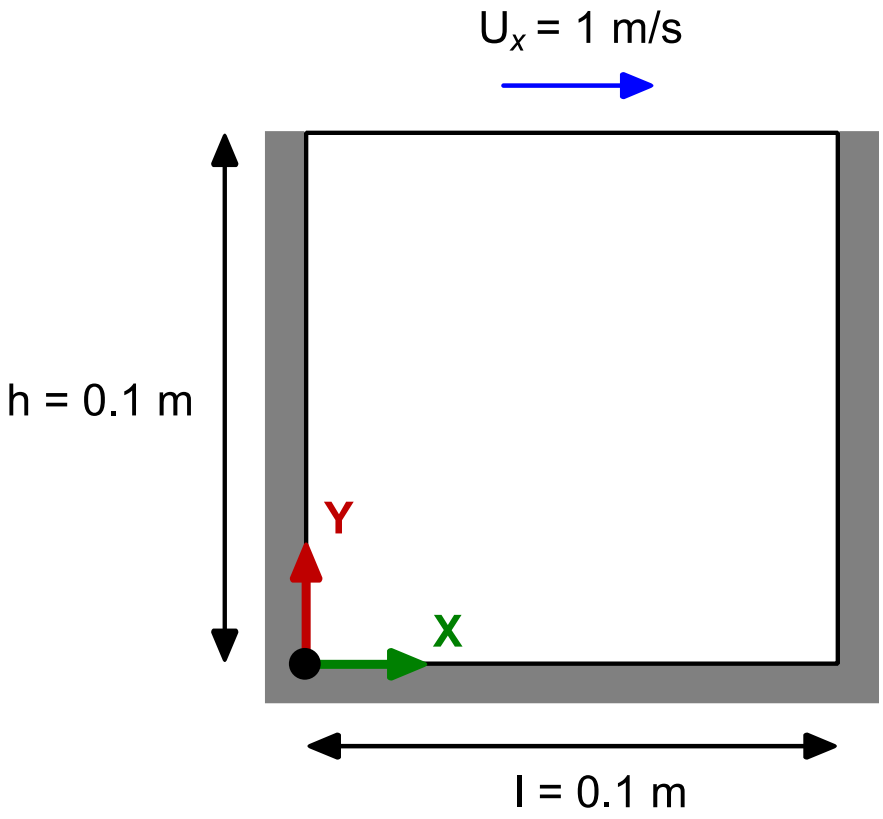
Running my first OpenFOAM® case

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

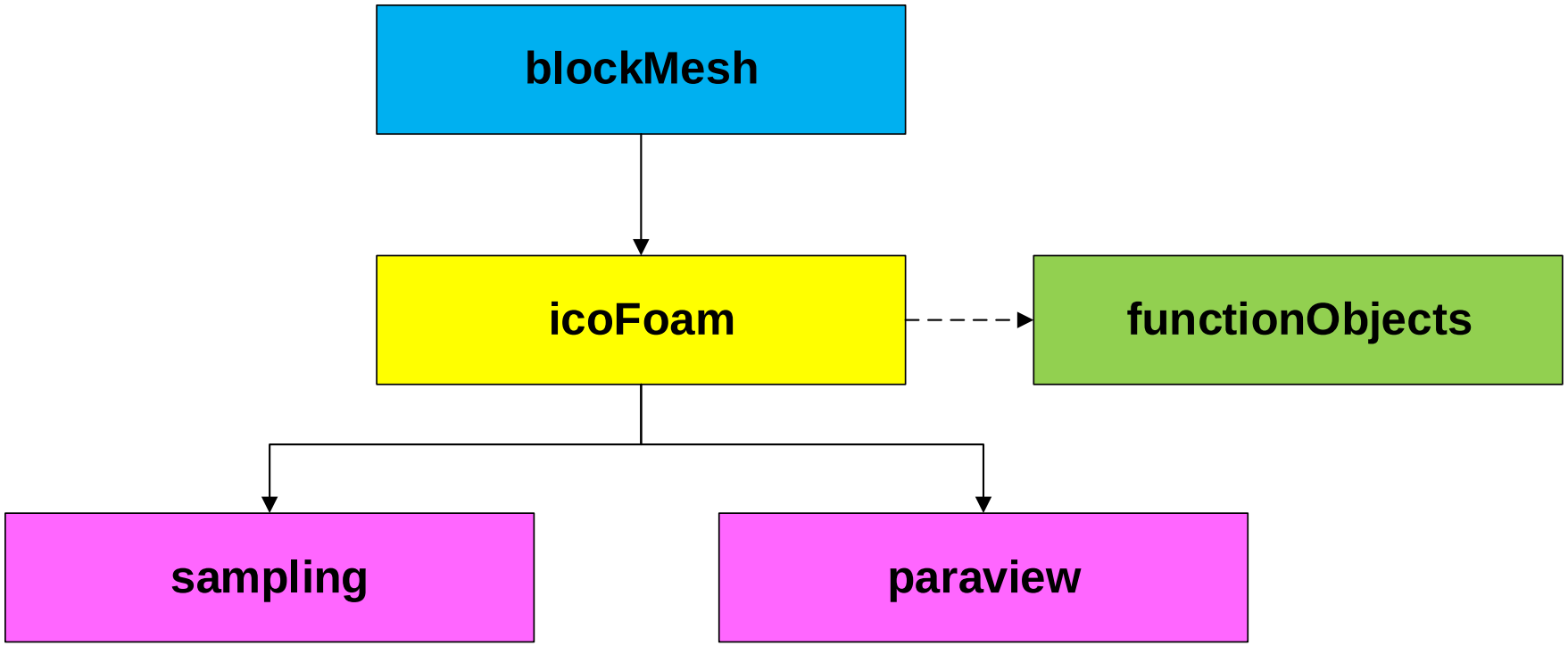


Physical and numerical side of the problem:

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

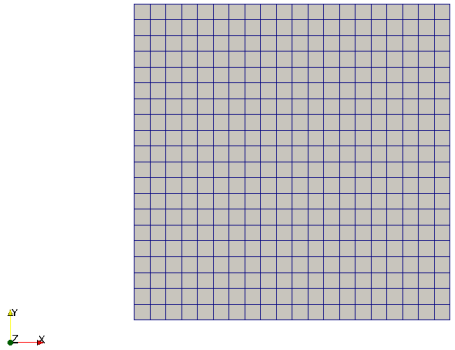
Running my first OpenFOAM® case

Workflow of the case



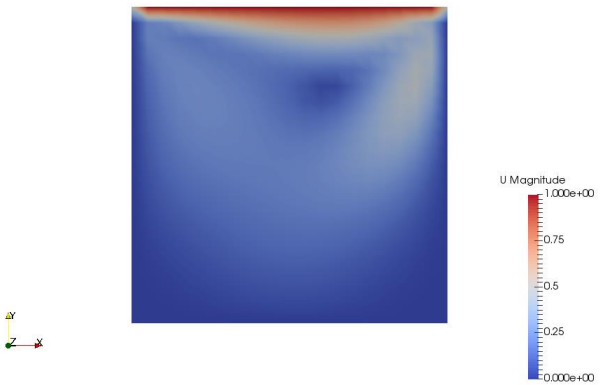
Running my first OpenFOAM® case

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



Mesh (very coarse and 2D)

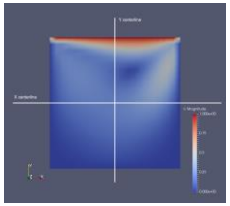
Pressure field (relative pressure)



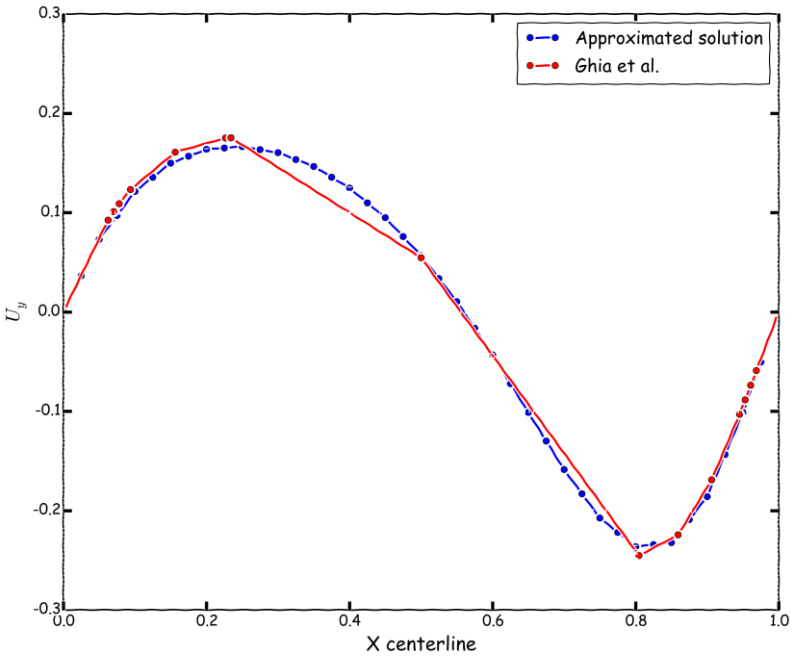
Velocity magnitude field

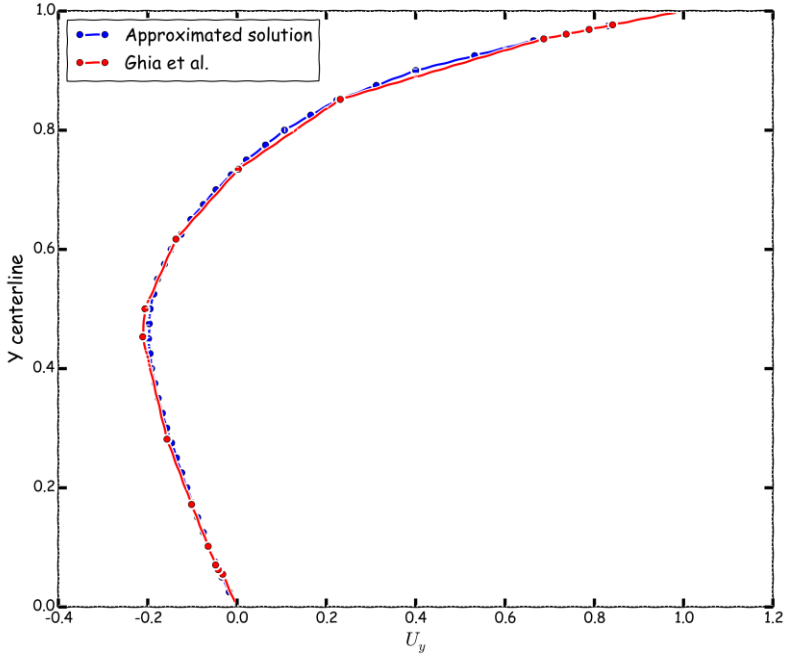
Running my first OpenFOAM® case

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



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





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

Running my first OpenFOAM® case

Loading OpenFOAM® environment

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

- To source OpenFOAM®, type in the terminal:

    - `$> of4x`

- To use PyFoam you will need to source it. Type in the terminal:

    - `$> anaconda2` or `$> anaconda3`

- Remember, every time you open a new terminal window you need to source OpenFOAM® and PyFoam.

- By default, when installing OpenFOAM® and PyFoam you do not need to do this. This is our choice as we have many things installed and we want to avoid conflicts between applications.

Running my first OpenFOAM® case

What are we going to do?

- We will use the lid-driven square cavity tutorial as a general example to show you how to set up and run solvers and utilities in OpenFOAM®.
- In this tutorial we are going to generate the mesh using `blockMesh`.
- After generating the mesh, we will look for topological errors and assess the mesh quality. For this we use the utility `checkMesh`. Later on, we are going to talk about what is a good mesh.
- Then, we will find the numerical solution using `icoFoam`, which is a transient solver for incompressible, laminar flow of Newtonian fluids. By the way, we hope you did not forget where to look for this information.
- And we will finish with some cool visualization and post-processing using `paraFoam`.
- While we run this case, we are going to see a lot of information on the screen (standard output stream or stdout), but it will not be saved. This information is mainly related to convergence of the simulation, we will talk about this later on.

Running my first OpenFOAM® case

Running the case blindfold

- Let us run this case blindfold. Later we will study in details each file and directory.
- In the terminal window type:

1. `$> cd $PTOFC/101OF/cavity`
   Remember, `$PTOFC` is pointing to the path where you unpacked the tutorials.
2. `$> ls -l`
3. `$> blockMesh`
4. `$> checkMesh`
5. `$> icoFoam`
6. `$> postProcess -func sampleDict -latestTime`
7. `$> gnuplot gnuplot/gnuplot_script`
8. `$> paraFoam`

Running my first OpenFOAM® case

Running the case blindfold

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

Running my first OpenFOAM® case

Crash introduction to paraFoam

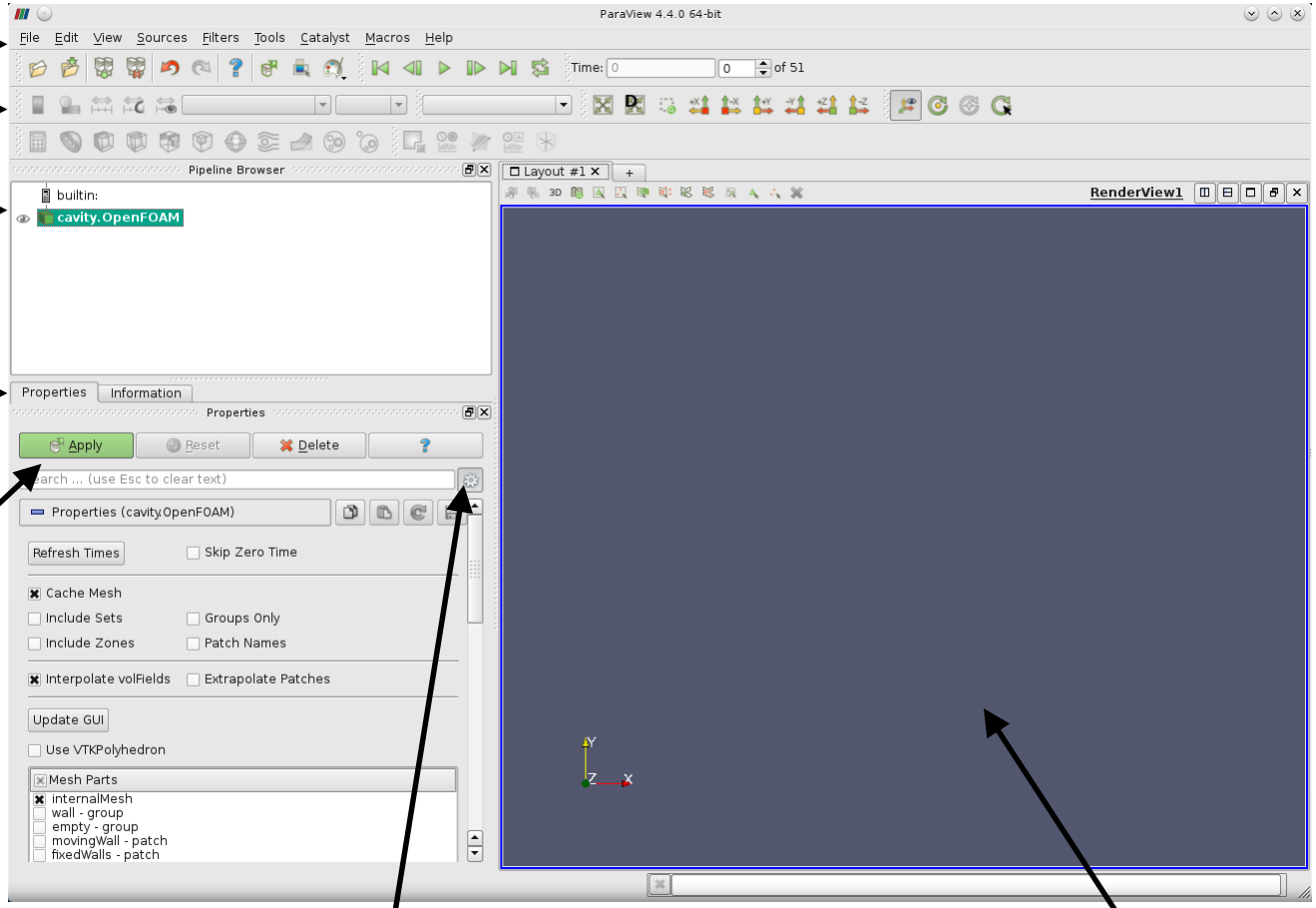
Menu Bar

Toolbars

Pipeline Browser

Properties panel

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



Advanced Toggle

3D View/Canvas

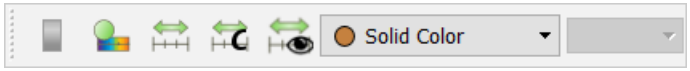
Running my first OpenFOAM® case

Crash introduction to paraFoam – Toolbars

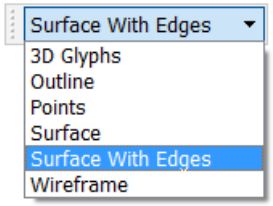
- Main Controls

- VCR Controls (animation controls)

- Current Time Controls



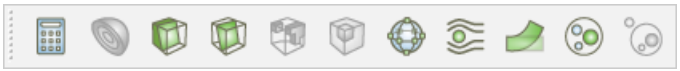
- Active Variable Controls



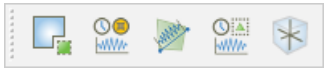
- Representation Toolbar

- Camera Controls (view orientation)

- Center Axes Controls



- Common Filters



- Data Analysis Toolbar

Running my first OpenFOAM® case

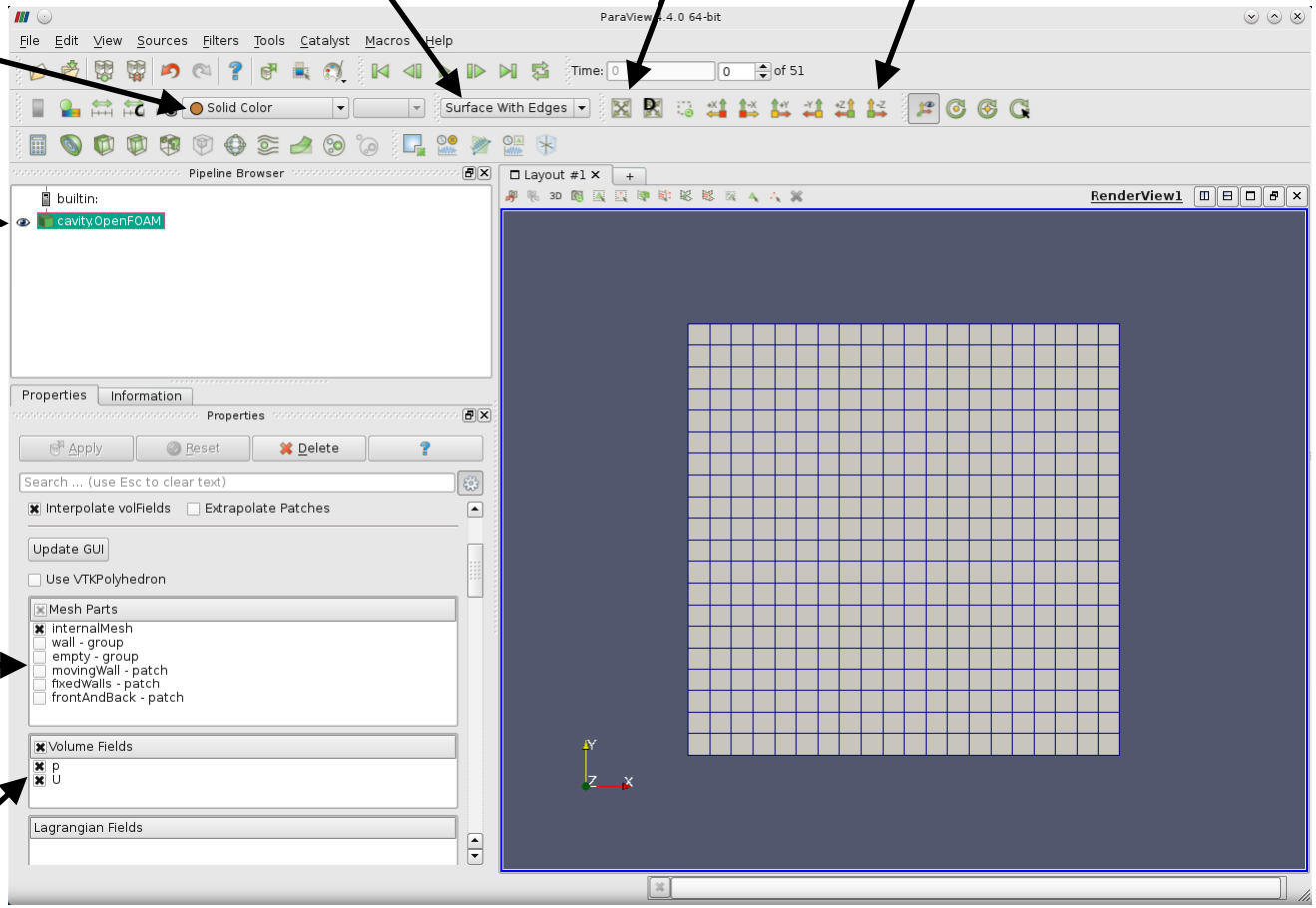
Crash introduction to paraFoam – Mesh visualization

Select `Surface With Edges` in the Representation Toolbar

Fit to screen

Select the -Z view

Select `Solid Color` in the Active Variable Controls



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

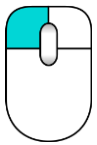
Select mesh parts to visualize. By default it will automatically select `internalMesh`

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

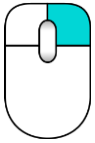
Running my first OpenFOAM® case

Crash introduction to paraFoam – 3D View and mouse interaction

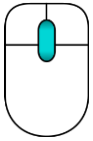
Mouse interaction in the 3D view



Rotate



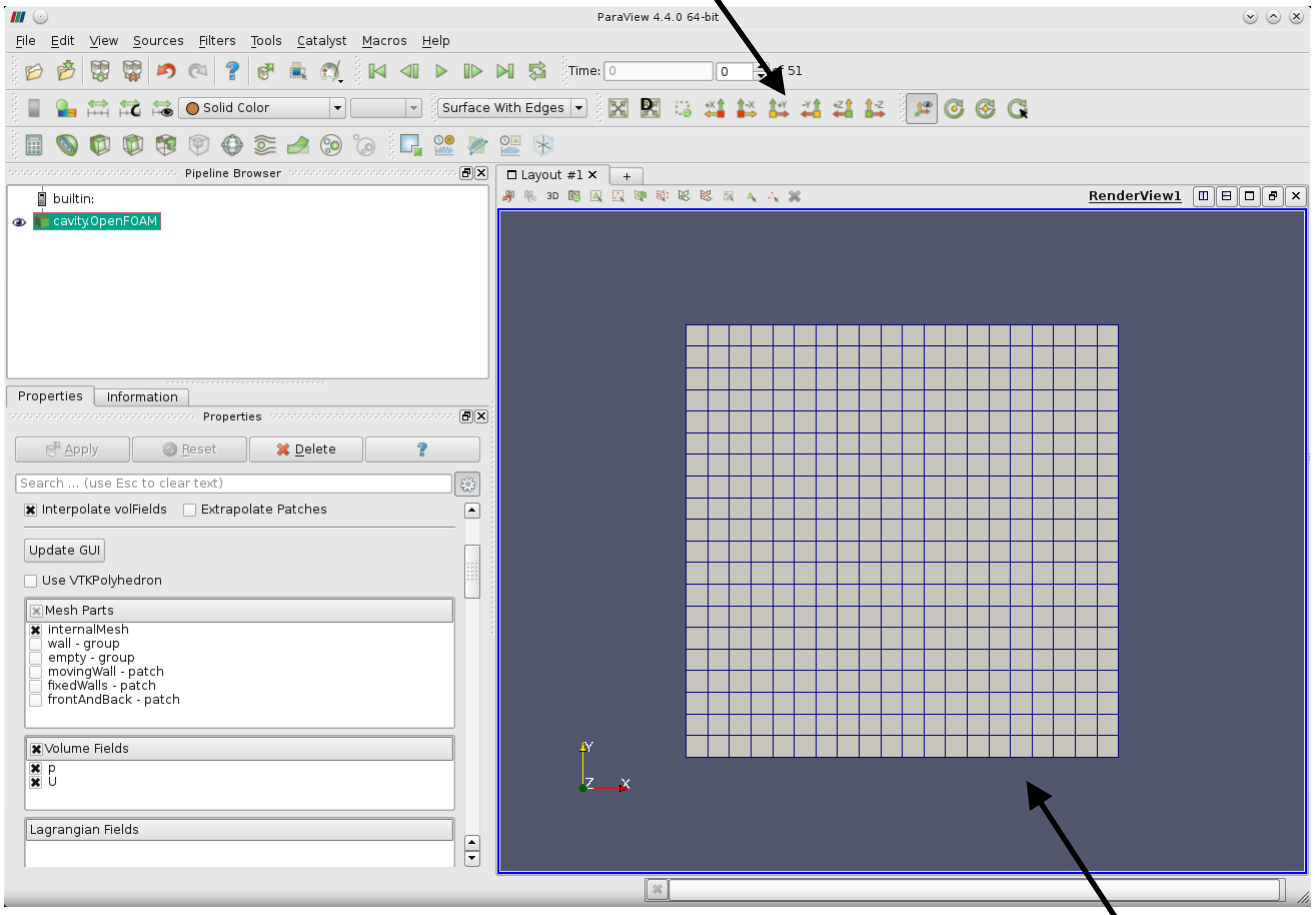
Zoom



Pan

Zoom

Select view orientation in the Camera Controls



3D View/Canvas

Running my first OpenFOAM® case

Crash introduction to paraFoam – Fields visualization

Select `Last Frame` in the VCR Controls

Current Time Controls

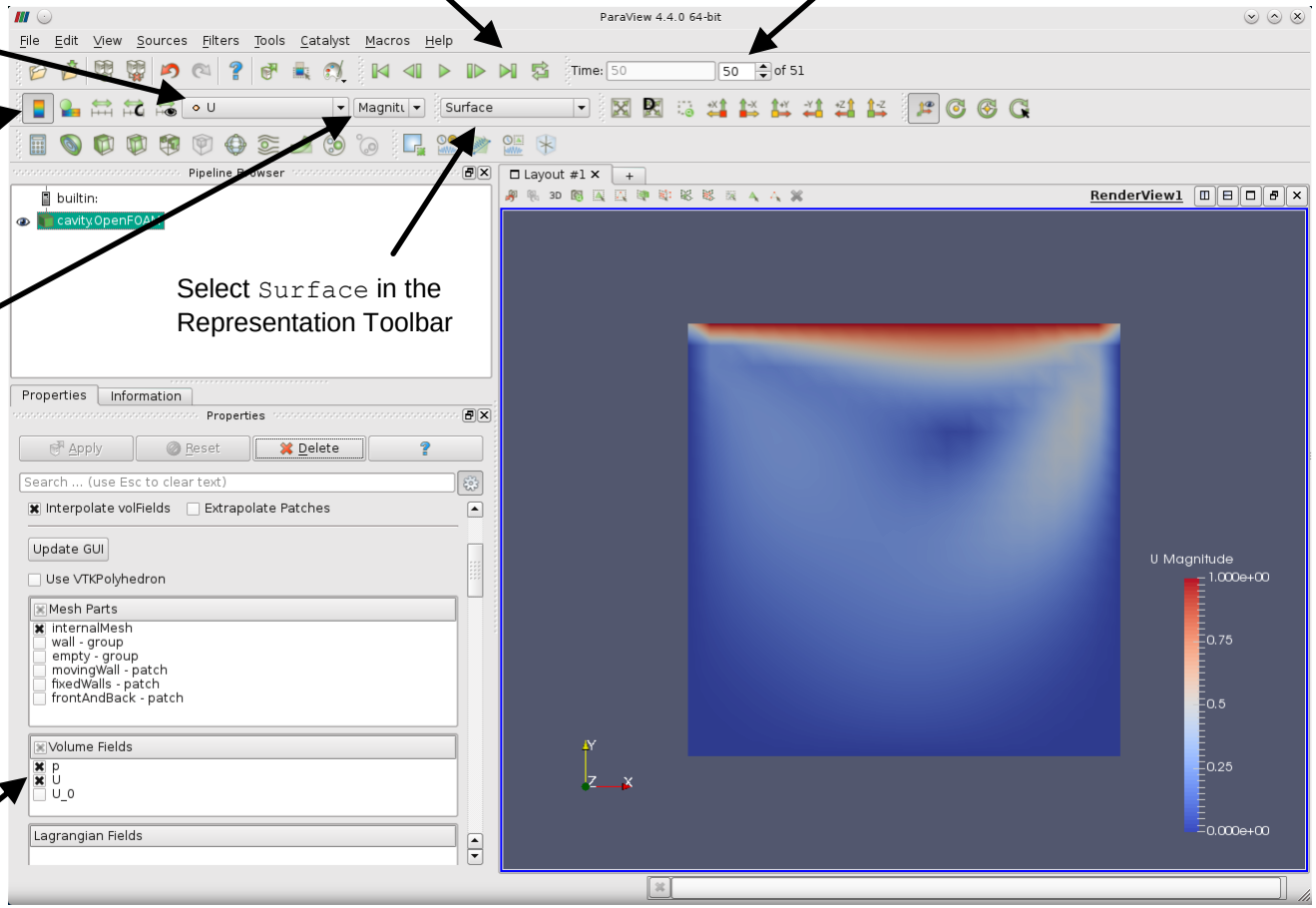
Select **U** in Active Variable Controls

Turn on/off color bar

Select `Magnitude` in the drop down menu

Select `Surface` in the Representation Toolbar

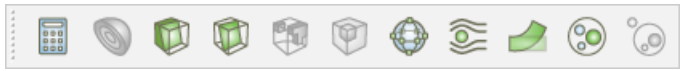
Select volume fields to visualize.
By default it will select **U** and **p.**



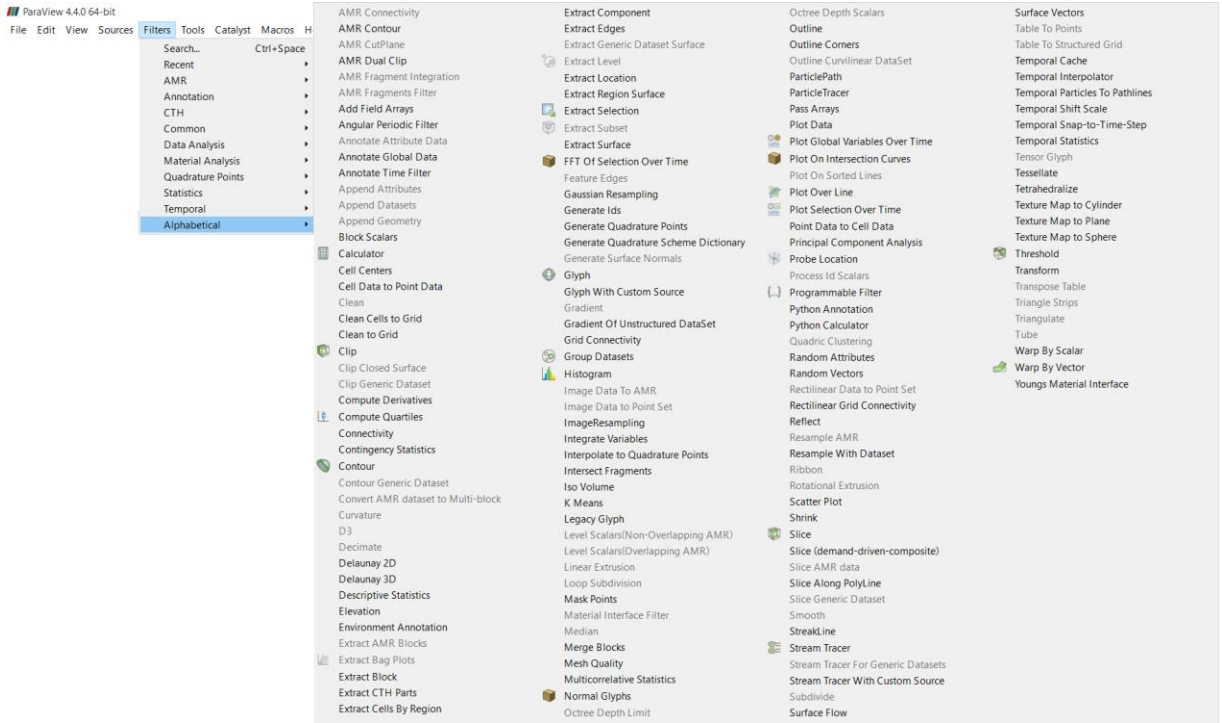
Running my first OpenFOAM® case

Crash introduction to paraFoam – Filters

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



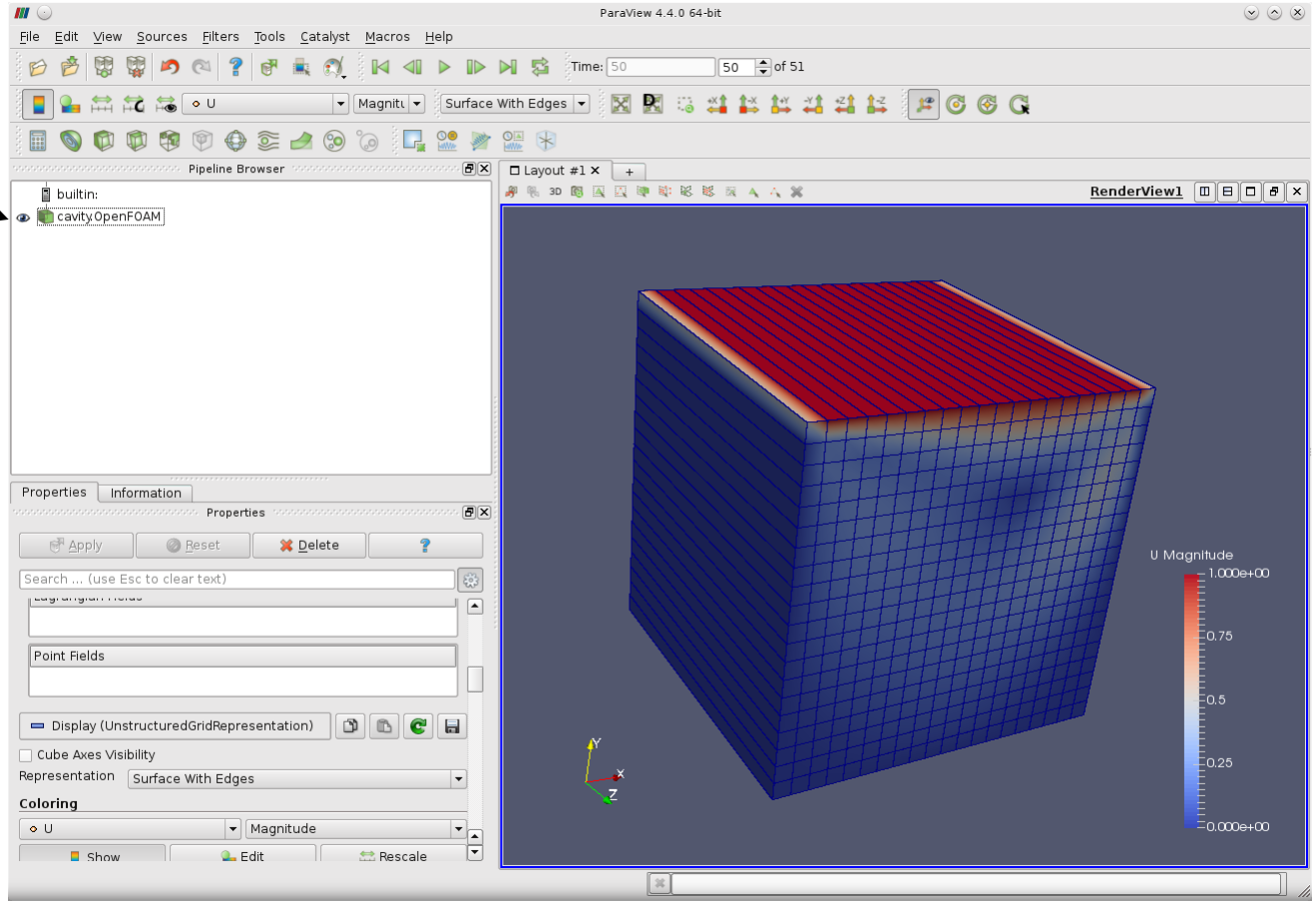
- You can access all the filters from the menu `Filter`.



Running my first OpenFOAM® case

Crash introduction to paraFoam – Filters

Filters are attached to the input data



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

Running my first OpenFOAM® case

Crash introduction to paraFoam – Slice filter

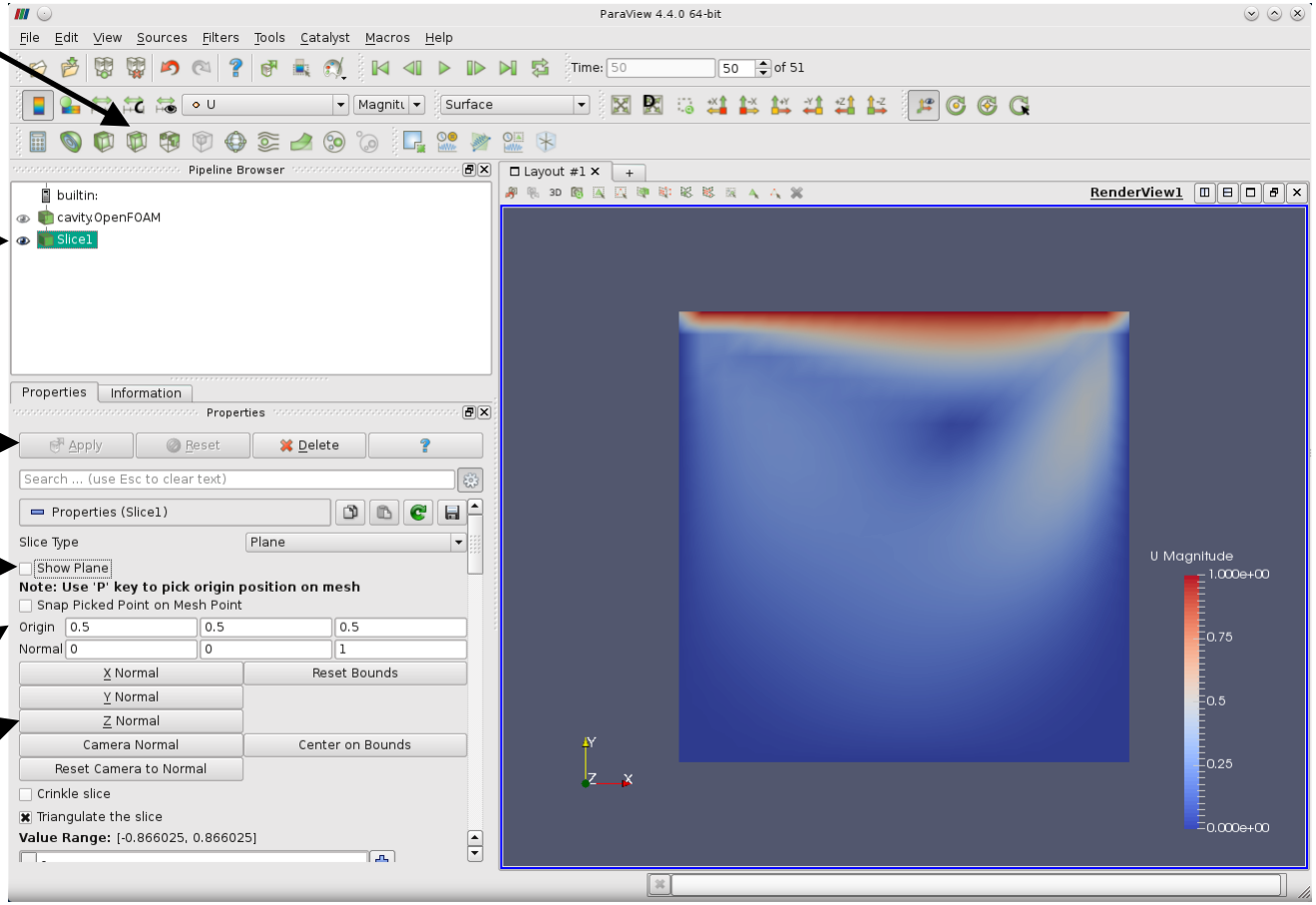
1. Select the `Slice` filter

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

4. Press `Apply`

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

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



Running my first OpenFOAM® case

Crash introduction to paraFoam – Glyph filter

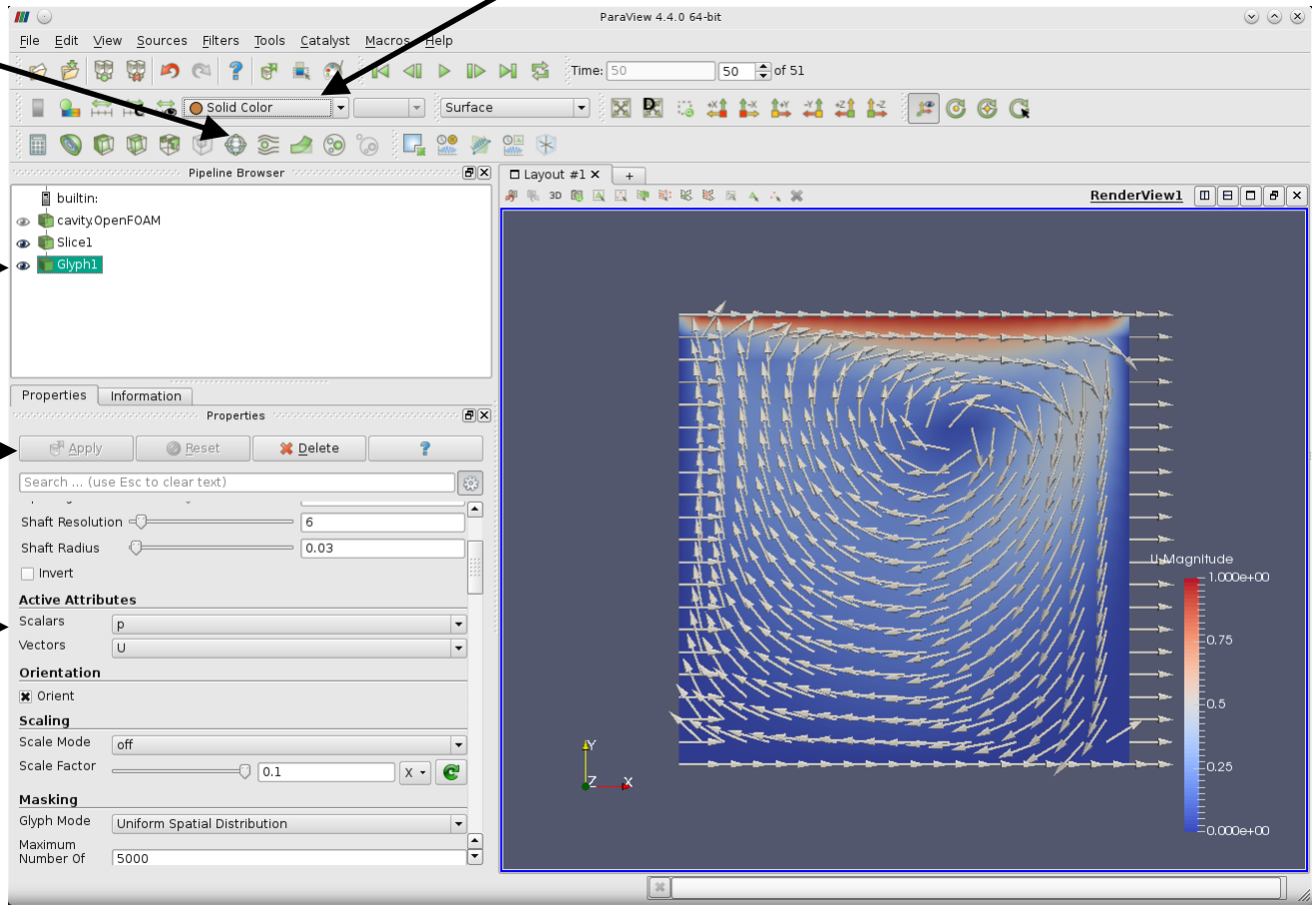
1. Select the `Glyph` filter. This filter will be applied on the `Slice1` filter

4. Color the colors using Solid Color

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

3. Press `Apply`

2. Filter options



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

Running my first OpenFOAM® case

Crash introduction to paraFoam – Plot Over Line filter

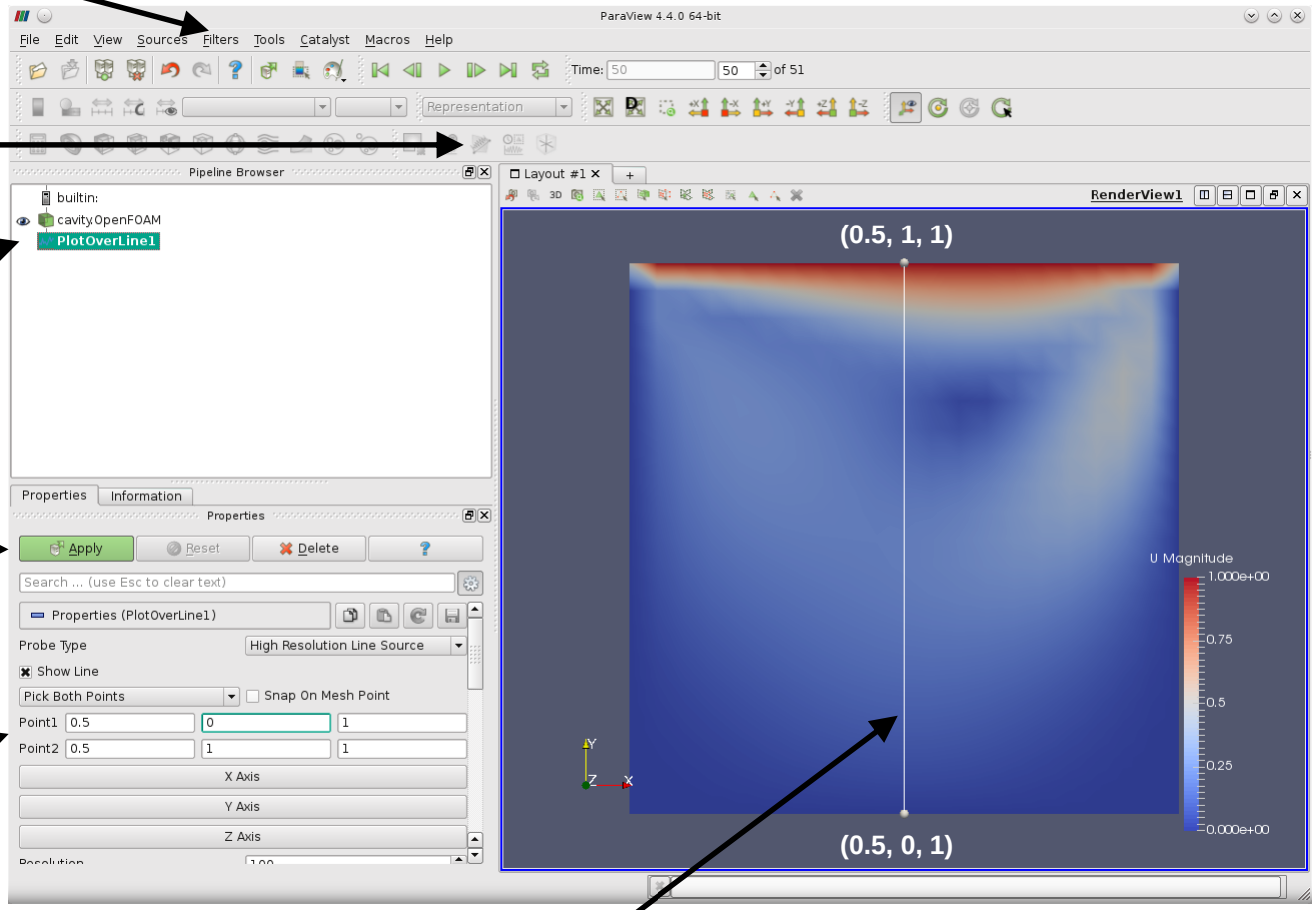
1.a. Select the `Plot Over Line` filter.

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

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

3. Press `Apply`

2. Enter the coordinates of the line



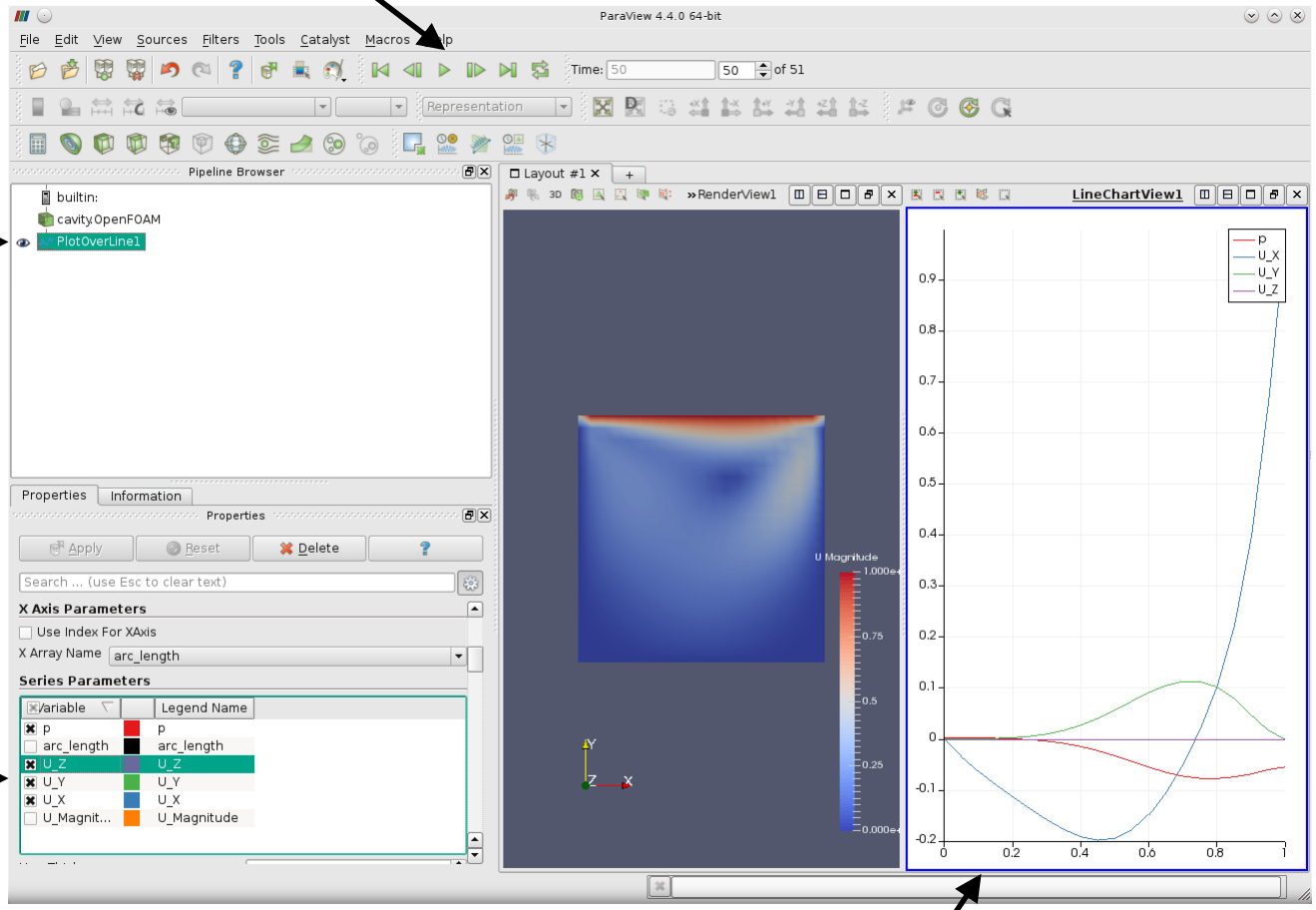
Running my first OpenFOAM® case

Crash introduction to paraFoam – Filters

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

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

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



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

Running my first OpenFOAM® case

Running the case blindfold with log files

- In the previous case, we ran the simulation but we did not save the standard output stream (stdout) in a `log` file. We saw the information on-the-fly.
- Our advice is to always save the standard output stream (stdout) in a `log` file.
- It is of interest to always save the `log` as if something goes wrong and you would like to do troubleshooting, you will need this information.
- Also, if you are interested in plotting the residuals you will need the `log` file.
- By the way, if at any point you ask us what went wrong with your simulation, we will ask you for this file. We might also ask for the standard error stream (stderr).

Running my first OpenFOAM® case

Running the case blindfold with log files

- To save a *log* file of the simulation, we proceed as follows:

```
1. $> foamCleanTutorials
2. $> foamCleanPolyMesh
3. $> blockMesh
4. $> checkMesh
5. $> icoFoam > log.icoFoam
6. $> gedit log.icoFoam &
7. $> foamLog log.icoFoam
8. $> gnuplot
```

These steps are optional

Running my first OpenFOAM® case

Running the case blindfold with log files

- In steps 1 and 2 we erase the mesh and all the folders, except for **`0`**, **`constant`** and **`system`**. These scripts come with your OpenFOAM® installation.
- In step 3, we generate the mesh using the meshing tool `blockMesh`.
- In step 4 we check the mesh quality.
- In step 5 we run the simulation. Hereafter, we redirect the standard output to an ascii file with the name *`log.icoFoam`* (it can be any name). However, you will not see the information on the fly. If you do not add the `> log.icoFoam` modifier you will see your standard output on the fly but it will not be saved.
- In step 6, we use gedit to open the file *`log.icoFoam`* (we run gedit in background). Remember, you can use any editor. Also, depending on the size of the log file, opening the file can be very time consuming.
- In step 7, we use the script `foamLog` (distributed with your OpenFOAM® installation), to extract the information inside the file *`log.icoFoam`*. This information is saved in an editable/plottable format in the directory **`logs`**.
- Finally, in step 8 we use gnuplot to plot the information extracted from the *`log.icoFoam`* file.

Running my first OpenFOAM® case

Running the case blindfold with log files

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

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

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

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

4. `gnuplot> reset`
   To reset the scales

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

6. `gnuplot> set logscale y`

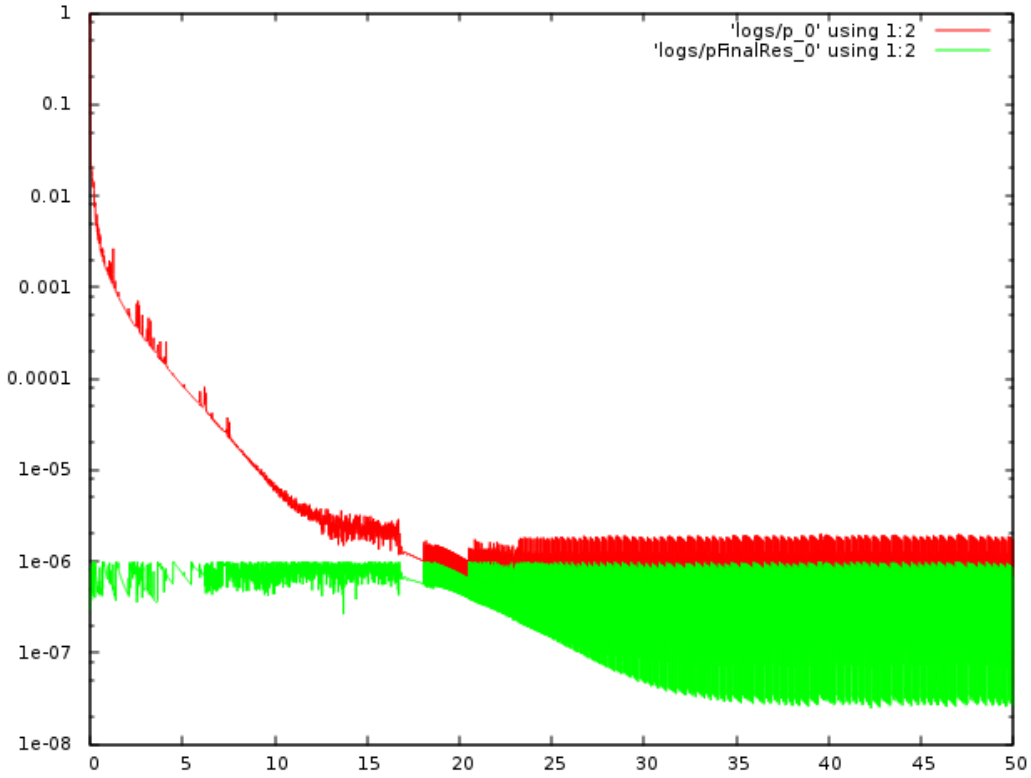
7. `gnuplot> plot [30:50][] 'logs/Ux_0' u 1:2 w l title 'Ux','logs/Uy_0' u 1:2 w l title 'Uy'`
   Set the x range from 30 to 50 and plot tow files and set legend titles

8. `gnuplot> exit`
   To exit gnuplot

Running my first OpenFOAM® case

Running the case blindfold with log files

- The output of step 3 is the following:



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

Running my first OpenFOAM® case

Running the case blindfold with log files

- But what if we want to save the standard output stream (stdout), and monitor the information on the fly?
- To do this and if you are using BASH shell, you can proceed as follows:
  - `$> icoFoam > log.icoFoam  | tail -f log.icoFoam`
- This will redirect your standard output to an ascii file with the name *`log.icoFoam`*. Then using the pipeline operator (|) it will use the `tail` command to show you the information on the fly.
- When the simulation is over, you will notice that the terminal window is blocked. To unblock the terminal window press `ctrl-c`.
- You can also save the standard output in the file *`log.icoFoam`*, open a new terminal window and then use `tail` to output the information on the fly. To do this you can proceed as follow:
  - `$> icoFoam > log.icoFoam`
- Now, in a new terminal window (or tab) and in the same directory where you are running the application, type in the terminal,
  - `$> tail -f log.icoFoam`
- This will use `tail` to show you the information on the fly. Have in mind that you need to be in the case directory.

Running my first OpenFOAM® case

Running the case blindfold with log files

- You can also save the standard output in a *log.icoFoam*, send the job to background and then use `tail` to output the information on the fly. To do this you can proceed as follows,
  - `$> icoFoam > log.icoFoam &`
- Now you can type in the terminal window,
  - `$> tail -f log.icoFoam`

  This will use `tail` to show you the information on the fly. Notice we are still working in the same terminal window or tab.
- If you want to stop the command `tail`, press `ctrl-c`.
- You can also use the Linux command `tee`,
  - `$> icoFoam | tee log.icoFoam`

  This will redirect your standard output to an ascii file with the name *log.icoFoam*, and it will show at the same time the information that is saved in the file.
- If for any reason you do not want to see the standard output stream and you are not interested in saving the *log* file, you can proceed as follows,
  - `$> icoFoam > /dev/null`

Running my first OpenFOAM® case

Running the case blindfold with log files

- You can also save the standard output stream (stdout) and the standard error stream (stderr), as follows
  - `$> icoFoam > log.icoFoam 2>&1 | tail -f log.icoFoam`
- This will redirect the standard output and standard error streams to an ascii file with the name *`log.icoFoam`*. Then using the pipeline operator (|) it will use `tail` to show you the information on the fly.
- Finally, when you are running in a cluster using a job scheduler, you are always interested in saving the *`log`* files in order to monitor the solution. Remember to always redirect the solver standard output and error streams to a *`log`* file.
- To monitor your solution, just login to the cluster, go to the working directory (the directory where you launched the solver) and type
  - `$> tail -f name_of_the_log_file`
- You can login and logout with no problem, everything is being managed by the job scheduler.
- Besides the *`log`* file you are saving, the job scheduler will save all the standard output stream (stdout) and standard error stream (stderr) in a default file. You can also access these files to monitor the solution.

Running my first OpenFOAM® case

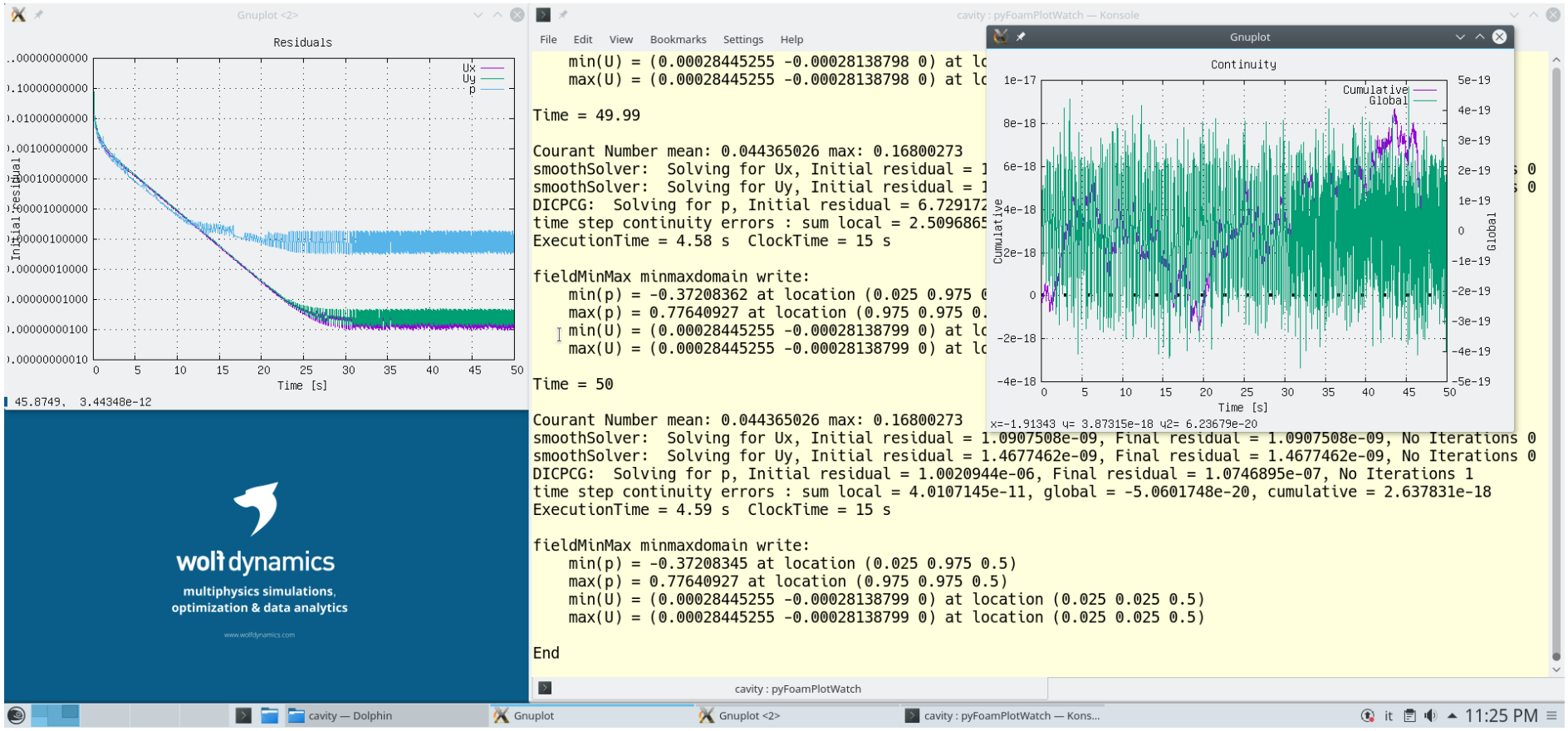
Running the case blindfold with log files and plotting the residuals

- It is also possible to plot the *log* information on the fly.
- The easiest way to do this is by using PyFoam (you will need to install it):
  - `$> pyFoamPlotRunner.py [options] <foamApplication>`
- If you are using our virtual machine or using the lab workstations, you will need to source PyFoam.
- To source PyFoam, type in the terminal:
  - `$> anaconda2` or `$> anaconda3`
- To run this case with `pyFoamPlotRunner.py`, in the terminal type:
  - `$> pyFoamPlotRunner.py icoFoam`
- If you need help or want to know all the options available,
  - `$> pyFoamPlotRunner.py --help`
- If you do not feel comfortable using `pyFoamPlotRunner.py` to run the solver, it is also possible to plot the information saved in the *log* file using PyFoam. To do so you will need to use the utility `pyFoamPlotWatcher.py`. For example, in the terminal type:
  - `$> icoFoam > log.icoFoam &`
  - `$> pyFoamPlotWatcher.py log.icoFoam`
- This will plot the information saved in *log.icoFoam*.
- You can also use `pyFoamPlotWatcher.py` to plot the information saved in an old *log* file.

Running my first OpenFOAM® case

Running the case blindfold with log files and plotting the residuals

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



Running my first OpenFOAM® case

Stopping the simulation

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

endTime 50;

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

```
1   /*--------------------------------*- C++ -*----------------------------------*\
2   | =========                 |                                                 |
3   | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4   |  \\    /   O peration     | Version:  4.x                                   |
5   |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6   |    \\/     M anipulation  |                                                 |
7   \*---------------------------------------------------------------------------*/
8   FoamFile
9   {
10      version     2.0;
11      format      ascii;
12      class       dictionary;
13      object      controlDict;
14  }
15  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17  application     icoFoam;
18
19  startFrom       startTime;
20
21  startTime       0;
22
23  stopAt          endTime;
24
25  endTime         50;
```

Running my first OpenFOAM® case

Stopping the simulation

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

stopAt writeNow;

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

```
1   /*--------------------------------*- C++ -*----------------------------------*\
2   | =========                 |                                                 |
3   | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4   |  \\    /   O peration     | Version:  4.x                                   |
5   |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
6   |    \\/     M anipulation  |                                                 |
7   \*---------------------------------------------------------------------------*/
8   FoamFile
9   {
10      version     2.0;
11      format      ascii;
12      class       dictionary;
13      object      controlDict;
14  }
15  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17  application     icoFoam;
18
19  startFrom       startTime;
20
21  startTime       0;
22
23  stopAt          writeNow;
24
25  endTime         50;
```

Running my first OpenFOAM® case

Stopping the simulation

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

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

44
45  runTimeModifiable true;
46
```

Running my first OpenFOAM® case

Stopping the simulation

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

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

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

Process `id` number

Running my first OpenFOAM® case

Stopping the simulation

- When working locally, we usually proceed in this way:
  - `$> icoFoam > log.icofoam | tail -f log.icofoam`

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

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

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

  stopAt writeNow;

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

Running my first OpenFOAM® case

Cleaning the case folder

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

```
$> foamCleanTutorials
```

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

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

```
$> foamCleanPolyMesh
```

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

```
$> foamListTimes -rm
```

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

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

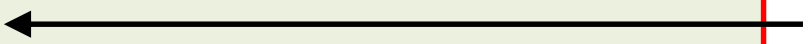
A deeper view to my first OpenFOAM® case setup

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

A deeper view to my first OpenFOAM® case setup

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

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



A deeper view to my first OpenFOAM® case setup

Let us explore the case directory

A deeper view to my first OpenFOAM® case setup

The **`constant`** directory

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

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

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

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

- Your dimensions shall be consistent.

A deeper view to my first OpenFOAM® case setup

Dimensions in OpenFOAM® (metric system)

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

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

A deeper view to my first OpenFOAM® case setup

The **constant** directory

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

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

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

has the following units

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

Which is equivalent to

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

A deeper view to my first OpenFOAM® case setup

The `constant` directory

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

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

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

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

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

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

A deeper view to my first OpenFOAM® case setup

The **`constant`** directory

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

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

A deeper view to my first OpenFOAM® case setup

The **`constant/polyMesh`** directory

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

- In this case, the **`polyMesh`** directory is initially empty. After generating the mesh, it will contain the mesh in OpenFOAM® format.
- To generate the mesh in this case, we use the utility `blockMesh`. This utility reads the dictionary *`blockMeshDict`* located in the **`system`** folder.
- We will now take a quick look at the *`blockMeshDict`* dictionary in order to understand what we have done. Do not worry, we are going to revisit this dictionary during the meshing session.
- Go to the directory **`system`** and open *`blockMeshDict`* dictionary with your favorite text editor, we will use gedit.

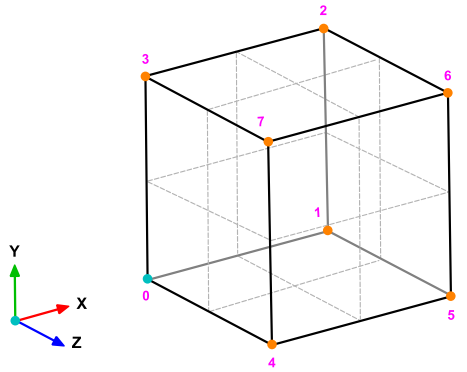
A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary first at all contains a list with a number of vertices:

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

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



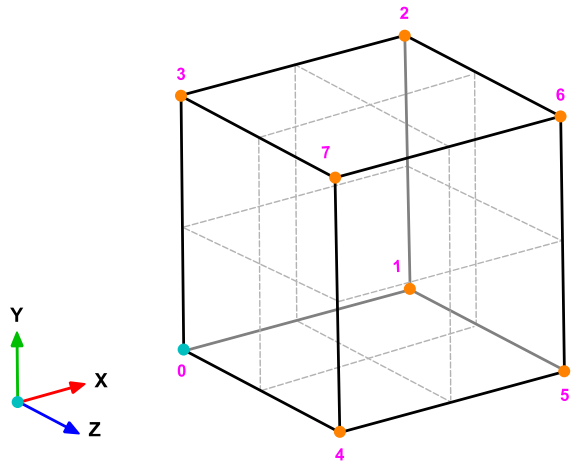
A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary then defines a block from the vertices:

```
60      blocks
61      (
62          hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
63      );
```

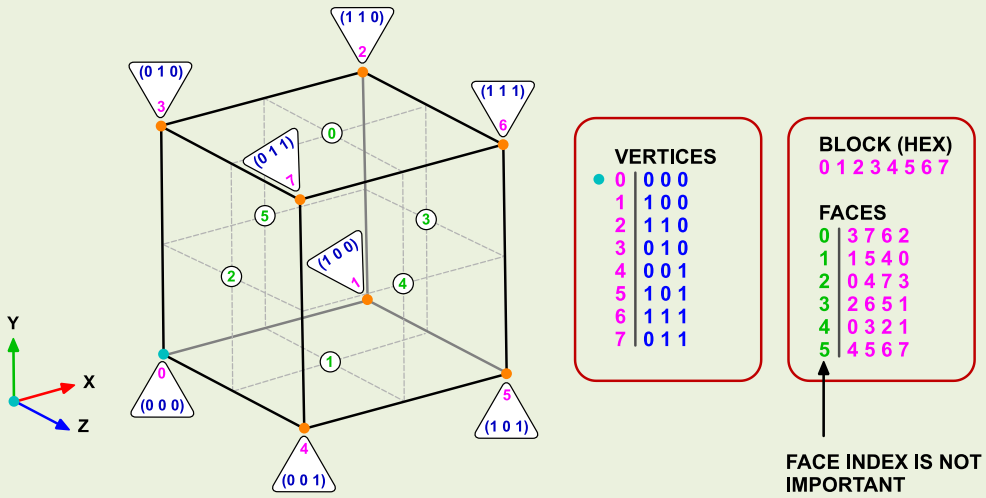
- In lines 60-63, we define the block topology, **hex** means that it is a structured hexahedral block. In this case, we are generating a rectangular mesh.
- **(0 1 2 3 4 5 6 7)** are the vertices used to define the block topology (and yes, the order is important). Each hex block is defined by eight vertices, in sequential order. Where the first vertex in the list represents the origin of the coordinate system.
- **($xcells $ycells $zcells)** is the number of mesh cells in each direction (**X Y Z**). Notice that we are using macro syntax, which is equivalent to **(20 20 1)**.
- **simpleGrading (1 1 1)** is the expansion ratio or mesh stretching in each direction (**X Y Z**), in this case the mesh is uniform.



A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary then defines a block from the vertices:



- Let us talk about the block ordering **hex (0 1 2 3 4 5 6 7)**, which is extremely important.
- **hex** blocks are defined by eight vertices in sequential order. Where the first vertex in the list represents the origin of the coordinate system (vertex **0** in this case).
- Starting from this vertex, we construct the block topology. So in this case, the first part of the block is made up by vertices **0 1 2 3** and the second part of the block is made up by vertices **4 5 6 7**.
- In this case, the vertices are ordered in such a way that if we look at the screen/paper (-**Z** direction), the vertices rotate counter-clockwise.

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary also defines edges:

```
65    edges
66    (
67        //arc 0 1 (0.5 -0.1 0)
68        //arc 4 5 (0.5 -0.1 1)
69    );
```

- Edges, are constructed from the vertices definition.
- Each edge joining two vertex is assumed to be straight by default.
- The user can specified any edge to be curved by entries in the block edges.
- Possible options are: arc, spline, polyline, BSpline, line.
- For example, to define an arc we first define the vertices to be connected to form an edge and then we give an interpolation point.
- In this case and as we do not specified anything, all edges are assumed to be straight lines.
- By the way, lines 67 and 68 are commented.

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary also defines the boundary patches:

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

Name

Type

Connectivity

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

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary also defines the boundary patches:

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

- We can also group many faces into one patch, for example, take a look at lines 86-88. In this case:
  - The name of the patch is **fixedWalls**
  - The **base type** is **wall**
  - The surface patch is made up by the faces **(0 4 7 3)**, **(2 6 5 1),** and **(1 5 4 0)**.
- The base type **empty** (line 93) is used to define a 2D mesh. There is only one cell in the direction of the vector connecting faces **(0 3 2 1)** and **(4 5 6 7)**.
- If you do not define a boundary patch, it will be automatically grouped in the patch **defaultFaces** of type **empty**.
- The name and type of the surface patch can be changed outside of the *blockMeshDict*, we are going to address this later.

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

The *blockMeshDict* dictionary also defines how to merge multiple blocks:

```
102     mergePatchPairs
103     (
104         //(interface1 interface2)
105     );
```

- A mesh can be created using more than 1 block. To do so we proceed in the same way, the only difference is that we need to connect the blocks.
- We can merge blocks in the section **mergePatchPairs** (lines 102-105). This requires that the block patches to be merged are first defined in the **boundary** list (**interface1** and **interface2** in this case), `blockMesh` then connect the two blocks.
- Line 104 is commented. We are not merging blocks, we will talk about this during the meshing session.

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

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

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

A deeper view to my first OpenFOAM® case setup

The *system/blockMeshDict* dictionary

- As you can see, the *blockMeshDict* dictionary can be really tricky.
- If you deal with really easy geometries (rectangles, cylinders, and so on), then you can use `blockMesh` to do the meshing (and by the way you are the luckiest guy in the world), but this is the exception rather than the rule.
- When using `snappyHexMesh`, (a body fitted mesher that comes with OpenFOAM®) you will need to generate a background mesh using `blockMesh`. We are going to deal with this later on.
- Our best advice is to create a template and reuse it until the end of the world.
- Also, take advantage of macro syntax for parametrization, and **#calc** syntax to perform inline calculations (lines 30-35 in the *blockMeshDict* dictionary we just studied).
- We are going to deal with **#codeStream** syntax and **#calc** syntax during the programming session.

The mesher blockMesh has many more features that we did not address in this short overview. Refer to the User Guide for more Information.

A deeper view to my first OpenFOAM® case setup

The `constant/polyMesh/boundary` dictionary

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

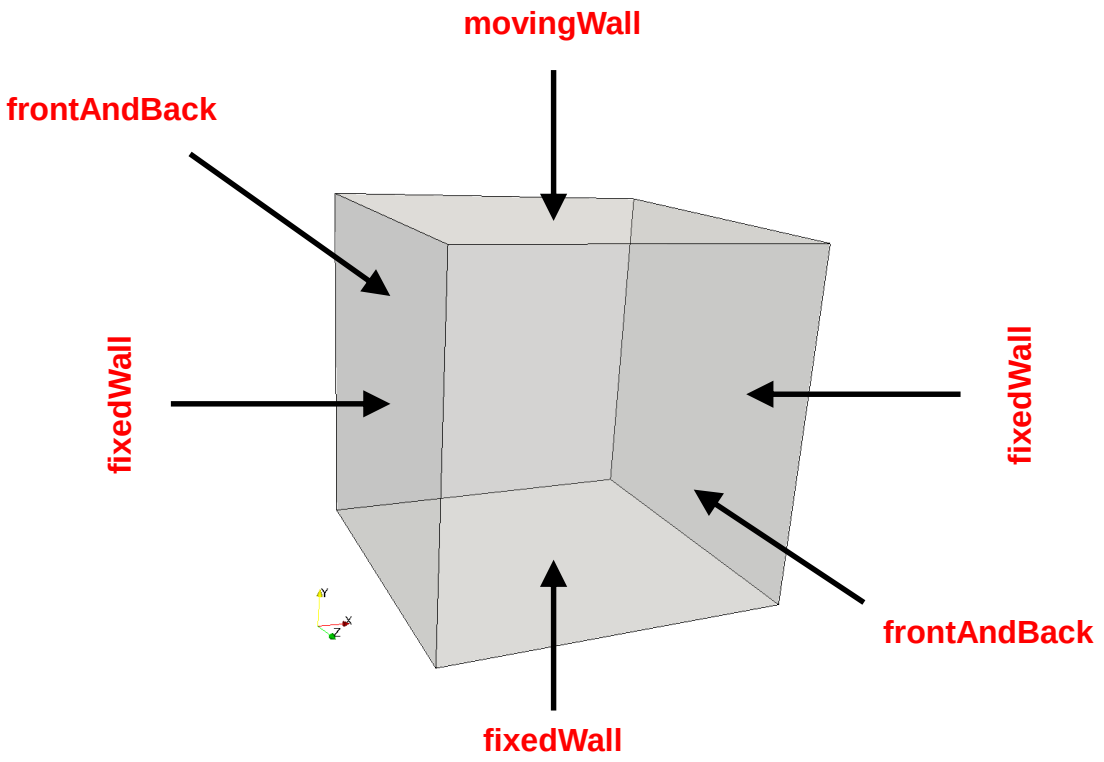
A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

```
18    3
19    (
20        movingWall
21        {
22            type            wall;
23            inGroups        1(wall);
24            nFaces          20;
25            startFace       760;
26        }
27        fixedWalls
28        {
29            type            wall;
30            inGroups        1(wall);
31            nFaces          60;
32            startFace       780;
33        }
34        frontAndBack
35        {
36            type            empty;
37            inGroups        1(empty);
38            nFaces          800;
39            startFace       840;
40        }
41    )
```

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

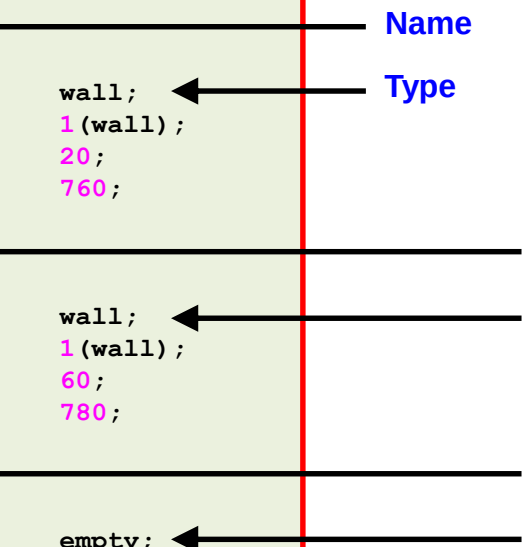


A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

```
18    3
19    (
20        movingWall
21        {
22            type            wall;
23            inGroups        1(wall);
24            nFaces          20;
25            startFace       760;
26        }
27        fixedWalls
28        {
29            type            wall;
30            inGroups        1(wall);
31            nFaces          60;
32            startFace       780;
33        }
34        frontAndBack
35        {
36            type            empty;
37            inGroups        1(empty);
38            nFaces          800;
39            startFace       840;
40        }
41    )
```



Name and type of the surface patches

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

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

```
18    3
19    (
20        movingWall
21        {
22            type            wall;
23            inGroups        1(wall);
24            nFaces          20;
25            startFace       760;
26        }
27        fixedWalls
28        {
29            type            wall;
30            inGroups        1(wall);
31            nFaces          60;
32            startFace       780;
33        }
34        frontAndBack
35        {
36            type            empty;
37            inGroups        1(empty);
38            nFaces          800;
39            startFace       840;
40        }
41    )
```

inGroups keyword

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

  2(wall wall1)

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

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

```
18    3
19    (
20        movingWall
21        {
22            type            wall;
23            inGroups        1(wall);
24            nFaces          20;
25            startFace       760;
26        }
27        fixedWalls
28        {
29            type            wall;
30            inGroups        1(wall);
31            nFaces          60;
32            startFace       780;
33        }
34        frontAndBack
35        {
36            type            empty;
37            inGroups        1(empty);
38            nFaces          800;
39            startFace       840;
40        }
41    )
```

nFaces and startFace keywords

- Unless you know what you are doing, **you do not need to modify this information.**
- Basically, this is telling you the starting face and ending face of the patch.
- This is created automatically when generating the mesh or converting the mesh.

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

```
18    3
19    (
20        movingWall
21        {
22            type            wall;
23            inGroups        1(wall);
24            nFaces          20;
25            startFace       760;
26        }
27        fixedWalls
28        {
29            type            wall;
30            inGroups        1(wall);
31            nFaces          60;
32            startFace       780;
33        }
34        frontAndBack
35        {
36            type            empty;
37            inGroups        1(empty);
38            nFaces          800;
39            startFace       840;
40        }
41    )
```

Remember

Boundary patches that are not recognize or assigned to a patch are grouped automatically in a default group named **defaultFaces** of type **empty**.

For instance, if you do not assign a patch to the patch **frontAndBack**, they will be grouped as follows:

```
defaultFaces
{
  type            empty;
  inGroups        1(empty);
  nFaces          800;
  startFace       840;
}
```

And as usual, you can manually change the name and type.

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

Very important information on the boundary conditions 

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

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

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

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

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

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

constant/polyMesh/boundary	*0/U (IC/BC)*	*0/p (IC/BC)*
wall	**type fixedValue;** **value uniform (U V W);**	**zeroGradient**

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

A deeper view to my first OpenFOAM® case setup

The *constant/polyMesh/boundary* dictionary

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

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

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

A deeper view to my first OpenFOAM® case setup

The **`system`** directory

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

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

A deeper view to my first OpenFOAM® case setup

The *controlDict* dictionary

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

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

A deeper view to my first OpenFOAM® case setup

The *controlDict* dictionary

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

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

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

- So your options are **nextWrite, writeNow, noWriteNow, endTime**
- We love to add bananas, it works with every dictionary.

A deeper view to my first OpenFOAM® case setup

The *controlDict* dictionary

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

- And how do we know that banana does not exist in the source code?
- Just type in the terminal:
  - `$> src`
  - `$> grep -r -n banana .`

- If you see some bananas in your output someone is messing around with your installation.
- You can use the same command to look for information in the OpenFOAM® installation, just replace the word banana for the word you are looking for.
- By the way, you can use any dummy word, but you have to be sure that it does not exist in OpenFOAM®.

A deeper view to my first OpenFOAM® case setup

The *controlDict* dictionary

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

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

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

- This output is just telling you that you are missing the keyword **endTime**.
- Do not pay attention to the words FATAL ERROR, maybe the developers of OpenFOAM® exaggerated a little bit.

A deeper view to my first OpenFOAM® case setup

The *fvSchemes* dictionary

```
17    ddtSchemes
18    {
19        default         backward;
20    }
21
22    gradSchemes
23    {
24        default         Gauss linear;
25        grad(p)         Gauss linear;
26    }
27
28    divSchemes
29    {
30        default         none;
31        div(phi,U)      Gauss linear;
32    }
33
34    laplacianSchemes
35    {
36        default         Gauss linear orthogonal;
37    }
38
39    interpolationSchemes
40    {
41        default         linear;
42    }
43
44    snGradSchemes
45    {
46        default         orthogonal;
47    }
```

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

A deeper view to my first OpenFOAM® case setup

The *fvSolution* dictionary

```
17    solvers
18    {
19        p
20        {
21            solver          PCG;
22            preconditioner  DIC;
23            tolerance       1e-06;
24            relTol          0;

39        }
40
41        pFinal
42        {
43            $p;
44            relTol          0;
45        }
46
47        U
48        {
49            solver          smoothSolver;
50            smoother        symGaussSeidel;
51            tolerance       1e-08;
52            relTol          0;
53        }
54    }
55
56    PISO
57    {
58        nCorrectors     1;
59        nNonOrthogonalCorrectors 0;
60        pRefCell        0;
61        pRefValue       0;
62    }
```

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

A deeper view to my first OpenFOAM® case setup

The *fvSolution* dictionary

```
17    solvers
18    {
19        p
20        {
21            solver          PCG;
22            preconditioner  DIC;
23            tolerance       1e-06;
24            relTol          0;

39        }
40
41        pFinal
42        {
43            $p;
44            relTol          0;
45        }
46
47        U
48        {
49            solver          smoothSolver;
50            smoother        symGaussSeidel;
51            tolerance       1e-08;
52            relTol          0;
53        }
54    }
55
56    PISO
57    {
58        nCorrectors     1;
59        nNonOrthogonalCcorrectors 0;
60        pRefCell        0;
61        pRefValue       0;
62    }
```

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

A deeper view to my first OpenFOAM® case setup

The **`system`** directory
(optional dictionary files)

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

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

A deeper view to my first OpenFOAM® case setup

The *sampleDict* dictionary

```
17    type sets;
18
19    setFormat raw;
20
23    interpolationScheme cellPointFace;
24
26    fields
27    (
28        U
29    );
30
31    sets
32    (
33
34        l1
35        {
36            type            midPointAndFace;
43            axis            x;
44            start           ( -1  0.5 0);
45            end             ( 2  0.5 0);
46        }
47
48        l2
49        {
50            type            midPointAndFace;
54            axis            y;
55            start           (0.5 -1 0);
56            end             (0.5 2 0);
57        }
58
59    );
```

Type of sampling, sets will sample along a line.

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

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

Fields to sample.

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

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

Sample method from the solution to the line.

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

A deeper view to my first OpenFOAM® case setup

The *sampleDict* dictionary

```
17    type sets;
18
19    setFormat raw;
20
23    interpolationScheme cellPointFace;
24
26    fields
27    (
28        U
29    );
30
31    sets
32    (
33
34        l1
35        {
36            type            midPointAndFace;
43            axis            x;
44            start           ( -1  0.5 0);
45            end             ( 2  0.5 0);
46        }
47
48        l2
49        {
50            type            midPointAndFace;
54            axis            y;
55            start           (0.5 -1 0);
56            end             (0.5 2 0);
57        }
58
59    );
```

The sampled information is always saved in the directory,

postProcessing/name_of_input_dictionary

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

postProcessing/sampleDict/50

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

Name of the output file

Name of the output file

A deeper view to my first OpenFOAM® case setup

The 0 directory

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

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

```
17    dimensions      [0 1 -1 0 0 0 0];
18
19    internalField   uniform (0 0 0);
20
21    boundaryField
22    {
23        movingWall
24        {
25            type            fixedValue;
26            value           uniform (1 0 0);
27        }
28
29        fixedWalls
30        {
31            type            fixedValue;
32            value           uniform (0 0 0);
33        }
34
35        frontAndBack
36        {
37            type            empty;
38        }
39    }
```

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

Uniform initial conditions.

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

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

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

A deeper view to my first OpenFOAM® case setup

The 0 directory

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

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

```
17    dimensions      [0 1 -1 0 0 0 0];
18
19    internalField   uniform (0 0 0);
20
21    boundaryField
22    {
23        movingWall
24        {
25            type            fixedValue;
26            value           uniform (1 0 0);
27        }
28
29        fixedWalls
30        {
31            type            fixedValue;
32            value           uniform (0 0 0);
33        }
34
35        frontAndBack
36        {
37            type            empty;
38        }
39    }
```

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

Numerical boundary condition for the patch **movingWall**

Numerical boundary condition for the patch **fixedWalls**

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

A deeper view to my first OpenFOAM® case setup

The 0 directory

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

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

```
17    dimensions      [0 2 -2 0 0 0 0];
18
19    internalField   uniform 0;
20
21    boundaryField
22    {
23        movingWall
24        {
25            type            zeroGradient;
26        }
27
28        fixedWalls
29        {
30            type            zeroGradient;
31        }
32
33        frontAndBack
34        {
35            type            empty;
36        }
37    }
38
```

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

Uniform initial conditions.

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

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

A deeper view to my first OpenFOAM® case setup

The 0 directory

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

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

```
17    dimensions      [0 2 -2 0 0 0 0];
18
19    internalField   uniform 0;
20
21    boundaryField
22    {
23        movingWall
24        {
25            type            zeroGradient;
26        }
27
28        fixedWalls
29        {
30            type            zeroGradient;
31        }
32
33        frontAndBack
34        {
35            type            empty;
36        }
37    }
38
```

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

Numerical boundary condition for the patch **movingWall**

Numerical boundary condition for the patch **fixedWalls**

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

A deeper view to my first OpenFOAM® case setup

A very important remark on the pressure field

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

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

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

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

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

A deeper view to my first OpenFOAM® case setup

- Coming back to the headers, and specifically the headers related to the field variable dictionaries (e.g. *U*, *p*).
- In the header of your field variables, the class type should be consistent with the type of field variable you are using.
- If the field variable is a scalar, the class should be **volScalarField**.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.x                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

A deeper view to my first OpenFOAM® case setup

- Coming back to the headers, and specifically the headers related to the field variable dictionaries (e.g. *U*, *p*).
- In the header of your field variables, the class type should be consistent with the type of field variable you are using.
- If the field variable is a vector, the class should be **volVectorField.**

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.x                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

A deeper view to my first OpenFOAM® case setup

- Coming back to the headers, and specifically the headers related to the field variable dictionaries (e.g. U, p).
- In the header of your field variables, the class type should be consistent with the type of field variable you are using.
- If the field variable is a tensor (e.g. the velocity gradient tensor), the class should be **volTensorField**.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.x                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volTensorField;
    object      gradU;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

A deeper view to my first OpenFOAM® case setup

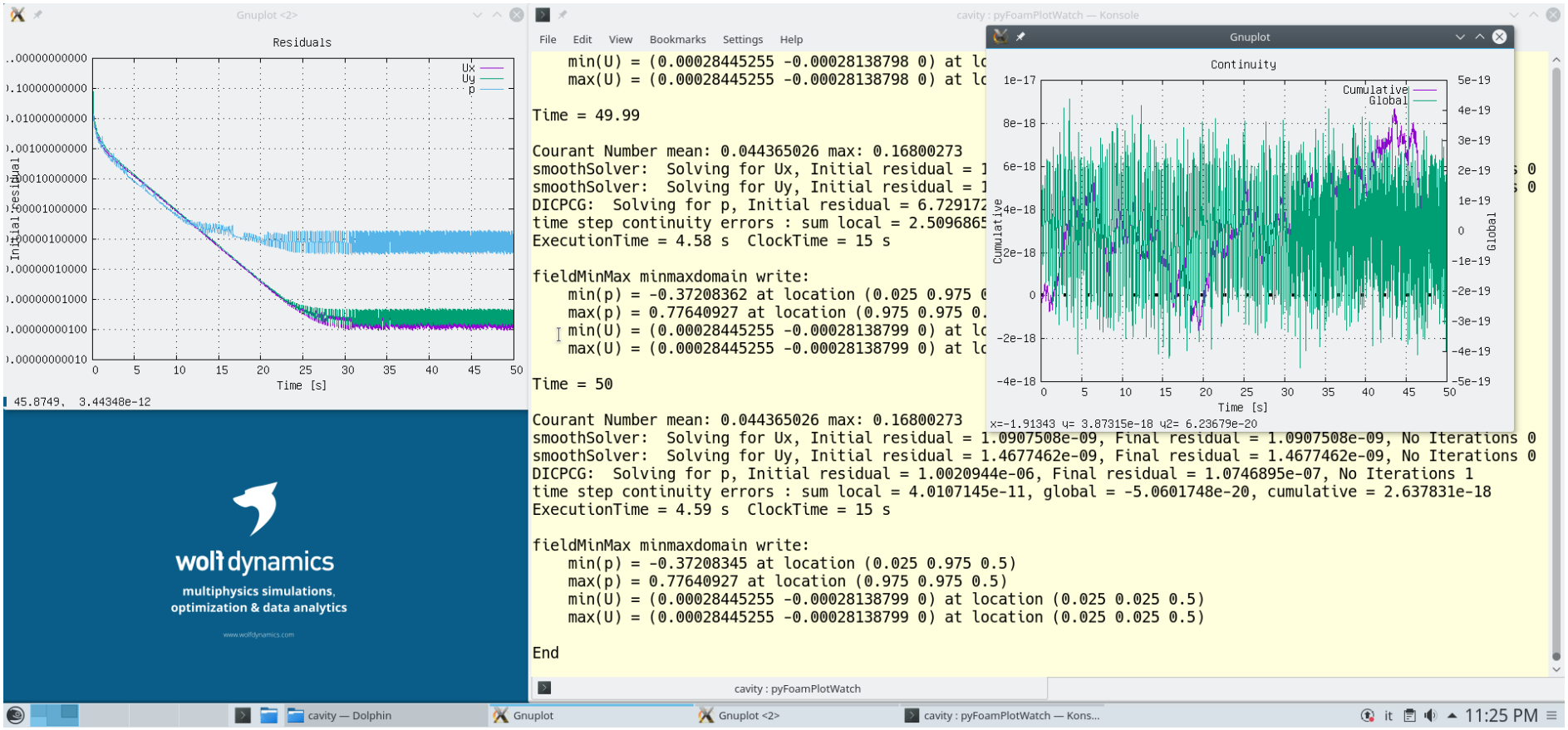
The *log.icoFoam* file

- If you followed the previous instructions you should now have the *log.icoFoam* file. This file contains all the residuals and convergence information.
- We already plotted this information using `foamLog` and `gnuplot`.
- Let us plot this information again but this time using PyFoam, in the terminal type:
  - `$> pyFoamPlotWatcher.py log.icoFoam --with-courant`
- The script `pyFoamPlotWatcher.py` will plot the information in the log file, even if the simulation is not running.
- The option `--with-courant` will plot the courant number.
- Remember, to use PyFoam you will need to source it. Type in the terminal:
  - `$> anaconda2` or `$> anaconda3`

A deeper view to my first OpenFOAM® case setup

The *log.icoFoam* file

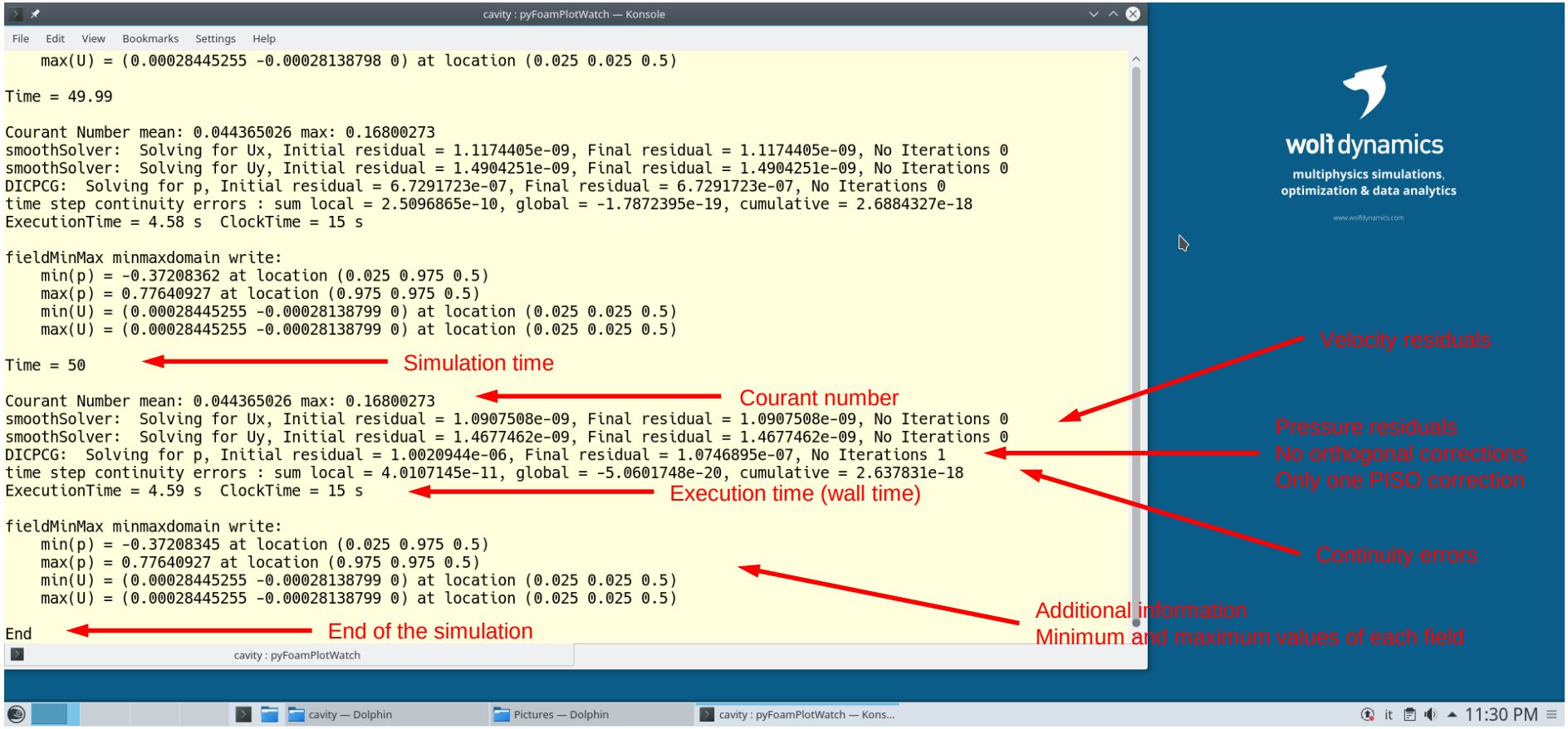
- In this case, `pyFoamPlotWatcher` is plotting the initial residuals, continuity errors and courant number.



A deeper view to my first OpenFOAM® case setup

The output screen

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



A deeper view to my first OpenFOAM® case setup

The output screen

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

```
49    functions
50    {
51
52    ////////////////////////////////////////////////////////////////////////
53
54    minmaxdomain
55    {
56        type fieldMinMax;
57
58        functionObjectLibs ("libfieldFunctionObjects.so");
59
60        enabled true; //true or false
61
62        mode component;
63
64        writeControl timeStep;
65        writeInterval 1;
66
67        log true;
68
69        fields (p U);
70    }
91
92    };
```

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

functionObject to use

Turn on/off functionObject

Output interval of functionObject

Save output of the functionObject in a ascii file

Field variables to sample

A deeper view to my first OpenFOAM® case setup

The output screen

- Another very important output information is the CFL or Courant number.
- In one dimension, the CFL number is defined as,

$$CFL = \frac{u\,\Delta t}{\Delta x}$$

- The CFL number is a measure of how much information (u) traverses a computational grid cell (Δx) in a given time-step (Δt).
- The Courant number imposes the **CFL number condition,** which is the maximum allowable CFL number a numerical scheme can use. For the n - dimensional case, the CFL number condition becomes,

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

A deeper view to my first OpenFOAM® case setup

The output screen

- The CFL number is a necessary condition to guarantee the stability of the numerical method.
- But not all numerical methods have the same stability constrains.
- By the way, when we talk about numerical methods we are referring to implicit and explicit methods.
- In OpenFOAM®, most of the solvers are implicit, which means they are **unconditionally stable**. In other words, they are not constrained to the **CFL number condition**.
- However, the fact that you are using a numerical method that is **unconditionally stable**, does not mean that you can choose a time step of any size.
- The time-step must be chosen in such a way that it resolves the time-dependent features, and it maintains the solver stability.
- For the moment and for the sake of simplicity, let us try to keep the CFL number below 2.0 and preferably less than 1.0
- Other properties of the numerical method that you should observe are: conservationess, boundedness, transportiveness, and accuracy. We are going to address these properties and the CFL number when we deal with the FVM theory.

A deeper view to my first OpenFOAM® case setup

The output screen

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

```
Time = 49.99

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

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

Time = 50

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

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

CFL number at time step n - 1

CFL number at time step n

A deeper view to my first OpenFOAM® case setup

The output screen

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

```
Time = 49.9

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

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

Time = 50

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

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

CFL number at time step n - 1

CFL number at time step n

A deeper view to my first OpenFOAM® case setup

The output screen

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

```
Time = 2

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

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

Time = 2.5

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

CFL number at time step n - 1

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

CFL number at time step n (way too high)

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

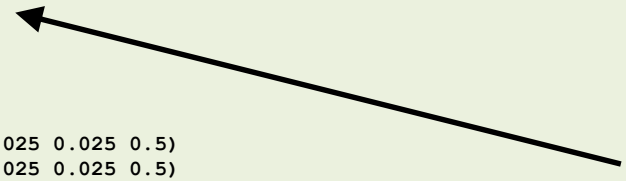
A deeper view to my first OpenFOAM® case setup

The output screen

- Another output you should monitor are the continuity errors.
- These numbers should be small (it does not matter if they are negative or positive).
- If these values increase in time (about the order of 1e-3), you better control the case setup because something is wrong.
- The continuity errors are defined in the following file

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

```
Time = 50

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

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



A deeper view to my first OpenFOAM® case setup

Error output

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

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.x                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 4.x-5d8318b22cbe
Exec   : icoFoam
Date   : Nov 02 2014
Time   : 00:33:41
Host   : "linux-cfd"
PID    : 3675
Case   : /home/cfd/my_cases_course/cavity
nProcs : 1
sigFpe : Enabling floating point exception trapping (FOAM_SIGFPE).
fileModificationChecking : Monitoring run-time modified files using timeStampMaster
allowSystemOperations : Allowing user-supplied system call operations

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

--> FOAM FATAL IO ERROR:
```

A deeper view to my first OpenFOAM® case setup

Error output

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

```
Build  : 4.x-5d8318b22cbe
Exec   : icoFoam
Date   : Nov 02 2014
Time   : 00:33:41
Host   : "linux-cfd"
PID    : 3675
Case   : /home/cfd/my_cases_course/cavity
nProcs : 1
sigFpe : Enabling floating point exception trapping (FOAM_SIGFPE).
fileModificationChecking : Monitoring run-time modified files using timeStampMaster
allowSystemOperations : Allowing user-supplied system call operations

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


--> FOAM FATAL IO ERROR:

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

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

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

FOAM exiting
```

The origin of the error

Possible options to correct the error

Location of the error

A deeper view to my first OpenFOAM® case setup

Error output

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

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

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

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

file: /home/joegi/my_cases_course/4x/101OF/cavity/0/p at line 0.

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

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- Missing keyword **class** in file *p*

```
--> FOAM FATAL IO ERROR:
keyword class is undefined in dictionary "/home/joegi/my_cases_course/4x/101OF/cavity/0/p"

file: /home/joegi/my_cases_course/4x/101OF/cavity/0/p from line 10 to line 13.

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

FOAM exiting
```

- Misspelled word in file *boundary*

```
--> FOAM FATAL IO ERROR:
unexpected class name spolyBoundaryMesh expected polyBoundaryMesh
    while reading object boundary

file: /home/joegi/my_cases_course/4x/101OF/cavity/constant/polyMesh/boundary at line 15.

    From function regIOobject::readStream(const word&)
    in file db/regIOobject/regIOobjectRead.C at line 136.

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- Mismatching patch name in file *p*

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

file: /home/joegi/my_cases_course/4x/101OF/cavity/0/p.boundaryField from line 25 to line 35.

    From function GeometricField<Type, PatchField, GeoMesh>::GeometricBoundaryField::readField(const
DimensionedField<Type, GeoMesh>&, const dictionary&)
    in file /home/joegi/OpenFOAM/OpenFOAM-4.x/src/OpenFOAM/lnInclude/GeometricBoundaryField.C at line 209.

FOAM exiting
```

- Missing compulsory keyword in *fvSchemes*

```
--> FOAM FATAL IO ERROR:
keyword div(phi,U) is undefined in dictionary
"/home/joegi/my_cases_course/4x/101OF/cavity/system/fvSchemes.divSchemes"

file: /home/joegi/my_cases_course/4x/101OF/cavity/system/fvSchemes.divSchemes from line 30 to line 30.

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

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

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

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

file: /home/joegi/my_cases_course/4x/101OF/cavity/system/fvSolution at line 68.

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

FOAM exiting
```

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

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

    From function checkMethod(const fvMatrix<Type>&, const fvMatrix<Type>&)
    in file /home/joegi/OpenFOAM/OpenFOAM-4.x/src/finiteVolume/lnInclude/fvMatrix.C at line 1295.

FOAM aborting

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

A deeper view to my first OpenFOAM® case setup

Error output

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

```
--> FOAM FATAL IO ERROR:
keyword deltaT is undefined in dictionary "/home/joegi/my_cases_course/4x/101OF/cavity/system/controlDict"

file: /home/joegi/my_cases_course/4x/101OF/cavity/system/controlDict from line 17 to line 69.

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

FOAM exiting
```

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

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

    From function Time::findInstance(const fileName&, const word&, const IOobject::readOption, const word&)
    in file db/Time/findInstance.C at line 203.

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- Unknown boundary condition type.

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

Valid patchField types are :

74
(
SRFFreestreamVelocity
SRFVelocity
SRFWallVelocity
activeBaffleVelocity

...
...
...

variableHeightFlowRateInletVelocity
waveTransmissive
wedge
zeroGradient
)


file: /home/joegi/my_cases_course/4x/101OF/cavity/0/U.boundaryField.movingWall from line 25 to line 26.

    From function fvPatchField<Type>::New(const fvPatch&, const DimensionedField<Type, volMesh>&, const
dictionary&)
    in file /home/joegi/OpenFOAM/OpenFOAM-4.x/src/finiteVolume/lnInclude/fvPatchFieldNew.C at line 143.

FOAM exiting
```

A deeper view to my first OpenFOAM® case setup

Error output

- This one is specially hard to spot

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  4.x                                   |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 4.x-5d8318b22cbe
Exec   : icoFoam
Date   : Nov 02 2014
Time   : 00:33:41
Host   : "linux-cfd"
PID    : 3675
fileName::stripInvalid() called for invalid fileName /home/cfd/my_cases_course/cavity0
    For debug level (= 2) > 1 this is considerd fatal
Aborted
```

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

- Instead of **`cavity 0`** you should use **`cavity_0`**.

A deeper view to my first OpenFOAM® case setup

Error output

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

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

```
// This is a line comment

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

- The **#include** directive (same as in C++):

```
#include "initialConditions"
```

Do not forget to create the respective include file *`initialConditions`*.

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

    3.14159

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

    (1.0 0.0 0.0)

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

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

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

    - A list of vectors is represented as follows:

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

```
name_of_the_list
(
    "word1"
    "word2"
    "word3"
);
```

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

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

internalField   uniform $vectorField;     //Use declared variable


scalarField     101328;                   //Declare variable

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

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


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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

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

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

- With inline calculations you can access all the mathematical functions available in C++.
- Macro expansions and inline calculations are really useful to parametrize dictionaries and avoid repetitive tasks.

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

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

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- You can write (the lazy way):

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

- You could also try (even lazier):

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

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

A deeper view to my first OpenFOAM® case setup

Dictionary files general features

- Switches: they are used to enable or disable a function or a feature in the dictionaries.
- Switches are logical values. You can use the following values:

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

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

  OpenFOAM-4.x/src/OpenFOAM/primitives/bools/Switch/Switch.C

A deeper view to my first OpenFOAM® case setup

Solvers and utilities help

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

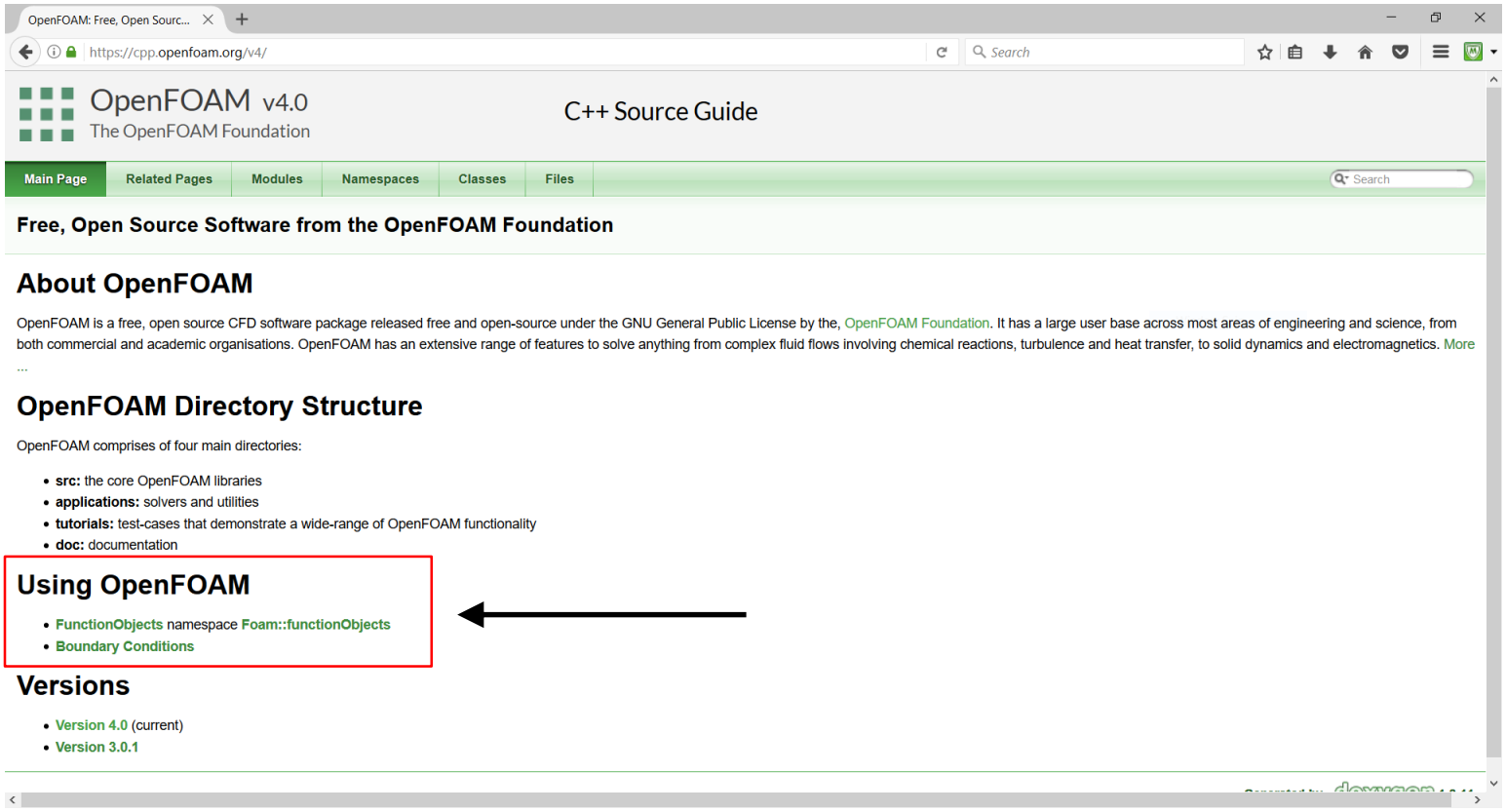
  will print some basic help and usage information about `icoFoam`

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

A deeper view to my first OpenFOAM® case setup

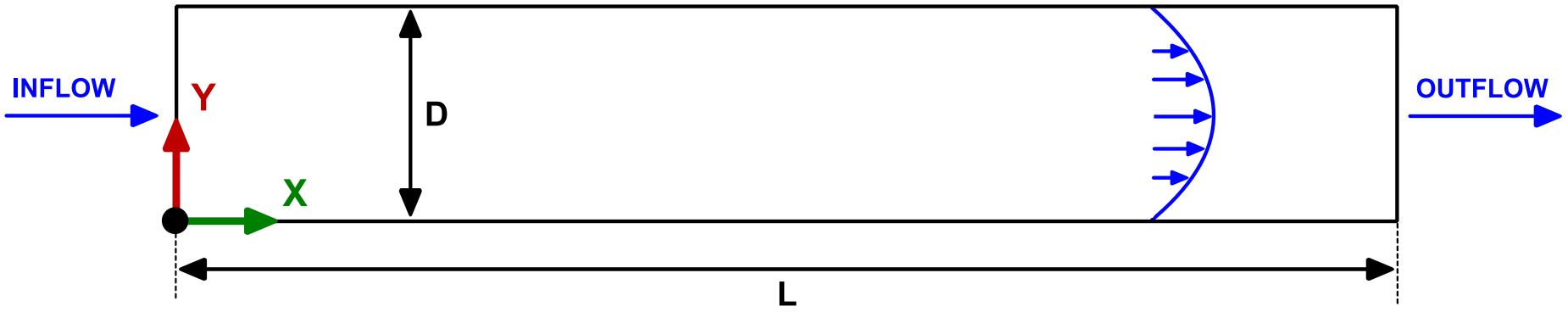
Solvers and utilities help

- To get more information about the boundary conditions and post-processing utilities available in OpenFOAM®, please read the Doxygen documentation. Just look for the **Using OpenFOAM** section at the bottom of the page.
- If you did not compile the Doxygen documentation, you can access the information online, http://cpp.openfoam.org/v4/



A simple validation case – Hagen-Poiseuille solution

Hagen-Poiseuille solution – Re = 100
Incompressible flow



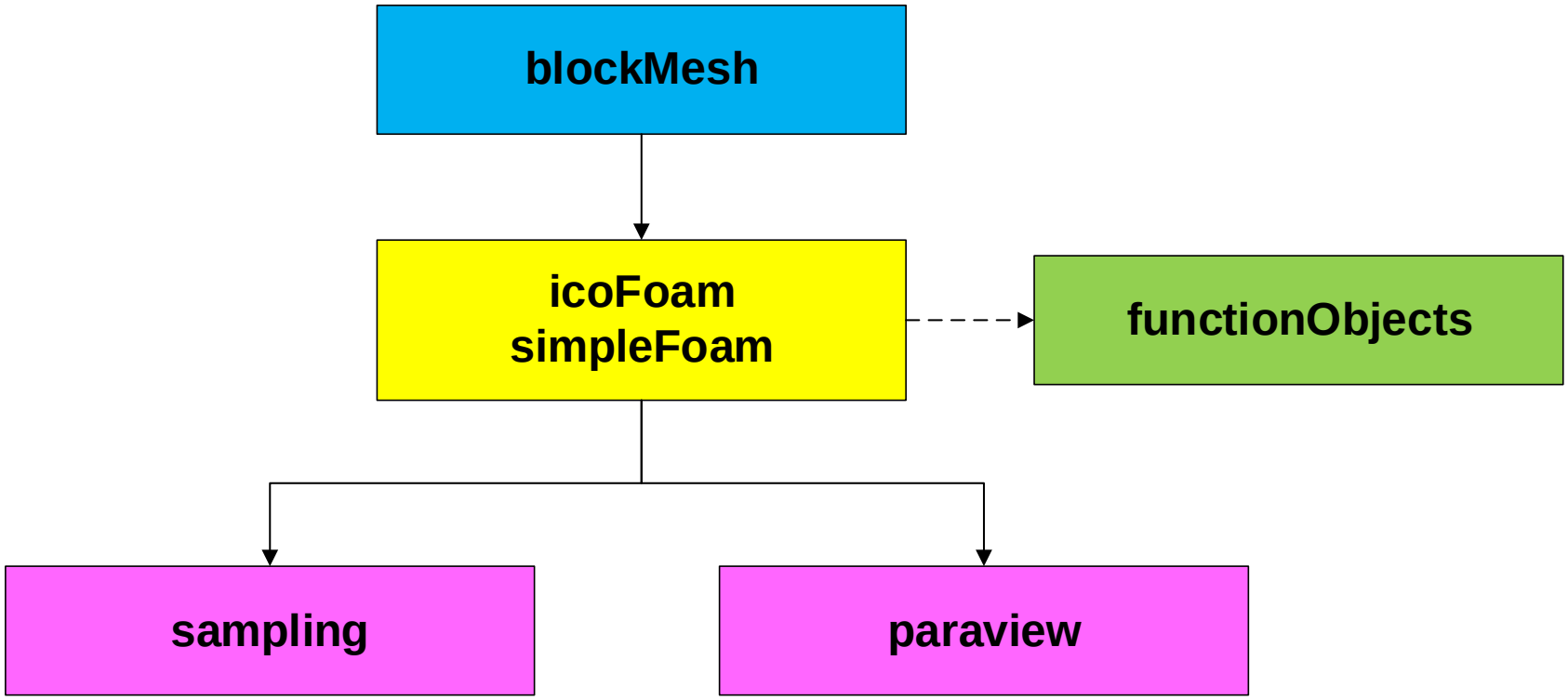
Physical and numerical side of the problem:

- The governing equations of the problem are the incompressible laminar Navier-Stokes equations.
- We are going to work in a 2D domain but the problem can be extended to 3D or axisymmetric problems easily.
- This problem has an analytical solution for the parabolic velocity profile

$$u = u_{max}\left[1 - \left(\frac{r}{r_{pipe}}\right)^2\right]$$

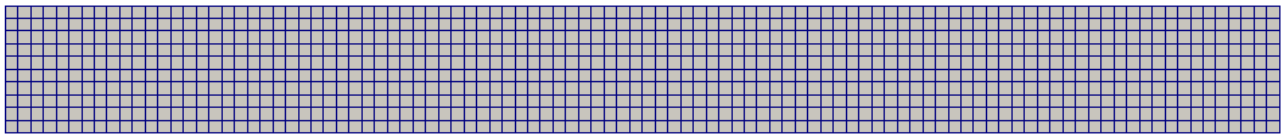
A simple validation case – Hagen-Poiseuille solution

Workflow of the case



A simple validation case – Hagen-Poiseuille solution

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

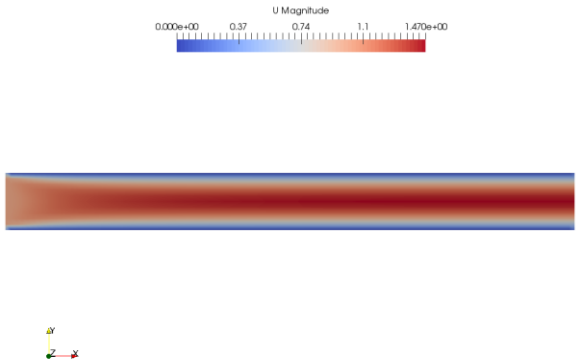


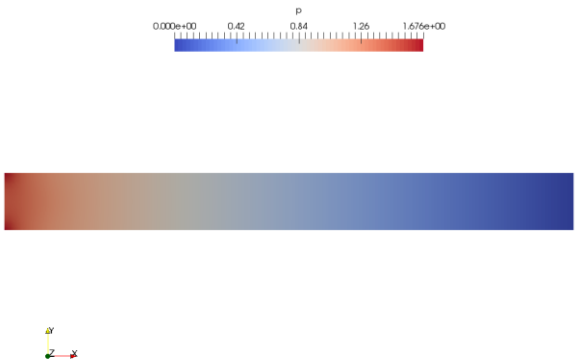
Mesh (coarse add 2D)

- This mesh is very coarse but for the physics involved it works fine.
- You can try to do successive refinements of this mesh in order to do a mesh independency study.
- If you deal with turbulence, you will need to refine the mesh close to the walls in order to resolve the oudnayr layers.

A simple validation case – Hagen-Poiseuille solution

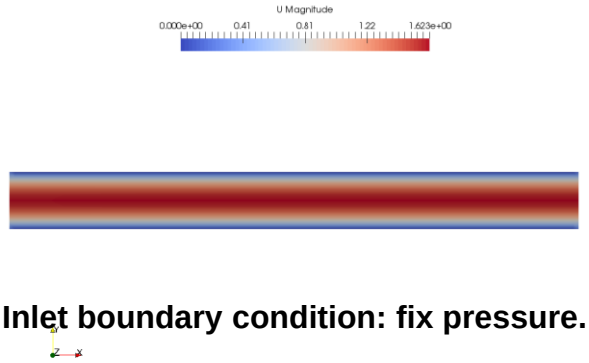
At the end of the day you should get something like this

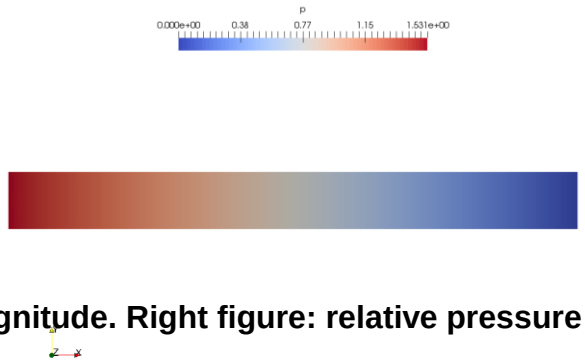




Inlet boundary condition: fix uniform velocity. Left figure: velocity magnitude. Right figure: relative pressure

---

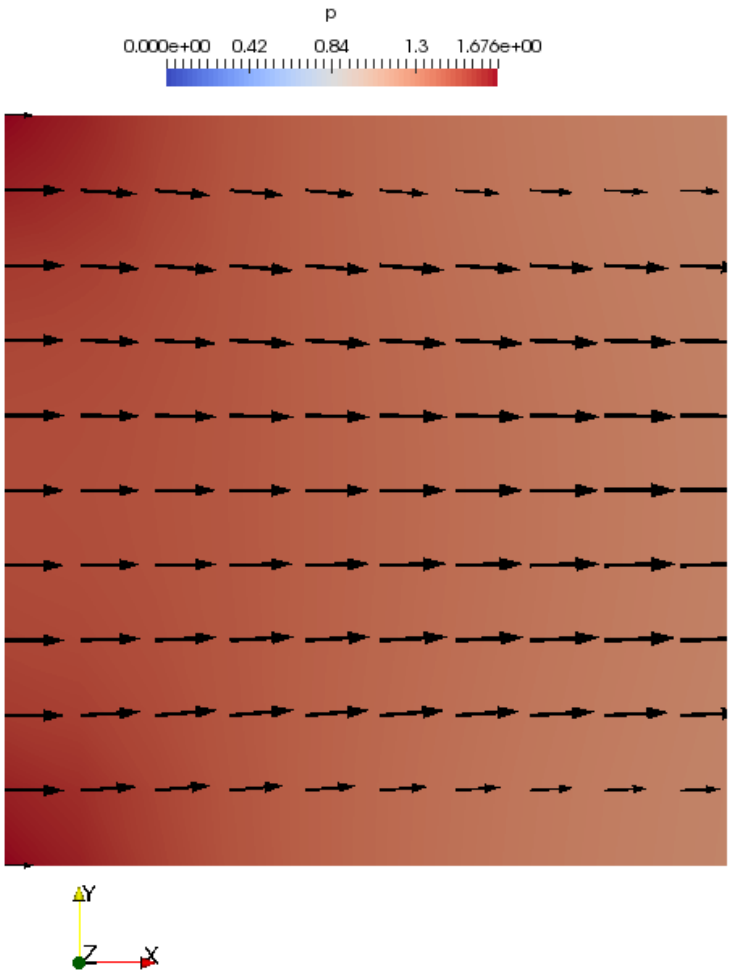




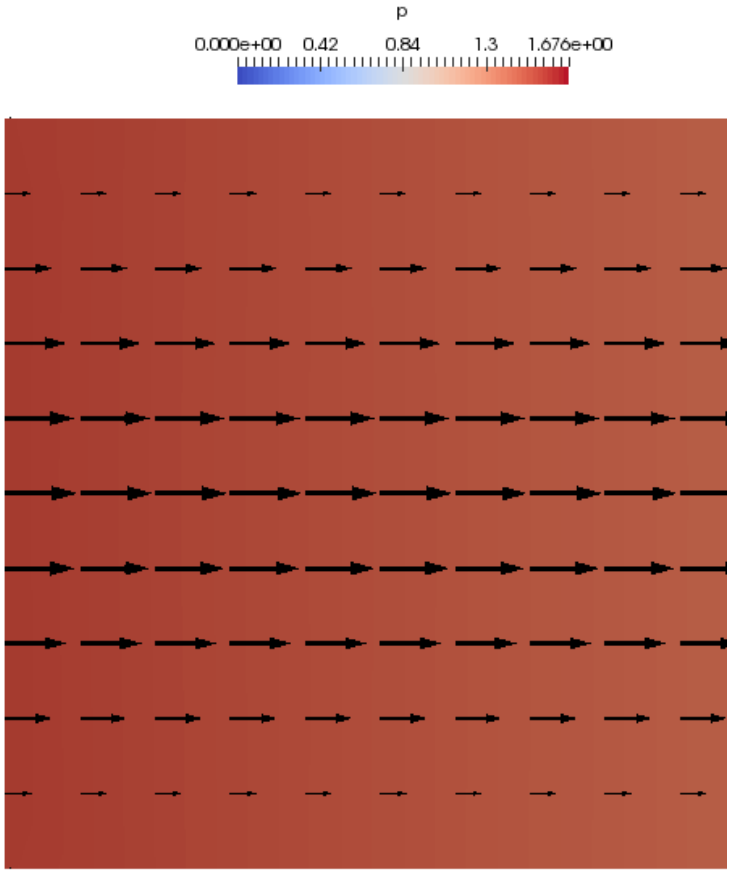
Inlet boundary condition: fix pressure. Left figure: velocity magnitude. Right figure: relative pressure

A simple validation case – Hagen-Poiseuille solution

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



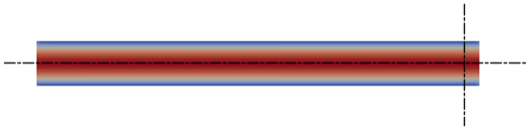
Velocity profile at the inlet
BC: fix uniform velocity



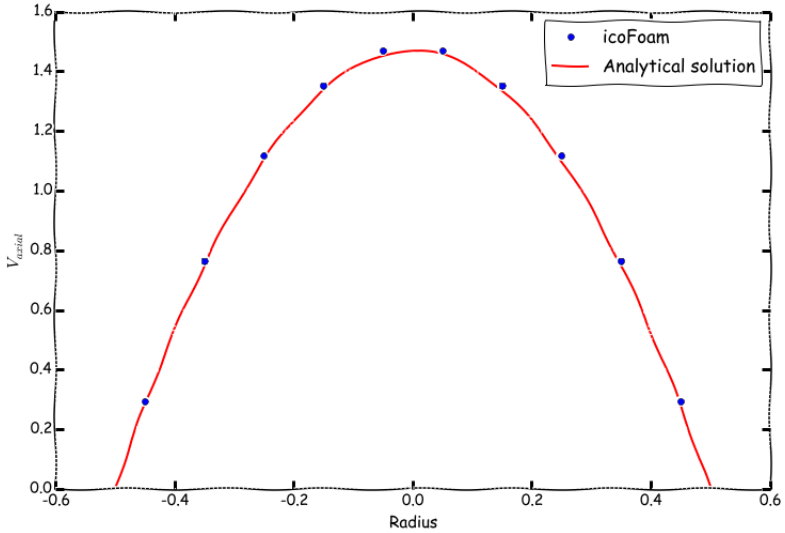
Velocity profile at the inlet
BC: fix pressure

A simple validation case – Hagen-Poiseuille solution

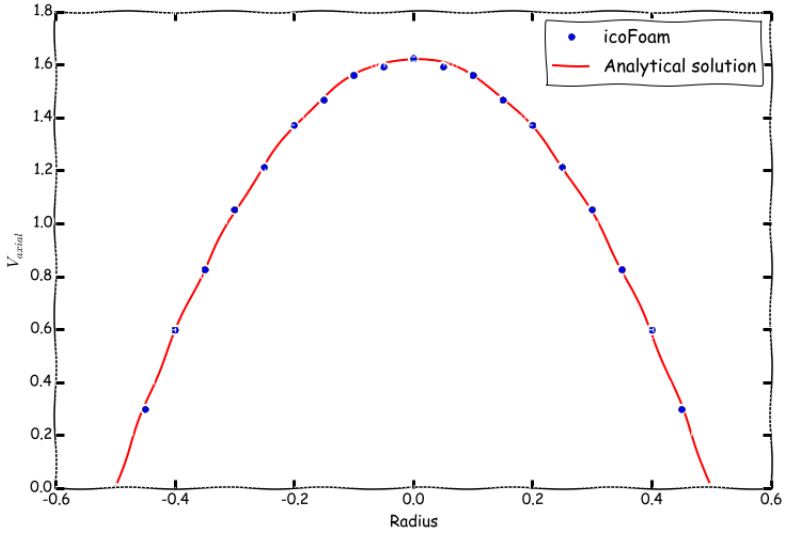
At the end of the day you should get something like this



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



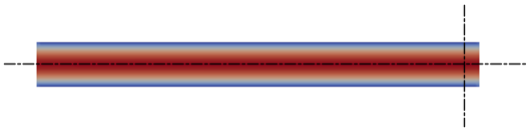
Velocity profile at the outlet
BC: fix uniform velocity



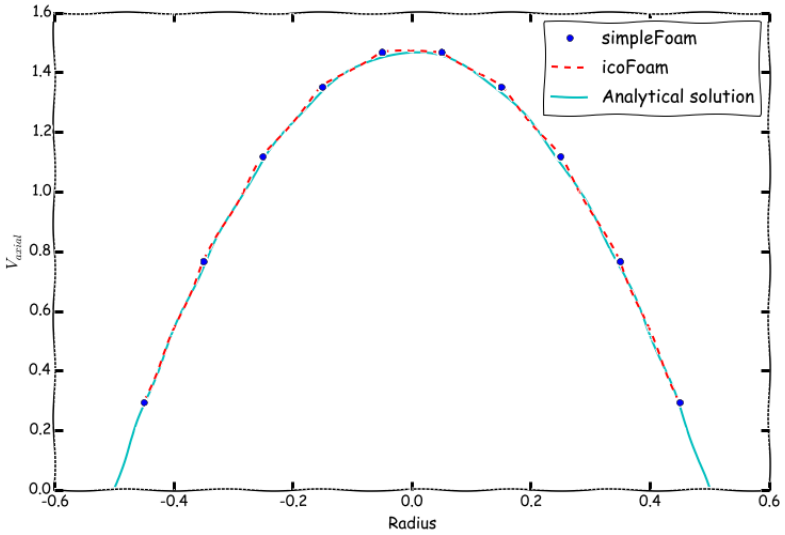
Velocity profile at the outlet
BC: fix pressure

A simple validation case – Hagen-Poiseuille solution

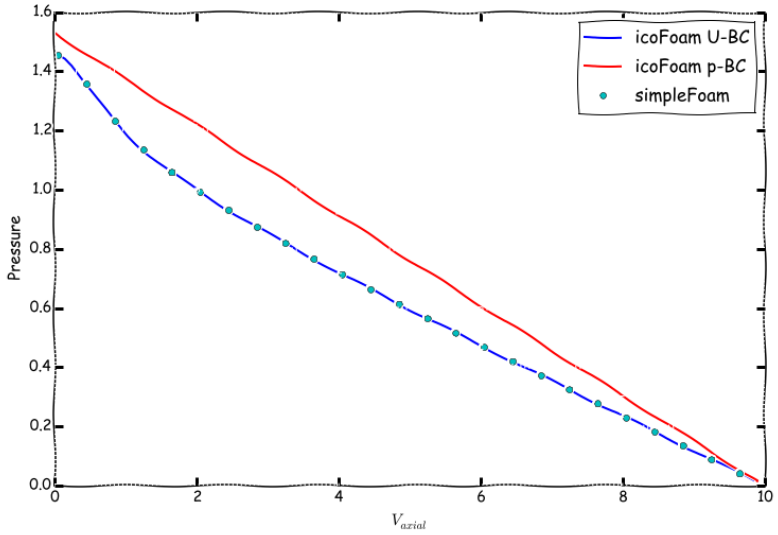
At the end of the day you should get something like this



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



Velocity profile at the outlet
simpleFoam vs. icoFoam vs. analytical solution



Pressure along the axis of the pipe
Comparison of the three cases
(icoFoam BC1, icoFoam BC2, simpleFoam)

A simple validation case – Hagen-Poiseuille solution

Loading OpenFOAM® environment

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

- To source OpenFOAM®**,** type in the terminal:

    - `$> of4x`

- To use PyFoam you will need to source it. Type in the terminal:

    - `$> anaconda2` or `anaconda3`

- Remember, every time you open a new terminal window you need to source OpenFOAM® and PyFoam.

- By default, when installing OpenFOAM® and PyFoam you do not need to do this. This is our choice as we have many things installed and we want to avoid conflicts between applications.

A simple validation case – Hagen-Poiseuille solution

What are we going to do?

- We will use this case to compare the numerical solution with the analytical solution.
- We will compare the solutions obtained when using different inlet boundary conditions.
- To find the numerical solution we will use two different solvers, namely, `icoFoam` and `simpleFoam`.
- `icoFoam` is a transient solver for incompressible, laminar flow of Newtonian fluids.
- `simpleFoam` is a steady-state solver for incompressible, laminar/turbulent flows.
- After finding the numerical solution we will do some sampling.
- Then we will do some plotting (using gnuplot or Python) and scientific visualization.

A simple validation case – Hagen-Poiseuille solution

Let us explore the case directory

A simple validation case – Hagen-Poiseuille solution

The *blockMeshDict* dictionary file

```
17      convertToMeters 1;
18
19      xmin 0;
20      xmax 10;
21      ymin -0.5;
22      ymax 0.5;
23      zmin 0;
24      zmax 0.1;
25
26      vertices
27      (
28          ($xmin   $ymin   $zmin)       //vertex 0
29          ($xmax   $ymin   $zmin)       //vertex 1
30          ($xmax   $ymax   $zmin)       //vertex 2
31          ($xmin   $ymax   $zmin)       //vertex 3
32          ($xmin   $ymin   $zmax)       //vertex 4
33          ($xmax   $ymin   $zmax)       //vertex 5
34          ($xmax   $ymax   $zmax)       //vertex 6
35          ($xmin   $ymax   $zmax)       //vertex 7
36      );
37
38      blocks
39      (
40          hex (0 1 2 3 4 5 6 7) (100 10 1)
            simpleGrading (1 1 1)
41      );
42
43     edges
44     (
45     );
```

- This dictionary is located in the **`system`** directory.
- We are not using scaling.
- **X**/**Y**/**Z** dimensions: **10.0**/**1.0**/**0.1**
- We are using one single block with uniform grading.
- Cells in the **X**, **Y**, and **Z** directions: **100** x **10** x **1** (there is only one cell in the **Z** direction because the mesh is 2D).
- All edges are straight lines by default.

A simple validation case – Hagen-Poiseuille solution

The *blockMeshDict* dictionary file

```
47      boundary
48      (
49          top
50          {
51              type wall;
52              faces
53              (
54                  (3 7 6 2)
55              );
56          }
57          inlet
58          {
59              type patch;
60              faces
61              (
62                  (0 4 7 3)
63              );
64          }
65          outlet
66          {
67              type patch;
68              faces
69              (
70                  (2 6 5 1)
71              );
72          }
73          bottom
74          {
75              type wall;
76              faces
77              (
78                  (1 5 4 0)
79              );
80          }
```

- The boundary patches **top** and **bottom** are of **base type** wall.
- The boundary patches **outlet** and **inlet** are of **base type** patch.
- Later on, we will assign the **primitive type** boundary conditions (numerical values), in the field files found in the directory *0*

A simple validation case – Hagen-Poiseuille solution

The *blockMeshDict* dictionary file

```
81          back
82          {
83              type empty;
84              faces
85              (
86                  (0 3 2 1)
87              );
88          }
89          front
90          {
91              type empty;
92              faces
93              (
94                  (4 5 6 7)
95              );
96          }
97      );
98
99      mergePatchPairs
100     (
101     );
```

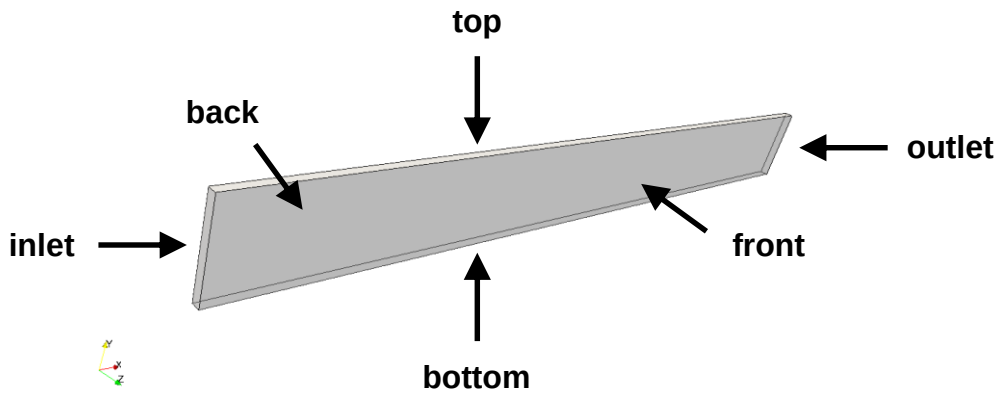
- The boundary patches **back** and **front** are of **base type** empty.
- Later on, we will assign the **primitive type** boundary conditions (numerical values), in the field files found in the directory *0*
- We do not need to merge faces (we have one single block).

A simple validation case – Hagen-Poiseuille solution

The *boundary* dictionary file

```
18    6
19    (
20        top
21        {
22            type            wall;
24            nFaces          100;
25            startFace       1890;
26        }
27        inlet
28        {
29            type            patch;
30            nFaces          10;
31            startFace       1990;
32        }
33        outlet
34        {
35            type            patch;
36            nFaces          10;
37            startFace       2000;
38        }
39        bottom
40        {
41            type            wall;
43            nFaces          100;
44            startFace       2010;
45        }
46        back
47        {
48            type            empty;
50            nFaces          1000;
51            startFace       2110;
52        }
53        front
54        {
55            type            empty;
57            nFaces          1000;
58            startFace       3110;
59        }
60    )
```

- This dictionary is located in the **`constant/polyMesh`** directory.
- This file was automatically created when generating the mesh.
- In this case, we do not need to modify this file. All the **base type** boundary conditions and **name** of the patches were assigned in the *`blockMeshDict`* file.
- In you change the **name** or the **base type** of a boundary patch, you will need to modify the field files in the directory **`0`**.



A simple validation case – Hagen-Poiseuille solution

The *transportProperties* dictionary file

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

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

- You can change this value on-the-fly.
- Reminder:
  - The pipe diameter and length are 0.5 m and 10 m, respectively.
  - And we are targeting for a Re = 100.

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

A simple validation case – Hagen-Poiseuille solution

The **0** directory

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

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

A simple validation case – Hagen-Poiseuille solution

The file *0/p*

```
19    internalField   uniform 0;
20    //internalField   uniform 101325;
21
22    boundaryField
23    {
24        inlet
25        {
26            type            zeroGradient;
27        }
28
29        outlet
30        {
31            type            fixedValue;
32            value           $internalField;
33
34            //type          zeroGradient;
35        }
36
37        top
38        {
39            type          zeroGradient;
40        }
```

- We are using uniform initial conditions and the numerical value is 0 (keyword **internalField** in line 19). **This is relative pressure.**
- For the **inlet** patch (lines 24-27), we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).
- For the **outlet** patch (lines 29-35), we are using a **fixedValue** boundary condition with a numerical value equal to 0**.** Notice that we are using macro expansion to assign the numerical value (**$internalField** is equivalent to **uniform 0**).
- For the **top** patch (lines 37-40), we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).

A simple validation case – Hagen-Poiseuille solution

The file *0/p*

```
42          bottom
43          {
44              type            zeroGradient;
45          }
46
47          front
48          {
49              type                empty;
50          }
51
52          back
53          {
54              type                empty;
55          }
56      }
```

- For the **bottom** patch (lines 42-45), we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).
- For the **front** and **back** patches (lines 47-55), we use an empty boundary condition. This boundary condition is used for 2D simulations. These two patches are normal to the direction where we assigned 1 cell (**Z** direction).
- At this point, if you take some time and compare the files *0/U* and *0/p* with the file *constant/polyMesh/boundary*, you will see that the name and type of each **primitive type** patch (the patch defined in *0*), is consistent with the **base type** patch (the patch defined in the file *constant/polyMesh/boundary*).

A simple validation case – Hagen-Poiseuille solution

The file *0/U*

```
19    internalField   uniform (0 0 0);
20
21    boundaryField
22    {
23        inlet
24        {
25            type            fixedValue;
26            value           uniform (1 0 0);
27        }
28
29        outlet
30        {
31          type          zeroGradient;
32        }
33
34        top
35        {
36            type            fixedValue;
37            value           uniform (0 0 0);
38        }
```

- We are using uniform initial conditions and the numerical value is **(0 0 0)** (keyword **internalField** in line 19).
- For the **inlet** patch (lines 23-27), we are using a **fixedValue** boundary condition with a numerical value equal to **(1 0 0)**
- For the **outlet** patch (lines 29-32), we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).
- The **top** patch is a no-slip wall (lines 34-38), therefore we impose a velocity of **(0 0 0)** at the wall.

A simple validation case – Hagen-Poiseuille solution

The file *0/U*

```
40          bottom
41          {
42              type            fixedValue;
43              value           uniform (0 0 0);
44          }
45
46          front
47          {
48              type            empty;
49          }
50
51          back
52          {
53              type            empty;
54          }
55      }
```

- The **bottom** patch is a no-slip wall (lines 40-44), therefore we impose a velocity of **(0 0 0)** at the wall.
- For the **front** and **back** patches (lines 46-54), we use an empty boundary condition. This boundary condition is used for 2D simulations. These two patches are normal to the direction where we assigned 1 cell (**Z** direction).
- At this point, if you take some time and compare the files *0/U* and *0/p* with the file *constant/polyMesh/boundary*, you will see that the name and type of each **primitive type** patch (the patch defined in *0*), is consistent with the **base type** patch (the patch defined in the file *constant/polyMesh/boundary*).

A simple validation case – Hagen-Poiseuille solution

The **`system`** directory

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

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

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
17    application     icoFoam;
18
19    startFrom       startTime;
20
21    startTime       0;
22
23    stopAt          endTime;
24
25    endTime         20;
26
27    deltaT          0.05;
28
29    writeControl    runTime;
30
31    writeInterval   1;
32
33    purgeWrite      0;
34
35    writeFormat     ascii;
36
37    writePrecision  8;
38
39    writeCompression off;
40
41    timeFormat      general;
42
43    timePrecision   6;
44
45    runTimeModifiable true;
```

- This case starts from time 0 (**startTime**).
- It will run up to 20 seconds (**endTime**).
- The time step of the simulation is 0.05 seconds (**deltaT**).
- It will write the solution every second (**writeInterval**) of simulation time (**runTime**).
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- The write precision is 8 digits (**writePrecision**). It will only save eight digits in the output files.
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
49      functions
50      {


            name_of_the_functionObject_dictionary
            {
                Dictionary with the functionObject entries
            }



204
207     };
```

- Let us take a look at the bottom of the *controlDict* dictionary file.
- Here we define **functionObjects**, which are functions that will do a computation while the simulation is running.
- We define the **functionObjects** in the sub-dictionary **functions** (line 49-207 in this case).
- Each **functionObject** we define, has its own name and its compulsory keywords and entries.
- In this case we are defining **functionObjects** to compute minimum and maximum values of the field variables, mass flow at the **inlet** and **outlet** patches, pressure average at the **inlet** patch, and maximum velocity at the **outlet** patch.
- In another variation of this case, we will use the output of the pressure average **functionObject** to set the numerical value of a pressure boundary condition at the **inlet** patch.
- The output of the maximum velocity **functionObject** will be used to plot the analytical solution (we can also use the `postProcess` utility to find this value).
- We are going to address **functionObjects** in details when we talk about post-processing.

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
49      functions
50      {
51
54      minmaxdomain
55      {
56          type fieldMinMax;
57
58          functionObjectLibs ("libfieldFunctionObjects.so");
59
60          enabled true; //true or false
61
62          mode component;
63
64          writeControl timeStep;
65          writeInterval 1;
66
67          log true;
68
69          fields ( p U );
70      }
71
```

- **fieldMinMax functionObject**
  - This **functionObject** is used to compute the minimum and maximum values of the field variables.
  - The output of this **functionObject** is saved in ascii format in the file *fieldMinMax.dat* located in the directory

    postProcessing/minmaxdomian/0

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
75      inMassFlow
76      {
77          type            surfaceRegion;
78          functionObjectLibs ("libfieldFunctionObjects.so");
79          enabled         true;
80
81          //writeControl     outputTime;
82          writeControl    timeStep;
83          writeInterval   1;
84
85          log             true;
86
87          writeFields     false;
88
89          regionType      patch;
90          Name            inlet;
91
92          operation       sum;
93
94          fields
95          (
96              phi
97          );
98      }
```

- **faceSource functionObject**
  - This **functionObject** is used to compute the mass flow in a boundary patch.
  - In this case, we are sampling the patch **inlet**.
  - The output of this **functionObject** is saved in ascii format in the file *faceSource.dat* located in the directory

    postProcessing/inMassFlow/0

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
102    outMassFlow
103    {
104        type            surfaceRegion;
105        functionObjectLibs ("libfieldFunctionObjects.so");
106        enabled         true;
107
108        //writeControl     outputTime;
109        writeControl    timeStep;
110        writeInterval   1;
111
112        log             true;
113
114        writeFields     false;
115
116        regionType      patch;
117        Name            outlet;
118
119        operation       sum;
120
121        fields
122        (
123            phi
124        );
125    }
```

- **faceSource functionObject**
  - This **functionObject** is used to compute the mass flow in a boundary patch.
  - In this case, we are sampling the patch **outlet**.
  - The output of this **functionObject** is saved in ascii format in the file *faceSource.dat* located in the directory

    postProcessing/outMassFlow/0

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
130    inPre
131    {
132        type            surfaceRegion;
133        functionObjectLibs ("libfieldFunctionObjects.so");
134        enabled         true;
135
136        //writeControl     outputTime;
137        writeControl    timeStep;
138        writeInterval   1;
139
140        log             true;
141
142        writeFields     false;
143
144        regionType      patch;
145        name            inlet;
146
147        operation       weightedAverage;
148
149        fields
150        (
151            phi
152            U
153            p
154        );
155    }
```

- **faceSource functionObject**
  - This **functionObject** is used to compute the weighted average in a boundary patch.
  - In this case, we are sampling the patch **inlet**.
  - The output of this **functionObject** is saved in ascii format in the file *faceSource.dat* located in the directory

    postProcessing/inPre/0

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
179     outMax
180     {
181         type            surfaceRegion;
182         functionObjectLibs ("libfieldFunctionObjects.so");
183         enabled         true;
184
185         //writeControl     outputTime;
186         writeControl    timeStep;
187         writeInterval   1;
188
189         log             true;
190
191         writeFields     false;
192
193         regionType      patch;
194         name            outlet;
195
196         operation       max;
197
198         fields
199         (
200             U
201             p
202         );
203     }
204
207     };
```

- **faceSource functionObject**
  - This **functionObject** is used to compute the maximum value in a boundary patch.
  - In this case, we are sampling the patch **outlet**.
  - The output of this **functionObject** is saved in ascii format in the file *faceSource.dat* located in the directory

    postProcessing/outMax/0

  - Finally, remember that you can use the banana method to get a list of the different options available for each keyword.
  - You can also read the source code or the doxygen documentation.

A simple validation case – Hagen-Poiseuille solution

The *fvSchemes* dictionary

```
17    ddtSchemes
18    {
19        default         Euler;
20    }
21
22    gradSchemes
23    {
24        default         Gauss linear;
27        grad(p)         Gauss linear;
28    }
29
30    divSchemes
31    {
32        default         none;
33        div(phi,U)      Gauss linear;
37    }
38
39    laplacianSchemes
40    {
41        default         Gauss linear orthogonal;
44    }
45
46    interpolationSchemes
47    {
48        default         linear;
49    }
50
51    snGradSchemes
52    {
53        default         orthogonal;
56    }
```

- In this case, for time discretization (**ddtSchemes**) we are using the **Euler** method.
- For gradient discretization (**gradSchemes**) we are using the **Gauss linear** method.
- For the discretization of the convective terms (**divSchemes**) we are using **linear** interpolation method for the term **div(phi,U)**.
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear** method with **orthogonal** corrections.
- This method is second order accurate but oscillatory.
- Remember, at the end of the day we want a solution that is second order accurate.

A simple validation case – Hagen-Poiseuille solution

The *fvSolution* dictionary

```
17    solvers
18    {
19        p
20        {
27            solver           GAMG;
28            tolerance        1e-6;
29            relTol           0.01;
30            smoother         GaussSeidel;
31            nPreSweeps       0;
32            nPostSweeps      2;
33            cacheAgglomeration on;
34            agglomerator     faceAreaPair;
35            nCellsInCoarsestLevel 100;
36            mergeLevels      1;
37        }
38
39        pFinal
40        {
41            $p;
42            relTol           0;
43        }
44
45        U
46        {
53            solver           PBiCG;
54            preconditioner   DILU;
55            tolerance        1e-08;
56            relTol           0;
57        }
58    }
59
60    PISO
61    {
62        nCorrectors     1;
63        nNonOrthogonalCorrectors 0;
66    }
```

- To solve the pressure (**p**) we are using the **GAMG** method with an absolute **tolerance** of 1e-6 and a relative tolerance **relTol** of 0.01 (the solver will stop iterating when it meets any of the conditions).
- The entry **pFinal** refers to the final correction of the **PISO** loop. In this case, we are using a tighter convergence criteria in the last iteration. Notice that we are using macro expansion (**$p)** to copy the entries from the sub-dictionary **p**.
- To solve **U** we are using the linear solver **PBiCG** and **DILU** preconditioner, with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0 (the solver will stop iterating when it meets any of the conditions).
- Solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.

A simple validation case – Hagen-Poiseuille solution

The *fvSolution* dictionary

```
17    solvers
18    {
19        p
20        {
27            solver           GAMG;
28            tolerance        1e-6;
29            relTol           0.01;
30            smoother         GaussSeidel;
31            nPreSweeps       0;
32            nPostSweeps      2;
33            cacheAgglomeration on;
34            agglomerator     faceAreaPair;
35            nCellsInCoarsestLevel 100;
36            mergeLevels      1;
37        }
38
39        pFinal
40        {
41            $p;
42            relTol           0;
43        }
44
45        U
46        {
53            solver           PBiCG;
54            preconditioner   DILU;
55            tolerance        1e-08;
56            relTol           0;
57        }
58    }
59
60    PISO
61    {
62        nCorrectors     1;
63        nNonOrthogonalCorrectors 0;
66    }
```

- The **PISO** sub-dictionary contains entries related to the pressure-velocity coupling (in this case the **PISO** method).
- Hereafter we are doing only one 1 **PISO** corrector and no non-orthogonal corrections.
- If we increase the number of **nCorrectors** and **nNonOrthogonalCorrectors** we gain more stability but at a higher computational cost.
- The choice of the number of corrections is driven by the quality of the mesh and the physics involve.
- You need to do at least one **PISO** loop (**nCorrectors**).

A simple validation case – Hagen-Poiseuille solution

The **system** directory

- In **system** directory you will find the following optional dictionary files:
  - *decomposeParDict*
  - *modifyMeshDict*
  - *sampleDict*

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

A simple validation case – Hagen-Poiseuille solution

The *sampleDict* dictionary

```
17    setFormat raw;
18
19    setFormat raw;
19
23    interpolationScheme cell;
24
26    fields
27    (
28        U
29        p
30    );
31
32    sets
33    (
34
36        s1
37        {
39            type            midPoint;
45            axis            x;
46            start           ( 0 0 0 );
47            end             ( 10 0 0);
48        }
49
67        s2
68        {
70            type            midPoint;
73            axis            y;
74            start           ( 9 -1 0 );
75            end             ( 9 1 0 );
76        }
77
78    );
```

- Let us visit again the *sampleDict* dictionary file.
- In this case we are sampling the field variables **U** and **p**.
- We are sampling in an horizontal line spanning from 0 to 10 (lines 38-50).
- We are sampling in a vertical line spanning from -1 to 1 (lines 52-61).
- If you want to sample in a different location feel free to add a new entry.

A simple validation case – Hagen-Poiseuille solution

Running the case

- You will find this tutorial in the directory **`$PTOFC/101OF/laminar_pipe/case0`**
- In the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> checkMesh`
4. `$> icoFoam > log | tail -f log`
5. `$> postProcess -func sampleDict -latestTime`
6. `$> gnuplot gnuplot/gnuplot_script`
7. `$> paraFoam`

A simple validation case – Hagen-Poiseuille solution

Running the case

- In step 1 we clean the case directory. It is highly advisable to always start form a clean case directory.
- In step 2 we generate the mesh.
- In step 3 we check the mesh quality.
- In step 4 we run the simulation. Notice that we are redirecting the output to the a log file and at the same time we are showing the information on-the-fly.
- In step 5 we do some sampling only of the last saved solution.
- In step 6 we use a gnuplot script to plot the sampled values. Feel free to take a look at the script and to reuse it.
- Finally, in step 7 we visualize the solution.

A simple validation case – Hagen-Poiseuille solution

Let us use different boundary conditions

- Instead of using a fixed value for the velocity, let us use a fixed value for the pressure.
- You can use any pressure value, but as in the previous case we computed the average pressure at the inlet it seems wise to use this value.
- At this point, get the average pressure value from the ascii file (we hope you remember the location of the file), change the boundary conditions, and run the simulation.
- To run simulation proceed as in the previous case.
- At the end, compare both cases. You should get very similar results.
- If you are feeling lazy, this case is already setup in the directory

```
$PTOFC/101OF/laminar_pipe/case1
```

A simple validation case – Hagen-Poiseuille solution

The file *0/p*

- We only need to change the boundary conditions of the **inlet** patch.

```
inlet
{
    type            fixedValue;
    value           uniform 1.53103;
}
```

- Do you think of an alternative to the **fixedValue** boundary condition?

A simple validation case – Hagen-Poiseuille solution

The file *0/U*

- We only need to change the boundary conditions of the **inlet** patch.

```
inlet
{
    type            pressureNormalInletOutletVelocity;
    phi             phi;
    rho             rho;
    value           uniform (0 0 0);
}
```

- If you want to know what is behind this esoteric boundary condition, refer to the doxygen documentation or the source code.
- FYI, you can also use **zeroGradient**.

A simple validation case – Hagen-Poiseuille solution

Running with a steady state solver

- At Re = 100 nothing is happening. No vortex shedding, no detached flow, no flow instabilities, no turbulence, and no shock waves (this is kind of a boring case). Therefore is safe to say that this is a steady flow.
- In this case, we can use a steady solver. Steady solvers are way much faster than unsteady solvers but they violate a lot of principles, this is a trick that CFDers use to speed-up things. If you are happy with this approximation use steady solvers with no remorse.
- In an ideal world, steady solvers should converge in one iteration. But due to the non-linearities in the governing equations we need to proceed in an iterative way, until we satisfy a convergence criteria.
- Let us run this case using `simpleFoam` (which is an incompressible steady solver).
- As we are using a new solver we need to do some changes in the dictionaries files.
- This case is already setup in the directory **`$PTOFC/101OF/laminar_pipe/case2`**
- At this point, let us explore the case directory.

A simple validation case – Hagen-Poiseuille solution

- The following dictionary files remains unchanged:
  - *system/blockMeshDict*
  - *constant/polyMesh/boundary*
  - *0/U*
  - *0/p*
- FYI, we are using the same setup as in case **case2**
- New dictionary files
  - *turbulenceProperties*
- The solver `simpleFoam` can be used for laminar and turbulent flows.
- The following dictionaries need to be modified:
  - *transportProperties*
  - *controlDict*
  - *fvSchemes*
  - *fvSolution*

A simple validation case – Hagen-Poiseuille solution

The *turbulenceProperties* dictionary file

- This dictionary file is located in the directory **constant**.
- In this dictionary file we select what model we would like to use (laminar or turbulent).
- As we are not interested in modeling turbulence, this dictionary should read as follows,

```
17    simulationType    laminar;
```

A simple validation case – Hagen-Poiseuille solution

The *transportProperties* dictionary file

- In this file we define the transport model and the kinematic viscosity (**nu**).

```
16    transportModel  Newtonian;
17
18    nu              nu [ 0 2 -1 0 0 0 0 ] 0.01;
```

- The file *transportProperties* used with the solver `icoFoam`, does not require the keyword **transportModel**. The solver `icoFoam` only uses the Newtonian model.

A simple validation case – Hagen-Poiseuille solution

The *controlDict* dictionary

```
17    application     simpleFoam;
18
19    startFrom       startTime;
20
21    startTime       0;
22
23    stopAt          endTime;
24
25    endTime         1000;
26
27    deltaT          1;
28
29    writeControl    runTime;
30
31    writeInterval   10;
32
33    purgeWrite      0;
34
35    writeFormat     ascii;
36
37    writePrecision  8;
38
39    writeCompression off;
40
41    timeFormat      general;
42
43    timePrecision   6;
44
45    runTimeModifiable true;
```

- As this is a steady solver it does not make any sense setting the time step.
- The time step is only used to advanced the solution (iterate) and to save the solution.
- The keyword **endTime** refers to the maximum number of iterations.

A simple validation case – Hagen-Poiseuille solution

The *fvSchemes* dictionary

```
17    ddtSchemes
18    {
19        default         steadyState;
20    }
21
22    gradSchemes
23    {
24        default         Gauss linear;
27        grad(p)         Gauss linear;
28    }
29
30    divSchemes
31    {
32        default         none;
33        div(phi,U)      bounded Gauss linear;
38        div((nuEff*dev2(T(grad(U))))) Gauss linear;
39    }
40
41    laplacianSchemes
42    {
43        default         Gauss linear orthogonal;
46    }
47
48    interpolationSchemes
49    {
50        default         linear;
51    }
52
53    snGradSchemes
54    {
55        default         orthogonal;
58    }
```

- These are the changes introduced in the dictionary:
  - Time discretization (**ddtSchemes**), is **steadyState**.
  - For the discretization of the convective terms (**divSchemes**) we are using a **bounded linear** interpolation method for the term **div(phi,U)**
  - We added the term **div((nuEff*dev2(T(grad(U)))))**. This term is related to the turbulence formulation. We must define it even if we are using the laminar model.

A simple validation case – Hagen-Poiseuille solution

The *fvSolution* dictionary

```
17    solvers
18    {
19        p
20        {
27            solver           GAMG;
28            tolerance        1e-6;
29            relTol           0.01;
30            smoother         GaussSeidel;
31            nPreSweeps       0;
32            nPostSweeps      2;
33            cacheAgglomeration on;
34            agglomerator     faceAreaPair;
35            nCellsInCoarsestLevel 100;
36            mergeLevels      1;
37        }
38
39        U
40        {
47            solver          PBiCG;
48            preconditioner  DILU;
49            tolerance       1e-08;
50            relTol          0;
51        }
52    }
```

- To solve the pressure (**p**) we are using the **GAMG** method with an absolute **tolerance** of 1e-6 and a relative tolerance **relTol** of 0.01 (the solver will stop iterating when it meets any of the conditions).
- To solve **U** we are using the solver **PBiCG** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0 (the solver will stop iterating when it meets any of the conditions).
- FYI, solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.

A simple validation case – Hagen-Poiseuille solution

The *fvSolution* dictionary

```
54    SIMPLE
55    {
56        nNonOrthogonalCorrectors 1;
61
62        residualControl
63        {
64            p     1e-4;
65            U     1e-4;
69        }
70    }
71
73    relaxationFactors
74    {
75        fields
76        {
77            P     0.3;
78        }
79        equations
80        {
81            U     0.7;
84        }
85    }
```

- The **SIMPLE** sub-dictionary contains entries related to the pressure-velocity coupling (in this case the **SIMPLE** method).
- Hereafter we are doing one non orthogonal correction.
- In the sub-dictionary **residualControl** we set the convergence criteria for each field variable. The solver will stop if it reach this criterion or the maximum number of iterations (**endTime**).
- In the sub-dictionary **relaxationFactors** we set the under-relaxation coefficients. The under-relaxation factors (URF) controls how fast the solution change between iterations. Choosing the optimal URF requires a lot experience. It is wise to stick to the commonly used values.

p	**0.3;**
U	**0.7;**
k	**0.7;**
omega	**0.7;**
epsilon	**0.7;**

A simple validation case – Hagen-Poiseuille solution

Running the case

- You will find this tutorial in the directory **`$PTOFC/101OF/laminar_pipe/case2`**
- In the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> checkMesh`
4. `$> simpleFoam > log | tail -f log`
5. `$> postProcess -func sampleDict -latestTime`
6. `$> gnuplot gnuplot/gnuplot_script`
7. `$> paraFoam`

A simple validation case – Hagen-Poiseuille solution

Running the case

- In step 1 we clean the case directory. It is highly advisable to always start form a clean case directory.
- In step 2 we generate the mesh.
- In step 3 we check the mesh quality.
- In step 4 we run the simulation. Notice that we are redirecting the output to the a log file and at the same time we are showing the information on-the-fly.
- In step 5 we do some sampling only of the last saved solution.
- In step 6 we use a gnuplot script to plot the sampled values. In this case, you will need to adapt this script to get the sampled data from the right directory.
- Finally, in step 7 we visualize the solution.
- Compare this solution with the solution of the case **`case0`**

A simple validation case – Hagen-Poiseuille solution

What about mesh quality?

- So far we have worked with perfect meshes, that is, meshes with non-orthogonality and skewness close to zero.
- But this is the exception rather than the rule.
- Getting a solution in this kind of meshes is quite easy.

```
Checking geometry...
    Overall domain bounding box (0 -0.5 0) (10 0.5 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 (7.8140697e-20 1.4221607e-17 5.4393739e-16) OK.
    Max cell openness = 8.6736174e-17 OK.
    Max aspect ratio = 1 OK.
    Minimum face area = 0.01. Maximum face area = 0.01.  Face area magnitudes OK.
    Min volume = 0.001. Max volume = 0.001.  Total volume = 1.  Cell volumes OK.
    Mesh non-orthogonality Max: 0 average: 0
    Non-orthogonality check OK.
    Face pyramids OK.
    Max skewness = 1.0658141e-13 OK.
    Coupled point location match (average 0) OK.

Mesh OK.
```

Non-orthogonality

Skewness

A simple validation case – Hagen-Poiseuille solution

What about mesh quality?

- Industrial meshes are far from being perfect.
- Mesh quality highly affect solution accuracy, stability, and convergence rate.
- To take into account mesh quality issues, we need to adjust the numerical method. We will deal with this during the FVM lecture.

```
Checking geometry...
    Overall domain bounding box (0 -0.5 0) (10 0.5 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 (7.8140697e-20 1.4221607e-17 5.539394e-16) OK.
    Max cell openness = 9.3768837e-17 OK.
    Max aspect ratio = 1.98 OK.
    Minimum face area = 0.00085. Maximum face area = 0.02154739.  Face area magnitudes OK.
    Min volume = 8.5e-05. Max volume = 0.001915.  Total volume = 1.  Cell volumes OK.
    Mesh non-orthogonality Max: 86.473612 average: 2.5674993
   *Number of severely non-orthogonal (> 70 degrees) faces: 1.
    Non-orthogonality check OK.
  <<Writing 1 non-orthogonal faces to set nonOrthoFaces
 ***Error in face pyramids: 2 faces are incorrectly oriented.
  <<Writing 2 faces with incorrect orientation to set wrongOrientedFaces
 ***Max skewness = 11.305066, 1 highly skew faces detected which may impair the quality of the results
  <<Writing 1 skew faces to set skewFaces
    Coupled point location match (average 0) OK.

Failed 2 mesh checks.
```

Too high non-orthogonality
Acceptable values are less than 80

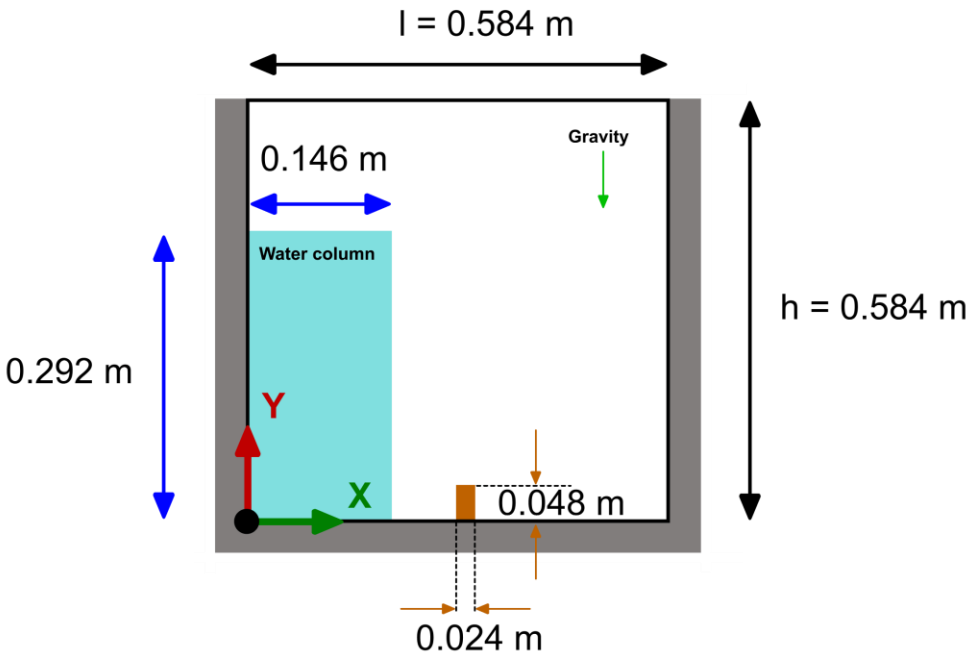
Failed sets can be visualized in paraFoam

Too high skewness
Acceptable values are less than 6

This does not mean that you can not run the simulation

Dam break free surface flow

Dam break free surface flow

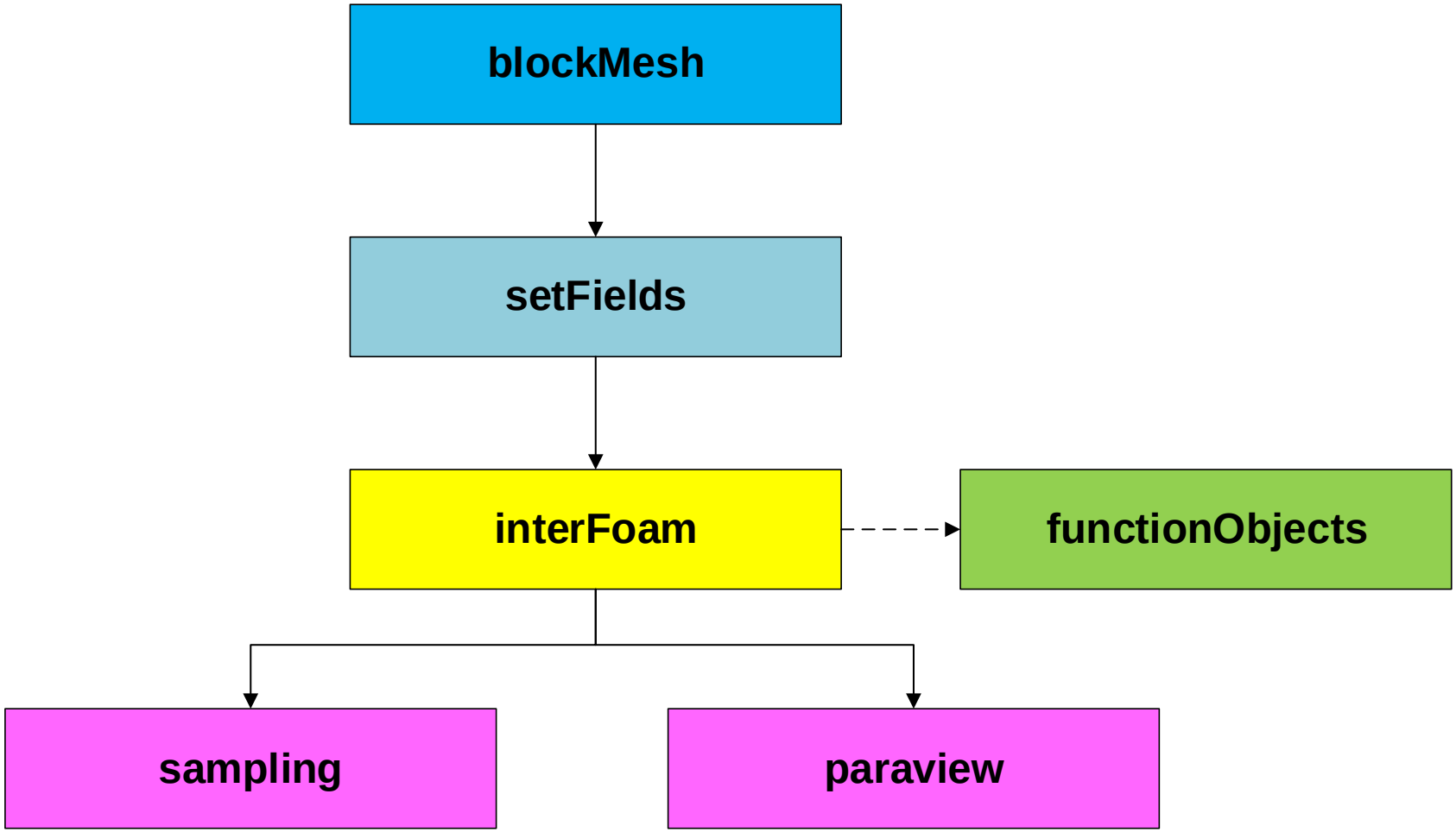


Physical and numerical side of the problem:

- In this case we are going to use the VOF method. This method solves the incompressible Navier-Stokes equations plus an additional equation to track the volume fraction (free surface location).
- We are going to work in a 2D domain but the problem can be extended to 3D easily.
- As this is a multiphase case, we need to define the physical properties for each phase involved (viscosity, density and surface tension).
- Additionally, we need to define the gravity vector and initialize the two flows.
- This is an unsteady case.

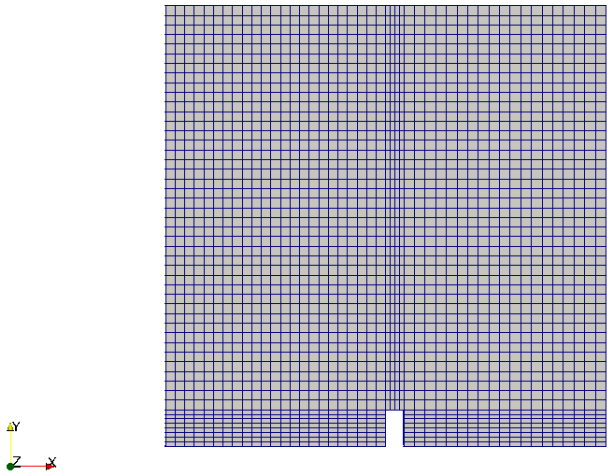
Dam break free surface flow

Workflow of the case



Dam break free surface flow

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

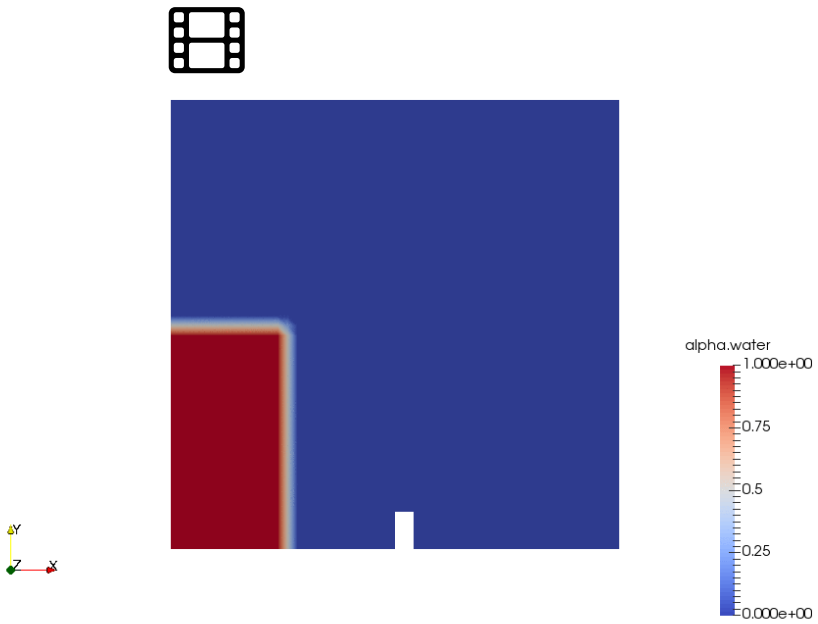


Mesh

Initial conditions

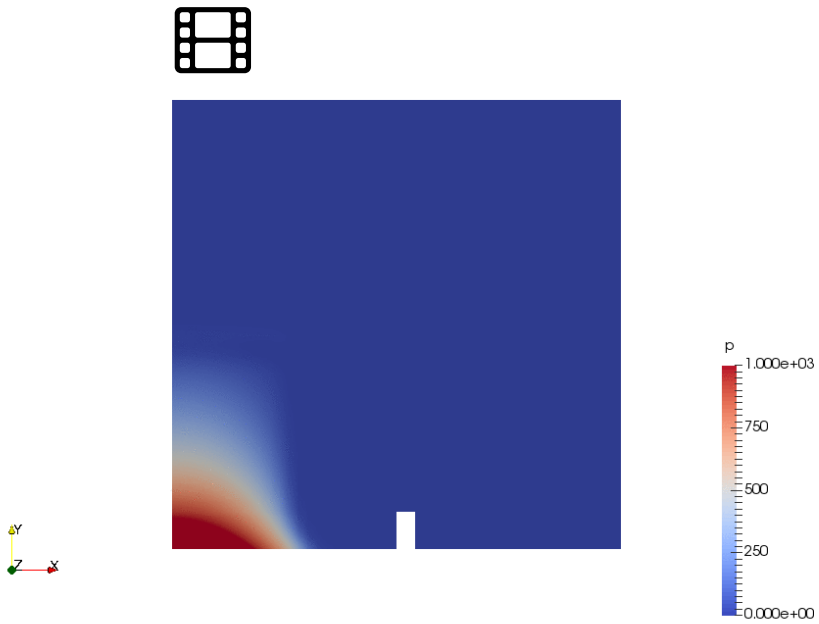
Dam break free surface flow

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



VOF Fraction

www.wolfdynamics.com/wiki/dambreak/ani1.gif



Hydrostatic pressure

www.wolfdynamics.com/wiki/dambreak/ani2.gif

Dam break free surface flow

What are we going to do?

- We will use this case to introduce the multiphase solver `interFoam`.
- `interFoam` is a solver for 2 incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach
- We will define the physical properties of two phases and we are going to initialize these phases.
- We will define the gravity vector in the dictionary `g`.
- After finding the solution, we will visualize the results. This is an unsteady case so now we are going to see things moving.
- We are going to briefly address how to post-process multiphase flows.

Dam break free surface flow

Let’s explore the case directory

Dam break free surface flow

The *blockMeshDict* dictionary file

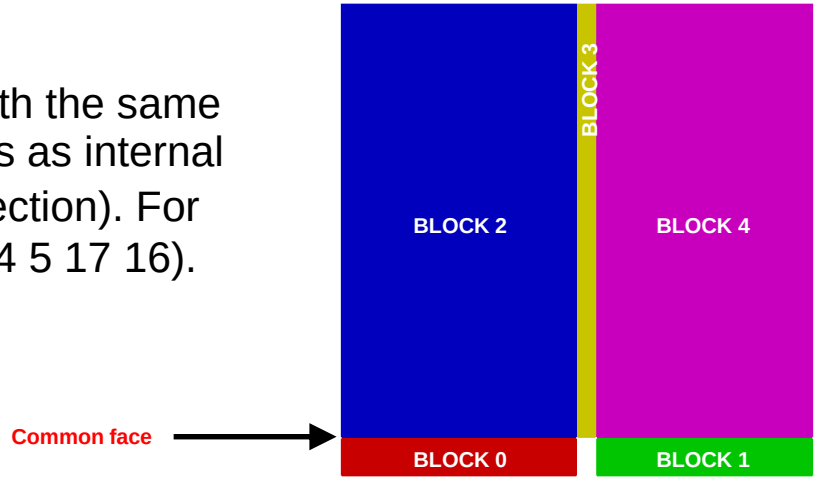
```
17      convertToMeters 0.146;
18
19      vertices
20      (
21          (0 0 0)                     //Vertex0
22          (2 0 0)
23          (2.16438 0 0)
24          (4 0 0)
25          (0 0.32876 0)
26          (2 0.32876 0)
27          (2.16438 0.32876 0)
28          (4 0.32876 0)
29          (0 4 0)
30          (2 4 0)
31          (2.16438 4 0)
32          (4 4 0)
33          (0 0 0.1)
34          (2 0 0.1)
35          (2.16438 0 0.1)
36          (4 0 0.1)
37          (0 0.32876 0.1)
38          (2 0.32876 0.1)
39          (2.16438 0.32876 0.1)
40          (4 0.32876 0.1)
41          (0 4 0.1)
42          (2 4 0.1)
43          (2.16438 4 0.1)
44          (4 4 0.1)                   //Vertex 23
45      );
```

- This dictionary is located in the **`system`** directory.
- We are using scaling (line 17).
- In lines 19-45, we define the vertices coordinates.

Dam break free surface flow

The *blockMeshDict* dictionary file

- In this case we are defining five blocks.
- In the common faces, the blocks share vertices with the same index number, `blockMesh` recognizes these faces as internal (we do not need to define them in the boundary section). For example, block 0 and block 2 share the vertices (4 5 17 16).
- We are using uniform grading in all blocks.
- All edges are straight lines by default.




```
47      blocks
48      (
49          hex (0 1 5 4 12 13 17 16) (23 8 1) simpleGrading (1 1 1)        //Block 0
50          hex (2 3 7 6 14 15 19 18) (19 8 1) simpleGrading (1 1 1)        //Block 1
51          hex (4 5 9 8 16 17 21 20) (23 42 1) simpleGrading (1 1 1)       //Block 2
52          hex (5 6 10 9 17 18 22 21) (4 42 1) simpleGrading (1 1 1)       //Block 3
53          hex (6 7 11 10 18 19 23 22) (19 42 1) simpleGrading (1 1 1)     //Block 4
54      );
55
56      edges
57      (
58      );
```

Dam break free surface flow

The *blockMeshDict* dictionary file

```
60      boundary
61      (
62          leftWall
63          {
64              type wall;
65              faces
66              (
67                  (0 12 16 4)
68                  (4 16 20 8)
69              );
70          }
71          rightWall
72          {
73              type wall;
74              faces
75              (
76                  (7 19 15 3)
77                  (11 23 19 7)
78              );
79          }
80          lowerWall
81          {
82              type wall;
83              faces
84              (
85                  (0 1 13 12)
86                  (1 5 17 13)
87                  (5 6 18 17)
88                  (2 14 18 6)
89                  (2 3 15 14)
90              );
91          }
```

- The boundary patches **leftWall, rightWall** and **lowerWall** are of **base type** wall.
- Notice that each boundary patch groups many faces.
- Remember, we assign the **primitive type** boundary conditions (numerical values), in the field files found in the directory *0*

Dam break free surface flow

The *blockMeshDict* dictionary file

```
92          atmosphere
93          {
94              type patch;
95              faces
96              (
97                  (8 20 21 9)
98                  (9 21 22 10)
99                  (10 22 23 11)
100             );
101         }
102     );
103
104     mergePatchPairs
105     (
106     );
```

- The boundary patch atmosphere is of **base type** patch.
- Notice that we do not define the front and back patches, these patches are automatically group in the boundary patch **defaultFaces** of **base type** empty.
- Remember, we assign the **primitive type** boundary conditions (numerical values), in the field files found in the directory *0*
- We do not need to merge faces.

Dam break free surface flow

The *boundary* dictionary file

- This dictionary is located in the **constant/polyMesh** directory.
- This file is automatically created when generating or converting the mesh.
- In this case, we do not need to modify this file. All the **base type** boundary conditions and **name** of the patches were assigned in the *blockMeshDict* file.
- The **defaultFaces** boundary patch contains all patches that we did not define in the boundary section.
- If you change the **name** or the **base type** of a boundary patch, you will need to modify the field files in the directory **0**.

```
47      defaultFaces
48      {
49          type            empty;
50          inGroups        1(empty);
51          nFaces          4563;
52          startFace       4640;
53      }
```

Dam break free surface flow

The **constant** directory

- In this directory, we will find the following compulsory dictionary files:
  - *g*
  - *transportProperties*
  - *turbulenceProperties*

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

Dam break free surface flow

The g dictionary file

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

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

Dam break free surface flow

The *transportProperties* dictionary file

Primary phase

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

- This dictionary file is located in the directory **`constant`**.
- We first define the name of the phases (line 18). In this case we are defining the names **water** and **air**. The first entry in this list is the primary phase (**water**).
- The name of the phases is given by the user.
- In this file we set the kinematic viscosity (**nu**), density (**rho**) and transport model (**transportModel**) of the phases.
- We also define the surface tension (**sigma**).

Dam break free surface flow

The *turbulenceProperties* dictionary file

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

```
18    simulationType    laminar;
```

Dam break free surface flow

The **0** directory

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

  - *alpha.water* (volume fraction of water phase)
  - *p_rgh* (pressure field minus hydrostatic component)
  - *U* (velocity field)

Dam break free surface flow

The file *0/alpha.water*

```
17  dimensions      [0 0 0 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      leftWall
24      {
25          type            zeroGradient;
26      }
27
28      rightWall
29      {
30          type            zeroGradient;
31      }
32
33      lowerWall
34      {
35          type            zeroGradient;
36      }
37
38      atmosphere
39      {
40          type            inletOutlet;
41          inletValue      uniform 0;
42          value           uniform 0;
43      }
44
45      defaultFaces
46      {
47          type            empty;
48      }
49  }
```

- This file contains the boundary and initial conditions for the non-dimensional scalar field **alpha.water**
- This file is named *alpha.water*, because the primary phase is water (we defined the primary phase in the *transportProperties* dictionary).
- Initially, this field is initialize as 0 in the whole domain (line 19). This means that there is no water in the domain at time 0. Later, we will initialize the water column and this file will be overwritten with a non-uniform field for the **internalField**.
- For the **leftWall**, **rightWall,** and **lowerWall** patches we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).
- For the **atmosphere** patch we are using an **inletOutlet** boundary condition**.** This boundary condition avoids backflow into the domain. If the flow is going out it will use **zeroGradient** and if the flow is coming back it will assign the value set in the keyword **inletValue** (line 41).
- The **defaultFaces** patch is of **primitive type empty**.

Dam break free surface flow

The file *0/p_rgh*

```
17  dimensions      [1 -1 -2 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      leftWall
24      {
25          type            fixedFluxPressure;
26          value           uniform 0;
27      }
29      rightWall
30      {
31          type            fixedFluxPressure;
32          value           uniform 0;
33      }
35      lowerWall
36      {
37          type            fixedFluxPressure;
38          value           uniform 0;
39      }
41      atmosphere
42      {
43          type            totalPressure;
44          p0              uniform 0;
45          U               U;
46          phi             phi;
47          rho             rho;
48          psi             none;
49          gamma           1;
50          value           uniform 0;
51      }
53      defaultFaces
54      {
55          type            empty;
56      }
57  }
```

- This file contains the boundary and initial conditions for the dimensional field **p_rgh**. The dimensions of this field are given in Pascal (line 17)
- This scalar field contains the value of the static pressure field minus the hydrostatic component.
- This field is initialize as 0 in the whole domain (line 19).
- For the **leftWall**, **rightWall,** and **lowerWall** patches we are using a **fixedFluxPressure** boundary condition (refer to the source code or doxygen documentation to know more about this boundary condition).
- For the **atmosphere** patch we are using the **totalPressure** boundary condition (refer to the source code or doxygen documentation to know more about this boundary condition).
- The **defaultFaces** patch is of **primitive type empty**.

Dam break free surface flow

The file *0/U*

```
17  dimensions      [0 1 -1 0 0 0 0];
18
19  internalField   uniform (0 0 0);
20
21  boundaryField
22  {
23      leftWall
24      {
25          type            fixedValue;
26          value           uniform (0 0 0);
27      }
28      rightWall
29      {
30          type            fixedValue;
31          value           uniform (0 0 0);
32      }
33      lowerWall
34      {
35          type            fixedValue;
36          value           uniform (0 0 0);
37      }
38      atmosphere
39      {
40          type            pressureInletOutletVelocity;
41          value           uniform (0 0 0);
42      }
43      defaultFaces
44      {
45          type            empty;
46      }
47  }
```

- This file contains the boundary and initial conditions for the dimensional vector field **U**.
- We are using uniform initial conditions and the numerical value is **(0 0 0)** (keyword **internalField** in line 19).
- The **leftWall**, **rightWall,** and **lowerWall** patches are no-slip walls, therefore we impose a **fixedValue** boundary condition with a value of **(0 0 0)** at the wall.
- For the **outlet** patch we are using a **zeroGradient** boundary condition (we are just extrapolating the internal values to the boundary face).
- For the **atmosphere** patch we are using the **pressureInlterOutletVelocity** boundary condition (refer to the source code or doxygen documentation to know more about this boundary condition).
- The **defaultFaces** patch is of **primitive type empty**.

Dam break free surface flow

The **`system`** directory

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

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

Dam break free surface flow

The *controlDict* dictionary

```
18      application     interFoam;
19
20      startFrom       startTime;
21
22      startTime       0;
23
24      stopAt          endTime;
25
26      endTime         1;
27
28      deltaT          0.001;
29
30      writeControl    adjustableRunTime;
31
32      writeInterval   0.05;
33
34      purgeWrite      0;
35
36      writeFormat     ascii;
37
38      writePrecision  8;
39
40      writeCompression uncompressed;
41
42      timeFormat      general;
43
44      timePrecision   8;
45
46      runTimeModifiable yes;
47
48      adjustTimeStep  yes;
49
50      maxCo           1;
51      maxAlphaCo      1;
52      maxDeltaT       1;
```

- This case starts from time 0 (**startTime**).
- It will run up to 1 second (**endTime**).
- The initial time step of the simulation is 0.001 seconds (**deltaT**).
- It will write the solution every 0.05 seconds (**writeInterval**) of simulation time (**runTime**). It will automatically adjust the time step (**adjustableRunTime**), in order to save the solution at the precise write interval.
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- The write precision is 8 digits (**writePrecision**). It will only save eight digits in the output files.
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.
- In line 48 we turn on the option **adjustTimeStep**. This option will automatically adjust the time step to achieve the maximum desired courant number (lines 50-51). We also set a maximum time step in line 52.
- Remember, the first time step of the simulation is done using the value set in line 28 and then it is automatically scaled to achieve the desired maximum values (lines 50-51).

Dam break free surface flow

The *controlDict* dictionary

```
58      functions
59      {

62
63      minmaxdomain
64      {
65          type fieldMinMax;
66
67          functionObjectLibs ("libfieldFunctionObjects.so");
68
69          enabled true; //true or false
70
71          mode component;
72
73          outputControl timeStep;
74          outputInterval 1;
75
76          log true;
77
78          fields (p U alpha.water);
79      }

109     };
```

- Let's take a look at the **functionObjects** definitions.
- In lines 63-79 we define the **fieldMinMax functionObject** which computes the minimum and maximum values of the field variables (**p U alpha.water**).

Dam break free surface flow

The *controlDict* dictionary

```
58      functions
59      {

84      water_in_domain
85      {
86          type            cellSource;
87          functionObjectLibs ("libfieldFunctionObjects.so");
88          enabled         true;
89
90          //outputControl     outputTime;
91          outputControl   timeStep;
92          outputInterval  1;
93
94          log             true;
95
96          valueOutput     false;
97
98          source          all;
99
100         operation       volIntegrate;
101         fields
102         (
103             alpha.water
104         );
105     }

109     };
```

- Let's take a look at the **functionObjects** definitions.
- In lines 84-105 we define the **cellSource functionObject** which computes the volume integral (**volIntegrate**) of the field variable **alpha.water** in all the domain.
- Basically, we are monitoring the quantity of water in the domain.

Dam break free surface flow

The *fvSchemes* dictionary

```
18    ddtSchemes
19    {
20        default         Euler;
21    }
22
23    gradSchemes
24    {
25        default         Gauss linear;
26    }
27
28    divSchemes
29    {
30        div(rhoPhi,U)  Gauss linearUpwind grad(U);
31        div(phi,alpha)  Gauss vanLeer;
32        div(phirb,alpha) Gauss linear;
33        div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
34    }
35
36    laplacianSchemes
37    {
38        default         Gauss linear corrected;
39    }
40
41    interpolationSchemes
42    {
43        default         linear;
44    }
45
46    snGradSchemes
47    {
48        default         corrected;
49    }
```

- In this case, for time discretization (**ddtSchemes**) we are using the **Euler** method.
- For gradient discretization (**gradSchemes**) we are using the **Gauss linear** method.
- For the discretization of the convective terms (**divSchemes**) we are using **linearUpwind** interpolation method for the term **div(rhoPhi,U)**.
- For the term **div(phi,alpha)** we are using **vanLeer** interpolation. For the term **div(phirb,alpha)** we are using **linear** interpolation. These terms are related to the volume fraction equation.
- For the term **div(((rho*nuEff)*dev2(T(grad(U)))))** we are using **linear** interpolation (this term is related to the turbulence modeling).
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear corrected** method
- This method is second order accurate but oscillatory.
- Remember, at the end of the day we want a solution that is second order accurate.

Dam break free surface flow

The *fvSolution* dictionary

```
18    solvers
19    {
20        "alpha.water.*"
21        {
22            nAlphaCorr      2;
23            nAlphaSubCycles 1;
24            cAlpha          1;
25
26            MULESCorr       yes;
27            nLimiterIter    3;
28
29            solver          smoothSolver;
30            smoother        symGaussSeidel;
31            tolerance       1e-8;
32            relTol          0;
33        }
34
35        pcorr
36        {
37            solver          PCG;
38            preconditioner  DIC;
39            tolerance       1e-8;
40            relTol          0;
41        }
42
43        p_rgh
44        {
45            solver          PCG;
46            preconditioner  DIC;
47            tolerance       1e-06;
48            relTol          0.01;
49        }
```

- To solve the volume fraction or **alpha.water** (lines 20-33) we are using the **smoothSolver** method.
- In line 26 we turn on the semi-implicit method MULES. The keyword **nLimiterIter** controls the number of MULES iterations over the limiter.
- To have more stability it is possible to increase the number of loops and corrections used to solve **alpha.water** (lines 22-23).
- The keyword **cAlpha** (line 24) controls the sharpness of the interface (1 is usually fine for most cases).
- In lines 35-41 we setup the solver for **pcorr** (pressure correction).
- In lines 43-49 we setup the solver for **p_rgh**.
- FYI, in this case **pcorr** is solved only one time at the beginning of the computation.

Dam break free surface flow

The *fvSolution* dictionary

```
51        p_rghFinal
52        {
53            $p_rgh;
54            relTol          0;
55        }
56
57        "(U|Ufinal)"
58        {
59            solver          smoothSolver;
60            smoother        symGaussSeidel;
61            tolerance       1e-06;
62            relTol          0;
70        }
71    }
72
73    PIMPLE
74    {
75        momentumPredictor   yes;
76        nOuterCorrectors    1;
77        nCorrectors         3;
78        nNonOrthogonalCorrectors 0;
79    }
80
81    relaxationFactors
82    {
83        fields
84        {
85            ".*" 1;
86        }
87        equations
88        {
89            ".*" 1;
90        }
91    }
```

- In lines 51-55 we setup the solver for **p_rghFinal**. This correspond to the last iteration in the loop (we can use a tighter convergence criteria to get more accuracy without increasing the computational cost)
- In lines 57-70 we setup the solver for **U**.
- In lines 73-79 we setup the entries related to the pressure-velocity coupling method used (**PIMPLE** in this case). Setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- To gain more stability we can increase the number of correctors (lines 76-78), however this will increase the computational cost.
- In lines 81-91 we setup the under relaxation factors related to te PIMPLE method. By setting the coefficients to one we are not under-relaxing.

Dam break free surface flow

The **system** directory

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

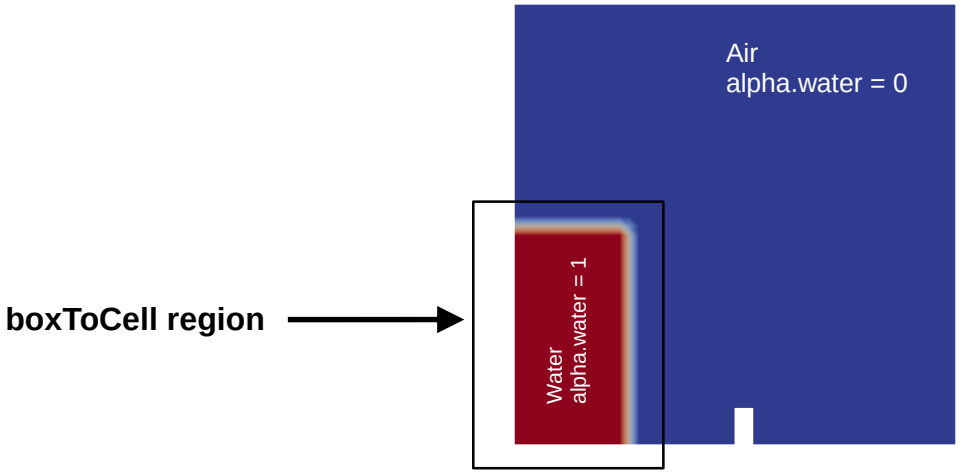
- *decomposeParDict* is read by the utility `decomposePar`. This dictionary file contains information related to the mesh partitioning. This is used when running in parallel.
- *setFieldsDict* is read by the utility `setFields`. This utility set values on selected cells/faces.
- *probesDict* is read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

Dam break free surface flow

The *setFieldsDict* dictionary

```
18  defaultFieldValues
19  (
20      volScalarFieldValue alpha.water 0
21  );
22
23  regions
24  (
25      boxToCell
26      {
27          box (0 0 -1) (0.1461 0.292 1);
28          fieldValues
29          (
30              volScalarFieldValue alpha.water 1
31          );
32      }
33  );
```

- This dictionary file is located in the directory **system**.
- In lines 18-21 we set the default value to be 0 in the whole domain (no water).
- In lines 25-32, we initialize a rectangular region (**box**) containing water (**alpha.water 1**).
- In this case, `setFields` will look for the dictionary file *alpha.water* and it will overwrite the original values according to the regions defined in *setFieldsDict*.
- If you are interested in initializing the vector field **U**, you can proceed as follows **volVectorFieldValue U (0 0 0)**



Dam break free surface flow

The *probesDict* dictionary

```
17    fields
18    (
19        alpha.water
20        U
21        p_rgh
22        p
23    );
24
25    probeLocations
26    (
27        (0.292 0 0)
28        (0.292 0.0240 0)
29        (0.292 0.0480 0)
30        (0.316 0.0480 0)
31        (0.316 0.0240 0)
32    );
```

Fields to sample.

Points location.

The sampled information is always saved in the directory **postProcessing/probes**

As we are sampling starting from time 0, the sampled data will be located in the directory:

postProcessing/probes/0

The files *alpha.water*, *p_rgh*, *p*, and *U* located in the directory **postProcessing/probes/0** contain the sampled data. Feel free to open them using your favorite text editor.

Dam break free surface flow

Running the case

- You will find this tutorial in the directory **`$PTOFC/101OF/damBreak`**
- In the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> checkMesh`
4. `$> cp 0/alpha.water.org 0/alpha.water`
5. `$> setFields`
6. `$> paraFoam`
7. `$> interFoam > log.interFoam | tail -f log.interFoam`
8. `$> probeLocations`
9. `$> paraFoam`

Dam break free surface flow

Running the case

- In step 2 we generate the mesh.
- In step 3 we check the mesh quality.
- In step 4 we copy the information of the backup file `alpha.water.org` to the file `alpha.water`. We do this because in the next step the utility `setFields` will overwrite the file `alpha.water`, so it is a good idea to keep a backup.
- In step 5 we initialize the solution using the utility `setFields`. This utility reads the dictionary *`setFieldsDict`* located in the **`system`** directory.
- In step 6 we use `paraFoam` to visualize the initialization. Remember to select the field `alpha.water` in `paraFoam`.
- In step 7 we run the simulation.
- In step 8 we use the utility `probeLocations` to sample field values at given locations. This utility reads the dictionary *`probesDict`*.
- Finally, in step 9 we visualize the solution.

Dam break free surface flow

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

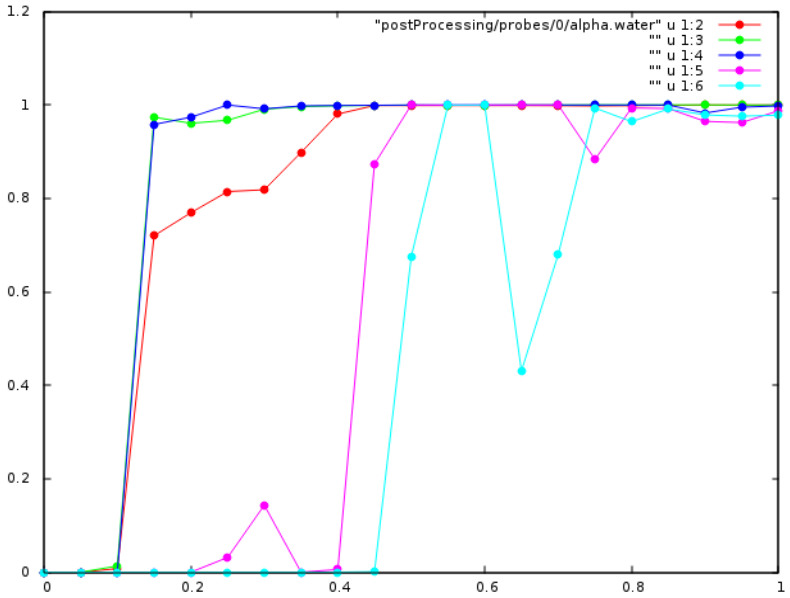
1.
```
$> gnuplot
```

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

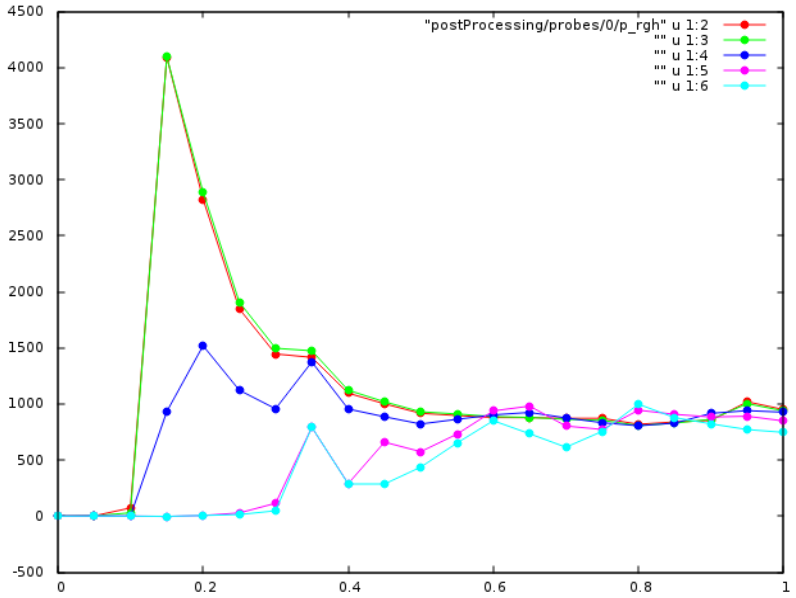
1.
```
gnuplot> plot [][0:1.2] “postProcessing/probes/0/alpha.water” u 1:2 pt 7 w lp,
        “ ” u 1:3 pt 7 w lp, “ ” u 1:4 pt 7 w lp,
        “ ” u 1:5 pt 7 w lp, “ ” u 1:6 pt 7 w lp
```
2.
```
gnuplot> plot [][] “postProcessing/probes/0/p_rgh” u 1:2 pt 7 w lp,
        “ ” u 1:3 pt 7 w lp, “ ” u 1:4 pt 7 w lp,
        “ ” u 1:5 pt 7 w lp, “ ” u 1:6 pt 7 w lp
```
3.
```
gnuplot> plot [][] “postProcessing/probes/0.05/p” u 1:2 pt 7 w lp,
        “ ” u 1:3 pt 7 w lp, “ ” u 1:4 pt 7 w lp,
        “ ” u 1:5 pt 7 w lp, “ ” u 1:6 pt 7 w lp
```
4.
```
gnuplot> exit
```
To exit gnuplot

Dam break free surface flow

- The output of steps 2 and 3 is the following:



alpha.water vs. time



p_rgh vs. time

Dam break free surface flow

The output screen

- This is the output screen of the `interFoam` solver.
- The interface courant number is more restrictive than the flow courant number.
- When solving multiphase flows, is always desirable to keep the interface courant number less than 1.

```
Courant Number mean: 0.134923 max: 0.684053
Interface Courant Number mean: 0.0189168 max: 0.427165
deltaT = 0.00137741
Time = 1

PIMPLE: iteration 1
smoothSolver:  Solving for alpha.water, Initial residual = 0.00337527, Final residual = 5.40522e-11, No Iterations 3
Phase-1 volume fraction = 0.127626  Min(alpha.water) = -2.58492e-09  Max(alpha.water) = 1
MULES: Correcting alpha.water
MULES: Correcting alpha.water
Phase-1 volume fraction = 0.127626  Min(alpha.water) = -5.15558e-06  Max(alpha.water) = 1
DILUPBiCG:  Solving for Ux, Initial residual = 0.00700056, Final residual = 2.94138e-09, No Iterations 3
DILUPBiCG:  Solving for Uy, Initial residual = 0.00998841, Final residual = 1.67247e-09, No Iterations 3
DICPCG:  Solving for p_rgh, Initial residual = 0.0158756, Final residual = 0.00013496, No Iterations 6
time step continuity errors : sum local = 3.17548e-05, global = -5.59901e-06, cumulative = -7.36376e-05
DICPCG:  Solving for p_rgh, Initial residual = 0.000889262, Final residual = 7.94541e-06, No Iterations 30
time step continuity errors : sum local = 1.86402e-06, global = -9.55375e-08, cumulative = -7.37331e-05
DICPCG:  Solving for p_rgh, Initial residual = 8.5497e-05, Final residual = 7.6903e-07, No Iterations 33
time step continuity errors : sum local = 1.80667e-07, global = 3.47462e-09, cumulative = -7.37296e-05
ExecutionTime = 9.47 s  ClockTime = 9 s

fieldMinMax minmaxdomain output:
    min(p) = -43.4411 at location (0.0698261 0.584 0.0073)
    max(p) = 979.237 at location (0.23487 0 0.0073)
    min(U) = (0.0129996 -0.0121795 0) at location (0.00634783 0.00299994 0.0073)
    max(U) = (0.0129996 -0.0121795 0) at location (0.00634783 0.00299994 0.0073)
    min(alpha.water) = -5.15558e-06 at location (0.272957 0.105428 0.0073)
    max(alpha.water) = 1 at location (0.0317391 0.00299994 0.0073)

cellSource water_in_domain output:
    volIntegrate() of alpha.water = 0.000633354
```

Flow courant number

Interface courant number

alpha.water residuals

nAlphaCorr 2

nAlphaSubCycles 1
Only one loop

3 pressure correctors and no non-orthogonal corrections

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

alpha.water is bounded between 0 and 1

Volume integral functionObject

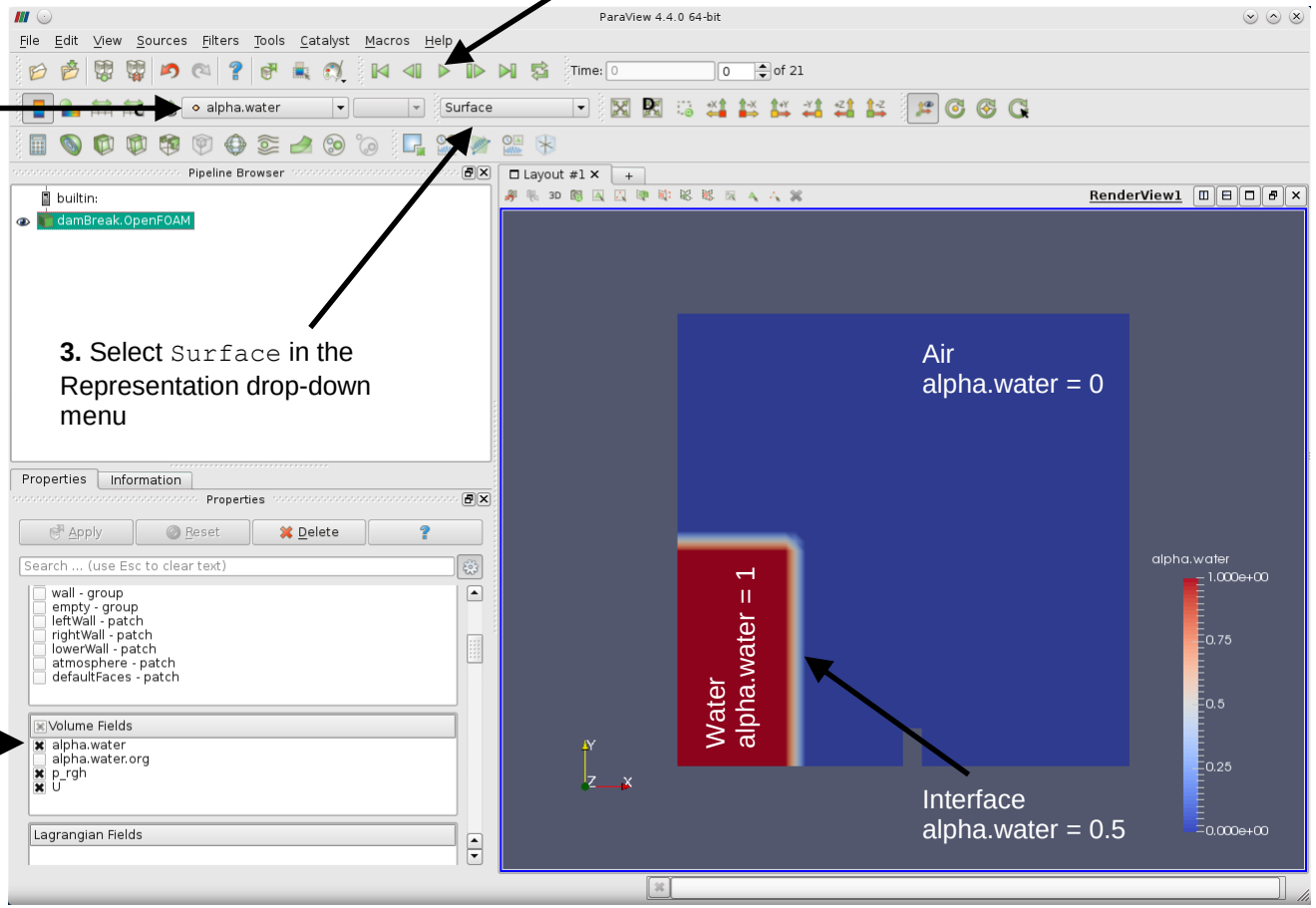
Dam break free surface flow

Post-processing multiphase flows in paraFoam

- To visualize the volume fraction, proceed as follows,

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

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



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

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

Dam break free surface flow

Post-processing multiphase flows in paraFoam

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

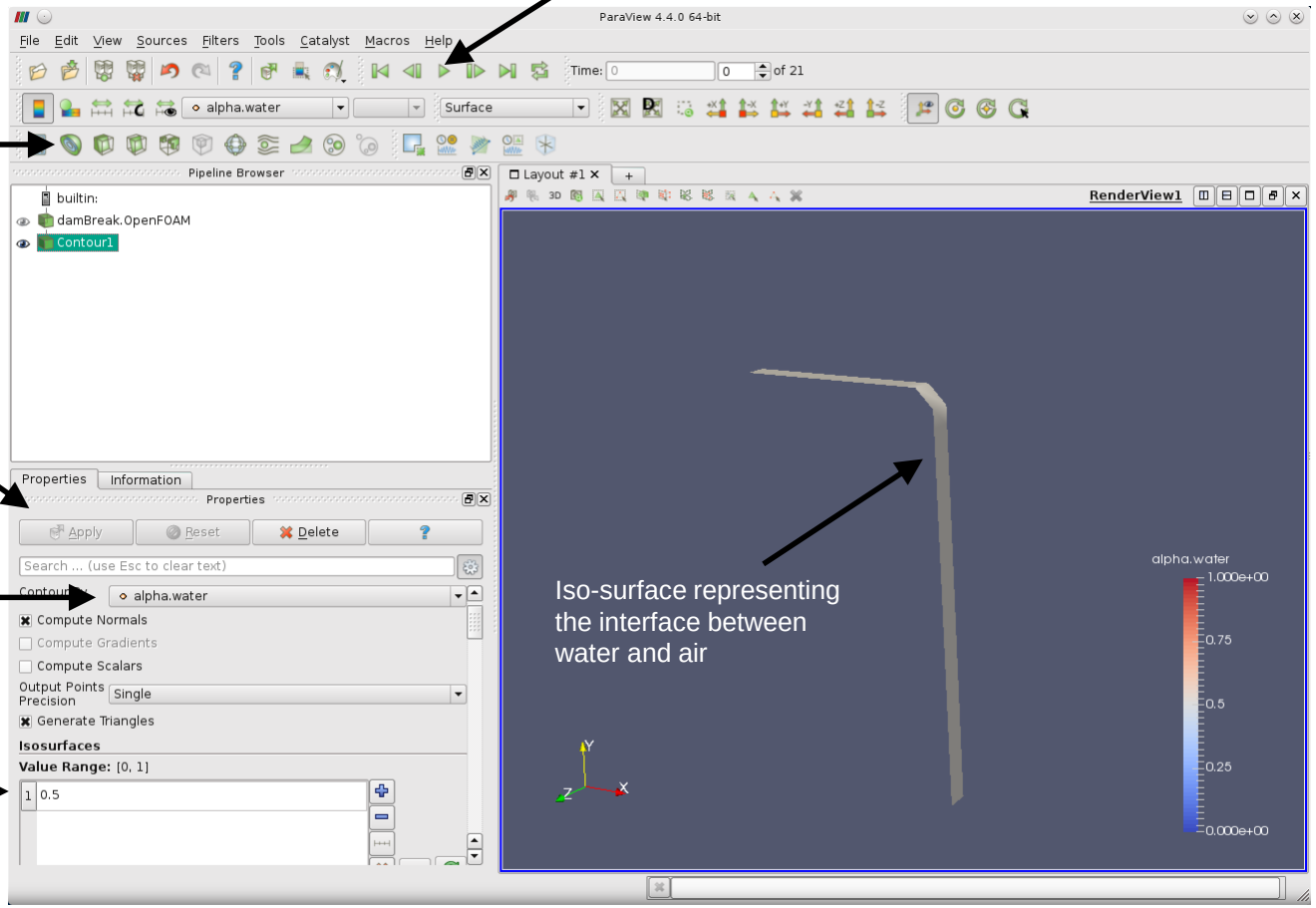
5. To animate the solution, press `Play` in the VCR Controls

1. Select the filter `Contour`

4. Press apply

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

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



Dam break free surface flow

Post-processing multiphase flows in paraFoam

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

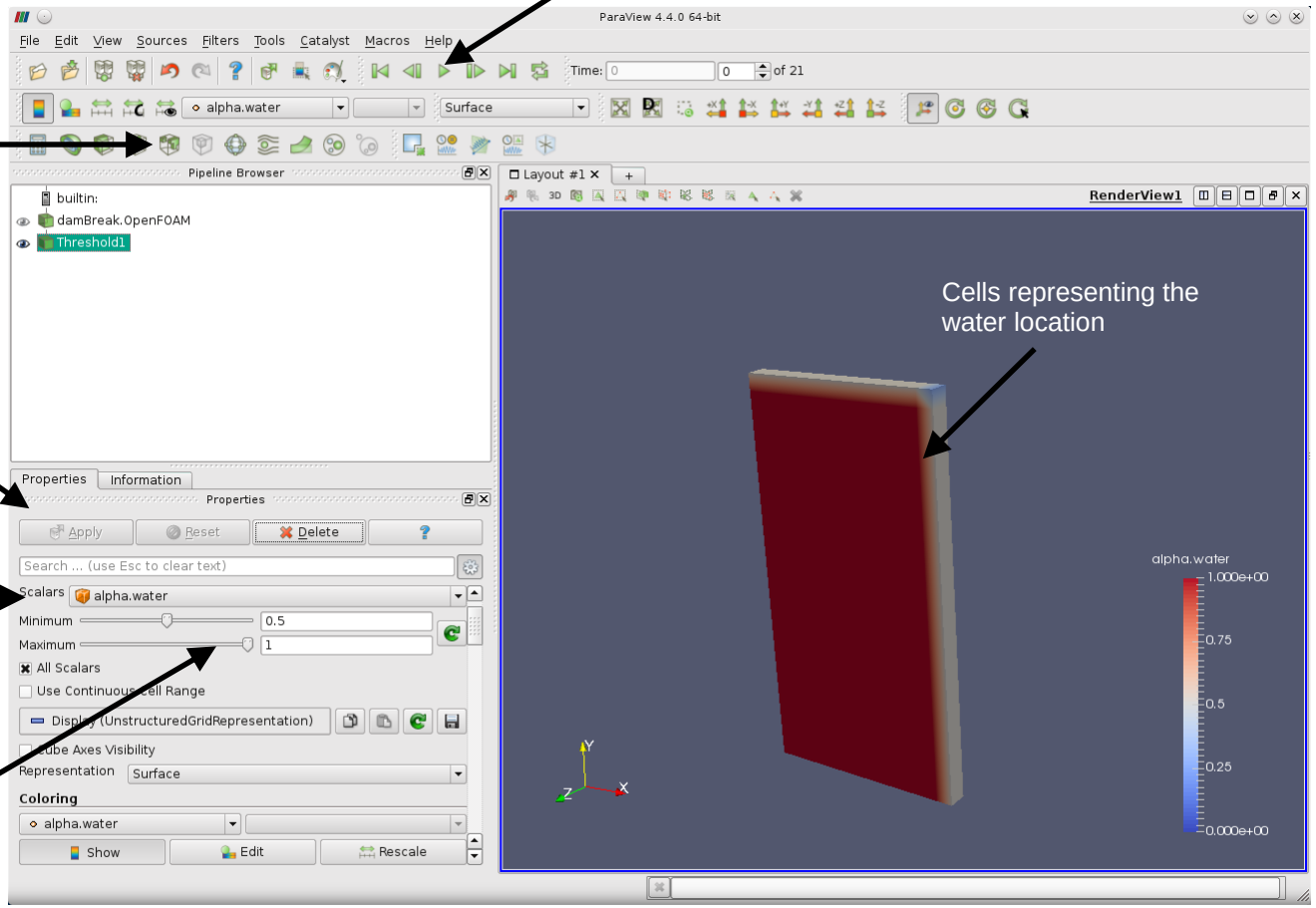
5. To animate the solution, press `Play` in the VCR Controls

1. Select the filter `Threshold`

4. Press apply

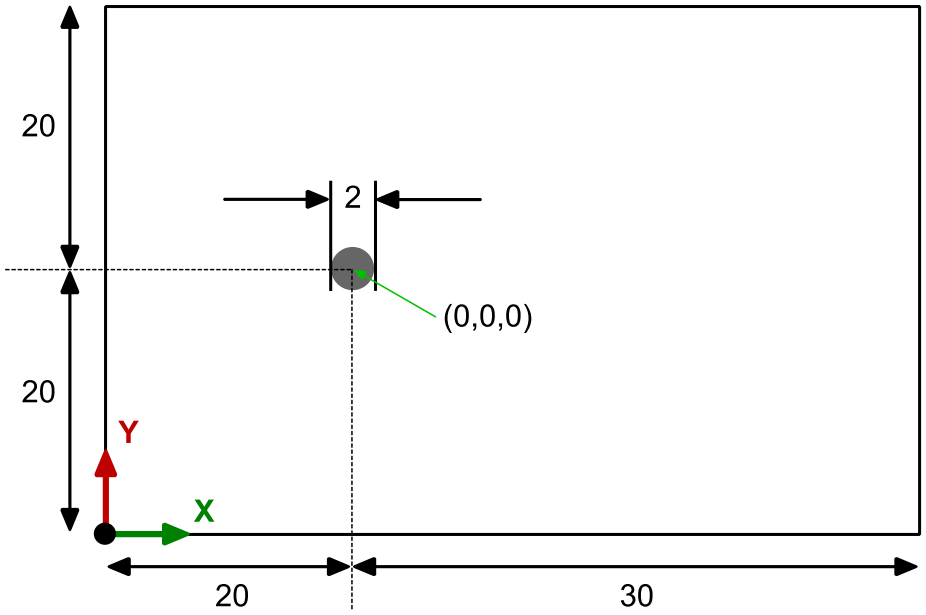
2. Select **alpha.water** or the field you want to use to visualize the cells (it has to be a scalar)

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



Flow past a cylinder – From laminar to turbulent flow

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



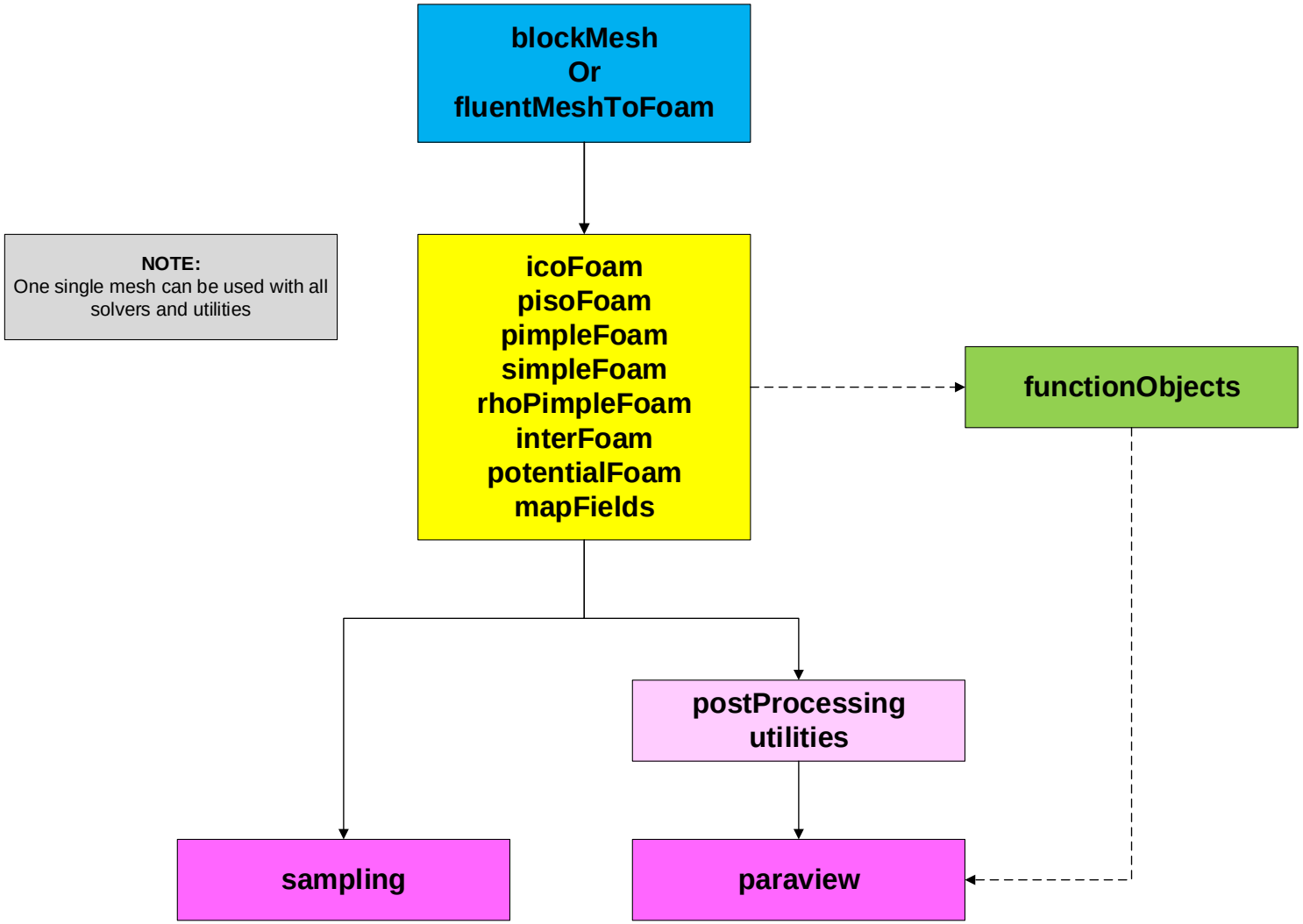
All the dimensions are in meters

Physical and numerical side of the problem:

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

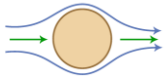
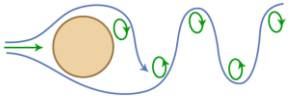
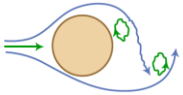
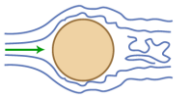
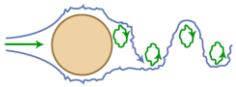
Flow past a cylinder – From laminar to turbulent flow

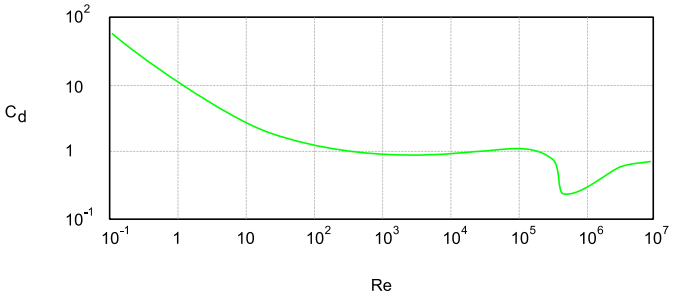
Workflow of the case



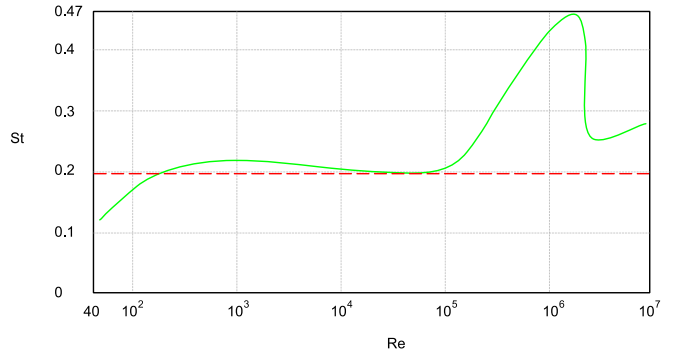
Flow past a cylinder – From laminar to turbulent flow

Vortex shedding behind a cylinder

 Creeping flow (no separation)	$Re < 5$
 A pair of stable vortices in the wake	$5 < Re < 40 - 46$
 Laminar vortex street (Von Karman street)	$40 - 46 < Re < 150$
 Laminar boundary layer up to the separation point, turbulent wake	$150 < Re < 300$ Transition to turbulence $300 < Re < 3 \times 10^5$
 Boundary layer transition to turbulent	$3 \times 10^5 < Re < 3 \times 10^6$
 Turbulent vortex street, but the wake is narrower than in the laminar case	$3 \times 10^6 < Re$



Drag coefficient



Strouhal number

Flow past a cylinder – From laminar to turbulent flow

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

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

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

[1] D. Tritton. Experiments on the flow past a circular cylinder at low Reynolds numbers. Journal of Fluid Mechanics, 6:547-567, 1959.
[2] M. Cuntanceau and R. Bouard. Experimental determination of the main features of the viscous flow in the wake of a circular cylinder in uniform translation. Part 1. Steady flow. Journal of Fluid Mechanics, 79:257-272, 1973.
[3] D. Rusell and Z. Wang. A cartesian grid method for modeling multiple moving objects in 2D incompressible viscous flow. Journal of Computational Physics, 191:177-205, 2003.
[4] D. Calhoun and Z. Wang. A cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. Journal of Computational Physics. 176:231-275, 2002.
[5] T. Ye, R. Mittal, H. Udaykumar, and W. Shyy. An accurate cartesian grid method for viscous incompressible flows with complex immersed boundaries. Journal of Computational Physics, 156:209-240, 1999.
[6] B. Fornberg. A numerical study of steady viscous flow past a circular cylinder. Journal of Fluid Mechanics, 98:819-855, 1980.
[7] J. Guerrero. Numerical simulation of the unsteady aerodynamics of flapping flight. PhD Thesis, University of Genoa, 2009.

Flow past a cylinder – From laminar to turbulent flow

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

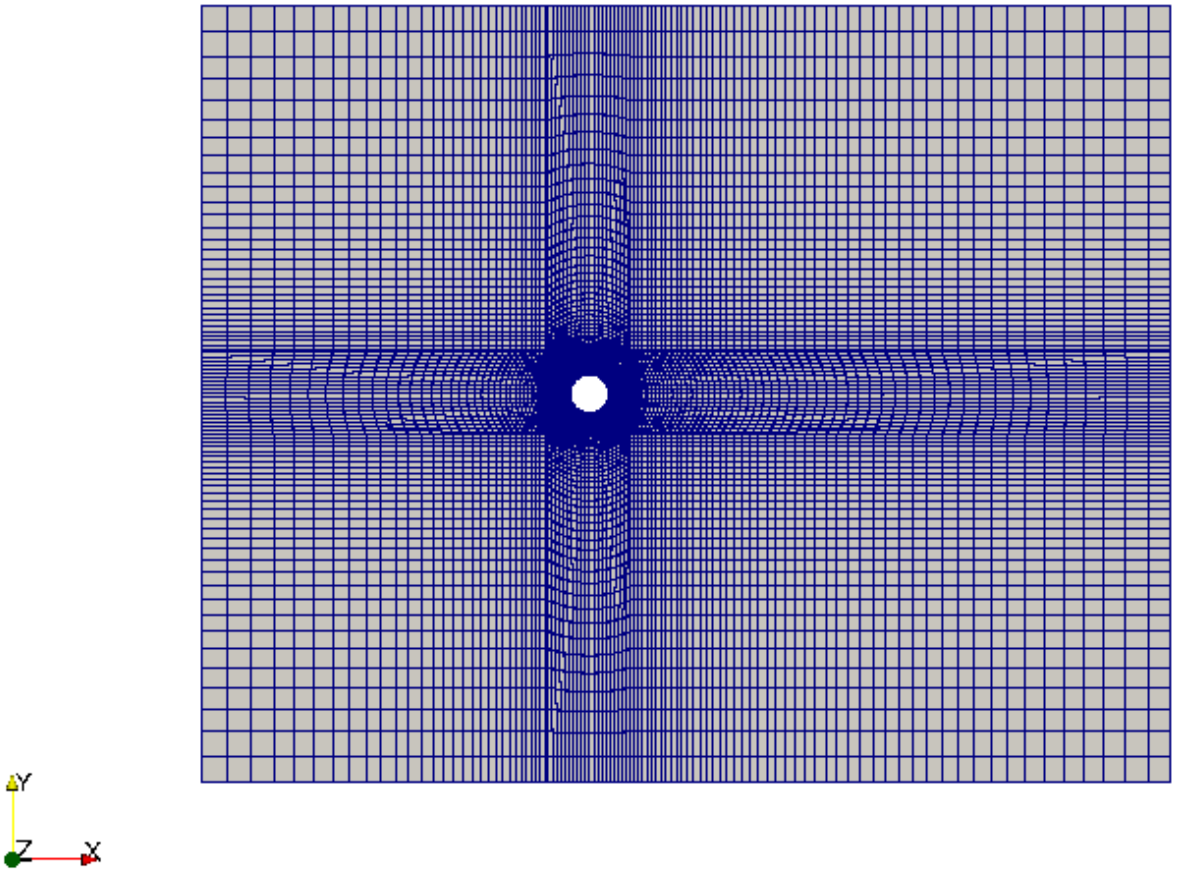
Reference	c_d – Re = 100	c_l – Re = 100	c_d – Re = 200	c_l – Re = 200
[1] Russel and Wang (N)	1.38 ± 0.007	± 0.322	1.29 ± 0.022	± 0.50
[2] Calhoun and Wang (N)	1.35 ± 0.014	± 0.30	1.17 ± 0.058	± 0.67
[3] Braza et al. (N)	1.386± 0.015	± 0.25	1.40 ± 0.05	± 0.75
[4] Choi et al. (N)	1.34 ± 0.011	± 0.315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

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

[1] D. Rusell and Z. Wang. A cartesian grid method for modeling multiple moving objects in 2D incompressible viscous flow. Journal of Computational Physics, 191:177-205, 2003.
[2] D. Calhoun and Z. Wang. A cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. Journal of Computational Physics. 176:231-275, 2002.
[3] M. Braza, P. Chassaing, and H. Hinh. Numerical study and physical analysis of the pressure and velocity fields in the near wake of a circular cylinder. Journal of Fluid Mechanics, 165:79-130, 1986.
[4] J. Choi, R. Oberoi, J. Edwards, an J. Rosati. An immersed boundary method for complex incompressible flows. Journal of Computational Physics, 224:757-784, 2007.
[5] C. Liu, X. Zheng, and C. Sung. Preconditioned multigrid methods for unsteady incompressible flows. Journal of Computational Physics, 139:33-57, 1998.
[6] J. Guerrero. Numerical Simulation of the unsteady aerodynamics of flapping flight. PhD Thesis, University of Genoa, 2009.

Flow past a cylinder – From laminar to turbulent flow

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

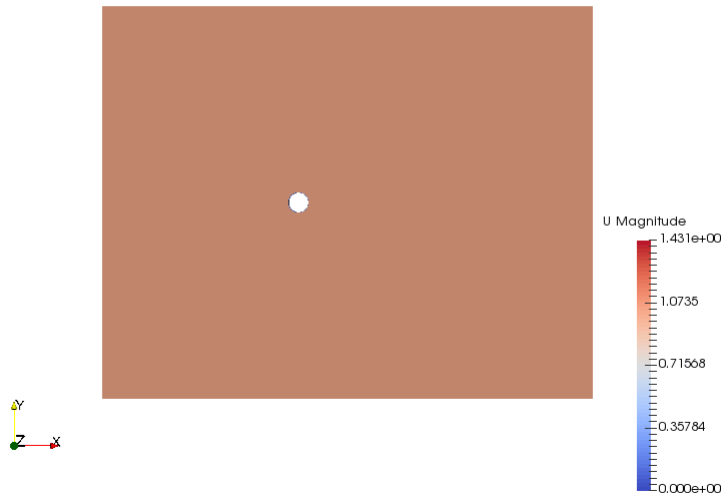


Mesh

Flow past a cylinder – From laminar to turbulent flow

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

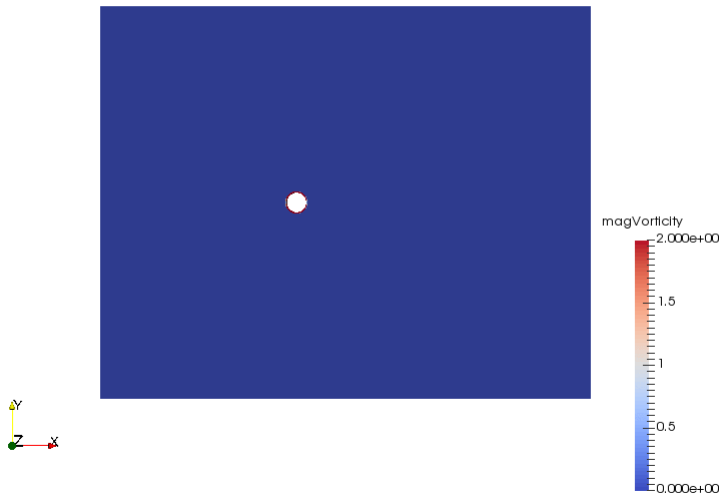
Time: 0.000000



Instantaneous velocity magnitude field

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

Time: 0.000000



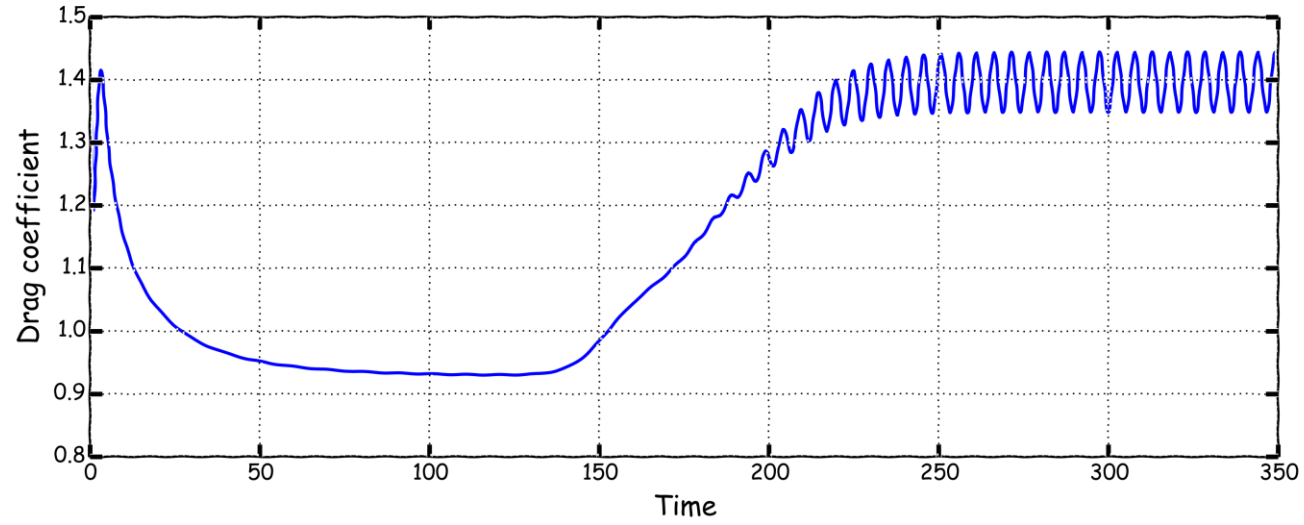
Instantaneous vorticity magnitude field

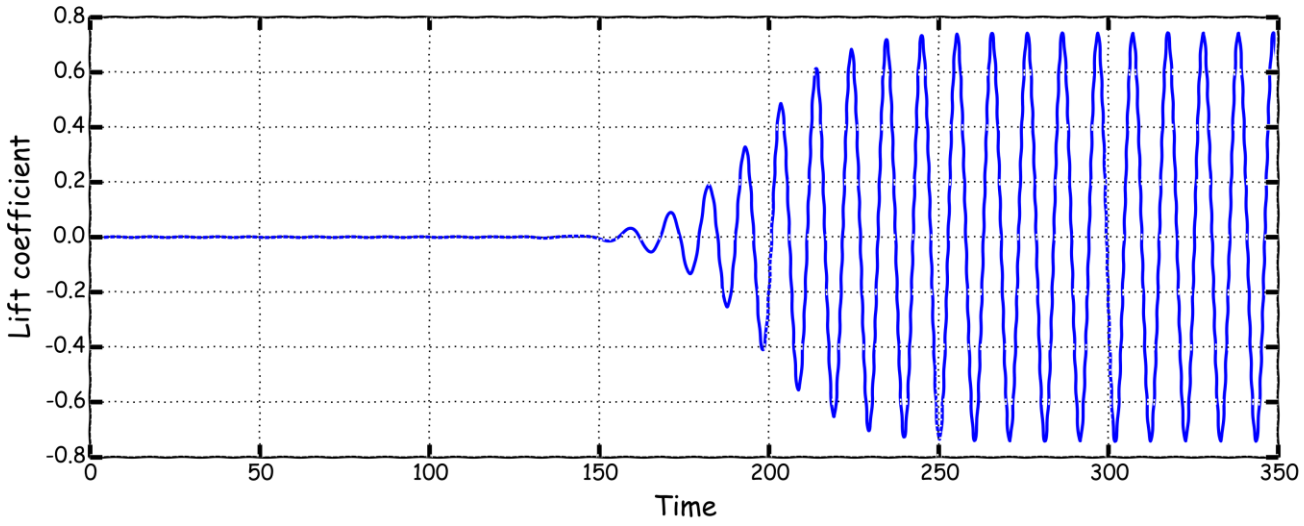
www.wolfdynamics.com/wiki/cylinder_vortex_shedding/movvort.gif

Incompressible flow – Reynolds 200

Flow past a cylinder – From laminar to turbulent flow

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



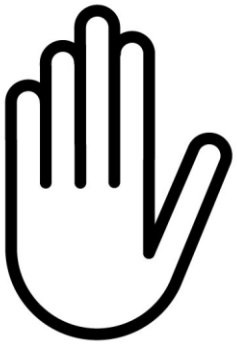


Incompressible flow – Reynolds 200

Flow past a cylinder – From laminar to turbulent flow

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

```
vortex_shedding
```



- From this point on, please follow me.
- We are all going to work at the same pace.
- Remember, `$PTOFC` is pointing to the path where you unpacked the tutorials.

Flow past a cylinder – From laminar to turbulent flow

What are we going to do?

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

Flow past a cylinder – From laminar to turbulent flow

Running the case

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

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

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

Flow past a cylinder – From laminar to turbulent flow

Running the case

- To avoid this problem, type in the terminal,

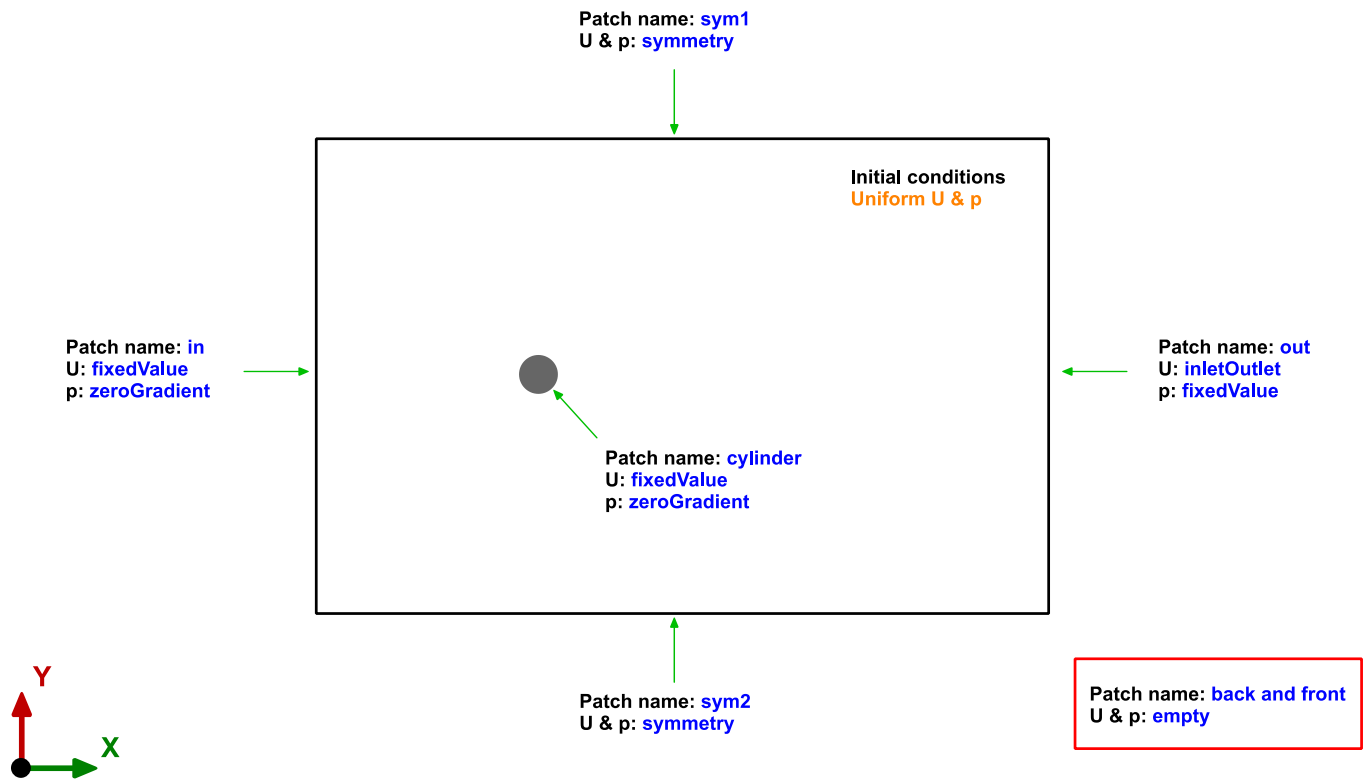
1. `$> paraFoam -builtin`

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

```
Created temporary 'c2.OpenFOAM'


--> FOAM FATAL IO ERROR:

    patch type 'patch' not constraint type 'empty'        <-- What
    for patch front of field p in file "/home/joegi/my_cases_course/4x/101OF/vortex_shedding/c2/0/p"   <-- Where

file: /home/joegi/my_cases_course/4x/101OF/vortex_shedding/c2/0/p.boundaryField.front from line 60 to line
60.

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

FOAM exiting
```

Flow past a cylinder – From laminar to turbulent flow

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



Flow past a cylinder – From laminar to turbulent flow

The *boundary* dictionary file

```
18    7
19    (
20        out
21        {
22            type            patch;
23            nFaces          80;
24            startFace       18180;
25        }
26        sym1
27        {
28            type            symmetry;
29            inGroups        1(symmetry);
30            nFaces          100;
31            startFace       18260;
32        }
33        sym2
34        {
35            type            symmetry;
36            inGroups        1(symmetry);
37            nFaces          100;
38            startFace       18360;
39        }
40        in
41        {
42            type            patch;
43            nFaces          80;
44            startFace       18460;
45        }
```

- This dictionary is located in the **`constant/polyMesh`** directory.
- This file is automatically created when converting the mesh.
- The type of the **out** patch is OK.
- The type of the **sym1** patch is OK.
- The type of the **sym2** patch is OK.
- The type of the **in** patch is OK.

Flow past a cylinder – From laminar to turbulent flow

The *boundary* dictionary file

```
46        cylinder
47        {
48            type            wall;
49            inGroups        1(wall);
50            nFaces          80;
51            startFace       18540;
52        }
53        back
54        {
55            type            patch;
56            nFaces          9200;
57            startFace       18620;
58        }
59        front
60        {
61            type            patch;
62            nFaces          9200;
63            startFace       27820;
64        }
65    )
```

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

Flow past a cylinder – From laminar to turbulent flow

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

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

Flow past a cylinder – From laminar to turbulent flow

Running the case

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

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

Flow past a cylinder – From laminar to turbulent flow

Running the case

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

Flow past a cylinder – From laminar to turbulent flow

- At this point try to use the following utilities. In the terminal type:
  - `$> postProcess -func vorticity -noZero`

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

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

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

    This utility will compute and write the divergence of the velocity field or `grad(U)` in the whole domain (including at the walls). You will need to add the keyword **div(U) Gauss linear**; in the dictionary `fvSchemes`.
  - `$> foamToVTK -time 50:300`

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

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

    This utility will compute and write the wall shear stresses at the walls. As no arguments are given, it will save the wall shear stresses for all time steps.

Flow past a cylinder – From laminar to turbulent flow

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

- In the previous case, it took about 150 seconds to onset the instability.
- If you are not interested in the initial transient, we can add a perturbation in order to promote the onset of the instability. Let us use `setFields` to initialize a non-uniform flow.
- This case is already setup in the directory

```
$PTOFC/101OF/vortex_shedding/c3
```

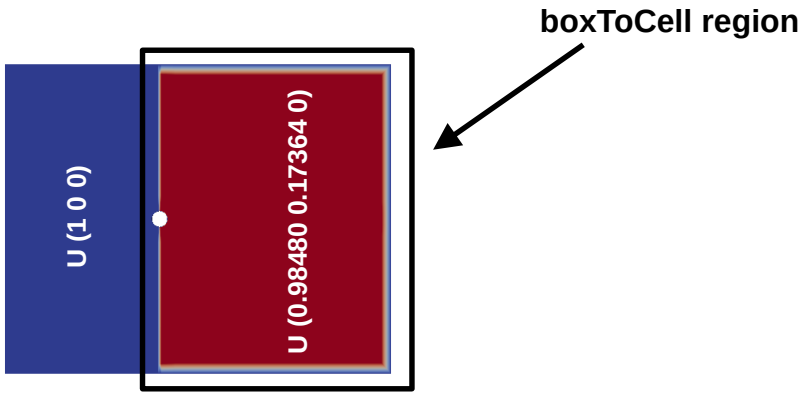
Flow past a cylinder – From laminar to turbulent flow

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

The `setFieldsDict` dictionary

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

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



Flow past a cylinder – From laminar to turbulent flow

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

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c3`**
- Feel free to use the Fluent mesh or the mesh generated with `blockMesh`.
- In this case, we will use `blockMesh`.
- To run this case, in the terminal window type:

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

Flow past a cylinder – From laminar to turbulent flow

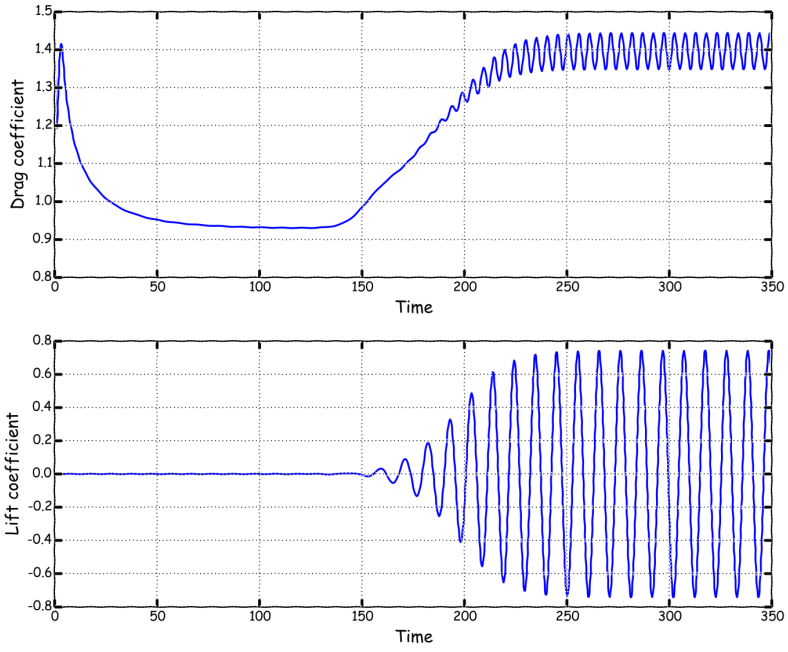
Running the case
Let us run the same case but using a non-uniform field

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

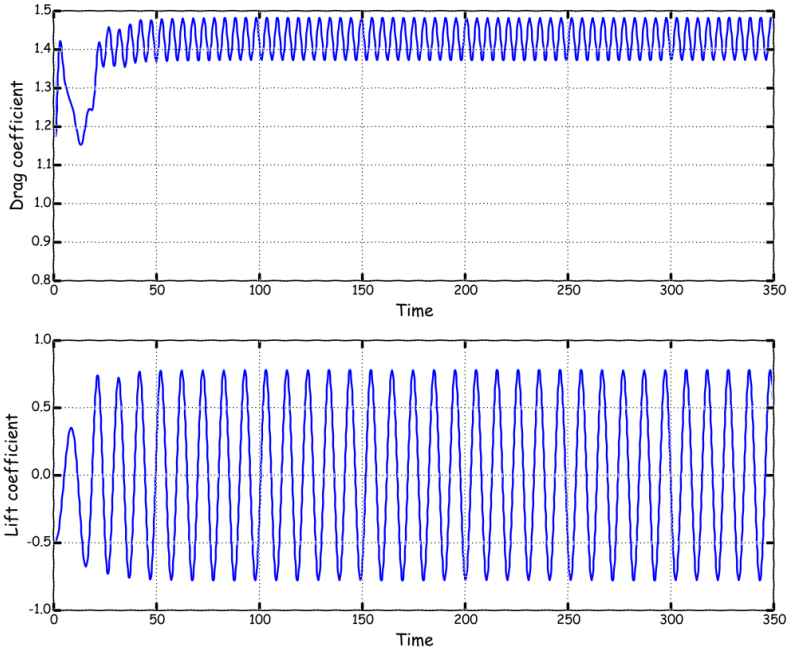
Flow past a cylinder – From laminar to turbulent flow

Does field initialization make a difference?

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



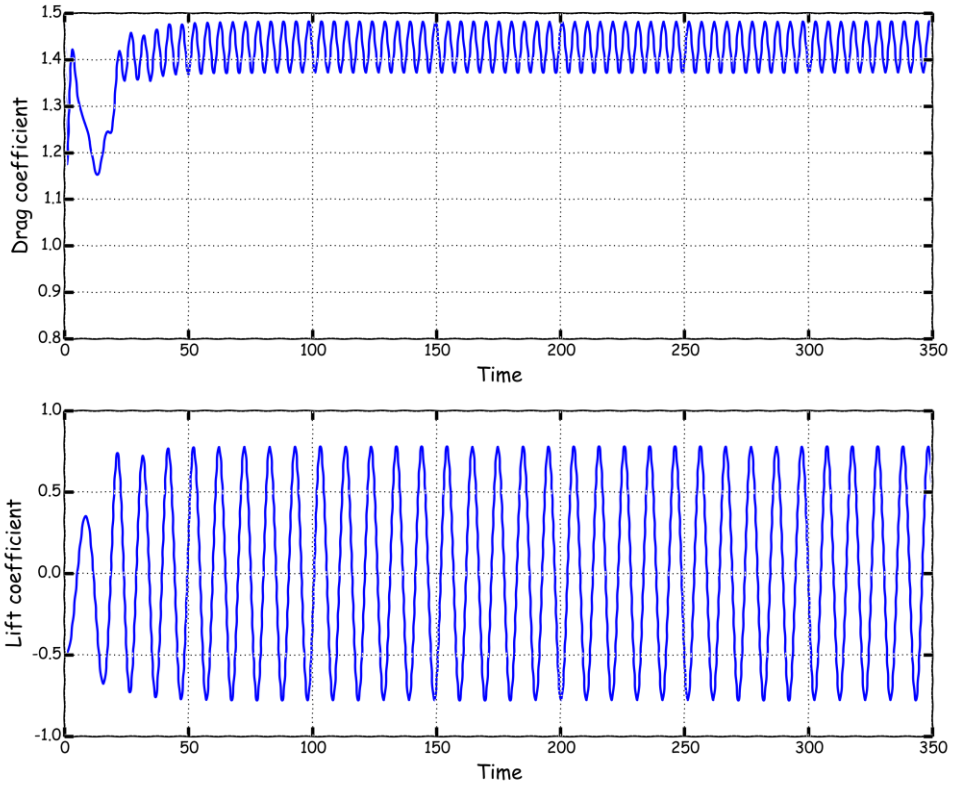
No field initialization



With field initialization

Flow past a cylinder – From laminar to turbulent flow

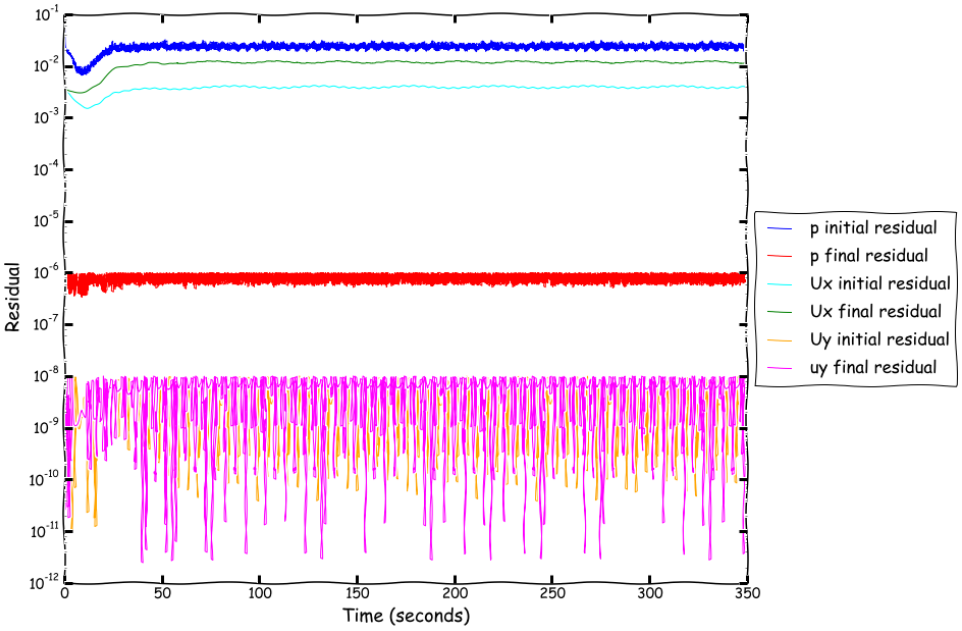
For how long should run the simulation?



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

Flow past a cylinder – From laminar to turbulent flow

What about the residuals?



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

Flow past a cylinder – From laminar to turbulent flow

How to compute force coefficients

```
68      functions
69      {

195         forceCoeffs_object
196         {
205             type forceCoeffs;
206             functionObjectLibs ("libforces.so");
208             patches (cylinder);
209
210             pName p;
211             Uname U;
212             rhoName rhoInf;
213             rhoInf 1.0;
214
215             //// Dump to file
216             log true;
217
218             CofR (0.0 0 0);
219             liftDir (0 1 0);
220             dragDir (1 0 0);
221             pitchAxis (0 0 1);
222             magUInf 1.0;
223             lRef 1.0;
224             Aref 2.0;
225
226             outputControl   timeStep;
227             outputInterval  1;
228         }

255     };
```

- To compute the force coefficients we use **functionObjects**.
- Remember, **functionObjects** are defined at the end of the *controlDict* dictionary file.
- In line 195 we give a name to the **functionObject**.
- In line 208 we define the patch where we want to compute the forces.
- In lines 212-213 we define the reference density value.
- In line 218 we define the center of rotation (for moments).
- In line 219 we define the lift force axis.
- In line 220 we define the drag force axis.
- In line 221 we define the axis of rotation for moment computation.
- In line 223 we give the reference length (for computing the moments)
- In line 224 we give the reference area (in this case the frontal area).
- The output of this **functionObject** is saved in the file *forceCoeffs.dat* located in the directory **forceCoeffs_object/0/**

Flow past a cylinder – From laminar to turbulent flow

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

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

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

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

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

4. `gnuplot> exit`
   To exit gnuplot

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

Flow past a cylinder – From laminar to turbulent flow

On the solution accuracy

```
17    ddtSchemes
18    {
20      default         backward;
22    }
23
24    gradSchemes
25    {
31        default           leastSquares;
37    }
38
39    divSchemes
40    {
41        default         none;
45        div(phi,U)      Gauss linearUpwind default;
49    }
50
51    laplacianSchemes
52    {
59        default         Gauss linear limited 1;
60    }
61
62    interpolationSchemes
63    {
64        default         linear;
66    }
67
68    snGradSchemes
69    {
71        default         limited 1;
72    }
```

- At the end of the day we want a solution that is second order accurate.
- We define the discretization schemes (and therefore the accuracy) in the dictionary *fvSchemes*.
- In this case, for time discretization (**ddtSchemes**) we are using the **backward** method.
- For gradient discretization (**gradSchemes**) we are using the **leastSquares** method.
- For the discretization of the convective terms (**divSchemes**) we are using **linearUpwind** interpolation method for the term **div(rho,U)**.
- For the discretization of the Laplacian (**laplacianSchemes** and **snGradSchemes**) we are using the **Gauss linear limited 1** method
- This method is second order accurate.

Flow past a cylinder – From laminar to turbulent flow

On the solution tolerance and linear solvers

```
17    solvers
18    {
31        p
32        {
33            solver           GAMG;
34            tolerance        1e-6;
35            relTol           0;
36            smoother         GaussSeidel;
37            nPreSweeps       0;
38            nPostSweeps      2;
39            cacheAgglomeration on;
40            agglomerator     faceAreaPair;
41            nCellsInCoarsestLevel 100;
42            mergeLevels      1;
43        }
44
45        pFinal
46        {
47            $p;
48            relTol           0;
49        }
50
51        U
52        {
53            solver           PBiCG;
54            preconditioner   DILU;
55            tolerance        1e-08;
56            relTol           0;
57        }
69    }
70
71    PISO
72    {
73        nCorrectors     2;
74        nNonOrthogonalCorrectors 2;
77    }
```

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

Flow past a cylinder – From laminar to turbulent flow

On the runtime parameters

```
17      application     icoFoam;
18
20      startFrom       latestTime;
21
22      startTime       0;
23
24      stopAt          endTime;
26
27      endTime         350;
29
33      deltaT          0.05;
34
35      writeControl    runTime;
43
44      writeInterval   1;
45
52      purgeWrite      0;
53
54      writeFormat     ascii;
55
56      writePrecision  8;
57
58      writeCompression off;
59
60      timeFormat      general;
61
62      timePrecision   6;
63
64      runTimeModifiable true;
```

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

Flow past a cylinder – From laminar to turbulent flow

The output screen

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

```
Time = 350

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

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

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

fieldAverage fieldAverage output:
    Calculating averages

    Writing average fields

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

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

nCorrector 1

nCorrector 2

Courant number

nNonOrthogonalCorrectors 2

pFinal

nCorrectors 2

Mass flow at in patch

Mass flow at out patch

Computing averages of fields

Force coefficients

Min and max values

Flow past a cylinder – From laminar to turbulent flow

Let us run the same case but this time using a potential solver

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

`$PTOFC/101OF/vortex_shedding/c4`

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

Try to spot the differences.

Flow past a cylinder – From laminar to turbulent flow

Running the case
Let us run the same case but this time using a potential solver

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

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

Flow past a cylinder – From laminar to turbulent flow

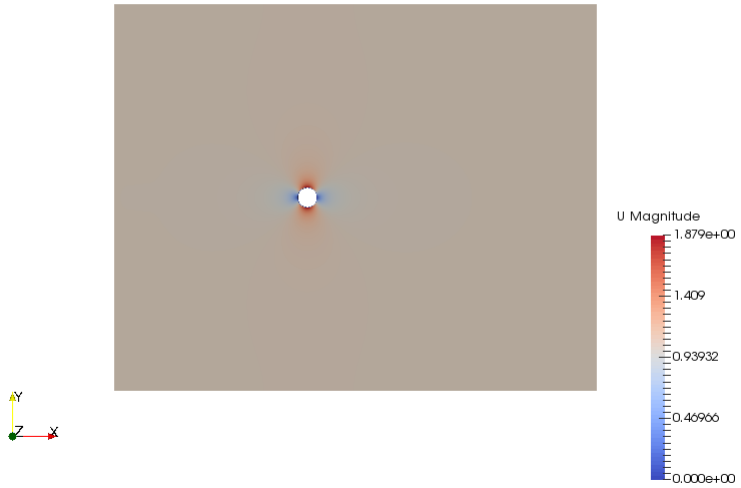
Running the case
Let us run the same case but this time using a potential solver

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

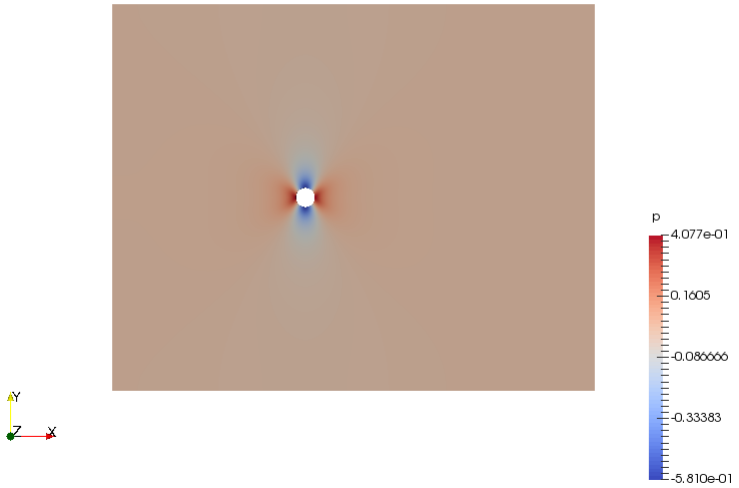
Flow past a cylinder – From laminar to turbulent flow

Potential solution

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



Velocity field



Pressure field

Flow past a cylinder – From laminar to turbulent flow

The output screen

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

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

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

End
```

Velocity computation

Initial approximation

nNonOrthogonalCorrectors 5

Pressure computation

Flow past a cylinder – From laminar to turbulent flow

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

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

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

Flow past a cylinder – From laminar to turbulent flow

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

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

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

Flow past a cylinder – From laminar to turbulent flow

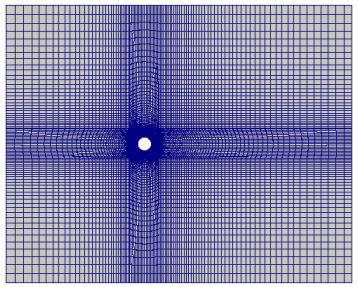
Running the case
Let us map a solution from a coarse mesh to a finer mesh

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

Flow past a cylinder – From laminar to turbulent flow

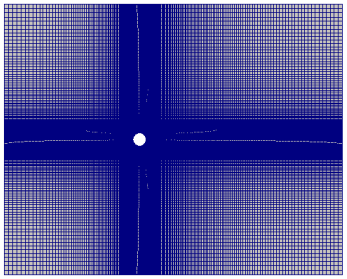
The meshes and the mapped fields

Coarse mesh



`mapFields`

Fine mesh



Flow past a cylinder – From laminar to turbulent flow

The output screen

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

```
Source: ".." "c4"
Target: "/home/joegi/my_cases_course/30x/101OF/vortex_shedding" "c6"
Mapping method: cellPointInterpolate

Create databases as time

Source time: 0
Target time: 0
Create meshes

Source mesh size: 9200  Target mesh size: 36800


Consistently creating and mapping fields for time 0

    interpolating Phi
    interpolating p
    interpolating U

End
```

Source case

Target case

Interpolation method

Source time

Target time

Source and target mesh cell count

Interpolated fields

Flow past a cylinder – From laminar to turbulent flow

Setting a turbulent case

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

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

Flow past a cylinder – From laminar to turbulent flow

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

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

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

Flow past a cylinder – From laminar to turbulent flow

The *transportProperties* dictionary file

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

```
16    transportModel  Newtonian;
17
19    nu              nu [ 0 2 -1 0 0 0 0 ] 0.0002;
```

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

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

Flow past a cylinder – From laminar to turbulent flow

The `turbulenceProperties` dictionary file

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

```
17    simulationType  RAS;          <-- RANS type simulation
18
19    RAS                           <-- RANS sub-dictionary
20    {
21        RASModel        kOmegaSST;    <-- RANS model to use
22
23        turbulence      on;           <-- Turn on/off turbulence.  Runtime modifiable
24
25        printCoeffs     on;           <-- Print coefficients at the beginning
26    }
```

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

Flow past a cylinder – From laminar to turbulent flow

The *controlDict* dictionary

```
17      application     pimpleFoam;
18
20      startFrom       latestTime;
21
22      startTime       0;
23
24      stopAt          endTime;
25
26      endTime         500;
27
28      deltaT          0.001;
32
33      writeControl    runTime;
41
42      writeInterval   1;
43
50      purgeWrite      0;
51
52      writeFormat     ascii;
53
54      writePrecision  8;
55
56      writeCompression off;
57
58      timeFormat      general;
59
60      timePrecision   6;
61
62      runTimeModifiable yes;
63
64      adjustTimeStep  yes;
65
66      maxCo           0.9;
67      maxDeltaT       0.1;
```

- This case will start from the last saved solution (**startFrom**). If there is no solution, the case will start from time 0 (**startTime**).
- It will run up to 500 seconds (**endTime**).
- The initial time step of the simulation is 0.001 seconds (**deltaT**).
- It will write the solution every 1 second (**writeInterval**) of simulation time (**runTime**).
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**).
- The write precision is 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on, we can modify all these entries while we are running the simulation.
- In line 64 we turn on the option **adjustTimeStep**. This option will automatically adjust the time step to achieve the maximum desired courant number **maxCo** (line 66).
- We also set a maximum time step **maxDeltaT** in line 67.
- Remember, the first time step of the simulation is done using the value set in line 28 and then it is automatically scaled to achieve the desired maximum values (lines 66-67).
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.

Flow past a cylinder – From laminar to turbulent flow

The *fvSchemes* dictionary

```
17    ddtSchemes
18    {
21        default         CrankNicolson 0.5;
22    }
24    gradSchemes
25    {
31        default           leastSquares;
36        grad(U)           cellMDLimited Gauss linear 1;
37    }
39    divSchemes
40    {
41        default         none;
47        div(phi,U)      Gauss linearUpwindV grad(U);
49        div((nuEff*dev2(T(grad(U))))) Gauss linear;
51        div(phi,k)          Gauss linearUpwind default;
52        div(phi,omega)      Gauss linearUpwind default;
63    }
65    laplacianSchemes
66    {
74         default         Gauss linear limited 1;
75    }
77    interpolationSchemes
78    {
79        default         linear;
81    }
83    snGradSchemes
84    {
86        default         limited 1;
87    }
89    wallDist
90    {
91        method meshWave;
92    }
```

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

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
17      solvers
18      {
31          p
32          {
33              solver           GAMG;
34              tolerance        1e-6;
35              relTol           0.001;
36              smoother         GaussSeidel;
37              nPreSweeps       0;
38              nPostSweeps      2;
39              cacheAgglomeration on;
40              agglomerator     faceAreaPair;
41              nCellsInCoarsestLevel 100;
42              mergeLevels      1;
44              minIter        2;
45          }
46
47          pFinal
48          {
49              solver           PCG;
50              preconditioner   DIC;
51              tolerance        1e-06;
52              relTol           0;
54              minIter        3;
55          }
56
57          U
58          {
59              solver           PBiCG;
60              preconditioner   DILU;
61              tolerance        1e-08;
62              relTol           0;
63              minIter        3;
64          }
```

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

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
17      solvers
18      {
77          UFinal
78          {
79              solver          PBiCG;
80              preconditioner  DILU;
81              tolerance       1e-08;
82              relTol          0;
83              minIter         3;
84          }
85
86          omega
87          {
88              solver          PBiCG;
89              preconditioner  DILU;
90              tolerance       1e-08;
91              relTol          0;
92              minIter         3;
93          }
94
95          omegaFinal
96          {
97              solver          PBiCG;
98              preconditioner  DILU;
99              tolerance       1e-08;
100             relTol          0;
101             minIter         3;
102         }
103
104         k
105         {
106             solver          PBiCG;
107             preconditioner  DILU;
108             tolerance       1e-08;
109             relTol          0;
110             minIter         3;
111         }
```

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

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

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

- To solve **kFinal** we are using the solver **PBiCG** with an absolute **tolerance** of 1e-8 and a relative tolerance **relTol** of 0. Notice that we are fixing the number of minimum iterations (**minIter**).
- In lines 123-133 we setup the entries related to the pressure-velocity coupling method used (**PIMPLE** in this case). Setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- To gain more stability we are using 2 outer correctors (**nOuterCorrectors**), 2 inner correctors or **PISO** correctors (**nCorrectors**), and 1 correction due to non-orthogonality (**nNonOrthogonalCorrectors**).
- Remember, adding corrections increase the computational cost.
- In lines 135-147 we setup the under relaxation factors used during the outer corrections (pseudo transient iterations). If you are working in **PISO** mode (only one outer correction or **nOuterCorrectors**), these values are ignored.

Flow past a cylinder – From laminar to turbulent flow

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

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

- As we are going to use the $\kappa - \omega\ SST$ model we need to define the initial conditions and boundaries conditions.
- To define the IC/BC we will use the free stream values of κ and ω

Flow past a cylinder – From laminar to turbulent flow

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

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

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

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

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

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

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

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

Flow past a cylinder – From laminar to turbulent flow

The file *0/k*

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

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

Flow past a cylinder – From laminar to turbulent flow

The file *0/omega*

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

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

Flow past a cylinder – From laminar to turbulent flow

The file *0/nut*

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

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

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

Flow past a cylinder – From laminar to turbulent flow

Running the case – Setting a turbulent case

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

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> renumberMesh -overwrite`
4. `$> pimpleFoam > log &`
5. `$> pyFoamPlotWatcher.py log`
6. `$> gnuplot scripts0/plot_coeffs`
   You will need to launch this script in a different terminal
7. `$> yPlus`
8. `$> paraFoam`

Flow past a cylinder – From laminar to turbulent flow

Running the case
Setting a turbulent case

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

Flow past a cylinder – From laminar to turbulent flow

`pimpleFoam` **output screen**

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

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

fieldAverage fieldAverage output:
    Calculating averages

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

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

Courant number

Time step

Simulation time

Outer iteration 1 (nOuterCorrectors)

Outer iteration 2 (nOuterCorrectors)

pFinal

kappa and omega residuals

Message letting you know that the variable is becoming unbounded

Force coefficients

Minimum and maximum values

Flow past a cylinder – From laminar to turbulent flow

The output screen

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

```
Time = 500.01
Reading field U

Reading/calculating face flux field phi

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

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

Writing yPlus to field yPlus
```

Transport model

Turbulence model

Model coefficients

Patch where we are computing y+

Minimum, maximum and average values

Writing the field to the solution directory

Flow past a cylinder – From laminar to turbulent flow

Using a compressible solver

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

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

- When working with compressible solver we need to define the thermodynamical properties of the working fluid and the temperature field (we are also solving the energy equation)
- This case is already setup in the directory

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

Flow past a cylinder – From laminar to turbulent flow

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

- Reminder:
  - The diameter of the cylinder is 0.0002 m.
  - The working fluid is air at 20° Celsius and at a sea level.
  - Isothermal flow.
  - And we are targeting for a Re = 200.

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

Flow past a cylinder – From laminar to turbulent flow

The **`constant`** directory

- In this directory, we will find the following compulsory dictionary files:
  - *`thermophysicalProperties`*
  - *`turbulenceProperties`*

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

Flow past a cylinder – From laminar to turbulent flow

The *thermophysicalProperties* dictionary file

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

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

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

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

Flow past a cylinder – From laminar to turbulent flow

The *thermophysicalProperties* dictionary file

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

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

Flow past a cylinder – From laminar to turbulent flow

The *turbulenceProperties* dictionary file

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

```
17    simulationType    laminar;
```

Flow past a cylinder – From laminar to turbulent flow

The **0** directory

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

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

Flow past a cylinder – From laminar to turbulent flow

The file *0/p*

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

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

Flow past a cylinder – From laminar to turbulent flow

The file *0/T*

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

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

Flow past a cylinder – From laminar to turbulent flow

The file *0/U*

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

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

Flow past a cylinder – From laminar to turbulent flow

The **`system`** directory

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

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

Flow past a cylinder – From laminar to turbulent flow

The *controlDict* dictionary

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

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

Flow past a cylinder – From laminar to turbulent flow

The *controlDict* dictionary

```
55      functions
56      {

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


235
236     };
```

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

Flow past a cylinder – From laminar to turbulent flow

The *fvSchemes* dictionary

```
17    ddtSchemes
18    {
19        default         Euler;
20    }
21
22    gradSchemes
23    {
29        default           leastSquares;
34    }
35
36    divSchemes
37    {
38        default         none;
39        div(phi,U)      Gauss linearUpwind default;
40
41        div(phi,K)      Gauss linear;
42        div(phi,h)      Gauss linear;
43
44        div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
45    }
46
47    laplacianSchemes
48    {
49        default         Gauss linear limited 1;
50    }
51
52    interpolationSchemes
53    {
54        default         linear;
55    }
56
57    snGradSchemes
58    {
59        default         limited 1;
60    }
```

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

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
17      solvers
18      {
20          p
21          {
22              solver          PCG;
23              preconditioner  DIC;
24              tolerance       1e-06;
25              relTol          0.01;
26              minIter         2;
27          }
46          pFinal
47          {
48              $p;
49              relTol          0;
50              minIter         2;
51          }
53          "U.*"
54          {
55              solver          PBiCG;
56              preconditioner  DILU;
57              tolerance       1e-08;
58              relTol          0;
59              minIter         2;
60          }
74          hFinal
75          {
76              solver          PBiCG;
77              preconditioner  DILU;
78              tolerance       1e-08;
79              relTol          0;
80              minIter         2;
81          }
83          "rho.*"
84          {
85              solver          diagonal;
86          }
87      }
```

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

Flow past a cylinder – From laminar to turbulent flow

The *fvSolution* dictionary

```
88
89      PIMPLE
90      {
91          momentumPredictor yes;
92          nOuterCorrectors 1;
93          nCorrectors     2;
94          nNonOrthogonalCorrectors 1;
95          rhoMin          0.5;
96          rhoMax          2.0;
97      }
```

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

Flow past a cylinder – From laminar to turbulent flow

Running the case – Using a compressible solver

- You will find this tutorial in the directory **`$PTOFC/101OF/vortex_shedding/c23`**
- Feel free to use the Fluent mesh or the mesh generated with `blockMesh`. In this case we will use `blockMesh`.
- To run this case, in the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> transformPoints -scale '(0.001 0.001 0.001)'`
4. `$> renumberMesh -overwrite`
5. `$> rhoPimpleFoam > log &`
6. `$> pyFoamPlotWatcher.py log`
7. `$> gnuplot scripts0/plot_coeffs`
   You will need to launch this script in a different terminal
8. `$> Mach`
9. `$> paraFoam`

Flow past a cylinder – From laminar to turbulent flow

Running the case
Using a compressible solver

- In step 3 we scale the mesh.
- In step 4 we use the utility `renumberMesh` to make the linear system more diagonal dominant, this will speed-up the linear solvers.
- In step 5 we run the simulation and save the log file. Notice that we are sending the job to background.
- In step 6 we use `pyFoamPlotWatcher.py` to plot the residuals on-the-fly. As the job is running in background, we can launch this utility in the same terminal tab.
- In step 7 we use the gnuplot script `scripts0/plot_coeffs` to plot the force coefficients on-the-fly. Besides monitoring the residuals, is always a good idea to monitor a quantity of interest. Feel free to take a look at the script and to reuse it.
- In step 8 we use the utility `Mach` to compute the Mach number.

Flow past a cylinder – From laminar to turbulent flow

`rhoPimpleFoam` **output screen**

```
Courant Number mean: 0.1280224248 max: 0.9885863338
deltaT = 3.816512052e-05
Time = 0.3

diagonal:  Solving for rho, Initial residual = 0, Final residual = 0, No Iterations 0
PIMPLE: iteration 1
DILUPBiCG:  Solving for Ux, Initial residual = 0.003594731129, Final residual = 3.026673755e-11, No Iterations 5
DILUPBiCG:  Solving for Uy, Initial residual = 0.01296036298, Final residual = 1.223236662e-10, No Iterations 5
DILUPBiCG:  Solving for h, Initial residual = 0.01228951539, Final residual = 2.583236461e-09, No Iterations 4
DICPCG:  Solving for p, Initial residual = 0.01967621449, Final residual = 8.797612158e-07, No Iterations 77
DICPCG:  Solving for p, Initial residual = 0.003109422612, Final residual = 9.943030465e-07, No Iterations 69
diagonal:  Solving for rho, Initial residual = 0, Final residual = 0, No Iterations 0
time step continuity errors : sum local = 6.835363016e-11, global = 4.328592697e-12, cumulative = 2.366774797e-09
rho max/min : 1.201420286 1.201382023
DICPCG:  Solving for p, Initial residual = 0.003160602108, Final residual = 9.794177338e-07, No Iterations 69
DICPCG:  Solving for p, Initial residual = 0.0004558492254, Final residual = 9.278622052e-07, No Iterations 58
diagonal:  Solving for rho, Initial residual = 0, Final residual = 0, No Iterations 0
time step continuity errors : sum local = 6.38639685e-11, global = 1.446434866e-12, cumulative = 2.368221232e-09
rho max/min : 1.201420288 1.201381976
ExecutionTime = 480.88 s  ClockTime = 490 s

faceSource inMassFlow output:
    sum(in) of phi = -7.208447027e-05

faceSource outMassFlow output:
    sum(out) of phi = 7.208444452e-05

fieldAverage fieldAverage output:
    Calculating averages

    Writing average fields

forceCoeffs forceCoeffs_object output:
    Cm    = -0.001269886395
    Cd    = 1.419350733
    Cl    = 0.6247248606
    Cl(f) = 0.3110925439
    Cl(r) = 0.3136323167

fieldMinMax minmaxdomain output:
    min(p) = 101322.7878 at location (-0.0001215826043 0.001027092827 0)
    max(p) = 101326.4972 at location (-0.001033408037 -4.061934599e-05 0)
    min(U) = (-0.526856427 -0.09305459972 -8.110485132e-25) at location (0.002039092041 -0.0004058872656 -3.893823418e-20)
    max(U) = (2.184751599 0.2867627526 4.83091257e-25) at location (0.0001663574444 0.001404596295 0)
    min(T) = 293.1487423 at location (-5.556854517e-05 0.001412635233 0)
    max(T) = 293.1509903 at location (-0.00117685237 -4.627394552e-05 3.016083257e-20)
```

Courant number

Time step

Solving for density (rho)

h residuals

pFinal

Solving for density (rhoFinal)

Max/min density values

Force coefficients

Minimum and maximum values

Flow past a cylinder – From laminar to turbulent flow

- In the directory:

```
vortex_shedding
```

You will find 26 variations of the cylinder case involving different solvers and models. Feel free to explore all them.

Flow past a cylinder – From laminar to turbulent flow

- This is what you will find in each directory,
  - c1 = blockMesh – icoFoam – Re = 200.
  - c2 = fluentMeshToFoam – icoFoam – Re = 200.
  - c3 = blockMesh – icoFoam – Field initialization – Re = 200.
  - c4 = blockMesh – potentialFoam – Re = 200.
  - c5 = blockMesh – mapFields – icoFoam – original mesh – Re = 200.
  - c6 = blockMesh – mapFields – icoFoam – Finer mesh – Re = 200.
  - c7 = blockMesh – pimpleFoam – Re = 200 – No turbulent model.
  - c8 = blockMesh – pisoFoam – Re = 200 – No turbulent model.
  - c9 = blockMesh – pisoFoam – Re = 200 – K-Omega SST turbulent model.
  - c10 = blockMesh – simpleFoam – Re = 200 – No turbulent model.
  - c11 = blockMesh – simpleFoam – Re = 40 – No turbulent model.
  - c12 = blockMesh – pisoFoam – Re = 40 – No turbulent model.
  - c14 = blockMesh – pimpleFoam – Re = 10000 – K-Omega SST turbulent model with wall functions.
  - c15 = blockMesh – pimpleFoam – Re = 100000 – K-Omega SST turbulent model with wall functions.
  - c16 = blockMesh – simpleFoam – Re = 100000 – K-Omega SST turbulent model with no wall functions.
  - c17 = blockMesh – simpleFoam – Re = 100000 – K-Omega SST turbulent model with wall functions.
  - c18 = blockMesh – pisoFoam – Re = 100000, LES Smagorinsky turbulent model.

Flow past a cylinder – From laminar to turbulent flow

- This is what you will find in each directory,
  - c19 = blockMesh – pimpleFoam – Re = 1000000 – Spalart Allmaras turbulent model with no wall functions.
  - c20 = blockMesh – sonicFoam – Mach = 2.0 – Compressible – Laminar.
  - c21 = blockMesh – sonicFoam – Mach = 2.0 – Compressible – K-Omega SST turbulent model with wall functions.
  - c22 = blockMesh – pimpleFoam – Re = 200 – No turbulent model – Source terms (momentum)
  - c23 = blockMesh – pimpleFoam – Re = 200 – No turbulent model – Source terms (scalar transport)
  - c24 = blockMesh – rhoPimpleFoam – Re = 200 – Laminar, isothermal
  - c26 = blockMesh – pimpleDyMFoam – Re = 200 – Laminar, moving cylinder (oscillating).
  - c27 = blockMesh – pimpleDyMFoam/pimpleFoam – Re = 200 – Laminar, rotating cylinder using AMI patches.
  - c28 = blockMesh – interFoam – Laminar, multiphase, free surface.