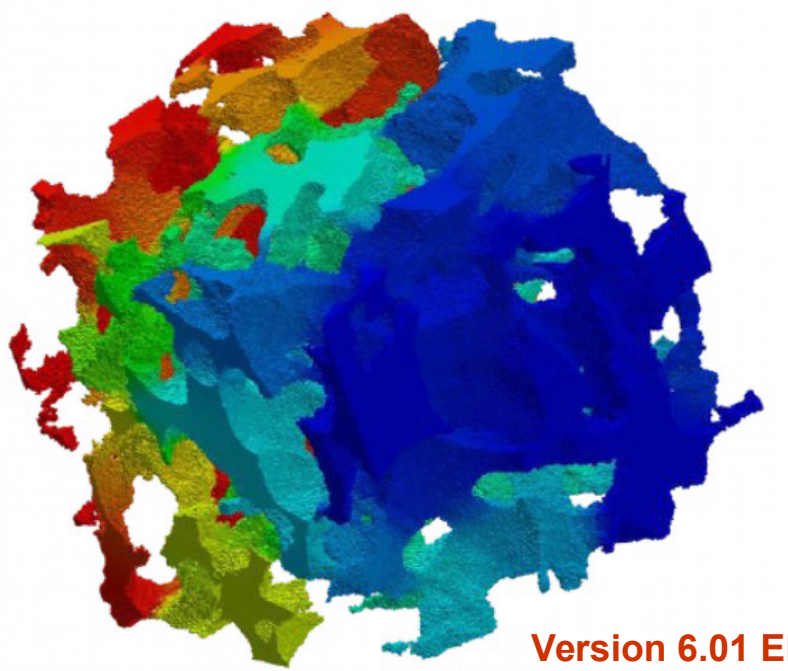
Introduction to open-source computational fluid dynamics using OpenFOAM® technology

« Simulation in porous media from pore to reservoir scale »

April 3-5, 2018, Orléans, France



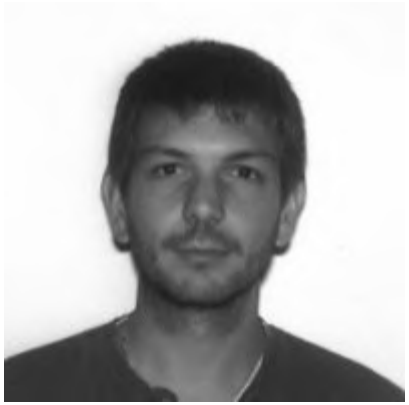
Cyprien Soulaine

Contact: cyprien.soulaine@gmail.com

https://www.cypriensoulaine.com

Version 6.01 EN
(OF v1712)

The instructors



Cyprien Soulaine, PhD

Cyprien is Research Associate in the Department of Energy Resources at Stanford's School of Earth Sciences, California. He has a PhD in fluid dynamics from the Institut de Mécanique des Fluides de Toulouse, France.

His expertise concerns the modeling of flow and transport in porous media at the pore-scale and its translation to larger scales. Cyprien used OpenFOAM® since 2009 for its own research both as an user and a developer using OpenFOAM® technology.

Cyprien has delivered training courses for OpenFOAM® to more than a hundred of students, researchers, engineers both in academia and industry.

Julien Maes, PhD

Julien is a Research Associate in the Institute of Petroleum Engineering of Heriot-Watt University, Scotland. Julien is a former Reservoir Engineer at Total. He holds a PhD in Petroleum Engineering from Imperial College London.

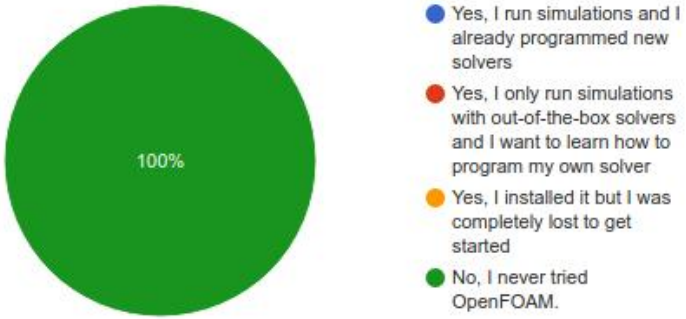
Julien is developing and applying direct numerical pore-scale simulation techniques with OpenFOAM® to model single- and multi-phase reactive flow with emphasis on wettability evolution in carbonate rocks.

cyprien.soulaine@gmail.com

Who are you?

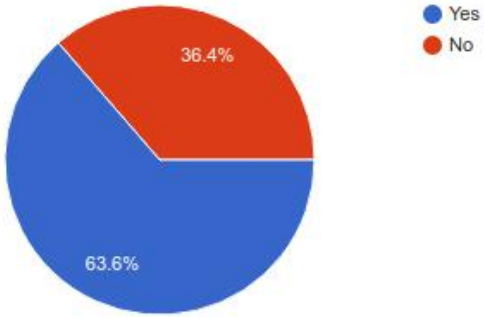
Have you tried OpenFOAM before?

11 responses



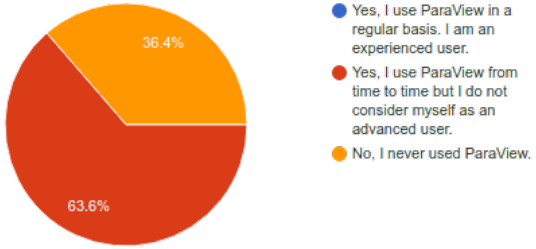
Do you have experience with Computational Fluid Dynamics? (Fluent, Comsol, OpenFOAM...)

11 responses



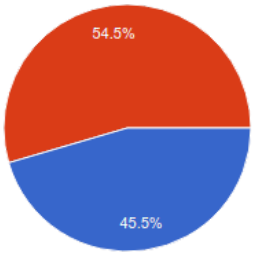
Do you use ParaView to visualize your simulations results?

11 responses



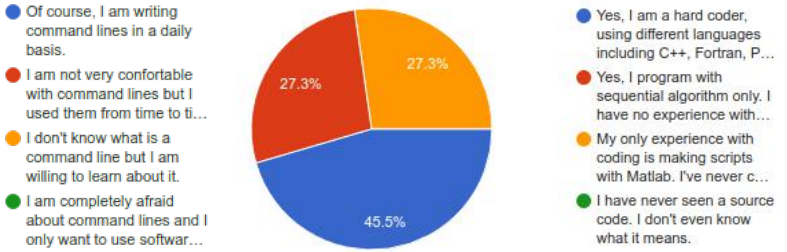
Are you familiar with command lines?

11 responses



Are you familiar with programming languages?

11 responses



What do you expect from the class?

Objectives

- Have an overview of the OpenFOAM® capabilities
- Be able to find help and documentation
- Know how to start and post-treat a simulation from existing tutorials
- Start your own simulation by modifying existing tutorials
- Understand what is behind the solvers to identify the most suitable solver for your specific problem
- Program your own solver by modifying an existing solver
- Join the OpenFOAM® adventure…

General introduction to OpenFOAM® technology

- *What is OpenFOAM®?*
- *Where can I find help and documentation?*

First simulations with OpenFOAM®

- *General structure of an OpenFOAM® case*
- *#1 – Heat diffusion*
- *#2 – Cavity*
- *#3 – Poiseuille flow*
- *#4 – Drainage experiment in a capillary tube*

How to mesh complex geometries with OpenFOAM®?

- *snappyHexMesh overview*
- *#5 – Mesh a pore-space with snappyHexMesh*
- *#6 – Scalar transport in porous media at the pore-scale*

Programming equations with OpenFOAM®

- *General structure of an application*
- *Basics of OpenFOAM programming*

Transport in porous media with OpenFOAM®

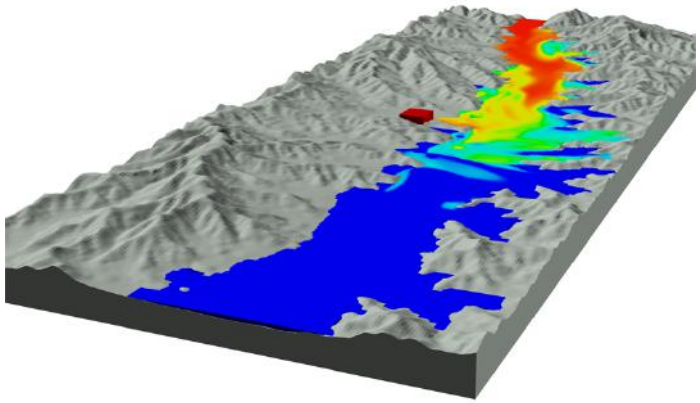
- *#8 – Create a « Darcy » solver*
- *#9 – Temperature in porous media*
- *#9 – Customize boundary conditions*
- *#10 – Two-equations model for heat transfer in porous media*

How to solve Navier-Stokes equation with OpenFOAM®?

From real life to numerical models

Real life

Numerical model



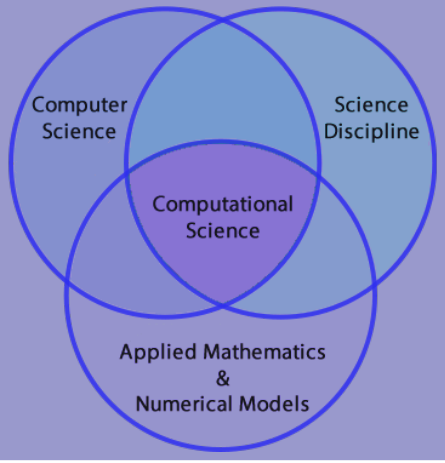
- Predictive models,
- Numerical experiments,
- Simulate complex problems where experiments are difficult or impossible (large scales, long period of time, nuclear reactors...)
- Perform optimization (shape, sensitivity analysis, process...)

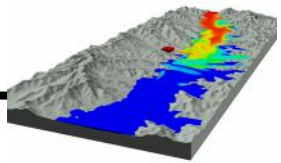
Numerical modeling

Observations*

Measurements*

Numerical modeling*

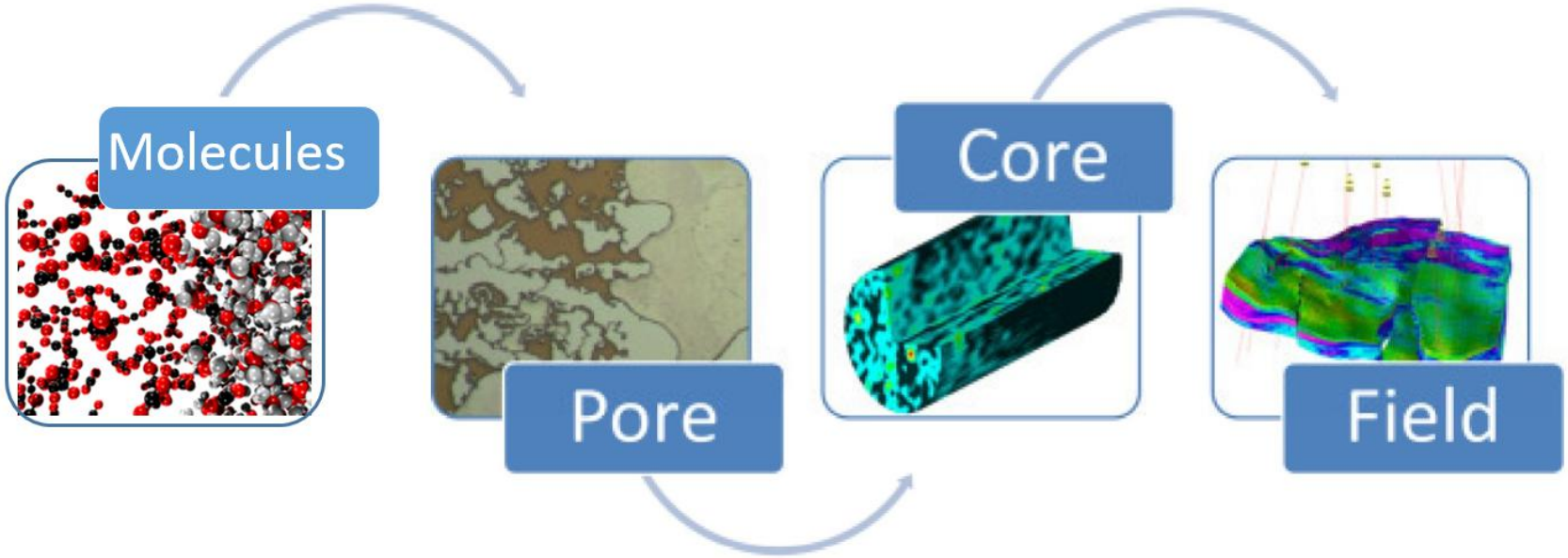




iterations

*error real life / digital life

Multi-scale modeling



Statistical mechanics

Continuum mechanics

Navier-Stokes

Darcy

OpenFOAM

What is OpenFOAM® ?

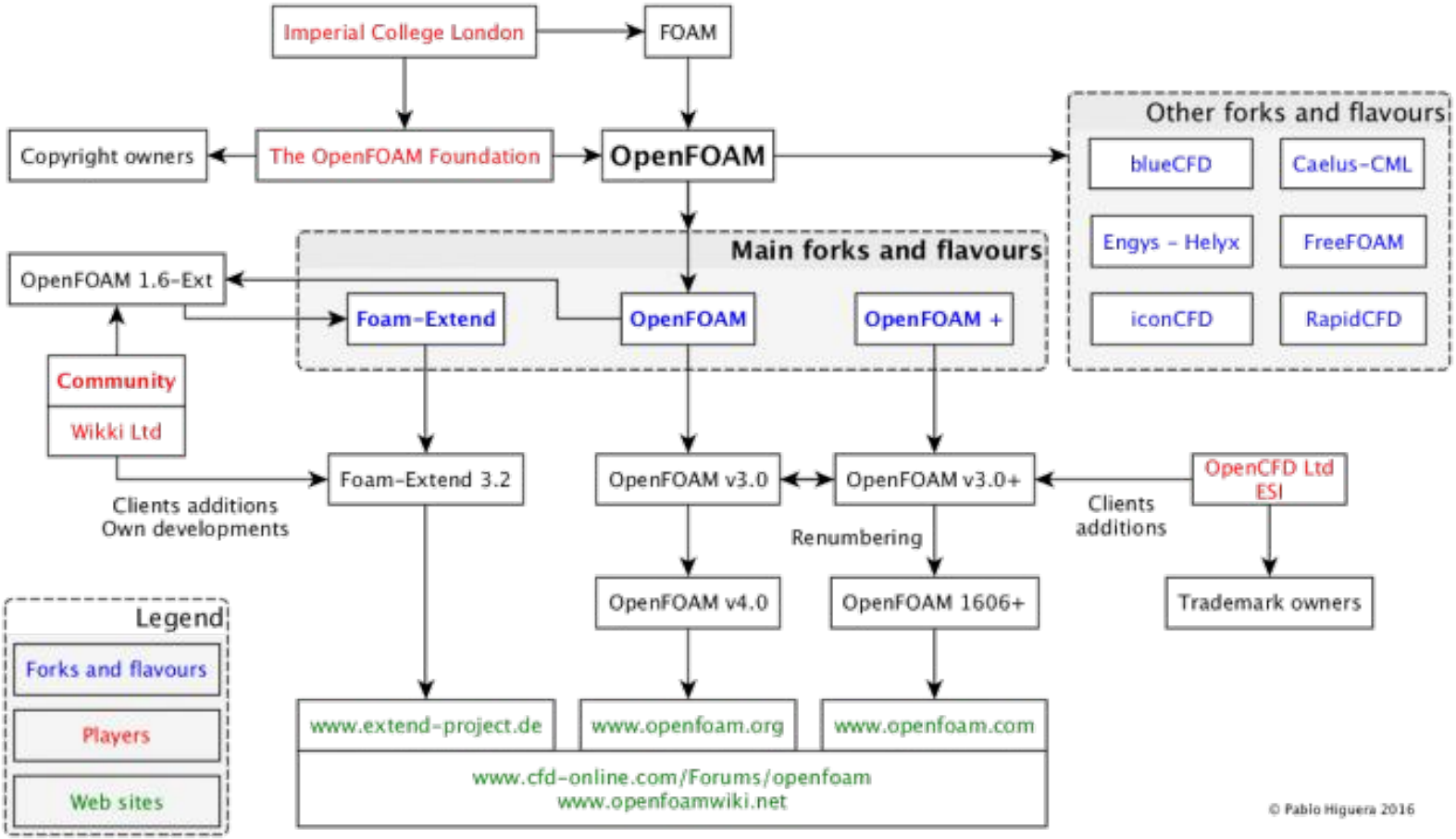
OpenFOAM®

= **Open** **F**ield **O**peration **A**nd **M**anipulation

- Solve the Partial Differential Equations using the finite volumes method
- Multiphysic simulation platform mainly devoted to fluid flow
- Manage 3D geometries by default
- Open-source software developed in C++ (object-oriented programming)
- Can be freely download at www.openfoam.org
- Designed as a toolbox easily customizable
- Parallel computation implemented at lowest level
- Cross-platform installation (Linux preferred)

- 1989 : First development at Imperial College London
- 1996 : First release of FOAM
- 2004 : OpenFOAM® released under GPL licence by OpenCFD Ltd.
- 2018 : OpenFOAM 5.0 ; OpenFOAM v1712 ; foam-extend4.0

The different OpenFOAM® players



The OpenFOAM® toolbox

OpenFOAM® = more than 200 programs (and not only 1 executable)

Pre-processing:

- Meshing (*blockMesh*, *snappyHexMesh, foamyHexMesh*…)
- Mesh conversion (Ansys, Salomé, ideas, CFX, Star-CD, Gambit, Gmsh…)

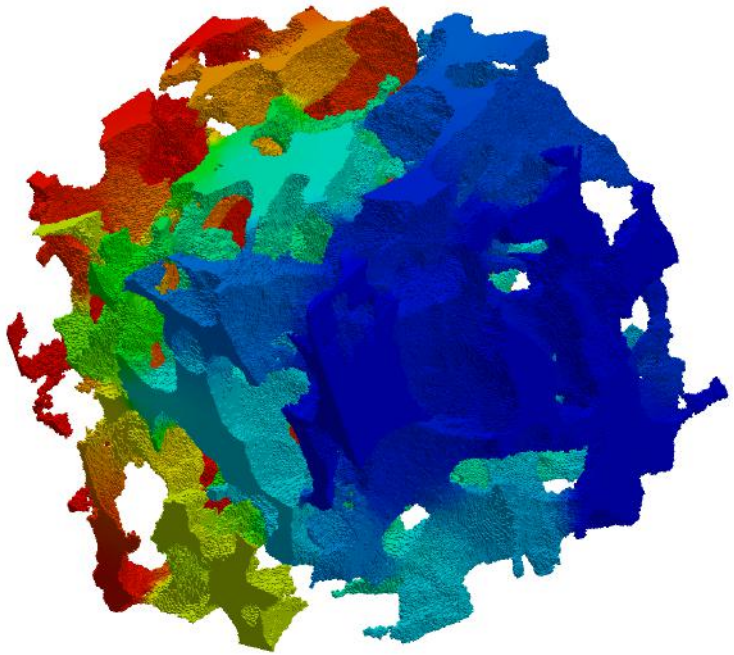
Solvers:

- incompressible / compressible flow
- multiphase flow (VOF, Euler-Euler...)
- combustion, electro-magnetism, solid mechanics
- heat transfer
- several turbulence approach (DNS, RANS, LES)
- etc…

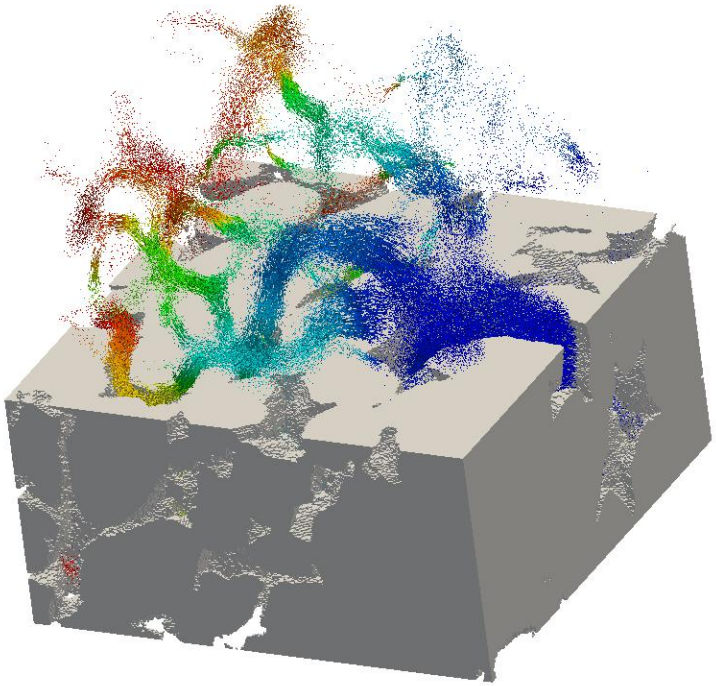
post-processing:

- Distributed with ParaView (and the famous *paraFoam*)
- Exportation to other post-treatment softwares (Fluent, Fieldview, EnSight, Tecplot…)
- «postProcess» utility for 1D or 2D sampling (export to gnuplot, Grace/xmgr et jPlot)

Example: Digital Rock Physics



$$K_{ij} = \mu_\beta \langle v_{\beta,i} \rangle \left(\frac{\triangle P}{L} \right)^{-1} \quad i = x, y, z.$$



- Digital rock obtained from microtomography imaging,
- Grid the void space,
- Solve steady-state Stokes equations,
- Up to 350 million cells,
- Account for the effect of sub-voxel porosity[1]

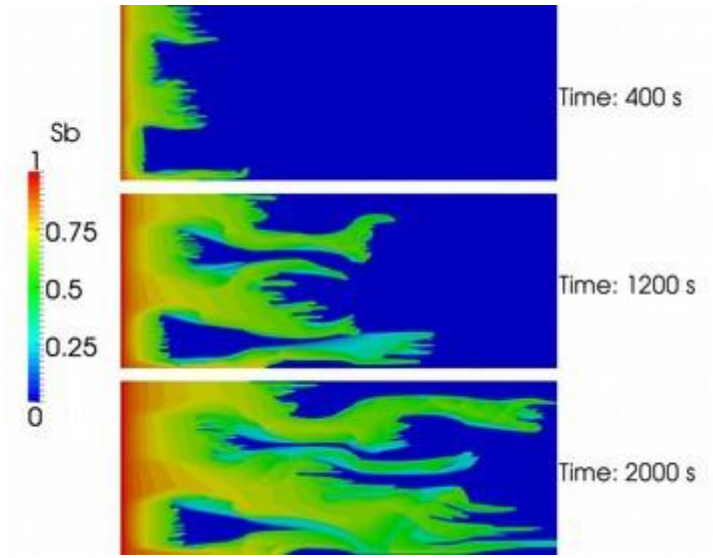
[1]Soulaine et al. *The impact of sub-resolution porosity of X-ray microtomography images on the permeability* Transport in Porous Media (2016)

Example: Two-phase flow in porous media

porousMultiphaseFoam toolbox[1,2]

https://github.com/phorgue/porousMultiphaseFoam.git

Generalized Darcy's law with capillarity and relative permeability (IMPES solver)[1]



Richards' equation[2]

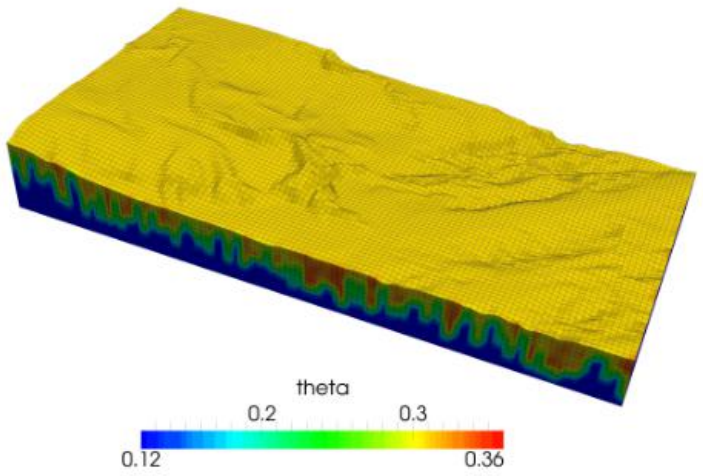


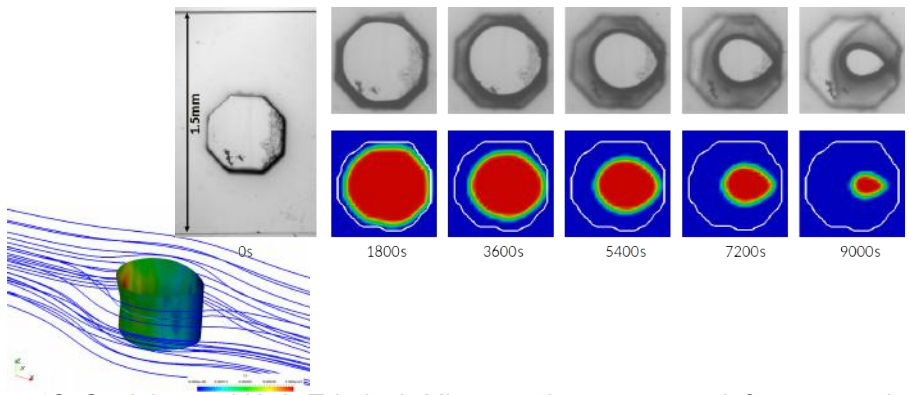
Figure 4: Saturation field at $t = 1000$ days

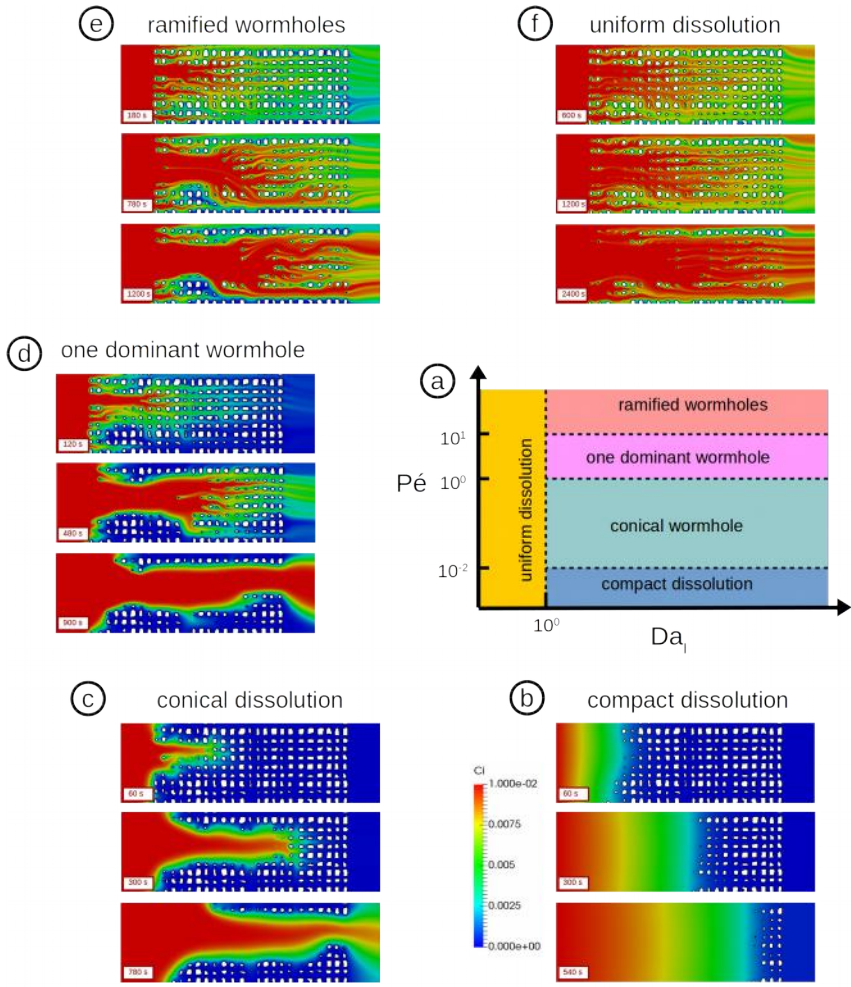
[1]P. Horgue et al., *An open-source toolbox for multiphase flow in porous media*, Computer Physics Communications 187 (2015)

[2]P. Horgue et al., *An extension of the open-source porousMultiphaseFoam toolbox dedicated to groundwater flows solving the Richards' equation*

Example: Mineral dissolution at the pore-scale

- A micro-continuum approach is proposed to simulate the dissolution of solid minerals at the pore-scale. The approach employ a the Darcy-Brinkman-Stokes[1] formulation and locally averaged conservation laws combined with immersed boundary conditions for the chemical reaction at the solid surface[2].
- The simulation framework is validated using an experimental microfluidic device to image the dissolution of a single calcite crystal. The evolution of the calcite crystal during the acidizing process is analyzed and related to flow conditions, i.e., Péclet and Damköhler numbers.
- Macroscopic laws for the dissolution rate are proposed by upscaling the pore-scale simulations.
- Finally, the emergence of wormholes during the injection of acid in a two-dimensional domain of calcite grains is discussed based on pore-scale simulations.



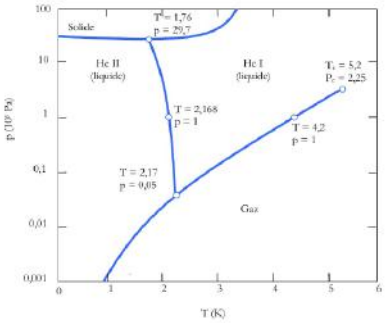


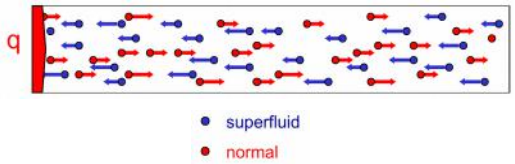
[1]C. Soulaine and H. A. Tchelepi, *Micro-continuum approach for pore-scale simulation of subsurface processes,* Transport in Porous Media (2016)

[2]C. Soulaine et al., *Mineral dissolution and wormholing from a pore-scale perspective,* Journal of Fluid Mechanics 827 (2017)

Example: Superfluid helium in porous media

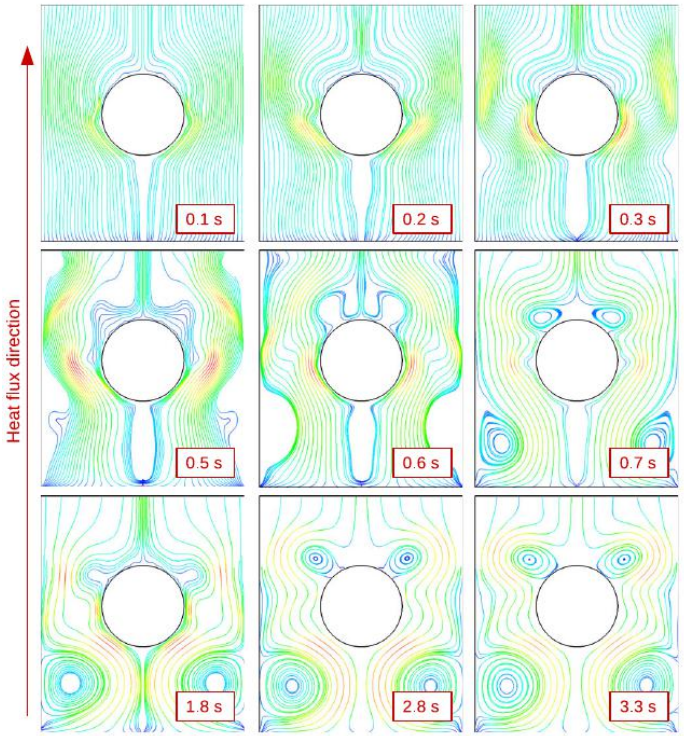
- Below 2.17 K, helium becomes superfluid. It can be thought as two inter-penetrating fluids that are fully miscible and have temperature dependent densities,

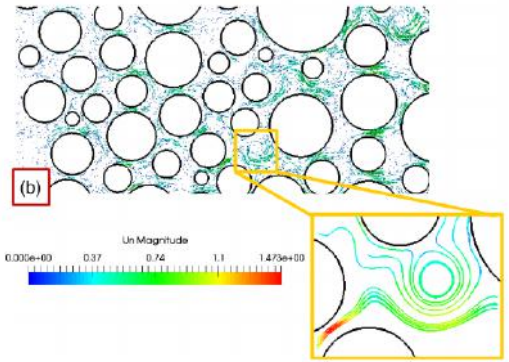




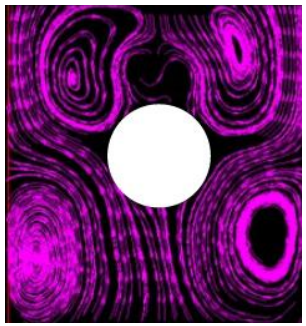
- Development and validation of a solver to simulate superfluid helium flow with the Landau's two-fluid model coupled with the Gorter-Mellink mutual friction forces[1],

- Simulation of thermal counterflow of He-II around cylinders[2]. The model captures the four eddies both up- and downstream of the obstacle observed experimentally with PIV[3].





PIV results[3]



[1]C. Soulaine et al., *A PISO-like algorithm to simulate superfluid helium flow with the two-fluid model*, Computer Physics Communications (2015)

[2]C. Soulaine et al., *Numerical Investigation of Thermal Counterflow of He-II Past Cylinders*, Physical Review Letters (2017)

[3]T. Zhang T and SW Van Sciver, *Large scale turbulent flow around a cylinder in counterflow superfluid ^{4}He (He II)*, Nature Physics (2005)

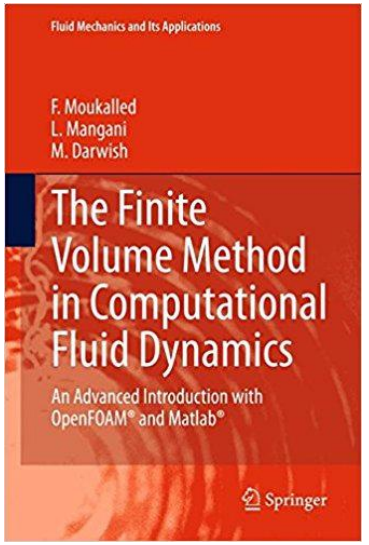
Why should I use OpenFOAM®?

- Completely free (No limitations due to licenses),
- Direct access to source code (not a black-box),
- An additional tool for code-to-code benchmarks,
- Regular updates (every 6 months),
- A lot of out-of-the-box solvers and their tutorials,
- Ease to program partial differential equations,
- A reactive and important community (online forum, conference, summer schools…),
- …..

- Need some time to learn,
- Lack of documentation..
- There is no official GUI,
- Unix command lines and C++ programing,
- Too many forks…

Where can I find help and documentation ?

- 2 official guides provided by the OpenFOAM Foundation *(« user guide »* and *« programmer guide »*) (Most of the time, this documentation is not enough...)
- CFD-direct : https://cfd.direct/openfoam/documentation/
- Several reference thesis (Hrvoje Jasak 1996, Henrik Rusche 2001, …)
- A tutorial per solver. Most of the time, it has a value of test-cases.
- Direct access to source-code (however, there is few comments in the code)
- Paying for technical support.



An active community !

- A discussion forum (www.cfd-online.com/Forums/openfoam/)
- A community-driven wiki (openfoamwiki.net)
- An annual Workshop (13th edition in 2018) (www.openfoamworkshop.org)
- FOAM-U : Association des utilisateurs francophones d'OpenFOAM (www.foam-u.fr)
- Chalmer CFD course with open-source software (http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/)
- A lot of tutorials, reports, scientific papers, presentations made by the community (search on Google)

General workflow of numerical modeling

Observations*

Measurements*

Geometry*

Mathematical modeling*

Boundary conditions*

Initial conditions*

Parametrization*

Gridding*

Operator's discretization*

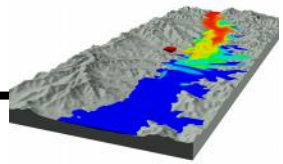
Linear solver*

Time-stepping*

Visualization

Post-treatment

Upscaling*



iterations

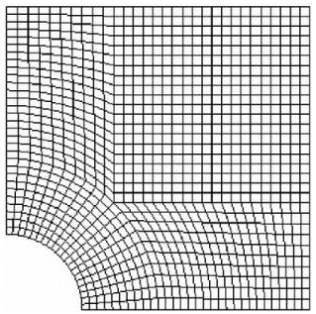
*error real life / digital life

How to draw and grid a geometry?

OpenFOAM®

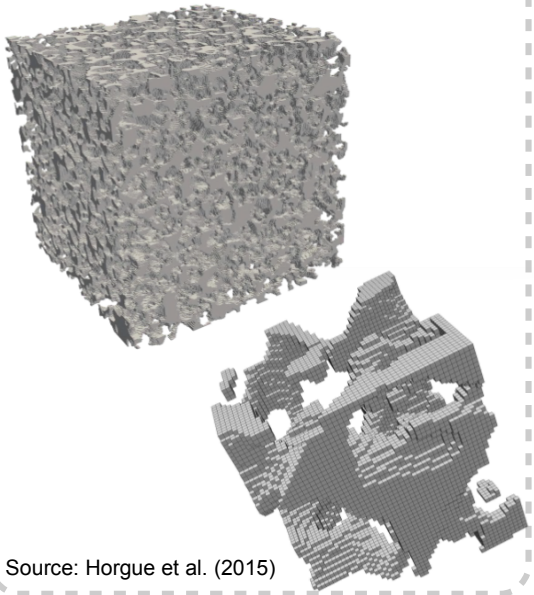
blockMesh

OpenFOAM®'s gridder for simple geometries



snappyHexMesh
foamyHexMesh

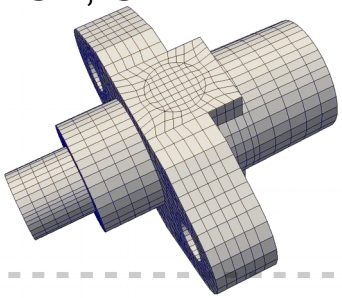
OpenFOAM®'s automatic gridders (geometries from CAD or micro-CT)



Source: Horgue et al. (2015)

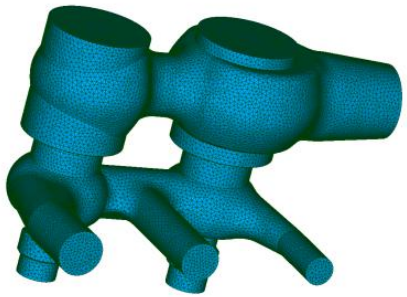
External gridder

Ansys, Fluent, Gambit, ideas, star-CD, CFX...



Salomé platform

Open-source gridder with GUI
http://www.salome-platform.org



Mathematical modeling

- Variables that vary in space and time: U(x,y,z,t) ; p(x,y,z,t) ; T(x,y,z,t)..
- Partial Differential Equation (PDE) = Equation governing the space and time evolution of U, p, T …
- Examples:
  - Navier-Stokes $\frac{\partial \rho \boldsymbol{v}}{\partial t} + \nabla . (\rho \boldsymbol{v} \boldsymbol{v}) = -\nabla p + \rho \boldsymbol{g} + \nabla . \left(\mu \left(\nabla \boldsymbol{v} + {}^{t} \nabla \boldsymbol{v} \right) \right)$
  - Heat equation $(\rho C_p) \frac{\partial T}{\partial t} = \nabla . (k \nabla T)$
  - Mass conservation $\frac{\partial \rho}{\partial t} + \nabla . (\rho \boldsymbol{v}) = 0$
  - Transport $\frac{\partial c}{\partial t} + \nabla . (\boldsymbol{v} c) = \nabla . (D \nabla c)$
- Boundary and initial conditions,
- Differential operator: laplacian, divergence, gradient, time derivative…
- Numerical approximation: Finite Difference Method (FDM), Finite Element Method (FEM), Finite Volume Method (FVM)...

Numerical modeling workflow

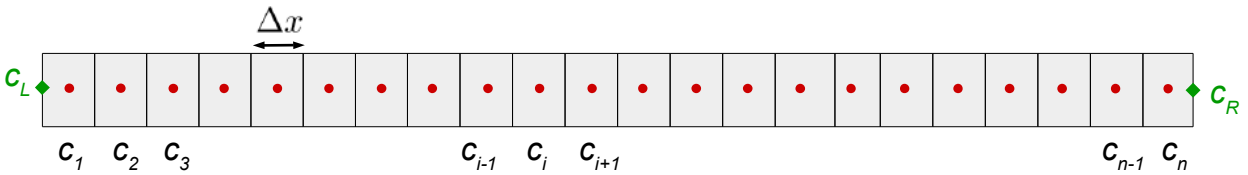
Space discretization

operator discretization

Build the matrix

Solve the linear algebra

$$c(x=0)=c_L \qquad \frac{\partial^2 c}{\partial x^2}=q \qquad c(x=H)=c_R$$



$$\frac{c_{i-1}-2c_i+c_{i+1}}{\Delta x^2}=q_i$$

$$\underbrace{\frac{1}{\Delta x^2}\begin{bmatrix} -3 & 1 & & & 0 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 0 & & & 1 & -3 \end{bmatrix}}_{\mathrm{A}} \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} q_1 - 2\frac{c_L}{\Delta x^2} \\ q_2 \\ \vdots \\ q_{n-1} \\ q_n - 2\frac{c_R}{\Delta x^2} \end{bmatrix}}_{\boldsymbol{y}}$$

$$\boldsymbol{x}=\mathrm{A}^{-1}\boldsymbol{y}$$

How to program equations in OpenFOAM®?

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot \phi \mathbf{U} - \nabla \cdot \mu \nabla \mathbf{U} = -\nabla p$$

```
solve
(
      fvm::ddt(rho,U)
    + fvm::div(phi,U)
    - fvm::laplacian(mu,U)
      ==
    - fvc::grad(p)
);
```

- The considered field *(U)* may be scalar, vector or tensor,
- Operators discretization does not need to be specified at the stage of the solver programming,
- The syntax is very closed to the mathematical formulation.

Some Unix commands

Navigation

pwd	Tells you the name of the working directory.
ls	List the files in the working directory.
cd	Change your working directory.

Visualization

cat	Outputs the contents of a specific file.

Manipulation of files

cp	To copy a file. Use the -r option to copy a directory.
mkdir	Create a directory.
rm	Delete a file. Use the -r option to remove a directory.
mv	Move or rename a file/folder.

I/O redirection

>	To redirect the output of an executable toward a file.
|	A pipeline to connect multiple commands together.
grep	A filter to output every line that contains a specified pattern of characters.

Some advices before we start...

```
$ cp  -r  ../test/test2 .
$ mkdir   -p   $FOAM_RUN
```

- These instructions mean that you write in a command-line interpreter the instruction “cp -r ../test/test2 .” followed by **[ENTER]** and then “mkdir -p $FOAM_RUN” followed by **[ENTER]**.

- $ means that you start from a new line after pressing **[ENTER]**. Don’t write $ at the beginning of a line.

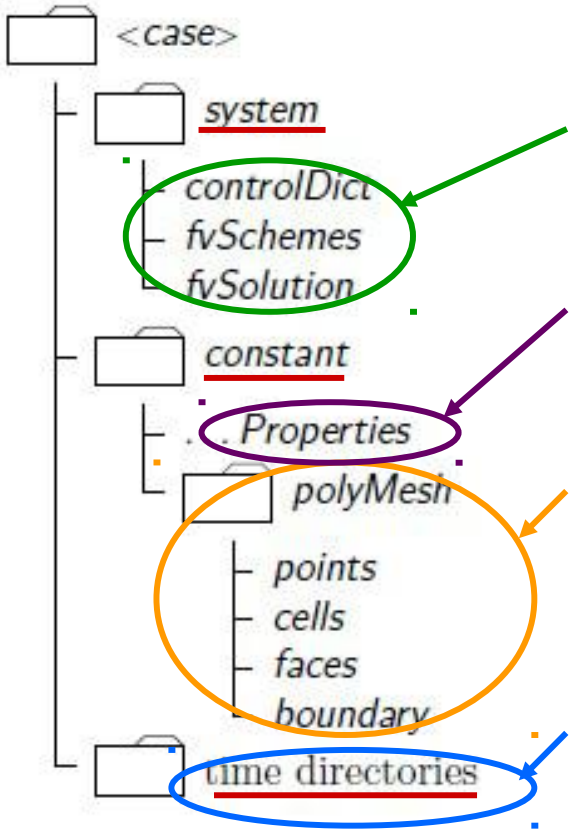
- The commands in the terminal are case sensistive. Mkdir is not the same than mkdir or MKDIR.

- When writing a command line, especially a directory, use the auto-completion **[TAB]** as much as you can. You save time and you avoid errors.

- You can replace “gedit” by any text editor.

General structure of an OpenFOAM® case

```
$ cd
$ mkdir   -p   $FOAM_RUN
```



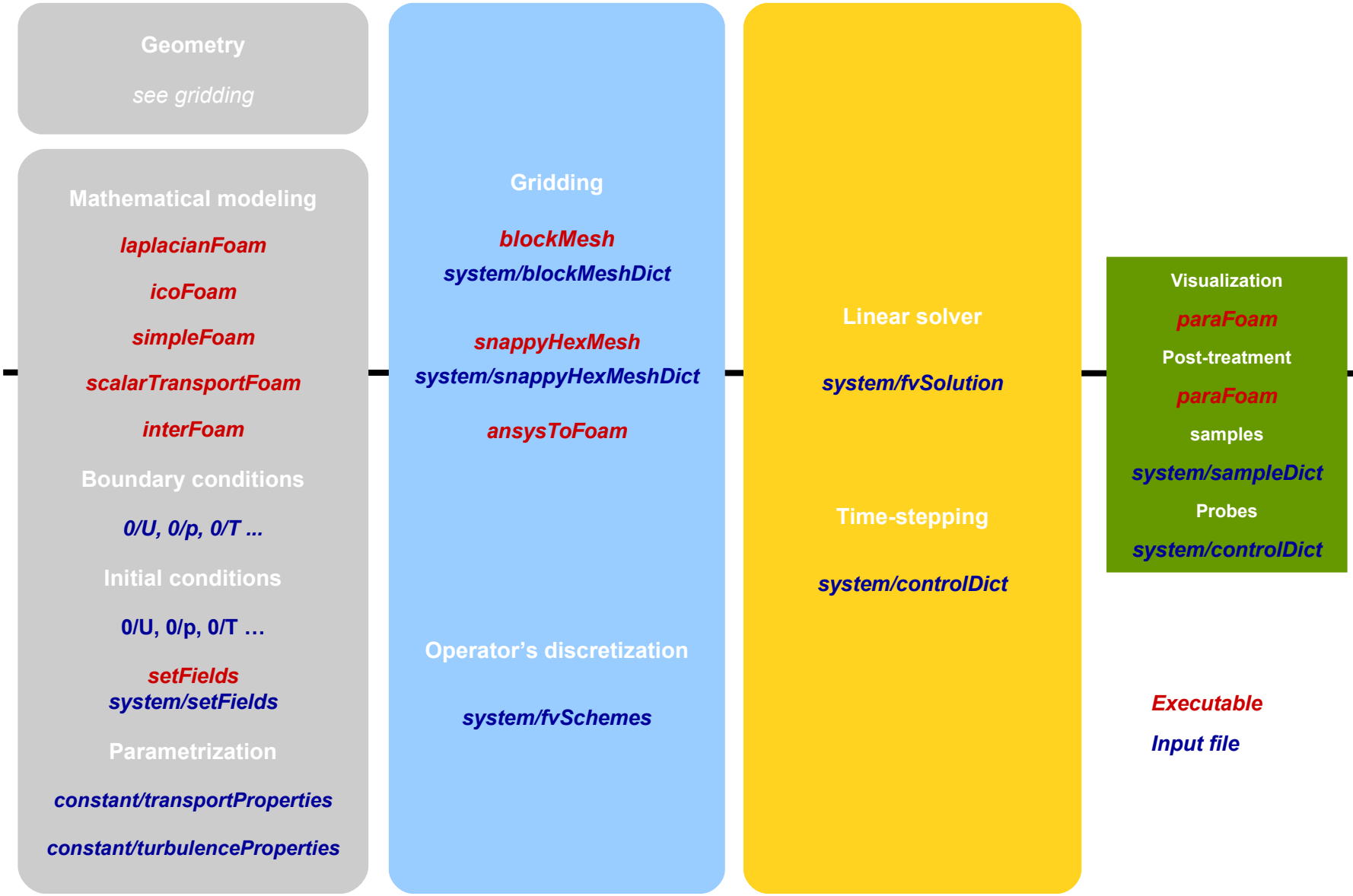
Simulation setup (choice of the linear solvers, of the discretization schemes, of the time steps, the output files...)

Everything regarding constant values (tranport properties, thermodynamic properties, turbulence model, etc...)

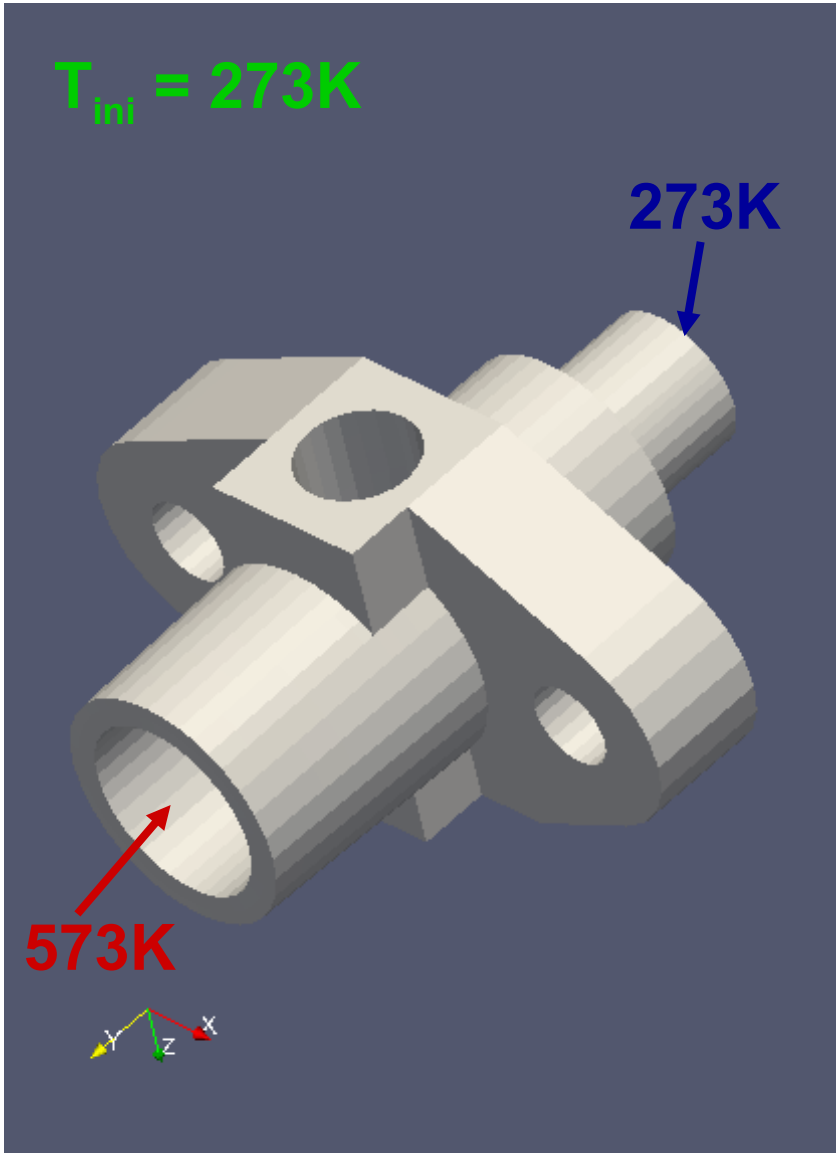
All the information related to the grid

One folder per time step. Each folder includes as many files as computed fields (T, U, p, Yi, k, Omega…)
Initial conditions are specified in the « 0 » directory.

Common programs and input files



#1 – Heat diffusion (1/5)



- Example from tutorials provided with OpenFOAM®
- Geometry and grid generated with Ansys

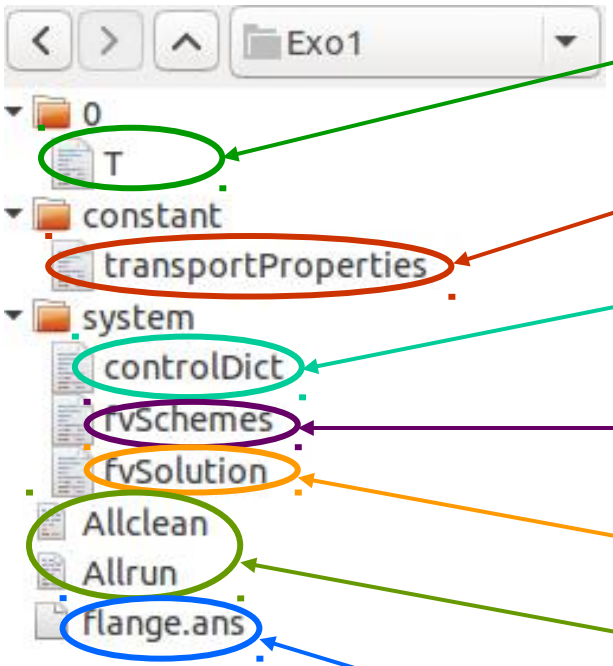
  Mesh conversion using the utility ansysToFoam
- Solution of the heat transfer equation

$$\frac{\partial T}{\partial t} = \nabla \cdot (D_T \nabla T)$$

Solver : *laplacianFoam*

#1 – Heat diffusion (2/5)

```
$ run
$ cp  –r  $FOAM_TUTORIALS/basic/laplacianFoam/flange/   Exo1
$ cd  Exo1
$ ls
```



Initial and boundary conditions for the temperature field *T*

Value of the diffusion coefficient (m^2/s)

Simulation parameters (time steps, output time...)

Discretization of the different operators (div, laplacian, ddt, grad…)

Set up of the linear solvers

Scripts to automatically start and clean the tutorial

Grid generated with Ansys

#1 – Heat diffusion (3/5)

- Convert the mesh from Ansys to OpenFOAM (with a scale factor)

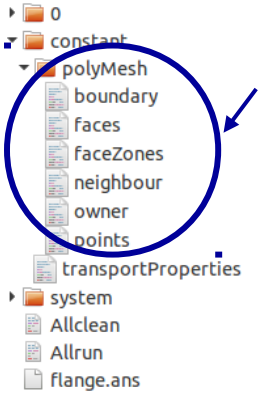
```
$ ansysToFoam   flange.ans   -scale   0.001
```

- Look at the change in the directory

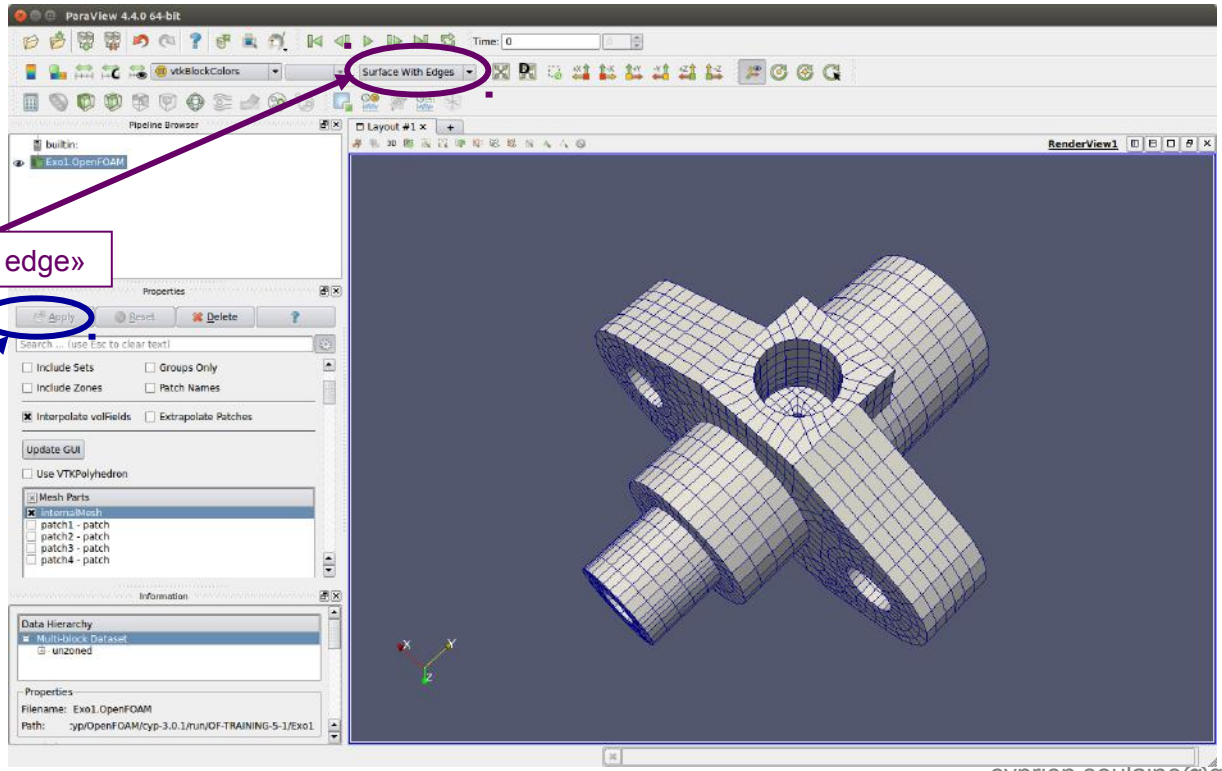
```
$ ls constant
$ ls constant/polyMesh
```

- Visualize the grid with ParaView

```
$ paraFoam
```



polyMesh contains all the information related to the grid: list of points, faces, neighbors, boundaries...



2 - Choose « surface with edge»

1 - « apply »

#1 – Heat diffusion (4a/5)

```
$ gedit 0/T
```

```
T ×
/*--------------------------------*- C++ -*----------------
| =========                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD
|  \\    /   O peration     | Version:  3.0.1
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|    \\/     M anipulation  |
\*---------------------------------------------------------
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 273;

boundaryField
{
    patch1
    {
        type            zeroGradient;
    }

    patch2
    {
        type            fixedValue;
        value           uniform 273;
    }

    patch3
    {
        type            zeroGradient;
    }

    patch4
    {
        type            fixedValue;
        value           uniform 573;
    }
}

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

Definition of initial and boundary conditions

Dimensions (units) of the field T

[kg m s K kgmol A cd]

Uniform initial temperature (T=273K) in the solid bulk

Boundary conditions for t=0s

Zero flux

Fixed value (T=273K)

Fixed value (T=573K)

#1 – Heat diffusion (4b/5)

$ gedit constant/transportProperties

transportProperties ×

```
/*--------------------------------*- C++ -*----------------
| =========                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD
|  \\    /   O peration     | Version:  3.0.1
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|    \\/     M anipulation  |
\*---------------------------------------------------------
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

DT              DT [0 2 -1 0 0 0 0] 4e-05;


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

The dimensions of the diffusivity DT are m^2/s

#1 – Heat diffusion (4c/5)

```
$ gedit system/controlDict
```

controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

application     laplacianFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         3;

deltaT          0.005;

writeControl    runTime;

writeInterval   0.1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;


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

#1 – Heat diffusion (5/5)

- Start the simulation

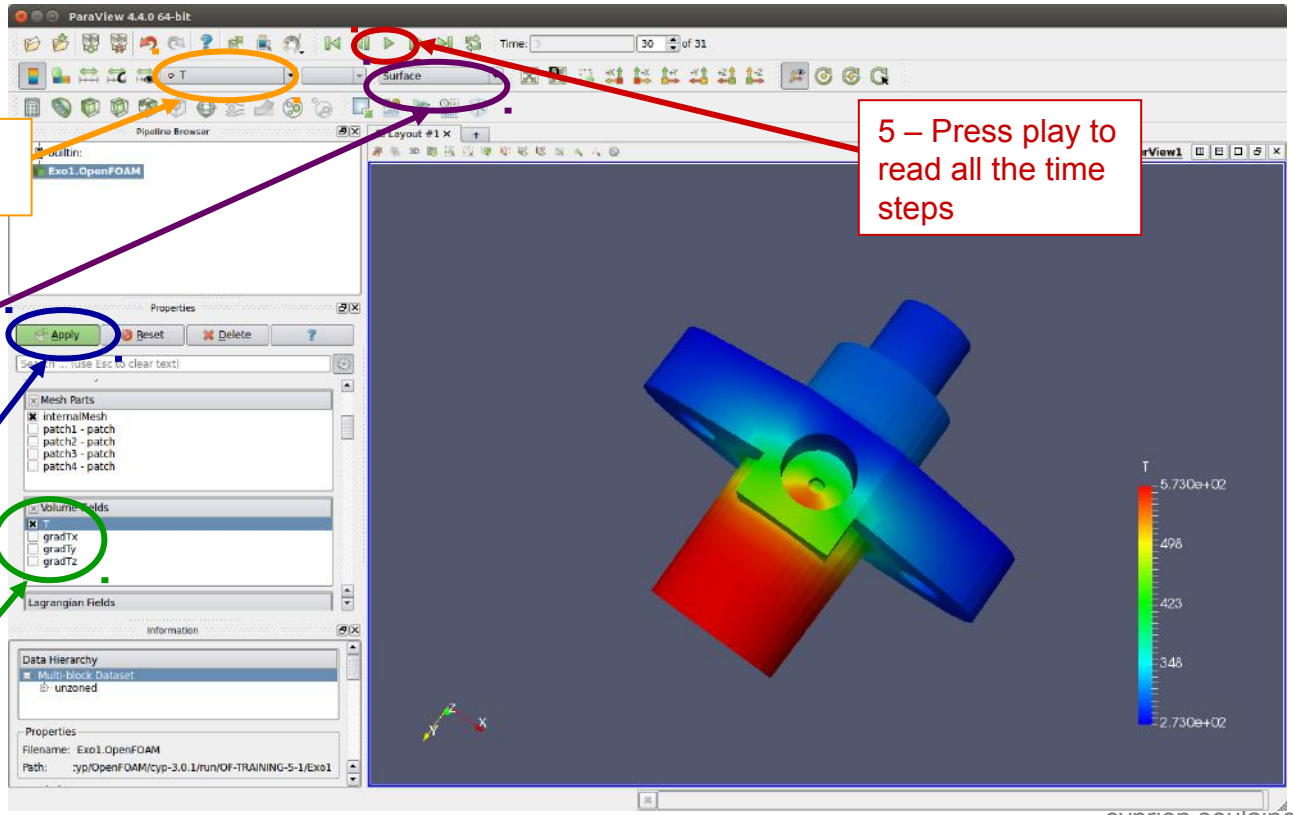
```
$ laplacianFoam
```

- Look at the change in the directories: new folders appeared, they corresponds to the time steps

```
$ ls
```

- View the results with ParaView

```
$ paraFoam
```

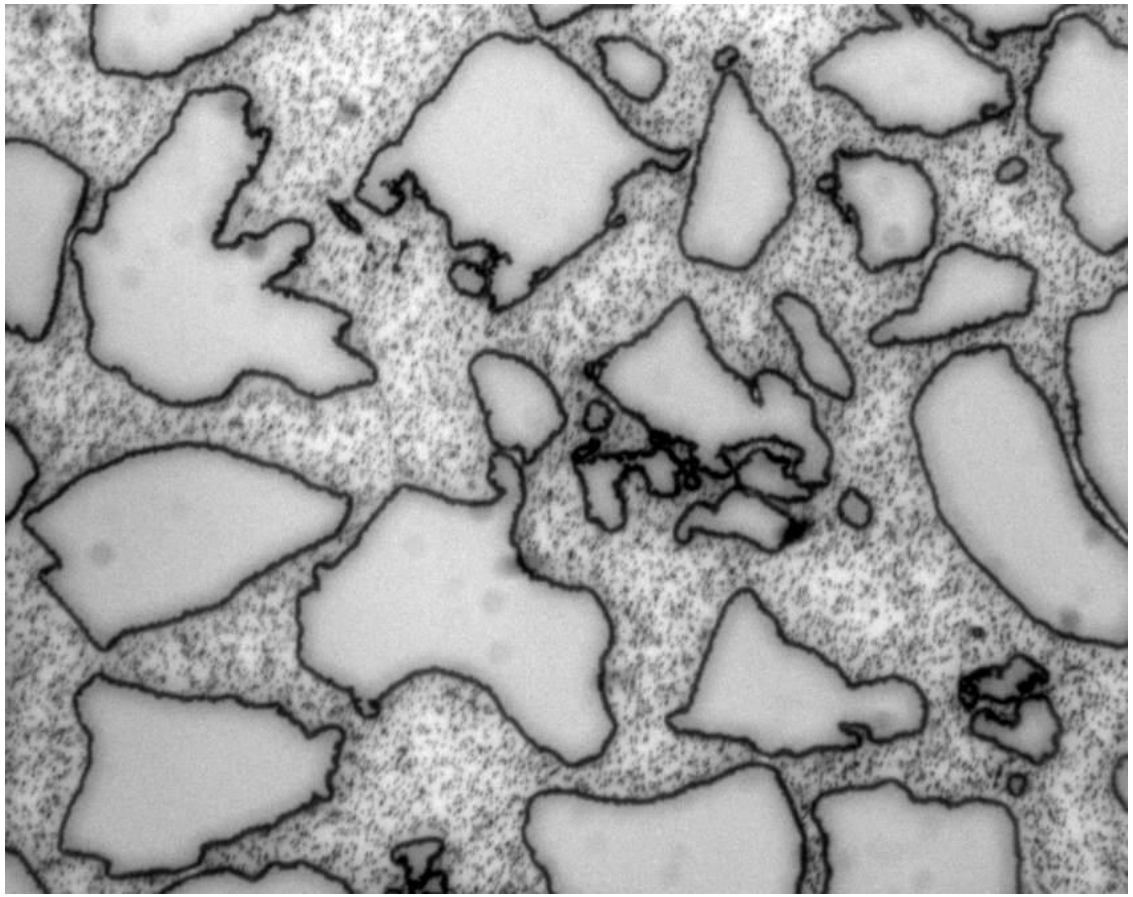


Exo1 bis :

DT = $1e^{-7}$ m²/s
Until steady state

Flow at the pore-scale

- Water seeded with micro-particles to enhance the flow visualization in the pore space (Sophie Roman)



- For a given location, the instantaneous velocity is always the same.

The Navier-Stokes equations

For an incompressible single-phase fluid, the flow motion equations read

Mass balance equation

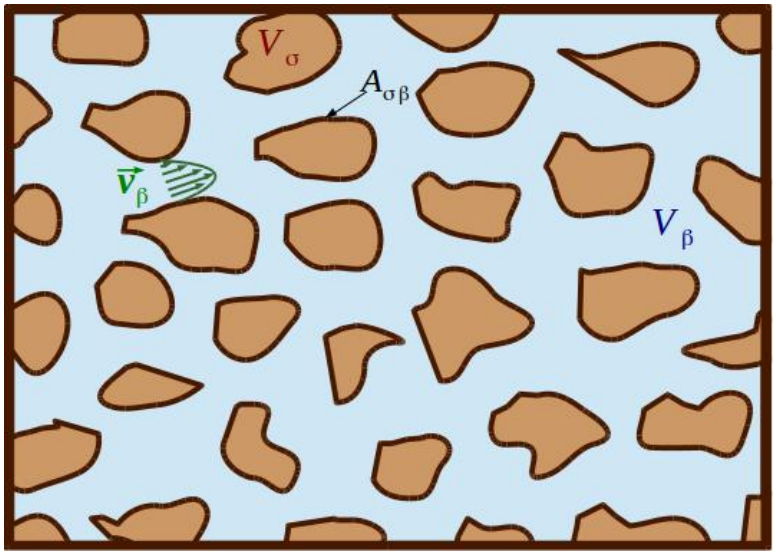
$$\nabla . \boldsymbol{v} = 0$$

Momentum balance equation

$$\underbrace{\frac{\partial \rho \boldsymbol{v}}{\partial t} + \nabla . (\rho \boldsymbol{v}\boldsymbol{v})}_{\text{inertia}} = \underbrace{-\nabla p}_{\text{pressure gradient}} + \underbrace{\rho \boldsymbol{g}}_{\text{gravity}} + \underbrace{\mu \nabla^2 \boldsymbol{v}}_{\text{viscous force}}$$

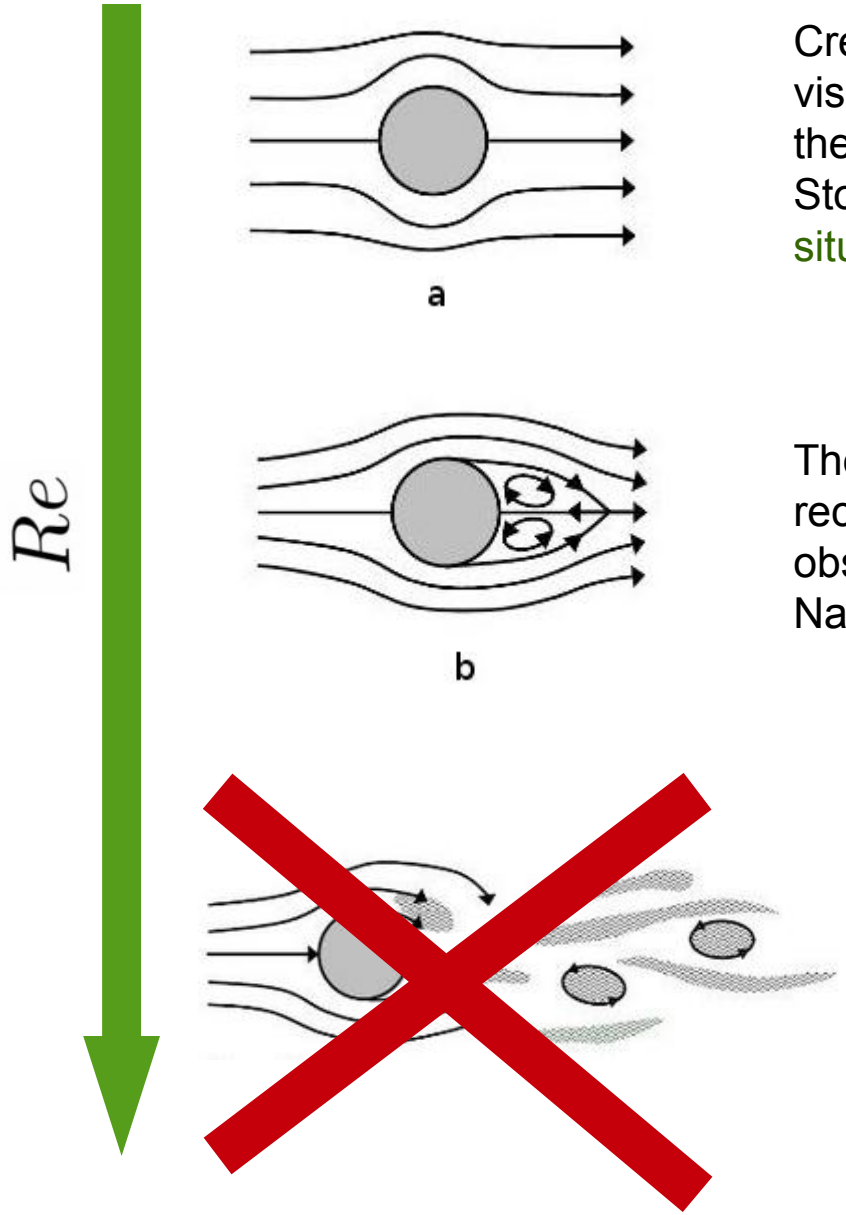
Non-slip condition at the solid boundary

$$\boldsymbol{v} = 0 \text{ at the solid walls}$$



- The Reynolds number, $Re = \dfrac{\rho U_0 L}{\mu}$, is a dimensionless number used to classify the flow regimes.
- If Re<1, the inertia effects are negligible in front of the viscous forces and Navier-Stokes becomes the Stokes equation. It is a particular case of Navier-Stokes, which means that all Navier-Stokes solvers are also valid for Stokes without any modification!
- The pressure-velocity coupling is handled numerically by different solution strategies.
- Cavity flow and Poiseuille flow are classic solutions of Navier-Stokes equations.

Different flow regimes

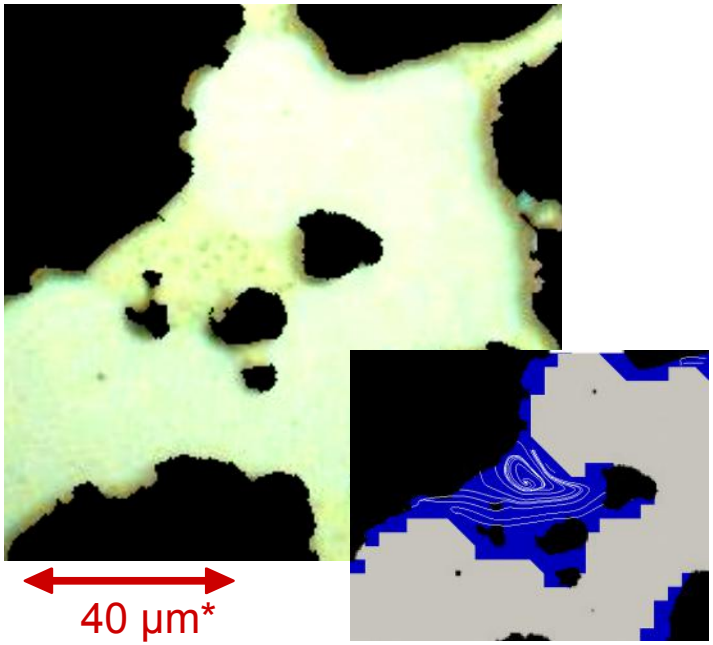


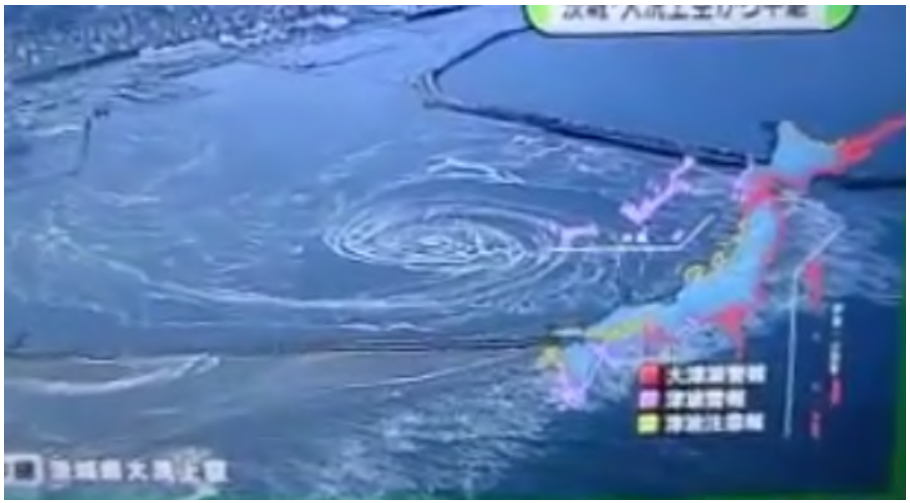
Creeping flow regime: the flow is governed by the viscous effects only and the streamlines embrace the solid structure. The flow is modeled by the Stokes equations. Most of the time, we are in this situation.

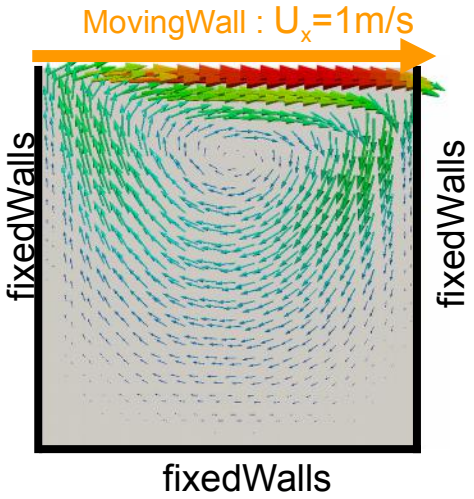
The inertia effects distort the streamlines and flow recirculations are generated downstream the obstacles. The flow is modeled by the laminar Navier-Stokes equations.

At very high Reynolds numbers, turbulence effects emerge. The flow is modeled by turbulent Navier-Stokes equations (RANS, LES...)

The lid-driven cavity flow



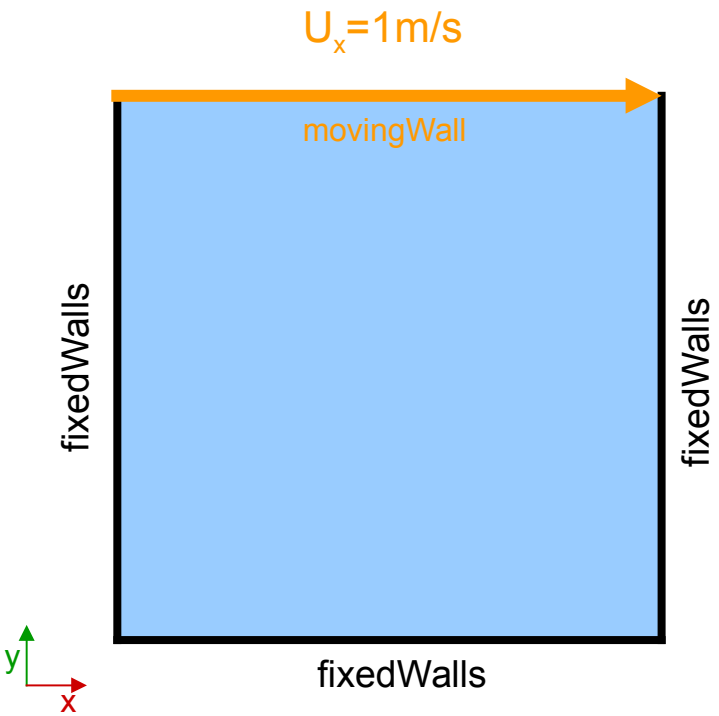




Same phenomenon modeled with the same equations... even though there are several orders of magnitude difference!!

* Roman et al. *Particle velocimetry analysis of immiscible two-phase flow in micromodels*, Advances in Water Resources *2016*

#2 – Cavity (1/6)



- Tutorial detailed in the official User Guide
- Design and meshing of the geometry with the utility *blockMesh*
- Solution of the laminar incompressible Navier-Stokes equations with the *icoFoam* solver

$$\nabla \cdot \mathbf{U} = 0$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) = \nabla \cdot (\nu \nabla \mathbf{U}) - \nabla p$$

- Post-processing with ParaView

```
$ run
$ cp      –r     $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity    Exo2
$ cd    Exo2
$ ls
```

#2 – Cavity (2/6)

blockMesh = pre-processing tool to design and mesh simple geometries

```
scale   0.1;

vertices
(
    (0 0 0)      //0
    (1 0 0)      //1
    (1 1 0)      //2
    (0 1 0)      //3
    (0 0 0.1)    //4
    (1 0 0.1)    //5
    (1 1 0.1)    //6
    (0 1 0.1)    //7
);

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

edges
(
);

boundary
(
    movingWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    fixedWalls
    {
        type wall;
        faces
        (
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```

Vertices definition

Definition of the hexahedral block. Pay attention to the numerotation order

Definition of the grid (Regular mesh 20x20, only one cell in the z direction because the simulation will be 2D)

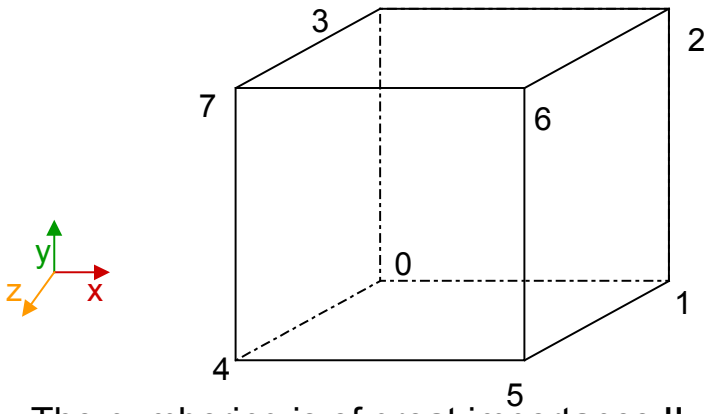
Definition of boundary of the domain to apply boundary conditions.

Faces normal to Oz are «empty» to specify that the simulation is 2D

- Geometry and grid are defined in the file *blockMeshDict*

  `$ gedit system/blockMeshDict`

- The geometry is always defined in 3D since OpenFOAM® only considers 3D geometry

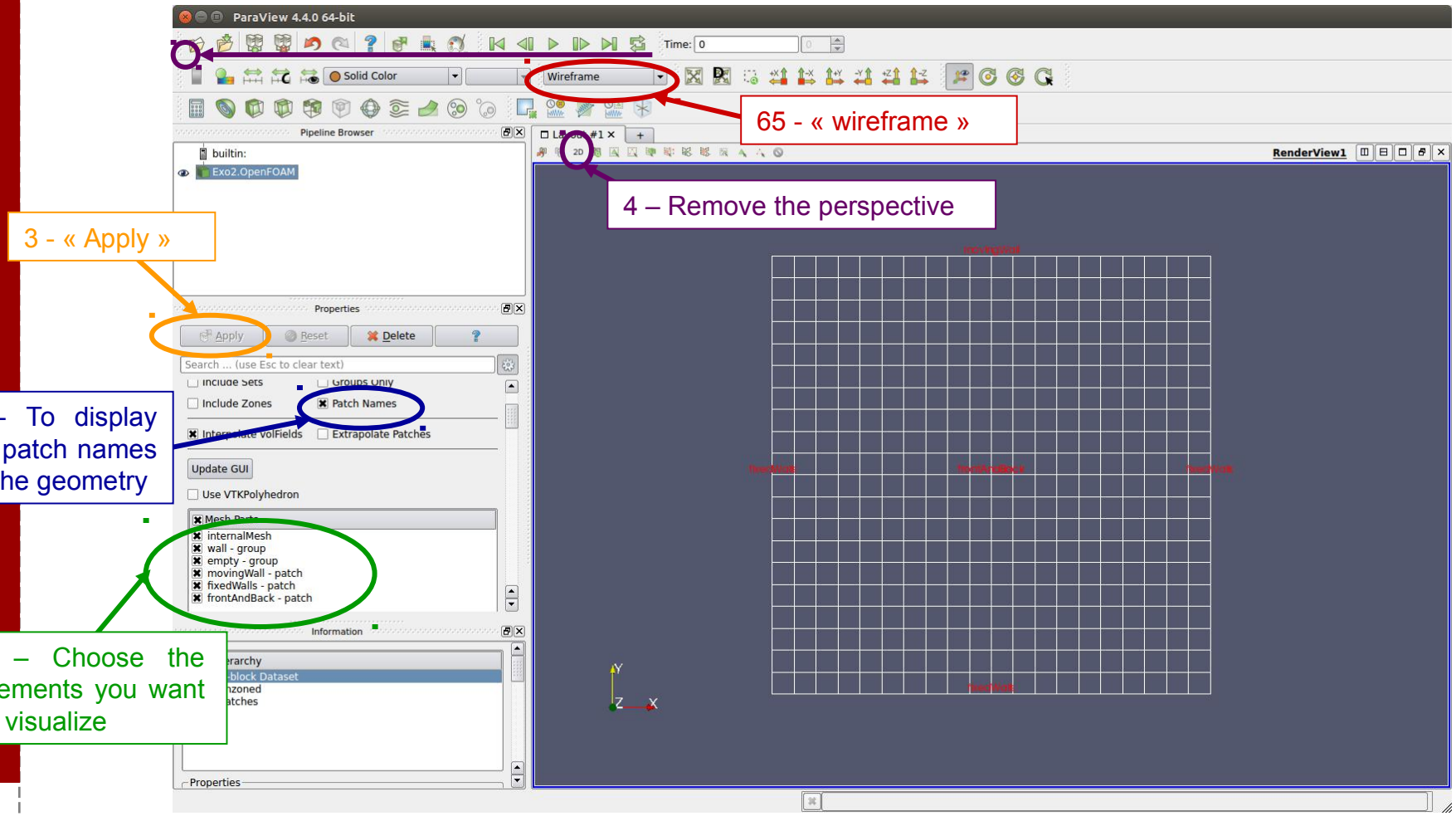


- The numbering is of great importance !!

- Boundaries may be of different types:
  - -patch *(generic type)*
  - -wall *(for solid wall condition, useful for turbulence)*
  - -cyclic *(for cyclic simulations)*
  - -symmetryPlane *(for symmetry plane)*
  - -empty *(to specify that the simulation will be 2D or 1D)*
  - -wedge *(for axi-symmetric simulations)*
  - -processor *(for parallel computation, automatically defined during the decomposition domain process)*

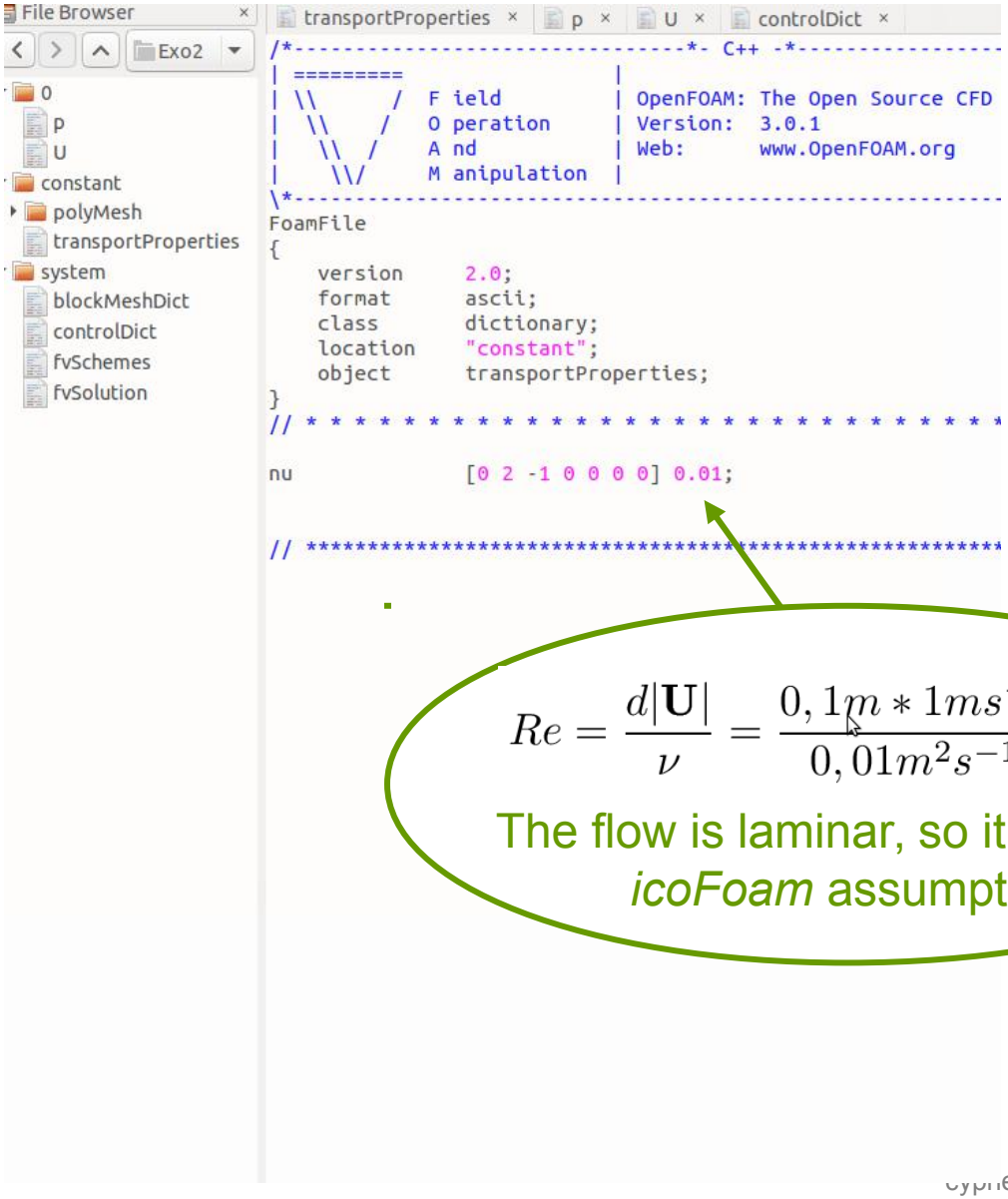
#2 – Cavity (3/6)

- Generate the grid mesh: `$ blockMesh` → Creation of *constant/polyMesh* and the files related to the grid.
- Check the mesh quality: `$ checkMesh`
- View the mesh : `$ paraFoam`



#2 – Cavity (4a/6)

$ gedit constant/transportProperties

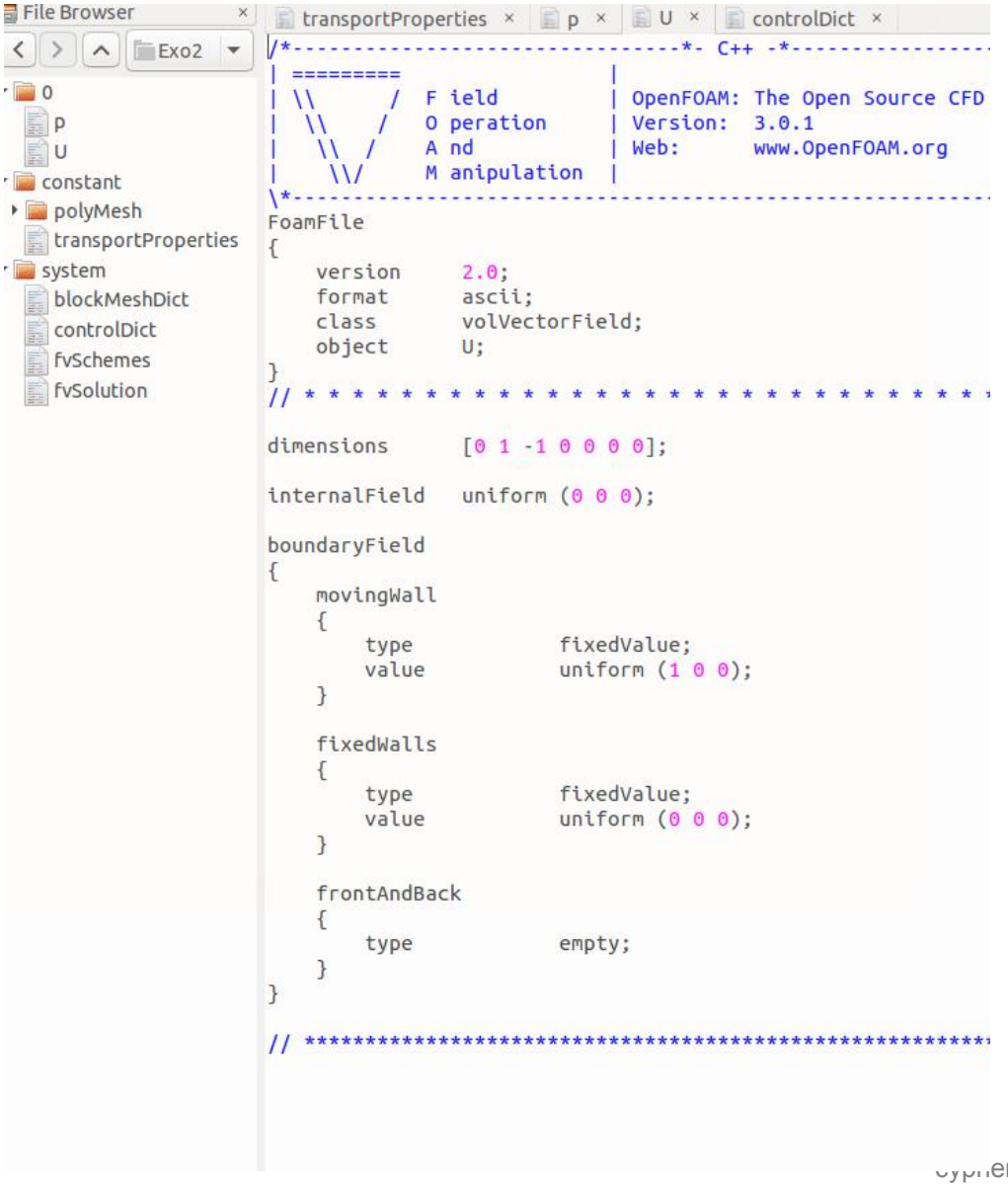


$$Re = \frac{d|\mathbf{U}|}{\nu} = \frac{0,1m * 1ms^{-1}}{0,01m^2s^{-1}} = 10$$

The flow is laminar, so it fullfills the *icoFoam* assumptions

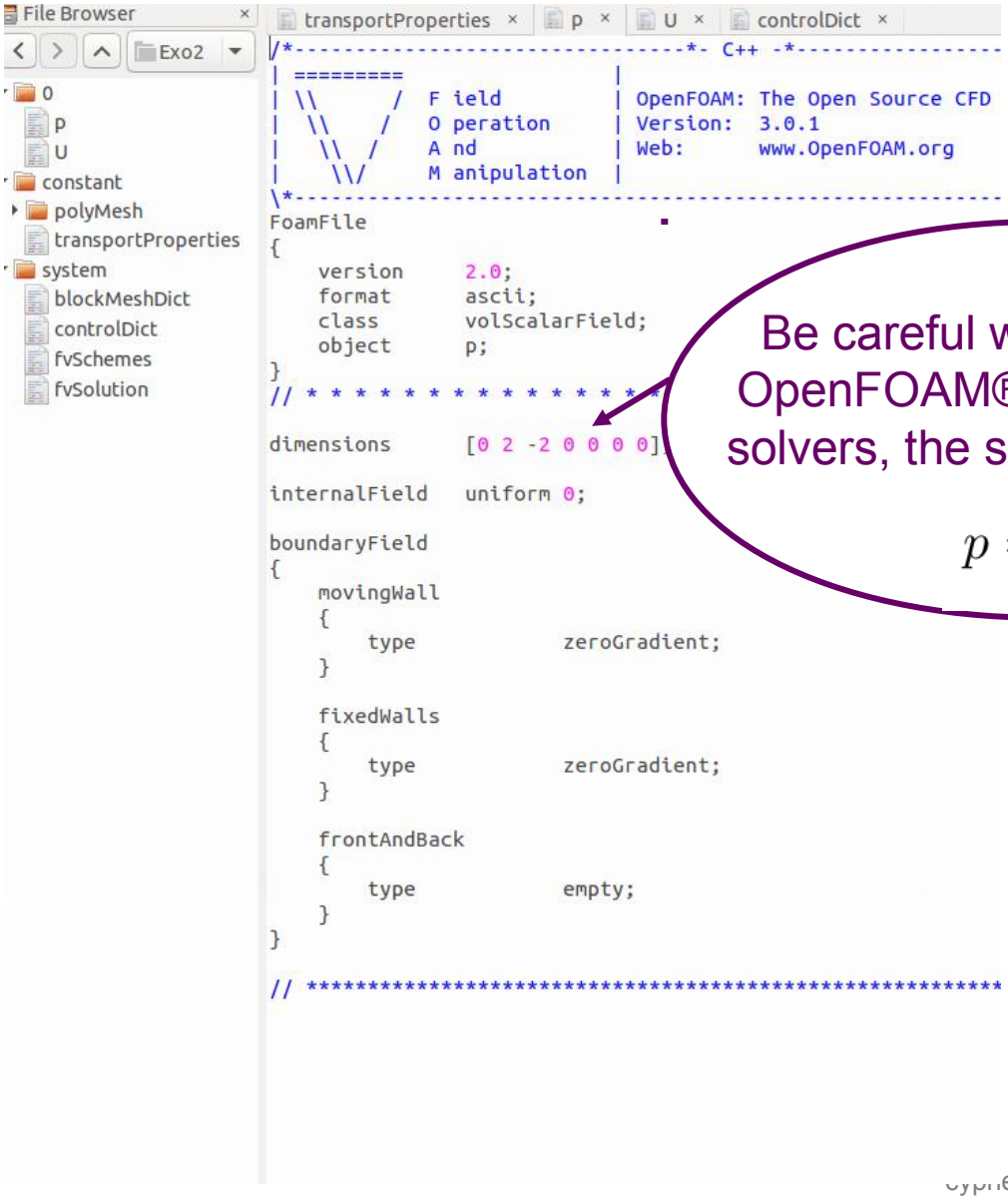
#2 – Cavity (4b/6)

$ gedit 0/U



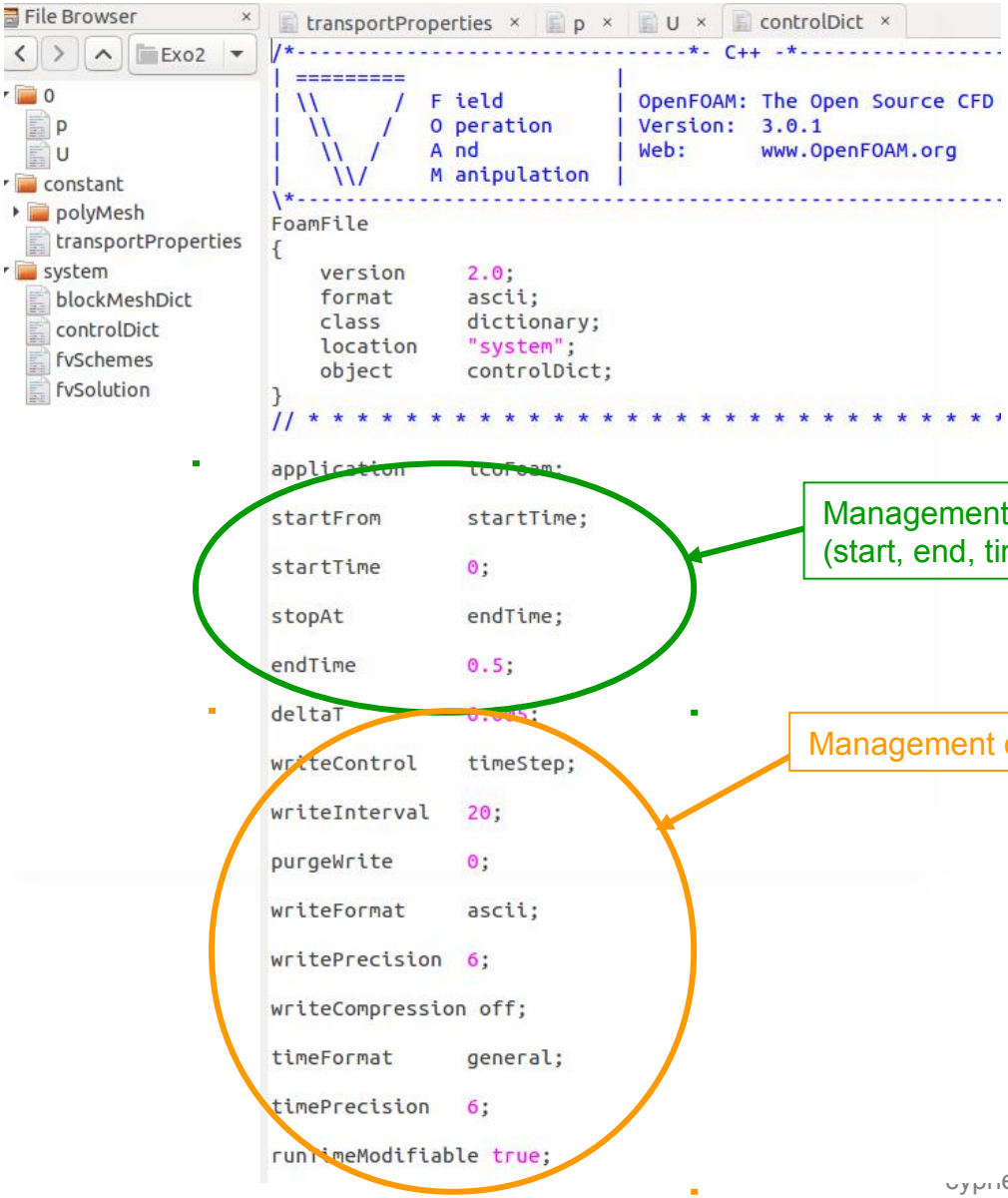
#2 – Cavity (4c/6)

$ gedit 0/p



#2 – Cavity (4d/6)

$ gedit system/controlDict



#2 – Cavity (5/6)

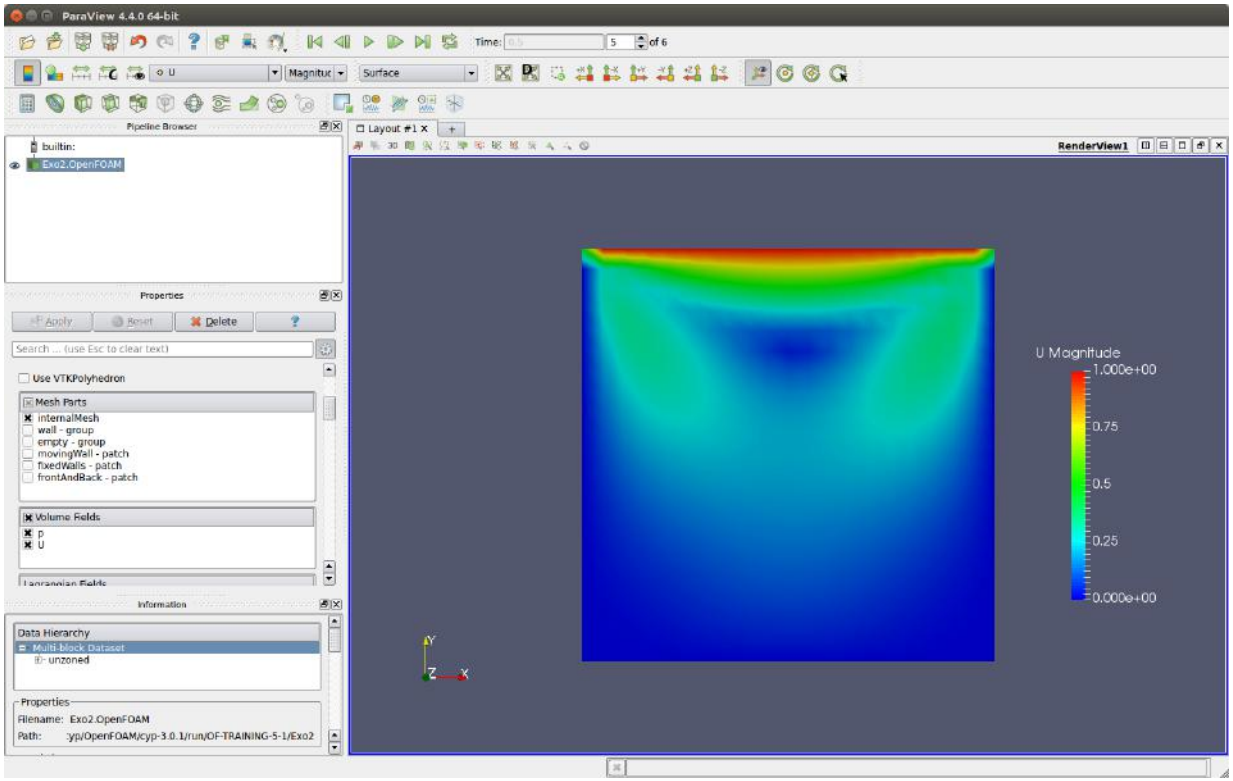
- Start the simulation :

```
$ icoFoam
```

```
DICPCG:  Solving for p, Initial residual = 8.33045e-07, Final residual = 8.33045e-07, No Iterations 0
time step continuity errors : sum local = 8.59385e-09, global = 5.07889e-19, cumulative = -1.54203e-18
ExecutionTime = 0.12 s  ClockTime = 0 s

Time = 0.5

Courant Number mean: 0.222158 max: 0.852134
smoothSolver:  Solving for Ux, Initial residual = 2.32737e-07, Final residual = 2.32737e-07, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 5.07002e-07, Final residual = 5.07002e-07, No Iterations 0
DICPCG:  Solving for p, Initial residual = 1.0281e-06, Final residual = 2.77237e-07, No Iterations 1
time step continuity errors : sum local = 4.0374e-09, global = -9.0204e-19, cumulative = -2.44407e-18
DICPCG:  Solving for p, Initial residual = 5.31987e-07, Final residual = 5.31987e-07, No Iterations 0
time step continuity errors : sum local = 6.12557e-09, global = -3.93738e-20, cumulative = -2.48344e-18
ExecutionTime = 0.12 s  ClockTime = 0 s

End
```

- Post-processing with ParaView :

```
$ paraFoam
```

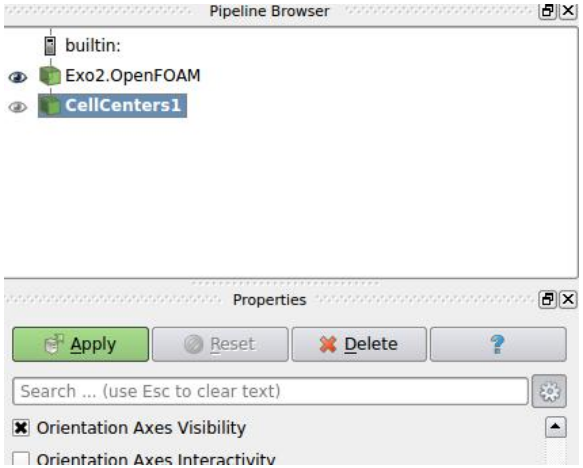


#2 – Cavity (6a/6)

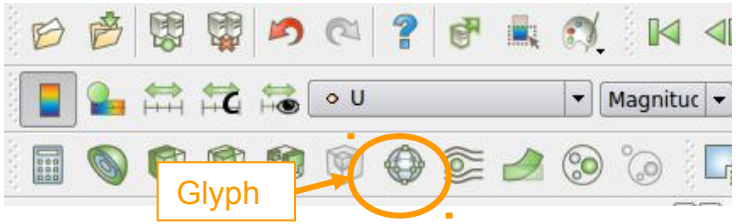
To view the velocity vectors

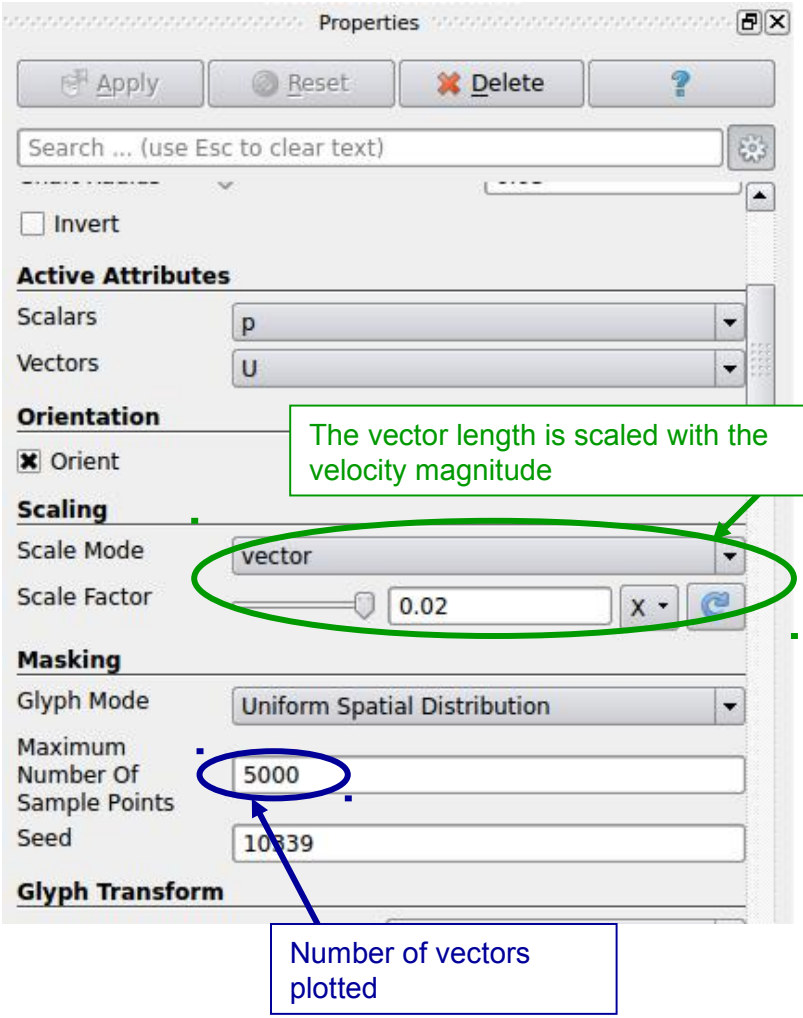
- We specify that the values are plotted at the cell centers with the filter *Cell Centers* (by default, ParaView plots the vector at the face centers whereas OpenFOAM® calculates at the cell centers)

filters>alphabetical>Cell Centers>Apply



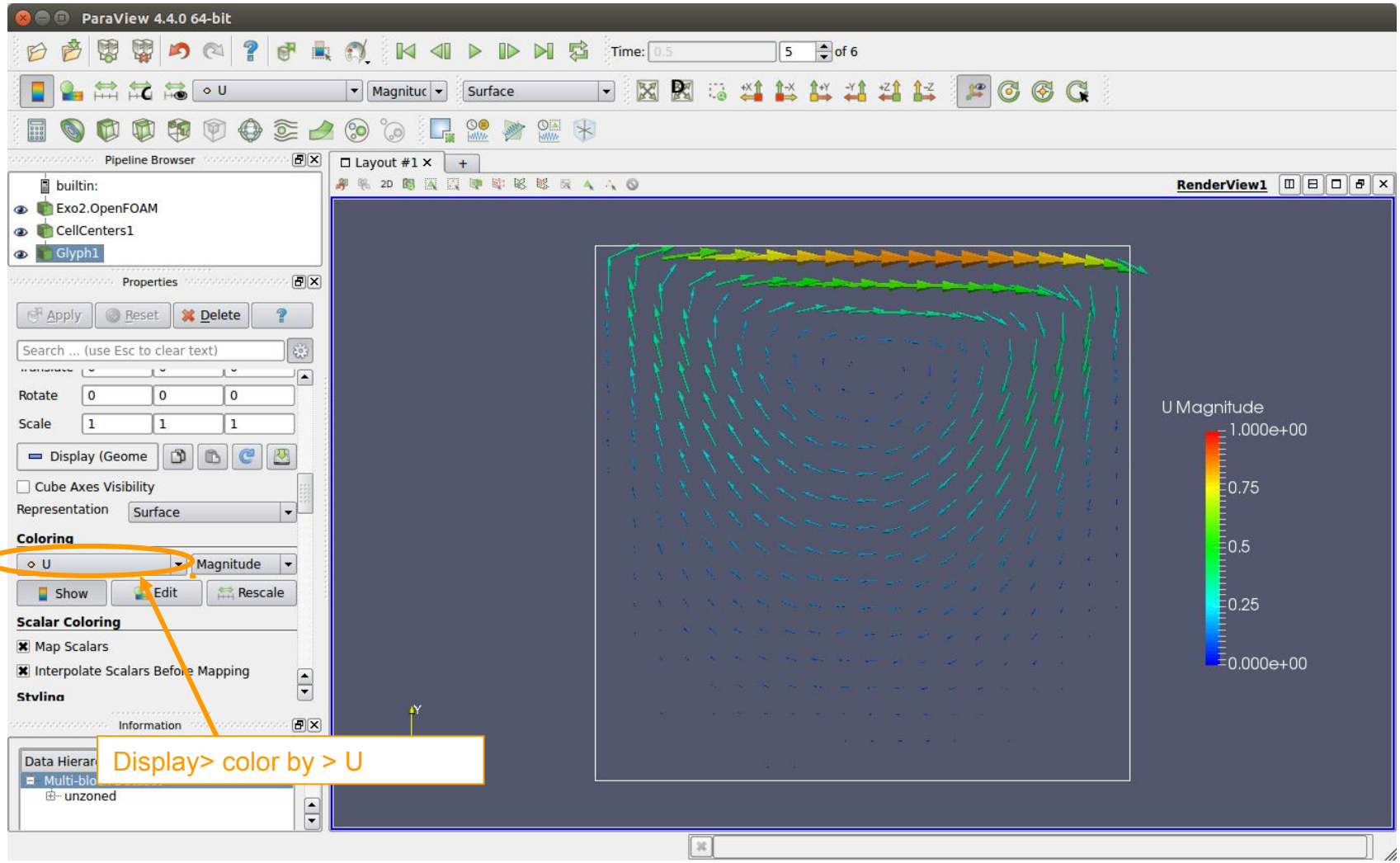
- We then apply the Glyph filter to plot the velocity vector:



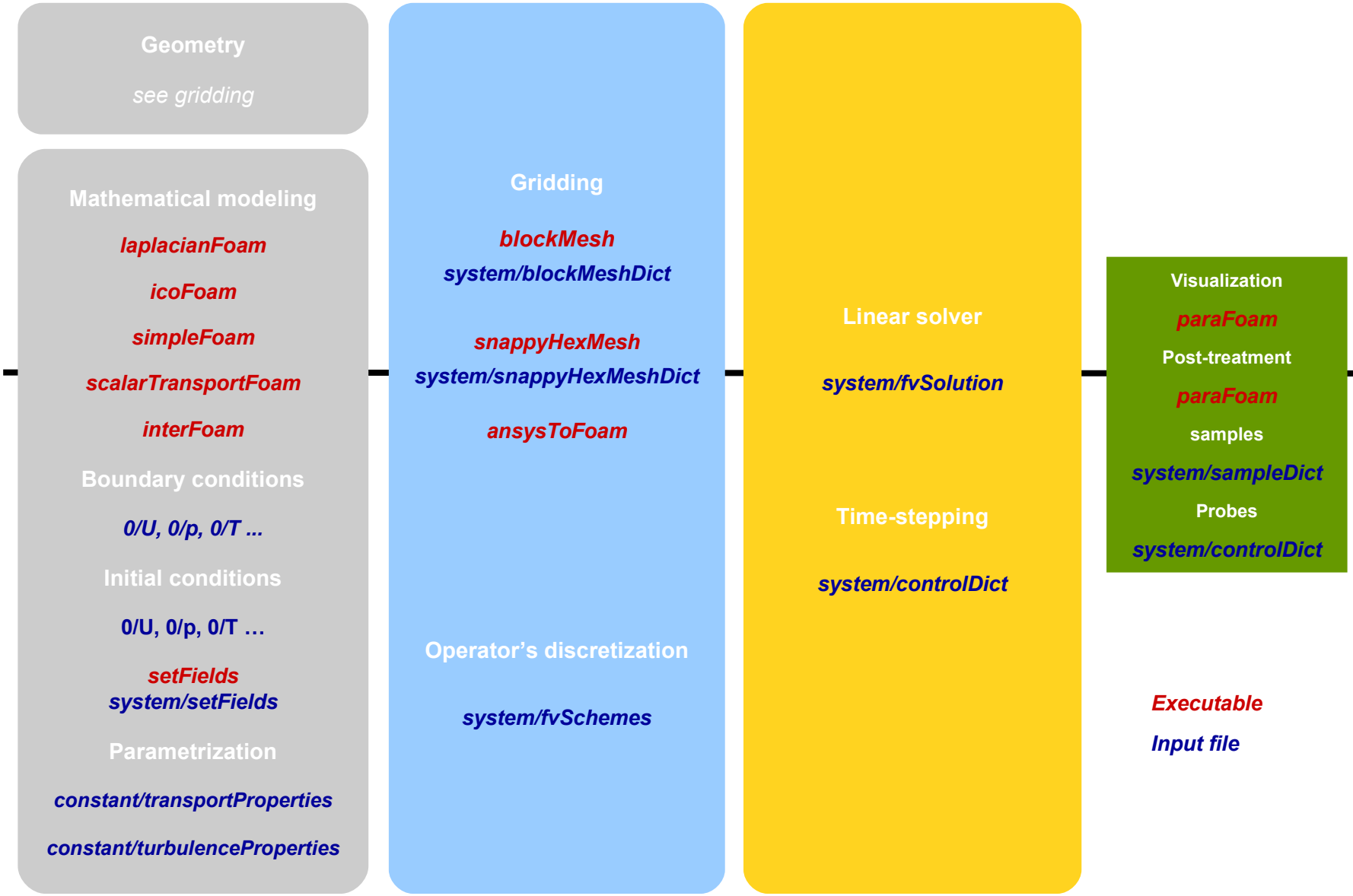


#2 – Cavity (6b/6)

To view the velocity vectors



Common programs and input files



PISO or SIMPLE?

Navier-Stokes (or Stokes) equations are solved in a sequential manner using predictor-corrector projection algorithms. In OpenFOAM®, you have to choose between PISO and SIMPLE:

algorithm	transient	Steady -state	comments	OpenFOAM solver
PISO* PIMPLE	YES	YES	Can be used to find the stationary solution by solving all the time steps	**icoFoam**, pisoFoam, pimpleFoam, interFoam, twoPhaseEulerFoam, rhoPimpleFoam...
SIMPLE**	NO	YES	Faster than PISO to converge to the steady state	**simpleFoam**, rhoSimpleFoam...

- PISO is not unconditionally stable and the time step is limited by a CFL condition.
- SIMPLE is an iterative procedure that under-relaxes the pressure field and velocity matrix at each iteration.
- To allow larger time steps, a combination of both algorithm is sometime proposed (PIMPLE).
- Multiphase Navier-Stokes equations are solved in the framework of the PISO solution procedure.
- Stokes momentum equation does not involve transient terms and can be solved with SIMPLE.

* Issa. *Solution of the Implicitly Discretised Fluid Flow Equations by Operator-Splitting.* Journal of Computational Physics, 62:40-65, 1985.

** Patankar. *Numerical Heat Transfer And Fluid Flow,* Taylor & Francis, 1980

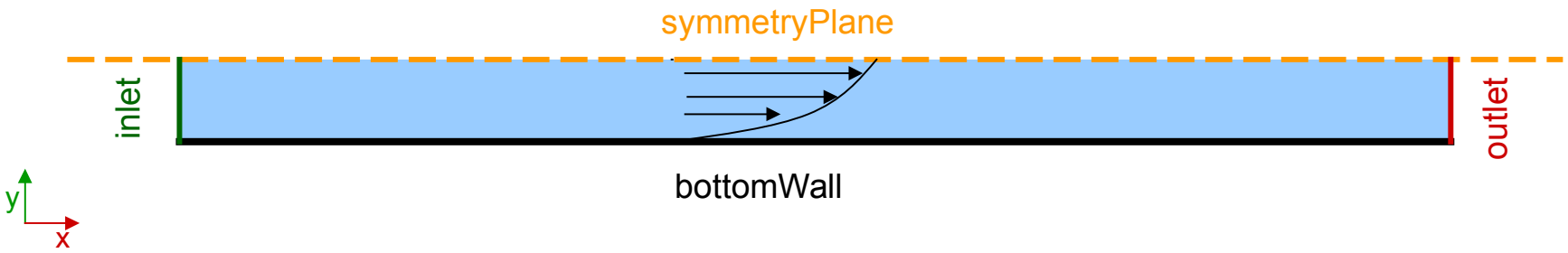
#3 – Poiseuille flow (1/4)

Objectives:

- Simulate a Poiseuille flow through a 2D pipe with symmetry plane condition
- Steady-state solution of laminar incompressible Navier-Stokes equations with the *simpleFoam* solver

$$\nabla \cdot \mathbf{U} = 0$$

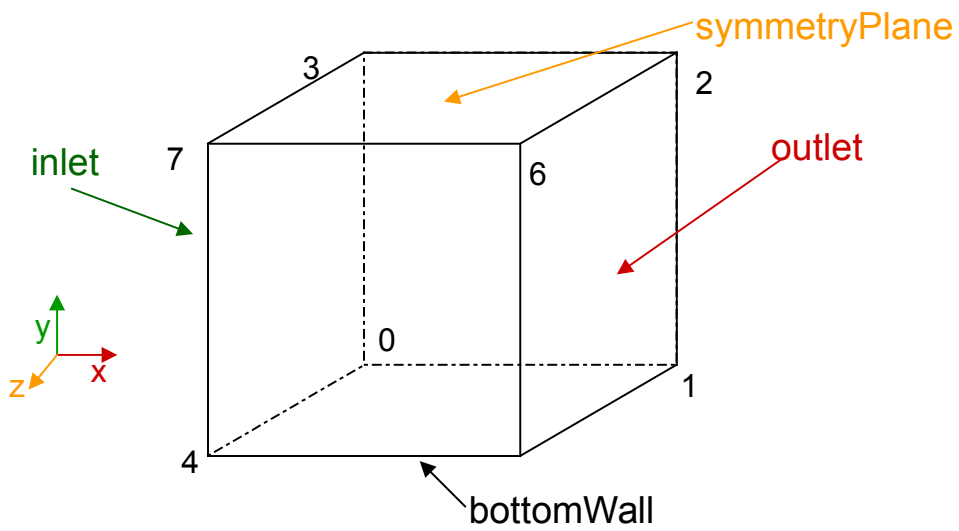
$$\nabla \cdot (\mathbf{U}\mathbf{U}) = \nabla \cdot (\nu \nabla \mathbf{U}) - \nabla p$$



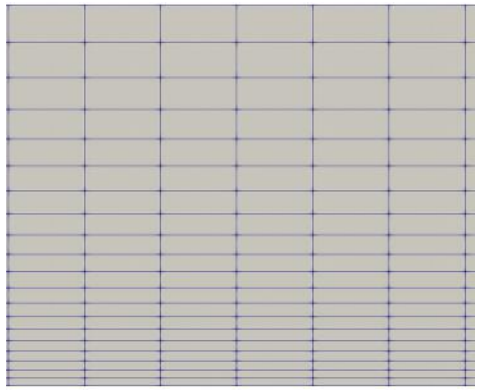

```
$ run
$ cp   –r   $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily      Exo3
$ cd   Exo3
$ cp   ../Exo2/system/blockMeshDict        system/.
```

#3 – Poiseuille flow (2/4)

$ gedit system/blockMeshDict




```
scale   1e-3;

vertices
(
    (0 0 0)
    (20 0 0)
    (20 1 0)
    (0 1 0)
    (0 0 0.1)
    (20 0 0.1)
    (20 1 0.1)
    (0 1 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (100 20 1) simpleGrading (1 5 1)
);

edges
(
);
```



```
boundary
(
    top
    {
        type symmetryPlane;
        faces
        (
            (3 7 6 2)
        );
    }
    left
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    right
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    bottom
    {
        type wall;
        faces
        (
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```

$ blockMesh
$ paraFoam

Untick U and p in the fields to view before you click on « apply »

#3 – Poiseuille flow (3a/4)

$ gedit constant/turbulenceProperties

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

simulationType laminar;

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

We specify that the simulation is laminar. All the references to RANS model are no longer necessary (discretization schemes, files 0/nut, 0/k ...)

```
$ rm 0/epsilon
$ rm 0/f
$ rm 0/k
$ rm 0/nut
$ rm 0/nuTilda
$ rm 0/omega
$ rm 0/v2
```

This step is not mandatory, the files will be simply not considered during the simulation

$ gedit constant/transportProperties

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

transportModel  Newtonian;

nu              1e-05;

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

#3 – Poiseuille flow (3b/4)

$ gedit 0/U

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

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

internalField   uniform (0 0 0);

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

    right
    {
        type            zeroGradient;
    }

    top
    {
        type            symmetryPlane;
    }

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

    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit 0/p

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

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

internalField   uniform 0;

boundaryField
{
    left
    {
        type            zeroGradient;
    }

    right
    {
        type            fixedValue;
        value           uniform 0;
    }

    top
    {
        type            symmetryPlane;
    }

    bottom
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

#3 – Poiseuille flow (3c/4)

$ gedit system/controlDict

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

application     simpleFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         2000;

deltaT          1;

writeControl    timeStep;

writeInterval   100;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

simpleFoam is a steady-state solver that uses an iterative algorithm called SIMPLE: the pressure field and the velocity matrix are under-relaxed to ease the convergence. Hence, in this case, the "time-step" refers to the iteration number (see *system/fvSolution* for convergence criteria)

Remove the "functions" block. It is not necessary in this exercise, and will cause problems since it is a laminar simulation

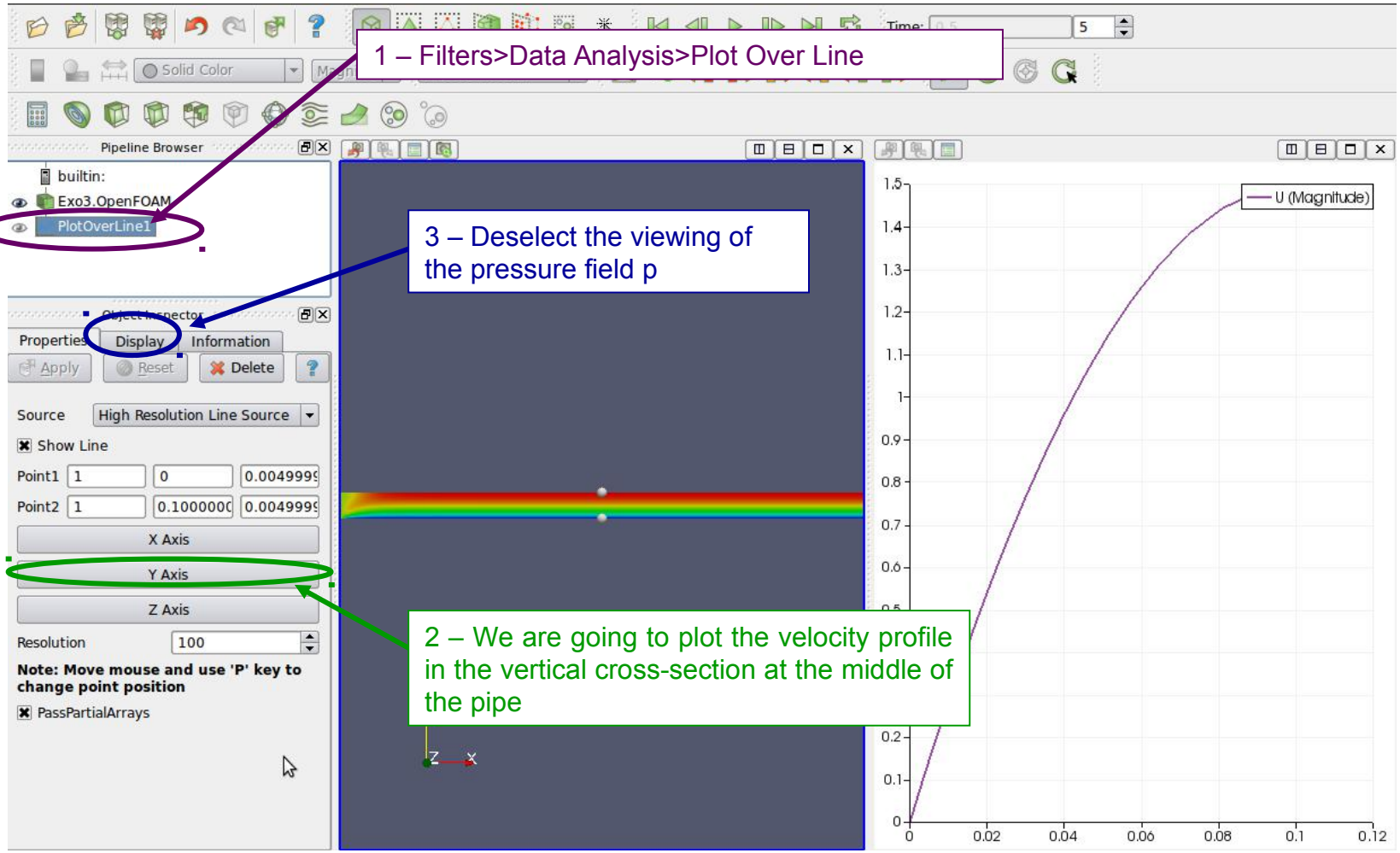
$ gedit system/fvSolution

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;
    consistent      yes;

    residualControl
    {
        p               1e-5;
        U               1e-5;
    }

}
```

The simulation stops when the residual are below this criteria.

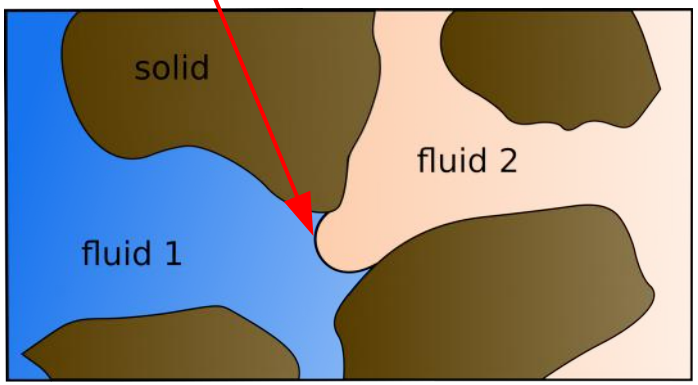
#3 – Poiseuille flow (4/4)

- To start the simulation: `$ simpleFoam`
- To view the results: `$ paraFoam`



The physics of two-phase flows

Immiscible interface



Particularity of multi-phase flow

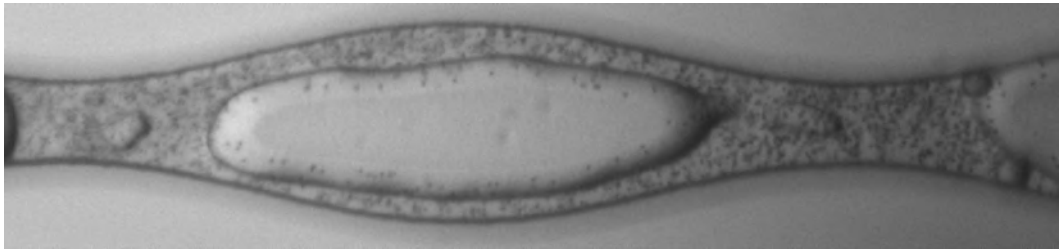
- Navier-Stokes equation in each phases
- Continuity of the tangential component of the velocity at the fluid/fluid interface
- Laplace law for a surface at the equilibrium

$$\Delta p = \sigma \left(\frac{1}{R_1} + \frac{1}{R_2} \right)$$

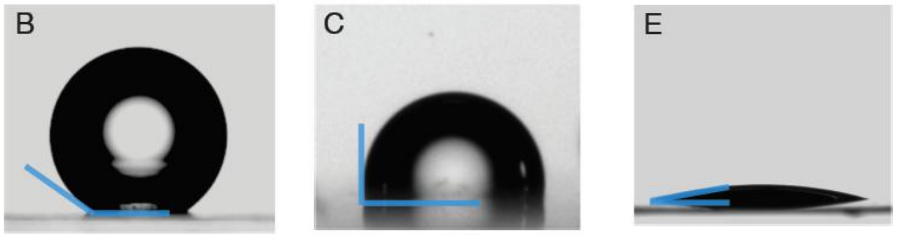
Surface tension (N/m)

- Contact line dynamics at the solid surface

Surface tension is the elastic tendency of a fluid surface which makes it acquire the least surface area possible



The contact angle quantifies the wettability affinity of a solid surface by a liquid



θ=150° non-wetting

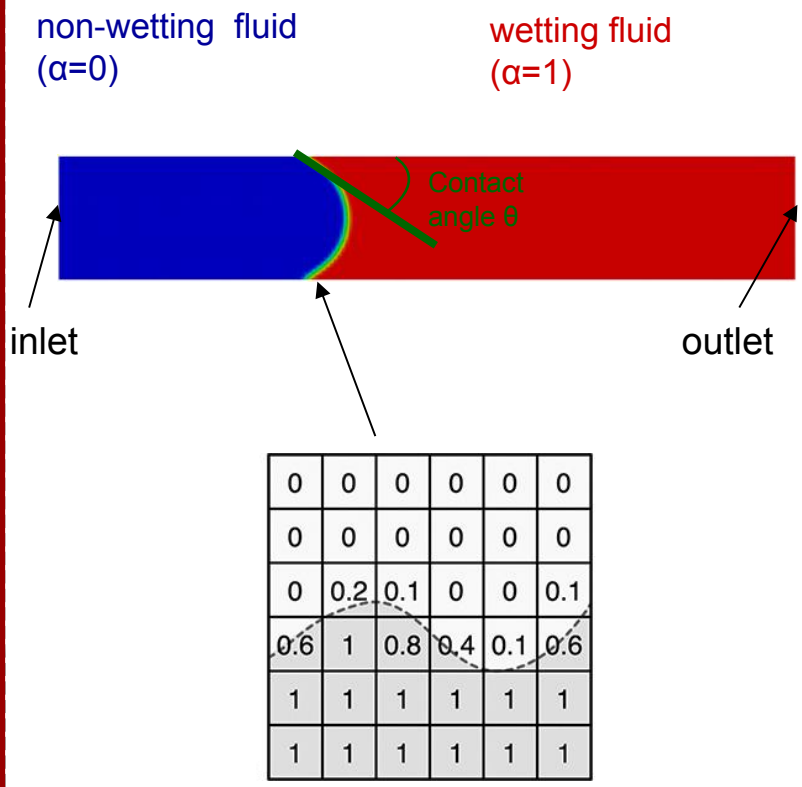
θ=90°

θ=7° wetting

Zhao et al. (2016), PNAS

The displacement of a wetting fluid by a non-wetting fluid (drainage) is different than the displacement of a non-wetting fluid by a wetting fluid (imbibition)

#4 – Drainage in a capillary tube (1/5)



Objectives:

- Simulate a drainage (a non-wetting fluid pushing a wetting fluid) experiment in a simple 2D capillary tube
- Example adapted fom the *damBreak* tutorial detailed in the official *user guide*
- Use of an interface capturing solver *(interFoam, VoF)*

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla . (\rho \mathbf{U}\mathbf{U}) = -\nabla p + \nabla . \left(\mu \left(\nabla \mathbf{U} + {}^{t}\nabla \mathbf{U}\right)\right) + \mathbf{F}_{\alpha}$$

$$\frac{\partial \alpha}{\partial t} + \nabla . (\mathbf{U}\alpha) = 0$$

$$\mathbf{U} = \alpha \mathbf{U}_l + (1-\alpha)\mathbf{U}_g \qquad \rho = \alpha \rho_l + (1-\alpha)\rho_g$$

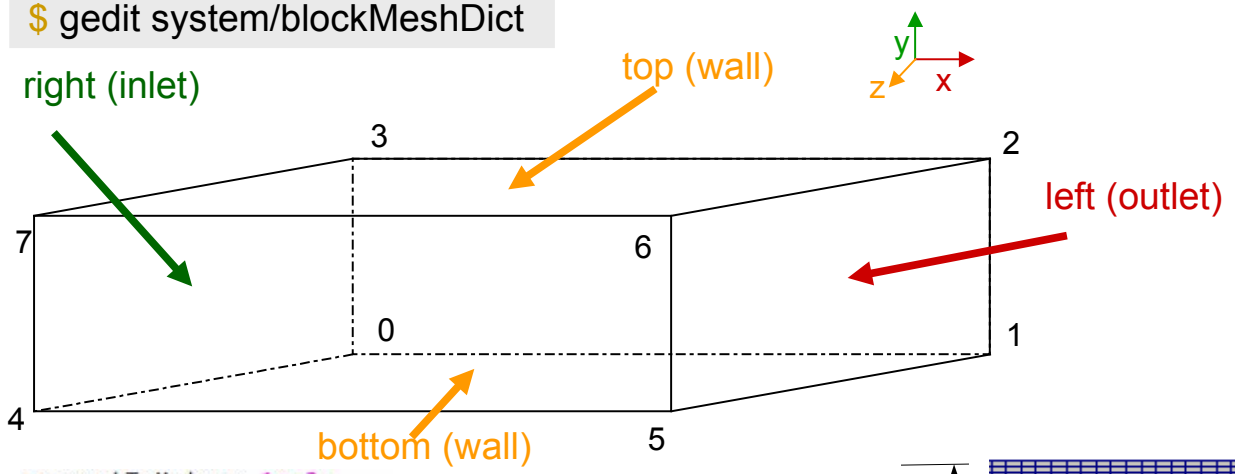
$$\mu = \alpha \mu_l + (1-\alpha)\mu_g$$

- Use of the *setFields* utility to initialize the phase distribution

```
$ run
$ cp –r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak/damBreak Exo4
$ cd Exo4
$ cp ../Exo3/system/blockMeshDict system/.
```

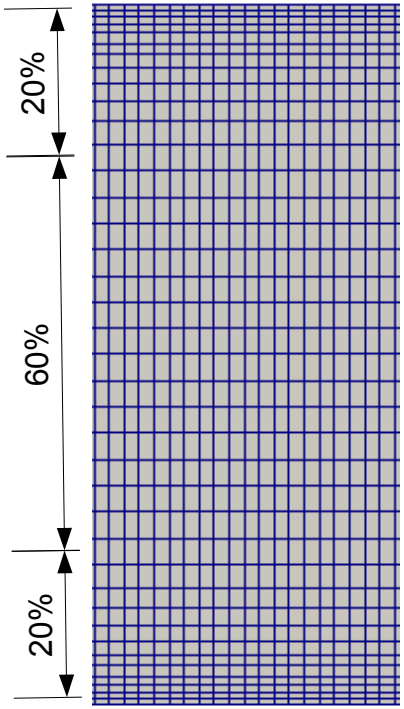
#4 – Drainage in a capillary tube (2/5)

$ gedit system/blockMeshDict




```
convertToMeters 1e-3;

vertices
(
    (0 0 0)
    (12 0 0)
    (12 2 0)
    (0 2 0)
    (0 0 0.1)
    (12 0 0.1)
    (12 2 0.1)
    (0 2 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (280 38 1)
    simpleGrading (
    1
    (
            (20 30 4)   // 20%, 30%…
            (60 40 1)
            (20 30 0.25)
    )
    1)
);
```

The grid is regular in x and z directions, but is refined near the walls in the y direction. 20% of the total length near the boundaries contain 30 % of the cells refined with a factor 4 between the closest to wall and the farthest.




```
boundary
(
    top
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    left
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    right
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    bottom
    {
        type wall;
        faces
        (
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```

#4 – Drainage in a capillary tube (3a/5)

$ gedit constant/transportProperties

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

phases (water oil);

water
{
    transportModel  Newtonian;
    nu              6e-05;
    rho             1000;
}

oil
{
    transportModel  Newtonian;
    nu              6e-05;
    rho             10;
}

sigma               0.097;
```

Name of the wetting phase (here « water ») and of the non-wetting phase (here « oil »). On the computational grid, the two phases are differentiated with the phase indicator alpha.water (=1 for the wetting phase, =0 for the non-wetting phase)

Mind the space between “phases” and “(“ !

Properties of the “water” phase

Properties of the “oil” phase

Surface tension

#4 – Drainage in a capillary tube (3b/5)

$ gedit 0/U

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * *

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

internalField   uniform (0 0 0);

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform (0.01 0 0);
    }
    right
    {
        type            zeroGradient;
    }
    "(bottom|top)"
    {
        type            noSlip;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit 0/p_rgh

```
FoamFile
{
    version     2.0;
    format      ascii;
    ss          volScalarField;
    ect         p_rgh;
}
// * * * * * * * * * * * * * * * * * * * * * * * * *

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

internalField   uniform 0;

boundaryField
{
    left
    {
        type            zeroGradient;
    }
    right
    {
        type            fixedValue;
        value           uniform 0;
    }
    "(bottom|top)"
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

We use a hydrostatic pressure

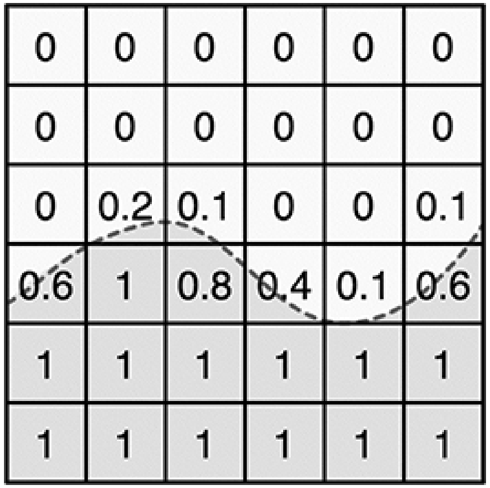
The patches « top » and « bottom » have the same boundary conditions

#4 – Drainage in a capillary tube (3c/5)

```
$ gedit   0/alpha.water.orig
```

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      alpha.water;
}
// * * * * * * * * * * * * * * * * * * * * * * * * *

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform 1;
    }

    right
    {
        type            zeroGradient;
    }

    "(bottom|top)"
    {
        type            constantAlphaContactAngle;
        value           uniform 1;
        theta0          45;
        limit           gradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

alpha.water represents the wetting/non-wetting phase distribution in the computational domain. (*alpha=0* for the non-wetting, *alpha=1* for the wetting)



Definition of the contact angle at the solid boundaries. Here theta=45 degrees

limit gradient to limit the wall-gradient such that alpha remains bounded on the wall.

#4 – Drainage in a capillary tube (3d/5)

Specify that the simulation will be without gravity in a laminar flow regime

$ gedit constant/g

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

dimensions      [0 1 -2 0 0 0 0];
value           (0 0 0);
```

$ gedit constant/turbulenceProperties

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

simulationType  laminar;
```

#4 – Drainage in a capillary tube (3e/5)

```
$ gedit   system/controlDict
```

```
application     interFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         1;

deltaT          1e-5;

writeControl    adjustableRunTime;

writeInterval   0.1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;

adjustTimeStep  yes;

maxCo           0.5;
maxAlphaCo      0.5;

maxDeltaT       1e-2;
```

Set the *writeControl* parameter to *adjustableRunTime* when using an adjustable time step. The output files will be written every *writeInterval* seconds.

If *yes* value, then it means that the *controlDict* file can be modified on the fly.

Switch on the automatic time step management according to the Courant Numbers value (*maxCo* for the pressure/velocity coupling and *maxAlphaCo* for the explicit transport of *alpha*).

maxDeltaT restricts the maximum value of the time step.

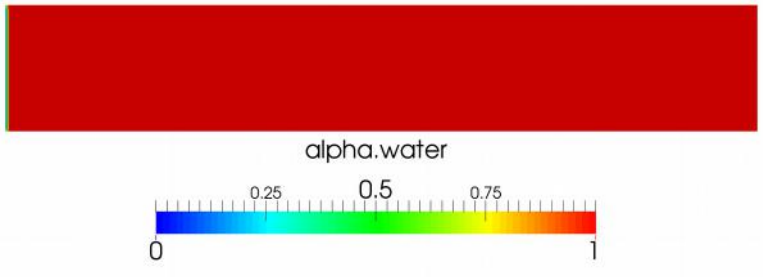
#4 – Drainage in a capillary tube (4/5)

Before we start the simulation, we are going to specify the initial phase distribution with *setFields*

```
$ cp    0/alpha.water.org     0/alpha.water
$ blockMesh
$ paraFoam
$ gedit    system/setFieldsDict
$ setFields
$ paraFoam
```

```
defaultFieldValues
(
    volScalarFieldValue alpha.water 1
);

regions
(
    boxToCell
    {
        box (0 0 0) (0.5e-3 2e-3 0.1e-3);
        fieldValues
        (
            volScalarFieldValue alpha.water 0
        );
    }
);
```

BEFORE



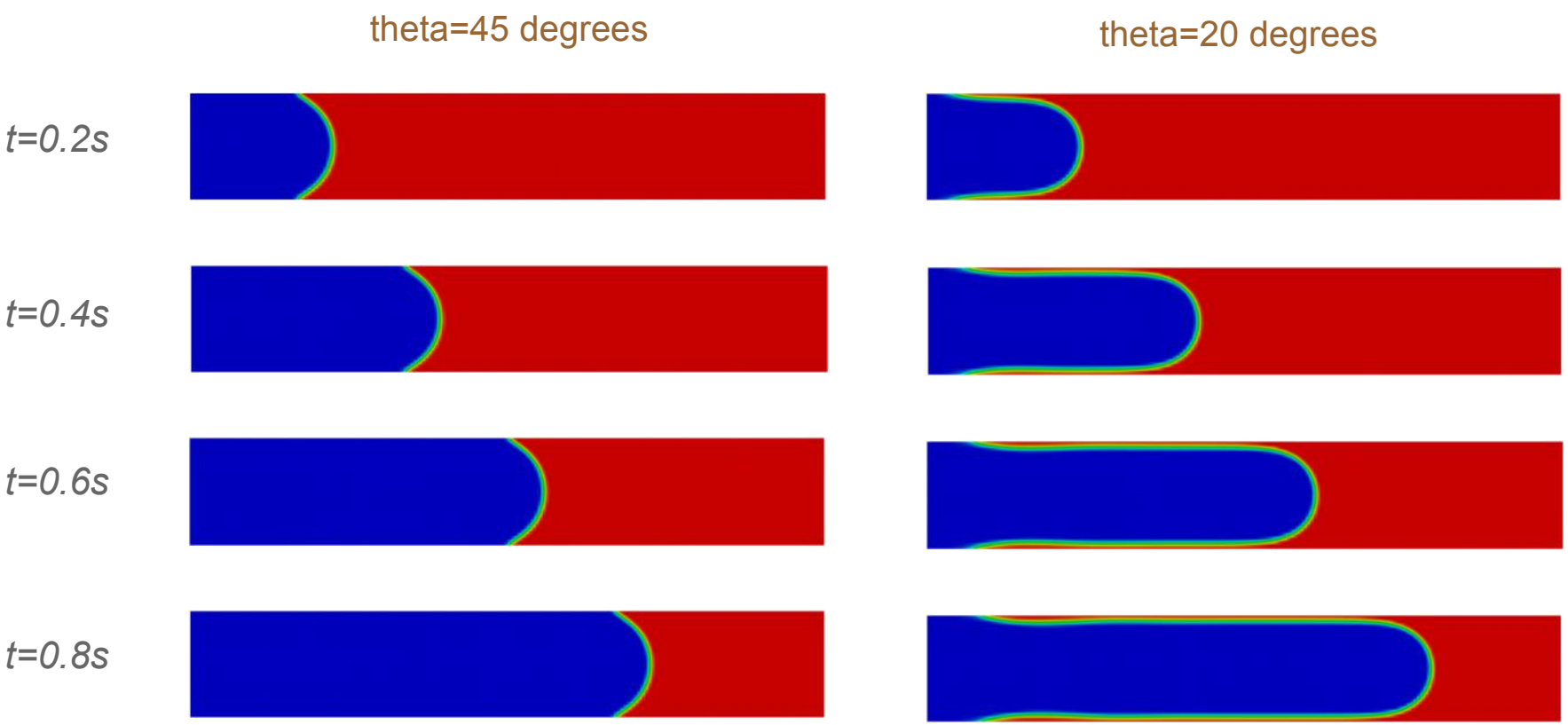
AFTER

setFields overwrites *0/alpha.water.* It is recommanded to make a backup before using it (that the purpose of *0/alpha.water.orig).*

#4 – Drainage in a capillary tube (5/5)

Start the immiscible two-phase flow simulation:

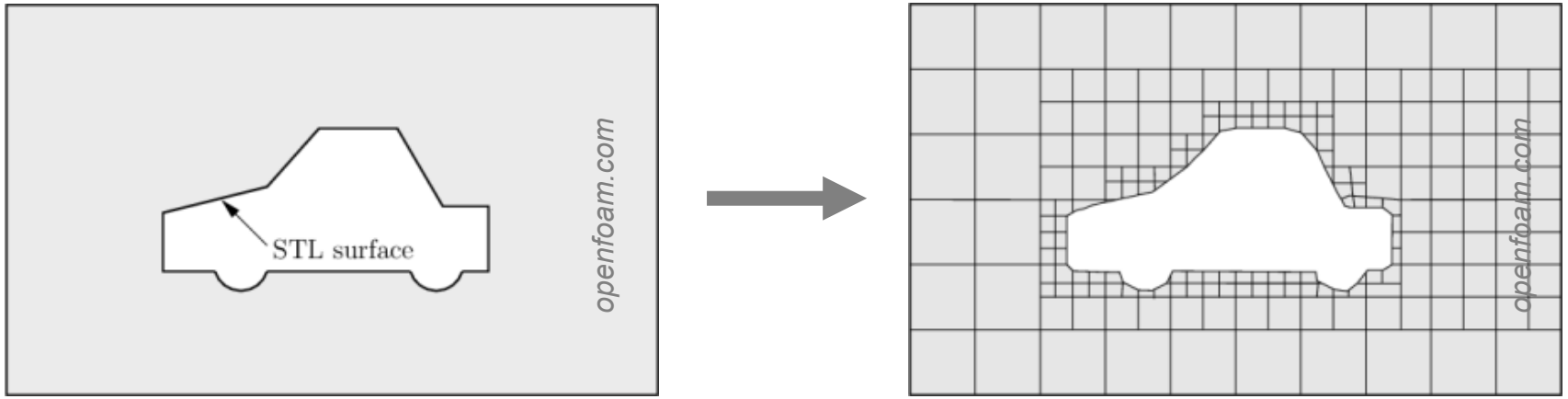
```
$ interFoam
```



Exo4bis : Same exercise with a contact angle of 20 degrees

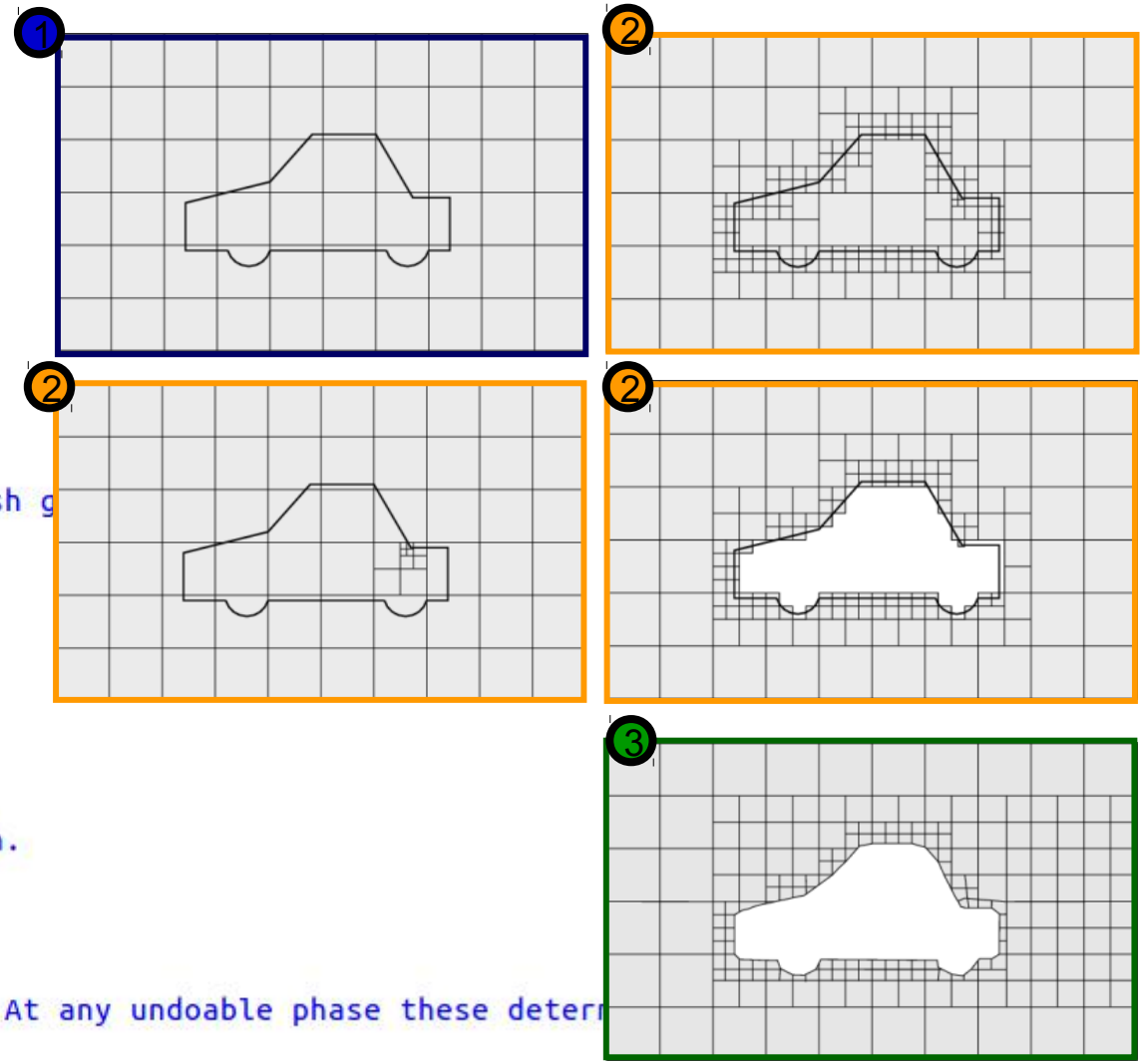
snappyHexMesh overview (1/2)

- *snappyHexMesh* is an automatic and robust mesher able to grid any complex geometry
- Mesh a region inside or/and around an object described by a surface mesh
- It is compatible with a lot of input formats resulting from CAD softwares or tomography imaging *(*.stl*, **.obj*, **.vtk* …)
- Maximize the number of hexahedral cells



snappyHexMesh overview (2/2)

```
// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       false;

geometry
{
    Exo5.stl
    {
        type triSurfaceMesh;
        name fixedWalls;
    }
};

// Settings for the castellatedMesh g
castellatedMeshControls
{
}

// Settings for the snapping.
snapControls
{
}

// Settings for the layer addition.
addLayersControls
{
}

// Generic mesh quality settings. At any undoable phase these deter
// where to undo.
meshQualityControls
{
}
```

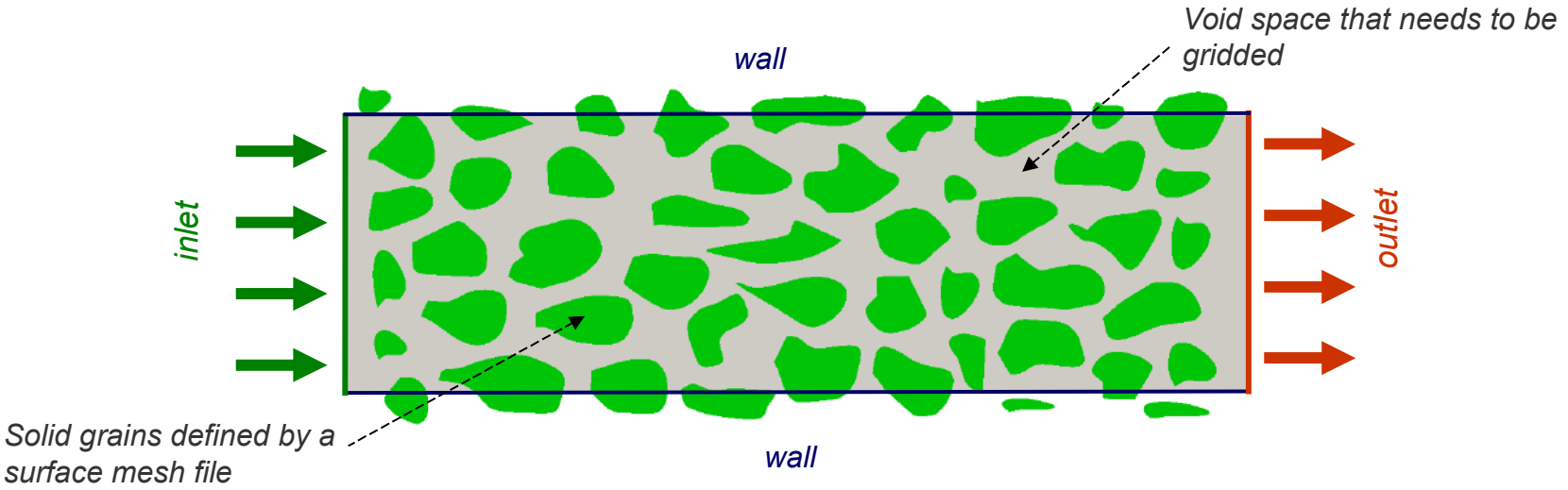


openfoam.com

#5 – Mesh a pore-space (1/6)

Objective:

- Mesh the void space of a porous medium with *snappyHexMesh*



1. Generate the surface mesh (here the solid grain).

2. Create a suitable background mesh with *blockMesh.* It needs to be fine enough to have at least 10 cells in the pore-throat thickness.

3. Detect the void space, remove the cells occupied by the solid and snap them to fit as close as possible the initial surface object with *snappyHexMesh.*

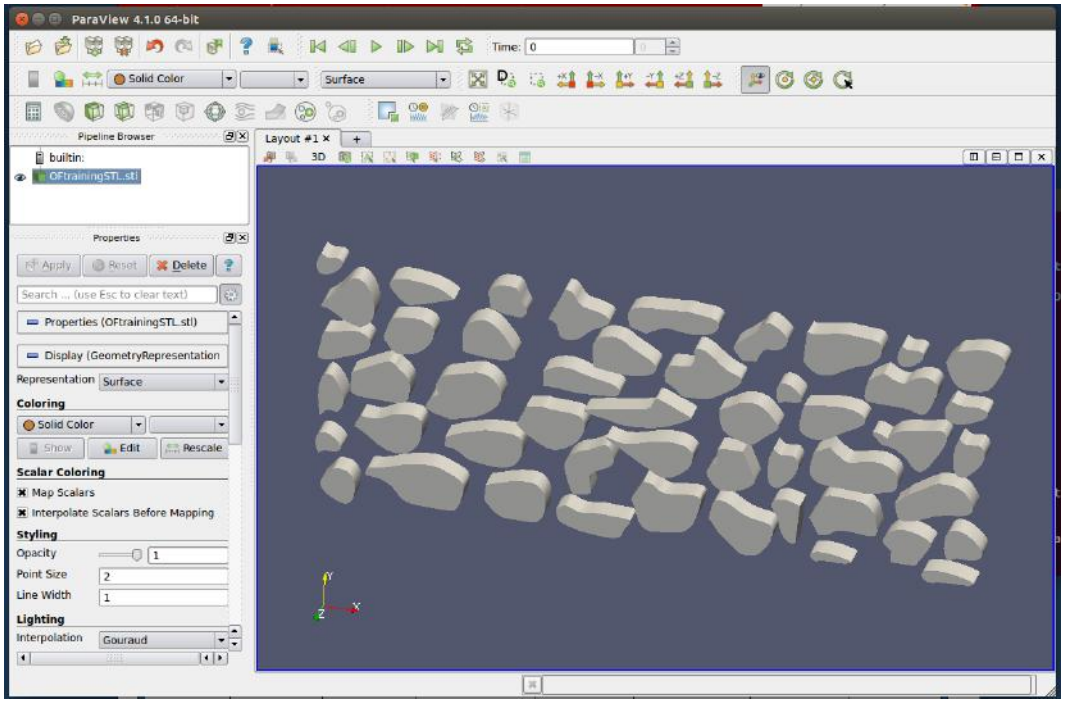
#5 – Mesh a pore-space (2/6)

We are going to adapt the *motorBike* tutorial and use an existing *stl* file*

```
$ run
$ cp   -r  $FOAM_TUTORIALS/incompressible/simpleFoam/motorBike/   Exo5
$ cd   Exo5
```

Download *Exo5.stl* and copy it to the folder *Exo5*. To view the geometry,

```
$ paraview     Exo5.stl
```



* This file can be downloaded at: http://web.stanford.edu/~csoulain/OF_Training/Exo5/Exo5.stl

#5 – Mesh a pore-space (3/6)

The next step consists in creating a background mesh with *blockMesh.* The size of this background domain depends on the bounding box of the surface object. It can be obtained using the *surfaceCheck* tool,

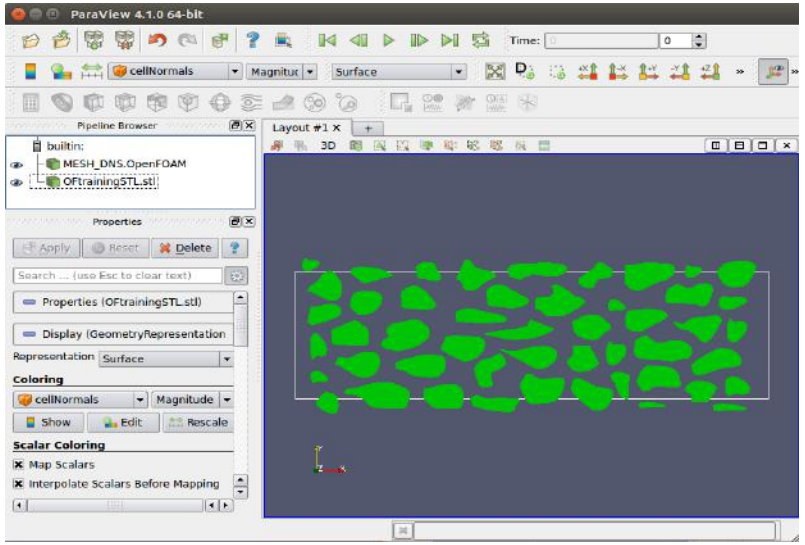
```
$ surfaceCheck Exo5.stl | grep –i 'bounding box'
```

```
Bounding Box : (-0.0447606 -0.0172845 -0.002) (0.0436509 0.0170043 0.002)
```

```
$ gedit system/blockMeshDict
```

```
convertToMeters 1;

lx0 -0.046;
ly0 -0.014;
lz0 -0.001;

lx1  0.046;
ly1  0.014;
lz1  0.001;

vertices
(
    ($lx0   $ly0    $lz0)       //0
    ($lx1   $ly0    $lz0)       //1
    ($lx1   $ly1    $lz0)       //2
    ($lx0   $ly1    $lz0)       //3
    ($lx0   $ly0    $lz1)       //4
    ($lx1   $ly0    $lz1)       //5
    ($lx1   $ly1    $lz1)       //6
    ($lx0   $ly1    $lz1)       //7
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (400 100 1) simpleGrading (1 1 1)
);
```

The grid has to be fine enough in order to capture all the details of the pore-space.




```
$ blockMesh
$ paraFoam
```

To superimpose the background grid and the surface mesh file, in ParaView:

file>open>Exo5.stl

Specify the boundaries *top, bottom, left, right, frontAndBack* *(see page 32)*

#5 – Mesh a pore-space (4/6)

The void space is meshed with *snappyHexMesh*, based on the background grid. *snappyHexMesh* needs an input dictionary located in the *system* folder and the object file (Exo5.stl) located in *constant/triSurface*,

```
$ mv    Exo5.stl    constant/triSurface/.
$ gedit    system/snappyHexMeshDict
```

```
// Which of the steps to run
castellatedMesh true;
snap            false;
addLayers       false;
```

The three different stages of the meshing can be performed separately or simultaneously:

- *castellatedMesh*: detects the intersections between the surface object and the background grid. It can eventually refine the background grid at the vicinity of the object. Then it removes either the inside or the outside of the object.
- *snap*: Introduces tetrahedral cells at the object boundary to match the actual geometry.
- *addLayers*: Add layers of cells on the boundaries.

```
// Geometry. Definition of all surfaces.
geometry
{
    Exo5.stl
    {
        type triSurfaceMesh;
        name fixedWalls;
    }

};
```

Insertion of the surface mesh object. Several objects may be inserted at the same times.

The solid boundaries will be named *"fixedWalls"*

#5 – Mesh a pore-space (5/6)

The « castellated mesh » step

```
castellatedMeshControls
{
    // Refinement parameters
    maxLocalCells 100000;
    maxGlobalCells 2000000;
    minRefinementCells 10;
    maxLoadUnbalance 0.10;
    nCellsBetweenLevels 3;

    // Explicit feature edge refinement
    features
    (
    );

    // Surface based refinement
    refinementSurfaces
    {
        fixedWalls
        {
            // Surface-wise min and max refinement level
            level (0 0);
        }
    }

    refinementRegions
    {
    }

    // Resolve sharp angles
    resolveFeatureAngle 30;

    // Mesh selection
    // ~~~~~~~~~~~~~~
    locationInMesh (-0.04 0 0);

    // Face zone
    allowFreeStandingZoneFaces false;
}
```

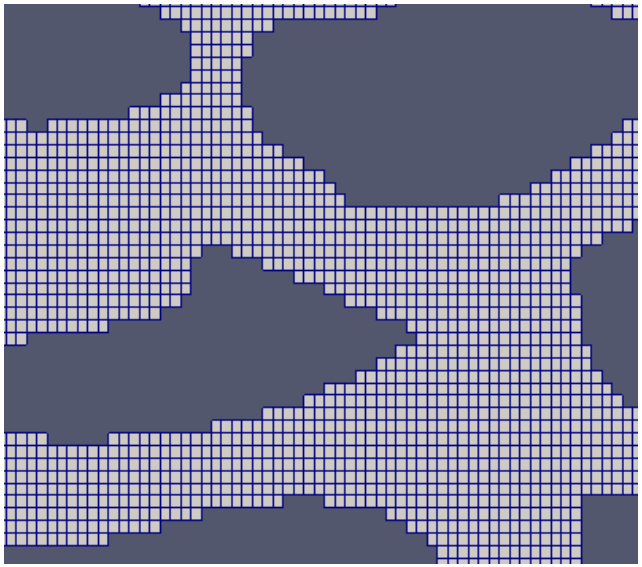
Set up the refinement level at the vicinity of the object. Actually, this feature only works for 3D geometry.

We can specify zero refinements with *level (0 0), i.e.* we only rely on the background grid

Point a location where there is void space. All cells outside the void space will be removed.

```
$ snappyHexMesh
$ paraFoam
```

The mesh is created in a new time step (*« 1 »*) directory. In order to view it, in ParaView, you have to go to the time *« 1 »*

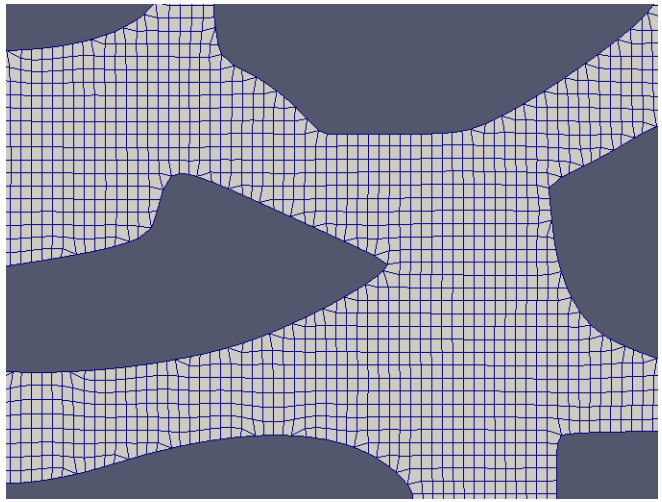


#5 – Mesh a pore-space (6/6)

- The next step « snap » is the snapping stage to fit the STL surface as close as possible. This stage will introduce some tetrahedral cells into the computational domain.

```
$ gedit system/snappyHexMeshDict
```

```
// Which of the steps to run
castellatedMesh false;
snap            true;
addLayers       false;

// Settings for the snapping.
snapControls
{
    nSmoothPatch 3;

    tolerance 2.0;

    nSolveIter 30;

    nRelaxIter 5;
}
```



```
$ snappyHexMesh
$ paraFoam
```

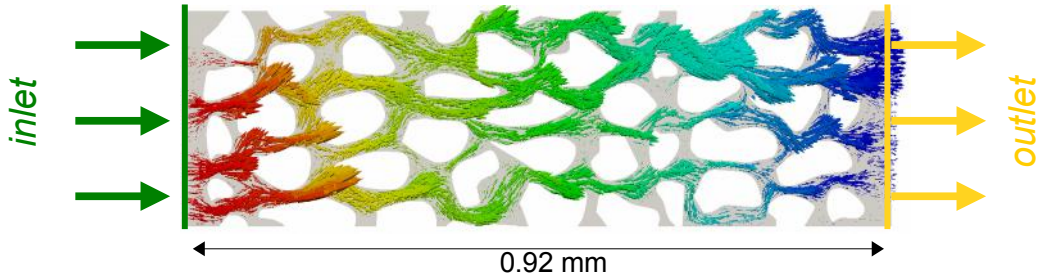
The mesh is created in another time step (« 2 ») folder. You have to select the time step « 2 » in ParaView to view it. You can use the option -overwrite to overwrite the grid in constant/polyMesh

- Eventually, we could have used the *addLayers* stage to add layers of cells around the solid grains.

#6 – Scalar transport in the pore space (1/8)

Objectives:

- Solve the flow in the void space gridded in the previous exercise (#5)
- Estimate the permeability of the porous medium
- Solve a passive scalar transport in the void space



- The flow and the scalar transport are uncoupled. They can therefore be solved one after the other.
- The flow is obtained solving a Stokes problem with *simpleFoam*. The case can be setup based on #3 (or $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily)

```
$ run
$ cp   -r   Exo3    Exo6a
$ cd   Exo6a
```

- The mesh from #3 is replaced by the grid of #5 and scaled to the actual size

```
$ rm   -r    constant/polyMesh  83/
$ cp   -r    ../Exo5/2/polyMesh    constant/.
$ transformPoints    -scale    '(0.01     0.01    0.01)'
$ checkMesh | grep  -i  'bounding box'
```

#6 – Scalar transport in the pore space (2a/8)

$ gedit 0/U

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

internalField   uniform (0 0 0);

boundaryField
{

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

    left
    {
        type            zeroGradient;
    }
    right
    {
        type            zeroGradient;
    }
    bottom
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    top
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit 0/p

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

internalField   uniform 0;

boundaryField
{

    fixedWalls
    {
        type            zeroGradient;
    }

    left
    {
        type            fixedValue;
        value           uniform 1e-3;
    }

    right
    {
        type            fixedValue;
        value           uniform 0;
    }

    bottom
    {
        type            zeroGradient;
    }

    top
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

#6 – Scalar transport in the pore space (2b/8)

- Setup the fluid properties

$ gedit constant/transportProperties

```
transportModel  Newtonian;

nu              nu   [ 0 2 -1 0 0 0 0 ] 1e-6;
```

- Switch off the turbulence model

$ gedit constant/turbulenceProperties

```
simulationType laminar;
```

$ gedit system/fvSolution

The simulation will stop when these residuals are reached.

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;
    consistent      yes;

    residualControl
    {
        p               1e-6;
        U               1e-6;
    }

}

relaxationFactors
{
    equations
    {
        U               0.9;
    }
}
```

Relaxation factors for the SIMPLE algorithm. The value 0.9 for *U* is faster for Stokes flows.

#6 – Scalar transport in the pore space (2c/8)

```
$ gedit    system/controlDict
```

```
application     simpleFoam;

startFrom       latestTime;

startTime       0.0;

stopAt          endTime;

endTime         1000.0;

deltaT          1;

writeControl    runTime;

writeInterval   1000;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;
```

Since we have specified convergence criteria in *system/fvSolution*, the simulation may stop before *endTime*. In that case, the latest simulated time step will be automatically written.

#6 – Scalar transport in the pore space (3/8)

- It might be useful to plot the residuals on-the-fly to check the simulation convergence. This can be achieved by redirecting the simulation log into a file and extracting the residual values with the following *gnuplot* script :

```
$ gedit    plotResiduals
```

```
set logscale y
set title "Residuals"
set ylabel 'Residual'
set xlabel 'Iteration'
plot "< cat log | grep 'Solving for Ux' | cut -d' ' -f9 | tr -d ','" title 'Ux' with lines,\
     "< cat log | grep 'Solving for Uy' | cut -d' ' -f9 | tr -d ','" title 'Uy' with lines,\
     "< cat log | grep 'Solving for p'  | cut -d' ' -f9 | tr -d ','" title 'p' with lines
pause 1
reread
```

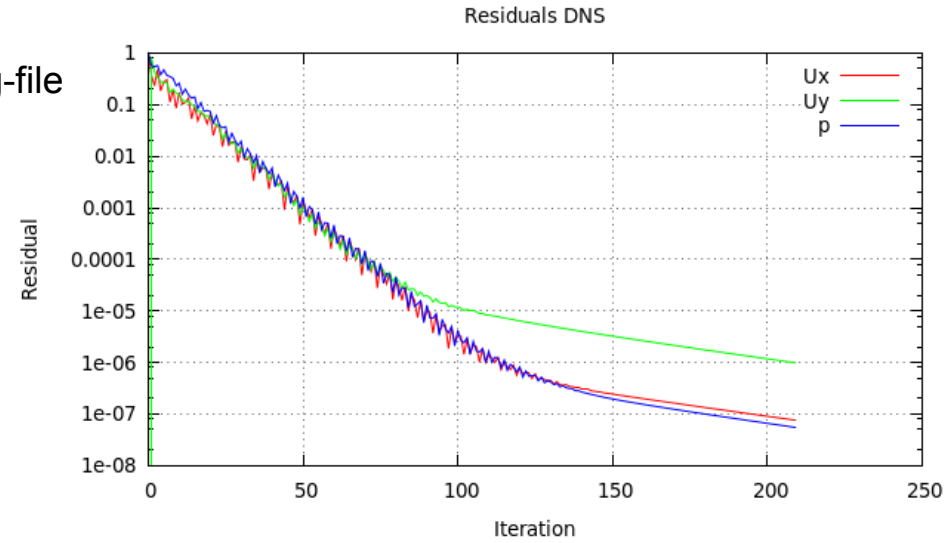
- Run the simulation and make it write out a log-file

```
$ simpleFoam    >    log    &
```

- Plot the residuals convergence with gnuplot

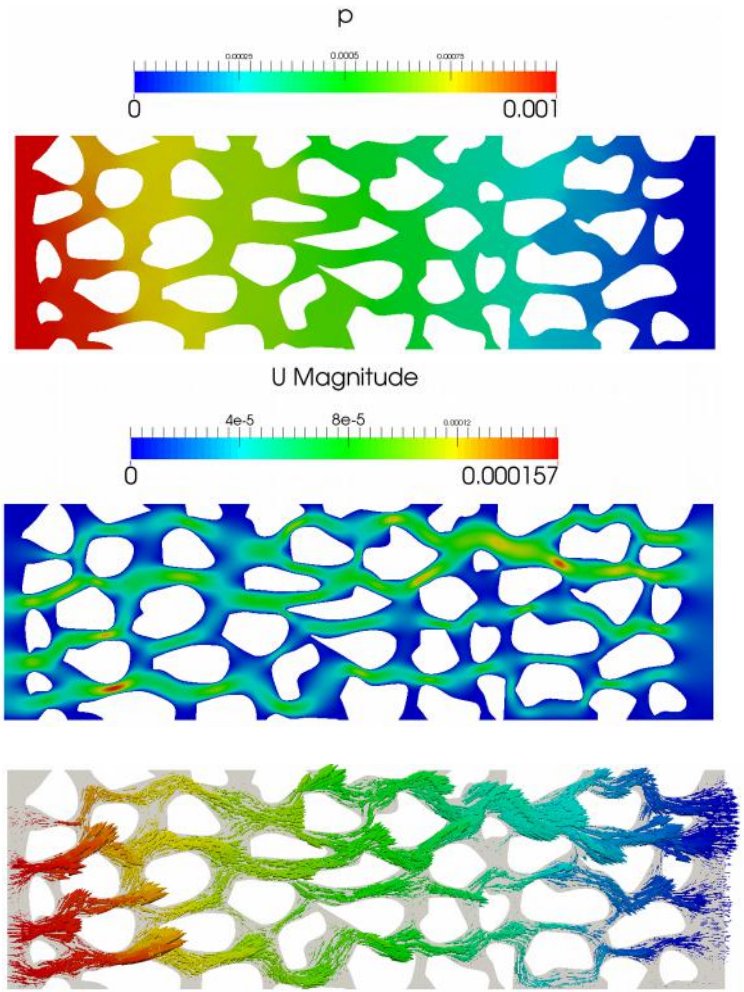
```
$ gnuplot     plotResiduals
```

(Ctrl + C to quit gnuplot)



#6 – Scalar transport in the pore space (4b/8)

```
$ paraFoam
```



The porosity (ε) is the volume of the mesh (V_{void}) over the volume (V) of the bounding box:

```
$ checkMesh | grep -i 'volume'
```

```
Min volume = 2.15708e-17. Max volume = 9.76861e-17.  Total volume = 2.92758e-12.  Cell volumes OK.
```

$V_{void} = 2.93 \times 10^{-12}\ m^3$

```
$ checkMesh | grep -i 'bounding box'
```

```
Overall domain bounding box (-0.00046 -0.00014 -1e-05) (0.00046 0.00014 1e-05)
```

$V = Lx*Ly*Lz = 5.15 \times 10^{-12}\ m^3$

$\varepsilon = 0.57$

The permeability K_{xx} is defined as:

$$K_{xx} = \mu \left(\frac{P_1 - P_0}{L_x} \right)^{-1} \left(\frac{1}{V} \int_V v_x dV \right)$$

The velocity can be integrated directly from ParaView with the filter:

filters>Data Analysis>integrateVariables

$K_{xx} = 1.6 \times 10^{-11}\ m^2$

Exo6a bis : Compare the residual convergence for other relaxation factors: 0.8 and 0.95

#6 – Scalar transport in the pore space (5/8)

- Once the flow is solved, we can use the velocity profile to transport (advection-diffusion) a scalar T with the solver *scalarTransportFoam,*

$$\frac{\partial T}{\partial t} + \nabla.(\mathbf{U}T) = \nabla.(D_T \nabla T)$$

- For that purpose, we adapt the tutorial *scalarTransportFoam/pitzDaily*

```
$ run
$ cp    -r    $FOAM_TUTORIALS/basic/scalarTransportFoam/pitzDaily      Exo6b
$ cd     Exo6b
$ rm   -r  constant/polyMesh
```

- We retreive the mesh and the resulting velocity profile from the previous simulation #6a

```
$ cp    -r    ../Exo6a/constant/polyMesh/    constant/.
$ cp    ../Exo6a/latestTime/U     0/U
```

#6 – Scalar transport in the pore space (6/8)

$ gedit 0/T

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

internalField   uniform 0;

boundaryField
{
    left
    {
        type            fixedValue;
        value           uniform 1;
    }

    right
    {
        type            zeroGradient;
    }

    bottom
    {
        type            zeroGradient;
    }

    top
    {
        type            zeroGradient;
    }

    fixedWalls
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit constant/transportProperties

```
DT              DT [ 0 2 -1 0 0 0 0 ] 1e-6;
```

$ gedit system/controlDict

```
application     scalarTransportFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         5;

deltaT          1e-1;

writeControl    runTime;

writeInterval   1e-1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

#6 – Scalar transport in the pore space (7/8)

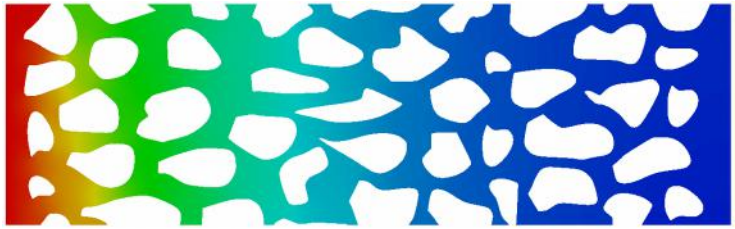
- Run the simulation

```
$ scalarTransportFoam
```

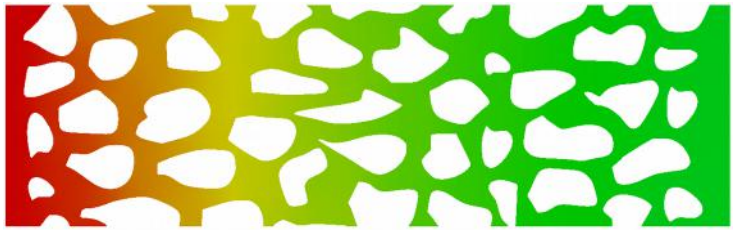
- Visualize the results

```
$ paraFoam
```

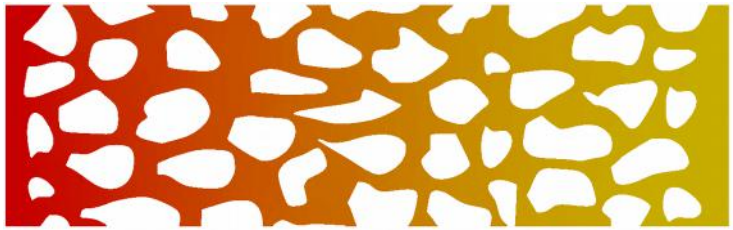
t=0.05s



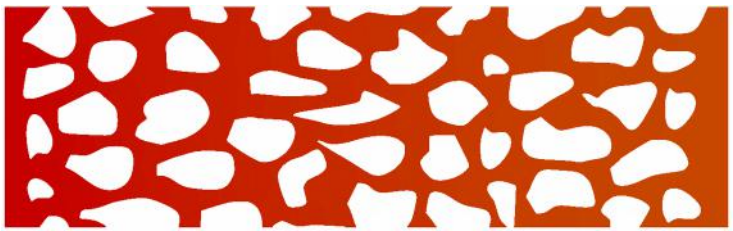
t=0.10s



t=0.15s



t=0.20s



#6 – Scalar transport in the pore space (8/8)

- To obtain a breakthrough curve (evolution of the concentration at the outlet), we look for the average value at the outlet boundary for all time steps

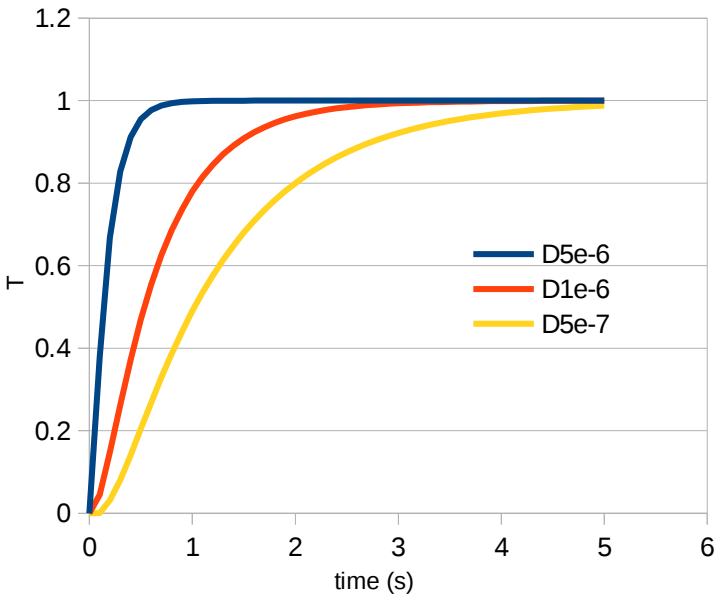
```
$ postProcess   -func  'patchAverage(name=right,T)'   >     log.patchAverage
```

- Extract the values from the log file with the following command

```
$ cat  log.patchAverage  |  grep   -i   'average(right) of T'   |    cut   -d' '   -f9   >    log.breakthrough
```

- Plot the breakthrough curve

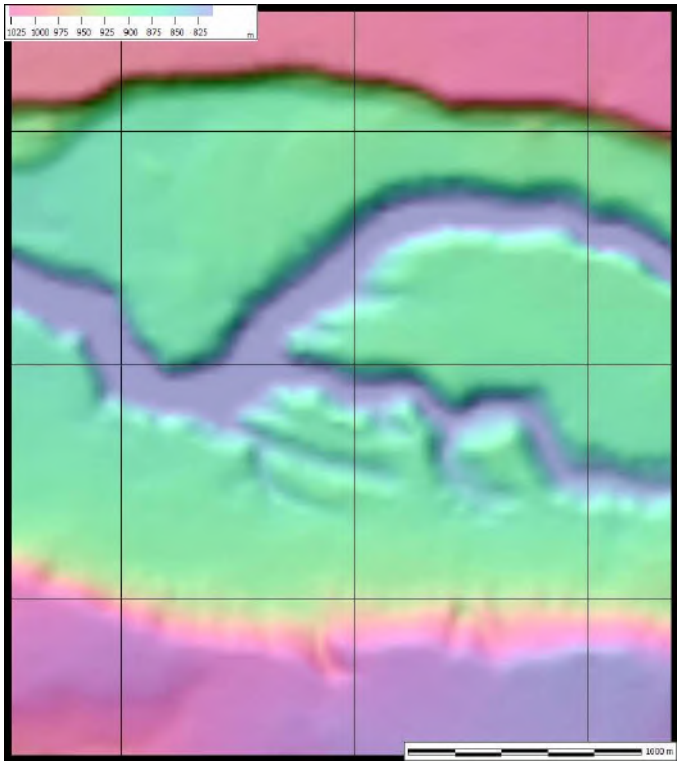
```
$ gnuplot
gnuplot> plot  'log.breakthrough'    with    lines  lw  4
```

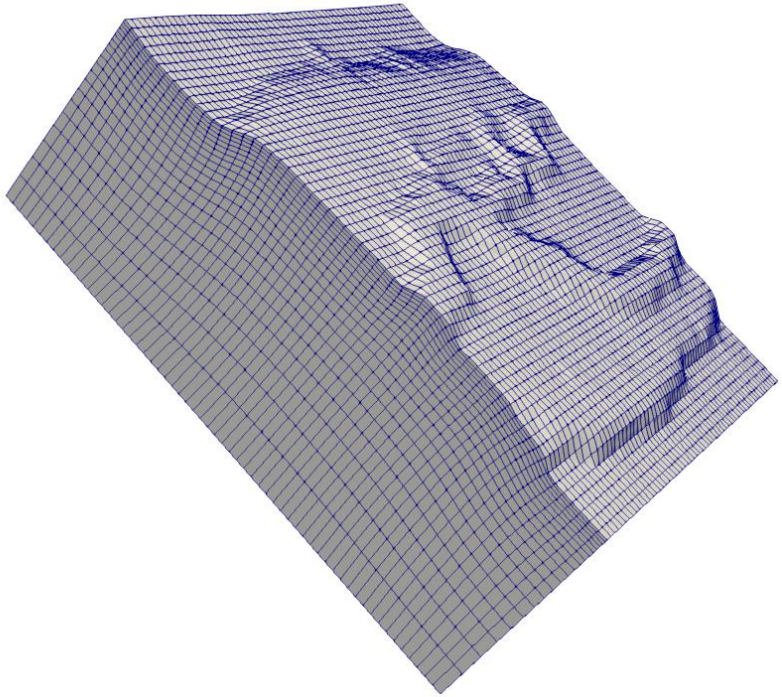


Exo6bis : Plot the breakthrough curves for different values of the diffusivity ($D=5x10^{-6}$ m^2/s and $D=5x10^{-7}$ m^2/s)

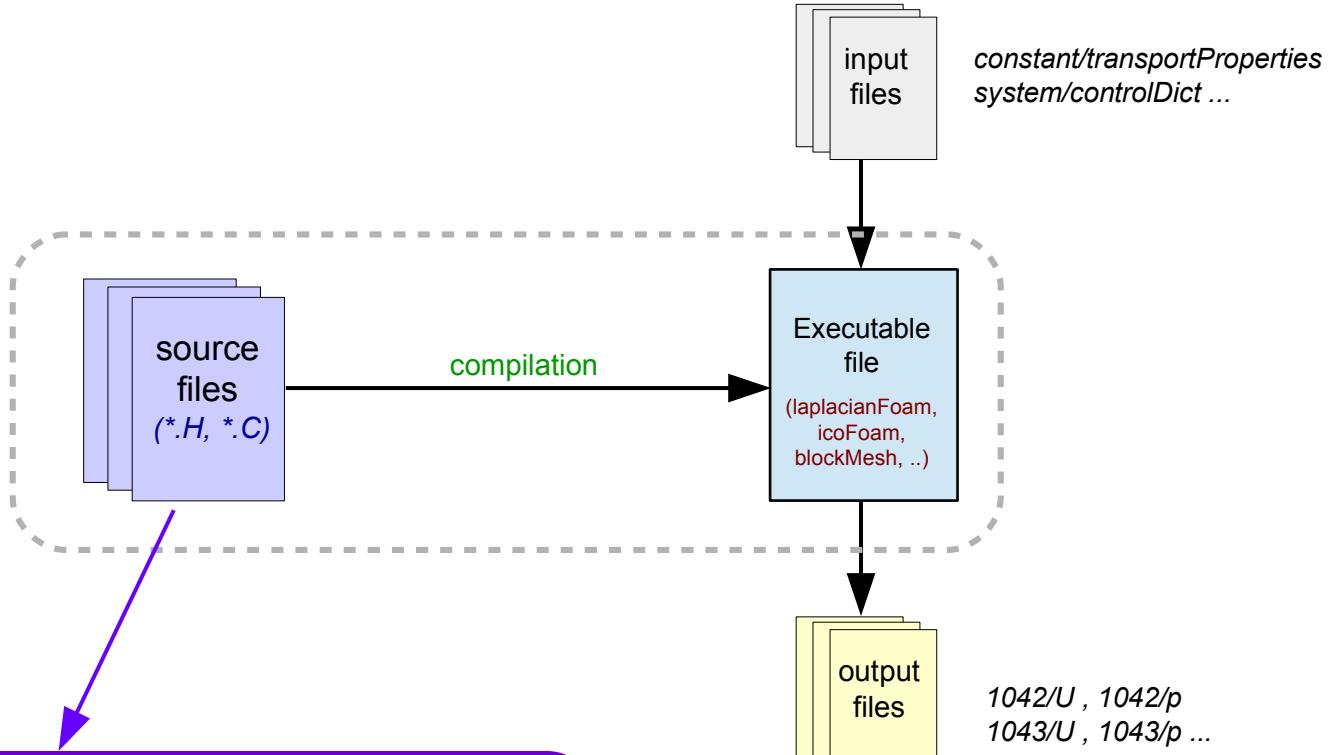
Snake River Canyon

```
$ run
$ cp    -r    $FOAM_TUTORIALS/mesh/moveDynamicMesh/SnakeRiverCanyon    .
$ cd    SnakeRiverCanyon
$ blockMesh
$ moveDynamicMesh
$ paraFoam
```





Programming in OpenFOAM®

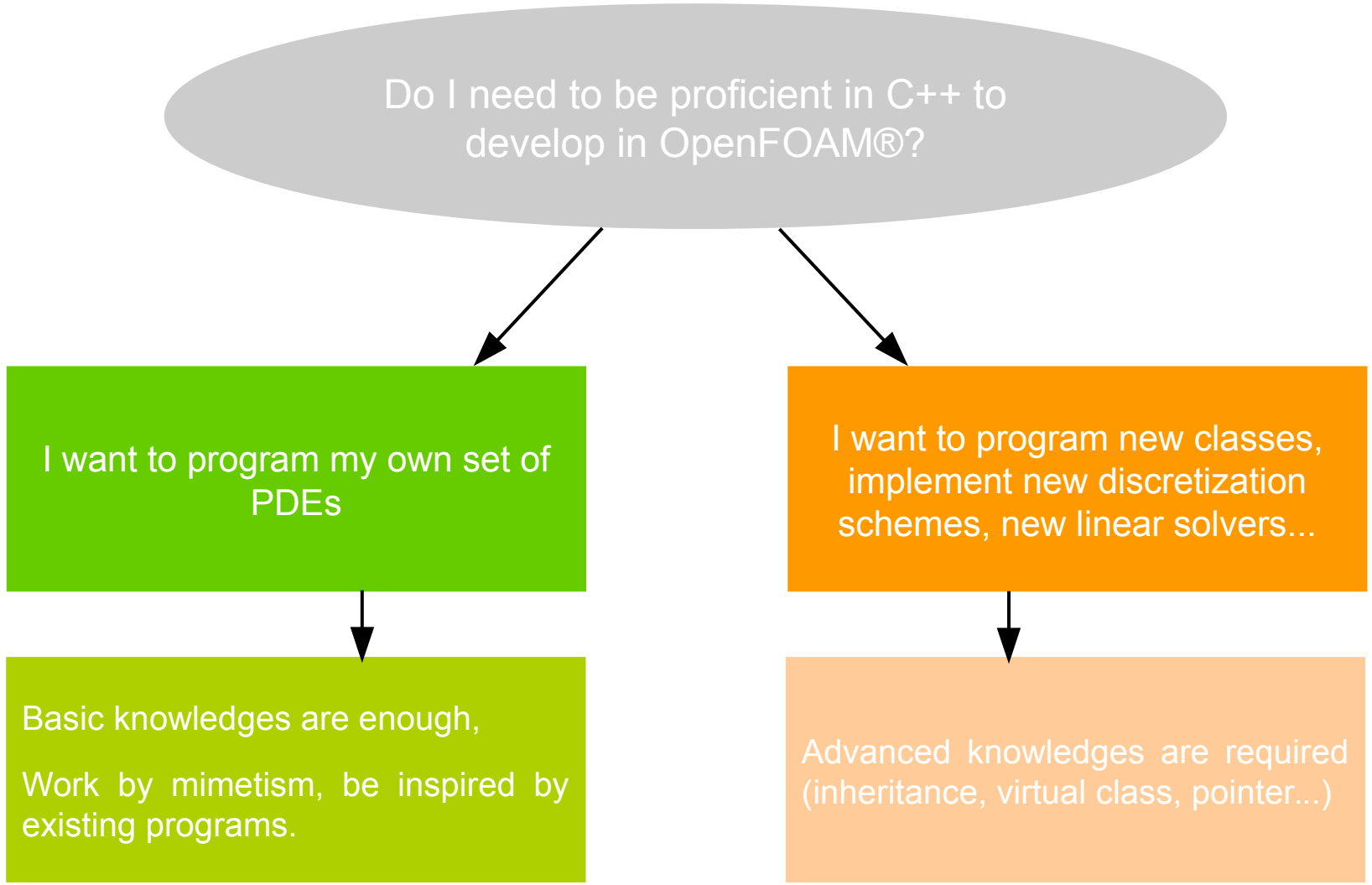


OpenFOAM® written in C++

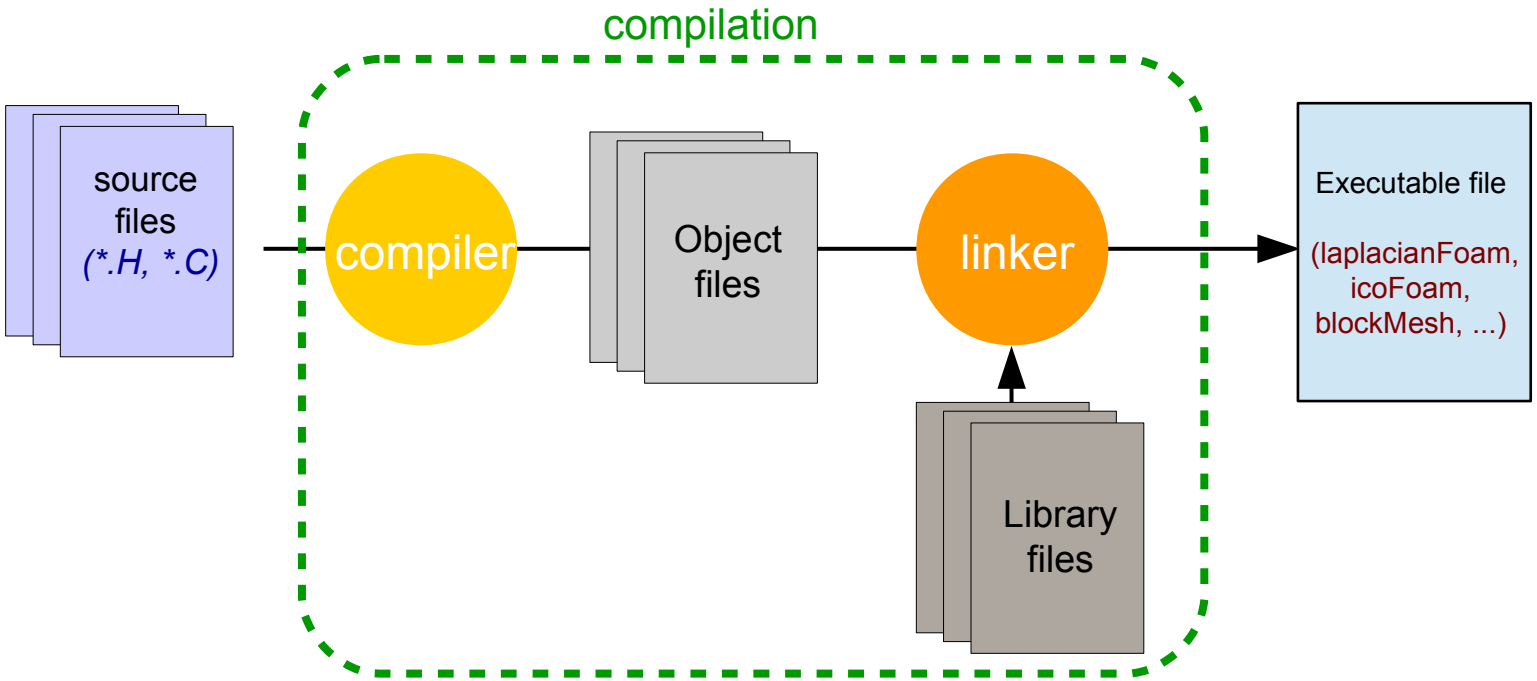
Uses Object Oriented Programming

Design as a metalanguage (API) to perform differential operations on fields defined on non-structured grids.

What level of programming profiency do I need?



Compilation of a source code



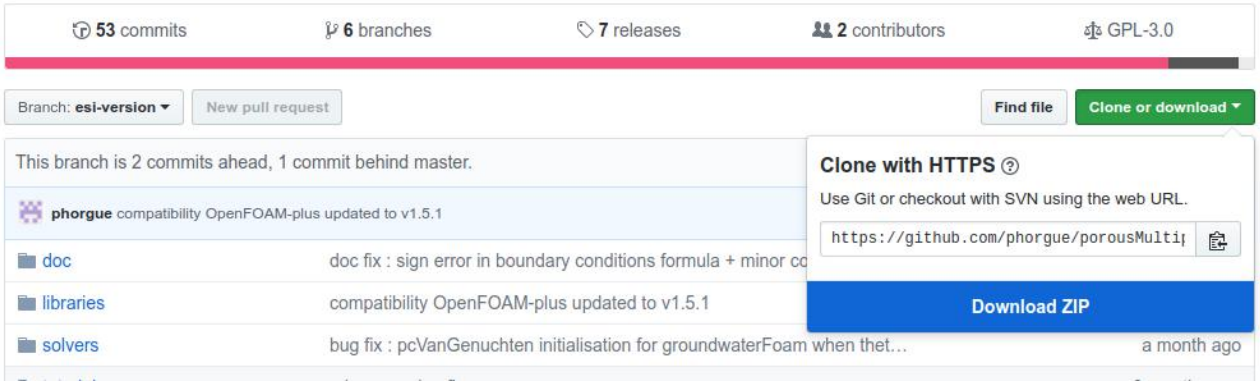
- In OpenFOAM®, the compilation is performed with the command **wmake**.
- **wmake** requires two input files:
  - *Make/files* (list of the source files to compile, name and location of the executable,
  - *Make/options* (list of the libraries to link).
- The command **wclean** is used to remove the object files and reset the compilation process.
- Compilation errors = problem in the source code, problem during the link.

#7 – How to install *porousMultiphaseFoam*? (1/2)

Objectives:

- download and install an OpenFOAM® program developed by a third party.
- Run two-phase flow simulations in porous media at Darcy's scale with *porousMultiphaseFoam* (Horgue et al. 2015).

1. Go to https://github.com/phorgue/porousMultiphaseFoam/tree/esi-version and download the source code



2. Copy and unzip the archive in your working directory (*applications/solvers*)

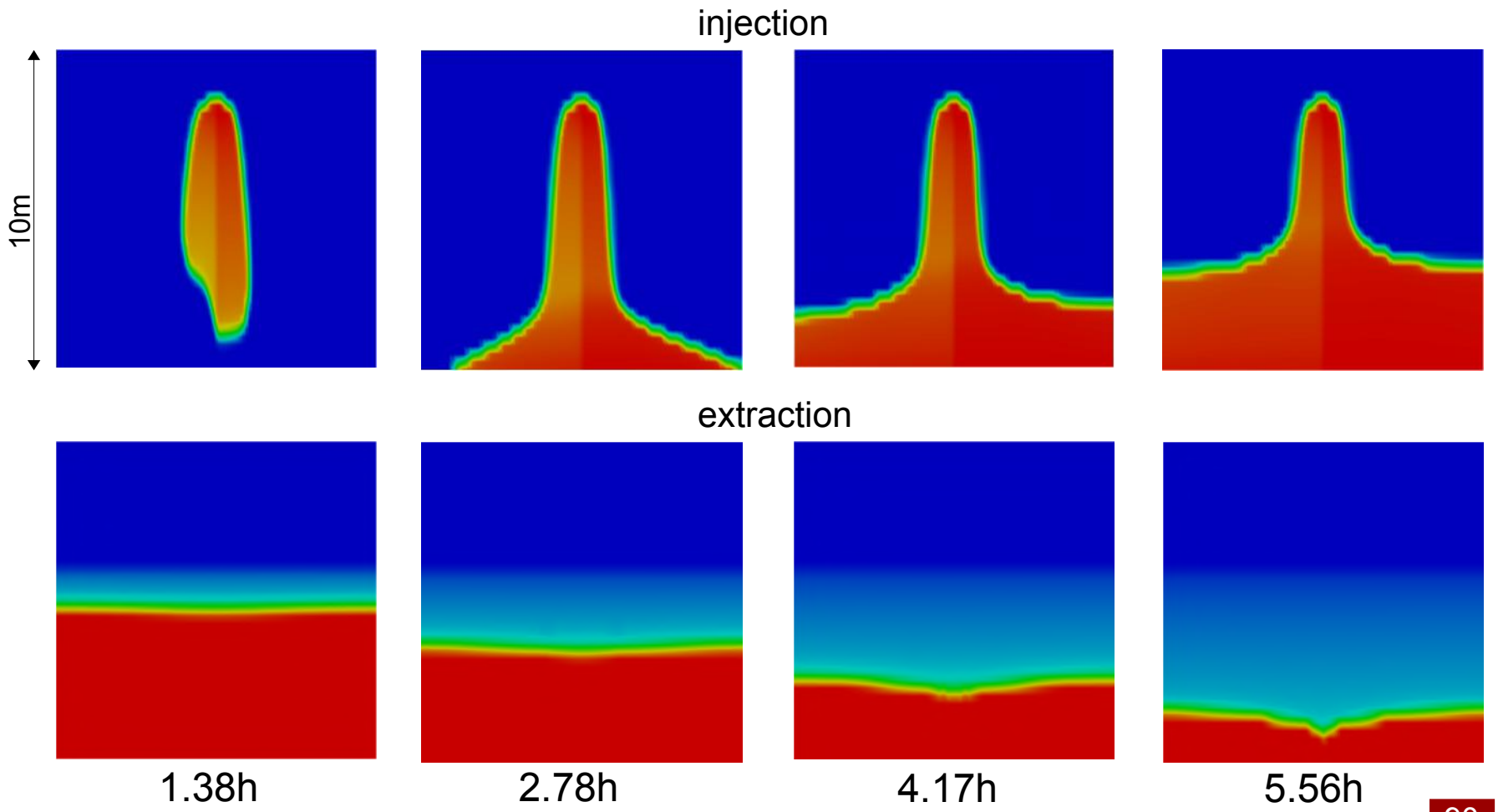
3. Go to the directory and compile the programs

```
$ cd  $WM_PROJECT_USER_DIR/applications/solvers/porousMultiphaseFoam-esi-version
$ ls
$ ./Allwmake
```

Script calling **wmake** for all the programs of the *porousMultiphaseFoam* toolbox

#7 – How to install *porousMultiphaseFoam*? (2/2)

```
$ cp    -r   tutorials/    $FOAM_RUN/porousMultiphaseFoam
$ run
$ cd   porousMultiphaseFoam/impesFoam-tutorials/injectionExtraction/injection
$ ./run
$ paraFoam
```



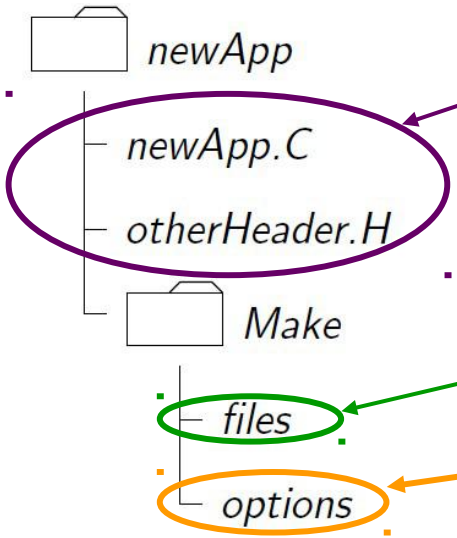
General structure of an application

- Create the directory for your personnal programs (this step needs to be done only once)

```
$ mkdir   -p   $WM_PROJECT_USER_DIR/applications/
```

- Create the structure of your first program

```
$ cd   $WM_PROJECT_USER_DIR/applications/
$ cd   foamNewApp  myFirstProgram
$ cd   myFirstProgram
$ ls
```



Source code

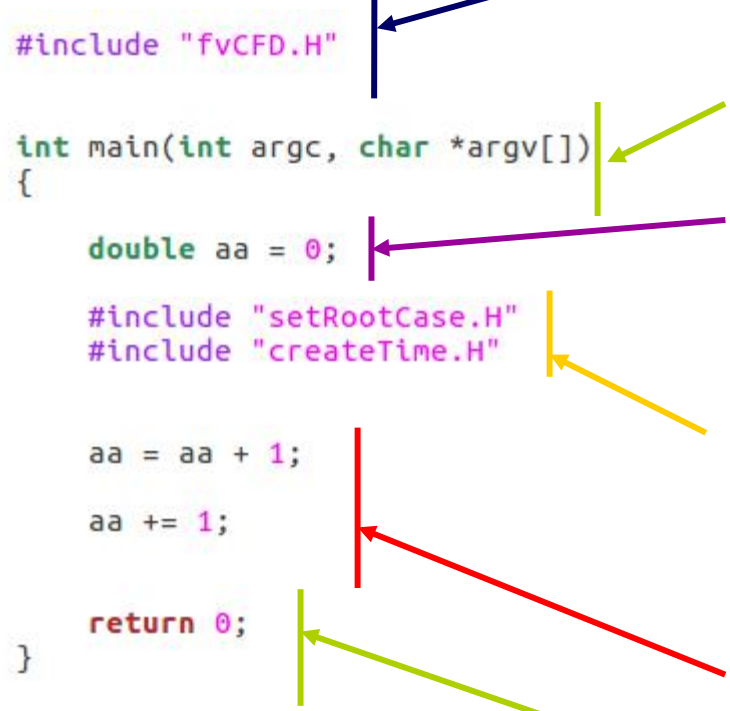
The file with the .C extension is the main file

List of files that will be compiled

Link to the OpenFOAM® librairies used in this program

Structure of a C++ code

```
$ gedit    myFirstProgram.C
```



Header : declaration of function and classes.

Use to call the OpenFOAM® librairies.

Begining of the main.

All variables have to be declared with types and to be initialized.

Include snippets of code written in another file to improve the readability of the code. In OpenFOAM®, the declaration of variables are often written in a separate file.

Operations, loops, if statement, I/O...

End of the main function.

```
$ wmake
```

The program *myFirstProgram* is created after the first compilation of the source code. You can now execute the program in a terminal

Loop, if statement, output screen

```
$ gedit    myFirstProgram.C
```

```
#include "fvCFD.H"

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

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

    double a  = 0;
    int    n  = 1000;
    scalar b  = 1.6;

    for (int i=1; i<=n ; i++)
    {
         a += 1./(i*i);
    }

    Info<< "value of a after "<<n
        <<" iterations = "<< a << nl << endl;


    if(a >= b)
    {
        Info<< "a is greater than "<<b << nl << endl;
    }

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


    return 0;
}
```

```
$ wmake
```

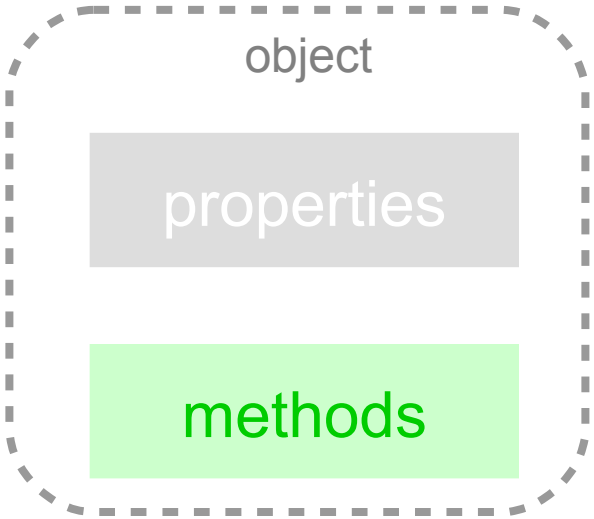
Open another terminal and execute your new program

```
$ run
$ cp   -r   Exo1   myFirstProgram
$ cd   myFirstProgram
$ rm   -r  0.*   [1-9]*
$ myFirstProgram
```

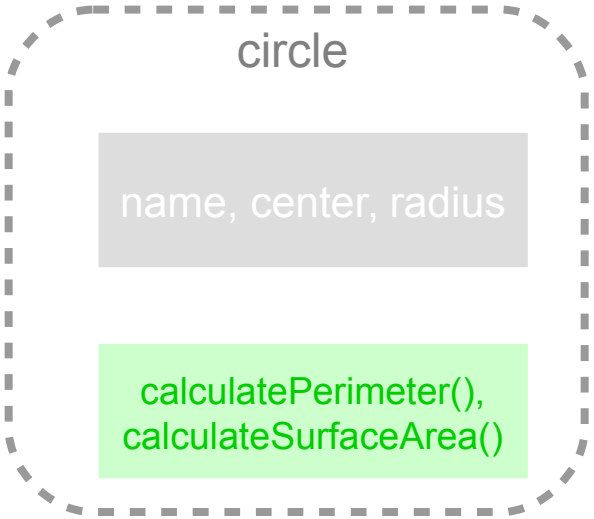
```
Create time

value of a after 1000 iterations = 1.64393

a is greater than 1.6

End
```

Object Oriented Programming

- In C++, you can create new “type” (class)
- A variable from a class is called an object



Example:



- Every objects have to be initialized (“constructed”). There are several ways of initializing an object. Examples:

```
circle circleA (“circleA”, (0 0 0), 3.4);
circle circleB (circleA);
```

- A method can be called at any time in the code:

```
circleA.calculatePerimeter();
```

- Examples of class in OpenFOAM: *volScalarField, dimensionedScalar, fvMesh, fvScalarMatrix, IOdictionary...*

The *dimensionedScalar* object

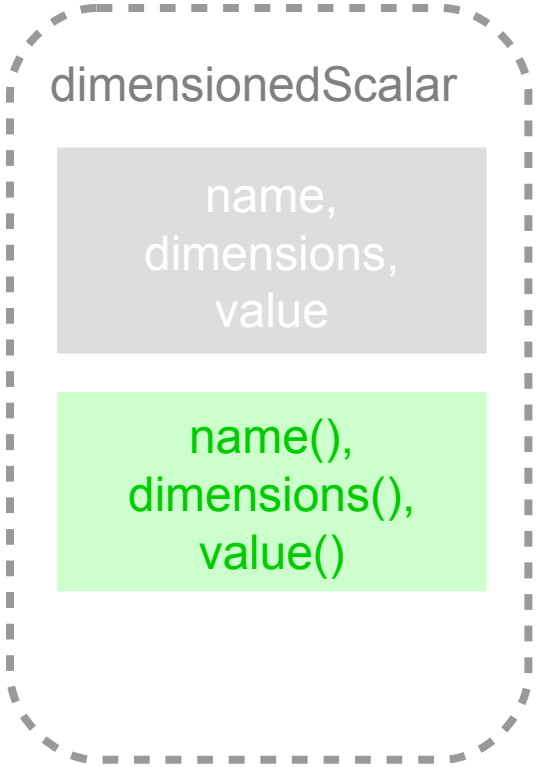
```
$ cd    $WM_PROJECT_USER_DIR/applications
$ foamNewApp    mySecondProgram
$ cd    mySecondProgram
$ gedit   mySecondProgram.C
```

```
#include "fvCFD.H"

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

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

    dimensionedScalar A ("A", dimensionSet(0,1,0,1,1,0,0), 3.4);


    Info<< " name = "       << A.name()         <<nl
        << " dimensions = " << A.dimensions()   <<nl
        << " value = "      << A.value()        <<nl
        << endl;


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

    return 0;
}
```

```
$ wmake
```

Open another terminal and execute your new program

```
$ run
$ cp   -r   myFirstProgram mySecondProgram
$ cd   mySecondProgram
$ mySecondProgram
```




```
name = A
dimensions = [0 1 0 1 1 0 0]
value = 3.4

End
```

Operations and *dimensionedScalar*

```
#include "fvCFD.H"

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

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

    dimensionedScalar A ("A", dimensionSet(0,1,0,1,1,0,0), 3.4);
    dimensionedScalar B ("B", dimensionSet(0,1,0,1,1,0,0), 5.0);
    dimensionedScalar C ("C", dimensionSet(0,1,-1,0,0,0,0), 5.0);

    scalar alpha =10.;

    Info<< "A+B name = "        << (A+B).name()         <<nl
        << "A+B dimensions = "  << (A+B).dimensions()   <<nl
        << "A+B value = "       << (A+B).value()        <<nl
        << endl;

    Info<< "A*C name = "        << (A*C).name()         <<nl
        << "A*C dimensions = "  << (A*C).dimensions()   <<nl
        << "A*C value = "       << (A*C).value()        <<nl
        << endl;

    Info<< "alpha*A name = "        << (alpha*A).name()         <<nl
        << "alpha*A dimensions = "  << (alpha*A).dimensions()   <<nl
        << "alpha*A value = "       << (alpha*A).value()        <<nl
        << endl;

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

    return 0;
}
```

```
A+B name = (A+B)
A+B dimensions = [0 1 0 1 1 0 0]
A+B value = 8.4

A*C name = (A*C)
A*C dimensions = [0 2 -1 1 1 0 0]
A*C value = 17

alpha*A name = (10*A)
alpha*A dimensions = [0 1 0 1 1 0 0]
alpha*A value = 34

End
```

Forbidden operations and *dimensionedScalar*

You can only add *dimensionedScalar* with the same units. The following code compiles but...

```
dimensionedScalar A ("A", dimensionSet(0,1,0,1,1,0,0), 3.4);
dimensionedScalar C ("C", dimensionSet(0,1,-1,0,0,0,0), 5.0);

Info<< "A+C name = "       << (A+C).name()       <<nl
    << "A+C dimensions = " << (A+C).dimensions()  <<nl
    << "A+C value = "      << (A+C).value()      <<nl
    << endl;
```

… leads to an error at the execution of the program.

```
--> FOAM FATAL ERROR:
LHS and RHS of + have different dimensions
    dimensions : [0 1 0 1 1 0 0] + [0 1 -1 0 0 0 0]


    From function Foam::dimensionSet Foam::operator+(const Foam::dimensionSet&,
const Foam::dimensionSet&)
    in file dimensionSet/dimensionSet.C at line 497.

FOAM aborting

#0  Foam::error::printStack(Foam::Ostream&) at ??:?
#1  Foam::error::abort() at ??:?
#2  Foam::operator+(Foam::dimensionSet const&, Foam::dimensionSet const&) at ??:
?
#3  Foam::dimensioned<double> Foam::operator+<double>(Foam::dimensioned<double>
const&, Foam::dimensioned<double> const&) at ??:?
#4  ? at ??:?
#5  __libc_start_main in "/lib/x86_64-linux-gnu/libc.so.6"
#6  ? at ??:?
Aborted (core dumped)
```

You can add *dimensionedScalar* with a *scalar* only if the units of the *dimensionedScalar* are dimensionSet(0,0,0,0,0,0,0).

The *dimensionedScalar*'s constructors

```
dimensionedScalar A ("A", dimensionSet(0,1,0,1,1,0,0), 3.4);
dimensionedScalar B ("B", dimensionSet(0,1,0,1,1,0,0), 5.0);
scalar alpha = 10.;

dimensionedScalar C (A);

Info<< "C name = "        << C.name()       <<nl
    << "C dimensions = "  << C.dimensions() <<nl
    << "C value = "       << C.value()      <<nl
    << endl;

dimensionedScalar D (A+B);

Info<< "D name = "        << D.name()       <<nl
    << "D dimensions = "  << D.dimensions() <<nl
    << "D value = "       << D.value()      <<nl
    << endl;

dimensionedScalar E ("E", A+B);

Info<< "E name = "        << E.name()       <<nl
    << "E dimensions = "  << E.dimensions() <<nl
    << "E value = "       << E.value()      <<nl
    << endl;

dimensionedScalar F (alpha*A);

Info<< "F name = "        << F.name()       <<nl
    << "F dimensions = "  << F.dimensions() <<nl
    << "F value = "       << F.value()      <<nl
    << endl;

dimensionedScalar G ("G", 0.*alpha*A);

Info<< "G name = "        << G.name()       <<nl
    << "G dimensions = "  << G.dimensions() <<nl
    << "G value = "       << G.value()      <<nl
    << endl;
```

```
C name = A
C dimensions = [0 1 0 1 1 0 0]
C value = 3.4

D name = (A+B)
D dimensions = [0 1 0 1 1 0 0]
D value = 8.4

E name = E
E dimensions = [0 1 0 1 1 0 0]
E value = 8.4

F name = (10*A)
F dimensions = [0 1 0 1 1 0 0]
F value = 34

G name = G
G dimensions = [0 1 0 1 1 0 0]
G value = 0

End
```

Construct *dimensionedScalar* from input files (1/2)

```
$ cd    $WM_PROJECT_USER_DIR/applications
$ foamNewApp    myThirdProgram
$ cd    myThirdProgram
$ gedit   myThirdProgram.C
```

```C++
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"

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

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

    dimensionedScalar A
    (
        transportProperties.lookup("A")
    );

    Info<< "A name = "       << A.name()       << nl
        << "A dimensions = " << A.dimensions() << nl
        << "A value = "      << A.value()      << nl
        << endl;


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

    return 0;
}
```

Create the "*runTime*" object

Create the "*mesh*" object

Creation of the "*transportProperties*" object. It is an *IOdictionary* used to load the *constant/transportProperties* file.

Creation of the "*transportProperties*" object. It is an *IOdictionary* used to load the *constant/transportProperties* file.

```
$ wmake
```

Construct *dimensionedScalar* from input files (2/2)

```
$ run
$ cp   -r   myFirstProgram myThirdProgram
$ cd   myThirdProgram
$ blockMesh
$ gedit  constant/transportProperties
```

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

A         A  [0 2 -1 0 0 0 0]    0.01;


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

```
$ myThirdProgram
```

```
Create time

Create mesh for time = 0

Reading transportProperties

A name = A
A dimensions = [0 2 -1 0 0 0 0]
A value = 0.01

End
```

```
$ gedit  constant/transportProperties
```

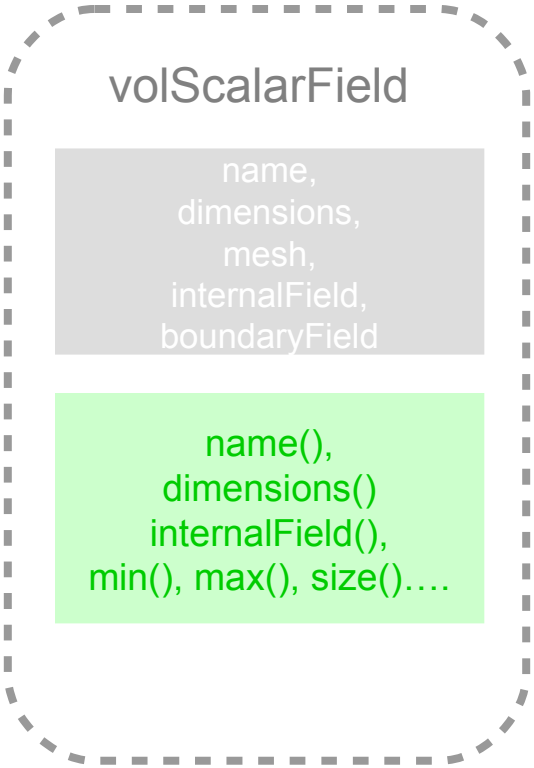
```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * *

A         B  [1 3 -1 0 0 0 0]    15.4;


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

```
$ myThirdProgram
```

```
Create time

Create mesh for time = 0

Reading transportProperties

A name = B
A dimensions = [1 3 -1 0 0 0 0]
A value = 15.4

End
```

The *volScalarField* object

volScalarField = field of scalars defined at the cells center (Example : temperature, concentration, pressure...)



- A *volScalarField* is related to a mesh
- It has a name, units
- Value defined on every cells
- *volScalarField* also contains the boundary conditions,
- *volScalarField* can be added, multiplied, subtracted, divided...
- Geometric differential operators (laplacian, gradient, ddt, divergence…) can be applied to volScalarField objects both explicitly and implicitly.

Examples of basic operations :

volScalarField * volScalarField

volScalarField + volScalarField

dimensonedScalar * volScalarField

dimensonedScalar + volScalarField

How to construct a *volScalarField*?

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

The *volScalarField p* is constructed from

- An input file (IOobject) located in *0/p*
  - Name and units are defined in *0/p*
  - T will be written at each time step in the corresponding folder (*runTime.timeName()*)
  - The boundary conditions are specified in *0/p*
- A *mesh.*

```
volScalarField G (p);
```

G is constructed as a clone of *p*. It has the same name, units, values, boundary conditions.

```
volScalarField Y ("Y", p);
```

Y is constructed as a clone of p but with a different name. It has the same units, values and boundary conditions.

```
volScalarField C ("C", 0.*p);
```

C is constructed as a clone of *p* with a different name and zero values everywhere. It has the same units and boundary conditions

```
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("T",dimensionSet(0,0,0,1,0,0,0),296)
);
```

The *volScalarField T* is not read from an input file. The last argument is an *dimensionedScalar* that provides a name, units and an initial value for T.

Other fields

<Type>=Scalar or Vector or Tensor

dimensioned<Type>

vol<Type>Field

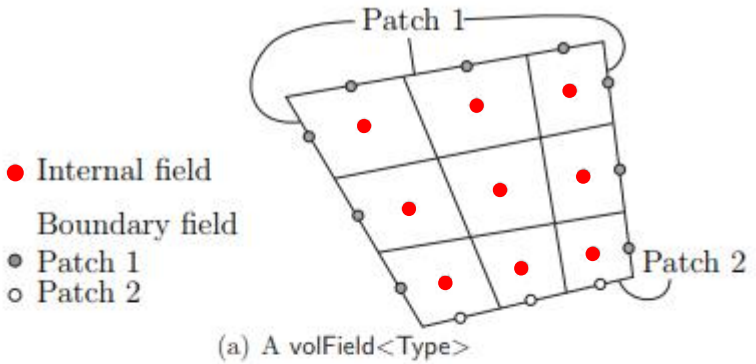
- Value located at the cell center
- Used for FVM calculation

surface<Type>Field

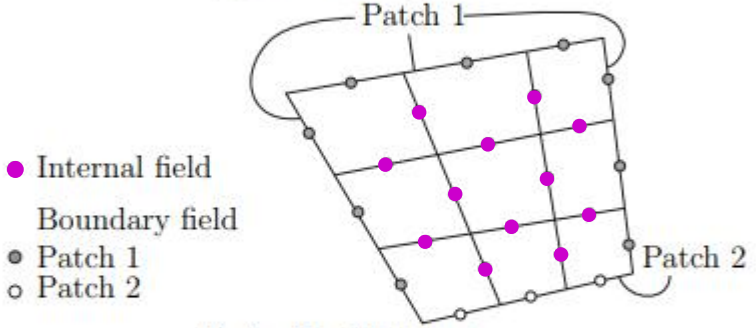
- Value located at the face center of the cells.
- Used to defined fluxes

point<Type>Field

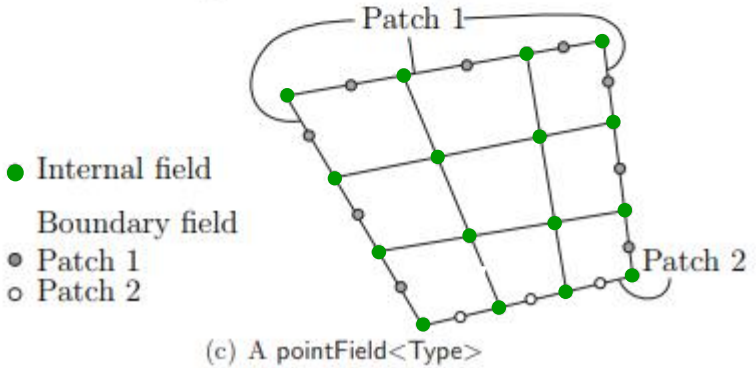
- Value located on the vertices of a cell.
- Used to move the grid



(a) A volField<Type>



(b) A surfaceField<Type>



(c) A pointField<Type>

(source: Programmer guide)

Algebraic tensor operation in OpenFOAM® (1/3)

Operation	Comment	Mathematical Description	Description in OpenFOAM
Addition		$\mathbf{a} + \mathbf{b}$	`a + b`
Subtraction		$\mathbf{a} - \mathbf{b}$	`a - b`
Scalar multiplication		$s\mathbf{a}$	`s * a`
Scalar division		$\mathbf{a}/s$	`a / s`
Outer product	rank $\mathbf{a}, \mathbf{b} >= 1$	$\mathbf{ab}$	`a * b`
Inner product	rank $\mathbf{a}, \mathbf{b} >= 1$	$\mathbf{a} \cdot \mathbf{b}$	`a & b`
Double inner product	rank $\mathbf{a}, \mathbf{b} >= 2$	$\mathbf{a} : \mathbf{b}$	`a && b`
Cross product	rank $\mathbf{a}, \mathbf{b} = 1$	$\mathbf{a} \times \mathbf{b}$	`a ^ b`
Square		$\mathbf{a}^2$	`sqr(a)`
Magnitude squared		$\lvert\mathbf{a}\rvert^2$	`magSqr(a)`
Magnitude		$\lvert\mathbf{a}\rvert$	`mag(a)`
Power	$n = 0, 1, ..., 4$	$\mathbf{a}^n$	`pow(a,n)`
Component average	$i = 1, ..., N$	$\overline{a_i}$	`cmptAv(a)`
Component maximum	$i = 1, ..., N$	$\max(a_i)$	`max(a)`
Component minimum	$i = 1, ..., N$	$\min(a_i)$	`min(a)`
Scale		$\text{scale}(\mathbf{a},\mathbf{b})$	`scale(a,b)`
Geometric transformation	transforms $\mathbf{a}$ using tensor $\mathbf{T}$		`transform(T,a)`

$\mathbf{a}, \mathbf{b}$ are tensors of arbitrary rank unless otherwise stated
s is a scalar, N is the number of tensor components

(source: Programmer guide)

Algebraic tensor operation in OpenFOAM® (2/3)

Operations exclusive to tensors of rank 2

Operation	Comment	Mathematical Description	Description in OpenFOAM
Transpose		$\mathbf{T}^T$	`T.T()`
Diagonal		$\text{diag}\,\mathbf{T}$	`diag(T)`
Trace		$\text{tr}\,\mathbf{T}$	`tr(T)`
Deviatoric component		$\text{dev}\,\mathbf{T}$	`dev(T)`
Symmetric component		$\text{symm}\,\mathbf{T}$	`symm(T)`
Skew-symmetric component		$\text{skew}\,\mathbf{T}$	`skew(T)`
Determinant		$\det\mathbf{T}$	`det(T)`
Cofactors		$\text{cof}\,\mathbf{T}$	`cof(T)`
Inverse		$\text{inv}\,\mathbf{T}$	`inv(T)`
Hodge dual		$*\mathbf{T}$	`*T`

$\mathbf{a}, \mathbf{b}$ are tensors of arbitrary rank unless otherwise stated
s is a scalar, N is the number of tensor components

(source: Programmer guide)

Algebraic tensor operation in OpenFOAM® (3/3)

Operations exclusive to scalars			
Sign (boolean)		$\mathrm{sgn}(s)$	`sign(s)`
Positive (boolean)		$s >= 0$	`pos(s)`
Negative (boolean)		$s < 0$	`neg(s)`
Limit	n scalar	$\mathrm{limit}(s, n)$	`limit(s,n)`
Square root		$\sqrt{s}$	`sqrt(s)`
Exponential		$\exp s$	`exp(s)`
Natural logarithm		$\ln s$	`log(s)`
Base 10 logarithm		$\log_{10} s$	`log10(s)`
Sine		$\sin s$	`sin(s)`
Cosine		$\cos s$	`cos(s)`
Tangent		$\tan s$	`tan(s)`
Arc sine		$\mathrm{asin}\, s$	`asin(s)`
Arc cosine		$\mathrm{acos}\, s$	`acos(s)`
Arc tangent		$\mathrm{atan}\, s$	`atan(s)`
Hyperbolic sine		$\sinh s$	`sinh(s)`
Hyperbolic cosine		$\cosh s$	`cosh(s)`
Hyperbolic tangent		$\tanh s$	`tanh(s)`
Hyperbolic arc sine		$\mathrm{asinh}\, s$	`asinh(s)`
Hyperbolic arc cosine		$\mathrm{acosh}\, s$	`acosh(s)`
Hyperbolic arc tangent		$\mathrm{atanh}\, s$	`atanh(s)`
Error function		$\mathrm{erf}\, s$	`erf(s)`
Complement error function		$\mathrm{erfc}\, s$	`erfc(s)`
Logarithm gamma function		$\ln \Gamma s$	`lgamma(s)`
Type 1 Bessel function of order 0		$\mathrm{J}_0\, s$	`j0(s)`
Type 1 Bessel function of order 1		$\mathrm{J}_1\, s$	`j1(s)`
Type 2 Bessel function of order 0		$\mathrm{Y}_0\, s$	`y0(s)`
Type 2 Bessel function of order 1		$\mathrm{Y}_1\, s$	`y1(s)`

a, **b** are tensors of arbitrary rank unless otherwise stated
s is a scalar, N is the number of tensor components

(source: Programmer guide)

Explicit operations on a geometric field: fvc

- Every fonctions fvc::function() perform an explicit operation on a geometric field
- The same function can be applied to a scalar, vector or tensor field
- The discretization schemes for the differential operators are specified in *system/fvSchemes*
- fvc= Finite Volume Calculus

```
dimensionedScalar DT ( .... );
volScalarField T ( .... );
volVectorField U ( .... );


volScalarField ddtT ("ddtT", fvc::ddt(T));

volScalarField laplacianT ("laplacianT", fvc::laplacian(T));

volScalarField laplacianDT ("laplacianDT", fvc::laplacian(DT,T));

volVectorField gradT ("gradT", fvc::grad(T));

surfaceScalarField Tf ("Tf", fvc::interpolate(T));

volTensorField gradU ("gradU", fvc::grad(U));

volTensorField strain ("strain", gradU+gradU.T());

surfaceScalarField phi ("phi", fvc::interpolate(U) & mesh.Sf());

volScalarField divPhi ("divPhi", fvc::div(phi));

volScalarField divPhiT ("divPhiT", fvc::div(phi,T));
```

Implicit operations on a geometric field: fvm

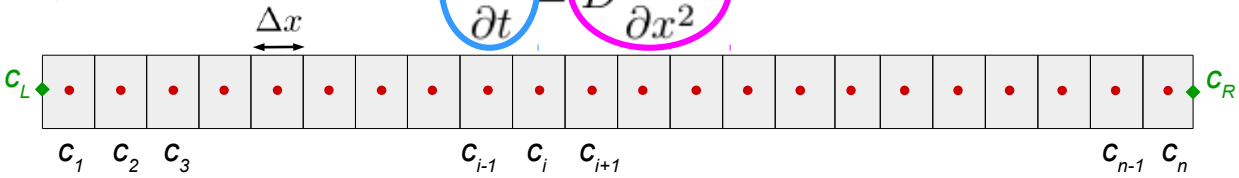
Space discretization

operators discretization

Build the matrix

Solve the linear algebra

$$c(x=0)=c_L \qquad \frac{\partial c}{\partial t}=D\frac{\partial^2 c}{\partial x^2} \qquad c(x=H)=c_R$$



$$\Delta x\frac{c_i^{k+1}-c_i^k}{\Delta t}=D\frac{c_{i-1}^{k+1}-2c_i^{k+1}-c_{i+1}^{k+1}}{\Delta x}$$

$$\underbrace{\frac{\Delta x}{\Delta t}\begin{bmatrix}1 & & & & \\ & 1 & & 0 & \\ & & \ddots & & \\ & 0 & & 1 & \\ & & & & 1\end{bmatrix}}_{\mathsf{A}_{ddt}}\underbrace{\begin{bmatrix}c_1^{k+1}\\ c_2^{k+1}\\ \vdots\\ c_{n-1}^{k+1}\\ c_n^{k+1}\end{bmatrix}}_{\boldsymbol{x}^{k+1}}=\underbrace{\frac{\Delta x}{\Delta t}\begin{bmatrix}c_1^{k}\\ c_2^{k}\\ \vdots\\ c_{n-1}^{k}\\ c_n^{k}\end{bmatrix}}_{\boldsymbol{y}_{ddt}}$$

$$\underbrace{\frac{D}{\Delta x}\begin{bmatrix}-3 & 1 & & & 0\\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1\\ 0 & & & 1 & -3\end{bmatrix}}_{\mathsf{A}_{laplacian}}\underbrace{\begin{bmatrix}c_1^{k+1}\\ c_2^{k+1}\\ \vdots\\ c_{n-1}^{k+1}\\ c_n^{k+1}\end{bmatrix}}_{\boldsymbol{x}^{k+1}}=\underbrace{\begin{bmatrix}-2\frac{c_L}{\Delta x^2}\\ 0\\ \vdots\\ 0\\ -2\frac{c_R}{\Delta x^2}\end{bmatrix}}_{\boldsymbol{y}_{laplacian}}$$

```
fvScalarMatrix TEqn (  fvm::ddt(T) == fvm::laplacian(DT,T)  );
```

$$\boldsymbol{x}=\mathsf{A}^{-1}\boldsymbol{y}$$

```
TEqn.solve();
```

Implicit operations in OpenFOAM®

- Every fonctions fvm::function() perform an implicit differential operation
- fvm::function() returns a fv<Type>Matrix
- fvm= Finite Volume Method
- The same function can be applied to a scalar, vector or tensor field
- The discretization schemes for the differential operators are specified in *system/fvSchemes*
- The linear solver to inverse the matrix are specified in *system/fvSolution*

Examples of differential operations :

```
fvm::ddt(T) == fvm::laplacian(D,T)

fvm::ddt(T) == fvc::laplacian(D,T)

fvm::ddt(T) == fvm::laplacian(D,C)

fvm::ddt(T) == fvc::laplacian(D,C)
```

Where is the source code of a solver?

- OpenFOAM® can be seen as an easily customizable toolbox.

- 1 solver = 1 program
  (for instance, the heat equation is solved using the program *laplacianFoam*)

- Where are the solvers in OpenFOAM®?

```
$ cd    $FOAM_APP/solvers
$ ls
```

- The solvers are sorted by type (basic, heat transfer, combustion, incompressible, multiphase….). Note that the tutorials have a similar organization.

- For example, *laplacianFoam* is in /basic

```
$ cd    basic/laplacianFoam
$ ls
```

Behind laplacianFoam: *laplacianFoam.C*

```
#include "fvCFD.H"
#include "fvOptions.H"
#include "simpleControl.H"

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

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

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

    simpleControl simple(mesh);

    #include "createFields.H"
    #include "createFvOptions.H"

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

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

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

        while (simple.correctNonOrthogonal())
        {
            fvScalarMatrix TEqn
            (
                fvm::ddt(T) - fvm::laplacian(DT, T)
             ==
                fvOptions(T)
            );

            fvOptions.constrain(TEqn);
            TEqn.solve();
            fvOptions.correct(T);
        }

        #include "write.H"

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

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

    return 0;
}
```

$ gedit laplacianFoam.C

Headers to call the OpenFOAM® librairies

Include the shared snippet of code

$$\frac{\partial T}{\partial t} = \nabla \cdot (D_T \nabla T)$$

Creation of the matrix

fvm:: implicit terms

fvc:: explicit terms

the variable T and DT are declared in *createFields.H*

Behind laplacianFoam: *createFields.H*

$ gedit createFields.H

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

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

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

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

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

dimensionedScalar DT
(
    "DT",
    dimArea/dimTime,
    transportProperties
);
```

The temperature field T is declared as an instance of the object *volScalarField*

- It is a scalar field with values defined at the cell center
- It must be read at the initial time step
- Dimensions (units) are defined in *0/T*
- T will be written at each time step in the corresponding folder (runTime.timeName())
- This object also includes boundary conditions that are specified in *0/T*

The dictionary *transportProperties* is loaded from the input file *constant/transportProperties*

Declaration of the variable DT

Its value is defined in the input file *constant/transportProperties*

How to quickly program an OpenFOAM® solver?

- The main idea is not to start a program from scratch but to customize existing OpenFOAM® programs,

- When editing a source code, you can copy/paste snippets of code to avoid errors and save time (especially for the declaration of vol<Type>Field),

- Compile as often as you can,

- Read and try to understand compiling errors,

- More advanced snippets of code:

```
$ cd    $FOAM_APP/test/
```

Create the directory for your personnal programs (this step needs to be done only once)

```
$ mkdir    -p    $WM_PROJECT_USER_DIR/applications/solvers/
```

#8 – Program a “Darcy” solver (1a/7)

- Objective: develop a program that solves the velocity and pressure in a saturated porous medium using Darcy's law.

$$\nabla.\mathbf{U} = 0 \qquad (1)$$

$$\mathbf{U} = -\frac{k}{\mu}\nabla p \qquad (2)$$

- How to solve this mathematical problem? The diffusion equation for the pressure field is obtained by combining equation *(1)* and *(2)*:

$$\nabla.\frac{k}{\mu}\nabla p = 0$$

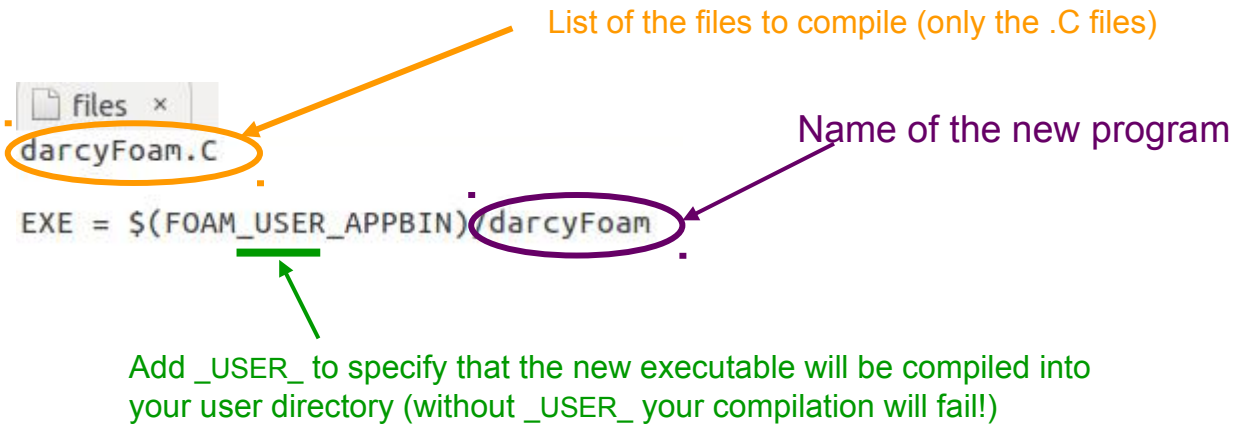
- This equation is closed to the heat diffusion equation. Hence, we are going to program our own solver on the basis of the existing *laplacianFoam*. To do so, we copy *laplacianFoam* in our workspace.

```
$ cd    $WM_PROJECT_USER_DIR/applications/solvers/
$ cp   -r    $FOAM_APP/solvers/basic/laplacianFoam   darcyFoam
```

#8 – Program a “Darcy” solver (1b/7)

- Once the *laplacianFoam* solver has been copied into the user directory, we rename the main file and edit the *Make/files*:

```
$ cd      darcyFoam
$ mv      laplacianFoam.C      darcyFoam.C
$ gedit   Make/files
```



- We can now clean the previous compilation with *wclean* and compile this new program with *wmake*.

```
$ wclean
$ wmake
```

- At this stage, we have a new program called *darcyFoam* that is am exact copy of *laplacianFoam* (you can run the flange tutorial or #1 with *darcyFoam* instead of *laplacianFoam*)
- It is recommanded to use wmake as often as possible during the programming process.

#8 – Program a “Darcy” solver (2/7)

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

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

volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedVector("U",dimensionSet(0,1,-1,0,0,0,0),vector::zero)
);

Info<< "Reading transportProperties\n" << endl;
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);

Info<< "Reading permeability k\n" << endl;
dimensionedScalar k
(
    transportProperties.lookup("k")
);

Info<< "Reading fluid viscosity mu\n" << endl;
dimensionedScalar mu
(
    transportProperties.lookup("mu")
);
```

$ gedit createFields.H

Declaration of the pressure field *p*

- It is an instance of the object *volScalarField* (scalar field defined at the cells center),
- The file «p» must be read at the frist time step to satisfy the constructor. The initial values and boundary conditions are defined during the loading of *0/p*.
- The file « p » will be written at every output times.

Declaration of the velocity vector field *U*

- It is an instance of the object *volVectorField* (vectpr field defined at the cells center),
- U is not read from a file (even if *0/U* exist)
- To satisfy the constructor of the object *volVectorField*, units and initial values are defined with an additional arguement. By default, the boundary conditions are *zeroGradient*,
- The file « U » will be written at every output times.

Declaration of the fluid viscosity *mu* and the permeability *k*. They will be loaded from *« constant/transportProperties »*

#8 – Program a “Darcy” solver (3/7)

```
#include "fvCFD.H"
#include "simpleControl.H"

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

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

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

    simpleControl simple(mesh);

    #include "createFields.H"

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

    Info<< "\nCalculating pressure distribution\n" << endl;

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

        while (simple.correctNonOrthogonal())
        {
            solve
            (
                fvm::laplacian(k/mu, p)
            );
        }

        U = -k/mu*fvc::grad(p);

        runTime.write();

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

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

    return 0;
}
```

```
$ gedit    darcyFoam.C
```

The pressure field *p* is solved implicitly by a diffusion equation

The velocity vector *U* is deduced from the pressure field using the Darcy's law.

```
$ rm -r    write.H overLaplacianDyMFoam
$ wclean
$ wmake
```

The useless files are removed and the *darcyFoam* executable is then compiled.

#8 – Program a “Darcy” solver (4/7)

```
convertToMeters 1;
Lx 10;
Ly 0.1;
Lz 0.1;

vertices
(
    (0   0   0  )
    ($Lx 0   0  )
    ($Lx $Ly 0  )
    (0   $Ly 0  )
    (0   0   $Lz)
    ($Lx 0   $Lz)
    ($Lx $Ly $Lz)
    (0   $Ly $Lz)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (60 1 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)

        );
    }
    outlet
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (1 5 4 0)
            (3 7 6 2)
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```

Tip: The vertices can be defined using variables Lx, Ly, Lz. It saves time to modify the size of the domain

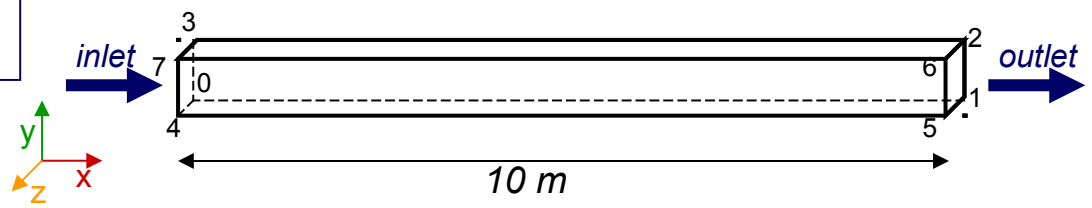
Mesh definition (homogeneous grid with a single cell in the y and z axis since the simulation is 1D)

Faces orthogonal to *y* and *z* axis are defined as «empty» to specify that the simulation is 1D

To prepare this « case » we are going to use the tutorial *laplacianFoam/flange*. The setup will be relatively similar since this latter also solves a diffusion equation.

```
$ run
$ cp -r $FOAM_TUTORIALS/basic/laplacianFoam/flange Exo8
$ cd Exo8
$ rm Allrun Allclean flange.ans
```

We are going to simulate flow in an 1D porous medium:



To save time, we can pick up and modify an existing *blockMeshDict*

```
$ cp $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity/system/blockMeshDict system/.
$ gedit system/blockMeshDict
```

The grid is generated using *blockMesh*

```
$ blockMesh
```

#8 – Program a “Darcy” solver (5a/7)

```
$ mv  0/T   0/p
$ gedit   0/p
```

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

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 1e2;
    }

    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

A pressure drop is imposed between the inlet and the outlet of the computational domain

```
$ constant/transportProperties
```

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

k               k  [0  2  0 0 0 0 0] 1e-09;
mu              mu [1 -1 -1 0 0 0 0] 1e-05;
```

#8 – Program a “Darcy” solver (5b/7)

```
application     darcyFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         1;

deltaT          1;

writeControl    runTime;

writeInterval   1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

```
$ gedit system/controlDict
```

Since *darcyFoam* is a steady-state solver without relaxation factor, only one time step is necessary.

#8 – Program a “Darcy” solver (5c/7)

$ gedit system/fvSchemes

```
ddtSchemes
{
    default         Euler;
}

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

divSchemes
{
    default         none;
}

laplacianSchemes
{
    default         none;
    laplacian((k|mu),p) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}
```

$ gedit system/fvSolution

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

SIMPLE
{
    nNonOrthogonalCorrectors 2;
}
```

#8 – Program a “Darcy” solver (6/7)

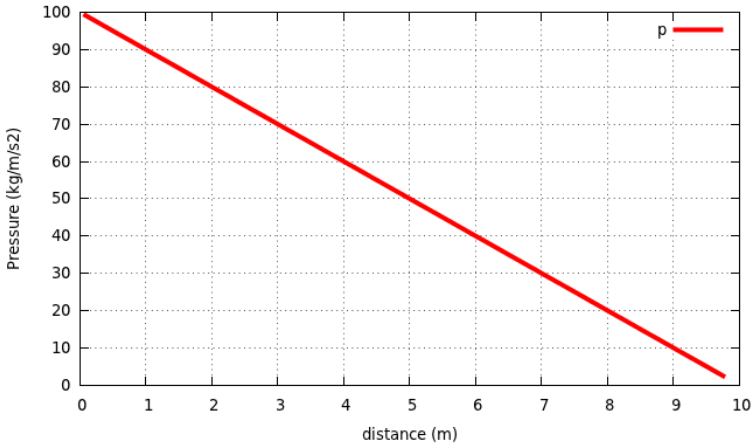
- Run the simulation :

```
$ darcyFoam
```

- Results will be plotted using the *postProcess* utility and the program Gnuplot. As *blockMesh*, the program *postProcess* requires an input dictionary located in */system* :

```
$ cp   $FOAM_ETC/caseDicts/postProcessing/graphs/singleGraph    system/.
$ gedit   system/singleGraph
```

```
\*---------------------------------------------------------

start   (0  0.05 0.005);
end     (10 0.05 0.005);
fields  (U p);

// Sampling and I/O settings
#includeEtc "caseDicts/postProcessing/graphs/sampleDict.cfg"

// Override settings here, e.g.
// setConfig { type midPoint; }

// Must be last entry
#includeEtc "caseDicts/postProcessing/graphs/graph.cfg"

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



- Run the *postProcess* tool:

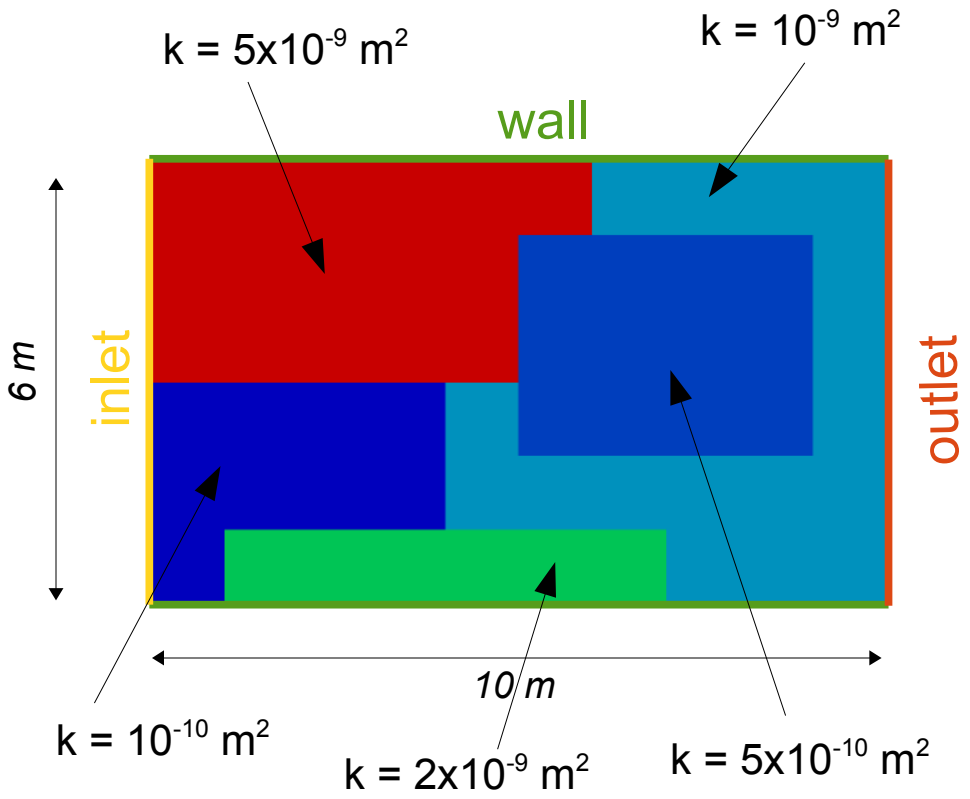
```
$ postProcess -latestTime -func singleGraph
```

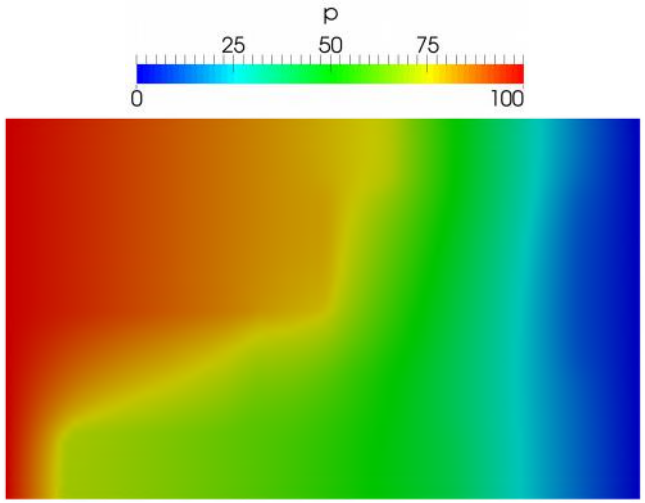
- Plot the pressure field with Gnuplot :

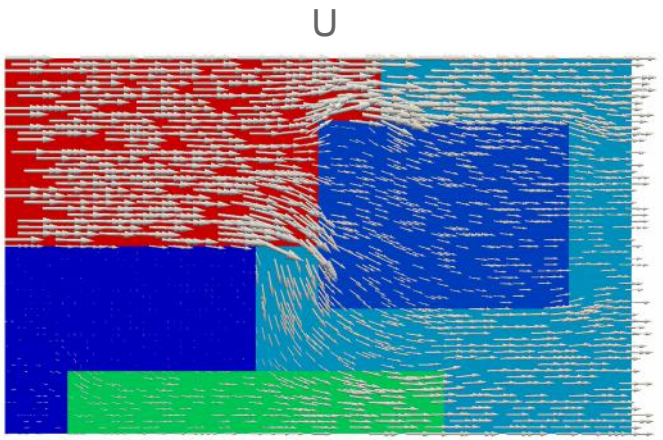
```
$ gnuplot
gnuplot> set xlabel "distance (m)"
gnuplot> set ylabel "Pressure (kg/m/s)"
gnuplot> plot “postProcessing/singleGraph/1/lineX1_p.xy" using 1:2 with lines lw 4 title "p“
```

#8 – Program a “Darcy” solver (7/7)

Exo8Bis: Program a new solver for heterogeneous porous media (*heterogeneousDarcyFoam*) defining the permeability as a *volScalarField* and assigning it different values with the utility *setFields*.







We can get *setFieldsDict* from #4 or:

```
$ cp   $FOAM_UTILITIES/preProcessing/setFields/setFieldsDict  system/.
```

#9 - Heat transfer in porous media (1/7)

- Objective 1: Develop a program that solves heat transfer in a porous medium where the velocity and pressure obey Darcy's law.

$$\nabla.\mathbf{U} = 0 \qquad (1)$$

$$\mathbf{U} = -\frac{k}{\mu}\nabla p \qquad (2)$$

$$\left(\varepsilon\left(\rho C_p\right)_f + \left(1-\varepsilon\right)\left(\rho C_p\right)_s\right)\frac{\partial T}{\partial t} + \left(\rho C_p\right)_f \nabla.\left(\mathbf{U}T\right) = \nabla.\left(D_T \nabla T\right) \qquad (3)$$

- Objective 2: Use probes to plot the temperature evolution vs time at some points of the domain
- Objective 3: Change the discretization schemes

This solver will be based on *darcyFoam (#8)*

```
$ cd    $WM_PROJECT_USER_DIR/applications/solvers/
$ cp   –r    darcyFoam    darcyTemperatureFoam
$ cd    darcyTemperatureFoam
$ mv   darcyFoam.C    darcyTemperatureFoam.C
$ gedit   Make/files
```

files ×

```
darcyTemperatureFoam.C

EXE = $(FOAM_USER_APPBIN)/darcyTemperatureFoam
```

```
$ wclean
$ wmake
```

#9 - Heat transfer in porous media (2a/7)

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

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

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

volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedVector("U", dimensionSet(0,1,-1,0,0,0,0), vector::zero)
);

surfaceScalarField phi ("phi", linearInterpolate(U) & mesh.Sf());

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

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

$ gedit createFields.H

Declaration of the velocity flux *phi*.

- It is a surface field (*U* is projected onto the face of each cell of the grid)
- It is necessary when using the divergence operator (fvm::div(phi,T))
- Can also be declared using #include ''createPhi.H

Declaration of the temperature field *T*. (Do not copy everything manually: copy/paste the declaration of *volScalarField p* and replace *p* by *T*)

$ wmake

#9 - Heat transfer in porous media (2b/7)

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

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


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

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

dimensionedScalar eps
(
    transportProperties.lookup("eps")
);

dimensionedScalar rhoCps
(
    transportProperties.lookup("rhoCps")
);

dimensionedScalar rhoCpf
(
    transportProperties.lookup("rhoCpf")
);


dimensionedScalar k
(
    transportProperties.lookup("k")
);

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

$ gedit createFields.H

Beside the viscosity *mu* and the permeability *k* of the porous medium, we also declare the thermal conductivity *DT,* the porosity *eps* and the heat capacities rhoCps and rhoCpf. They are loaded from the file *« constant/transportProperties »*

$ wmake

#9 - Heat transfer in porous media (3/7)

$ gedit darcyTemperatureFoam.C

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

    while (simple.correctNonOrthogonal())
    {
        solve
        (
            fvm::laplacian(k/mu, p)
        );
    }


    U = -k/mu*fvc::grad(p);


    phi = linearInterpolate(U) & mesh.Sf();


    solve
    (
          (eps*rhoCpf+(1.-eps)*rhoCps)*fvm::ddt(T)
        + rhoCpf*fvm::div(phi,T)
            ==
          fvm::laplacian(DT, T)
    );


    runTime.write();

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

The surface flux *phi* is updated from the new value of the velocity profile *U*.

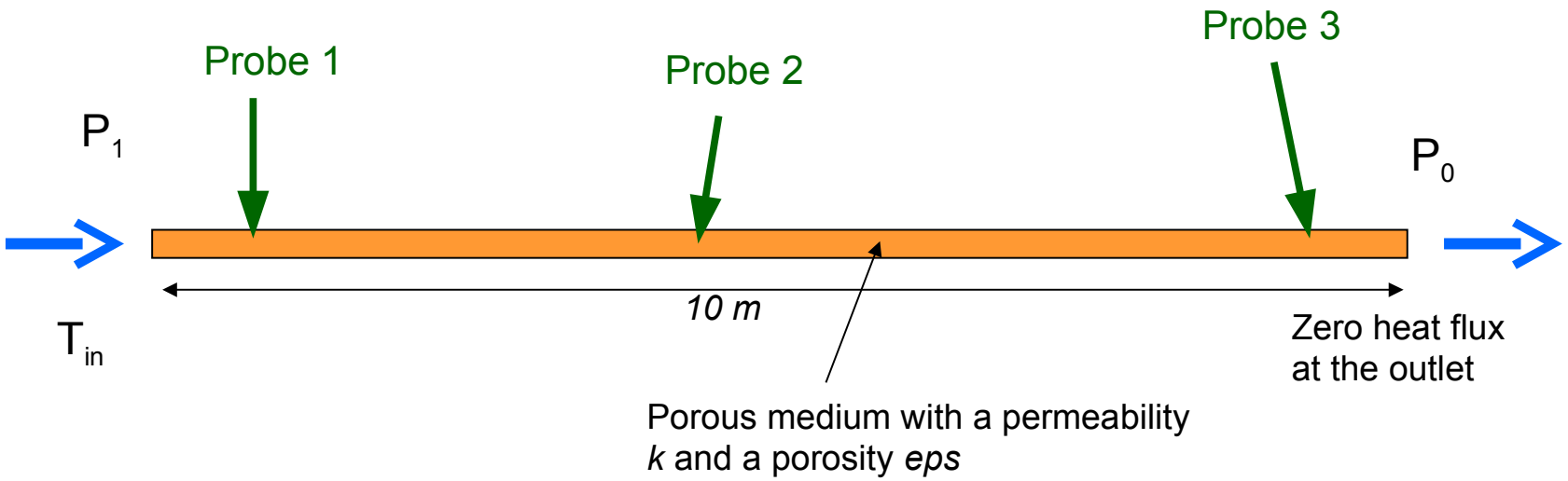
Solve the advection/diffusion equation for the temperature transport

$ wmake

Compilation of *darcyTemperatureFoam*

#9 - Heat transfer in porous media (4/7)

We want to estimate the temperature evolution in a 1D porous medium



To save time, we can adapt the previous exercise (#7) to setup the case

```
$ run
$ cp   –r   Exo8   Exo9
$ cd   Exo9
$ rm    –r    0.*  1*   2*   3*   4*   5*   6*   7*   8*   9*   postProcessing
$ cp   0/p   0/T
$ gedit   0/T
```

#9 - Heat transfer in porous media (5a/7)

$ gedit 0/T

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

internalField   uniform 273;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 573;
    }

    outlet
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit 0/p

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 1e2;
    }

    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

#9 - Heat transfer in porous media (5b/7)

$ gedit constant/transportProperties

```
mu          mu      [ 1 -1 -1 0 0 0 0 ]  1e-05;

k           k       [ 0 2 0 0 0 0 0 ]    1e-09;

eps         eps     [ 0 0 0 0 0 0 0 ]    0.4;

DT          DT      [ 1 1 -3 -1 0 0 0 ]  1e-02;

rhoCps      rhoCps  [ 1 -1 -2 -1 0 0 0 ] 2e4;

rhoCpf      rhoCpf  [ 1 -1 -2 -1 0 0 0 ] 5e3;
```

$ gedit system/fvSolution

```
solvers
{

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

    T
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 2;
}
```

$ gedit system/fvSchemes

```
ddtSchemes
{
    default         Euler;
}

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

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

laplacianSchemes
{
    default         none;
    laplacian((k|mu),p) Gauss linear corrected;
    laplacian(DT,T) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    T               ;
}
```

#9 - Heat transfer in porous media (5c/7)

```
startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         60000;

deltaT          100;

writeControl    runTime;

writeInterval   1000;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

functions
{
    probes
    {
        type            probes;
        functionObjectLibs ("libsampling.so");
        enabled         true;
        outputControl   timeStep;
        outputInterval  1;

        fields
        (
            T
        );

        probeLocations
        (
            ( 2 0.05 0.05)  // Probe 1
            ( 5 0.05 0.05)  // Probe 2
            ( 9 0.05 0.05)  // Probe 3
        );
    }
}
```

$ gedit system/controlDict

The probes are functions that are executed on-the-fly during the simulation.

They allow to record the temperature evolution vs time at the probe location.

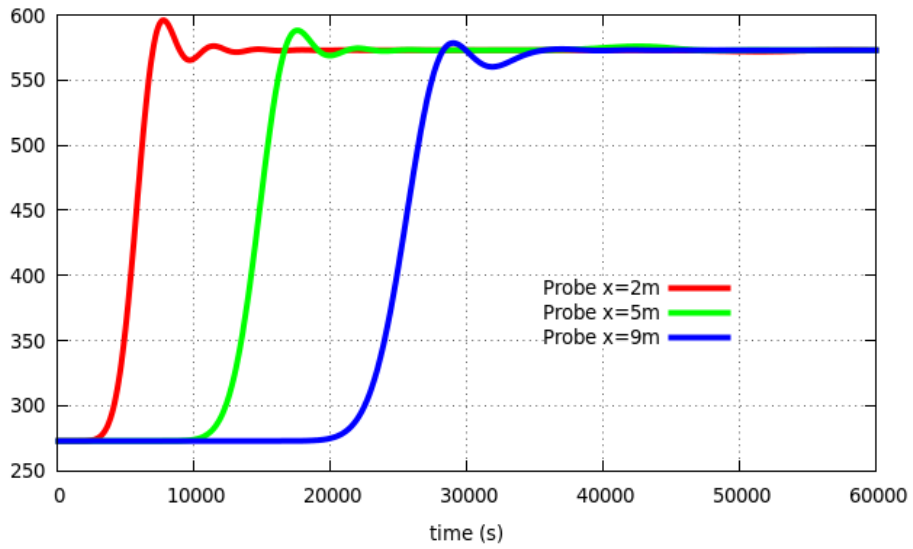
You can specify as many probes as you want.

#9 - Heat transfer in porous media (6/7)

- Run the simulation : `$ darcyTemperatureFoam`
- We are going to plot the probe results with the following gnuplot script `$ gedit plot_probes`

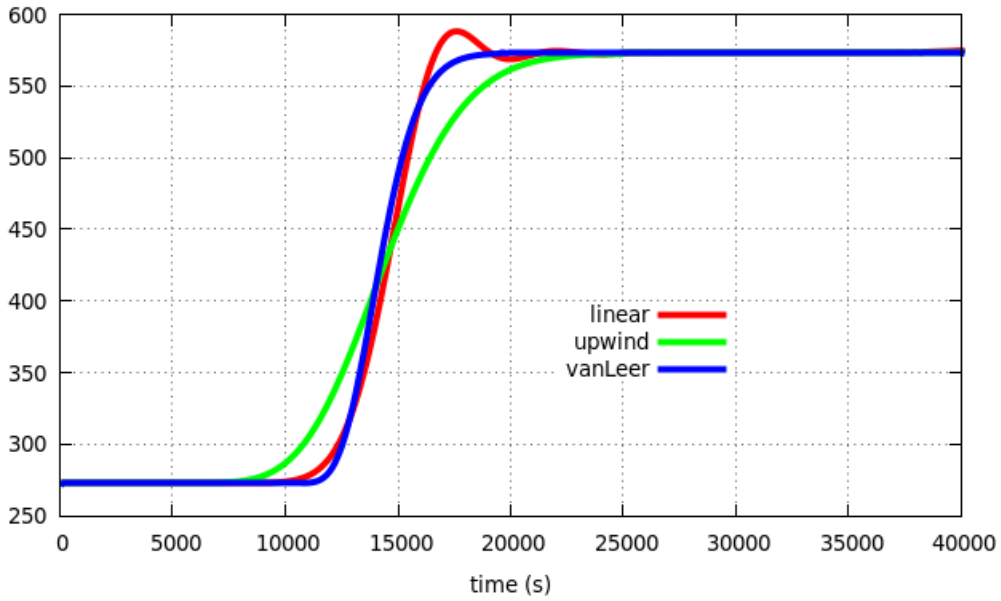
```
set key at 50000, 400
set ylabel "temperature (K)"
set xlabel "time (s)"

plot "postProcessing/probes/0/T" using 1:2 with lines lw 4 title "Probe x=2m" ,\
     "postProcessing/probes/0/T" using 1:3 with lines lw 4 title "Probe x=5m" ,\
     "postProcessing/probes/0/T" using 1:4 with lines lw 4 title "Probe x=9m"
```

`$ gnuplot –persist plot_probes`



#9 - Heat transfer in porous media (7/7)

- Note in the previous simulation some unphysical oscillations at the temperature front. They are due to the numerical scheme used to descretize the convection term. To improve the numerical stability, you can use an upwind scheme or a flux limiter by specifying *Gauss upwind* or *Gauss vanLeer* in *system/fvSchemes* instead of *Gauss linear*.



- The upwind scheme is better than the linear but also more diffusive. The flux limiter schemes are more suitable for this kind of simulation.

- More benchmarks on OpenFOAM® numerical schemes :

http://www.holzmann-cfd.de/index.php/en/numerical-schemes

#10 – Customize boundary conditions (1/4)

Objective: Create customized boundary conditions

- By default, OpenFOAM® can handled a lot of different boundary conditions. Their code source is located in the following directory:

```
$ cd  $FOAM_SRC/finiteVolume/fields/fvPatchFields
$ ls
```

```
basic  constraint  derived  doc  fvPatchField
```

- All these conditions are derived from basic conditions like *fixedValue* (Dirichlet), *fixedGradient* (Neumann) or *mixed*.

```
$ ls derived
```

```
activePressureForceBaffleVelocity  freestream                      pressureDirectedInletOutletVelocity  totalPressure
advective                          freestreamPressure              pressureDirectedInletVelocity        totalTemperature
codedFixedValue                    inletOutlet                     pressureInletOutletParSlipVelocity   translatingWallVelocity
codedMixed                         inletOutletTotalTemperature     pressureInletOutletVelocity          turbulentInlet
cylindricalInletVelocity           interstitialInletVelocity       pressureInletUniformVelocity         turbulentIntensityKineticEnergyInlet
externalCoupledMixed               mappedField                     pressureInletVelocity                uniformDensityHydrostaticPressure
fan                                mappedFixedInternalValue        pressureNormalInletOutletVelocity    uniformFixedGradient
fanPressure                        mappedFixedPushedInternalValue  prghPressure                         uniformFixedValue
fixedFluxPressure                  mappedFixedValue                prghTotalPressure                    uniformInletOutlet
fixedInternalValueFvPatchField     mappedFlowRate                  rotatingPressureInletOutletVelocity  uniformJump
fixedJump                          mappedVelocityFluxFixedValue    rotatingTotalPressure                uniformJumpAMI
fixedJumpAMI                       movingWallVelocity              rotatingWallVelocity                 uniformTotalPressure
fixedMean                          oscillatingFixedValue           slip                                 variableHeightFlowRate
fixedNormalInletOutletVelocity     outletInlet                     supersonicFreestream                 variableHeightFlowRateInletVelocity
fixedNormalSlip                    outletMappedUniformInlet        surfaceNormalFixedValue              waveSurfacePressure
fixedPressureCompressibleDensity   outletPhaseMeanVelocity         swirlFlowRateInletVelocity           waveTransmissive
flowRateInletVelocity              partialSlip                     syringePressure
```

- To define boundary conditions that depends on time or on other variables, there are several possibilities,
  - Hardcoding in the solver,
  - **Program customized boundary conditions,**
  - With an additional package such as swak4Foam (http://openfoamwiki.net/index.php/Contrib/swak4Foam)

#10 – Customize boundary conditions (2/4)

- In the previous exercises(#8 and #9), the flow in porous media is evaluated from solving a partial differential equation on the pressure. Therefore, boundary conditions for the pressure field have to be specified. Sometime, however, it is more convenient to define an inlet velocity rather than a pressure condition. This inlet condition for the velocity can be described in term of boundary condition for the pressure using the relation:

$$\mathbf{n}.\nabla p = -\frac{\mu}{k}\mathbf{n}.\mathbf{U}$$

- We are going to create a new boundary condition, inspired by *fixedFluxPressure* itself derived from *fixedGradient*

```
$ mkdir   -p   $WM_PROJECT_USER_DIR/boundary/
$ cd     $WM_PROJECT_USER_DIR/boundary/
$ cp   -r  $FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/fixedFluxPressure  darcyGradPressure
$ cd  darcyGradPressure
$ rename    's/fixedFluxPressure/darcyGradPressure/g'     *.*
$ sed     -i   's/fixedFluxPressure/darcyGradPressure/g'     *.*
$ mkdir   Make
```

The string « fixedFluxPressure » is replaced by « darcyGradPressure » within all the files of the directory

```
$ gedit  Make/files
```

files

```
darcyGradPressureFvPatchScalarField.C

LIB = $(FOAM_USER_LIBBIN)/ldarcyGradPressure
```

```
$ gedit  Make/options
```

options

```
EXE_INC = -I$(LIB_SRC)/finiteVolume/lnInclude

LIB_LIBS = -lfiniteVolume
```

```
$ wclean
$ wmake
```

#10 – Customize boundary conditions (3a/4)

$ gedit darcyGradPressureFvPatchScalarField.H

```
#ifndef darcyGradPressureFvPatchScalarFields_H
#define darcyGradPressureFvPatchScalarFields_H

#include "fvPatchFields.H"
#include "fixedGradientFvPatchFields.H"

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

namespace Foam
{

/*---------------------------------------------------------------------------
            Class darcyGradPressureFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class darcyGradPressureFvPatchScalarField
:
    public fixedGradientFvPatchScalarField
{
    // Private data

        //- Name of the volicity field used to calculate grad(p)
        word UName_;

public:

    //- Runtime type information
    TypeName("darcyGradPressure");


    // Constructors

        //- Construct from patch and internal field
        darcyGradPressureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        darcyGradPressureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );
```

This boundary condition derives from the class *fixedGradient*

Name of the new type of boundary condition that will be specified in files 0/p

Declaration of constructors

#10 – Customize boundary conditions (3b/4)

```
        //- Construct by mapping given darcyGradPressureFvPatchScalarField onto a new patch
        darcyGradPressureFvPatchScalarField
        (
            const darcyGradPressureFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        darcyGradPressureFvPatchScalarField
        (
            const darcyGradPressureFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new darcyGradPressureFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        darcyGradPressureFvPatchScalarField
        (
            const darcyGradPressureFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field referenc
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new darcyGradPressureFvPatchScalarField(*this, iF)
            );
        }

    // Member functions

        //- Update the patch pressure gradient field
        virtual void updateCoeffs();

        //- Write
        virtual void write(Ostream&) const;
};
```

Declaration of constructors and constructors of copy.

Declaration of the function **updateCoeffs()**. It is in this function that the expression of the boundary condition is defined.

Declaration of the function **write()** that writes the value at the boundaries in the files *timeDirectory/p*.

#10 – Customize boundary conditions (4a/4)

$ gedit darcyGradPressureFvPatchScalarField.C

```
#include "darcyGradPressureFvPatchScalarField.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * * * * * Constructors  * * * * *

Foam::darcyGradPressureFvPatchScalarField::
darcyGradPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedGradientFvPatchScalarField(p, iF),
    UName_("U")
{}


Foam::darcyGradPressureFvPatchScalarField::
darcyGradPressureFvPatchScalarField
(
    const darcyGradPressureFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedGradientFvPatchScalarField(ptf, p, iF, mapper),
    UName_(ptf.UName_)
{}
```

```
Foam::darcyGradPressureFvPatchScalarField::
darcyGradPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedGradientFvPatchScalarField(p, iF),
    UName_(dict.lookupOrDefault<word>("U","U"))
{
    fvPatchField<scalar>::operator=(patchInternalField());
    gradient() = 0.0;
}


Foam::darcyGradPressureFvPatchScalarField::
darcyGradPressureFvPatchScalarField
(
    const darcyGradPressureFvPatchScalarField& wbppsf
)
:
    fixedGradientFvPatchScalarField(wbppsf),
    UName_(wbppsf.UName_)
{}


Foam::darcyGradPressureFvPatchScalarField::
darcyGradPressureFvPatchScalarField
(
    const darcyGradPressureFvPatchScalarField& wbppsf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedGradientFvPatchScalarField(wbppsf, iF),
    UName_(wbppsf.UName_)
{}
```

Definition of the constructors and copy constructors

#10 – Customize boundary conditions (4b/4)

```
// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //

void Foam::darcyGradPressureFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvPatchField<vector>& U =
        patch().lookupPatchField<volVectorField, vector>(UName_);

    //Extract the dictionary from the database
    const dictionary& transportProperties = db().lookupObject<IOdictionary> ("transportProperties");

    //Extract viscosity and permeability
    dimensionedScalar mu(transportProperties.lookup("mu"));
    dimensionedScalar  k(transportProperties.lookup("k"));

    gradient() = -mu.value()/k.value()*(U & patch().nf());

    fixedGradientFvPatchScalarField::updateCoeffs();
}


void Foam::darcyGradPressureFvPatchScalarField::write(Ostream& os) const
{
    fixedGradientFvPatchScalarField::write(os);
    writeEntryIfDifferent<word>(os, "U", "U", UName_);
    writeEntry("value", os);
}


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

namespace Foam
{
    makePatchTypeField
    (
        fvPatchScalarField,
        darcyGradPressureFvPatchScalarField
    );
}
```

We recover the value of U at the boundary

The viscosity and permeability values are read in the file *transportProperties*

The pressure gradient is evaluated at the boundary with the formula:

$$\mathbf{n}.\nabla p = -\frac{\mu}{k}\mathbf{n}.\mathbf{U}$$

mu.value() allows the access of the value of the object mu declared as a *dimensionedScalar*

patch().nf() returns the normal vector to the patch

```
$ wclean
$ wmake
```

The librairy *ldarcyGradPressure.so* is now compiled and available for all the solvers.

#11 - Two-equations model (1/6)

- Objective n°1 : Solve heat transfer in porous media with a two temperatures model (for the fluid and for the solid),

$$\nabla.\mathbf{U} = 0 \quad (1)$$

$$\mathbf{U} = -\frac{k}{\mu}\nabla p \quad (2)$$

$$\varepsilon\left(\rho_f C_{p_f}\right)\frac{\partial T_f}{\partial t} + \left(\rho_f C_{p_f}\right)\nabla.\mathbf{U}T_f = \nabla.\varepsilon D_{T_f}\nabla T_f + h_{sf}\left(T_f - T_s\right) \quad (3)$$

$$\left(1-\varepsilon\right)\left(\rho_s C_{p_s}\right)\frac{\partial T_s}{\partial t} = \nabla.\left(1-\varepsilon\right)D_{T_s}\nabla T_s - h_{sf}\left(T_f - T_s\right) \quad (4)$$

- Objective n°2 : Use customized boundary conditions

This solver will be based on *darcyTemperatureFoam (#8)*

```
$ cd    $WM_PROJECT_USER_DIR/applications/solvers/
$ cp  –r   darcyTemperatureFoam   darcyTwoTemperaturesFoam
$ cd   darcyTwoTemperaturesFoam
$ mv   darcyTemperatureFoam.C   darcyTwoTemperaturesFoam.C
$ gedit   Make/files
```

files ×

```
darcyTwoTemperaturesFoam.C

EXE = $(FOAM_USER_APPBIN)/darcyTwoTemperaturesFoam
```

```
$ wclean
$ wmake
```

#11 - Two-equations model (2/6)

```
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#include "createPhi.H"

Info<< "Reading field Ts\n" << endl;
volScalarField Ts
(
    IOobject
    (
        "Ts",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field Tf\n" << endl;
volScalarField Tf
(
    IOobject
    (
        "Tf",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

$ gedit createFields.H

The field *U* is now initialized from *0/U*, which allows us to defined boundary condition for U

phi is created by calling *createPhi.H*

Declaration of temperature fields for the solid and the fluid (do not write manually but use copy/paste from the former block « *volScalarField T ... »*)

The model constants are loaded from the file *constant/transportProperties*. (Again, do not write manually everything ! use copy/paste!)

```
Info<< "Reading permeability k\n" << endl;
dimensionedScalar k
(
    transportProperties.lookup("k")
);

Info<< "Reading fluid viscosity mu\n" << endl;
dimensionedScalar mu
(
    transportProperties.lookup("mu")
);

Info<< "Reading fluid conductivity DTf\n" << endl;
dimensionedScalar DTf
(
    transportProperties.lookup("DTf")
);

Info<< "Reading solid conductivity DTs\n" << endl;
dimensionedScalar DTs
(
    transportProperties.lookup("DTs")
);

Info<< "Reading porosity eps\n" << endl;
dimensionedScalar eps
(
    transportProperties.lookup("eps")
);

Info<< "Reading heat capacity rhoCps\n" << endl;
dimensionedScalar rhoCps
(
    transportProperties.lookup("rhoCps")
);

Info<< "Reading heat capacity rhoCpf\n" << endl;
dimensionedScalar rhoCpf
(
    transportProperties.lookup("rhoCpf")
);

Info<< "Reading heat exchange coefficient h\n" << endl;
dimensionedScalar h
(
    transportProperties.lookup("h")
);
```

#11 - Two-equations model (3/6)

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

    while (simple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(k/mu, p)
        );
        pEqn.solve();

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

    U = -k/mu*fvc::grad(p);
    U.correctBoundaryConditions();

    fvScalarMatrix TfEqn
    (
        eps*rhoCpf*fvm::ddt(Tf) + rhoCpf*fvm::div(phi,Tf)
      ==
        fvm::laplacian(eps*DTf,Tf,"laplacian(DT,T)") - fvm::Sp(h,Tf) + h*Ts
    );
    TfEqn.solve();

    fvScalarMatrix TsEqn
    (
        (1.-eps)*rhoCps*fvm::ddt(Ts)
      ==
        fvm::laplacian((1.-eps)*DTs,Ts,"laplacian(DT,T)") - fvm::Sp(h,Ts) + h*Tf
    );
    TsEqn.solve();


    runTime.write();

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

$ gedit darcyTwoTemperaturesFoam.C

The matrix for the pressure *pEqn* is declared as a *fvScalarMatrix* and constructed from the discretization with the finite volume method of the laplacian operator. This type of declaration offers more flexibility than *solve(fvm::laplacian(…)..)*. The matrix is inversed with *pEqn.solve()*

The flux of velocity at the cell face is directly updated from the new coefficients of the matrix pressure.

U is calculated pointwise with Darcy's law. The boundary conditions may have been altered and do not correspond to what was specified in *0/U*. This function means that the boundary conditions have to be those specified in *0/U*.

Solve the temperature in the fluid. The laplacian discretization scheme is specified in *system/fvSchemes* in front of the keyword « *laplacian(DT,T)* ». A part of the exchange term is treated implicitly, the other part explicitly.

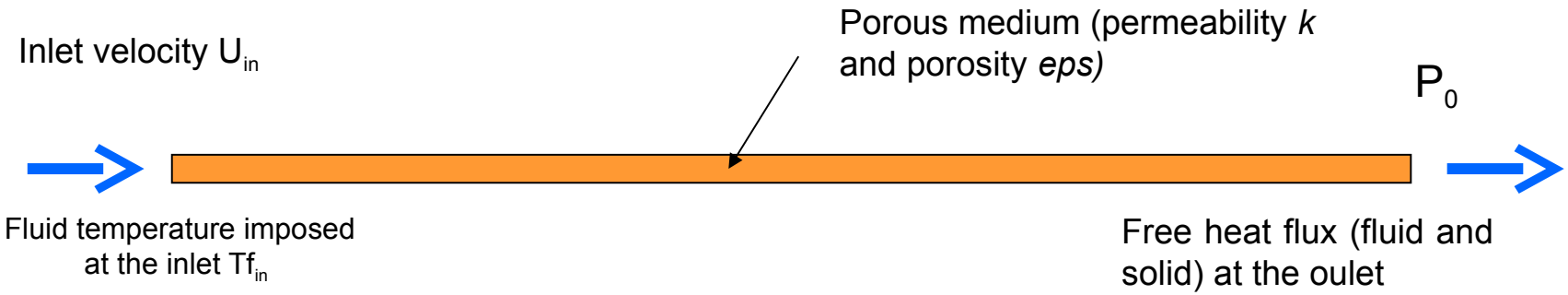
Solve the temperature in the solid.

$ wclean
$ wmake

#11 - Two-equations model (4/6)

- We want to estimate heat transfer in a 1D porous medium with a two temperatures model.

- In this example, a porous medium initially at 573K is cooled down with the injection of a fluid at 273K



To set up the simulation, we use #8

```
$ run
$ cp   –r   Exo9   Exo11
$ cd   Exo11
$ rm    –r    0.*   [1-9]*   postProcessing
$ mv   0/T    0/Tf
$ cp   0/Tf   0/Ts
$ cp   0/p    0/U
```

#11 - Two-equations model (5a/6)

$ gedit 0/U

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

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

internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (1.4e-4 0 0);
    }

    outlet
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit 0/p

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

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            darcyGradPressure;
        value           uniform 0;
    }

    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

$$\mathbf{n}.\nabla p = -\frac{\mu}{k}\mathbf{n}.\mathbf{U}$$

A fluid velocity is imposed at the domain inlet. This velocity is used to evaluate the pressure gradient at the inlet with the boundary condition *darcyGradPressure* we have developed in #9. To use this boundary condition, we must specify in *system/controlDict* that we load the library, *ldarcyGradPressure.so.*

#11 - Two-equations model (5b/6)

$ gedit 0/Ts

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

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 573;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

$ gedit 0/Tf

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

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 573;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 273;
    }

    outlet
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

#11 - Two-equations model (5c/6)

$ gedit constant/transportProperties

```
k               k      [0  2  0  0 0 0 0] 1e-09;
mu              mu     [1 -1 -1  0 0 0 0] 1e-05;

eps             eps    [0  0  0  0 0 0 0] 0.4;

DTf             DTf    [1  1 -3 -1 0 0 0] 1e-04;

DTs             DTs    [1  1 -3 -1 0 0 0] 1e-02;

rhoCps          rhoCps [1 -1 -2 -1 0 0 0] 2e4;

rhoCpf          rhoCpf [1 -1 -2 -1 0 0 0] 5e3;

h               h      [1 -1 -3 -1 0 0 0] 5e-1;
```

$ gedit system/controlDict

```
application     darcyTwoTemperaturesFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         400000;

deltaT          100;

writeControl    runTime;

writeInterval   10000;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

libs ("ldarcyGradPressure.so");

functions
{
    probes
    {
        type            probes;
        functionObjectLibs ("libsampling.so");
        enabled         true;
        outputControl   timeStep;
        outputInterval  1;

        fields
        (
            Tf
            Ts
        );
```

We specify that we load the library *ldarcyGradPressure.so* in order to use the customized boundary condition *darcyGradPressure.*

#11 - Two-equations model (5g/6)

$ gedit system/fvSolution

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

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

    Tf
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-06;
        relTol          0;
    }

}

SIMPLE
{
    nNonOrthogonalCorrectors 2;
}
```

$ gedit system/fvSchemes

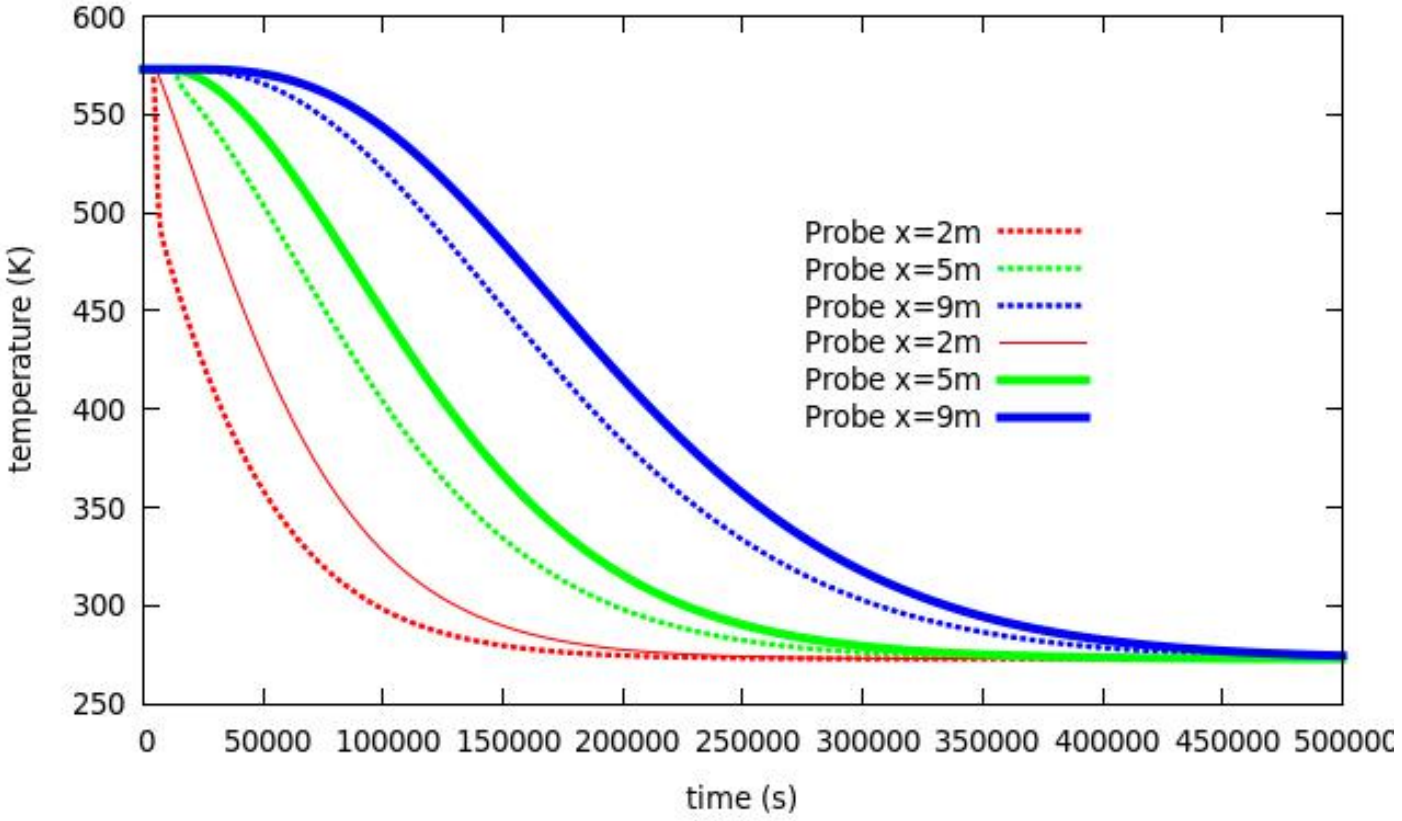
```
ddtSchemes
{
    default         Euler;
}

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

divSchemes
{
    default         none;
    div(phi,Tf)     Gauss vanLeer;
}

laplacianSchemes
{
    default         none;
    laplacian((k|mu),p) Gauss linear corrected;
    laplacian(DT,T) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    p;
}
```

#11 - Two-equations model (6/6)

We start the simulation:

```
$ darcyTwoTemperaturesFoam
```

We then post-treat the evolution *Ts* and *Tf* with time for the 3 probes



Exo11Bis: Change the exchange coefficient value and re-do the simulation. For large values, we recover the solution of #9.

Navier-Stokes with icoFoam (1/5)

- Navier-Stokes equations are made of a continuity equation and a momentum equation

$$\nabla.\mathbf{U} = 0 \qquad (1)$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla.\mathbf{U}\mathbf{U} = -\nabla p + \nabla.\nu\nabla\mathbf{U} \qquad (2)$$

- We look for *(U,p)* solution of this problem. How can we solve this problem in a segregated manner ? (one equation after the other) ?

- First, we derive a pressure equation combining (1) and (2),
- Then, we use a predictor/corrector strategy to solve this system (ex : PISO for transient solutions, SIMPLE for steady-state simulations, PIMPLE which is a mix of these two algorithms allows larger time steps),
- In this section, we learn how to solve NS with the PISO algorithm implemented in *icoFoam*

Navier-Stokes with icoFoam (2a/5)

- With the finite volume method in OpenFOAM®, the advection velocity in the divergence operator is defined at the cell faces (*phi*). Since the fluid density is constant, the solved pressure is in fact the actual pressure divided by rho:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla . \phi \mathbf{U} - \nabla . \nu \nabla \mathbf{U} = -\nabla p$$

- To derive a pressure equation, we write the former equation is a semi-discretized formulation :

$$\nu \frac{\mathbf{U}_P^{n+1} - \mathbf{U}_P^n}{\delta t} = -a'_P \mathbf{U}_P^{n+1} + \sum_{NP} a'_{NP} \mathbf{U}_{NP}^{n+1} - \nabla p$$

Discretization of the convective and the diffusive terms.

It can be recast into

$$\underbrace{\left(\frac{\nu}{\delta t} + a'_P \right)}_{a_P} \mathbf{U}_P^{n+1} = \underbrace{\sum_{NP} a'_{NP} \mathbf{U}_{NP}^{n+1} + \frac{\nu}{\delta t} \mathbf{U}_P^n}_{\mathrm{H}(\mathbf{U})} - \nabla p$$

Navier-Stokes with icoFoam (2b/5)

Or,

$$a_P \mathbf{U}_P = \mathrm{H}(\mathbf{U}) - \nabla p$$

Diagonal coefficients of the matrix for the velocity U

Contains the off-diagonal coefficients and the source terms (body forces + half of the discretization of the transient term)

Or,

$$\mathbf{U}_P = \frac{\mathrm{H}(\mathbf{U})}{a_P} - \frac{1}{a_P}\nabla p$$

- Combining this equation with continuity equation leads to the pressure equation:

$$\nabla . \left(\frac{1}{a_P} \nabla p \right) = \nabla . \left(\frac{\mathrm{H}(\mathbf{U})}{a_P} \right)$$

- In this equation, a_P et *H(U)* are evaluated from the velocity field of the previous iteration or previous time step.

Navier-Stokes with icoFoam (3/5)

```
$ sol
$ cd      incompressible/icoFoam
$ gedit  icoFoam.C
```

1. Beginning of the time loop

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

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

    #include "CourantNo.H"
```

2. U* is predicted solving implicitly the momentum equation (matrix UEqn) with the pressure of the previous time step

```
    // Momentum predictor

    fvVectorMatrix UEqn
    (
        fvm::ddt(U)
      + fvm::div(phi, U)
      - fvm::laplacian(nu, U)
    );

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

Construction of the matrix UEqn. The convective term is linearized with phi evaluated at the previous time step.

U* is predicted from the pressure field of the previous time step

$$\mathrm{UEqn}.\mathbf{U}^* = -\nabla p^n$$

Navier-Stokes with icoFoam (4a/5)

3 On entre dans la boucle PISO

```
// --- PISO loop
while (piso.correct())
{
    volScalarField rAU(1.0/UEqn.A());

    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn.H();
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (fvc::interpolate(HbyA) & mesh.Sf())
      + fvc::interpolate(rAU)*fvc::ddtCorr(U, phi)
    );

    adjustPhi(phiHbyA, U, p);

    // Non-orthogonal pressure corrector loop
    while (piso.correctNonOrthogonal())
    {
        // Pressure corrector

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

        pEqn.setReference(pRefCell, pRefValue);

        pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));

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

    #include "continuityErrs.H"

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

Update of a_P from U freshly computed

Update of H/a_p or “H by A” from the U freshly computed

Projection of H/a_P over the cell faces to calculate the divergence *div(HbyA)*

$$\nabla.\left(\frac{1}{a_P}\nabla p\right) = \nabla.\left(\frac{\mathrm{H}(\mathbf{U})}{a_P}\right)$$

Here, we recover the right value of the flux velocity

$$\phi = \mathrm{S}.\left(\frac{\mathrm{H}(\mathbf{U})}{a_P}\right)_{c\to f} - \mathrm{S}.\left(\frac{1}{a_P}\nabla p\right)_{c\to f}$$

Velocity corrector stage:

$$\mathbf{U}_P = \frac{\mathrm{H}(\mathbf{U})}{a_P} - \frac{1}{a_P}\nabla p$$

4 At least 2 iterations are required. Then one exit the PISO loop and go to the next time step.

Navier-Stokes with icoFoam (4b/5)

- Some additional details

```
// --- PISO loop
while (piso.correct())
{
    volScalarField rAU(1.0/UEqn.A());

    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn.H();
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (fvc::interpolate(HbyA) & mesh.Sf())
      + fvc::interpolate(rAU)*fvc::ddtCorr(U, phi)
    );

    adjustPhi(phiHbyA, U, p);

    // Non-orthogonal pressure corrector loop
    while (piso.correctNonOrthogonal())
    {
        // Pressure corrector

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

        pEqn.setReference(pRefCell, pRefValue);

        pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));

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

    #include "continuityErrs.H"

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

Insure mass conservation by adjusting the in-coming and out-coming flux if the boundary conditions are ill-defined (no *fixedValue* for *p* for instance)

If there is no *fixedValue* among the pressure BCs, then the value of *p* at the cell with the reference *pRefCell* is fixed to *pRefValue*.

At the previous line *U* has been calculated point-wise from *p*. The BCs do no longer correspond to the ones in *0/U*. This function means that the boundary conditions of *U* must be the ones precised in *0/U*.

Navier-Stokes with icoFoam (5/5)

Some variation (exercise):

- Write a PISO algorithm with the actual pressure

$$\nabla.\mathbf{U} = 0$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla.\mathbf{U}\mathbf{U} = -\frac{1}{\rho}\nabla p + \nabla.\nu\nabla\mathbf{U}$$

- Write a PISO algorithm with a body source term :

$$\nabla.\mathbf{U} = 0$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla.\mathbf{U}\mathbf{U} = -\frac{1}{\rho}\nabla p + \mathbf{g} + \nabla.\nu\nabla\mathbf{U}$$

- Write a PISO algorithm with a mass source term:

$$\nabla.\mathbf{U} = \Gamma$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla.\mathbf{U}\mathbf{U} = -\frac{1}{\rho}\nabla p + \nabla.\nu\nabla\mathbf{U}$$

- Write a PISO algorithm for a Darcy-Brinkman system:

$$\nabla.\mathbf{U} = 0$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla.\mathbf{U}\mathbf{U} = -\frac{1}{\rho}\nabla p + \nabla.\nu\nabla\mathbf{U} - \nu k^{-1}\mathbf{U}$$

Bibliography:

- *Solution of the Implicitly Discretised Fluid Flow Equations by Operator-Splitting*, Issa, 1985
- *Numerical Heat Transfer and Fluid Flow*, Patankar, 1980
- *Computational Methods for Fluid Dynamics*, Ferziger and Peric, 2002
- *Micro-continuum approach for pore-scale simulation of subsurface processes, Soulaine and Tchelepi, 2016*
- *A PISO-like algorithm to simulate superfluid helium flow with the two-fluid model, Soulaine et al., 2016*