion is calculated by solving an ODE for every single particle (Newton-Euler equation of motion).
- The particles can interact with the boundaries, they can escape, bounce, stick, or form a wall film.
- This formulation accounts for particle interaction and mass transfer.
- The governing equations can be written as follows,

$$m\frac{d\mathbf{U}}{dt} = \mathbf{F}_{drag} + \mathbf{F}_{pressure} + \mathbf{F}_{virtual\,mass} + \mathbf{F}_{other}$$

$$+$$

Any of the Eulerian formulations (single or multi-phase)

Multiphase solvers in OpenFOAM®

- In OpenFOAM®, there are many interfacial momentum transfer models implemented.
- There are also many models for Eulerian-Lagrangian methods.
- No need to say that turbulence also applies to multiphase flows.
- There is no universal model, it is up to you to choose the model that best fit the problem you are solving.
- You need to know the applicability and limitations of each model, for this, refer to the literature.
- When dealing with multiphase flows in OpenFOAM®, you can use VOF, Eulerian-Eulerian, Eulerian-Eulerian with VOF, and Eulerian-Lagrangian methods.
- The solution methods can account for turbulence models, interface momentum transfer models, mass transfer models, particle interaction models and so on.
- It is also possible to add source terms, deal with moving bodies or use adaptive mesh refinement.
- You will find the source code of all the multiphase solvers in the directory:
    - **`OpenFOAM-8/applications/solvers/multiphase`**
- You will find the source code all the particle tracking solvers in the directory:
    - **`OpenFOAM-8/applications/solvers/lagrangian`**

Multiphase solvers in OpenFOAM®

- These are the multiphase solvers that you will use most of the time in OpenFOAM®.

- The VOF approach:
  - interFoam family solvers

- The Eulerian-Eulerian approach:
  - twoPhaseEulerFoam, multiphaseEulerFoam

- The Eulerian-Granular KTGF (kinetic theory of granular flows) approach.
  - twoPhaseEulerFoam

- The Eulerian-Lagrangian framework,
  - DPMFoam, MPPICFoam

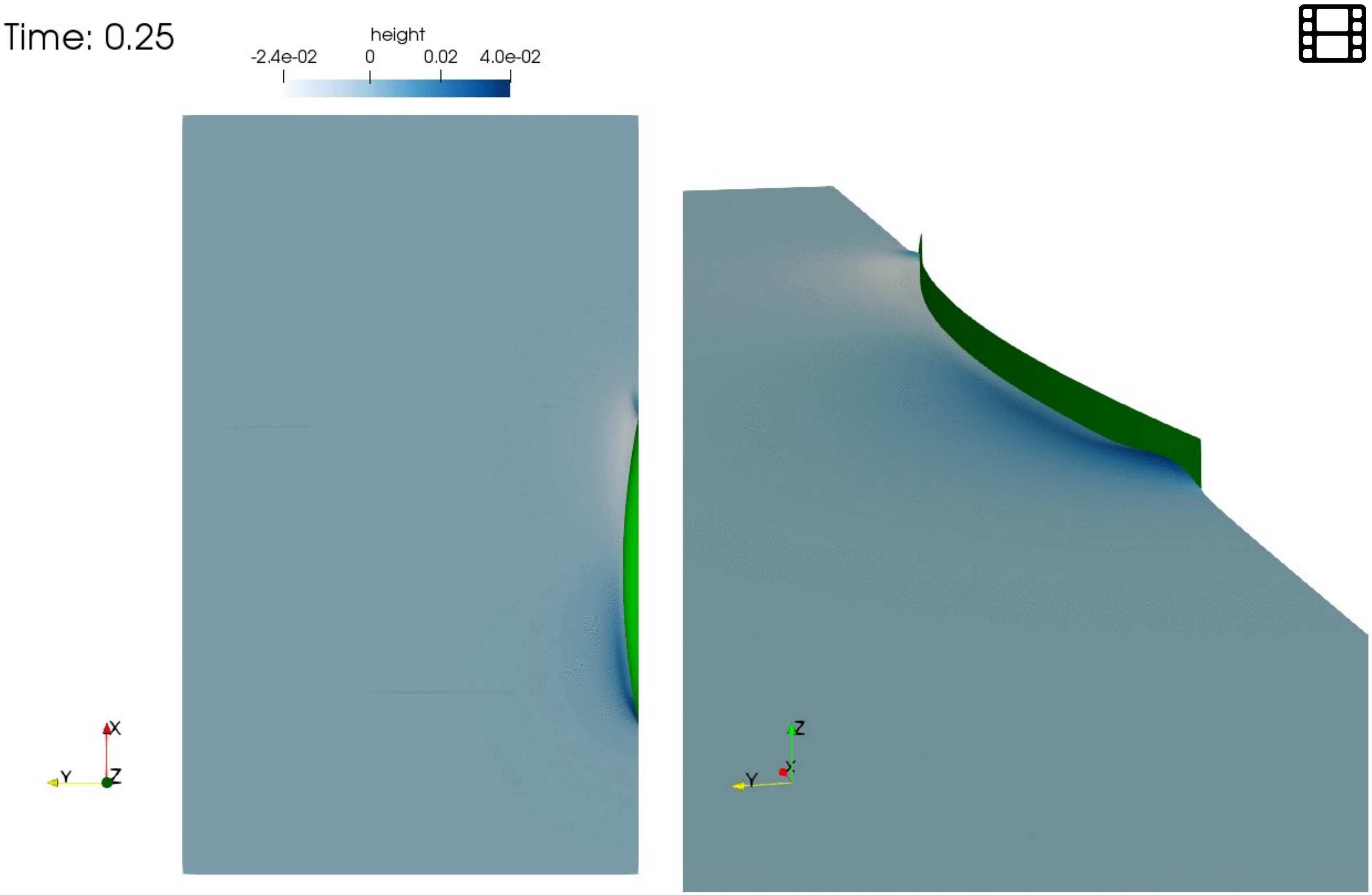
Multiphase flows hands-on tutorials

- Free surface – Ship resistance simulation
- Let us run this case. Go to the directory:

```
$PTOFC/advanced_physics/multiphase/wigleyHull
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Free surface – Ship resistance simulation

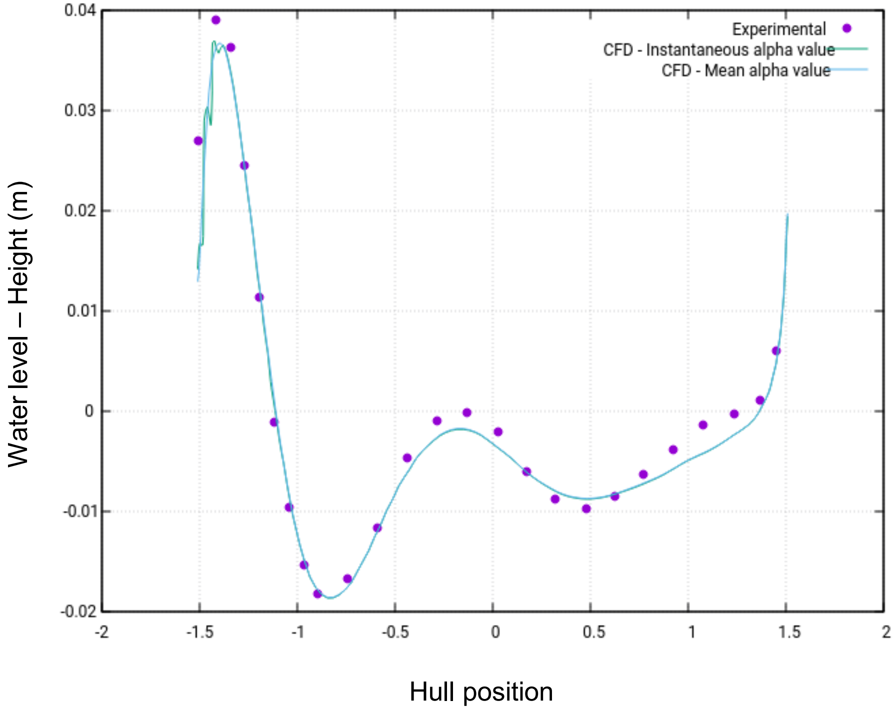


Free surface colored by height

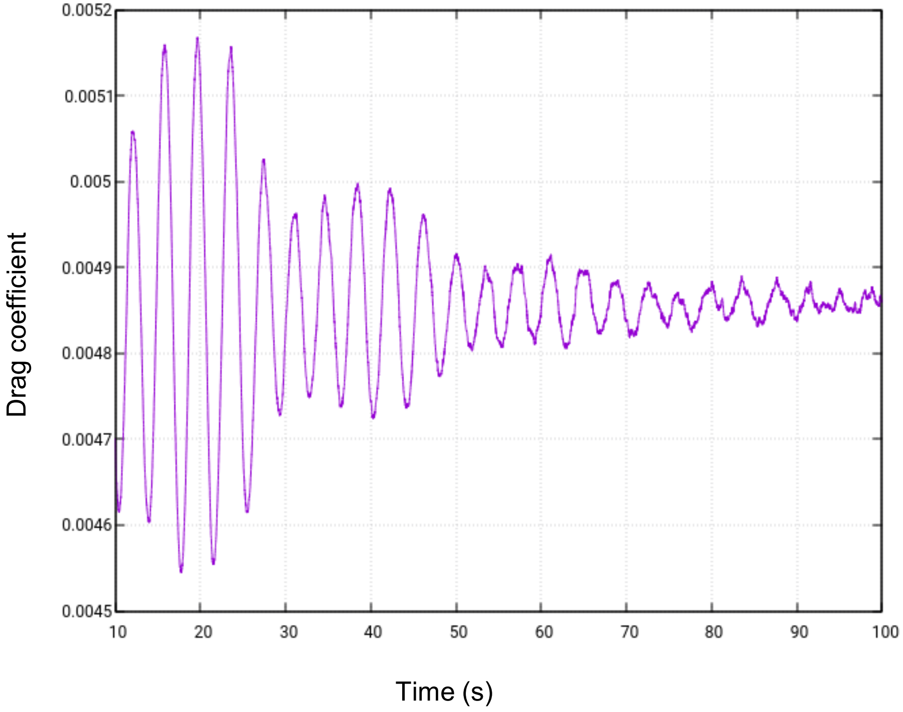
http://www.wolfdynamics.com/training/mphase/image45.gif

Multiphase flows hands-on tutorials

Free surface – Ship resistance simulation



Comparison of water level on hull surface



Drag coefficient monitor

Free surface – Ship resistance simulation

- We are going to use the following solver: **interFoam**
- The first step is to set the physical properties. In the dictionary *constant/transportProperties* we defined the phases.
- Go to the directory **constant** and open the dictionary *transportProperties*.

The first phase is always considered the primary phase

Phases naming convention. The name of the phases is chosen by the user.

water properties

air properties

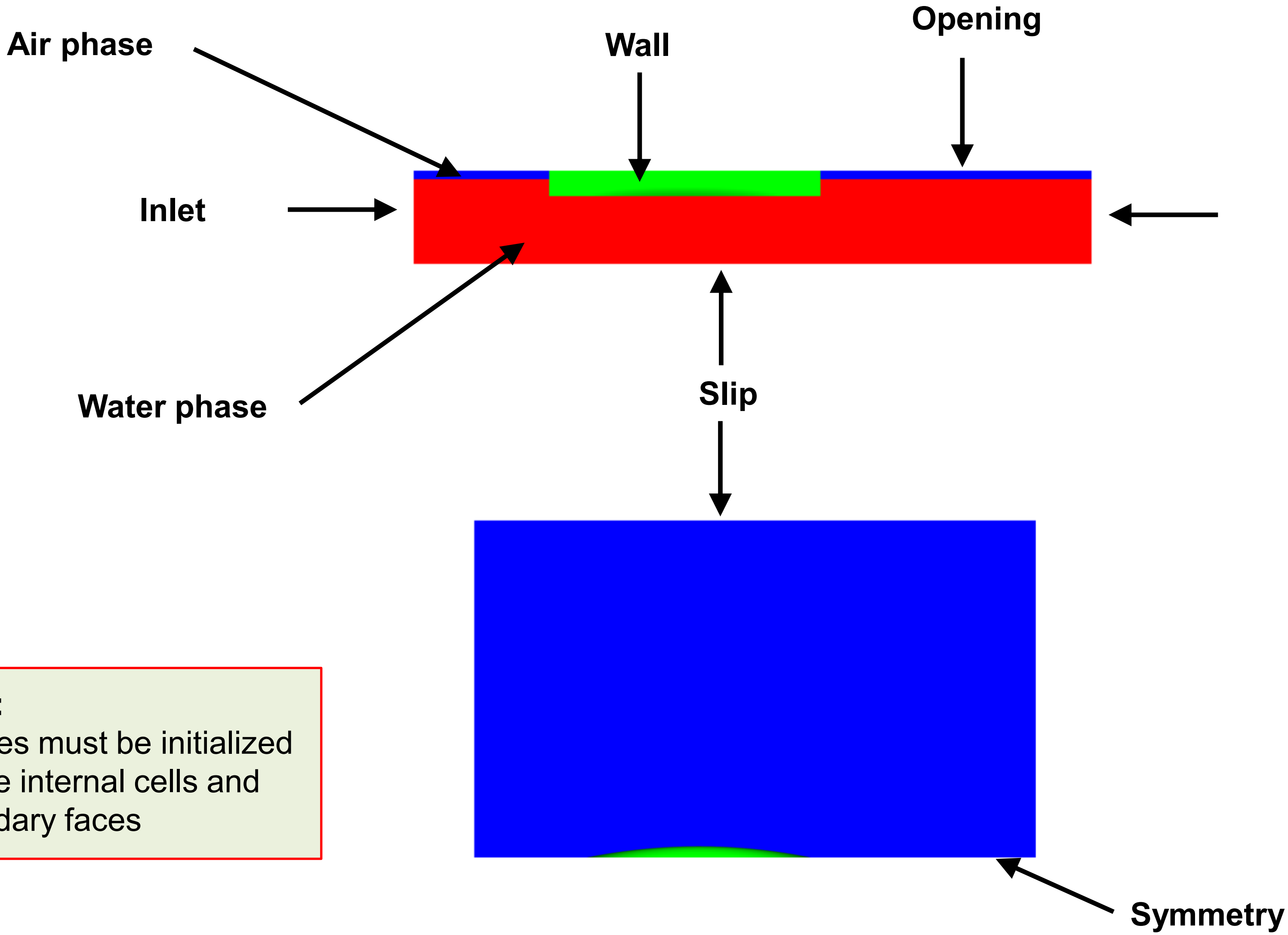
Surface tension between phase1 and phase2

```
phases (water air);

water
{
        transportModel      Newtonian;
        nu                  nu [ 0 2 -1 0 0 0 0 ] 1.09e-06;
        rho                 rho [ 1 -3 0 0 0 0 0 ] 998.8;
}

air
{
        transportModel      Newtonian;
        nu                  nu [ 0 2 -1 0 0 0 0 ] 1.48e-05;
        rho                 rho [ 1 -3 0 0 0 0 0 ] 1;
}

sigma sigma [ 1 0 -2 0 0 0 0 ] 0.07;
```

Free surface – Ship resistance simulation

- The next step is to set the boundary conditions and initial conditions.
- Therefore, in the directory `0` we define the dictionary *`alpha.water`* that will take the values of the phase water.
- In this case, you will find the directory **`0_org`**, here is where we keep a backup of the original files as we are doing field initialization using `setFields`.
- In the directory `0`, you will find the dictionary *`p_rgh`*, in this dictionary we set the boundary and initial conditions for the pressure field, and the dimensions are in Pascals.
- The turbulence variables values were calculated using an eddy viscosity ratio equal to 1, turbulence intensity equal 5%, and the water properties.
- If you are simulating numerical towing tanks, the setup of the boundary conditions is always the same.
- Feel free to reuse this setup.
- The dictionaries used in this case are standard for the VOF solvers (`interFoam` family solvers).
- If you are using a different solver (e.g., `twoPhaseEulerFoam`), you will need to use additional dictionaries where you define the interfacial models and so on.
- Remember, you should always conduct production runs using a second order discretization scheme

Free surface – Ship resistance simulation



Note:
Phases must be initialized on the internal cells and boundary faces

Physical domain and boundary patches

Multiphase flows hands-on tutorials

Free surface – Ship resistance simulation

Patch name	Pressure	Velocity	Turbulence fields	alpha.water
inflow	fixedFluxPressure	fixedValue	fixedValue calculated (nut)	fixedValue
outflow	inletOutlet or zeroGradient	outletPhaseMeanVelocity	inletOutlet calculated (nut)	variableHeightFlowRate
bottom	symmetry	symmetry	symmetry	symmetry
midplane	symmetry	symmetry	symmetry	symmetry
side	symmetry	symmetry	symmetry	symmetry
top	totalPressure	pressureInletOutletVelocity	inletOutlet calculated (nut)	inletOutlet
ship	fixedFluxPressure	fixedValue	kqRWallFunction (k) omegaFunction (omega) nutkWallFunction (nut)	zeroGradient

Typical setup of boundary conditions for numerical towing tank simulations

Free surface – Ship resistance simulation

- OpenFOAM® solves the following modified volume fraction convective equation to track the interface between the phases,

$$\frac{\partial \alpha}{\partial t} + \underbrace{\nabla \cdot \mathbf{U}\alpha}_{\text{(phi, alpha)}} + \underbrace{\nabla \cdot \overset{c_\alpha|\mathbf{U}|}{\mathbf{U}_c}\alpha(1-\alpha)}_{\text{(phirb, alpha)}} = 0$$

(phi, alpha)
Use a TVD scheme with gradient limiters.
Good choice is the vanLeer scheme.

(phirb, alpha)
Use a high order scheme. The use of linear interpolation is fine for this term.

- Where a value of $c_\alpha = 1$ (cAlpha), is recommended to accurately resolve the sharp interface.
- To solve this equation, OpenFOAM® uses the semi-implicit MULES method.
- The MULES options can be controlled in the *fvSolution* dictionary.

Multiphase flows hands-on tutorials

Free surface – Ship resistance simulation

- MULES options in the *fvSolution* dictionary.
- The semi-implicit MULES offers significant speed-up and stability over the explicit MULES.

```
“alpha.*”
{
        MULESCorr               yes;
        nAlphaSubCycles         1;

        nAlphaCorr              3;



        nLimiterIter            10;



        alphaApplyPrevCorr      yes;
        …
}
```

MULESCorr: Turn on/off semi-implicit MULES

nAlphaSubCycles: For semi-implicit MULES use 1. Use 2 or more for explicit MULES.

nAlphaCorr: Number of corrections.
Use 2-3 for slowly varying flows.
Use 3 or more for highly transient, high Reynolds, high CFL number flows.

nLimiterIter: Number of iterations to calculate the MULES limiter. Use 3-5 if CFL number is less than 3. Use 5-10 if CFL number is more than 3.

alphaApplyPrevCorr: Use previous time corrector as initial estimate. Set to yes for slowly varying flows. Set to no for highly transient flows.

Free surface – Ship resistance simulation

- Additional notes on the *fvSolution* dictionary.

```
momentumPredictor              yes;

nOuterCorrectors               1;



nCorrector                     3;

nNonOrthogonalCorrectors       2;
```

Set to yes for high Reynolds flows, where convection dominates

Recommended value is 1 (equivalent to PISO).
Increase to improve the stability of second order time discretization schemes (LES simulations).
Increase for highly coupled problems.

Recommended to use at least 2 correctors.
It improves accuracy and stability.

Recommend to use at least 1 corrector.
Increase the value for bad quality meshes.

- If you are planning to use large time-steps (CFL number larger than 1), it is recommended to do at least 3 nCorrector, otherwise you can use 2.

Multiphase flows hands-on tutorials

Free surface – Ship resistance simulation

- Finally, we need to set the discretization schemes
- This is done in the dictionary *fvSchemes*.
- In this dictionary we set the discretization method for every term appearing in the governing equations.
- Convective terms discretization is set as follows:

```
divSchemes
{
    div(rhoPhi,U)                    Gauss linearUpwind grad(U);

    div(phi,alpha)                   Gauss interfaceCompression vanLeer 1;

    div(((rho*nuEff)*dev2(T(grad(U)))))        Gauss linear;
}
```

This term is related to the volume fraction equation

- Notice that we are using a high-resolution scheme for the surface tracking (div(phi,alpha)).

Free surface – Ship resistance simulation

- For time discretization we can use an unsteady formulation (Euler in this case).
- This scheme requires setting the time-step, and it should be choosing in such a way that it resolves the mean physics.
- Remember, as the free surface is a strong discontinuity, for stability and good resolution we need to use a CFL less than one for the interface courant.

```
ddtSchemes
{
        default Euler;
}
```

- Hereafter, we are using what is know as global time stepping, that is, the CFL number is limited by the smallest cell.
- The simulation is time-accurate, but it requires a lot of CPU time to reach a steady state (if it reaches one).

Multiphase flows hands-on tutorials

Free surface – Ship resistance simulation

- A way to accelerate the convergence to steady state, is by using local time stepping (LTS).
- In LTS, the time-step is manipulated for each individual cell in the mesh, making it as high as possible to enable the simulation to reach steady-state quickly.
- When we use LTS, the transient solution is no longer time accurate.
- The stability and accuracy of the method are driven by the local CFL number of each cell.
- To avoid instabilities caused by sudden changes in the time-step of each cell, the local time-step can be smoothed and damped across the domain.
- Try to avoid having local time-steps that differ by several order of magnitudes.
- To enable LTS, we use the localEuler method.

```
ddtSchemes
{
        default localEuler;
}
```

- LTS in OpenFOAM® can be used with any solver that supports the **PISO** or **PIMPLE** loop (**PISO ITA**).

Free surface – Ship resistance simulation

- In the LTS method, the maximum flow CFL number, maximum interface CFL number, and the smoothing and damping of the solution across the cells, can be controlled in the dictionary *fvSolution*, in the sub-dictionary **PIMPLE**.

```
PIMPLE
{
          momentumPredictor                     yes;

          nOuterCorrectors                      2;
          nCorrector                            3;
          nNonOrthogonalCorrectors              2;

          maxCo                                 10;
          maxAlphaCo                            1;

          rDeltaTSmoothingCoeff                 0.05;
          rDeltaTDampingCoeff                   0.5;

          maxDeltaT                             1;
}
```

Maximum flow Courant

Maximum interface Courant

Local time step smoothing

Local time step damping

Limit the maximum time-step size

Multiphase flows hands-on tutorials

Free surface – Ship resistance simulation

- At this point, we are ready to run the simulation.
- Remember to adjust the numerics according to your physics.
- You can choose between running using global time stepping or unsteady (directory **`uns`**) or local time stepping (directory **`LTS`**).
- You will find the instructions of how to run the cases in the file *`README.FIRST`* located in the case directory.

Roadmap

A crash introduction to:

1. ~~Turbulence modeling in OpenFOAM®~~
2. ~~Multiphase flows modeling in OpenFOAM®~~
3. **Compressible flows in OpenFOAM®**
4. Moving bodies in OpenFOAM®
5. Source terms in OpenFOAM®
6. Scalar transport pluggable solver

What are compressible flows?

- In few words, compressible flows are flows where the density change.
- The changes in density can be due to velocity, pressure, or temperature variations.
- Compressible flows can happen at low speed (subsonic) or high speed (transonic, supersonic, hypersonic and so on).
- Buoyancy-driven flows are also considered compressible flows. After all, the buoyancy is due to temperature gradients.
- In compressible flows, the viscosity also changes with temperature.
- The thermodynamical variables are related via an equation of state (e.g., ideal gas law).
- In principle, all flows are compressible.
- Usually, compressibility effects start to become significant when the Mach number is higher than 0.3.

A few compressible flows applications

- The following applications fall within the compressible flows classification:
    - External and internal aerodynamics (high speed).
    - Heat transfer and conjugate heat transfer.
    - Fire dynamics.
    - Buoyancy driven flows
    - Heating, ventilation, and air conditioning (HVAC).
    - Thermal comfort.
    - Turbomachinery.
    - Combustion.
    - Chemical reactions.
    - Condensation, evaporation, and melting.
    - Cavitation.
    - And many more.

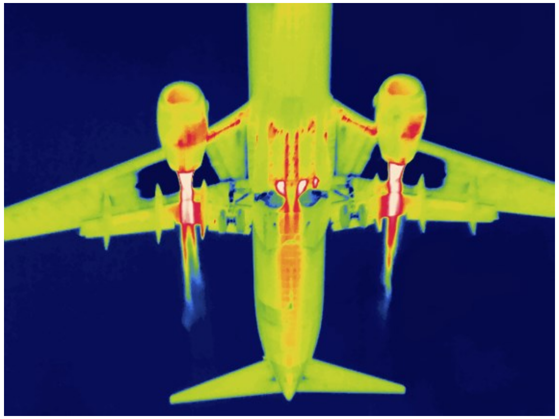
- As you can see, the range of applicability is very wide.

A crash introduction to compressible flows modeling OpenFOAM®

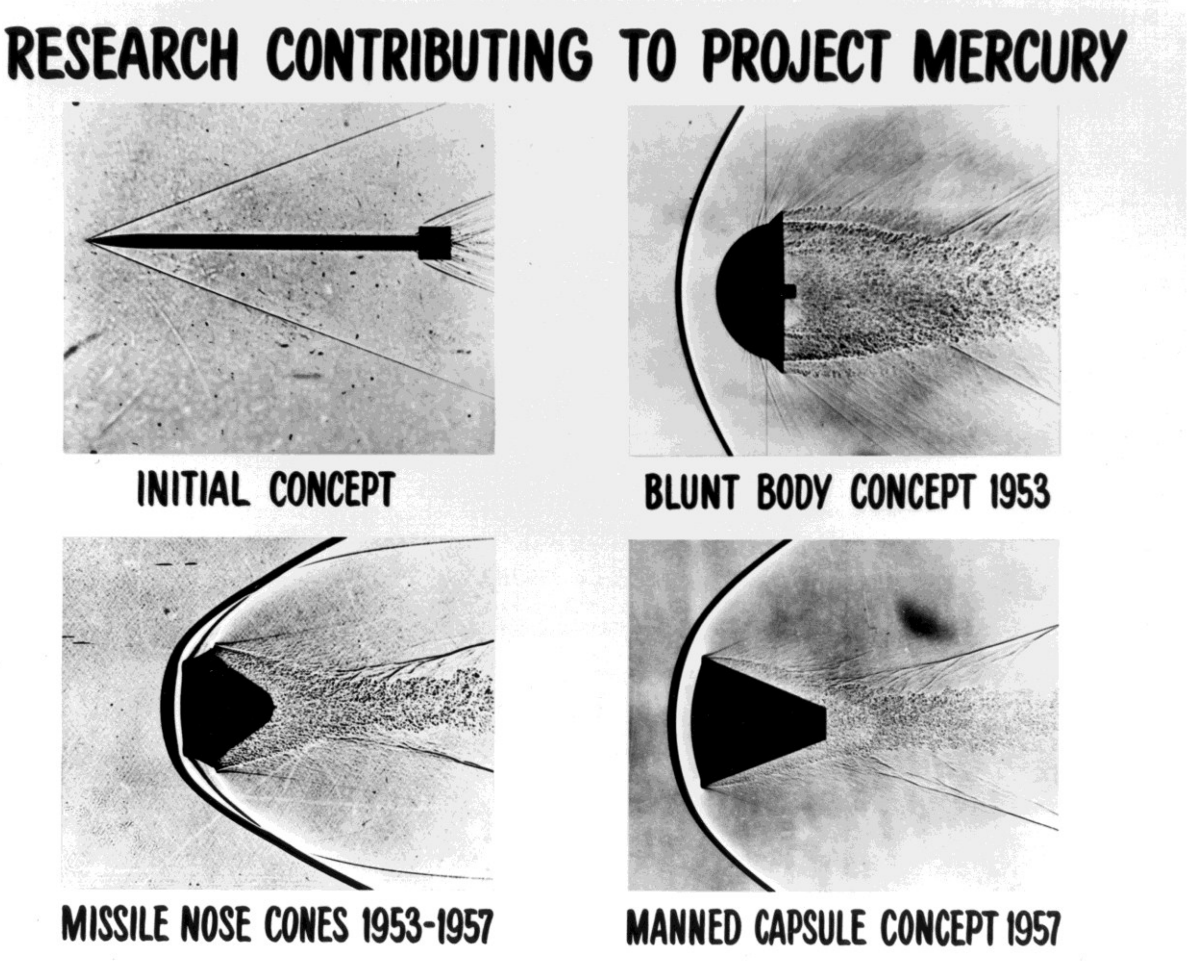
Large Natural Convection Plume, as effect of combustion of excess non-useable gases behind oilfield.
https://en.wikipedia.org/wiki/Plume_(fluid_dynamics)#/media/File:Naturalconvectionplume.JPG

Rayleigh–Bénard convection cells
https://en.wikipedia.org/wiki/File:B%C3%A9nard_cells_convection.ogv

Iron melting
https://commons.wikimedia.org/wiki/File:Iron_-melting.JPG



Airplane thermal image
http://www.blackroc.com/wp-content/uploads/2016/03/thermal-image.jpg


RESEARCH CONTRIBUTING TO PROJECT MERCURY
INITIAL CONCEPT
BLUNT BODY CONCEPT 1953
MISSILE NOSE CONES 1953-1957
MANNED CAPSULE CONCEPT 1957

Shadowgraph Images of Re-entry Vehicles
Photo credit: NASA on the Commons.
https://www.flickr.com/photos/nasacommons/

Compressible flows – Starting equations

NSE

$$\left\{\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0, \\ \frac{\partial \left(\rho \mathbf{u}\right)}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) &= -\nabla p + \nabla \cdot \tau, \\ \frac{\partial \left(\rho e_t\right)}{\partial t} + \nabla \cdot (\rho e_t \mathbf{u}) &= \nabla \cdot q - \nabla \cdot (p\mathbf{u}) + \tau \mathbf{:} \nabla \mathbf{u}, \end{aligned}\right.$$

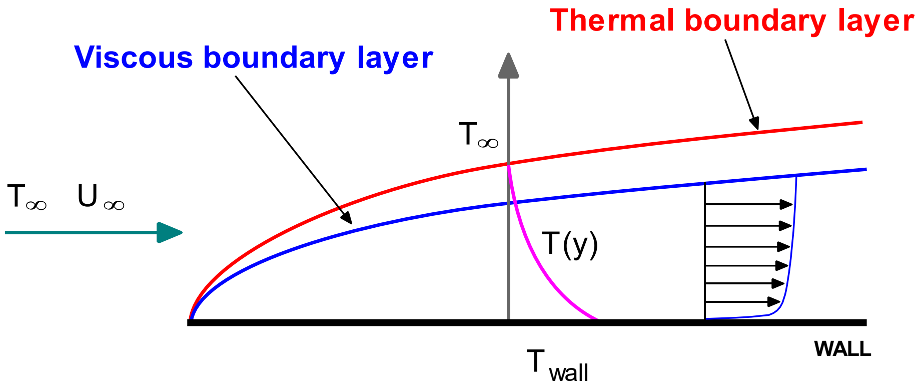
$$+$$

Equation of state and thermodynamics relations (thermophysical models)

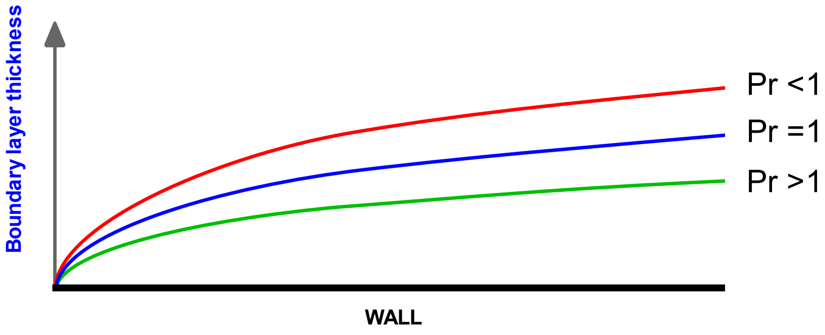
$$+$$

Additional closure equations for turbulence models, multiphase models, combustion, particles, source terms, and so on

Compressible flows – Boundary layer



Thermal boundary layer vs. Viscous boundary layer
Forced convection

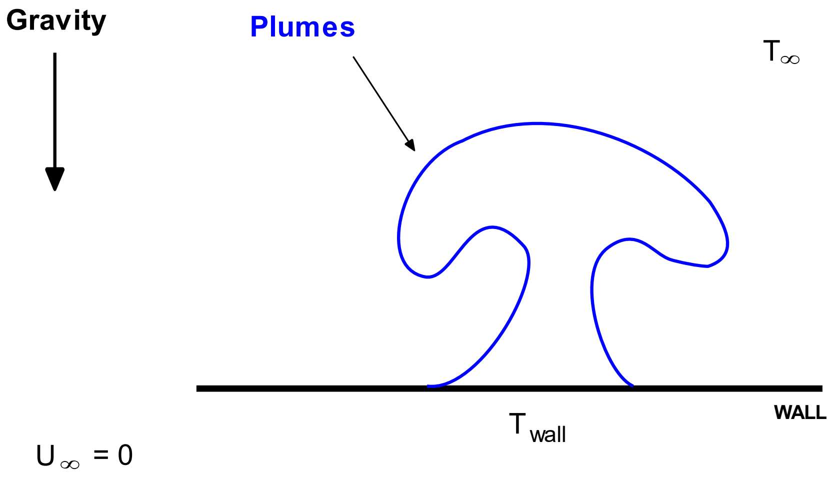


Thermal boundary layer in function of Prandtl number (Pr)

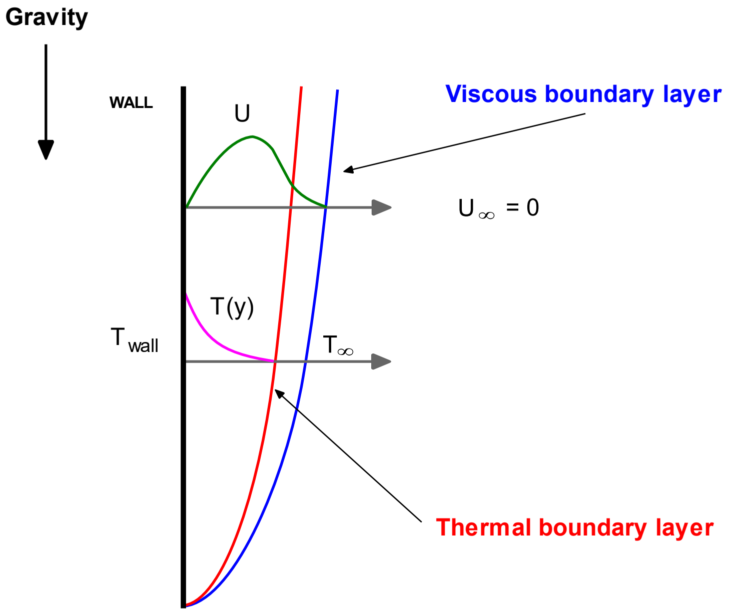
Momentum and thermal boundary layer

- Just as there is a viscous (or momentum) boundary layer in the velocity distribution, there is also a thermal boundary layer.
- Thermal boundary layer thickness is different from the thickness of the viscous sublayer (or momentum), and is fluid dependent.
- The thickness of the thermal sublayer for a high Prandtl number fluid (e.g. water) is much less than the momentum sublayer thickness.
- For fluids of low Prandtl numbers (*e.g.*, air), it is much larger than the momentum sublayer thickness.
- For Prandtl number equal 1, the thermal boundary layer is equal to the momentum boundary layer.

Compressible flows – Boundary layer



Horizontal heated plate immersed in a quiescent fluid.
Natural convection



Vertical heated plate immersed in a quiescent fluid.
Natural convection.

Natural convection in a heated plate

- As the fluid is warmed by the plate, its density decreases, and a buoyant force arises which induces flow motion in the vertical or horizontal direction.
- The force is proportional to $(\rho - \rho_\infty) \times g$, therefore gravity must be considered.

Compressible solvers in OpenFOAM®

- Dealing with compressible flows in OpenFOAM® is not so different from what we have done so far.
- The new steps are:
  - Define the thermophysical variables.
  - Define the boundary conditions and initial conditions for temperature.
  - If you are dealing with turbulence (most of the times), you will need to define the boundary conditions and initial conditions for the turbulent thermal diffusivity.
    - Do not forget to choose the near-wall treatment.
  - Depending on the thermophysical model and physics involved, you will need to define discretization schemes and linear solvers for the new variables and equations, that is, **T**, **h**, **e** and so on.
  - Define solver parameters for the new variables, that is, under-relaxation factors, **SIMPLE**/**PISO**/**PIMPLE** corrections, maximum and minimum allowable pressure or density values, high-speed corrections, and so on.
  - Remember, as for pressure, mesh non-orthogonality and skewness also introduces secondary gradients in the energy equation, $f\left(\nabla T\right)$.

Compressible solvers in OpenFOAM®

- Additionally, the numerics of compressible solvers is a little bit more delicate.
  - Temperature is a bounded quantity, so we need to use accurate and stable methods (preferably TVD).
  - If you are in the presence of shock waves, you need to use TVD methods and gradient limiters.
  - The solvers are very sensitive to overshoots and undershoots in the gradients, so you need to use aggressive limiters.
  - If you are dealing with chemicals reactions or combustion, you need to use accurate and stable methods (preferably TVD).
  - TVD methods requires good meshes and CFL number below 1 for good accuracy and stability.
  - Using steady solvers requires tuning of the under-relaxation factors. Usually, the default values do not work well.
  - The use of local time stepping to reach steady state can improve the convergence rate.
- FYI, we have found that it is tricky to achieve good convergence using a low-RE approach with steady solvers in high-speed compressible flows.

Compressible solvers in OpenFOAM®

- OpenFOAM® comes with many solvers and models that can address a wide physics.
- Compressibility can be introduced in all the modeling capabilities we have seen so far (turbulence modeling and multiphase flows).
- It is also possible to add source terms, deal with moving bodies, or use adaptive mesh refinement.
- You will find the source code of all the compressible solvers in the directories:
  - **`OpenFOAM-8/applications/solvers/compressible`**
  - **`OpenFOAM-8/applications/solvers/combustion`**
  - **`OpenFOAM-8/applications/solvers/heatTransfer`**
  - **`OpenFOAM-8/applications/solvers/lagrangian`**
  - **`OpenFOAM-8/applications/solvers/multiphases`**
- You will find the source code of the thermophysical models in the directory:
  - **`OpenFOAM-8/src/thermophysicalModels`**
  - **`OpenFOAM-8/src/ThermophysicalTransportModels`**

Compressible solvers in OpenFOAM®

- These are the compressible solvers that you will use most of the time in OpenFOAM®.
  - HVAC and low-speed aerodynamics:
    - rhoSimpleFoam, rhoPimpleFoam
  - High-speed aerodynamics:
    - rhoSimpleFoam, rhoPimpleFoam, rhoCentralFoam
  - Buoyancy driven flows (including Boussinesq approximation):
    - buoyantSimpleFoam, buoyantPimpleFoam
  - Conjugate heat transfer
    - chtMultiRegionFoam

Selecting thermophysical properties

```
1     thermoType
2     {
3         type            hePsiThermo;
4         mixture         pureMixture;
5         transport       const;
6         thermo          hConst;
7         equationOfState perfectGas;
8         specie          specie;
9         energy          sensibleEnthalpy;
10    }
11
12    mixture
13    {
14        specie
15        {
16            nMoles      1;
17            molWeight   28.9;
18        }
19        thermodynamics
20        {
21            Cp          1005;
22            Hf          0;
23        }
24        transport
25        {
26            mu          0;
27            Pr          0.713;
28        }
29    }
```

- The thermophysical properties are set in the dictionary *thermophysicalProperties*.
- This dictionary file is located in the directory `constant`.
- Thermophysical models are concerned with energy, heat and physical properties.
- In the sub-dictionary **thermoType** (lines 1-10), we define the thermophysical models.
- The entries in lines 3-4, are determined by the choice of the solver (they are hardwired to the solver).
- The **transport** keyword (line 5). concerns evaluating dynamic viscosity. In this case the viscosity is constant.
- The thermodynamic models (**thermo** keyword) are concerned with evaluating the specific heat Cp (line 6). In this case Cp is constant.
- The **equationOfState** keyword (line 7) concerns to the equation of state of the working fluid. In this case,

$$\rho = \frac{p}{RT}$$

- Line 8 is a fixed option (hardwired to the solver).

Selecting thermophysical properties

```
1      thermoType
2      {
3          type             hePsiThermo;
4          mixture          pureMixture;
5          transport        const;
6          thermo           hConst;
7          equationOfState  perfectGas;
8          specie           specie;
9          energy           sensibleEnthalpy;
10     }
11
12     mixture
13     {
14         specie
15         {
16             nMoles       1;
17             molWeight    28.9;
18         }
19         thermodynamics
20         {
21             Cp           1005;
22             Hf           0;
23         }
24         transport
25         {
26             mu           1.84e-05;
27             Pr           0.713;
28         }
29     }
```

- The form of the energy equation to be used is specified in line 9 (**energy**).
- In this case we are using enthalpy formulation (**sensibleEnthalpy**).
- In this formulation, the following equation is solved,

$$\frac{\partial \rho h}{\partial t} + \nabla \cdot (\rho \mathbf{u} h) + \frac{\partial \rho K}{\partial t} + \nabla \cdot (\rho \mathbf{u} K) - \frac{\partial p}{\partial t} = \nabla \cdot (\alpha_{eff} \nabla e) + \rho \mathbf{g} \cdot \mathbf{u} + S$$

- Alternatively, we can use the **sensibleInternalEnergy** formulation, where the following equation is solved for the internal energy,

$$\frac{\partial \rho e}{\partial t} + \nabla \cdot (\rho \mathbf{u} e) + \frac{\partial \rho K}{\partial t} + \nabla \cdot (\rho \mathbf{u} K) + \nabla \cdot (\mathbf{u} p) = \nabla \cdot (\alpha_{eff} \nabla e) + \rho \mathbf{g} \cdot \mathbf{u} + S$$

- In the previous equations, the effective thermal diffusivity is equal to,

$$\alpha_{eff} = \alpha_{turbulent} + \alpha_{laminar} = \frac{\rho \nu_t}{Pr_t} + \frac{\mu}{Pr} = \frac{\rho \nu_t}{Pr_t} + \frac{k}{c_p}$$

- And $K \equiv |\mathbf{u}|^2/2$ is the kinetic energy per unit mass.

Selecting thermophysical properties

```
1      thermoType
2      {
3          type              hePsiThermo;
4          mixture           pureMixture;
5          transport         const;   ←
6          thermo            hConst;
7          equationOfState   perfectGas;
8          specie            specie;
9          energy            sensibleEnthalpy;
10     }
11
12     mixture
13     {
14         specie
15         {
16             nMoles        1;
17             molWeight     28.9;
18         }
19         thermodynamics
20         {
21             Cp            1005;
22             Hf            0;
23         }
24         transport
25         {
26             mu            1.84e-05;
27             Pr            0.713;
28         }
29     }
```

- In the sub-dictionary **mixture** (lines 12-29), we define the thermophysical properties of the working fluid.
- In line 17, we define the molecular weight.
- In line 21, we define the specific heat.
- The heat of formation is defined in line 22 (not used in this case).
- In this case, we are defining the properties for air at 20° Celsius and at a sea level.
- As we are using the transport model **const** (line 5), we need to define the dynamic viscosity and Prandtl number (lines 26 and 27).
- If you set the viscosity to zero, you solve the Euler equations.
- Remember, transport modeling (line 5), concerns evaluating dynamic viscosity, thermal conductivity and thermal diffusivity.

Selecting thermophysical properties

```
1       thermoType
2       {
3           type            hePsiThermo;
4           mixture         pureMixture;
5           transport       sutherland;
6           thermo          hConst;
7           equationOfState perfectGas;
8           specie          specie;
9           energy          sensibleEnthalpy;
10      }
11
12      mixture
13      {
14          specie
15          {
16              nMoles      1;
17              molWeight   28.9;
18          }
19          thermodynamics
20          {
21              Cp          1005;
22              Hf          0;
23          }
24          transport
25          {
26              As          1.4792e-06;
27              Ts          116;
28          }
29      }
```

- If you use the transport model **sutherland** (line 5), you will need to define the coefficients of the Sutherland model.
- The Sutherland model is defined as follows (OpenFOAM® uses the 2 coefficients formulation):

$$\mu = \frac{A_s \sqrt{T}}{1 + T_s/T}$$

- The Sutherland coefficients are defined in lines 26-27.
- **Remember, you can use the banana method to know all the options available.**

Adjusting the numerical method

- If you choose the **sensibleEnthalpy** formulation, you need to define the convective discretization schemes and linear solvers of the energy equation (enthalpy formulation).

fvSchemes

```
divSchemes
{
        div(phi,K) Gauss linear;
        div(phi,h) Gauss linear;
        div(phid,p) Gauss linear;
        …
        …
        …
}
```

$$\frac{\partial \rho h}{\partial t} + \nabla \cdot (\rho \mathbf{u} h) + \frac{\partial \rho K}{\partial t} + \nabla \cdot (\rho \mathbf{u} K) - \frac{\partial p}{\partial t} = \nabla \cdot (\alpha_{eff} \nabla e) + \rho \mathbf{g} \cdot \mathbf{u} + S$$

fvSolution

```
"(h|rho)"
{
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-8;
        relTol          0.01;
}
        …
        …
        …
```

- Remember, temperature is a bounded quantity, so you need to use non-oscillatory methods.
- For low speed flows, the kinetic energy **K** and the enthalpy **h** can be discretized using the linear method. For high speed flows, is better to use bounded methods.
- Remember to use gradient limiters.
- If you are using a steady solver, remember to set the under-relaxation factors for **h** and **rho**.

Adjusting the numerical method

- If you choose the **sensibleInternalEnergy** formulation, you need to define the convective discretization schemes and linear solvers of the energy equation (internal energy formulation).

fvSchemes

```
divSchemes
{
          div(phi,K) Gauss linear;
          div(phi,e) Gauss linear;
          div(phiv,p) Gauss linear;
          …
          …
          …
}
```

$$\frac{\partial \rho e}{\partial t} + \nabla \cdot (\rho \mathbf{u} e) + \frac{\partial \rho K}{\partial t} + \nabla \cdot (\rho \mathbf{u} K) + \nabla \cdot (\mathbf{u} p) = \nabla \cdot (\alpha_{eff} \nabla e) + \rho \mathbf{g} \cdot \mathbf{u} + S$$

fvSolution

```
"(e|rho)"
{
          solver            PBiCGStab;
          preconditioner    DILU;
          tolerance         1e-8;
          relTol            0.01;
}
          …
          …
          …
```

- Remember, temperature is a bounded quantity, so you need to use non-oscillatory methods.
- For low speed flows, the kinetic energy **K** and the internal energy **e** can be discretized using the linear method. For high speed flows, is better to use bounded methods.
- Remember to use gradient limiters.
- If you are using a steady solver, remember to set the under-relaxation factors for **e** and **rho**.

Final remarks

- When solving the enthalpy formulation of the energy equation,

$$\frac{\partial \rho h}{\partial t} + \nabla \cdot (\rho \mathbf{u} h) + \frac{\partial \rho K}{\partial t} + \nabla \cdot (\rho \mathbf{u} K) - \frac{\partial p}{\partial t} = \nabla \cdot (\alpha_{eff} \nabla e) + \rho \mathbf{g} \cdot \mathbf{u} + S$$

  the pressure work term $\partial p / \partial t$ can be excluded from the solution.

- This has a stabilizing effect on the solution, specially if you are using steady solvers.
- To turn off the pressure work term $\partial p / \partial t$, set the option dpdt to no (**dpdt no;**) in the *thermophysicalProperties* dictionary.

- Finally, when you work with compressible solvers you use absolute pressure and the working units are in Pascals.

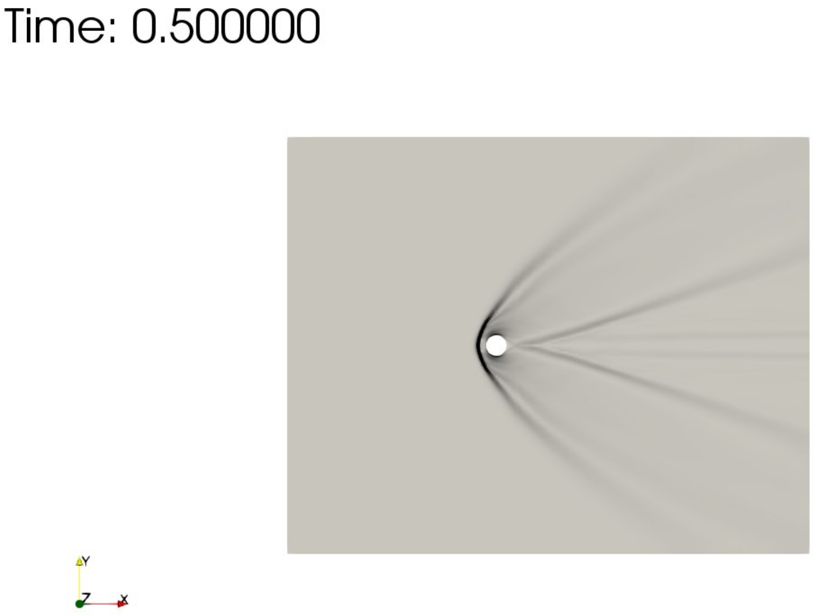
Compressible flows hands-on tutorials

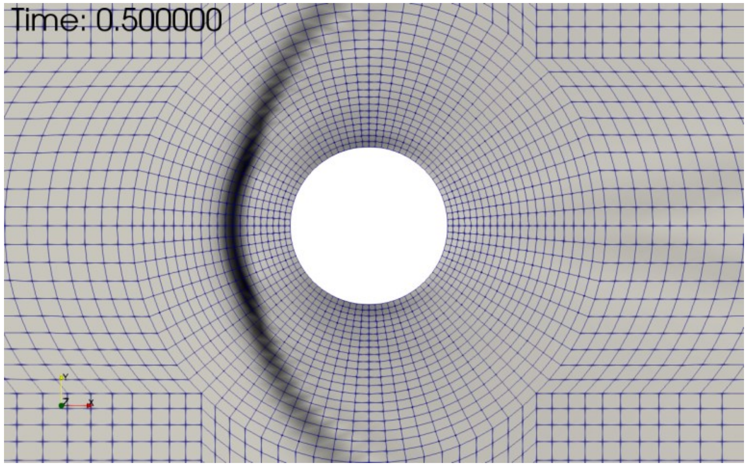
- 2D supersonic cylinder – Shock waves
- Let us run this case. Go to the directory:

```
$PTOFC/advanced_physics/compressible/supersonic_cyl
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

2D supersonic cylinder – Shock waves



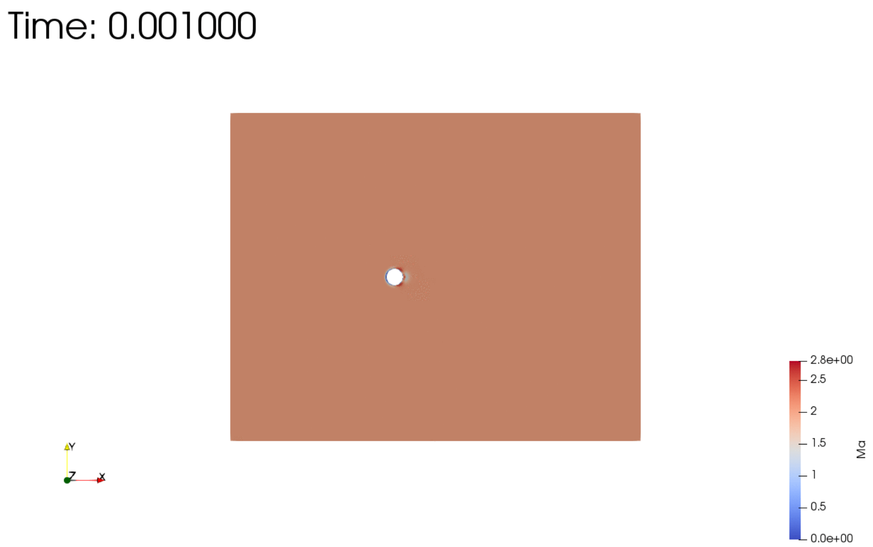


Shock wave visualization using numerical Schlieren (density gradient)

- Shock waves are strong discontinuities that need to be treated using high resolution schemes.
- Additionally, the non-orthogonality add extra complications to this problem.

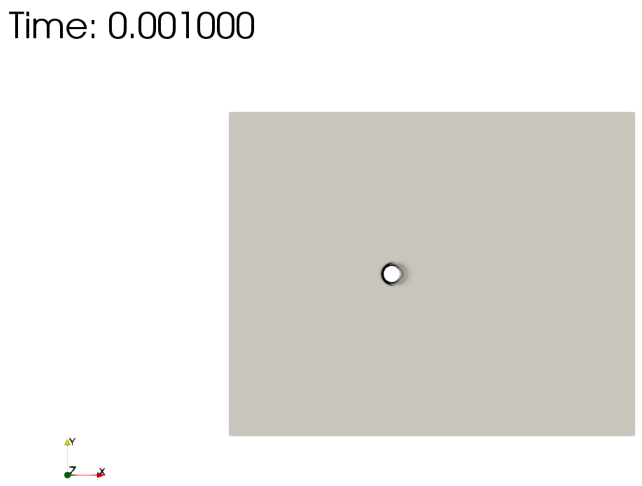
Compressible flows hands-on tutorials

2D supersonic cylinder – Shock waves



Mach number contours

http://www.wolfdynamics.com/training/compressible/image3.gif



Schlieren contours

http://www.wolfdynamics.com/training/compressible/image4.gif

Compressible flows hands-on tutorials

2D supersonic cylinder – Shock waves

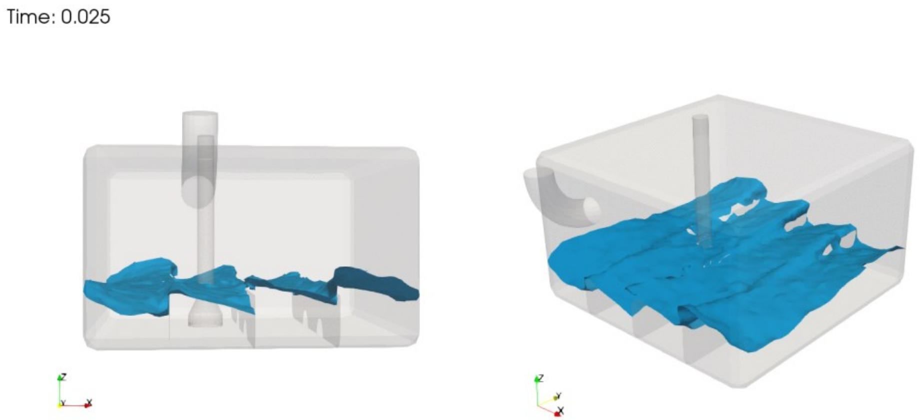
- In this case we will use the solver `rhoPimpleFoam` with transonic corrections.
- By enabling transonic correction we can use this solver to tackle trans-sonic/supersonic flows.
- Transonic corrections are enabled in the **PIMPLE** block of the dictionary *fvSolution*,
    - **transonic yes;**
- `rhoPimpleFoam` is an unsteady solver, but if you are interested in a steady solution you can use local time stepping.
- As the flow is compressible, we need to define the thermodynamical properties of the working fluid.
- This is done in the dictionary *constant/thermophysicalProperties*.
- We also need to define the boundary conditions and initial conditions for the temperature field.
- Additionally, if you are using a turbulence model, you will need to define wall functions for the thermal diffusivity.
    - The rest of the turbulent variables are defined as in incompressible flows.
- Finally, adjust the numerics according to your physics.
- You will find the instructions of how to run the cases in the file *README.FIRST* located in the case directory.

Roadmap

A crash introduction to:

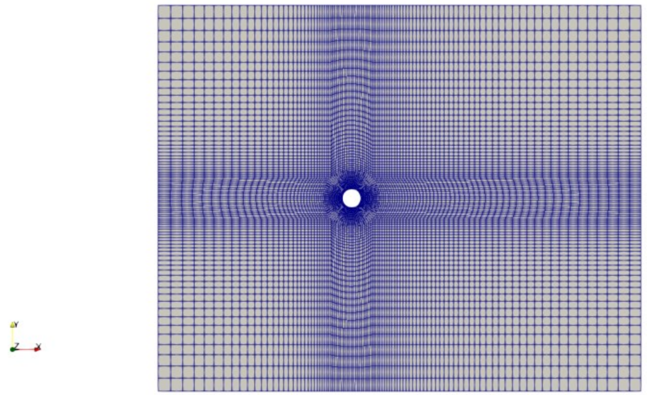
1. ~~Turbulence modeling in OpenFOAM®~~
2. ~~Multiphase flows modeling in OpenFOAM®~~
3. ~~Compressible flows in OpenFOAM®~~
4. **Moving bodies in OpenFOAM®**
5. Source terms in OpenFOAM®
6. Scalar transport pluggable solver

Moving bodies in OpenFOAM® – A few examples



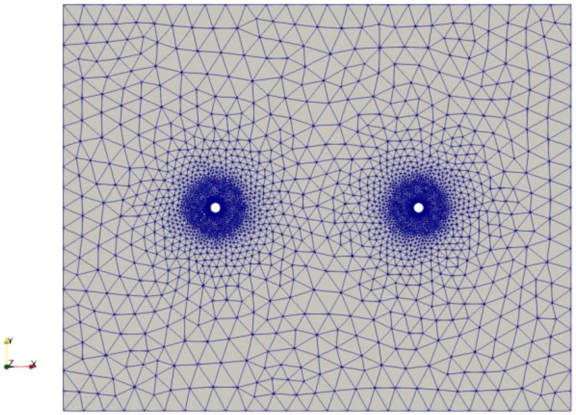
Sloshing tank

http://www.wolfdynamics.com/training/dynamicMeshes/sloshing1.gif



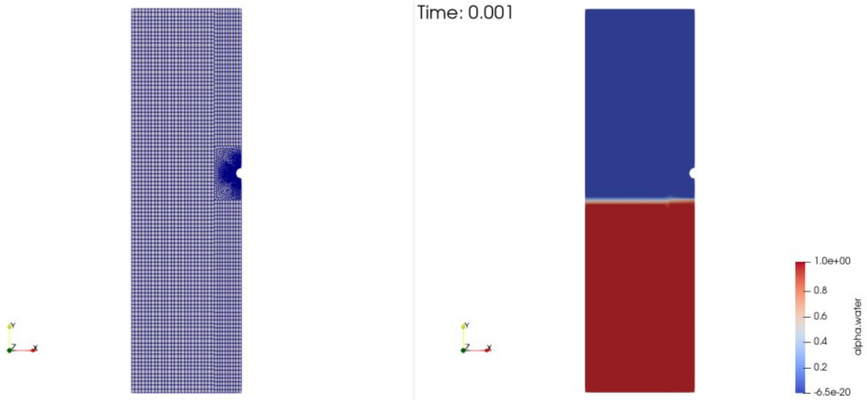
Oscillating cylinder (prescribed motion)

http://www.wolfdynamics.com/training/dynamicMeshes/meshMotion1



Prescribed motion with multiple bodies

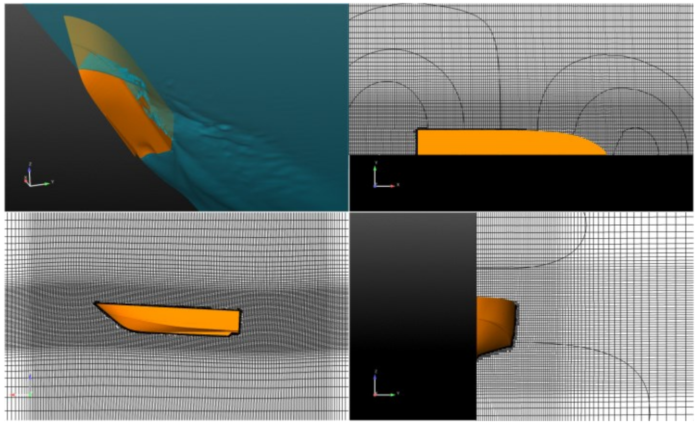
http://www.wolfdynamics.com/training/dynamicMeshes/meshMotion1



Layering with mesh zones interface

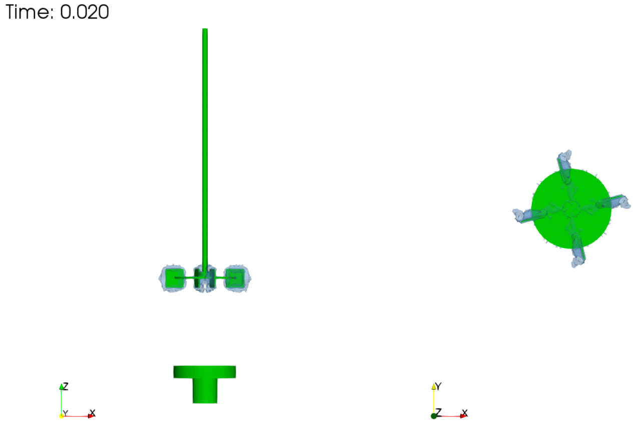
http://www.wolfdynamics.com/training/dynamicMeshes/layeringMesh.gif

Moving bodies in OpenFOAM® – A few examples



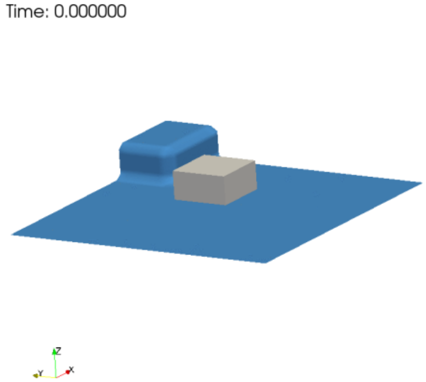
Sea keeping

http://www.wolfdynamics.com/training/dynamicMeshes/seakeeping.gif



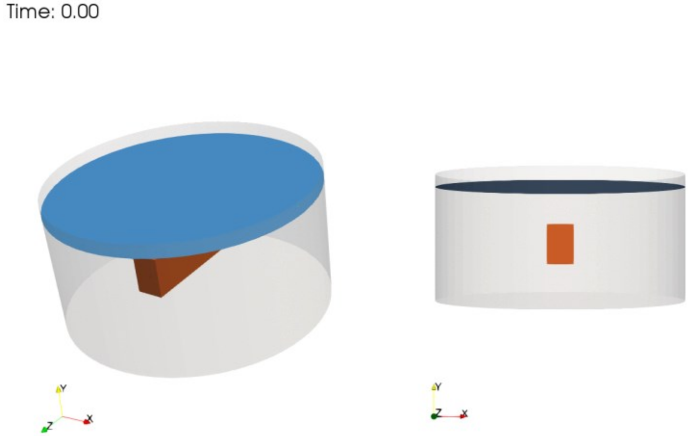
Continuous stirring tank reactor (CSTR)

http://www.wolfdynamics.com/training/movingbodies/image13.gif



Falling body (6 DoF)

http://www.wolfdynamics.com/training/movingbodies/image5.gif



VOF with sliding meshes

http://www.wolfdynamics.com/training/mphase/image33.gif

Moving bodies in OpenFOAM®

- OpenFOAM® comes with many solvers and models that can address a wide physics.
- Moving bodies can be added to all the modeling capabilities we have seen so far (turbulence modeling, multiphase flows, and compressible flows).
- Several class of motions can be simulated in OpenFOAM®:
    - Prescribed motion.
    - Rigid body motion.
    - Sliding meshes.
    - MRF.
- Setting moving bodies simulations is not so different from what we have done so far.
- The main difference is that we must assign a motion type to a surface patch, a cell region, or the whole domain.

Moving bodies in OpenFOAM®

- The mesh motion solver is selected in the dictionary *constant/dynamicMeshDict*.
- In the case of prescribed motion of a boundary patch, the motion is assigned in the dictionary *0/pointDisplacement*.
- Also, the boundary type of the moving walls must be **movingWallVelocity**, this is set in the dictionary `0/U`.
- And as usual, you will need to adjust the numerics according to your physics.
- To use the moving bodies capabilities, you will need to use solvers able to deal with dynamic meshes.
- To find which solvers work with dynamic meshes, go to the solvers directory by typing `sol` in the command line interface. Then type in the terminal:
  - `$> grep -r dynamicFvMesh.H`
- The solvers containing this header file support dynamic meshes.
- A few solvers that work with dynamic meshes: `interFoam`, `pimpleFoam`, `rhoPimpleFoam`, `buoyantPimpleFoam`.

Moving bodies in OpenFOAM®

- You will find the source code of all the mesh motion libraries in the directories:
  - `OpenFOAM-8/src/dynamicFvMesh`
  - `OpenFOAM-8/src/dynamicMesh`
  - `OpenFOAM-8/src/fvMotionSolver`
  - `OpenFOAM-8/src/rigidBodyDynamics`
  - `OpenFOAM-8/src/rigidBodyMeshMotion`
  - `OpenFOAM-8/src/rigidBodyState`
  - `OpenFOAM-8/src/sixDoFRigidBodyMotion`
  - `OpenFOAM-8/src/sixDoFRigidBodyState`
- You will find the source code of the prescribed patch motion in the directory:
  - `OpenFOAM-8/src/fvMotionSolver/pointPatchFields/derived`
- You will find the source code of the restraints/constraints of rigid body motion solvers in the directory:
  - `OpenFOAM-8/src/rigidBodyDynamics/joints`
  - `OpenFOAM-8/src/sixDoFRigidBodyMotion/sixDoFRigidBodyMotion`

Moving bodies hands-on tutorials

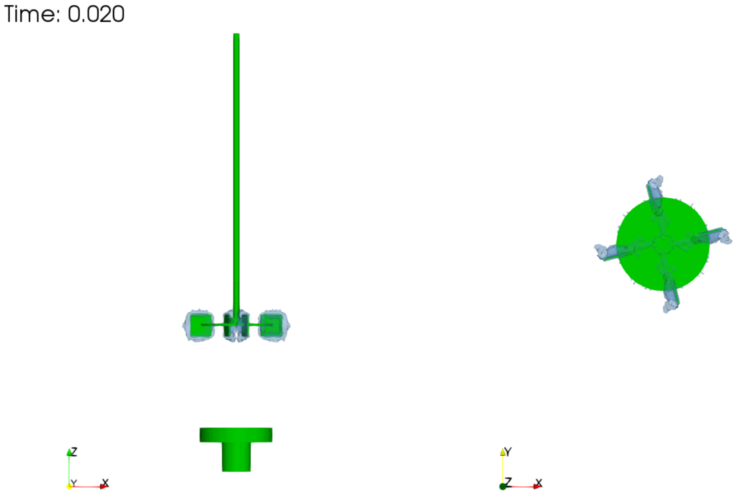
- Continuous stirring tank reactor – Sliding meshes and MRF
- Let us run this case. Go to the directory:

```
$PTOFC/advanced_physics/sliding_meshes_MRF/CSTR
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

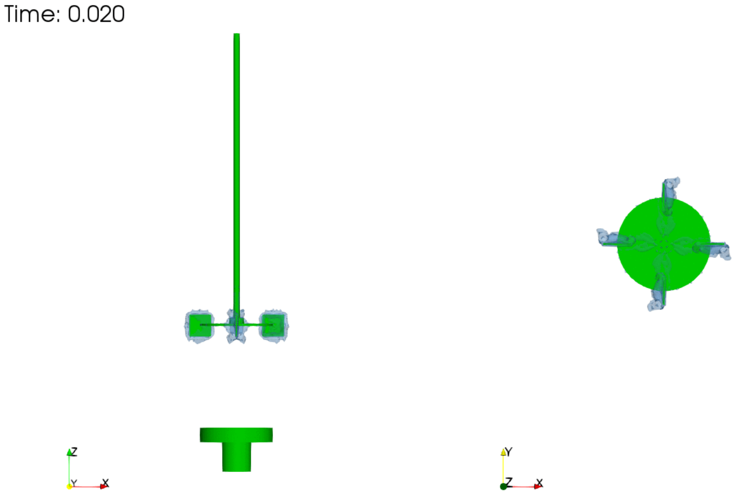
Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF



Sliding grids – Unsteady solver

http://www.wolfdynamics.com/training/movingbodies/image13.gif



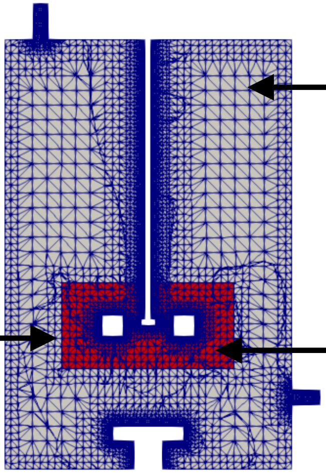
MRF – Steady solver

http://www.wolfdynamics.com/training/movingbodies/image14.gif

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF

- We already created this mesh during the meshing module.
- We will only address the differences in meshing for an MRF simulation and a sliding mesh simulation.
- In this case, at the end of the meshing stage we obtained a **faceZone** and a **cellZone**.
- This is a conforming mesh, that is, the cells in the interface of the inner and outer regions are perfectly matching.
- For MRF simulations, the **cellZone** can be used to assign the MRF properties to the rotating zone.
- For sliding meshes or non-conforming meshes, there is an extra step where we need to split the mesh in two regions and create the interface patches between the fix zone and the rotating zone (the solution will be interpolated in these patches).



Cell region 2
(fix region)s

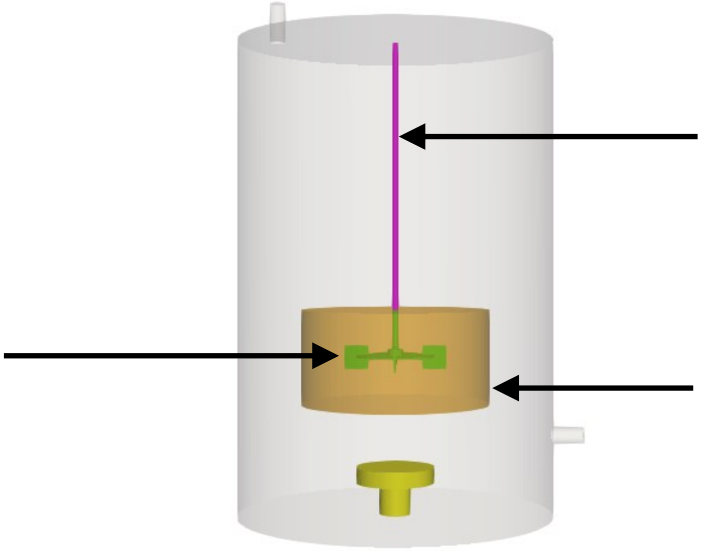
Face region between fix and
rotating regions
(face_inner_volume)

Cell region 1
(cell_inner_volume)

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF

- In the MRF approach, the governing equations are solved in a relative rotating frame in the selected rotating zone.
- Additional source terms that model the rotation effect are taken into account.
- You select the rotating zone and set the rotation properties in the dictionary `constant/MRFProperties`.
- In this case, the mesh is conforming.



Shaft
type rotatingWallVelocity;
origin (0 0 0);
axis (0 0 1);
omega constant 12.566370;
value uniform (0 0 0);

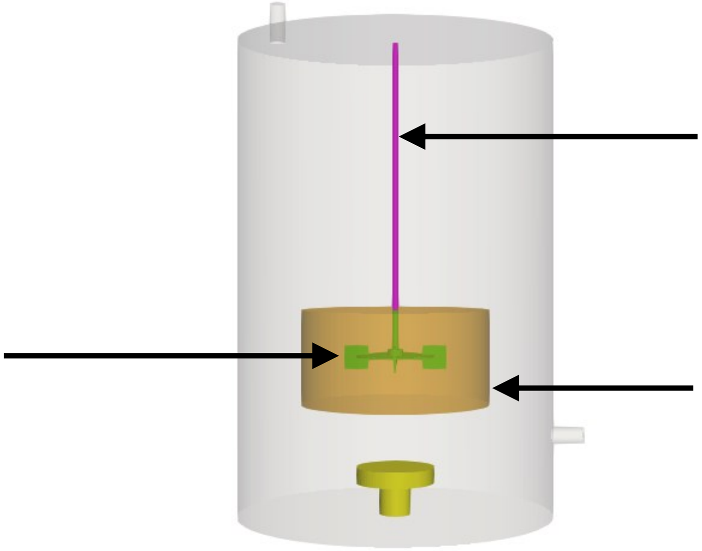
Impeller
type movingWallVelocity;
value uniform (0 0 0);

Inner region generated during meshing - `MRFProperties`

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF

- In the sliding meshes approach, the selected rotating region is physically rotating.
- As the meshes are non-conforming, the solution between the rotating region and the fix region must be interpolated using arbitrary mesh interface.
- In the sliding meshes approach, is not enough to only identify the rotating region.
- We also need to create the interface patches between the fix zone and the rotating zone.



Shaft
type rotatingWallVelocity;
origin (0 0 0);
axis (0 0 1);
omega constant 12.566370;
value uniform (0 0 0);

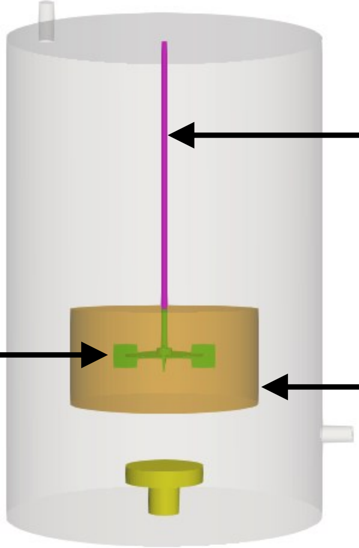
Impeller
type movingWallVelocity;
value uniform (0 0 0);

Inner region – `dynamicMeshDict`

Arbitrary mesh interface – `createBafflesDicts`

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF



Shaft
type rotatingWallVelocity;
origin (0 0 0);
axis (0 0 1);
omega constant 12.566370;
value uniform (0 0 0);

Impeller
type movingWallVelocity;
value uniform (0 0 0);

Inner region and arbitrary mesh interface

`constant/dynamicMeshDict` – For sliding meshes

```
dynamicFvMesh       dynamicMotionSolverFvMesh;
motionSolverLibs ( "libfvMotionSolvers.so" );
solver                    solidBody;
cellZone                  cell_inner_volume;
solidBodyMotionFunction  rotatingMotion;
origin    (0 0 0);
axis      (0 0 1);
omega     constant 12.566370;
```

`constant/MRFProperties` – For MRF approach

```
cellZone    cell_inner_volume;
active      yes;

// Fixed patches (by default they move’ with the MRF zone)
nonRotatingPatches ();

origin    (0 0 0);
axis      (0 0 1);
omega     constant 12.566370;
```

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF

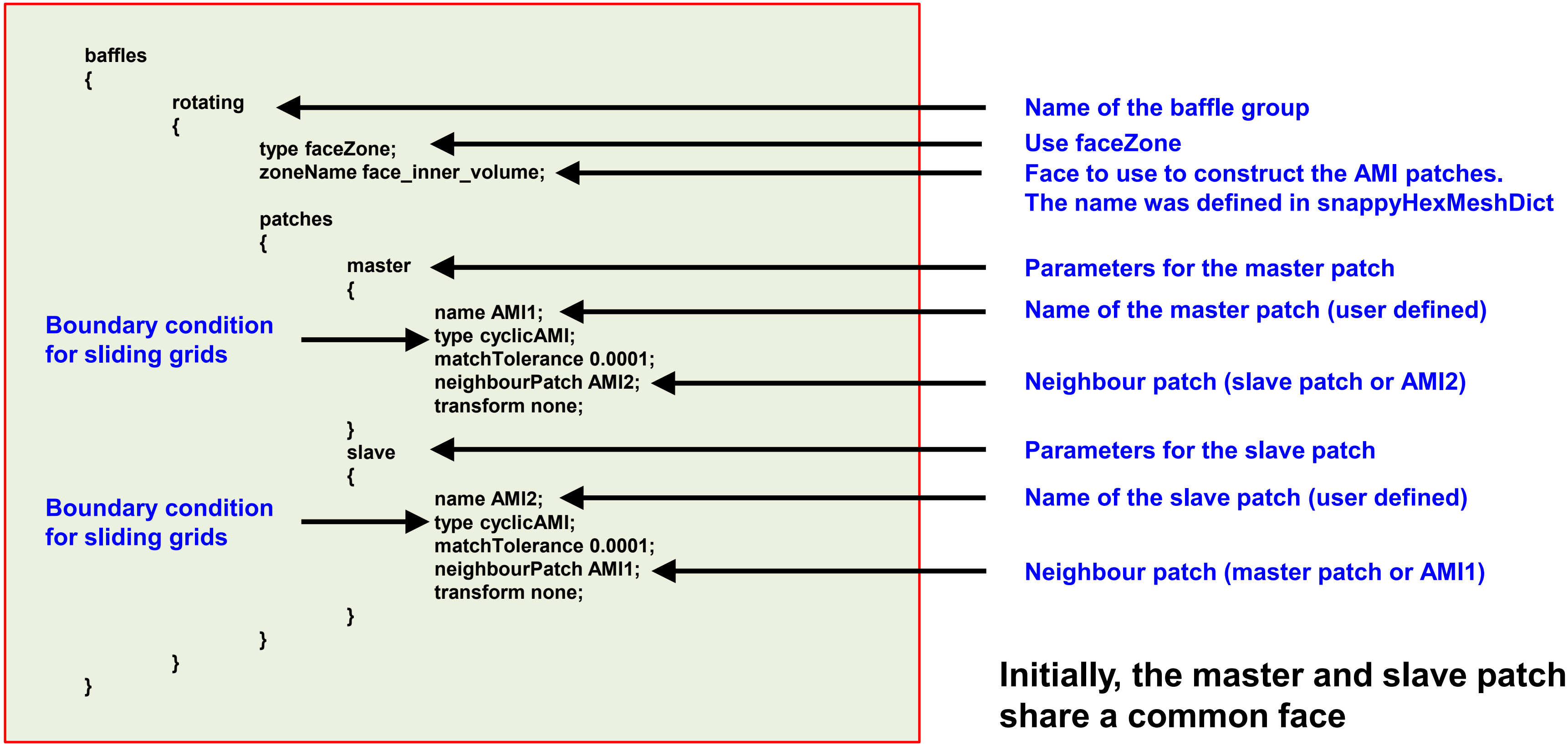
- For siding meshes, we need to create separated regions.
- In this case, to create the two regions we proceed as follows,

```
1.  $> createBaffles -overwrite
2.  $> mergeOrSplitBaffles -split -overwrite
```

- In step 1, we split the mesh in regions using the baffles (**faceZone**), created during the meshing stage.
- We also create the **cyclicAMI** patches **AMI1** and **AMI2**.
- At this point we have two regions and one zone. However, the two regions are stich together via the patches **AMI1** and **AMI2**.
- In step 2, we topologically split the patches **AMI1** and **AMI2**. As we removed the link between **AMI1** and **AMI2**, the regions are free to move.

Continuous stirring tank reactor – Sliding meshes and MRF

- The utility `createBaffles`, reads the dictionary *`createBafflesDict`*.
- With this utility we create the interface patches between the fix zone and the rotating zone.



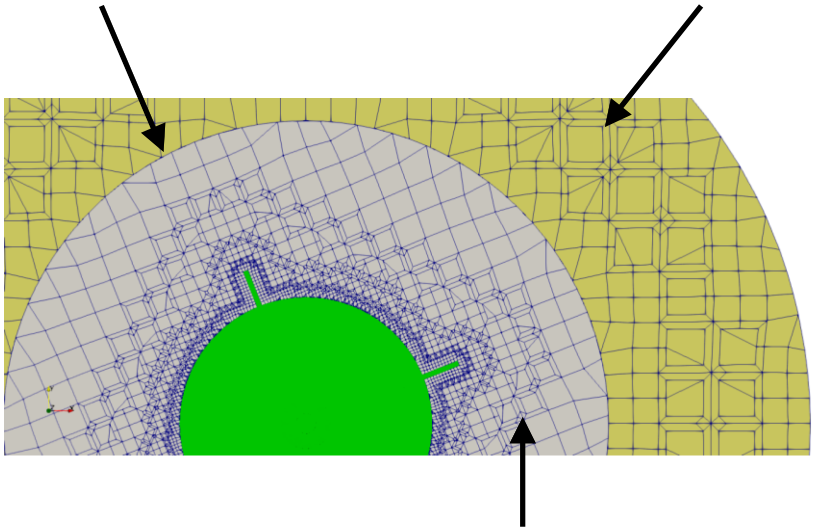
```
baffles
{
          rotating
          {
                    type faceZone;
                    zoneName face_inner_volume;

                    patches
                    {
                              master
                              {
                                        name AMI1;
                                        type cyclicAMI;
                                        matchTolerance 0.0001;
                                        neighbourPatch AMI2;
                                        transform none;
                              }
                              slave
                              {
                                        name AMI2;
                                        type cyclicAMI;
                                        matchTolerance 0.0001;
                                        neighbourPatch AMI1;
                                        transform none;
                              }
                    }
          }
}
```

Name of the baffle group
Use faceZone
Face to use to construct the AMI patches.
The name was defined in snappyHexMeshDict
Parameters for the master patch
Name of the master patch (user defined)
Neighbour patch (slave patch or AMI2)
Parameters for the slave patch
Name of the slave patch (user defined)
Neighbour patch (master patch or AMI1)
Boundary condition for sliding grids
Boundary condition for sliding grids

Initially, the master and slave patches share a common face

Continuous stirring tank reactor – Sliding meshes and MRF

- In sliding mesh simulations, the solution is interpolated back-and-forth between the regions.
- The interpolation is done at the arbitrary mesh interface patches (AMI) .
- To reduce interpolation errors at the AMI patches, the meshes should be similar in the master and slave patches.

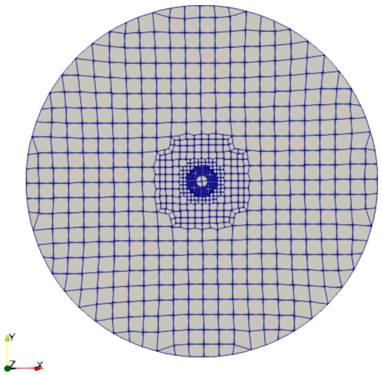


AMI interface

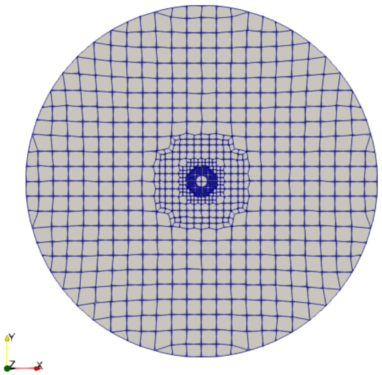
Fix domain

Rotating domain

http://www.wolfdynamics.com/training/movingbodies/image8.gif



Rotating patch
Master patch



Fix patch
Slave patch

Moving bodies hands-on tutorials

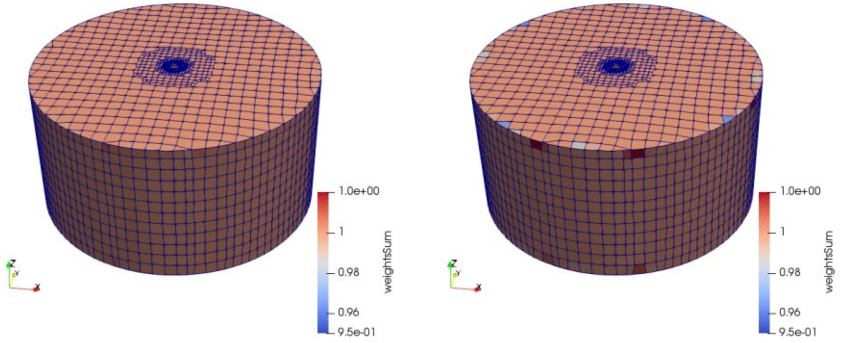
Continuous stirring tank reactor – Sliding meshes and MRF

- At this point, the mesh is ready to use.
- You can visualize the mesh using `paraFoam`.
- If you use `checkMesh`, it will report that there are two regions.
- In the dictionary *`constant/dynamicsMeshDict`* we set which region will move and the rotation parameters.
- To preview the region motion, in the terminal type:
  - `$> moveDynamicMesh`
- To preview the region motion and check the quality of the AMI interfaces, in the terminal type:
  - `$> moveDynamicMesh -checkAMI -noFunctionObjects`
- In our YouTube channel you can find a step-by-step video explaining this case. ℹ

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF

- The command `moveDynamicMesh –checkAMI` will print on screen the quality of the AMI interfaces for every time step.
- Ideally, you should get the AMI patches weights as close as possible to one.
- Weight values close to one will guarantee a good interpolation between the AMI patches.



http://www.wolfdynamics.com/training/movingbodies/image9.gif

Name of the AMI patch

Name of the AMI patch

```
...
...
...

Calculating AMI weights between owner patch: AMI1 and neighbour patch: AMI2

AMI: Creating addressing and weights between 2476 source faces and 2476 target faces

AMI: Patch source sum(weights) min/max/average = 0.94746705, 1.0067199, 0.99994232

AMI: Patch target sum(weights) min/max/average = 0.94746692, 1.0004497, 0.99980782

...
...
...
```

Number of faces in the AMI patches

AMI1 patch weights

AMI2 patch weights

Moving bodies hands-on tutorials

Continuous stirring tank reactor – Sliding meshes and MRF

- At this point, we are ready to run the simulation.
- You can choose between running using sliding meshes (directory **`sliding_piso`**) or MRF (directory **`MRF_simple`**).
- You can use the solver `pimpleFoam` for sliding meshes and the solver `simpleFoam` for MRF.
- You can also use `pimpleFoam` (unsteady solution) for the MRF. However, it is not computationally efficient, the idea of MRF is to reach a steady solution fast.
- You can also try `pimpleFoam` (with local time stepping (LTS).
- You will find the instructions of how to run the cases in the file *`README.FIRST`* located in the case directory.

Moving bodies hands-on tutorials

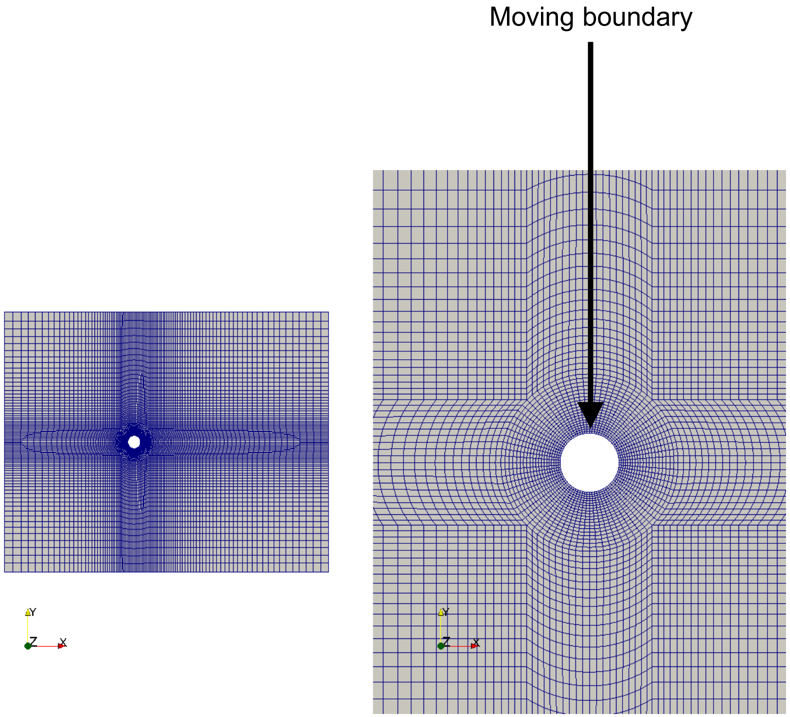
- Oscillating cylinder – Prescribed motion
- Let us run this case. Go to the directory:

```
$PTOFC/advanced_physics/prescribed_motion/oscillatingCylinder
```

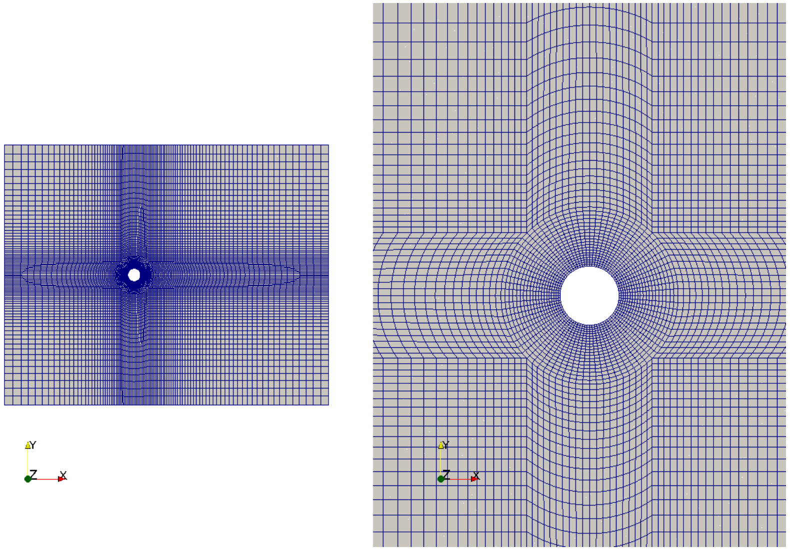
- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Moving bodies hands-on tutorials

Oscillating cylinder – Prescribed motion



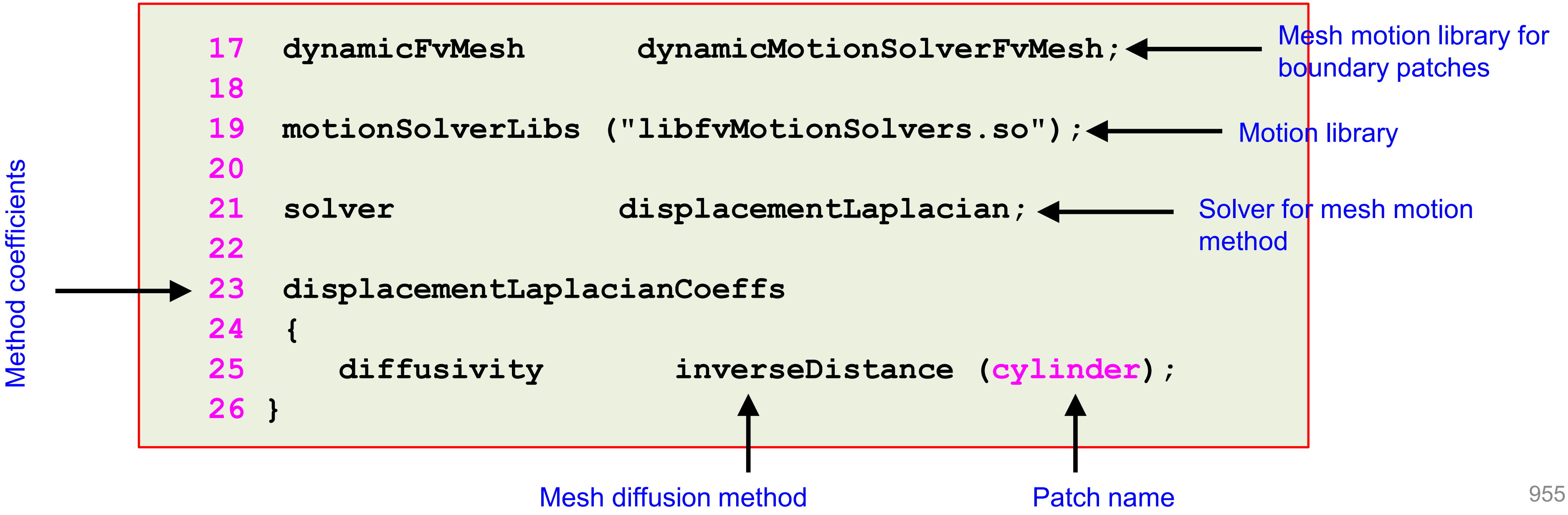
http://www.wolfdynamics.com/training/movingbodies/image10.gif



Cylinder prescribed motion – Oscillating motion

Oscillating cylinder – Prescribed motion

- In the dictionary *constant/dynamicMeshDict* we select the mesh morphing method and the boundary patch that it is moving (lines 21 and 25, respectively).
- There are many mesh morphing methods implemented in OpenFOAM®.
- Mesh morphing is based in diffusing or propagating the mesh deformation all over the domain.
- You will need to find the best method for your case.
- The setup used in this case works fine most of the times.



```
17  dynamicFvMesh       dynamicMotionSolverFvMesh;
18
19  motionSolverLibs ("libfvMotionSolvers.so");
20
21  solver             displacementLaplacian;
22
23  displacementLaplacianCoeffs
24  {
25      diffusivity       inverseDistance (cylinder);
26 }
```

Mesh motion library for boundary patches

Motion library

Solver for mesh motion method

Method coefficients

Mesh diffusion method

Patch name

Oscillating cylinder – Prescribed motion

- In the dictionary *0/pointDisplacement* we select the prescribed body motion.
- In this case we are using **oscillatingDisplacement** (line 44) for the **cylinder** patch.
- Each method has different input values. In this case it is required to define the amplitude (line 45) and the angular velocity (line 46) in rad/s.
- If the patch is not moving, we assign to it a fixedValue boundary conditions (lines 37-41).

```
37  in
38  {
39      type                fixedValue;
40      value               uniform (0 0 0);
41  }
42  cylinder
43  {
44      type                oscillatingDisplacement;
45      amplitude           ( 0 1 0 );
46      omega               6.28318;
47      value               uniform ( 0 0 0 );
48  }
```

← Dummy value for paraview

Oscillating cylinder – Prescribed motion

- You must assign the boundary condition **movingWallVelocity** to all patches that are moving.
- This is done in the dictionary *0/U*.

```
cylinder
{
        type                    movingWallVelocity;
        value                   uniform (0 0 0);
}
```

- And as usual, you will need to adjust the numerics according to your physics.
- In this case we need to solve the new fields **cellDisplacement** and **diffusivity**, which are related to the mesh motion and morphing.
- In the dictionary *fvSolution*, you will need to add a linear solver for the field **cellDisplacement**.
- In the dictionary *fvSchemes*, you will need to add the discretization schemes related to the mesh morphing diffusion method, **laplacian(diffusivity, cellDisplacement)**.
- If you are dealing with turbulence modeling the treatment of the wall functions is the same as if you were working with fixed meshes.

Moving bodies hands-on tutorials

Oscillating cylinder – Prescribed motion

- At this point, we are ready to run the simulation.
- You will find the instructions of how to run the cases in the file *README.FIRST* located in the case directory.
- Before running the simulation, you can check the mesh motion.
- During this check, you can use large time-steps as we re not computing the solution, we are only interested in checking the motion.
- To check the mesh motion, type in the terminal:

```
$> moveDynamicMesh -noFunctionObjects
```

Moving bodies hands-on tutorials

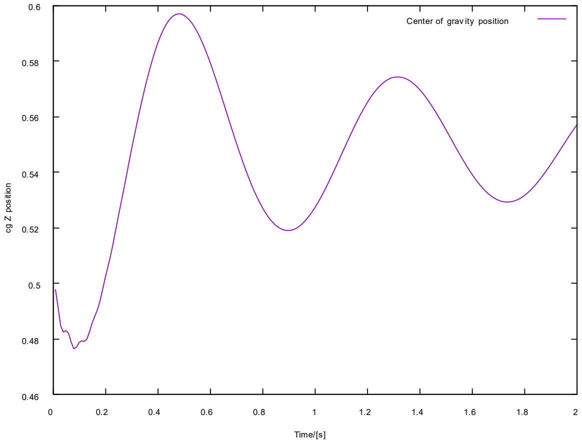
- Floating body – Rigid body motion
- Let us run this case. Go to the directory:

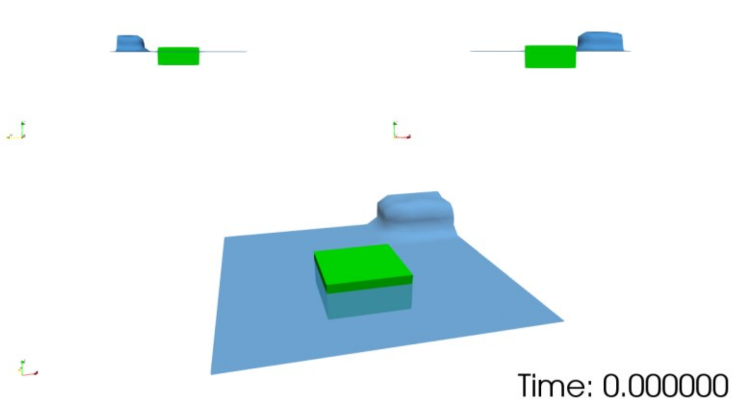
```
$PTOFC/advanced_physics/rigid_body_motion/floatingObject
```

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

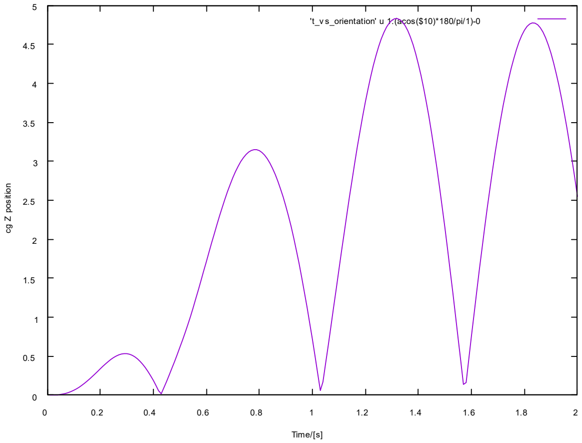
Moving bodies hands-on tutorials

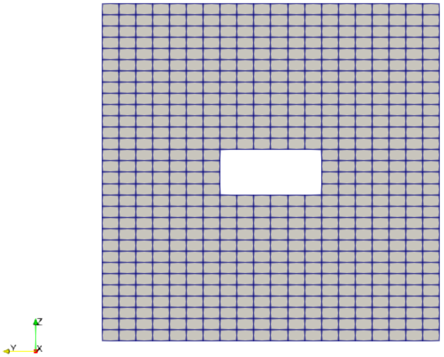
Floating body – Rigid body motion





http://www.wolfdynamics.com/training/movingbodies/image11.gif





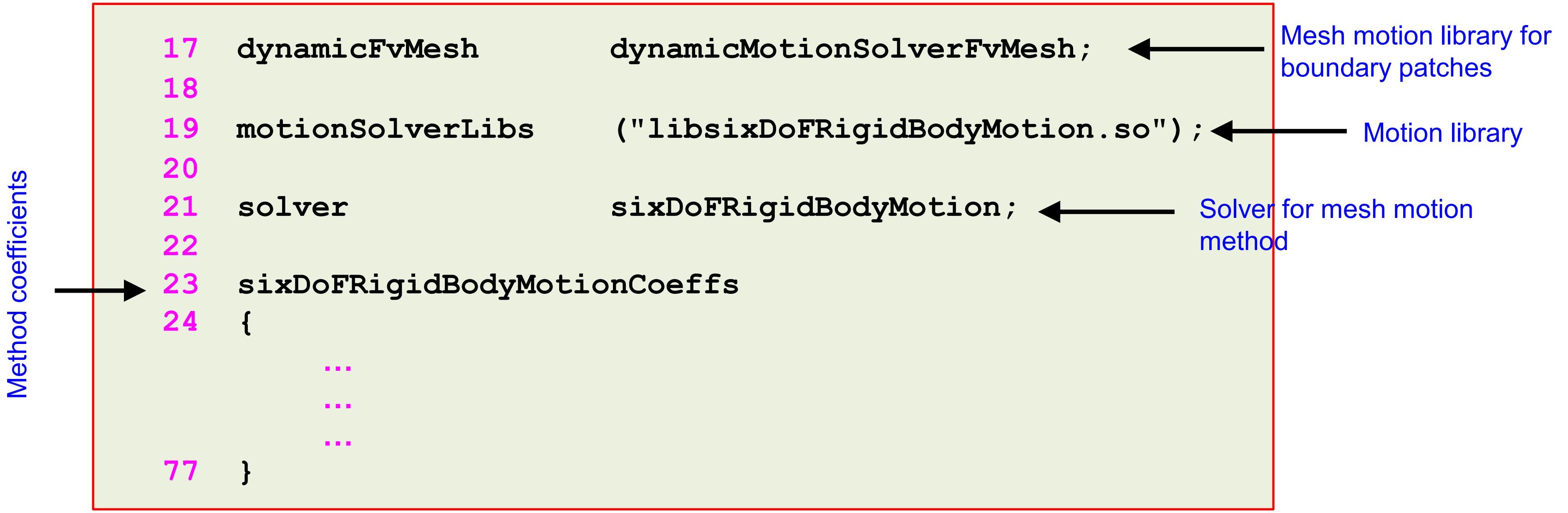
http://www.wolfdynamics.com/training/movingbodies/image12.gif

Floating body simulation with VOF and turbulence modeling.

Moving bodies hands-on tutorials

Floating body – Rigid body motion

- As for prescribed motion, in rigid body motion the mesh morphing is based in diffusing or propagating the mesh deformation all over the domain.
- In the dictionary *constant/dynamicMeshDict* we select the mesh morphing library and rigid body motion library (lines 17-21).
- The rigid motion solver will compute the response of the body to external forces.
- In lines 23-77, we define all the inputs required by the rigid motion solver.



```
17   dynamicFvMesh          dynamicMotionSolverFvMesh;
18
19   motionSolverLibs       ("libsixDoFRigidBodyMotion.so");
20
21   solver                 sixDoFRigidBodyMotion;
22
23   sixDoFRigidBodyMotionCoeffs
24   {
          ...
          ...
          ...
77   }
```

Mesh motion library for boundary patches

Motion library

Solver for mesh motion method

Method coefficients

Floating body – Rigid body motion

- The dictionary *constant/dynamicMeshDict* (continuation).

```
sixDoFRigidBodyMotionCoeffs
{
    patches         (floatingObject);

    innerDistance   0.1;
    outerDistance   0.4;

    centreOfMass    (0.5 0.5 0.5);
    mass            5;
    momentOfInertia (0.08 0.08 0.1);

    report          on;


    solver
    {
        type Newmark;
    }
      ...
      ...
      ...
```

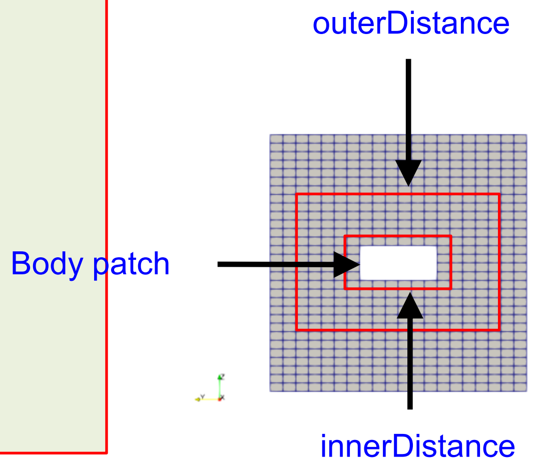
Moving patch

Mesh deformation limits.
The mesh will not be deformed in the fringe located within innerDistance and outerDistance (distance normal to the wall)

Physical properties of the body

Report on screen position of the body

Rigid body motion solver



Set it to zero if you do not want to apply mesh morphing to the inner region

Moving bodies hands-on tutorials

Floating body – Rigid body motion

- The dictionary *constant/dynamicMeshDict* (continuation).

```
    constraints
    {
        fixedAxis
        {
            sixDoFRigidBodyMotionConstraint axis;
            axis (0 1 0);
        }

        fixedLine
        {
            sixDoFRigidBodyMotionConstraint line;
            centreOfRotation (0.5 0.5 0.5);
            direction (0 0 1);
        }

    }

    restraints
    {

    }

}
```

Motion constraints
If you do not give any constraint, the body is free to move in all directions.

Body restraints
Restraints can be used to damp the acceleration of the body.
In this case, we are not using restraints

Moving bodies hands-on tutorials

Floating body – Rigid body motion

- In the dictionary *0/pointDisplacement* we select the body motion.
- For rigid body motion, the body motion is computed by the solver, therefore, we use the boundary condition calculated.

```
floatingObject
{
    type                calculated;
    value               uniform (0 0 0);
}
```

- You must assign the boundary condition **movingWallVelocity** to all patches that are moving. This is done in the dictionary *0/U*.

```
floatingObject
{
    type                movingWallVelocity;
    value               uniform (0 0 0);
}
```

- If you are dealing with turbulence modeling the treatment of the wall functions is the same as if you were working with fixed meshes.

Floating body – Rigid body motion

- And as usual, you will need to adjust the numerics according to your physics.
- In the case directory, you will find the script *extractData*.
- This script can be used to extract the position of the body during the simulation.
- In order to use the *extractData* script, you will need to save the log file of the simulation.
- At this point, we are ready to run the simulation.
- We will use the solver `interFoam`.
- You will find the instructions of how to run the cases in the file *README.FIRST* located in the case directory.

Roadmap

A crash introduction to:

1. ~~Turbulence modeling in OpenFOAM®~~
2. ~~Multiphase flows modeling in OpenFOAM®~~
3. ~~Compressible flows in OpenFOAM®~~
4. ~~Moving bodies in OpenFOAM®~~
5. **Source terms in OpenFOAM®**
6. Scalar transport pluggable solver

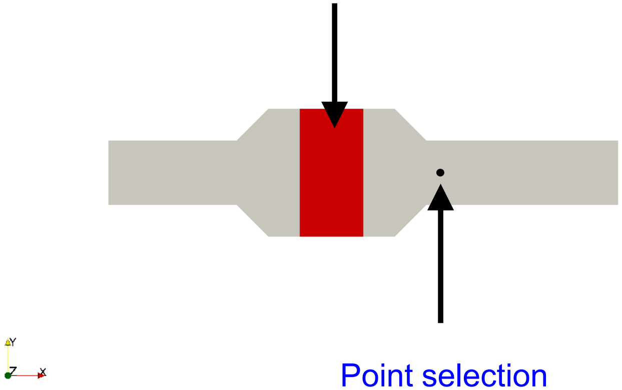
Source terms in OpenFOAM®

- In addition to all modeling capabilities we have seen so far, you can also add source terms to the governing equations without the need of modifying the original source code.
- This functionality is provided via the dictionary *fvOptions*, which is located in the directory **system**.
- There are many source terms implemented in OpenFOAM®, you can even use **codeStream** to program your own source term without the need of recurring to high level programming.
- The *fvOptions* functionality work with most of the solvers that deal with advanced modeling capabilities.
- Remember, the solver icoFoam is very basic with no modeling capabilities. Therefore, you can not use the *fvOptions* functionality and many other modeling and postprocessing capabilities with it.
- You will find the source code of the source terms in the directory:
  - **OpenFOAM-8/src/fvOptions**

Source terms in OpenFOAM®

- To use source terms, you must first select where you want to apply it
- You can apply a source term in the whole domain, a set of cells, a cell zone, or a point (or group of points).
- You can create the set of cells at meshing time or you can use the utility `topoSet` to select a group of cells.
- By the way, boundary conditions are source terms added at the boundary patches, and MRF are source terms added to a cell selection.

Set of cell



Point selection

Source terms in OpenFOAM®

```
1   user_defined_name
2   {
3       type           name_of_source_term;
4
5       active         true;
6
7       input_coefficients
8       {
9           timeStart       0.5;
10          duration        2.0;
11
12          selectionMode   points;
13          points
14          (
15              (3 0 0)
16          );
17
18          //selectionMode    cellZone;
19          //cellZone         filter;
20
21          volumeMode       absolute;
22

            ...
            ...
            ...

23      }
24  }
```

- The source terms can be selected in the dictionary *fvOptions*, and they can be modified on-the-fly.
- This dictionary file is located in the directory **system**.
- Hereafter we show a generic *fvOptions* dictionary.
- According to the source term you selected, you will need to give different input parameters in the input coefficients section (lines 7-23).
- The input parameters indicated with an arrow can be used with any source term. Most of then are self explanatory.
- The **volumeMode** keyword (line 21) let you choose between absolute and specific.
  - **absolute**: input values are given as quantity/volume.
  - **specific**: input values are given as quantity.

- Remember, you can use the banana method to know all source terms and options available.
- You can also read the source code.

Source terms hands-on tutorials

- Filter source term
- Let us run this case. Go to the directory:

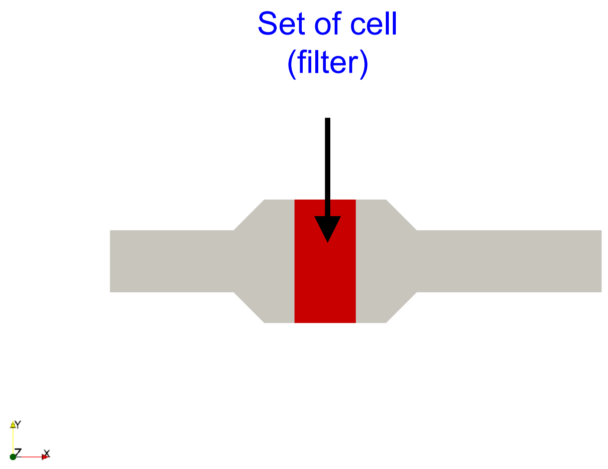
```
$PTOFC/advanced_physics/source_terms/filter/porous_source
```

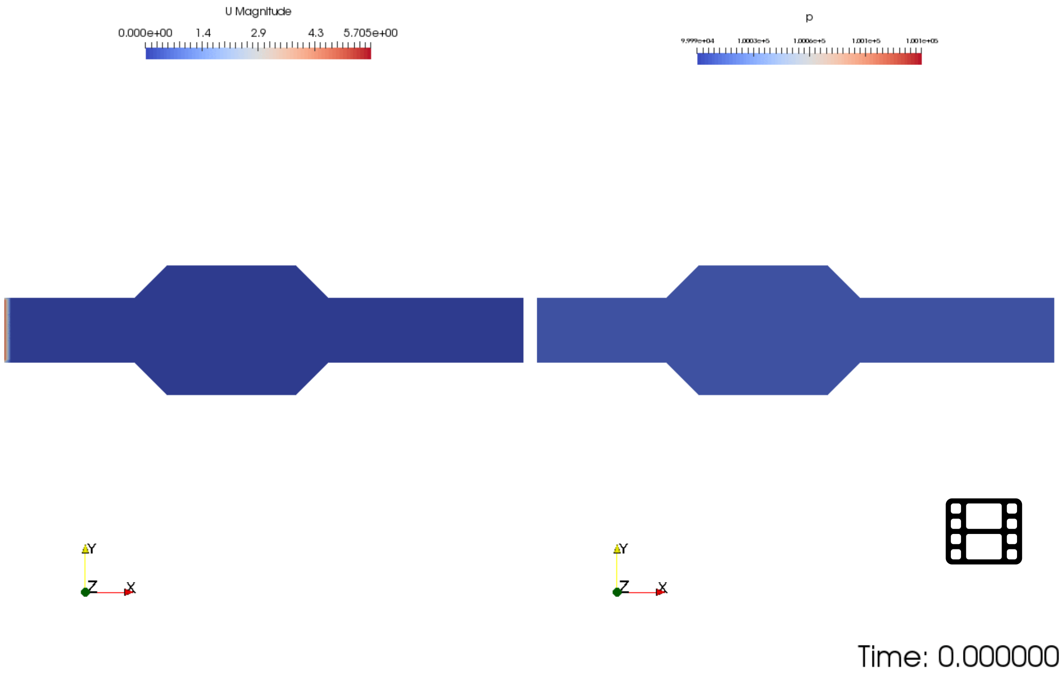
- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Source terms hands-on tutorials

Filter source term

- In this case we are going to use the source term **explicitPorositySource**.
- Using this source term we can apply a porous region (e.g., Darcy-Forchheimer) in the cell selection.
- The source term is activated after 2 seconds of simulation time.





http://www.wolfdynamics.com/training/sourceterms/image1.gif

Source terms hands-on tutorials

Filter source term

system/fvOptions

```
17  filter1
18  {
19      type   explicitPorositySource;
20      active          yes;
22      explicitPorositySourceCoeffs
23      {
25          timeStart       2;
26          duration        5;
28          selectionMode   cellZone;
29          cellZone        filter;
31          type            DarcyForchheimer;

38          DarcyForchheimerCoeffs
39          {
41              d    (5000000 5000000 5000000);
44              f    (0  0  0);
46              coordinateSystem
47              {
49                  type    cartesian;
50                  origin  (0  0  0);
52                  coordinateRotation
53                  {
54                      type    axesRotation;
55                      e1  (1  0  0);
56                      e2  (0  1  0);
57                  }
59              }
60          }
61      }
62  }
```

- The source terms can be selected in the dictionary *fvOptions*, and they can be modified on-the-fly.
- In this case we are using the source term **explicitPorositySource** (line 19).
- Using this source term we can apply a porous region (of the type Darcy-Forchheimer) in the cell selection (line 28).
- The source term is used in a **cellZone** (line 28) named **filter** (line 29), this zone must be created at meshing time or using the utility `topoSet`.
- In lines 38-60, we define the input parameters of the model.
- In this case, the coefficients **f** and **d** are resistance/impermiability coefficients, and **e1** and **e2** are the vectors that are used to specify the porosity.

Source terms hands-on tutorials

Filter source term

system/topoSetDict

```
actions
(
    // filter
    {
        name    filter;

        type    cellSet;

        action  new;

        source  boxToCell;
        sourceInfo
        {
            box (1.5 -10 -10) (2 10 10);
        }
    }
    ...
    ...
    ...
)
```

- To create the **cellZone** used in *fvOptions*, we first create a **cellSet**.
- The set of cells (**cellSet**) is constructed using the utility `topoSet`.
- This utility reads the dictionary *topoSetDict*, which is located in the directory **`system`**.
- Hereafter we show the dictionary inputs used to create the **cellSet** named **filter**.

Name of the selection

Select a set of cells

Create a new selection

Use a box to select the set of cells

Source terms hands-on tutorials

Filter source term

system/topoSetDict

```
actions
(
    ...
    ...
    ...
    {
        name    filter;

        type    cellZoneSet;

        action  new;

        source  setToCellZone;
        sourceInfo
        {
            set filter;
        }
    }
)
```

- To create the **cellZone** used in *fvOptions*, we first create a **cellSet**.
- Now that we have the **cellSet filter**, we can convert it to a **cellZoneSet** that can be used by *fvOptions*.
- The name of the new **cellZoneSet** is **filter**.
- Remember, you can apply the source terms in a **cellSet** or in a **cellZone**.

Name of the selection

Select a zone set (cellZoneSet)

Create a new selection

Convert the cellSet filter to a cellZoneSet named filter

Source terms hands-on tutorials

Filter source term

- At this point, we are ready to run the simulation.
- We will use the solver `pimpleFoam`, which can use source terms.
- Before running the simulation, remember to use the utility `topoSet` to create the filter region used by the source term.
- You can visualize the region using paraview.
- Finally, remember to adjust the numerics according to your physics.
- You will find the instructions of how to run the cases in the file *README.FIRST* located in the case directory.

Roadmap

A crash introduction to:

1. ~~Turbulence modeling in OpenFOAM®~~
2. ~~Multiphase flows modeling in OpenFOAM®~~
3. ~~Compressible flows in OpenFOAM®~~
4. ~~Moving bodies in OpenFOAM®~~
5. ~~Source terms in OpenFOAM®~~
6. **Scalar transport pluggable solver**

Scalar transport pluggable solver

The functionObject scalarTransport

- In addition to all modeling capabilities we have seen so far, you can also add scalar transport equations (or convection-diffusion equation), without the need of modifying the original source code.
- This functionality is provided via **functionObjects**, and it can be seen as a solver that can be plug into another one.
- To setup the scalar transport equation you need to:
  - Define the **functionObject** in the dictionary *`controlDict`*.
  - Add the discretization schemes and linear solvers for the new equations.
  - Define the boundary conditions and initial conditions of the transported scalars.
- You will find the source code of the scalar transport pluggable solver in the directory:
  - **`OpenFOAM-8/src/functionObjects/solvers`**

Scalar transport pluggable solver

The functionObject scalarTransport

- The scalar transport **functionObject** is defined in the dictionary `controlDict`.
- Remember, the input parameters can be modified on-the-fly.

```
scalar1
{
    type   scalarTransport;
    functionObjectLibs ("libsolverFunctionObjects.so");

    enabled true;

    writeControl outputTime;

    log yes;

    nCorr 1;

    D 0;
    //alphaD 0;
    //alphaDt 0;




    field s1;



    //schemesField U;
}
```

Select scalarTransport functionObject

Number of corrector iterations.
It is recommended to do t least one iteration.

Diffusion coefficient.
If turbulent modeling is in use, you can define the laminar diffusion coefficient alphaD and the turbulent diffusion coefficient alphaDt

Name of the new field. You will need to select the discretization schemes and linear solvers for this field. You will also need to define the boundary conditions and initial conditions for this field.

Option to use the same numerical scheme as the one used for the filed U

Scalar transport pluggable solver

Boundary conditions

0/s1

```
dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
  walls
  {
      type              zeroGradient;
  }

  inlet
  {
      type              fixedValue;
      value             uniform 1;
  }

  outlet
  {
      type              inletOutlet;
      inletValue        uniform 0;
      value             uniform 0;
  }
}
```

- Assuming that you named the new scalar s1, you will need to define the boundary conditions and initial conditions for the field s1.
- This is done in the dictionary *0/s1*.
- In this case, the scalar is entering in the patch **inlet** with a value of 1 (this is a concentration therefore it has no dimensions).
- The initial concentration of the scalar is zero.

Scalar transport pluggable solver

Discretization schemes and linear solvers

- Finally, and assuming that you named the new scalar s1, you need to define the discretization schemes and linear solvers of the new equations.
- Remember, this is a bounded quantity, so it is a good idea to use TVD schemes and gradient limiters.

system/fvSchemes

```
gradSchemes
{
    default       Gauss linear;
    grad(s1)      cellLimited Gauss linear 1;
}

divSchemes
{
    default       none;
    div(phi,U)    Gauss linearUpwindV default;
    div(phi,s1)   Gauss vanLeer;
}
```

system/fvSolution

```
s1
{
    solver        smoothSolver;
    smoother      symGaussSeidel;
    tolerance     1e-08;
    relTol        0;
}
```

Scalar transport pluggable solver hands-on tutorials

- Scalar transport in an elbow – Internal geometry
- Let us run this case. Go to the directory:

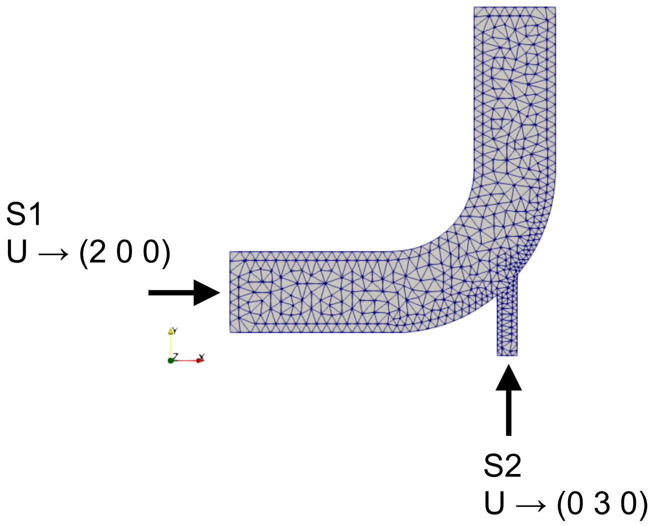
```
$PTOFC/advanced_physics/source_terms/2Delbow_passive_scalar
```

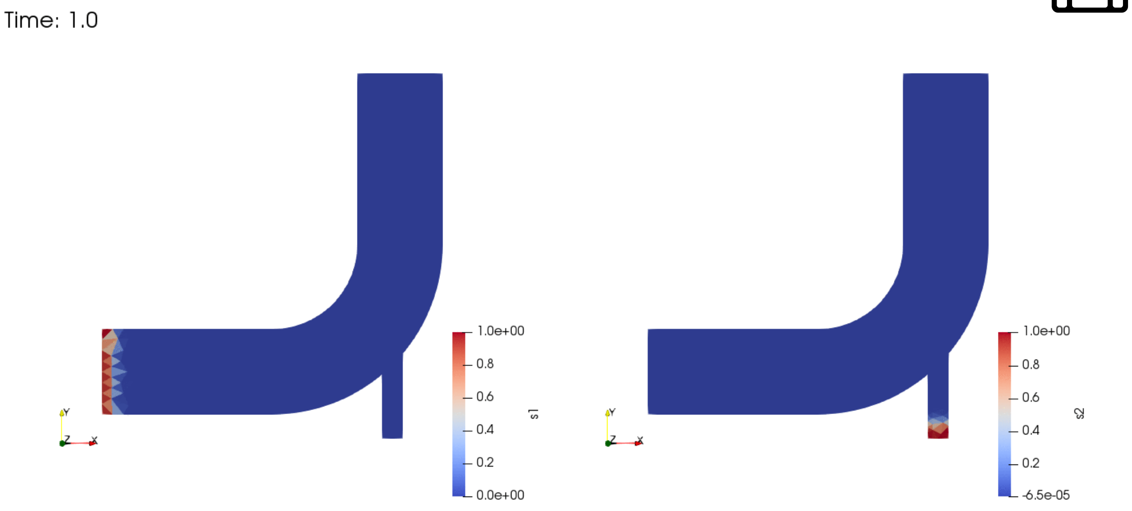
- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Scalar transport pluggable solver hands-on tutorials

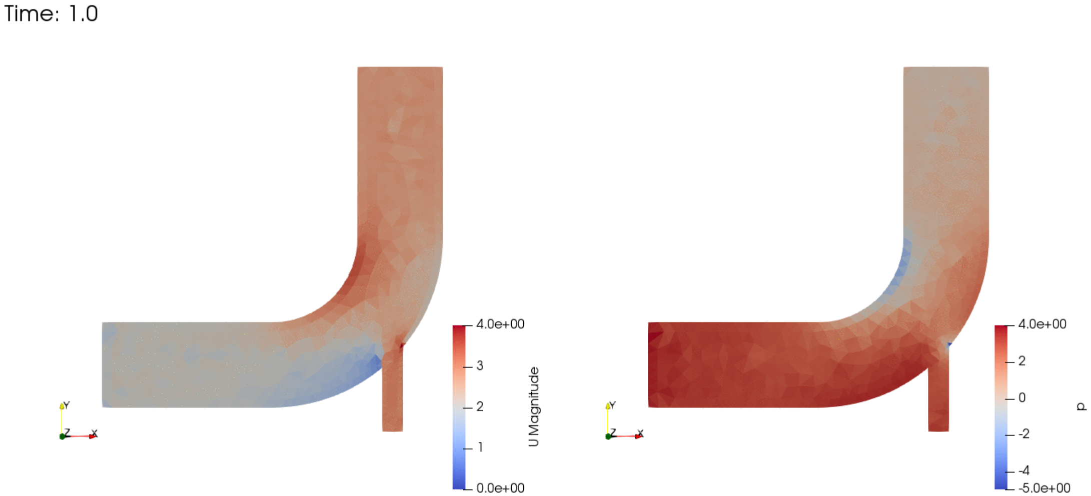
Scalar transport in an elbow – Internal geometry

- Notice that we are adding two scalars, s1 and s2.





http://www.wolfdynamics.com/training/sourceterms/image2.gif



http://www.wolfdynamics.com/training/sourceterms/image3.gif

Scalar transport in an elbow – Internal geometry

- At this point, we are ready to run the simulation.
- We will use the solver `pisoFoam`.
- Remember to adjust the numerics according to your physics.
- Do not forget to create the boundary conditions and initial conditions of the new field variables.
- You will find the instructions of how to run the cases in the file *`README.FIRST`* located in the case directory.

This is the end

- Some kind of conclusion,
  - Good mesh – good results.
  - Start robustly and end with accuracy.
  - Stability, accuracy and boundedness,
    play by these terms and you will succeed.
  - Do not sacrifice accuracy and stability over computing speed.
  - Select wisely the boundary conditions.

This is the end

That was only the tip of the iceberg

Now the rest is on you

This is the end

- We hope you have found this training useful and we hope to see you in one of our advanced training sessions:
    - OpenFOAM® – Multiphase flows
    - OpenFOAM® – Naval applications
    - OpenFOAM® – Turbulence Modeling
    - OpenFOAM® – Compressible flows, heat transfer, and conjugate heat transfer
    - OpenFOAM® – Advanced meshing
    - DAKOTA – Optimization methods and code coupling
    - Python – Programming, data visualization, and exploratory data analysis
    - Python and R – Data science and big data
    - ParaView – Advanced scientific visualization and python scripting
    - And many more available on request
- Besides consulting services, we also offer ‘**Mentoring Days**’ which are days of one-on-one coaching and mentoring on your specific problem.
- For more information, ask your trainer, or visit our website http://www.wolfdynamics.com/

This is the end

TGIF & TGIO

Enjoy OpenFOAM®

el fin

Be collaborative, be innovative, be cloud

wolf dynamics
multiphysics simulations,
optimization & data analytics

Let’s connect

guerrero@wolfdynamics.com

www.wolfdynamics.com