OpenFOAM

A Tutorial with Focus on HPC

by ✉ M. Ohlerich, Dec 8, 2022

Content

License Terms and Conditions....3
License....3
Disclaimer....3
Requirements....3
Part 1 – Bird’s Eye View on OpenFOAM....4
What is it? What is it not?....4
Getting Started – Installation and Testing....5
Installation from Source under Linux (LRZ)....6
Installation of pre-build OpenFOAM on Windows....10
Workflow Example (Motivation)....10
Paraview – Pre-/Post-Processing of OpenFOAM Cases....11
Getting started – Installation and Tutorial....11
Opening OpenFOAM Cases in Paraview....12
Exercise – Paraview Visualization motorBike....13
Part 2 – OpenFOAM Workflow in Detail....14
Structure and Tools of the OpenFOAM Software....14
The essential OpenFOAM Directories....14
Other relevant Directories....16
User Directories....16
OpenFOAM Cases....17
The general Workflow Plan....17
OpenFOAM Case Workflow Example – Step-by-Step with Gmsh....18
Import from other Tools ...ToFoam....26
blockMesh – OpenFOAM Geometry and Meshing Tool....26
Exercise – Kármán Vortex Sheet with blockMesh-generated Mesh....27
STL Files into OpenFOAM – snappyHexMesh, cfMesh....27
Reasonable Practice – Version Control and Backup....28
Own/User-provided or external Solvers and Tools....28
Part 3 – HPC Topics....33
Parallel OpenFOAM....33
When to use Parallelism?....33
Basic Case Setup and Workflow for parallel Runs....34
Exercise – MPI parallel Workflow....35
MPI Scaling – How many parallel Ranks should I use?....35
Advanced File IO....37
GPFS and Workflow Proposals....38
Parallel Reconstruction....39
Parallel Decomposition and Reconstruction....39
Binary, Compressed, and Collated I/O....39
Check-Pointing....40
Function Objects – In-Situ Analysis....41

Selected HPC Topics and Workflows....43
Monitoring, Profiling, Debugging – Automation....43
Parallelism-Resistant Workflows....45
Uncertainty Quantification and Parameter Studies....46
HPC Post-Processing....48
Addendum....55
Useful Links and Literature....55
Bash/Shell/Linux....55
C++....55
CAD....55
Environment Module....55
OpenFOAM....55
Paraview....56
PDEs/CFD....56
Slurm....56
SSH....56
VNC....56
Example Files....57
Gmsh Script of the Kármán Vortex Sheet Example....57
Kármán constant/polyMesh/boundary....57
Kármán 0/p....58
Kármán 0/U....59
Kármán system/controlDict....60
Kármán blockMeshDict....60
Kármán blockMeshDict advanced....62
Swift-Hohenberg: createFields.H....64
Swift-Hohenberg: mySolver.C....65
Swift-Hohenberg: system/blockMeshDict....66
Swift-Hohenberg: constant/transportProperties....67
Swift-Hohenberg: 0/U....67
Swift-Hohenberg: 0/psi....68
Swift-Hohenberg: system/controlDict....69
Swift-Hohenberg: system/fvSchemes and system/fvSolution....69

License Terms and Conditions

License

This document is published under a Creative Commons License (CC BY-SA 4.0).

Disclaimer

This tutorial is not approved or endorsed by ESI Group or ESI-OpenCFD®, the producer of the OpenFOAM® software and owner of the OpenFOAM® trademark. This tutorial is not approved or endorsed by Kitware Inc.®, the producer of the Paraview® software and owner of the Paraview® trademark.

The major purpose of this tutorial is to pave and ease the way to using OpenFOAM on HPC systems. It is designed as self-study tutorial. But if you really have questions you cannot figure out by using Google, or asking colleagues, then you can write me an e-mail (address is linked to the author above).

Requirements

You should be familiar with the following topics and tools, in order to follow this tutorial's lines and OpenFOAM's workflows. But of course you might not need all of the tutorials content.

- Mathematics/Physics: PDEs, CFD (to understand *Why at all?* you use OpenFOAM)
- Linux (Bash/Coreutils – for essentially all OpenFOAM workflows)
- Environmental modules (LRZ HPC specific)
- SSH (keys, tunnel – for HPC workflows)
- Slurm (basic usage – for HPC workflows)
- MPI (basic usage – for HPC workflows)
- C++ (for solver/tool development in OpenFOAM (for building OpenFOAM[1]))

We do not put any efforts here into teaching any of these prerequisites, and simply assume that you know them, or are willing to learn them on the way (definitely the more difficult, but shorter way – just learn for the moment, what you really need).

Some links to web pages and books for an introduction to these topics are included in the Addendum: Useful Links and Literature. We hope they are helpful.

1 At least recommended for the case you face compilation errors. This includes also knowledge about the build tool chain.

Part 1 – Bird's Eye View on OpenFOAM

What is it? What is it not?

In a sentence: OpenFOAM® is a free (open source) C++ framework for manipulating fields on meshes, which includes (as most interesting part) the solution of partial differential equations (PDEs) using the finite volume method. This includes many CFD solvers/workflows. Is however not limited to them.

What it makes interesting for most users is that it does not include license costs. Furthermore, it is designed to be adaptable to larger scale HPC-grade (*High Performance Computing*) use cases by employing MPI (*Message Passing Interface*) parallelism.

Still, there is a price to pay: Time! The entrance level to learning the usage of OpenFOAM is quite high, due to the complexity of the OpenFOAM concept. There is no GUI, primarily. Although such solutions can be found on the Internet (e.g. Helyx OS), the primary workflow still involves basically ASCII file handling, and command-line interaction.

This can however also be considered as a strength, because GUI workflows are often not easy to automate.

As C++ framework, OpenFOAM is highly extensible. Writing software in C++ is but not amenable for everyone. And learning C++ class hierarchies and APIs definitely not so, too. Even less are users probably interested in things like software project management and coding style.

Fortunately, many users may use OpenFOAM just as a ready-to-use toolbox without the need to write own solvers (the run-time tools). But this does not make the business necessarily easier. Unless OpenFOAM is already provided on a system, the OpenFOAM user has to install it for him-/herself. That is usually quite a tedious procedure, although some efforts were undertaken to simplify this procedure. So, even if you do not need to write own solvers, you need to be familiar with the OpenFOAM tools and environments, and even the build system.

Another important issue with OpenFOAM is its schism. Currently, at least three OpenFOAM providers have established on the market. For beginners, it is usually very difficult to decide for the one or the other. And with the time that has passed since the breakup, the OpenFOAM APIs and user interfaces started to deviate. Although ESI tries to prevent this, it is probably impossible in the long run.

So, OpenFOAM is not an easy-to-use-make-me-happy tool. And certainly it is therefore not a tool for everybody.

However, for students of engineering and physics, the openness of the community and the source code are very valuable for learning much about the fields of shell handling, programming, computational science, engineering and science workflows, without much thinking about monetary costs. And despite its complexity and drawbacks, OpenFOAM is used very widely in science, engineering and industry – not the least due to the strong, ambitious and flexible community.

Under the line, the decision for or against OpenFOAM must be made by everyone individually. There are many pros and cons. But that is the case for each piece of software.

In this tutorial, we assume that you decide to go on.

Getting Started – Installation and Testing

This step might prove already the most difficult obstacle for many OpenFOAM beginners. Unfortunately, there is no single installer available. But only some options, we try to summarize.

You must first decide on

1. which Operating System you use:
   Linux, Mac OS X, Windows
2. which OpenFOAM distribution you want to use:
   ESI OpenFOAM, Foundation OpenFOAM, foam-extend
3. prebuild (pure usage ot run-time tools), or compiled from source (own solver; needed feature)

There are maybe more OpenFOAM distributions out there, we don't know of.

And although there are definitely more operating systems, we restrict us to the three officially supported ones. Where however "supported" must be put into perspective. As open source C++ framework, you could build OpenFOAM virtually everywhere where a C++ compiler is available. However, already the build process requires some shell (bash or csh – where we prefer bash). Also the third-party dependencies require the operating system support, what is not always given, e.g. for Windows.

Windows offers the *Windows Subsystem for Linux* (WSL), which essentially virtualizes a Linux (Ubuntu/Debian) operating system. An alternative would be to use VirtualBox (also available for Mac OS X) in order to virtualize a Linux operating system. The price to pay is that you need to learn to administer Linux.

But also environments like Cygwin and MSYS2 are available, which offer a rather complete shell environment under Windows. Still, building OpenFOAM in such an environment might prove difficult (OpenFOAM in Cygwin, OpenFOAM in MinGW).

For the tutorial, we decide to use ESI OpenFOAM v2112, as it provides the widest scope of offer and flexibility. There are also Docker images, Windows pre-builds, Linux packages, etc. available, possibly not available for other OpenFOAM distributions. Many pre-build packages only provide the basic run-time tools for using OpenFOAM. For extension and development, the sources are missing. But please check the download pages! Things may change.

For installation, we show two examples:

- building from source on Linux (reasonable for LRZ cluster systems and Linux desktops and Windows WSL/VirtualBox; and also to show how to use the complete OpenFOAM software)
- using the MinGW prebuild installer (as Windows is mostly used on desktop systems; and for an introduction to OpenFOAM, it may suffice)

Installation from Source under Linux (LRZ)

Although the ESI build from source guide is rather complete, we add some more steps in order to ease the installation procedure for beginners, and to prepare you for the wider use and development of OpenFOAM. If not interested, you can skip this section (at least in a first reading).

The steps are as follows:

1. check the requirements
2. download and unpack the OpenFOAM and Third-Party sources
3. prepare and source the etc/bashrc (setup of the environment)
4. build paraview (optional; needed for in-situ post-processing and catalyst)
5. build other optional third-party packages (optional)
6. build OpenFOAM
7. check the installation

1. Check the Requirements

You need a C/C++ compiler (GCC/Intel/…); bash shell environment (coreutils in many cases); CGAL/MPF/MPC/GMP (foamyQuadMesh etc.); a MPI library and headers (devel-package) if MPI-parallel execution is desired; CMake; maybe more for other third-party packages

2. Download and unpack the OpenFOAM and Third-Party Sources

The documentation pages provide the links for the download. For ESI OpenFOAM v2112, they might look like these (this is changing from time-to-time):

https://dl.openfoam.com/source/v2112/OpenFOAM-v2112.tgz
https://dl.openfoam.com/source/v2112/ThirdParty-v2112.tgz

There has prevailed the habit to install OpenFOAM into **`~/$HOME/OpenFOAM`**. But now, any other folder would be fine as well. After the download, unpack the two tarballs.

```
> wget https://dl.openfoam.com/source/v2112/OpenFOAM-v2112.tgz
> wget https://dl.openfoam.com/source/v2112/ThirdParty-v2112.tgz
> tar xf OpenFOAM-v2112.tgz
> tar xf ThirdParty-v2112.tgz
```

This might take some while. The result are two folders: **`OpenFOAM-v2112`** and **`ThirdParty-v2112`**.

3. Prepare and Source the etc/bashrc

The first thing to do is to open ***OpenFOAM-v2112/etc/bashrc*** with an editor (vi(m), emacs, nano, jedit, … an editor!!) and make necessary changes. The most important once are probably on **`WM_COMPILER`** and **`WM_MPLIB`**, the compiler[2] and MPI library to be used.

2 If you use GCC, the latest versions are maybe too critical about C++. Older versions usually work more smoothly.

Which options are available is documented inside the *bashrc*. For instance, an Intel Compiler (somewhere installed on the system) would require the settings

```
export WM_COMPILER_TYPE=system
export WM_COMPILER=Icc
```

For Intel MPI (as available in the environment like from *module*) only requires[3]

```
export WM_MPLIB=INTELMPI
```

Many more options (e.g. on integer label size, or floating point precision) are available. We recommend to parse once through the *bashrc* in order to get a slight overview.

For users, who want to use c-shell (csh), please use the *cshrc* instead. It looks similar to the *bashrc*. The environment variables are necessarily the same.

The finer grained configuration possibilities for the third-party dependencies can be found in ***etc/config.sh*** (*etc/config.csh*). But before doing any modifications here, try first with the default settings. They are usually quite reasonable.

The last step is to source the *bashrc* (modify your path if different):

```
> source ~/OpenFOAM/OpenFOAM-v2112/etc/bashrc
```

This should not throw any errors. If it does, please read carefully, and try to fix this in the configuration. A general recommendation here is difficult as every system might be different enough to throw very different errors. But this does not happen often, in our experience, with the defaults.[4]

With ESI OpenFOAM, you can now check whether your system is ready for the OpenFOAM installation.

```
> foamSystemCheck

Checking basic system…
-----------------------------------------------------------------
Shell:       bash
Host:        cm2login3
OS:          Linux version 4.12.14-197.78-default

System check: PASS
==================
Can continue to OpenFOAM installation.
```

Again, if there appear errors, try to figure out (in worst cases by means of Google) what the cause might be!

4. Build Paraview (optional)

Now, change to the Third-Party directory (e.g. **`cd ~/OpenFOAM/ThirdParty-v2112`**; but

```
> cd $WM_THIRD_PARTY_DIR
```

should definitely work after sourcing the *bashrc*), and try to build *paraview*. There is a build script to support your efforts – **`makeParaview`**. It has quite some options, as you can see from

3 **`MPI_ROOT`** needs to be set to an external/system MPI.
4 If you think that you messed the shell, just restart from a fresh state.

```
> ./makeParaView -help
```

Depending on what one needs, one must provide some of the dependencies (Qt, Python, MPI, …). If not available on the system, you would need to build them on your own, and make them available to your *paraview* installation script (usually as some environment settings).

```
> ./makeParaview -python3 -mpi=0 -no-qt
```

was usually successful. But it might be necessary to play around and find a working mode.

5. Build other optional Third-Party Packages (optional)

The folder **$WM_THIRD_PARTY_DIR** contains a lot of **make…** scripts. *Qt*, *MPICH*/*MVAPICH*/*OPENMPI*, *PETSC*, *KAHIP*/*SCOTCH*/*METIS*, *VTK*, … essentially almost all third-party dependency could be build by hand. Even the GCC compiler could be build and used. If you succeed in doing so, you can just tell OpenFOAM in the *config.sh* directory that you use the packages NOT system-sided but from the Third-Party.

However, only few things are *really* needed. Even *paraview* is not really necessary. And as we build it here, it is sort of *amputated*, as we built it without GUI. We will show later, why that's not so important. And as only *paraview* would require Qt, we can also dispense with building Qt.

If you provide a MPI with your system, you also don't need to build any of the MPI variants here. Just set the **WM_MPLIB** to the one you provide, e.g. **SYSTEMOPENMPI** (and the **MPI_ROOT** environment variable).

KAHIP, *SCOTCH* and *METIS* are graph partitioners, required to decompose the mesh of a case into several disjunctive pieces for MPI parallel execution. *SCOTCH* is automatically provided and build with OpenFOAM. If you want *METIS*, too, you must download the sources and unpack them in **$WM_THIRD_PARTY_DIR**, and run **./makeMETIS**. But if you don't want or can't use MPI, the partitioners are irrelevant, too.

Adios2 is a file format. We never used it.

CGAL is a tool used by **foamyQuadMesh** for meshing with hexahedral grid cells (cubes). Usually, **snappyHexMesh** and **cfMesh** are more powerful, easier to use, and don't require CGAL.

FFTW/PETSC are maybe interesting sometimes if you can use them for the linear system solvers or pre-conditioners to accelerate the numerical integration.

Whenever successful, the Third-Party packages are installed under
$WM_THIRD_PARTY_DIR/platforms/${WM_ARCH}${WM_COMPILER}
for the configuration independent packages, and under
$WM_THIRD_PARTY_DIR/platforms/${WM_ARCH}${WM_COMPILER}${WM_PRECISION_OPTION}${WM_LABEL_OPTION}
for those packages (e.g. *SCOTCH*/*METIS*) that are sensitive to the floating point precision, and the label size.

This has some advantage if you need two or more different configurations in parallel. *OpenFOAM was designed with maximum flexibility in mind.*

Guru's remark: Sometimes, it is good to know this structure, if you for instance can build Third-Party packages just with one compiler (e.g. GCC), but want to use OpenFOAM with another one (e.g. Intel). Then building the Third-Party package with GCC, and afterwards renaming the directory inside the *platforms* directory is quite often successful.

6. Build OpenFOAM

The more or less final step is now to install OpenFOAM itself. So, go back to the OpenFOAM directory and start the compilation, after a **wmRefresh**.

```
> wmRefresh                        # after paraview built successfully
> foam                             # is an alias created with sourcing the bashrc
> WM_NCOMPPROCS=4 ./Allwmake
```

foam is an alias to **cd $WM_PROJECT_DIR**.

This now takes again some time … depending on how many parallel processors you have available. Just specify them via **WM_NCOMPPROCS**.

The documentation page says here the following: "*Run this command as often as necessary until no errors appear anymore.*" This is it what makes it so hard to trust OpenFOAM. ☹

But what is meant by that is that you can iteratively and incrementally build the OpenFOAM tools. On errors, your don't need to start again from the beginning. Just go on with what you already have. This also means that you can essentially build only what you really need.
(Often here appears a question like "*I got a solver from my supervisor. How can I build it with OpenFOAM?*". We will handle this in Own/User-provided or external Solvers and Tools.)

7. Check the Installation

The very last step is to check the installation. Again, ESI provides a nice script **foamInstallationTest**. It carefully separates also critical and non-critical deficiencies. You should at least see something like

```
Summary
-------------------------------------------------------------
Base configuration ok.
Critical systems ok.

Done
```

at the end. If not, scrutinize the reason for the problem. Not always is it devastating, and can sometimes even be ignored.

It is also worth to look into **$WM_PROJECT_DIR/platforms/${WM_ARCH}${WM_COMPILER}${WM_PRECISION_OPTION}${WM_LABEL_OPTION}${WM_COMPILE_OPTION}/{bin,lib}**. Here, all the libraries and solvers built can be found be found. On a complete installation, the number of binaries and libraries should not be too small. At least, you can check whether your desired solver is available. And if not, check why not!

A final, really good test is to use one of the ***tutorial cases*** – preferably one that uses your desired solver – which are provided with OpenFOAM. We will exercise this in the next chapter, and so show by example how an OpenFOAM workflow may look like.

Installation of pre-build OpenFOAM on Windows

On the ESI-OpenFOAM web page, there is a menu item "*Download*". Clicking here on Windows, you find some options on how to use OpenFOAM on Windows. A very fast way to get OpenFOAM into operation is to go to "*Native-windows*", and download the installer.

This installer does not require any administrative rights. It just unpacks the MSYS2 environment with the installed OpenFOAM software in a folder of your choice (you can create one). Only exception is that MSYS2 does not like white spaces in its path names.

One also needs to install MSMPI. This unfortunately, requires admin rights. At least the license condition of this product are rather acceptable (you usually don't need to pay).

But otherwise, that's it. A desktop icon is created, which opens a MSYS2 window, and you can start. We recommend to create a desktop link to the HOME or OpenFOAM directory of the MSYS2 installation (**`<MSYS2-install-path>\home\ofuser`**) in order to find it easier when opening the cases with *paraview*.

For this pre-build OpenFOAM package, the **`etc/bashrc`** does not need to be *source*d. Check the installation quickly by executing

```
> blockMesh -help
```

The output should not contain error messages.

Workflow Example (Motivation)

For a thorough testing, OpenFOAM comes with an extensive *tutorial* suite. But already to disillusion you … it is not really a guided tour as you may expect from the name *tutorial*. It is more a collection of application cases, to found in **`$FOAM_TUTORIALS`**, which should correctly run when the OpenFOAM installation was successful. They are often small enough for educational and test purposes. But some can also be used on bigger systems to test parallelism.

```
> tut                              or      > cd $FOAM_TUTORIALS
```

do actually the same thing[5]. Switching to the *tutorial* directory.

It is usually recommended to copy the *tutorial* cases to a different location

```
> cp -r $FOAM_TUTORIALS ~/OpenFOAM/
> cd ~/OpenFOAM/tutorials
```

In this way, you can change anything you want, without losing the original as reference. For a somewhat more descriptive overview of the available tutorial cases, please look here or here.

1. A serial Case – icoFoam

`icoFoam` is an incompressible laminar solver. Its first test case is the lid-driven cavity[6] – very simple and serial (not parallel). For the test, just issue

```
> cd ~/OpenFOAM/tutorials/incompressible/icoFoam/cavity
> ls cavity
```

5 The alias **`tut`** is maybe not everywhere defined.

6 From the content of this web page, you may guess why it's called *tutorial*. We recommend you to familiarize with it. We really can't give an improvement on it. That's why this present tutorial is kept so short.

```
0  constant  system
> ./Allrun
> cd cavity
> ls
0  0.1  0.2  0.3  0.4  0.5  constant  log.blockMesh  log.icoFoam
system
```

`Allrun` just automates the workflow (and creates automatically log-files for the solvers used). We could achieve the same as above by executing

```
> cd ~/OpenFOAM/tutorials/incompressible/icoFoam/cavity/cavity
> blockMesh > log.blockMesh
> icoFoam > log.icoFoam
```

This may already give you some feeling, how OpenFOAM operates with tools and solvers in a case directory.

Important Information: All OpenFOAM tools have a `-help` option for an overview of command-line options!

2. A parallel Case – simpleFoam

Another case, which however is already a little bit more complicated, and requires MPI for execution, is the *motorBike tutorial* case.

```
> cd ~/OpenFOAM/tutorials/incompressible/simpleFoam/motorBike
> ./Allrun                  # Take care to have at least 6 processors available!
```

Again ideally, there should be created new folders and log-files, where none of the latter should contain errors. Even simpler is it to assess problems, when the case runs only for seconds. The log-files hopefully give you some hint about a reason.

This seems rather straightforward. And for the tutorial cases, it mostly is. The real application cases you want to handle later, however, will usually require more work!

Paraview – Pre-/Post-Processing of OpenFOAM Cases

Getting started – Installation and Tutorial

Producing simulation data using solvers is one aspect of CFD simulations. But pre- and post-processing is by no means less important. Early in the development of OpenFOAM, *paraview* was selected as the tool of choice. It is even provided with the Third-Party package.

Nowadays, however, even the OpenFOAM educators discourage the usage of the self-build Third-Party *paraview* – except for the run-time post-processing. Specifically, because it becomes more and more difficult to correctly build *paraview* with all its features.

Specifically, there is a much simpler way to deploy *paraview*. Provided by the *paraview* developers themselves. Going to the Download page of Paraview, you can select the *paraview* version you want, the operating system (under the *version selection menu* on the right), and simply download the desired pre-build package. Also for Windows, you can find packages that just need to be

unpacked (unzipped), and are ready for use. Just go to the ***bin*** directory, and start **`paraview`**.[7] Starting it opens the *paraview* GUI.

Also on that download page, you find a ***tutorial*** (a PDF) under *Documentation*. *Paraview* itself is a very advanced data analysis tool. Teaching it fills complete courses on its own. ***We can only highly recommend to pass through the tutorial at least ones!*** You will learn the most important essentials of data analysis and visualization using *paraview*! Also Youtube offers a lot of courses, tutorials, and hands-ons. Please look into the Useful Links and Literature section.
We will show later in this tutorial only the HPC-relevant workflows and configurations.

Opening OpenFOAM Cases in Paraview

The next step is actually now trivial. Go for instance to one of the *tutorial* cases (*motorBike*), and create a file with the ending "*.foam*". The name really does not matter! And the file does not require any content!

```
> cd ~/OpenFOAM/tutorials/incompressible/simpleFoam/motorBike
> touch bla.foam
```

Now, from within the *paraview* GUI, open this file!

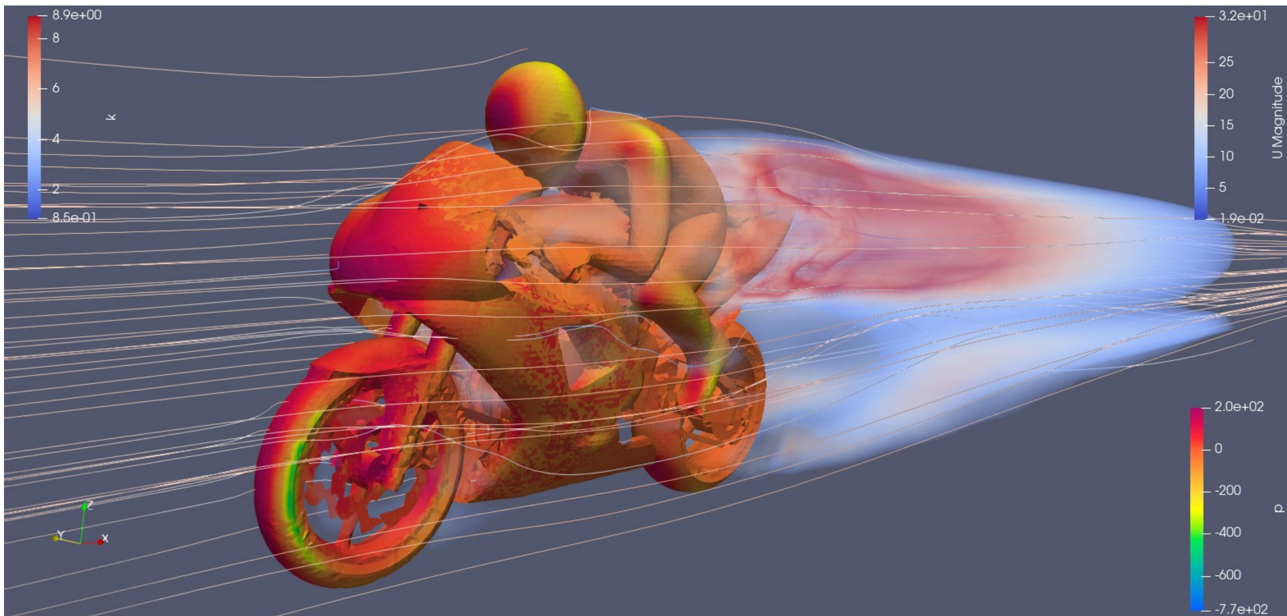
As you are hopefully already a bit familiar with *paraview* now, you see that the *Properties* tab gives you some options. Among others the *Label Size* and the *Scalar Size*. These must agree with the values of **`$WM_LABEL_SIZE`** and **`$WM_COMPILER_LIB_ARCH`**, respectively, for which you compiled OpenFOAM.

---

7 The OSMesa build does not contain the *paraview* GUI. It is used for off-screen rendering in HPC, for instance. For Windows, you may start with the non-MPI version. The GUI should ***never*** be started in parallel anyway!

Exercise – Paraview Visualization motorBike

With some practice and patience, you may reproduce the following picture easily.



As a hint: *Extract Block* (pressure, *p*), *Iso Volume* (turbulent kinetic energy, *k*), and *Stream Tracer* (flow velocity, *U* magnitude) filters were used, all descending from the original input data. The turbulent kinetic energy was rendered then with *Volume Rendering*.[8]

8 Depending on GPU hardware and driver available, your result might look differently.

Part 2 – OpenFOAM Workflow in Detail

For the for the following, we assume that you have a complete installation available. As already mentioned earlier, the packaged installation usually contains less, and usually does not allow for user-provided extensions.

Again, we align the explanations here to the ESI OpenFOAM version v2112. In other distributions it is maybe similar, or maybe not. But understanding at least one OpenFOAM architecture usually suffices to get on with the others (or the older versions of ESI OpenFOAM).

Structure and Tools of the OpenFOAM Software

The essential OpenFOAM Directories

When unpacking the **`OpenFOAM-*.tgz`**, you find some files and directories in the top-level directory. Let us look first on the directories:

```
applications  bin  doc  etc  modules  src  tutorials  wmake
```

The ***etc*** directory is probably the first one gets in contact with. It contains the ***bashrc*** and ***config.sh/*** directory (and those for *csh*, respectively) for configuring the complete OpenFOAM run-time and build environment. Usually, it is mandatory to *source* the *bashrc* before using OpenFOAM – either for running its tools and solvers, or for build it or external/own components. In order to make this environment persistent, you can add to you ***~/.bashrc*** file

```
source ~/OpenFOAM/OpenFOAM-v2112/OpenFOAM-v2112/etc/bashrc
```

where you must change the path to the *bashrc* when you have a deviating OpenFOAM installation location.

On the LRZ clusters, LRZ-provided OpenFOAM environments can be conveniently set via the module system:

```
> module av openfoam
  […]
> module load openfoam/2006-gcc8-impi-i32
```

for instance (versions and names may change).

The ***etc*** directory but also contains a ***controlDict*** file, via which you can set global run-time flags and switches, for instance concerning output verbosity, or optimization and debugging settings. These can be overwritten by case-local *controlDict* files, individually. This global *controlDict* file can serve as a reference then.

This optimization and debugging is sort of “OpenFOAM internal”. As you saw maybe already in the *bashrc*, the **`$WM_COMPILE_OPTION`** environment variable sets the compile flags for optimization (**`Opt`**), debugging (**`Debug`**), or profiling (**`Prof`**). We call this *external* optimization or debugging. Usually, you will use only the **`Opt`** build. And LRZ provides only those. **`Debug`** is mostly only for OpenFOAM developers facing a problem with implementations deep inside OpenFOAM, where external debuggers are needed. **`Prof`** is used if you assume a severe performance problem that you hope to get analyzed using an external *profiler*.

However, for many purposes, the *internal optimization* and *debugging* flags are already sufficient to tune the solvers, and investigate occurring problems, because most of these problems are produced by the OpenFOAM user him/herself.

The other folders and files are not so interesting most of the time. If they become, you will know it, and understand them then. For instance, *caseDicts* and *codeTemplates* provide references/templates for dictionaries and solver or tool codes, respectively.

In short, *etc* contains central settings, configurations, and information.

The settings in the *bashrc* concerning compilers and compiler flags used are mapped to the content of the ***wmake*** directory. This directory contains all the ingredients of OpenFOAM's build system. Most of the time, you will ignore it completely. But if you want to extend your compile flags, ***wmake/rules*** is some place to do it centrally.

Sometimes (hopefully rarely), you face some strange errors in association with the *wm*-tools themselves (**`wmake`**, **`wdep`**, …). In the past at least, OpenFOAM built them at the first call of ***Allwmake***, when building OpenFOAM. And with some compiler (or changing the compiler), this went wrong sometimes.

Otherwise, **`wmake`** is the central tool of OpenFOAM to build user-provided tools and solvers.

applications, ***src***, and ***modules*** contain the source code of OpenFOAM. ***src*** contains the central OpenFOAM library parts, including the libraries for meshes, time integration, and parallel execution. ***applications*** contains tools, utilities and solvers, something you probably will mostly work with. These tools actually are built against the central libraries of ***src***. Therefore, if you want to build own solvers, the content of ***applications*** will probably give you enough of good examples on how to use the OpenFOAM API.

modules contains the source code of some external utility packages like ***cfMesh***. They are not really part of OpenFOAM (in the sense that their developers are maybe different people than the OpenFOAM developers). But with some agreement, the code was included in OpenFOAM's source package, which warrants us at least some sort of good integration, and test.

There is also a ***bin*** folder, which contains some tools and utilities. But those are mostly bash scripts. They don't need to be built. **`foamSystemCheck`** is such a tool. Other convenience scripts are **`foamNew`**. If you want to know what they do, just look into them with an editor of your choice, and read the description in their headers. Much of that stuff could most probably be ignored and realized in a different way. But hey! Why reinventing the wheel?!

tutorials, we had already. The *tutorial* cases follow several purposes. On the one hand, they serve as educational and training samples for beginners. Usually, setting up a new case is probably accomplished by just copying an existing one, which is then modified and adapted to the own needs. This saves definitely a lot of time. And you don't need to memorize all the details that accompany the creation of a new OpenFOAM case.

On the other hand, as we outlined in the previous part already, the *tutorial* cases also serve as test-bed for the OpenFOAM installation itself. If they do not work, you definitely have a problem in your OpenFOAM installation. (This does unfortunately not mean that a solution of the problem is easier to achieve.)

Last but not least, ***doc*** contains the documentation. This is essentially the Doxygen generated documentation, which you can find also online (see Useful Links and Literature). As a rule of thumb for help is – help yourself (first). Meant by that is that OpenFOAM provides the complete code base (*open source*) with a lot of inline documentation. And although it is be no means complete, yet, the efforts of improving this situation are very well visible. For instance, you can now find for many solvers a human-readable description about which equations are solved, and which numerical method is used (e.g. incompressible fluid with PISO). If you want, you are kindly invited to improve or contribute to the documentation.

In any case, if you know to read C++ code, you have everything in your hands. No secrets.

Other relevant Directories

Sometimes, there is an explicit ***build*** directory. It is used only for building the libraries and executables from source out-of-source. It could be removed after a successful build, in principle, in order to same some space.

The other probably more important directory is ***platforms***. It contains different sub-directories for each build-configuration you built OpenFOAM with. Different compilers, 32/64 bit scalars, 32/64 bit labels, Opt/Debug/Prof, … . In each of these folders is a ***bin*** and a ***lib*** directory, where ***bin*** contains the binary executables (tools, solvers, utilities, …), and ***lib*** the libraries.[9] The *bashrc* directs **PATH** and **LD_LIBRARY_PATH** (among others) to these paths such that the binaries and libraries are correctly found during run-time.

On a first gaze, it might be a lot of waste of hard-disk space to have many different installations in parallel. But thinking twice about it, this reveals also a rather large flexibility for developers. And that's most probably the reason for this design. OpenFOAM was never meant as *use-only* program package.

For the case that you vote for such a flexible development option, we recommend to generate different *bashrc* files (with some suitable labeling), one for each configuration – although the aliases set by it would allow for fast switch between e.g. SP and DP (single precision = 32 bit scalar, double precision = 64 bit scalar).[10]

User Directories

If you create own libraries and solvers, you usually shouldn't put them into OpenFOAM's source tree. That's also not necessary. And on central installations like those of the LRZ clusters even not possible, as you won't get permissions to create directories and files in the central installation folder.

You can then put your sources in an arbitrary directory of your choice, where you have permissions, and build the libraries and/or executables from there. OpenFOAM will place – on successful built – the resulting binaries and libraries in the directories specified by **$FOAM_USER_APPBIN** and **$FOAM_USER_LIBBIN** (and possibly some others …. Sigh!), respectively. Go sure that these are set correctly! E.g. via

```
> printenv | grep FOAM_USER
```

9 Under Windows, that's a bit different, as Windows addresses library search paths also with **%PATH%**.
10 Only take care with **wmRefresh**! It per default reloads a file called ***bashrc***!

Please, also check whether they exist. Actually, most of this should be done automatically. But you know Murphy's Laws: "*Everything what can go wrong, **will** go wrong!*" and "*Complex system produce complex errors!*". Well, OpenFOAM *is very* complex!

Also take care that **`$FOAM_USER_APPBIN`** is in the **`PATH`**, and **`$FOAM_USER_LIBBIN`** in the **`LD_LIBRARY_PATH`**! To check this, just issue

```
> echo $PATH | grep $FOAM_USER_APPBIN
> echo $LD_LIBRARY_PATH | grep $FOAM_USER_LIBBIN
```

Adding, if not present, is done by

```
> export PATH=$FOAM_USER_APP:$PATH
> export LD_LIBRARY_PATH=$FOAM_USER_LIBBIN:$LD_LIBRARY_PATH
```

To make this permanent, just add these lines at the end of OpenFOAM's **`bashrc`**. But actually, ***this should not be necessary, as the bashrc already should do it correctly!***

OpenFOAM Cases

The general Workflow Plan

Let us now come to the usual use case of OpenFOAM: Solving some *partial differential equation* (*PDE*) with some *initial conditions* (*IC*) and some *boundary conditions* (*BC*).

The steps all the use cases have in common (btw. for any numerical tool that solves numerically/computationally PDEs) are:

- create a *geometry*; define boundaries and volumes (internal space)[11]
- create *initial conditions* and *boundary conditions*
- create a *mesh* for the boundaries and volumes (spatial discretization needs to be adapted to BC and physical solution)

These steps represent the so-called *pre-processing*. Some of the steps need to be repeated several times (like the meshing), until a satisfying result is achieved. Sometimes even after the following steps maybe.

- define and configure the numerical solver scheme; set the time step size;
- optional: set write out periods (I/O); set online ("in-situ") analysis steps, e.g. for monitoring
- run the simulation; monitor the progress

After a successful simulation, you want to analyze/visualize the resulting data. That's what is called *post-processing*.

However, as might have been noticed, the frontiers between these steps become more and more fuzzy, specifically the more you approach more elaborate HPC-scale use cases. For instance, saving all the data for a separate and independent post-processing might be too expensive in terms of storage. Maybe, it is also not necessary. When you know already the questions to your numerical

11 1D, 2D, 3D, nD *volume* doesn't matter. We use the term *volume* (and also *boundary*) here in a more abstract sense. It's of course relevant for the system of equations you want to solve which prescribes the space on which it operates.

simulation beforehand, you can write out from in-situ analysis steps exactly only those required data. A better compression you probably can't get.

But, of course, there is also a trade-off between getting results on success, and need for information for debugging on failure or problems. We will get back to that later when looking at the HPC-relevant aspects.

OpenFOAM Case Workflow Example – Step-by-Step with Gmsh

According to the above guideline, we try to fill each of the steps with essence, and include all relevant aspects helpful for setting up own cases. Please feel free to consult in parallel also the OpenFOAM User Guide.

1. Preparation

Before you start OpenFOAM, you should have developed already a very good imagination of what you are going to investigate. This is actually true for any kind of simulation. Numerical simulation are used if you run out of better options! Try to collect as many information and questions you can get beforehand!

- Make hand-drawings of your system! Figure out what level of detail you know already and you really need! (geometry, boundaries, physical quantities, …)
- Pose questions!
  What do you want to figure out? Do you have already a guess? Are there analytical answers already – even if only approximate? Is there experimental data material? If so, in which form do I want to compare it afterwards? …
  All this will help you later to set up the simulation case, and avoids more work than necessary.

Example:
Let us, for instance, investigate a *Kármán vortex sheet*. It is a 2D arrangement, which can be setup as a long channel, and a circular obstacle within the flow.

The fluid can be incompressible. So, we have two (three) velocity components, $\vec{u}$, and a pressure, p (density ρ is assumed constant), meaning we need three (four) equations – continuity and Navier-Stokes equations. Gravity also does not matter here.

$$\partial_t \vec{u} + (\vec{u} \cdot \nabla) \vec{u} = -(1/\rho) \nabla p + (\mu/\rho) \nabla^2 \vec{u} \quad ; \quad \nabla \cdot \vec{u} = 0$$

or, in dimensionless form,

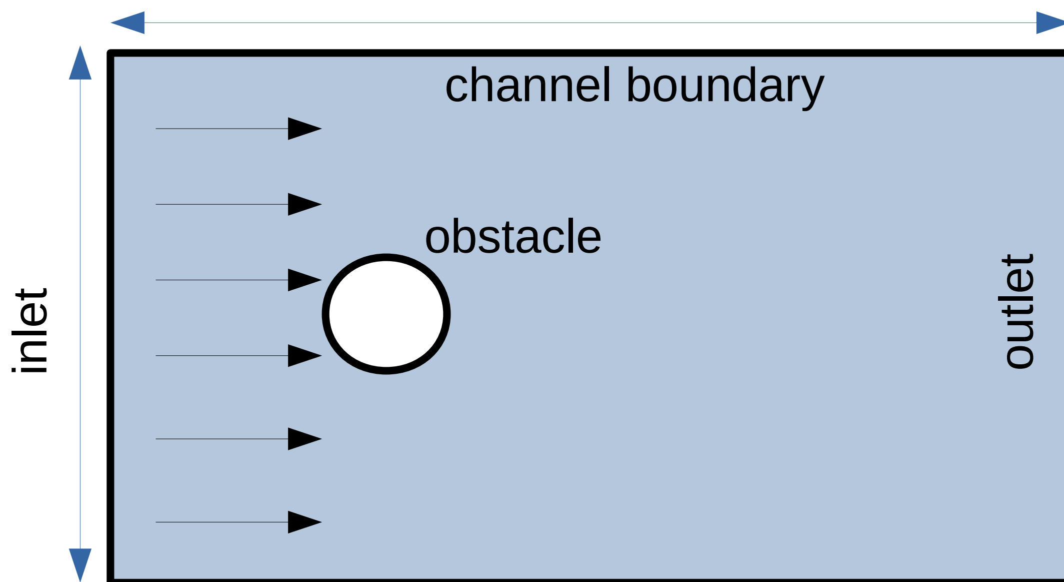
$$\partial_\tau \vec{U} + (\vec{U} \cdot \tilde{\nabla}) \vec{U} = -\tilde{\nabla} P + (1/\mathrm{Re}) \tilde{\nabla}^2 \vec{U} \quad ; \quad \tilde{\nabla} \cdot \vec{U} = 0$$

with the Reynolds number

$$\mathrm{Re} = \frac{L \rho u_\infty}{\mu} \quad .$$

We can assume a uniform constant flow speed u_∞ at the channel inlet (and actually also at the outlet – but we will use here a different boundary condition). About the pressure, we can't say much, yet. Let us see later, what OpenFOAM offers.

L is a *characteristic* length scale. We will take the diameter of the circular obstacle as the reference length. The vortex sheet appears above about $Re \geq Re_{crit} \approx 90$.



So, what about the dimensions of our geometry? Of what width and length should the channel be with respect to the obstacle diameter? How should the mesh look like? Which OpenFOAM solver should we use? Which boundary and initial conditions should we set? Can/should we apply symmetries?

Which ***solver*** to use, is maybe simple. Looking at the standard solver page of ESI OpenFOAM lets us find *"Transient solver for incompressible, laminar flow of Newtonian fluids"*. That's probably the right one as we expect transient (time-dependent) behavior. And a turbulence model is not required here, because the vortex dissipation is realized by transport them out of the volume. So, we will try ***icoFoam***.

Boundaries are the obstacle itself, the inlet and outlet, and the channel boundaries.

- On the ***obstacle***, we want *no-slip* boundary conditions, meaning ***velocity is zero***. For the ***pressure***, we chose ***zeroGradient***. We will see later, which more options are available.
- At the ***inlet***, the ***velocity is constant***, u_∞ , the value serving as *control parameter*. We could set the ***pressure*** here to ***zero*** arbitrarily – we just need some arbitrary reference pressure, because the Navier-Stokes equations only depend on the gradient of the pressure. But it is usually not a good idea to fix velocity and pressure at the same boundary. So, ***zeroGradient*** is probably better.
- At the ***outlet***, we chose some ***outflow boundary condition***. ***zeroGradient*** for the ***velocity***. And ***zero pressure***.
- At the ***channel boundaries***, we only need to keep consistency with the equations and the other boundary conditions. ***Periodic*** or ***constant velocity*** or ***slip*** boundary conditions are possible, because this boundary should actually not matter for the vortex sheet – i.e. this boundary should be too far away from transient places.

This brings us back to the ***dimensions*** of the geometry now. The obstacle diameter is arbitrarily at *1 LU* (length unit).[12] We could set the channel width then to *10 LU*, so the channel boundary is about *4.5 LU* from the obstacle boundary away. Before the obstacle, we need some place for

12 We will see later that OpenFOAM allows to set some dimension units. So, we could say *1 meter*, here.

arrangement of the flow pattern. *4.5 LU* should be reasonable, again. Behind the obstacle, we want to see the Kármán vortex sheet. If we set the channel length to *20 LU*, and assume the wave length of the vortex sheet to be also around 1 LU, we should see about 20 vortexes (depending on the inlet flow speed).

2. Geometry Generation and Meshing

To some extent, this cannot be always split into different steps, as you will see in a moment.

Actually, there is quite some flexibility in OpenFOAM, to provide geometries and even complete meshes. We cannot be comprehensive here at all. And users can even extend these capabilities according to their needs and skills. For some more extended description of the OpenFOAM meshes, please consult the OpenFOAM Documentation.

We demonstrate here one way using ***Gmsh***. Gmsh is a free, open source tool for creating and viewing geometries and meshes, and actually much more (it has some interface to solve PDEs – so, it is some sort of a concurrent of OpenFOAM).

Gmsh can be simply downloaded from the Gmsh web page for Windows, Linux, or Mac OS X. The package can be unpacked and the ***gmsh***[13] executable simply started. A window opens, which visualizes the geometry and mesh.

In Linux, it looks rather like this (please check for newer versions – Gmsh is still actively extended).

```
> wget https://gmsh.info/bin/Linux/gmsh-4.9.5-Linux64.tgz
> tar xf gmsh-4.9.5-Linux64.tgz
> export PATH=$PATH:$PWD/gmsh-4.9.5-Linux64/bin
> gmsh --help
> gmsh                                # in order to start the GUI
```

Gmsh comes with a very well readable ASCII input file format, which is much easier to handle in an editor. The GUI is usually used only to visualize, or to support the scripting.

Because the *Gmsh* documentation is really good, and also nice tutorials are available on Youtube[14], we dispense with a *Gmsh* usage guide here, and focus on the OpenFOAM-relevant parts. Let us summarize it the development process as follows. You first must define points, then lines from these points, and from the lines (close loops) surfaces. Finally, from the surfaces, volumes are defined. Units here are taken as meters by OpenFOAM (but later scaling is possible).

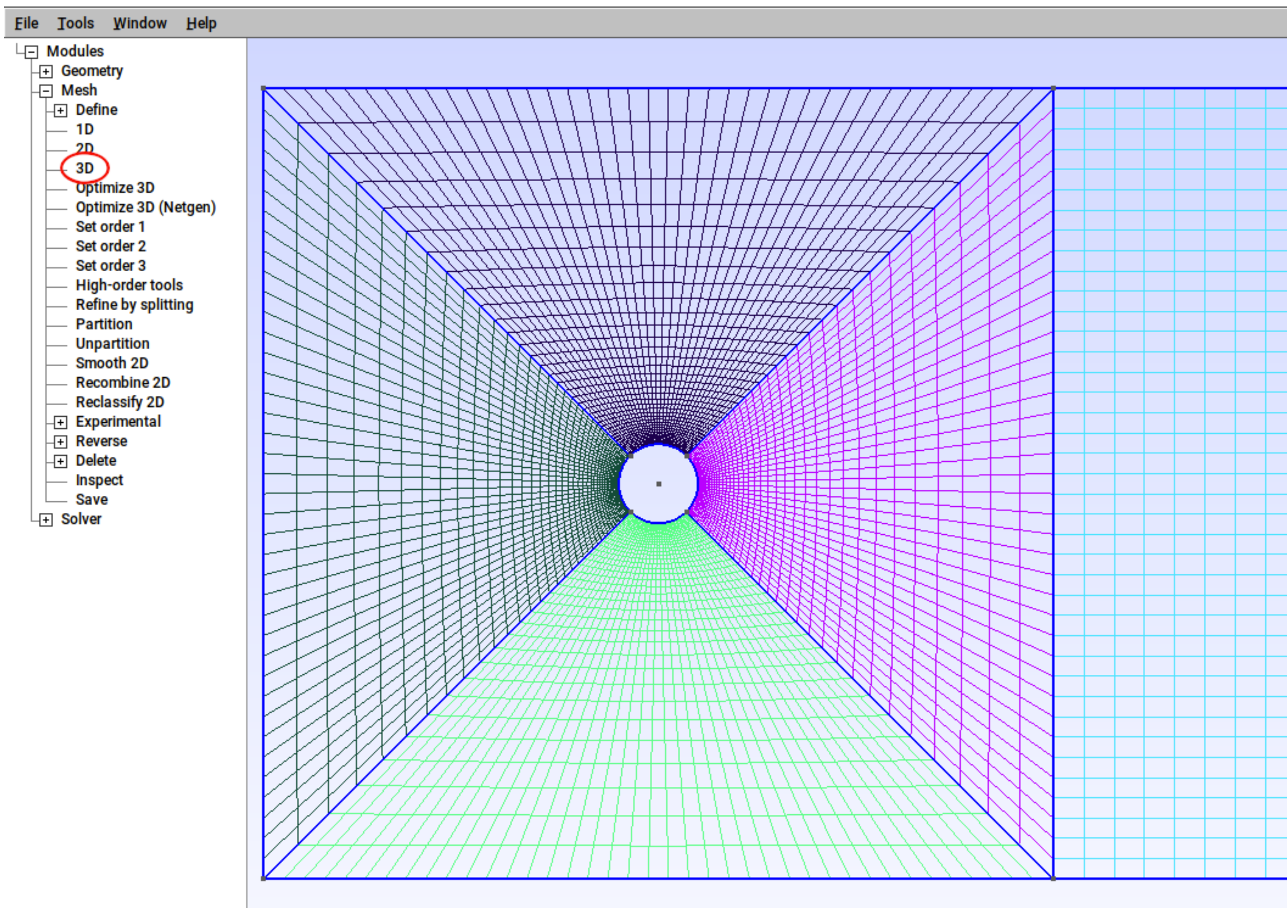
For OpenFOAM, we still need to define *physical surfaces*, and a *physical volume*. The surfaces become boundary conditions in OpenFOAM. And the volume – we call it *internal* – is filled by the mesh, where OpenFOAM operates on with its solvers and tools. You can find in the addendum a Gmsh Script of the Kármán Vortex Sheet Example. Paste it into a file called *karman.geo*! Open this file using *Gmsh*,

```
> gmsh karman.geo
```

and click in the *Mesh* menu on *3D*! The mesh is generated and shown.

---

13 In Linux, it is located in the bin directory. In Windows, the ***gmsh.exe*** is just in the top-level directory.
14 See, for instance, here and here.



Afterwards, *File → Save Mesh* saves the mesh to a file called *karman.msh*. That is, what OpenFOAM requires.

In order to create an OpenFOAM case from that, copy the cavity tutorial case,

```
> cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity karman_OF
> cd karman_OF
```

Copy the *karman.msh* file into that directory. The content should now look similar to

```
> ls
0  constant  karman.msh  system
```

That's the right moment to somewhat introduce already something of the OpenFOAM case directory and file schemes.

constant contains all constant parts of the simulation like the geometry description, the mesh, the physical material properties, etc. ***system*** contains the so called dictionaries, i.e. ASCII files, which govern the run-time behavior of the solvers and tools. We will look into the details later. The directory ***0*** contains the initial and boundary conditions of the physical field quantities. For *icoFoam*, this is the velocity *U*, and the pressure *p*. We will look into these files later.

For later use, save the *constant/transportProperties* files somewhere, and delete the *constant* directory.

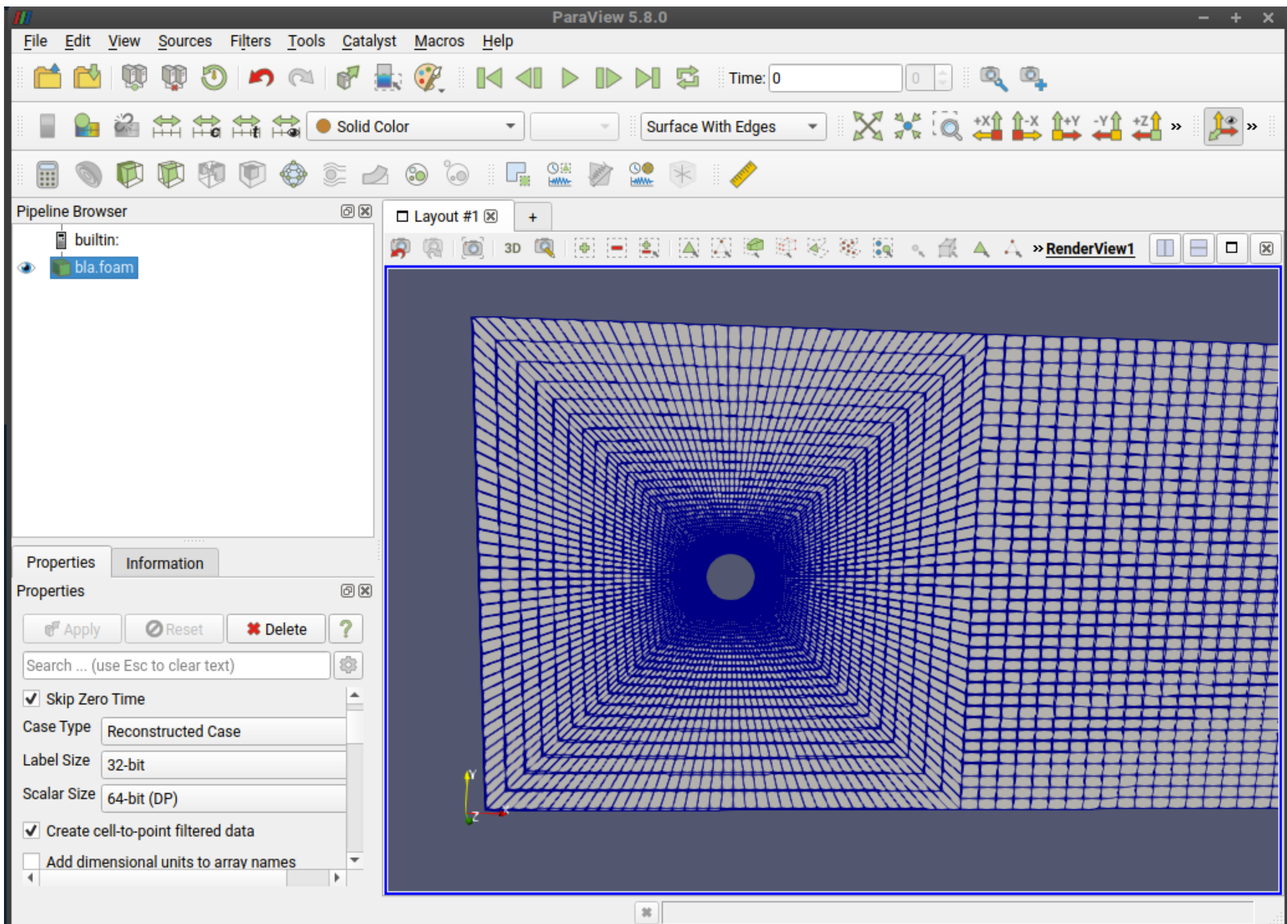
```
> cp  constant/transportProperties .
> rm -rf constant
```

That's it. Now we can transform the *Gmsh* mesh into the OpenFOAM mesh.

```
> gmshToFoam karman.msh
```

The output should be consulted for errors. And also whether the boundaries have been recognized correctly. Some warning might appear. They should not be ignored, but are usually not critical.

A visual inspection using *paraview* is probably also a good idea. Again, create a *.foam* file, and investigate the mesh in *paraview*.



OpenFOAM also provides some tool for checking the mesh quality.

```
> checkMesh
[…]
Checking geometry…
    Overall domain bounding box (-5 -5 0) (20 5 0.1)
    Mesh has 2 geometric (non-empty/wedge) directions (1 1 0)
    Mesh has 2 solution (non-empty) directions (1 1 0)
    All edges aligned with or perpendicular to non-empty directions.
    Boundary openness (4.80206e-20 -1.67601e-18 -4.10236e-18) OK.
    Max cell openness = 2.53896e-16 OK.
    Max aspect ratio = 8.19123 OK.
    Minimum face area = 2.17705e-05. Maximum face area = 0.068867.  Face area
magnitudes OK.
    Min volume = 2.17705e-06. Max volume = 0.0068867.  Total volume =
24.9215.  Cell volumes OK.
    Mesh non-orthogonality Max: 44.2169 average: 15.1579
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 2.42319 OK.
    Coupled point location match (average 0) OK.
Mesh OK.
```

What these mesh quality indicators are and mean, and what values are acceptable, is business of the finite volume method. However, an “`OK.`” should signal that the values for the present mesh are

acceptable for OpenFOAM. However, when you find numerical instabilities in your solution, you should return to checking the mesh. (This might prove somewhat difficult for adaptive meshes.)

Another tool OpenFOAM provides is **renumberMesh**. It tries to reshuffle the mesh cells, in order to reduce the system matrix's bandwidth, which might improve the time-to-solution.

```
> renumberMesh -overwrite
```

It can be further fine-tuned with a **renumberMeshDict** inside the **system** directory of the case. Consult the **mesh/parallel/cavity/system/renumberMeshDict-random** file in the *tutorial* directory for some overview. The default (no such dictionary) is often but acceptable.

Optimization should be done, when you spare substantially more time for the simulation than you spend for the optimization.

3. OpenFOAM Case preparation

When the mesh is prepared, one can step to the preparation of the simulation case itself. At this place, we now need to introduce a bit more about the OpenFOAM case directory structure.

The ***constant*** directory contains, next to the ***transportProperties*** file, a directory ***polyMesh*** with some files. These files are ASCII, and you can look into them.

```
boundary cellZones faces faceZones neighbour owner points
pointZones
```

These files contain all the information about the mesh – points, faces, cells, neighbor-lists etc. – all what is needed for the finite volume method. The details are usually not too relevant to know – unless you really must. But easier and less error prone is it to create these files using tools instead of writing or manipulating them manually.

The ***boundary*** file is something you must manipulate once after the mesh import. It contains next to a default header a dictionary (yes, they call it the same as the files …. Sigh!) with the boundaries – the labels are those we gave to them in the *Gmsh* file *karman.geo*.

We must set **FrontAndBack** to **empty**. The reason for this is that OpenFOAM is actually a 3D simulation software. But our case shall run as 2D case. **empty** effectively means that no boundary conditions are applied here.

physicalType patch; is to be commented out. That's done in a C/C++-like fashion (**//** comment until end of line; **/* … */** comment everything in between).

patch for the other faces is ok. Which other options are available, and explanations, can you find in the OpenFOAM documentation. The final Kármán constant/polyMesh/boundary can be found in the addendum.

Back in the top-level directory, we need to adapt also the files ***p*** and ***U*** in the ***0*** directory. Kármán 0/p and Kármán 0/U can also be found in the addendum. It is important to label the surfaces correctly as are written in the ***constant/polyMesh/boundary*** file. Otherwise, they are not found. But OpenFOAM will tell us surely if we do it wrongly.

Information: Here another *trick*. If you don't know the labels for dictionary flags like **empty** or **slip** for boundary conditions, just enter something weird (Hrvoje Jasak calls this ***the banana***

trick, meaning that you just enter the word ***banana***). Running with a sensitive solver/tool then throws an error, also telling you the possible alternatives. This works for many of the dictionaries and is usually faster to accomplish than to search in the sources.

Before stating the actual time integration (simulation), we want to mention some other dictionaries. The three dictionaries ***controlDict***, ***fvSchemes***, and ***fvSolution*** are the most important ones for the moment.

In short, ***fvSchemes*** contains the information about the schemes, how the operators of the PDE should be approximated. Usually, you don't need to touch it. But if stability problems occur, changing the one or other scheme might be necessary.

fvSolution contains the information about the iterative matrix solver and pre-conditioner to be used. Again, you usually do not change anything here unless you see that your simulation is not converging, or if it is not converging sufficiently fast – and other issues (like bad boundary conditions or bad mesh quality) are ruled out.

For now, leave ***fvSchemes*** and ***fvSolution*** from the cavity tutorial case unchanged.

By far the most important dictionary is the ***controlDict***. Here you control, when/where the simulation starts, when it ends, how many time steps, the step size, the I/O options. Kármán system/controlDict in the addendum shows the one-pass-through ***controlDict*** for the Kármán vortex sheet example.

4. Simulation and Visualization

In an ideal world, you would just issue

```
> icoFoam
```

and wait until the simulation finishes.

But our world is not perfect, and usually you proceed slightly differently. Specifically, with time series integration, you may not know exactly how long you will have to run. Or, at which steps you want to write out data. Or whether your simulation starts at all successfully – until you tried.

So, a good way is to first start the simulation for some integration steps, which you can observe on the terminal. Check the *Courant* number (specifically the maximum one)! Check the errors on the PISO loop! You can always interrupt with "Ctrl+C", and scrutinize the current advancement. Often, it is also a good idea to set **`writeInterval`** initially rather small such that the fields are written to file at each integration step. Then you can investigate with *paraview*, how the fields evolve initially.

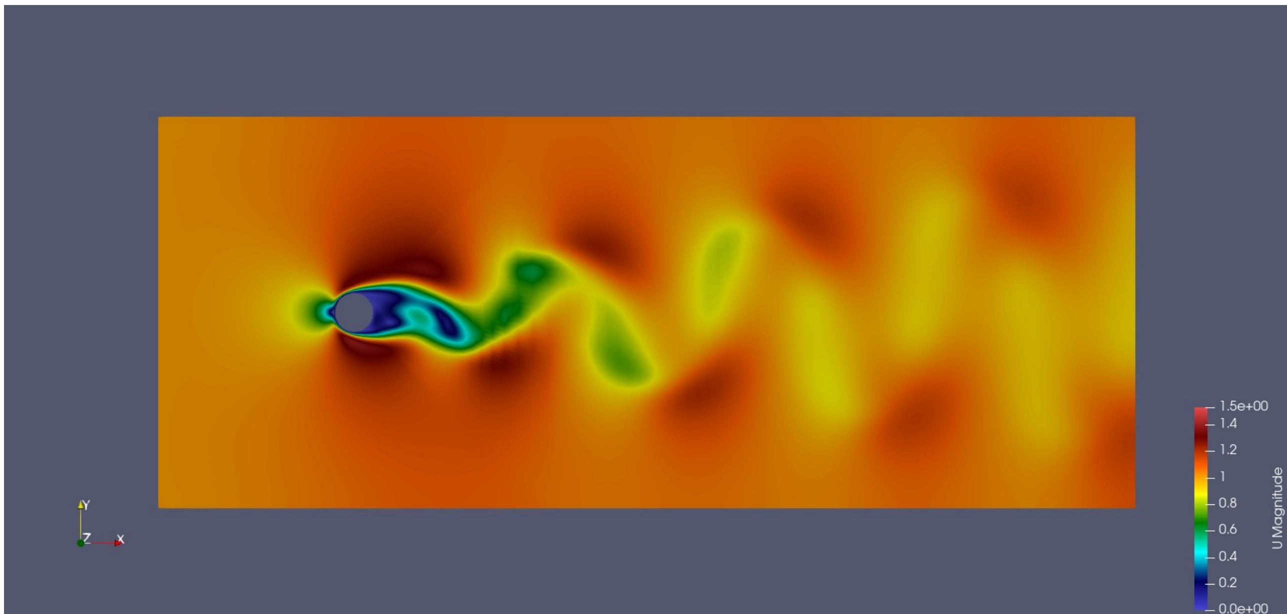
If you want to continue the simulation from the last time step written, just set

```
startFrom        latestTime;
```

Of course, it requires that **`endTime`** is larger than the latest written output.

If you finished, you should have many number-labeled directories in the top-level directory, one for each written time-step, containing the field data for each field (here, *p* and *U*). When loading the result into *paraview*, you can see the time evolution when clicking on the *play* button in the *time-series control*.

An example snapshot from the simulation looks as follows.



5. Concluding Remarks

Of course, the tutorial cases always work. Other than my own simulation case! Well, true! Experience is part of the business. And not all experiences can be just transferred in a tutorial.

But a thorough theoretical background can be very helpful. Question: What is the Reynolds-Number of our Kármán vortex sheet example? Answer:

$$\mathrm{Re} = \frac{L u_\infty}{\nu} = \frac{1\,m \cdot 1\,m/s}{0.01\,m^2/s} = 100 > \mathrm{Re}_{crit}$$

Here, $\nu = \mu/\rho = 0.01\,m^2/s$ is the kinematic viscosity from the ***constant/transportProperties*** file. However, that's not very obvious from the cavity tutorial's file content. OpenFOAM comes with often not immediately visible defaults. This here is one. Fully written, **`nu`** should be defined as

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

We have seen such dimension scheme already earlier for pressure and velocity in the ***0*** directory. The scheme contains in that sequence, mass (kilogram), length (meter), time (second), temperature (Kelvin), Quantity (mole), current (ampere), luminous intensity (candela).

Also implicit was here **`transportModel Newtonian;`**.

Please keep these things in mind when trying to interpret the results of your simulation. Align this analysis on your beforehand expectation. Don't take for granted that the tool is *thinking* for you!

Finally, did we do correctly? That's a tougher question than one might initially guess. A next step could be to take another or finer mesh, and observer whether the solution is still the same. That is, for instance, are the field amplitudes still the same? Is the onset of the instability still at the same time? Are the flow characteristics still the same? Are the conserved quantities really conserved?

Warning: Even if you convinced yourself that your simulation went correctly, you only know that you solved a PDE correctly. Whether this equation also describes the physical system under consideration with sufficient precision is yet another question. The comparison with experimental

results is therefore always a mandatory validation step. Results of such an analysis can be found on the verification and validation page of the OpenFOAM documentation.

This completes the example case workflow.

Import from other Tools ...ToFoam

OpenFOAM comes with quite a list of conversion tools for importing meshes from other tools. One example we saw already in the previous section for *Gmsh*.

```
> ls $FOAM_APPBIN | grep ToFoam
ansysToFoam           fluentMeshToFoam    netgenNeutralToFoam
ccmToFoam             gambitToFoam        plot3dToFoam
cfx4ToFoam            gmshToFoam          smapToFoam
chemkinToFoam         ideasUnvToFoam      star4ToFoam
datToFoam             kivaToFoam          tetgenToFoam
fireToFoam            mshToFoam           vtkUnstructuredToFoam
fluent3DMeshToFoam
```

What else needs to be done, please take from the OpenFOAM User Guide, or the documentation. Also for post-processing, conversion back to others tool is possible. Look for **foamTo…** tools!

blockMesh – OpenFOAM Geometry and Meshing Tool

OpenFOAM itself has some built-in geometry construction and meshing tool, which is sometimes astonishingly useful and powerful. It is however a bit clumsy to use, as absolutely no GUI is given. You just work in an ASCII file – ***blockMeshDict*** (inside the case's ***system*** directory) – and issue from time to time

```
> blockMesh
```

Afterwards, you can investigate the result using *paraview*.

The code scheme of the ***blockMeshDict*** is described in the OpenFOAM User Guide. One essentially defines again points (*vertices*), and from them volumes (*blocks*). *Edges* can be deformed afterwards. Last but not least, one needs to define the boundary surfaces, which are later used for setting the boundary conditions.

In the addendum, we provide a Kármán blockMeshDict file, which can be placed in the Kármán vortex sheet example of the previous section – into the ***system*** directory. The rest being otherwise the same, you should be able to repeat the simulation then with the following steps.

```
> rm -rf constant
> blockMesh
> checkMesh
> renumberMesh -overwrite
> icoFoam
```

For an elaborate overview of the ***blockMeshDict*** "language", please also look into the *tutorial* suite into the *mesh* subdirectory. What you can do with ***blockMeshDict***, can be guessed maybe by Kármán blockMeshDict advanced, which does otherwise the same as the previous ***blockMeshDict***.

Information: Also for the other available meshers of OpenFOAM, look into the *tutorial* suite!

Exercise – Kármán Vortex Sheet with blockMesh-generated Mesh

The ***blockMeshDict*** allows some variants. For the flow around a cylinder, you can also adapt the blockMeshDict from the OpenFOAM User Guide. Try to make it comparable in geometry and mesh to what we already provided here! For some more extended analysis, please consider this presentation, and try to repeat this analysis! Much of that analysis can be done in *paraview* such as determining the drag coefficient.

Also try now different solvers like **`simpleFoam`** or **`pimpleFoam`**! In order to do this, look into the *tutorial* cases, and the respective necessary dictionaries. Copy and modify them for your needs, and let them run. OpenFOAM will tell you if something is missing or wrong.

STL Files into OpenFOAM – snappyHexMesh, cfMesh

Of course, OpenFOAM is a 3D PDE solving system. Even if we illustrated the workflow with 2D examples, the more interesting use cases are surely in 3D. Usually, you will have probably some highly complex geometrical structures, where a fluid is flowing through or around.

Gmsh as an external meshing tool is surely viable, as others are, as long as there is some **`...ToFoam`** tool available. But **`blockMesh`** is not very useful in such cases. Still, OpenFOAM offers also some more advanced mesh generation tools. **`snappyHexMesh`**[15] and ***cfMesh***, where the latter in turn is a collection of several mesh generation tools. These tools can generate hexahedral (**`cartesianMesh`**, **`cartesian2DMesh`**), tetrahedral (**`tetMesh`**) or even polyhedral (**`pMesh`**) meshes.

These tools allow for region-wise automatic mesh refinement, and specifically a good handling of boundary layer meshes relevant for CFD applications.[16] They have quite some amount of tuning options, and can be executed also in parallel.

For both there are very good user guides (see Useful Links and Literature under the OpenFOAM section).

Stays the question, how to provide the geometry? Both tools allow for STL files. Such files can be created with any CAD software. But if you want to stay on the non-commercial trip, we can recommend FreeCAD (docu) and Blender (docu), amongst others.[17]

There are lot of nice tutorials on Youtube, and we recommend to consult them (just look for "*OpenFOAM FreeCAD*", for instance, or "*blender NACA airfoil*", and the like).

For handling the STL geometry (*move*, *rotate*, *scale*, …), OpenFOAM offers the **`surfaceTransformPoints`** tool. And certainly, there is more.

This section is more a *perspective* and a *hint where to find*, in order to illustrate the spirit behind OpenFOAM, instead to describe really how to do it. You can easily extend your OpenFOAM workflow with other – mostly free – tools, to be as productive as you want to be. It is of course in your responsibility to use these tools correctly and efficiently. So, for reasonable CFD results, you should use well and finely grained geometries, for instance, in order not to generate (nonphysical) artifacts.

---

15 There are **`foamyHexMesh`** and **`foamyQuadMesh`** available, too, which for some cases are certainly useful.

16 OpenFOAM also has the tools **`refineHexMesh`**, **`refinementLevel`**, **`refineMesh`**, and **`refineWallLayer`**.

17 Both are free and available for the major OSs, and usually do not require that you build them from source, or install them as administrator.

Reasonable Practice – Version Control and Backup

As your cases develop and evolve, you make often changes, where you later still need to know that you did them. And why. Here, a *version control system* like git or svn is a valuable help, and often not difficult to use.[18] Just add your scripts and dictionaries to the version control. That's it! Not much, but with a tremendous advantages.

The results (of the simulation) are usually not put under version control. In worst cases, you can reproduce any result of a simulation whenever you need it.

A second issue is *backup*. Surely, you don't want to lose all your precious work just because a hard disk gives up operation. There are really many possibilities in the 21st century to secure the own work. Cloud services are viable as well as local USB disks. You don't want that others can access your work? No problem! Compress and encrypt your backups. Just take care not to lose the password!

What should be backed up? Well, everything under the version control is a good beginning. You can also backup the results – after some moderate compression. Here, you must decide, what's more expensive: The storage space consumed, or the time to repeat the simulations? But please, be realistic!

Own/User-provided or external Solvers and Tools

This section requires you to have a full-fledged OpenFOAM (with all sources and headers) at hand. The run-time alone is not sufficient! Furthermore, some aspects are very C++-loaded. So, without this knowledge, some difficulties to understand might occur.

Still, we won't introduce here the OpenFOAM API (which is actually a topic of a complete course on its own; furthermore, the API changes still quite substantially, what makes it difficult to keep pace when preparing a more sustainable tutorial).

1. Preparation

We would like to solve the Swift-Hohenberg Equation, which represents a model equation for convection instabilities in thin liquid layers, heated from below (cf. Marangoni effect). We use the following form of the equation,

$$\frac{\partial \psi}{\partial t} = [\varepsilon - (1 - \Delta)^2]\,\psi + A\,\psi^2 - \psi^3$$

where

- ψ is a real-valued function of time, t, and 2D space coordinates, (x, y)
- ε is a control parameter; instability of solution $\psi = 0$ occurs for $\varepsilon > 0$
- A is another parameter, which determines, whether roles ($A = 0$) or hexagonal convection patterns (e.g. $A = 0.5$) occur ($A < 1$); (cf. Figure 1)
- Δ is the 2D Laplacian, $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$

18 git has the advantage that you can use it without a server, and no need to push to github or so.

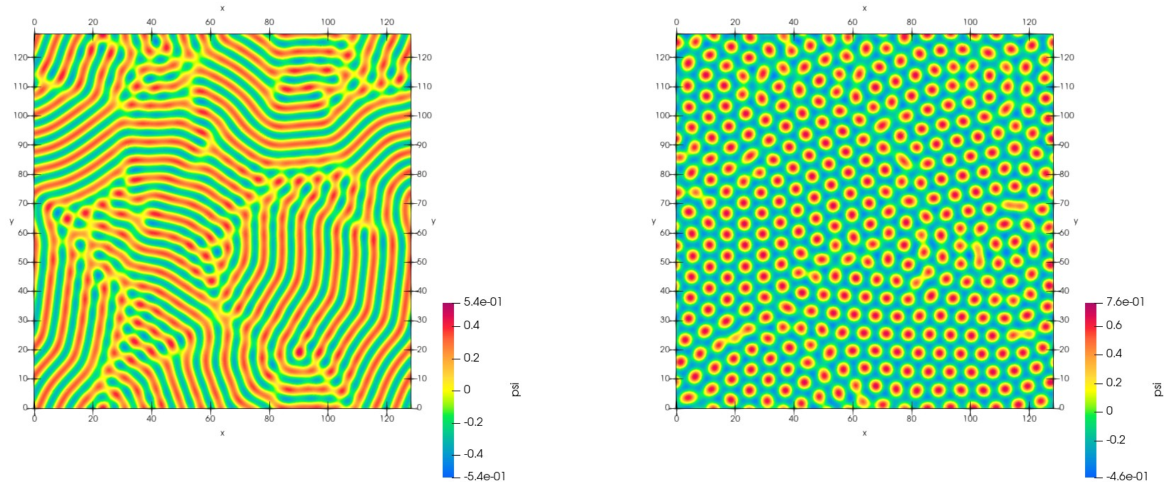


Figure 1: Solutions of the Swift-Hohenberg equation. Left: Roles ($\varepsilon = 0.1, A = 0.0$); Right:Hexagons: ($\varepsilon = 0.1, A = 0.5$). Array size is 256 × 256, with $\Delta x = 0.5$ length units (so, total area is 128 × 128 unit length squared); critical wavelength is 2π per unit length (i.e. $k_{cr} = 1$; so around 20 roles or hexagons per 128 unit length should appear).

For simplicity, we choose $\psi = 0$ as ***initial condition*** (with some *noise* to accelerate the occurrence of the instability). Further, we choose ***periodic boundary conditions***, i.e. $\psi(x+L,y) = \psi(x,y)$ for some system size L. And so also in y-direction.

From a so-called linear stability analysis, one can determine that the homogeneous stationary solution $\psi = 0$ becomes unstable if $\varepsilon > 0$. For $A = 0$ (or at least rather small), roles become the stable configuration. If A is sufficiently large, hexagons become the dominant structure.

The characteristic wave number is equal to the critical one, and here by construction equal to one unit length ($\varepsilon - (k_{cr} + \Delta)^2$; $k_{cr} = 1$). The corresponding wavelength is $\lambda_{cr} = 2\pi/k_{cr} = 2\pi$.

2. Creating the OpenFOAM Solver

Although you will rarely split the solver development and the case setup so cleanly as we show here, for brevity of the representation it is the most suitable approach.

In a first step, choose one of the existing solvers that most closely matches your needs. Rewriting is easier than development completely from scratch. The Swift-Hohenberg equation is a non-linear scalar transport equation. So, we reuse **`scalarTransportFoam`** here. Somewhere in your **`HOME`** directory, just copy the solver source to your location, and start some modifications.

```
> cp -r $FOAM_APP/solvers/basic/scalarTransportFoam mySolver
> cd mySolver/
> mv scalarTransportFoam.C mySolver.C
> sed -i 's/scalarTransportFoam/mySolver/g' Make/files
> sed -i 's/FOAM_APPBIN/FOAM_USER_APPBIN/g' Make/files
  # also for FOAM_LIBBIN -> FOAM_USER_LIBBIN if necessary
> wmake
```

We call our solver here **`mySolver`**. You can surely use more fantasy!

After calling **wmake**, this new solver should compile without errors, and finally be placed into **$FOAM_USER_APPBIN**. If errors occur, read them carefully, and figure out by what they are caused. C++ knowledge and experience is inevitable here.

Once successfully done, you can start to modify the solver's source code. We start with the Swift-Hohenberg: createFields.H. For simplicity, we marked everything red, what we changed.[19] That's indeed not too much. Actually, only three lines. The rest is just output of information. And most of what we changed, could essentially be *seen* from the original.

The only thing requiring an explanation might be **dimensionSet(0,0,0,0,0,0,0)** for ε and A. In principle, we could avoid to set the dimensions here, and do it later in the ***transportProperties*** dictionary. But enforcing consistency here makes the solver fail early if something is wrong.

The question might appear why we left the velocity still in, as it is not actually necessary. To be honest, we simply don't know exactly, where everywhere in the included headers the velocity is required. Just commenting out the definition of U resulted in many errors, what became even worse, when trying to remove also the header includes which threw the errors. So, we finally decided to leave U in, as was the case for the **scalarTransportFoam**, where it is also just an *inactive* component not being required to be solved for (no equation is there for U).

Next, we modify the Swift-Hohenberg: mySolver.C. Again, we did not change too much.[20] Unfortunately, we could not figure out, how to use completely dimensionless equations in OpenFOAM. Therefore, we needed to introduce some *dimensionedScalar*s, which compensate for the dimension of **fvm::ddt** and **fvc::laplacian**. (Well, we didn't claim that OpenFOAM is the best tool for solving the Swift-Hohenberg equation, after all.)

This compiled successfully, and is now ready for the use. But the last judge will be the test.

Remark: Please consult the book of [Moukalled] (see OpenFOAM in the addendum)! It contains some quite readable introduction also into the OpenFOAM API … even though the API is changing, it is really helpful to get a more general overview.

3. Setting up the OpenFOAM Test Case

Again, we need to define the geometry and the boundary conditions, first. We use for simplicity again **blockMesh**. Please, consider the Swift-Hohenberg: system/blockMeshDict. The geometry is quite straightforward in this case. The only new thing are the periodic boundary conditions, which are called **cyclic** in OpenFOAM, for a simple reason – they are more general than just periodic boundaries. All they require then is the neighbor boundary, and the specification of the face itself. So, no further magic here.

```
> blockMesh
```

from the top-level directory creates then the constant folder with the polyMesh subdirectory, which again contains the mesh description.

---

19 For the API documentation, please consult the Extended Source Guide! Use the search capabilities there!

20 We must admit that we are not so familiar with the OpenFOAM API, either. So, there is maybe a more clever way to exploit the implicit **Foam::fvm::** namespace, in order to implement a more stable solver. The **Foam::fvc::** namespace contains explicit representations, as far as we know. Still, we got it working.

Now, we can create also the Swift-Hohenberg: constant/transportProperties. It only contains the values for ε and A. You can also place some dimension specifier for them. If you do it wrong, meaning not being dimensionless, **`mySolver`** will fail with a telling error message.

The next step is to set the initial fields. For the velocity, which in fact requires some syntactically correct ***0/U*** file, you can simply set a uniform zero field otherwise. So, Swift-Hohenberg: 0/U will do.

The initialization of ψ is however crucial. Actually, we would like to have some random value here, uniformly distributed in a small interval around zero. OpenFOAM's capabilities have matured to accomplish this in a very elegant way using codeStream. Without further ado, this inline C++ coding creates dynamically loadable libraries, which are compiled, linked and executed at run-time. Please look into Swift-Hohenberg: 0/psi. That's essentially the same C++ you need to write solvers, tools, and utilities in OpenFOAM.

For many other cases, there are already tools like **`setFields`**, ***swak4foam*** (when you still get it compiled with your OpenFOAM version), etc. With **`setFields`**, you can even use STL files to define geometric regions. But *codeStream* appears very convenient, once you understand it.

We are not yet ready, yet. We need a ***controlDict***, a ***fvSchemes*** and a ***fvSolution*** file. These we can lend from **`$FOAM_TUTORIALS`*****/basic/scalarTransportFoam/pitzDaily/system***, and just modify what's needed. The end time, and time step, as well as the write intervals need to be tested out in the Swift-Hohenberg: system/controlDict. Too large a time-step will make the simulation diverge, as we have quite a bit explicit parts in the discretization. Further more, the spatial discretization is quite fine. So, the time-step needs to be small, too. How long will we need to run the simulation? As we use a dimensionless equation, we can say that there should happen something within one time-unit. Indeed that's what we observe. But the real structure formation saturates around 50 time units, or a bit more. Finally, we must try.

However, just starting the **`mySolver`** now results in some other error. It will tell us that some *laplacian scheme is missing*. The implementation of the Swift-Hohenberg equation contains some spatial derivatives, where OpenFOAM not immediately knows what to do with them. Just modify the ***fvSchemes*** and ***fvSolution*** files as shown under Swift-Hohenberg: system/fvSchemes and system/fvSolution.

So, surely not all schemes are really required. But for laziness we let them in. For the solutions, the default pre-conditioner was **`PILU`**. As it is not symmetric, OpenFOAM complains. But it gives you also a list of available pre-conditioners. That's exactly the ***banana trick***, we described above. If you want to use **`GAMG`**, you are also asked for a smoother – please go and find one! But **`DIC`** and **`FDIC`** work without further requirements.

4. Final Remarks

This section should illustrate, how one could in principle extend OpenFOAM with further tools and solvers, and how to approach and analyze the cases. Step-by-step. However, usually it will not go through so smoothly and systematically as it might appear here. Still, the systematic approach appears us the most promising one.

In any case, one can gain a lot of insight into the functionality and setup of OpenFOAM. And it might already now make you feel the flexibility OpenFOAM offers, but definitely its complexity.

Please consult the books of *Saad* and *Moukalled*, which we placed in this tutorial’s literature list.

Last but not least, what is true for your cases, also holds true especially for your solvers and tools! ***Version control*** and ***backup***! Developing solvers is software development. Maybe others want to use your work, too. Thus, ***documentation*** is inevitable. But even more importantly so for yourself.

Part 3 – HPC Topics

If you are able to understand that a solution is the one of your problem, then you probably understand your problem well enough, and may have found this solution also by yourself.

Ce veut dire … If you never touch the limits of a computer, your program never crashed, your hard-disk never ran full, you may not yet understand this part – the solutions that are offered here. That's quite ok from our side. You can come back here, and consult these chapters, when you are ready.

Parallel OpenFOAM

When to use Parallelism?

There are mostly two reasons for going parallel:

1. the (projected) case run-time is too long (aka *impatience*)
2. the (projected or experienced) memory consumption is too large for the current system

OpenFOAM is (currently still) an MPI-only code, which means, that you can only decompose a single case into several domains, which independently solve the equations, and communicate the information on the boundary fluxes to the next neighbors via MPI (*Message Passing Interface*).

This imposes some restrictions on the capabilities of system provided parallelism. Modern systems may have wide vector registers, where the same operation can be executed in parallel on several input data. Except for KNL (*Intel Knights Landing*), OpenFOAM seems not to support wider vector registers, yet. At least using them appeared not to have a speed advantage.

Modern processors have several physical CPU cores. As they share common memory, communication could happen through this faster memory communication. This is usually accomplished via threading (phtreads, OpenMP, TBB, …). OpenFOAM seems not officially to support this either, although single efforts can be found on the internet.

However, with several MPI processes (ranks) spawned, a multi-CPU processor can be utilized for OpenFOAM in parallel. One needs to care for placing the different MPI ranks on different physical CPU cores, as overlapping processes on a single core usually burdens heavily the performance.

MPI has a big advantage in comparison to thread (shared memory) parallelism. One can easily spawn the MPI processes also across several (multi-CPU processor) nodes, which are connected via some (high-speed) network. However, the communication might become some sort of a bottleneck for performance. So, clever domain decomposition strategies must be used, which minimize the cross-domain communication. OpenFOAM offers several options to accomplish this.

Using several nodes for parallelizing OpenFOAM also offers a second advantage. Each node has presumably its own piece of memory. So, using two nodes with the same amount of resources doubles automatically the amount of CPUs and memory available.

But rarely is the amount of memory per CPU core so ideal that you can exploit both. In most cases, you will be able to use all CPUs, but only small parts of the total memory. Here, you even might want to press as much data into the *cache hierarchy* as possible, as caches are much faster than the main memory (commonly known as *RAM*). But there might also be cases, where memory per CPU

core is the limiting factor. In an extreme case, you may only use one CPU and all the memory of a node.

OpenFOAM seems to be very well behaved, such that you can freely find the optimum ratio of CPUs and memory per node for your simulation. Still, there are some other issues related to parallelism in OpenFOAM – specifically on HPC systems –, we will discuss below.

Basic Case Setup and Workflow for parallel Runs

The general workflow in OpenFOAM is to first decompose the geometry with **decomposePar**. It is controlled via ***system/decomposeParDict***. After the decomposition, you have several ***processor**** directories in your top-level case directory. One for each MPI rank (or better, one for each rank that does I/O; see below).

How the decomposition looks like can be investigated with *paraview* again. One can open the geometry of each *processor* directory, after creating a *.foam* file therein e.g. via

```
> for i in processor*; do touch $i/bla.foam; done
```

A less painful way is however to use the **–cellDist** option of **decomposePar**. It creates a ***cellDist*** inside the ***0*** folder, which can be used in *paraview* to color the different domains.

The solver then operates in parallel on these directories. Once the simulation is done, **reconstructPar** recombines the case output split into and distributed over the *processor* directories into top-level directories – one for each written time-step.

In the simplest case, your workflow looks like this

```
> decomposePar –cellDist
> mpiexec –n 4 icoFoam –parallel
> reconstructPar
```

where **icoFoam** can be replaced by other solvers. Those, which allow for the **–parallel** option – check the command-line options with **–help**! Our self-written solver of the previous part also works in parallel. That's some of the magic of OpenFOAM.

So simple that may sound, and in part actually is, there are also intricacies. First of all, you must explicitly specify the number of MPI ranks in the ***decomposeParDict***, e.g. as

```
numberOfSubdomains  4;
```

Starting the **mpiexec**/**mpirun** with an inconsistent number of MPI ranks (option **–n <no of ranks>**) results in an error. Similarly, **–parallel** forgotten results probably in a mess. In an ideal case, you get some error. But mostly, there just run several serial solver instances simultaneously, which causes possibly some chaos at I/O.

To make the workflow a bit less error-prone, OpenFOAM comes with some bash-wrapping support you can find in the ***Allrun*** scripts of the *tutorial* cases.

OpenFOAM knows several domain decomposition methods. The simplest is to write

```
method hierarchical;
```

into the ***decomposeParDict***, which requires the specification of the domains in *x*, *y*, and *z* direction

```
coeffs {n (2 2 1);}
```

Their product needs to match the **numberOfSubdomains**.

Other decomposition methods are *metis*, *kahip*, and *scotch*, and others, of which *scotch* comes usually per default. For the other libraries, one needs to install them separately first, and re-run **`Allwmake`** for OpenFOAM again, such that the respective libraries are built.

For usage, just place for instance the following into the ***decomposeParDict*** additionally to the **`numberOfSubdomains`**,

```
method scotch;
```

For more information on details, please consult the OpenFOAM documentation.

`decomposePar` is intrinsically serial, as **`reconstructPar`** is. However, both have an option **`-time <range>`**, which can be used to execute the decomposition or reconstruction on individual time steps. Because this can be done independently, this offers some path to parallelism (e.g. GNU parallel, or MPI parallel).

Btw. also the meshing can be accomplished in parallel. Check **`reconstructParMesh`**!

Exercise – MPI parallel Workflow

Use one of the previous example cases, or one *tutorial* case (***motorBike*** or ***windAroundBuildings*** in **`$FOAM_TUTORIALS`*/incompressible/simpleFoam/*** should be large enough), and exercise the workflow. Also investigate the results in *paraview*, for both decomposed and reconstructed cases.

MPI Scaling – How many parallel Ranks should I use?

1. Theoretical Considerations

The answer to this question depends, of course, somewhat on your bottleneck, and the intended workflow. For the classical monolithic OpenFOAM case, you can consider the following.

Amdahl's Law states that you can speedup a program, which requires on one CPU a run-time T_1 (e.g. seconds), by a factor of

$$S_n = \frac{T_1}{T_n} = \frac{1}{(1-p)+p/n}$$

where T_n is the run-time of the same program, but running parallel on *n* CPUs. *p* is the fraction of the serial program run-time that can be parallelized ($1-p$ is thus the inherently serial part of the program).

In order to know how fast a program runs on *n* CPUs, we only need the serial run-time and the speedup,

$$T_n = T_1 / S_n = T_1 (1 - p + p/n)$$

In an ideal case, *p* is one (perfect parallelizability). In this case, one could reduce the run-time to arbitrary small values by just increasing the used amount of CPUs. It is clear that this is unrealistic. But we can compare the actual run-time with this ideal run-time, and define a parallel efficiency,

$$\varepsilon_n = \frac{T_1/n}{T_n} = [n(1-p)+p]^{-1}$$

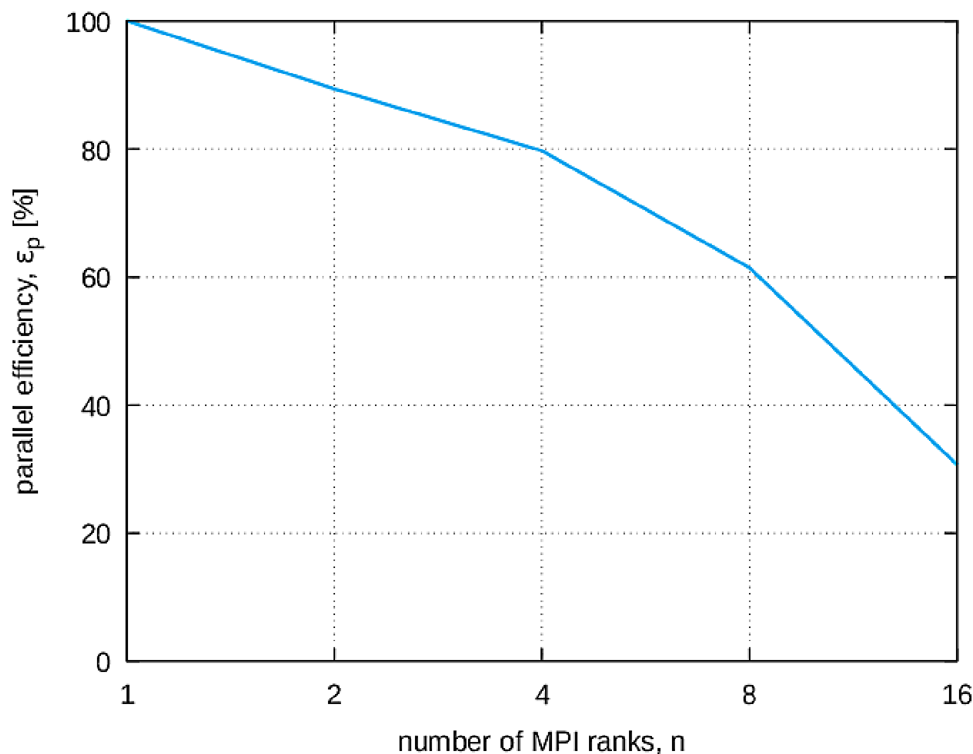
and say, further scaling makes sense (brings a speedup) only as long as the parallel efficiency is larger than say 70%.

2. How does it look in Practice?

Considering our rather small Kármán vortex sheet test case of the previous part, we find the following run-times for 10 time-steps (with *scotch* domain decomposed).

$n=$	1	2	4	8	16
$T_n=$	118 s	66 s	37 s	24 s	24 s
$T_1/n=$	118 s	59 s	29.5 s	14.75 s	7.375 s
$\varepsilon_n=$	100%	89.4%	79.7%	61.5%	30.7%

Effectively, we see already no speedup anymore with more than 8 MPI ranks. This case is too small (has too few mesh cells – please look at the **`checkMesh`** output), such that at this level of



parallelism, no speed can be gained anymore, and bottlenecks become dominant. More than about 5 or 6 MPI ranks are not reasonable.

3. What if that's still not enough, and the simulation would still take intolerably long?

Every time you pose this question, you should start some optimization considerations. MPI parallelism is only one solution, however. We mentioned already earlier that vectorization would be another solution. Sure! If a single computation can work with 4 or 8 numbers simultaneously instead of just with one, a single time-step would then take only 25% or less.

OpenFOAM seems not very amenable to vectorization, unfortunately. But you have more tuning screws available. One was **`renumberMesh`**. And also the mesh quality can be checked again. If the corrections like those for non-orthogonality take some substantial time, a mesh with orthogonal mesh cells might largely reduce this.

One can also tune the iterative solver settings, change the pre-conditioner, etc. Usually, these settings are quite reasonable for the *tutorial* cases. Yet, for your case maybe not!

Something, maybe too obvious to be mentioned here, is that you may also reconsider the solver used. Maybe another does the same job for your system and questions under consideration, but with less effort. This may even include the use of simplified equations with approximations.

More optimization possibilities are thinkable. Be creative!

Finally, sometimes, we simply must admit that a further optimization is (currently) not possible. In such cases, one simply needs to look for different approaches.

4. Preparation of large Cases

With the meshers today, it is so easy to just make the mesh finer – and play in the league of HPC.

Caution: Never assume that a larger, mesh-refined case on a HPC systems runs with the same speed as your smaller case on your workstation![21] Always measure and assess!

- Start with a smaller case: Complete geometry, but coarser mesh.[22] Check the main memory consumption (`\time -v …` can be helpful here). Determine the run-time of some time-steps. Try low MPI parallelism (2 would be fine, if it works), and observe whether parallelism helps at all.
- Increase the mesh by a factor of two, and repeat the steps of above. Are the results scaling? For instance, is main memory also just doubled? Is the run-time also just doubled? Does parallelism still reduce the run-time proportionally? With the same efficiency as for the case with a smaller mesh?
- Also check the I/O. How many data you need to write out. Is enough disk space available? Or do I need to do in-situ post-processing in order to reduce the file-system burden (see below)?

In this way, you slowly and systematically approach the unknown regions (case and parallelism size), and gain the necessary experience to handle larger cases. And you learn the possibly unexpectedly occurring bottlenecks and show-stopper, with some chance to remedy. With the impatient path of scaling immediately without this investigation, debugging might be impossible, and in many cases at least very frustrating. The more information you have, the better your chance for success.

Advanced File IO

This and the following section emphasize the peculiarities of HPC clusters. To have access to the one or the other is therefore helpful and necessary to follow the example workflows. We also present here some tools, which are mostly available on such clusters in the one or the other form. To install them on laptops or smaller clusters is usually not so useful (meaning that it is usually not worth the effort), but surely also possible.

---

21 This might be especially true as HPC processors have usually a smaller clock frequency than desktop CPUs.
22 If this does not fit into a single node, take more nodes until it fits, and your program does not run into a out-of-memory. Usually, you can estimate (about *number of fields times number of mesh cells times bytes per floating point value*) the minimum required size. Additional stuff might increase this. Measure how much more!

This section is not completely focused on OpenFOAM but can be useful also for other HPC software packages. Still, it is very relevant for large scale OpenFOAM.

GPFS and Workflow Proposals

Many HPC clusters have some high-speed, large-volume file-system like GPFS, which for I/O performance reasons have a rather large block size, e.g. of 8 or 16 MByte. OpenFOAM notoriously produces a lot of small files – for each I/O rank for each time-step for each field written out – which are much smaller than this block size, and so rendering the storage on GPFS quite inefficient. Even more problematic for large cases is that each file and directory consumes inodes (file system metadata), the number of which is usually also limited.

We are talking here about case sizes consuming *hundreds of millions and more* inodes! In such cases, this means, ***reducing the amount of the OpenFOAM output files must actively be managed***!

The reconstruction of the case might already be helpful. At least the MPI rank multiplicity is reduced. So, maybe does the collated I/O (see below).

For an overview, you can use the mpifileutils.

```
> mpiexec -n 10 dwalk -l -v <path to OpenFOAM case directory>
[…]
[2022-03-10T08:44:04]   Directories: 163457834
[2022-03-10T08:44:04]   Files: 36893459343
[2022-03-10T08:44:04]   Links: 134534
```

Leaving out the option **-l**, you also get a statement about the amount of data, and the mean size per file. But it will take much longer to run on large file-sets.

But the *mpifileutils* can do quite a bit more. Using **dtar** and **dbz2**, it is quite easily possible to bundle and compress large amounts of files and directories into a single file.

```
> mpiexec -n 10 dtar -cf archive.tar <directory>
> mpiexec -n 10 dbz2 --compress archive.tar
```

creates a file called *archive.tar.dbz2*. You can look into it via

```
> tar tf archive.tar.dbz2
```

or

```
> tar tf archive.tar.dbz2 | less    # a pager might help
```

and also extract some specific folders or sub-directories (e.g. for post-processing)

```
> tar xf archive.tar.dbz2 --wildcards "example_case/processor*/200"
```

This workflow is maybe also reasonable for creating archives, which shall be stored on tape for later revisions of analyses.

Parallel Reconstruction

We already mentioned that **reconstructPar** is actually quite serial. However, many OpenFOAM cases produce a lot of time-step output folders, which can be reconstructed independently of each other. That is, we can do this more or less in parallel.[23]

In order to ***parallel***ize the **reconstructPar** step on a HPC Slurm cluster, the following script might be illustrative (for the principle – there is always more than just one possibility to realize a workflow).

```
#!/bin/bash
#SBATCH HEADER
# load OpenFOAM environment

cat > reconstruct_job.sh << EOT
#!/bin/bash
REC_STEPS="\$(cd processor0; echo [0-9]* | sed 's/^0 //' | xargs -n \
            $SLURM_NTASKS echo | sed -n ''\$((\$PMI_RANK+1))''p | tr ' ' ',')"
echo "reconstruct: \$REC_STEPS"
reconstructPar -time "\$REC_STEPS"
EOT

chmod u+x reconstruct_job.sh

mpiexec -l -n $SLURM_NTASKS ./reconstruct_job.sh
```

$SLURM_NTASKS is defined by Slurm (**salloc**, **sbatch**). For other schedulers, please consult their docu for its replacement. **$PMI_RANK** is defined by Intel MPI (**mpiexec**), and contains the rank ID. For Slurm's **srun**, this variable is called **$SLURM_PROCID**.

$REC_STEPS essentially just contains a comma separated list of time-step labels, which are present in ***processor0***. The ***0*** time-step is excluded (what does not matter much).

Parallel Decomposition and Reconstruction

Since quite a while, there exist the new tool **redistributePar**. It is completely parallel, and supposed to replace **decompsePar**, **reconstructPar**, and **reconstructParMesh**. This makes the last section obsolete then.

For decomposing, just the command-line flag **-decompose** in required. Also a *decomposeParDict* is required, and **mpiexec** must be called with as many MPI ranks as are **numberOfSubdomains** specified in the dictionary. As is can only run in parallel, also the option **-parallel** needs to be added … somewhat redundantly.

Reconstruction is accomplished by replacing **-decompose** by **-reconstruct**.

There are more options e.g. for specifying the time steps to be reconstructed. As this is again feasible in an independent manner, one can parallelize according to the previous section, possibly.

Binary, Compressed, and Collated I/O

A large part of documentation can be found on the OpenFOAM docu page again.

OpenFOAM allows for quite a flexible file output handling. The output data can be **ascii** – and here then also compressed (gzip – **writeCompression on;**) or uncompressed (**writeCompression off;**) – or **binary** (uncompressed only) data. This is specified in

23 With some clever implementation, you could do this reconstruction even in parallel to the ongoing simulation.

writeFormat. For **ascii**, you can also specify a **writePrecision**, what for **binary** is irrelevant.

In all those cases, there is no change on how to analyze the data using *paraview* – also for decomposed cases.

Since OpenFOAM v1712 (OpenFOAM-6), a new feature was added, which is supposed to alleviate the GPFS issues mentioned above. But it also accounts for the multi-core structure of modern compute nodes, where I/O per node can be optimized when bundled and buffered node-locally (exploiting also the higher speed of on-node MPI communication). Exactly what we need for HPC! We talk about ***collated I/O***.

The workflow is as follows.

```
> export FOAM_IORANKS="(0 4)"              # mandatory here
> decomposePar -fileHandler collated
> mpiexec -n 8 Solver -parallel -fileHandler collated
> reconstructPar
```

$FOAM_IORANKS must contain a OpenFOAM list with the MPI ranks that are supposed to do I/O – here rank 0 and 4. Usually, you would use here the rank with the smallest ID on each node in the job. *For a 100 node job with 50 CPU cores per node, this can reduce the number of parallel write-out directories (and proportionally also files) from 5000 to 100.*

decomposePar creates bundled directories with for example the name scheme of ***processors8_0-3*** – meaning 8 MPI ranks, and ranks 0 until 3 write herein into the files together.

Also the solver requires to be called with the **-fileHandler collated** option. Alternatively, if you don't want the command-line flags, you can simply add the following to the ***controlDict***.

```
OptimisationSwitches {
    fileHandler collated;
    maxThreadFileBufferSize 2e9;
    maxMasterFileBufferSize 2e9;
}
```

For more information, please consult the OpenFOAM documentation.

As an *exercise*, use one of the ***tutorial*** cases, which can be parallelized (*incompressible* → *simpleFoam* → *motorBike* is large enough), and play with the settings!

Check-Pointing

For larger cases, when running on many nodes, it might become relevant that node and system failures occur. This usually aborts a running job, where one loses all computations done since the last write. That is, CPU budget is wasted here. One hour on 1000 nodes with each about 50 CPU cores, for instance, this makes already 50000 CPU-hour. For e.g. 2.5 cents per CPU-hour, this is already 1250 €. So, CPU budget wasted is money wasted!

One can compensate for this by doing check-pointing – i.e. writing restart states to disk/file system, from which you can restart the simulation.

However, check-pointing takes some time[24]. And possibly, in HPC, a quite substantial amount of time. So, there is maybe a trade-off between check-pointing often in order to avoid loss, and check-pointing rarely in order to use more time for the simulation itself than for I/O.

24 Unless one can parallelize the I/O with the ongoing computations. But OpenFOAM can't.

One may consider some heuristics to find an optimal time interval for check-pointing. If you assume an exponential failure rate for the nodes, one can state the survival probability for a job with N nodes and a run-time of t_c of the computation as

$$P_N^s(t_c|\lambda)=e^{-\lambda N t_c}$$

where λ is the mean failure rate of one node. Is the computation time t_c too large, the probability of a failure grows. The larger the job (the larger N), the more severe is the issue.

On the other hand, if we let T_c be the time needed for a check-point writing, we may like to maximize the ratio

$$\frac{t_c}{t_c+T_c}$$

So, optimizing

$$w(t_c)=\frac{t_c}{t_c+T_c}e^{-\lambda N t_c}$$

with respect to t_c, results in

$$\hat{t}_c=\frac{T_c}{2}\left(\sqrt{1+\frac{4}{\lambda N T_c}}-1\right)$$

As an example, for a $N = 786$ node job, with a check-point duration of $T_c = 5\ min$, the optimal interval between two check-points would be around four hours, for λ corresponding to about 15 years, what is quite realistic.

In OpenFOAM, if you are just out to keep some limited set of the written case history, say for instance the last two time-steps as check-points, one sets

```
purgeWrite 2;
```

in the ***system/controlDict***. This keeps the last two time-step data. If a new time-step is written (underlying the normal **`writeControl`** and **`writeInterval`** settings), the oldest one is deleted.

Btw., on HPC systems, too, you usually have access to the case directory, even of a running job, through the file-system. If you have set

```
runTimeModifiable true;
```

in the ***controlDict***, you can set also values here during run-time. With this flag, the ***controlDict*** is read after each computation step. You could e.g. increase the frequency of write steps, or stop your simulation in a well-defined shutdown by just stetting values and save the file. That’s maybe not a super fancy job control. But very effective, flexible and powerful. As usual, however, with great power … : Be careful how you use it. Crashing an otherwise successfully running job by inadvertent ***controlDict*** modification can be really annoying. But as we are in the expert section of the tutorial, we assume you know what you do.

Function Objects – In-Situ Analysis

Function Objects are another facility in OpenFOAM to create additional information and data, which can be used to minimize persistent output data. This can be used also during the post-

processing step. But some information is maybe desired for all time-steps, not only at the two last ones (see **purgeWrite** above).

For instance, you want the *vorticity* or *Q criterion* of a field because you are interested in, or *minimum-maximum* for some monitoring purposes. There are already a lot of functions available. What exactly, you can figure out via

```
> postProcess -list
```

postProcess is also the tool, with which you can apply these function objects after the simulation (on data that are still available – and even in parallel). If you wish to apply this function object during the simulation, you can put a section into the ***controlDict***. For instance,

```
functions
{
   minmax_psi
    {
        type                fieldMinMax;
        fields              ("psi");
        libs                (fieldFunctionObjects);
        executeControl      timeStep;
        executeInterval     1;
        writeControl        writeTime;
    }
}
```

writes out the *minimum* and *maximum* of the *psi* field (and the positions where they occur) at each time-step to the standard output. But also into ***uniform/functionObjects/functionObjectProperties*** in each time-step folder. And also collected/summarized for all time-steps in ***postProcessing/minmax_psi/0/fieldMinMax.dat***. This may somewhat depend on the type of information the function object writes out.

As with solvers, you can write your own function objects. The easiest way is again to copy the OpenFOAM provided function object library, and rename everything as you like.

```
> cp -r $FOAM_SRC/functionObjects/field myFOLibs
> cd myFOLibs/
```

Change in ***Make/files*** again **FOAM_LIBBIN** to **FOAM_USER_LIBBIN**, and **libfieldFunctionObjects** into **libmyFOLibs**. Throw out (delete) what you don't need or want, consistently inside ***Make/files*** and in top-level directory – e.g. remove everything except for ***fieldMinMax*** and ***Make***. Change what you want to have changed in the remaining files – begin with the class names – consistently. For instance, we rename everything as **myFieldMinMax**. Then again

```
> wmake
```

until no errors occur anymore, and the library ***libmyFOLibs.so*** is build in your **$FOAM_USER_LIBBIN** folder.

Finally, use it. Replace in the ***controlDict*** the corresponding labels

```
type myFieldMinMax;
libs (myFOLibs);
```

That’s it. If done correctly, you now see successful output of your function object, and can start modifying it further for your needs.

Remark: We do not go much into detail here, but with the catalyst interface, OpenFOAM can also perform some in-situ visualization using Paraview/VTK during the simulation.[25]

Also *paraview* could be used to do the post-processing. And if you have to write out anyway all time-step results, it is maybe even the more convenient tool to do this. We later show some workflow examples.

For HPC, please remember the words about *preparation* at the beginning of this part. A lot of problems – specifically ***big data*** problems – can possibly be avoided by careful ***data economics***.

Selected HPC Topics and Workflows

This chapter is meant as support for the parallelism efforts in HPC, which become more important the larger your ambitions – meaning your resource requirements of your use cases.

Monitoring, Profiling, Debugging – Automation

As you could see, OpenFOAM has a very open and accessible structure of the cases. You can look into them during the run-time (if you want, everything is in ASCII), and you can modify things during run-time (dictionaries, specifically). This might scare most of the users at the beginning. But it represents also a unique chance.

Next to the normal standard output of a simulation, you can acquire more information by putting more debugging flags into the corresponding **`DebugSwitches`** section of your ***controlDict***, which takes effect almost instantly when you have set **`runTimeModifiable true;`** there.

Specifically for large jobs, running in a dark-center (HPC cluster), having more information for debugging – in case something goes wrong – is always better. But usually not anymore, if performance becomes spoiled thereby. So, systematics is important to handle this trade-off.

What can happen?

1. Immediate abort – with some error message.
2. Delayed abort – with some error message.
3. No (apparent) progress at all – no output to terminal or to file.
4. Everything seems right – on a first glance.
5. Your simulation takes much longer than expected.

1. would be great. Usually, the interpretation of error messages is not always easy. But failing early saves a lot of time. You can try to classify the error, and take counter measures – and retry. Often, OpenFOAM outputs some ominous *File IO*-error. This mostly means that something is wrong with the input files (dictionaries or the mesh). Look for more information. Maybe increase some debugging level via the switches mentioned above. Also file encoding errors when transferring files from Windows to Linux are possible … (Sigh! Yes. In the 21st century.)

25 Where to find the *catalyst* library and *paraview*/*VTK* interface in the OpenFOAM source and installation changes from version to version! Best is to first search in the sources/applications/modules for *catalyst*. There, you also may find some *tutorials*, which illustrate the usage.

If you assume some immediate numerical problem, reduce the time-step size. However, this would mean that you already reached the time-step integration phase. So, reading the dictionary and the mesh and initial data was already successful. That's at least soothing.

If you suspect some problem with the solver itself, use one of the corresponding tutorial cases. If they do not run, you know already much more – even if debugging the solvers is by no means simpler.

2. is not so nice – and some waste of time. But it is still a demonstration that something works at least in principle. Of course, there might always break some hardware, like a node or a network connection. But fortunately, you have some check-points, and can restart from the last check-point. If the problem then recurs, the probability is high that you met some problem of numerical stability. Check the Courant numbers and residuals. Use e.g. **`foamMonitor`**, where suitable for monitoring the residuals as function over time (in order that is worked, *Gnuplot* must be available – installed, and executable from the shell).

3. is not nice. No reaction at all. You must increase the output level! Figure out where the program stops, or hangs, or idles. Sometimes you may hit some real program issue like a dead-lock. But in order that a developer can fix it, one really needs to identify it, and create a *100% reproducing test-case*. We categorize such a case as rather improbable for OpenFOAM from our experience.

4. is probably the most devious case. Well, we hope that if no errors occur, there runs everything fine. But you should go sure! Some parameter set wrongly is easy to accomplish. You seriously don't want to figure out this after hours, possibly days or even months of simulation!

You can look into the processor directories, and check whether the time-step output happens regularly – at least you can see when the last output was written (if at all). Using **`reconstructPar`** on a specific time-step, and analyzing it using *paraview* is always possible (see below for some HPC *paraview* workflows). If something is not as expected, you can always intervene, and in final consequence also cancel your job.

Having the feeling of control is very important. HPC is expensive! So, double-check!

5. already assumes that you really did some systematic scaling tests. So, you have a realistic expectation of your anticipated time-integration progress, and there should be no sudden surprises! Question is then what has changed since? Often, this question is not easy to answer. System and IO performance degradation definitely can occur. But before contacting the administrators, be sure that this is a real performance drop on the system! For the first time of your simulation, your expectations might have been wrong. Double-check! If you are sure that a simulation before was faster, prove it! Collect performance data for a more or less permanent surveillance – Choose some reasonable and convenient, but still expressive metrics!
Sometimes, performance may change only slowly over time.[26] Also document when you changed some parameters in your case! We mentioned some version control like *git* that can support you here. Try to correlate your changes with the appearance of your performance change observation.

If you use adaptive mesh methods, reconstruction and decomposition of the mesh from time to time might be necessary to keep some level of workload balance among the MPI ranks. External

26 Btw., the system CPU frequency settings may change intentionally according to some HPC admin's policy.

coupling of OpenFOAM with other software may also result in performance degradation (or, in worst case scenarios, in numerical instabilities). Here, some special care must be exercised.

OpenFOAM also can be build with profiling or debugging compiler flags, as we already mentioned earlier. However, unless you are a developer of OpenFOAM (not just a solver or utility programmer, or even only a user!), this won't really help you. And the vast majority of problems we have seen so far are issues caused by wrong handling, unsystematic approaches to HPC workflows, or real system problems (MPI, IO, OS). The latter ones are usually attempted to be caught by the HPC system maintainers. But hints and help from users are welcome, of course.

Last but not least, we vote for automation. OpenFOAM already comes with some bash and python (pyFoam) scripting interface. Although it needs time to get used to it, HPC is all about automation – specifically the workflows. It is faster (in the sense of workflow development), more efficient in resource exploitation, less error-prone, self-documenting, repeatable, extensible, … Please look into the ***Allrun*** scripts of the tutorial cases! They are intuitive enough, even if you don't decide to become a bash guru. Using these scripts can already be the first step for a job script (or part of a sub-job script – see below).

But keep in mind that some steps are notoriously serial. Try to judge whether a login node is maybe better than hundreds of nodes idling. Not everything can be automated on the cluster with efficiency.

Parallelism-Resistant Workflows

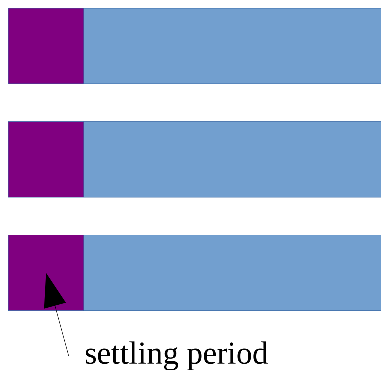
Some of the current CFD workflows cause some problems on HPC systems, when they are implemented as just moderately parallel, long-term simulations – such as, for instance, turbulence averaging studies in DNS. They block some smaller resources for a long time. Other larger jobs (with more nodes) just may not be able to start – even if all the other nodes are idle.

In HPC, one should always have a look for opportunities of more parallelism, instead of occupying resources for a rather long time period.

Please, consider if time-averages can be replaced by ensemble-averages, or a combination of both with much shorter time-averaging periods. The advantage is that the ensemble-averages are independent of each other, and can run in parallel.

serial

parallel



The idea would be – according to the Ergodic Hypothesis – that the smaller scale turbulence does not change substantially the averages. This means, different turbulent flows with slightly different initial conditions should lead to the same averages. So, randomly disturbing the initial conditions of the different samples in the ensemble (maybe perturbations underlying some sort of control to keep conserved quantities at the desired values) would lead to a set of time-averages y_i, with some uncertainty (standard deviation) σ_i (index i denotes the sample value of a simulation with one specific initial condition in the ensemble), which can be combined via

$$\hat{y}=\frac{\sum_i w_i y_i}{\sum_i w_i} \quad , \quad \hat{\sigma}^2=\left(\sum_i w_i\right)^{-1} \quad \text{with} \quad w_i=\frac{1}{\sigma_i^2}$$

A problem might occur if the initial settling phase until a stable global turbulent flow takes quite a long time. This phase cannot be parallelized much. Still, there are maybe ways to accelerate, and thus shorten it – even if an extended parallelism is not possible. As the details of this initialization phase are not of much interest, one can skip all types of elaborate analysis on these, and reduce the I/O to some minimum required. Multi-grid methods, and also other pre-conditioning techniques, can be used to settle the flow as fast as possible. Once this stage is reached, small perturbations can then be imposed for each of the ensemble members differently (randomly). The subsequent settling phase is presumably much shorter, and one then has many samples running in parallel, which is comparable in workflow handling as described in the next section.

There are possibly more such workflows that appear to resist parallelism. HPC and parallelism are hard work. Maybe there is a way if you try harder. But it should be accepted when there is possibly no way. This then is probably not a viable case for an HPC system, yet.

Uncertainty Quantification and Parameter Studies

Uncertainty quantification and parameter studies are two use cases, where one usually does not have large cases, but many smaller ones, which in the sum are as resource hungry as a single big case. In both these use cases, the input set might be more or less the same, and only certain model or geometry parameters, or initial or boundary conditions, change from run to run. But again, the cases for specific parameter sets can run independently, and therefore also in parallel.

OpenFOAM does not really provide a framework for this. But by means of bash, python, or any other scripting language, it is more or less easy to realize it manually. These efforts can be subsumed as ***job*** or ***task farming***.

We showed already in a previous chapter how reconstruction steps can run in parallel. The major difference now is that the OpenFOAM solvers can work already in parallel. So, the question is: *Should I create many thousands of jobs, and let the resource scheduler do all the resource management work? Or is it my own responsibility to manage the resources in a single big job?*

From an HPC cluster administrator's point of view, clearly the second! From user's perspective – well, the first one – we guess. So, how can we put this together?

Fortunately, Slurm offers some capabilities to let users also schedule smaller jobs in bigger jobs. **`srun`** is capable to be called inside a **`sbatch`**/**`salloc`** submitted Slurm job.

What's to do for the user? Two things: (1) analyze your tasks, (2) write a job script! Well, and finally run the job and monitor the advancement, … and possibly debug occurring problems.

(1) is simple. Take one example task (with a fixed parameter set), and let it run. Check how much memory it requires, and how many parallel MPI ranks are reasonable. This determines the *pipe-line structure* of your job-farming job later. Meant by this is that all the smaller jobs are supposed to require similar resources – *memory* and *run-time*. So it makes possibly sense to distribute the big job over as many nodes *N* such that *M* smaller jobs can run in parallel – exploiting as much of the resources (CPUs or memory) as possible. If a smaller job runs for a time period of about *T*, and you have *K* such small jobs in total, then your single big job with *N* nodes would need to run about

$$t_{\text{job}} = \frac{K}{M} T$$

Example: $K = 100000$ small jobs of mean run-time of $T = 5$ *minutes*, on $N = 200$ *nodes* with say $M = 400$ small jobs in parallel (one small job occupies half a node) would last about $t_{\text{job}} = 21$ *hours*. That's quite acceptable when considering that 100000 five-minute jobs running in series otherwise would last for more than a year.

(2) Implementing this is a bit more difficult, of course, and depends to some extent on the HPC cluster your are using. We indicate here one viable way with Slurm. The best is to bundle the workflow of a small job/task again into a script, where you parameterize the input data set again on some environment variable. **`srun`**, for instance, defines the **`$SLURM_STEP_ID`** environment variable. By it you can e.g. parameterize the line of an input file, or a database entry.

The main job script may look roughly like this.

```
#!/bin/bash
#SBATCH Header
# job environment setup

task() {
   srun <resource-requirement-spec> <subtask> &> log.$1
}

export -f task

parallel -P <M> task ::: {1..K}
```

Here, we use again GNU parallel. It is very convenient. *M* and *K* are the number as above. So, GNU parallel executes exactly *M* tasks simultaneously out of the range 1 … *K*. Here, you can adapt the range – e.g. if you only have to re-run the last half of tasks.

The **`<resource-requirement-spec>`** specify the single small jobs CPU and memory requirements. They must be specified such that Slurm puts them on independent resources. That's not always trivial, e.g. if you want only some few CPUs per node for a subtask, and so some of the

subtasks would run on CPUs of two different nodes. The best is first to start with some dummy to check the correct MPI task placement.[27]

`<subtask>` is some sub-task command or script executing the single case workflow in one directory (possibly creating it before, and copying input files there). It can also be a Python script. You can give **`$1`** as a parameter, if your script accepts it – and maps it to some input parameter set.

What is not included is sort of a bookkeeping, which of the jobs already ran successfully, and which not. This is highly relevant if your big job fails prematurely before finishing, and you need to restart. Maybe best is not to restart the subtasks where they where interrupted, but at their beginning. And jobs that already succeeded might set a **`lockfile`**[28], which can be checked by the subtask script. And if it is there, the script just finishes immediately.

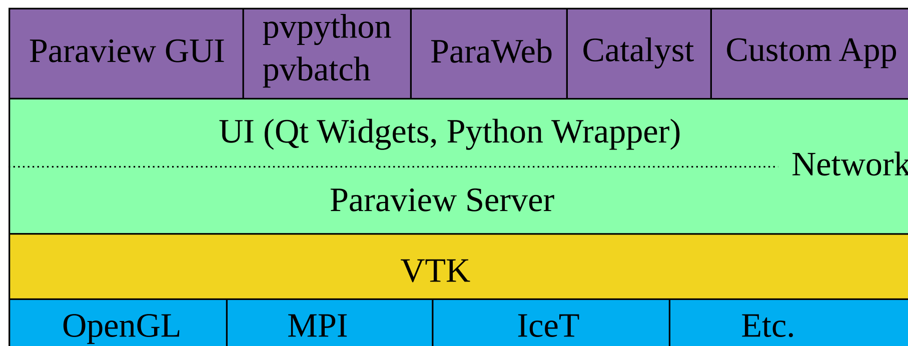
More sophisticated methods, but also more complex ones, might include some database. There are but also already ready-to-use frameworks for job-farm processing (e.g. radical cybertools, or flux).

HPC Post-Processing

We would like to revisit *paraview* at this place, as it is made for big data and HPC, in fact. It achieves this capability by a somewhat more complex construction, sometimes not easy to comprehend by beginners. Furthermore, it involves network communication, which is an extra complication for HPC beginners. But there is no need to be scared.

For the following, we assume that you have already some familiarity with the *local paraview* workflows. That is, you can start the *paraview* GUI, load some locally residing OpenFOAM case, apply some filters, and render some nice images. Essentially, this is always the same workflow, also for what comes next.

The architecture of *paraview* looks roughly as follows.

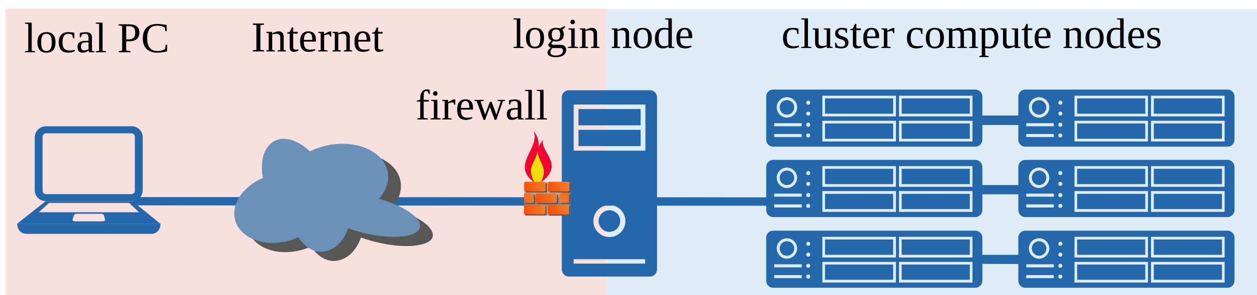


Paraview GUI, and other tools are build on top of the *paraview* Server, which essentially wraps mostly VTK (*Visualization Tool Kit*). This client-server architecture is the necessary prerequisite for the parallelism. Luckily, a user will mostly be agnostic about how exactly this parallelism works. *Paraview* does already a lot automatically for us. The *paraview* Documentation devotes a complete chapter for Parallel Data Visualization. Please take it as reference!

1. Remote Visualization

When working on HPC systems, your situation is roughly as follows.

27 **`srun`** knows a **`–mpi`** option, just for the case you wonder where **`srun`** knows from, which MPI it should use!
28 See for instance the lockfile man page.



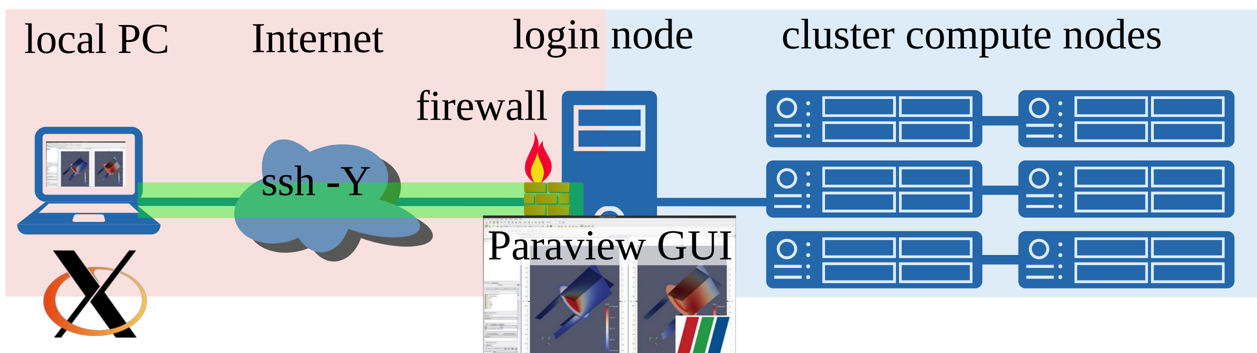
You have your *local PC* or *laptop* somehow connected to the *internet*, via which you can reach the HPC center's *login node*. For security reasons, these servers (login nodes) are secured by a *firewall*, such that you can access them only with special tools (browser for web-servers, SSH client for SSH-servers, …).

Behind these login nodes, you find protected the cluster with its *compute nodes*. They are not directly accessible, usually, but only via some RSM (*Resource Scheduler and Manager*) – e.g. like Slurm. This is necessary for some optimal (efficient) resource management and a fair distribution of the resources among many users.

Still, users can also start some server on the compute node (via the RSM), and connect to them via network. Login and compute nodes are connected to the network file system (GPFS, for instance), where the large simulation data are located, which you don't want to move/copy.[29]

So, the question is, how to best analyze these data?

If you want to interactively analyze the data, you need to start a *paraview* GUI somewhere. There are actually only two places – on your local PC, or on the login-node (e.g. within a SSH X-Forwarding[30], or VNC server – for the latter you need a VNC client on your local device).



a) You start the ***GUI on the login node***, you have immediate access there to the data. With X-Forwarding, you see the GUI on your local monitor, and can immediately start the analysis. That's the most convenient solution. However, you are mostly limited to the resources on this node, which you share with other users.

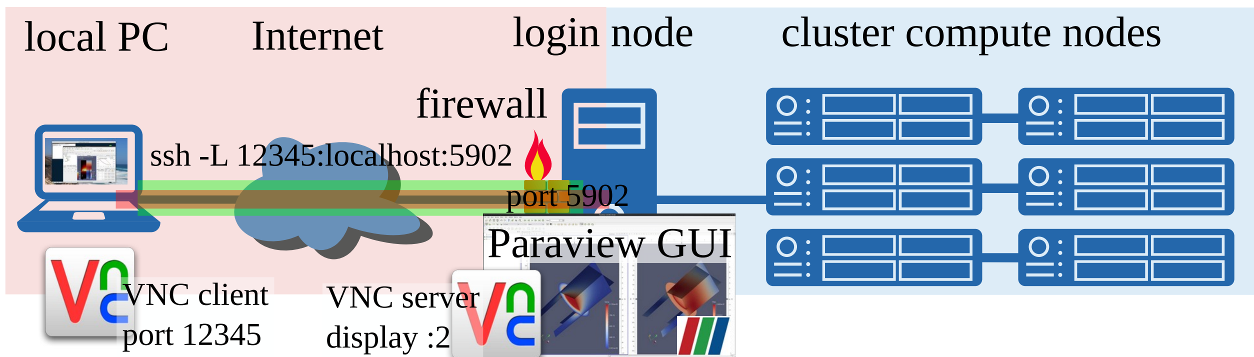
This can be accomplished (with local X-server running; *Xming* for instance under Windows) via

```
local> ssh -Y myusername@login     # credentials are required
login> module load paraview        # check before available paraview modules
login> paraview
```

`myusername` and **`login`** must, of course, be replaced by your *user ID* and the correct *login server's name*, respectively.

29 We really assume here that you have large amounts of data now. So, these workflows are borne out of necessity instead of convenience.

30 That is not feasible, if you have no local X-Server running. And it is not really a convenient solution, if your network connection to the login node is slow.



b) You start a ***VNC server remotely***, and are connected with your ***local client***, you should see a *remote desktop session* – sort of. You can open a terminal therein like on your local device, but your *are* remote. Now, you can load there a *paraview* module, and start the *paraview* GUI. The rest of the analysis workflow is the same as under a).

In order to create such a VNC session, there are some things to do, however, rendering this workflow a bit more complicated than simple X-Forwarding. The advantage is that you can interrupt the connection to the VNC server, and reconnect later. For unstable network connections, that's definitely preferable than to lose the work accomplished so far inside of the session. Furthermore, VNC connections are usually faster and the interaction more responsive than X-Forwarding, because only the graphics needs to be send to you local device.

Prerequisites: a local VNC client (TigerVNC offers standalone VNC clients), a remote VNC Server (with a window manager; must be provided by the cluster administrators).

1. To *start* the remote server, *ssh* to the login-node, and start the VNC server.[31]

```
local> ssh myusername@login
login> vncserver
    […]
New 'login:1 (myusername)' desktop is login:1
```

You can also start the server with a definite *display number* via

```
login> vncserver :3
    […]
New 'login:3 (myusername)' desktop is login:3
```

This display must not already be occupied by another user, of course. You can also check whether you have already a VNC server running on this login node via **vncserver -list**, and just re-use it.

2. The *next step* is to open a SSH port-forwarding tunnel, connecting a remote port to a local port. This is the way to let servers and clients communicate through a secure SSH connection. There are several ways to setup this forward tunnel – depending a bit also on your local SSH client. The most general one that seems to work with any client is to switch to the *SSH command line shell*.

```
login>      # press escape sequence ~C such that these characters are not visible in the shell[32]
ssh>        # you are now in the SSH command line shell
```

To set up a forward tunnel is just **-L <local-port>:remote-name:<remote-port>**. For the **local-port**, just choose some number larger 1024 and smaller than 65536, which is not yet occupied on your system. For example, we take here **12345**. The **remote-port** is that port, where your VNC server is listening on. It is always 5900 plus the VNC server's display number. Above, we chose **:3** as the display. So, this server listens on port **5903**. Stays to figure out **remote-name**. We will see later that we can take here a server name, which is different from the

31 Calling **vncserver** the first time, you are required to set a VNC session password. Don't leave it empty! You can always change it later via **vncpasswd**. But changes become effective only after a VNC server was restarted.

32 Possibly press Enter once to bring the shell into a state where it accepts this escape sequence.

login node. But as the VNC server runs on the login node, we can simply use **`localhost`** (or the local IP address of the listening network card)

```
ssh> -L 12345:localhost:5903
```

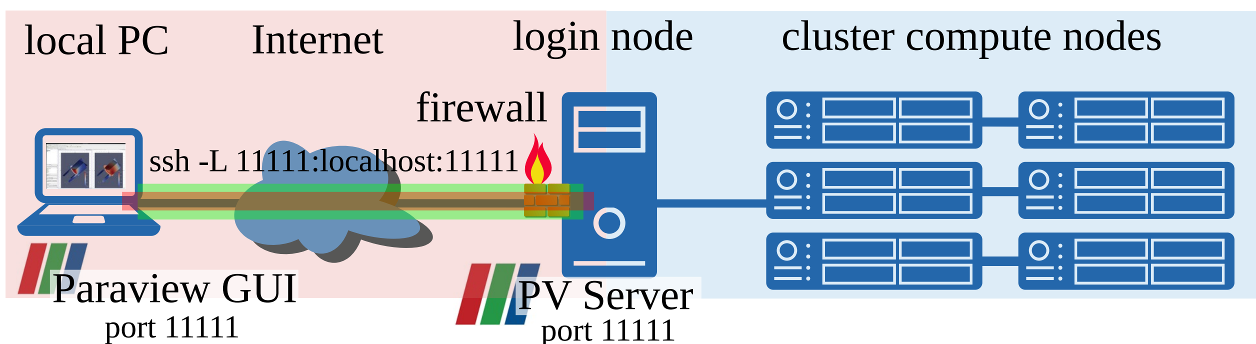
This tunnel will be removed when you close the SSH connection, where it is attached to.

3. In the *final step*, open the VNC client on your local device, and connect to **`localhost:12345`**. This *localhost* is now indeed your local device. If you find this confusing … Well, it is. After entering the VNC session password (you hopefully did not forget, yet), you should see the normal window manager GUI, where you can now work.

Once you finished, ***please stop the VNC server*** again.

```
login> vncserver -kill :3
```

As you can have several VNC servers running in parallel, you must address the one you want to get killed. Switching off the VNC server should be done for security reasons. ***Don't leave resources running unattendedly if you don't need them!***



c) ***Paraview*** also allows directly a ***server-client connection***. In fact, this is always used by *paraview*. But you can ran the *paraview* GUI on one device (e.g. on your laptop) and the *paraview server* on another (e.g. on the login node). In order that this works, you must have the ***local and remote paraview*** of the ***same version***. The workflow is as follows.

1. Connect to the remote system

```
local> ssh myusername@login
login> pvserver
Waiting for client…
Connection URL: cs://login:11111
Accepting connection(s): login:11111
```

We recommend to start the **`pvserver`** from the ***OSMesa installation*** in order to also have remote rendering capabilities. Per default, **`pvserver`** listens on port 11111. We will just need it to set up a SSH forward-tunnel. You can change this port via **`--server-port=`**<port-number>, for instances

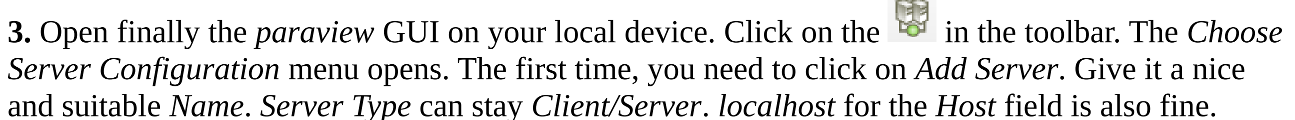
```
login> pvserver --server-port=11112
```

Choose one free port. If a port is already occupied, you will be informed.

2. Set up the SSH forward tunnel. As above, switch to the SSH command line shell, and insert the port. For instance, for our second example

```
ssh> -L 11111:localhost:11112
```

The local and remote ports really don't need to be the same.



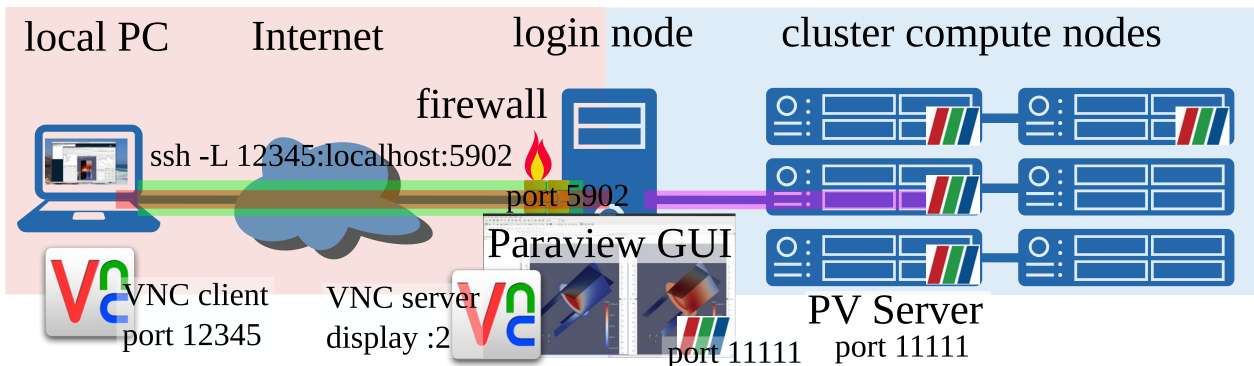
3. Open finally the *paraview* GUI on your local device. Click on the in the toolbar. The *Choose Server Configuration* menu opens. The first time, you need to click on *Add Server*. Give it a nice and suitable *Name*. *Server Type* can stay *Client/Server*. *localhost* for the *Host* field is also fine.

Finally, enter the local port 11111 (or which you chose) into the *Port* field.
Configure → *Save*. Ready. Now, you can click on *Connect*.

In the SSH terminal, you should see something like **`Client connected.`** You can open the *View* → *Memory Inspector* to see how much memory *paraview* consumes locally and remotely. You can now open the OpenFOAM case as usual. In the file open menu, you see the remote file system, and can navigate to the OpenFOAM case.

When you close the *paraview* GUI, or you *disconnect* (tool menu), the remote *pvserver* stops.

Don't wonder that also this mode is as fast as your internet network bandwidth permits.



d) You can combine essentially the workflows of b) and c). So, first you can setup a ***VNC session on the login node*** as under b). Then open a ***Paraview GUI on*** that ***login-node*** (within the VNC session), which you now connect to a ***pvserver running on compute nodes*** – much in the same way as described in c). You just don't need any further SSH tunnel, because compute and login nodes are not separated by a firewall.

The real fun comes now from the fact that you can run the **`pvserver`** MPI parallel. So, after setting up the VNC session and starting the Paraview GUI, you submit a Slurm job with the following content.

```
#!/bin/bash
#SBATCH Header
# job environment setup
mpiexec -ppn 4 pvserver
```

This is for e.g. 4 MPI ranks per node. You can also adjust the Slurm *SBATCH* header accordingly, and **`mpiexec`** might react on those settings. Just submit it via

```
login> sbatch pv-job.sh
```

When the job is running, you can check the output of it for where the server runs. Take this compute node's name, and the port number (here just 11111 – the default), and connect your GUI with that server.

Disconnecting the server will automatically stop the *pvserver*, and thus also the job.

If you really cannot get any VNC server running, you can also implement solution c), but with the ***pvserver*** running ***on the compute nodes***. Submit the *pvserver* job as before, and figure out, on which node it is running (and at which port). Create a SSH tunnel.

```
-L 11111:compute-node:11111
```

Here, **`compute-node`** must be replaced by the name or IP address of the compute node where the *pvserver* was started. Finally, open your local *paraview* GUI (on your local device), and connect to **`localhost:11111`**.

Again, don't wonder if that's slow. Yet, to not have to copy the data might outweigh for the loss of speed.

2. Remarks to Parallel Visualization.

In order to really benefit from parallelism, you should check in the ***Memory Inspector*** that memory is uniformly occupied. If not, you run specifically for large cases in the problem of load imbalances and, in the worst case, into a *out-of-memory* (OOM) situations, in which the system simply kills your processes.

The *paraview* Guide hints on that you should setup the ***visualization pipeline*** in such a way that you ***reduce the amount of data*** as fast as possible. That's the more true for big amount of data.

The decomposed OpenFOAM cases are indeed read in parallel by *paraview*.

We already mentioned that on the login node there is usually no X-server running, and that you should use the ***pvserver build with OSMesa***. This holds true specifically for the compute nodes. Offscreen (software) rendering is usually slower than hardware-supported rendering. But when no hardware is at hand, that's the best you can do.

The parallel visualization might produce spurious graphics effects and artifacts at the boundaries of the domains, which represent the volumes for each MPI rank. Specifically for volume rendering, this might have detrimental and ugly effects. The ***D3 filter*** is a possible solution. You can define some ghost cell layers, which prevent from those artifacts.

3. Non-Interactive Visualization (pvbatch)

When you want to do some visualization repeatedly – possibly just with different input data sets – *paraview*'s (actually VTK's) pipeline architecture is exactly designed for this purpose. This means that you can define a visualization pipeline, and apply it to different input data sets.

The *paraview* GUI can be used to visually setup this pipeline. You can save it also as Python script, which can be manipulated and executed via **`pvpython`** or **`pvbatch`**. The latter has the advantage of being MPI parallelizable. So writing a job script with

```
mpiexec pvbatch some_pv_script.py
```

is actually quite the same as an OpenFOAM simulation – from a HPC point of view – meaning you just start a MPI parallel program to process some input, and to create some output.

Such script can also be created in a different fashion. In the *paraview* GUI, you can click *Tools → Start Trace*, and go on with loading the data, and setting up the visualization pipeline. You can even save the picture/animation. Just do that on a small data set in order not to waste too much time with waiting for *paraview* to finish working. When you finished, just *Tools → Stop Trace*. A window will pop up with a lot of Python code – the captured trace of what you just did within the GUI. You can save this trace, and manipulate it with an editor, and – right – call it again with **`pvbatch`** on your actual data sample to be analyzed on the cluster.

4. Remark: OpenFOAM to VTK and paraFOAM

OpenFOAM provides the tool **`foamToVTK`**. It transforms the OpenFOAM output into the native VTK file format, which is often better suitable for parallel handling. Executing **`foamToVTK`** in the OpenFOAM top level case directory just creates a ***VTK*** directory, with a bunch of *.vtm* files, which can be opened in *paraview* natively.

Again, you have some fine control over which time steps or regions you want to transform into the VTK format. And, it can run in parallel. Consult the output of **`foamToVTK –help`** for more information!

Remark: **`paraFoam`** can also be used if you really need or want to. Essentially, it is a script, and it creates just a *.foam* file, and then calls **`paraview`**. In order that this works, **`paraview`** must be findable in the **`PATH`**.

Addendum

Useful Links and Literature

The following list is by no means complete. Nor is it meant for a *serial read through*! Only if you feel that you need help, please consider to consult these references. And use the search capabilities of web pages to navigate to the place you desire your answer from.

In most cases, Google will be a good first starting point when having concrete questions/issues.

Bash/Shell/Linux

- [TLDP] Bash Beginner's Guide
  [TLDP] Advanced Bash-Scripting Guide
- [GNU] Bash Reference Manual
- [GNU] Software
  [GNU] Coreutils User Guide

More tools can be involved like sed, awk, gnuplot, python, … But is not mandatory.

C++

- [cpp reference] C++ online reference
- LRZ C++ Beginner’s Course
- [ESI OpenFOAM] API Guide
- [Foundation OpenFOAM] C++ Source Code Guide

CAD

- [Blender] (docu modelling)
- [FreeCAD] (docu)

Environment Module

- https://modules.readthedocs.io

OpenFOAM

- [Foundation OpenFOAM], [ESI OpenFOAM]
- [Youtube] CFD by József Nagy
- [Github] Basic OpenFOAM Programming Tutorials
- [PDF] OpenFOAM Programmer's Guide
- [Youtube] OpenFOAM Tutorials
- [Wiki Openfoam] Introduction to OpenFOAM (Kenneth Hoste, Hrvoje Jasak)

- Tomislav Maric, Jens Höpken, and Kyle Mooney. *The OpenFOAM Technology Primer*. sourceflux, 2014.
- F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics - An Advanced Introduction with OpenFOAM ® and Matlab ®*. Number 113 in Fluid Mechanics and its Applications. Springer, 2016.
- [Wiki Openfoam PDF] A Comprehensive Tour of snappyHexMesh
- [PDF] User Guide cfMesh

Paraview

- [Paraview] Download (Software/Guide/Tutorial)
- [Youtube] Introduction to Scientific Visualization with ParaView
- [Youtube] Paraview Postprocessing
- [Youtube] Advanced Scientific Visualization with Paraview (CSCS course 2019)
- [Youtube] Advanced Scientific Visualization with Paraview (CSCS 2020)
- [Paraview] User Guide
- [VTK] User Guide

PDEs/CFD

- [Youtube] Fluid Mechanics 101
- Pijush K. Kundu, Ira M. Cohen, and David R. Dowling. *Fluid Mechanics*. Elsevier, 5. edition, 2012.
- Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2003.
- Joel H. Ferziger and Milovan Peric. *Computational Methods for Fluid Dynamics*. Springer, 3 edition, 2002.
- [BLOG] How not to mesh it up (Meshing Guidelines)

Slurm

- https://slurm.schedmd.com/

SSH

- LRZ SSH Tutorial

VNC

- [Wiki] VNC
- TigerVNC (offers stand-alone client (and server) for all OS; no admin rights needed)
- noVNC (VNC through a browser)

Example Files

Gmsh Script of the Kármán Vortex Sheet Example

```
//- Parameters -//
channelWidth=10.0;
channelLength=25.0;
meshNumber=50;
focalProgr=1.1;
//--- Do not edit!
halfChannelWidth = channelWidth/2.;
wakeLength=channelLength-halfChannelWidth;
//- Points -//
Point(0) = { 0.0, 0.0, 0, 1.0};   // center point
// obstacle points  (0.5*cos(45 deg)=0.5/sqrt(2)=0.353553591)
Point(1) = {-0.353553591,  0.353553591, 0, 1.0};  // top left
Point(2) = { 0.353553591,  0.353553591, 0, 1.0};  // top right
Point(3) = { 0.353553591, -0.353553591, 0, 1.0};  // bottom right
Point(4) = {-0.353553591, -0.353553591, 0, 1.0};  // bottom left
// channel boundary points
Point(5)  = { -halfChannelWidth, halfChannelWidth, 0, 1.0};   // top left
Point(6)  = {  halfChannelWidth, halfChannelWidth, 0, 1.0};   // top right
Point(7)  = {  halfChannelWidth,-halfChannelWidth, 0, 1.0};   // bottom right
Point(8)  = { -halfChannelWidth,-halfChannelWidth, 0, 1.0};   // bottom left
Point(9)  = {        wakeLength, halfChannelWidth, 0, 1.0};   // wake top right
Point(10) = {        wakeLength,-halfChannelWidth, 0, 1.0};   // wake bottom right
//- Lines -//
// outer channel boundary lines
Line(1)   = {5, 6}; Transfinite Curve {1}   = meshNumber Using Progression 1;   // top
Line(2)   = {6, 7}; Transfinite Curve {2}   = meshNumber Using Progression 1;   // inner outlet
Line(3)   = {7, 8}; Transfinite Curve {3}   = meshNumber Using Progression 1;   // bottom
Line(4)   = {8, 5}; Transfinite Curve {4}   = meshNumber Using Progression 1;   // inlet
Line(13)   = {9, 6}; Transfinite Curve {13} = meshNumber Using Progression 1;   // top wake
Line(14)   = {10,9}; Transfinite Curve {14} = meshNumber Using Progression 1;   // wake outlet
Line(15)   = {7,10}; Transfinite Curve {15} = meshNumber Using Progression 1;   // bottom wake
// inner non-boundary lines
Line(5)   = {1, 5}; Transfinite Curve {5} = meshNumber Using Progression focalProgr;   // top left
Line(6)   = {2, 6}; Transfinite Curve {6} = meshNumber Using Progression focalProgr;   // top right
Line(7)   = {3, 7}; Transfinite Curve {7} = meshNumber Using Progression focalProgr;   // bottom
right
Line(8)   = {4, 8}; Transfinite Curve {8} = meshNumber Using Progression focalProgr;   // bottom
left
// obstacle lines
Circle(9) = {  1, 0, 2}; Transfinite Curve  {9} = meshNumber Using Progression 1; // top
Circle(10)= {  2, 0, 3}; Transfinite Curve {10} = meshNumber Using Progression 1; // right
Circle(11)= {  3, 0, 4}; Transfinite Curve {11} = meshNumber Using Progression 1; // bottom
Circle(12)= {  4, 0, 1}; Transfinite Curve {12} = meshNumber Using Progression 1; // left
//- Surfaces -//
// front face
Curve Loop(1) = {5, 1, -6, -9};  Plane Surface(1) = {1}; Transfinite Surface {1}; Recombine Surface
{1}; // top
Curve Loop(2) = {10, 7, -2, -6}; Plane Surface(2) = {2}; Transfinite Surface {2}; Recombine Surface
{2}; // right
Curve Loop(3) = {8, -3, -7, 11}; Plane Surface(3) = {3}; Transfinite Surface {3}; Recombine Surface
{3}; // bottom
Curve Loop(4) = {4, -5, -12, 8}; Plane Surface(4) = {4}; Transfinite Surface {4}; Recombine Surface
{4}; // left
Curve Loop(5) = {2, 15, 14, 13}; Plane Surface(5) = {5}; Transfinite Surface {5}; Recombine Surface
{5}; // wake
// back face
Extrude {0, 0, 0.1} {
  Surface{1}; Surface{2}; Surface{3}; Surface{4}; Surface{5};
  Layers{1};
  Recombine;
}

Physical Surface("FrontAndBack", 126) = {37, 103, 81, 59, 125, 5, 2, 3, 4, 1};
Physical Surface(       "Inlet", 128) = {90};
Physical Surface(      "Outlet", 129) = {120};
Physical Surface("TopAndBottom", 130) = {28, 72, 116, 124};
Physical Surface(    "cylinder", 132) = {36, 98, 46, 80};
Physical Volume(     "internal", 133) = {1, 4, 2, 3, 5};
```

Kármán constant/polyMesh/boundary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2112                                  |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    arch        "LSB;label=32;scalar=64";
    class       polyBoundaryMesh;
    location    "constant/polyMesh";
    object      boundary;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

5

(
    FrontAndBack
    {
        type            empty;                          // ← empty
//        physicalType    patch;                        // commented out
        nFaces          34810;
        startFace       34515;
    }

    TopAndBottom
    { … }

    cylinder
    { … }

    Inlet
    { … }

    Outlet
    { … }

)
// ************************************************************************* //
```

The rest was shortened for better comprehensibility. Also, your numbers might deviate from those here.

Kármán 0/p

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2112                                  |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version     2.0;
    format      ascii;
    arch        "LSB;label=32;scalar=64";
    class       volScalarField;
    location    "0";
    object      p;
```

```
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 0;

boundaryField
{
    FrontAndBack
    {
        type            empty;
    }

    TopAndBottom
    {
        type            zeroGradient;
    }

    cylinder
    {
        type            zeroGradient;
    }

    Inlet
    {
        type            zeroGradient;
    }

    Outlet
    {
        type            fixedValue;
        value           uniform 0;
    }
}
// ************************************************************************* //
```

Kármán 0/U

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2112                                  |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    arch        "LSB;label=32;scalar=64";
    class       volVectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

boundaryField
{

    FrontAndBack
    {
        type            empty;
    }
```

```
    TopAndBottom
    {
        type            slip;
    }

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

    Inlet
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }

    Outlet
    {
        type            zeroGradient;
    }
}
// ************************************************************************* //
```

Kármán system/controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application       icoFoam;
startFrom         startTime;
startTime         0;
stopAt            endTime;
endTime           150;
deltaT            0.005;
writeControl      timeStep;
writeInterval     200;
purgeWrite        0;
writeFormat       ascii;
writePrecision    6;
writeCompression  off;
timeFormat        general;
timePrecision     6;
runTimeModifiable true;
// ************************************************************************* //
```

Kármán blockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
```

```
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

scale   1;

vertices (
    (-5   5 0)                          // 0 top left front
    ( 5   5 0)                          // 1 top right front
    ( 5 -5 0)                           // 2 bottom right front
    (-5 -5 0)                           // 3 bottom left front
    (-0.353553591  0.353553591 0)       // 4 cyl top left front
    ( 0.353553591  0.353553591 0)       // 5 cyl top right front
    ( 0.353553591 -0.353553591 0)       // 6 cyl bottom right front
    (-0.353553591 -0.353553591 0)       // 7 cyl bottom left front
    (25   5 0)                          // 8 wake top front
    (25 -5 0)                           // 9 wake bottom front

    (-5   5 0.1)                        // 10 top left back
    ( 5   5 0.1)                        // 11 top right back
    ( 5 -5 0.1)                         // 12 bottom right back
    (-5 -5 0.1)                         // 13 bottom left back
    (-0.353553591  0.353553591 0.1)     // 14 cyl top left back
    ( 0.353553591  0.353553591 0.1)     // 15 cyl top right back
    ( 0.353553591 -0.353553591 0.1)     // 16 cyl bottom right back
    (-0.353553591 -0.353553591 0.1)     // 17 cyl bottom left back
    (25   5 0.1)                        // 18 wake top back
    (25 -5 0.1)                         // 19 wake bottom back
);

blocks (
    hex (0 4 5 1 10 14 15 11) (60 60 1) simpleGrading (0.05  1  1) // top
    hex (3 7 4 0 13 17 14 10) (60 60 1) simpleGrading (0.05  1  1) // left
    hex (2 6 7 3 12 16 17 13) (60 60 1) simpleGrading (0.05  1  1) // bottom
    hex (1 5 6 2 11 15 16 12) (60 60 1) simpleGrading (0.05  1  1) // right
    hex (1 2 9 8 11 12 19 18) (60 60 1) simpleGrading (1     1  1) // wake
);

edges (
    arc  4  5 ( 0    0.5 0)      // cyl top front
    arc 14 15 ( 0    0.5 0.1)    // cyl top back
    arc  7  4 (-0.5  0   0)      // cyl left front
    arc 17 14 (-0.5  0   0.1)    // cyl left back
    arc  6  7 ( 0   -0.5 0)      // cyl bottom front
    arc 16 17 ( 0   -0.5 0.1)    // cyl bottom back
    arc  5  6 ( 0.5  0   0)      // cyl right front
    arc 15 16 ( 0.5  0   0.1)    // cyl right back
);

boundary (
    FrontAndBack {
        type empty;
        faces (
            ( 0  1  5  4)  // top front
            (10 14 15 11)  // top back
            ( 3  0  4  7)  // left front
            (13 17 14 10)  // left back
            ( 2  3  7  6)  // bottom front
```

```
            (12 16 17 13)   // bottom back
            ( 1  2  6  5)   // right front
            (11 15 16 12)   // right back
            ( 1  8  9  2)   // wake front
            (11 12 19 18)   // wake back
        );
    }

    TopAndBottom {
        type patch;
        faces (
            ( 0 10 11  1)   // top
            ( 1 11 18  8)   // top wake
            ( 2 12 13  3)   // bottom
            ( 9 19 12  2)   // bottom wake
        );
    }

    Inlet {
        type patch;
        faces (
            (0 3 13 10)
        );
    }

    Outlet {
        type patch;
        faces (
            (8 18 19 9)
        );
    }

    cylinder {
        type patch;
        faces (
            ( 5 15 14  4)   // top
            ( 4 14 17  7)   // left
            ( 7 17 16  6)   // bottom
            ( 6 16 15  5)   // right
        );
    }
);
// ************************************************************************* //
```

Kármán blockMeshDict advanced

In the following we leave out the default comments to save space. They are not required for a successful OpenFOAM operation.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

scale   1;

channelWidth 10;         // width of the channel
wakeLength 20;           // length of wake channel
numberOfMeshPoints 60;   // number of mesh cells per base line
grading 0.05;            // mesh concentration along lase line
w 0.1;                   // width in z direction
```

```
//------------------------------------------------------
r 0.5;
h #eval{ $channelWidth/2.};
l #eval{$h + $wakeLength};
c #eval{$r/sqrt(2)};
n $numberOfMeshPoints;
g $grading;

vertices (
    name v0 (-$h  $h 0)           // 0 top left front
    name v1 ( $h  $h 0)           // 1 top right front
    name v2 ( $h -$h 0)           // 2 bottom right front
    name v3 (-$h -$h 0)           // 3 bottom left front
    name v4 (-$c  $c 0)           // 4 cyl top left front
    name v5 ( $c  $c 0)           // 5 cyl top right front
    name v6 ( $c -$c 0)           // 6 cyl bottom right front
    name v7 (-$c -$c 0)           // 7 cyl bottom left front
    name v8 ($l  $h 0)            // 8 wake top front
    name v9 ($l -$h 0)            // 9 wake bottom front

    name v10 (-$h  $h $w)         // 10 top left back
    name v11 ( $h  $h $w)         // 11 top right back
    name v12 ( $h -$h $w)         // 12 bottom right back
    name v13 (-$h -$h $w)         // 13 bottom left back
    name v14 (-$c  $c $w)         // 14 cyl top left back
    name v15 ( $c  $c $w)         // 15 cyl top right back
    name v16 ( $c -$c $w)         // 16 cyl bottom right back
    name v17 (-$c -$c $w)         // 17 cyl bottom left back
    name v18 ($l  $h $w)          // 18 wake top back
    name v19 ($l -$h $w)          // 19 wake bottom back
);

blocks (
    hex (v0 v4 v5 v1 v10 v14 v15 v11) ($n $n 1) simpleGrading ($g 1 1) // top
    hex (v3 v7 v4 v0 v13 v17 v14 v10) ($n $n 1) simpleGrading ($g 1 1) // left
    hex (v2 v6 v7 v3 v12 v16 v17 v13) ($n $n 1) simpleGrading ($g 1 1) // bottom
    hex (v1 v5 v6 v2 v11 v15 v16 v12) ($n $n 1) simpleGrading ($g 1 1) // right
    hex (v1 v2 v9 v8 v11 v12 v19 v18) ($n $n 1) simpleGrading (1  1 1) // wake
);

edges (
    arc  v4  v5 ( 0   $r 0)      // cyl top front
    arc v14 v15 ( 0   $r $w)     // cyl top back
    arc  v7  v4 (-$r  0  0)      // cyl left front
    arc v17 v14 (-$r  0  $w)     // cyl left back
    arc  v6  v7 ( 0  -$r 0)      // cyl bottom front
    arc v16 v17 ( 0  -$r $w)     // cyl bottom back
    arc  v5  v6 ( $r  0  0)      // cyl right front
    arc v15 v16 ( $r  0  $w)     // cyl right back
);

boundary (
    FrontAndBack {
        type empty;
        faces (
            ( v0  v1  v5  v4)  // top front
            (v10 v14 v15 v11)  // top back
            ( v3  v0  v4  v7)  // left front
            (v13 v17 v14 v10)  // left back
            ( v2  v3  v7  v6)  // bottom front
            (v12 v16 v17 v13)  // bottom back
            ( v1  v2  v6  v5)  // right front
            (v11 v15 v16 v12)  // right back
            ( v1  v8  v9  v2)  // wake front
            (v11 v12 v19 v18)  // wake back
```

```
        );
    }

    TopAndBottom {
        type patch;
        faces (
            ( v0 v10 v11 v1)  // top
            ( v1 v11 v18 v8)  // top wake
            ( v2 v12 v13 v3)  // bottom
            ( v9 v19 v12 v2)  // bottom wake
        );
    }

    Inlet {
        type patch;
        faces (
            (v0 v3 v13 v10)
        );
    }

    Outlet {
        type patch;
        faces (
            (v8 v18 v19 v9)
        );
    }

    cylinder {
        type patch;
        faces (
            ( v5 v15 v14 v4)  // top
            ( v4 v14 v17 v7)  // left
            ( v7 v17 v16 v6)  // bottom
            ( v6 v16 v15 v5)  // right
        );
    }
);
```

Swift-Hohenberg: createFields.H

```
Info<< "Reading field psi\n" << endl;

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

Info<< "Reading field U\n" << endl;

volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
```

```
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading transportProperties:" << endl;
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);
Info<< "Reading eps = ";
dimensionedScalar eps("eps", dimensionSet( 0, 0, 0, 0, 0, 0, 0),
                                                  transportProperties);
Info<< eps << endl << "Reading A = ";
dimensionedScalar A("A", dimensionSet( 0, 0, 0, 0, 0, 0, 0),
                                                  transportProperties);
Info<< A << endl;
#include "createPhi.H"
#include "createFvOptions.H"
```

Swift-Hohenberg: mySolver.C

```
#include "fvCFD.H"
#include "fvOptions.H"
#include "simpleControl.H"
int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Solve Swift-Hohenberg Equation."
    );
    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H"
    simpleControl simple(mesh);
    #include "createFields.H"
    Info<< "\nSolving Swift-Hohenberg Equation\n" << endl;
    #include "CourantNo.H"
    dimensionedScalar ons(dimensionSet( 0, 0, -1, 0, 0, 0, 0), 1);
    dimensionedScalar tns(dimensionSet( 0, 4, -1, 0, 0, 0, 0), 1);
    dimensionedScalar two(dimensionSet( 0, 2, -1, 0, 0, 0, 0), 2);
    while (simple.loop())
    {
```

```
        Info<< "Time = " << runTime.timeName() << nl << endl;
        while (simple.correctNonOrthogonal())
        {
            fvScalarMatrix psiEqn
            (
              fvm::ddt(psi)
              - (eps - 1)*psi*ons
              + fvm::laplacian(two, psi)
              + fvc::laplacian(tns*fvc::laplacian(psi))
              - psi*psi*(A-psi)*ons
             ==
                fvOptions(psi)
            );
            psiEqn.relax();
            fvOptions.constrain(psiEqn);
            psiEqn.solve();
            fvOptions.correct(psi);
        }
        runTime.write();
    }
    Info<< "End\n" << endl;
    return 0;
}
```

Swift-Hohenberg: system/blockMeshDict

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

scale   1;
wi  128;              // length and width of the domain
wir 256;              // number of cells along each x and y

vertices
(
    name v0 (0   0   0)
    name v1 ($wi 0   0)
    name v2 ($wi $wi 0)
    name v3 (0   $wi 0)
    name v4 (0   0   0.1)
    name v5 ($wi 0   0.1)
    name v6 ($wi $wi 0.1)
    name v7 (0   $wi 0.1)
);

blocks
(
    hex (v0 v1 v2 v3 v4 v5 v6 v7) ($wir $wir 1) simpleGrading (1 1 1)
);

edges ( );

boundary
(
    top
```

```
    {
        type cyclic;
        neighbourPatch bottom;
        faces ((v3 v7 v6 v2));
    }

    bottom
    {
        type cyclic;
        neighbourPatch top;
        faces ((v0 v1 v5 v4));
    }

    left
    {
        type cyclic;
        neighbourPatch right;
        faces ((v0 v4 v7 v3));
    }

    right
    {
        type cyclic;
        neighbourPatch left;
        faces ((v2 v6 v5 v1));
    }

    frontAndBack
    {
        type empty;
        faces
        (
            (v0 v3 v2 v1)
            (v4 v5 v6 v7)
        );
    }
);
```

Swift-Hohenberg: constant/transportProperties

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      transportProperties;
}

eps         0.1;
A           0.0;    // for hexagons A = 0.5; for roles A = 0.0
```

Swift-Hohenberg: 0/U

```
FoamFile {
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}

dimensions      [0 1 -1 0 0 0 0];
internalField   uniform  (0 0 0);
boundaryField {
```

```
    top           { type  cyclic; }
    bottom        { type  cyclic; }
    left          { type  cyclic; }
    right         { type  cyclic; }
    frontAndBack  { type  empty; }
}
```

Swift-Hohenberg: 0/psi

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

dimensions      [0 0 0 0 0 0 0];
// internalField   uniform 0;

internalField #codeStream
{
    codeInclude
    #{
       #include "fvCFD.H"
    #};
    codeOptions
    #{
       -I$(LIB_SRC)/finiteVolume/lnInclude \
       -I$(LIB_SRC)/meshTools/lnInclude
    #};
    codeLibs
    #{
       -lmeshTools \
       -lfiniteVolume
    #};
    code
    #{
       const IOdictionary& d = static_cast<const Iodictionary&>(dict);
       const fvMesh& mesh = refCast<const fvMesh>(d.db());
       scalarField psi(mesh.nCells(), 0.);
       std::mt19937 gen(12345);                          // C++ STL
       const double ampl = 0.001;
       std::uniform_real_distribution<> dis(-ampl, ampl); // C++ STL

       forAll(psi, I)
       {
// We left in the commented out code for illustration what's possible
//           const scalar x = mesh.C()[i][0];
//           const scalar y = mesh.C()[i][1];
//           const scalar z = mesh.C()[i][2];
//           psi[i] = sin(x)*sin(y);
           psi[i] = dis(gen);
       }
       psi.writeEntry("", os);
    #};
};

boundaryField {
    top          { type cyclic; }
```

```
    bottom       { type cyclic; }
    left         { type cyclic; }
    right        { type cyclic; }
    frontAndBack { type empty;  }
}
```

Swift-Hohenberg: system/controlDict

```
FoamFile {
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}

application       mySolver;
startFrom         startTime;
startTime         0;
stopAt            endTime;
endTime           300.;
deltaT            0.001;
writeControl      timeStep;
writeInterval     10000;
purgeWrite        0;
writeFormat       ascii;
writePrecision    6;
writeCompression  off;
timeFormat        general;
timePrecision     6;
runTimeModifiable true;
```

Swift-Hohenberg: system/fvSchemes and system/fvSolution

```
FoamFile {
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSchemes;
}

ddtSchemes {
    default          Euler;
}

gradSchemes {
    default          Gauss linear;
}

divSchemes {
    default          none;
//    div(phi,T)       Gauss linearUpwind grad(T);
}

laplacianSchemes {
    default          none;
    laplacian(psi)   Gauss linear corrected;
    laplacian((1*laplacian(psi)))  Gauss linear corrected;
    laplacian(2,psi)   Gauss linear corrected;
}
```

```
interpolationSchemes {
    default          linear;
}

snGradSchemes {
    default          corrected;
}
```

```
FoamFile {
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}

solvers {
    psi {
        solver          PbiCGStab;
        preconditioner  FDIC; // GAMG;
        tolerance       1e-06;
        relTol          0;
    }
}

SIMPLE {
    nNonOrthogonalCorrectors 0;
}
```