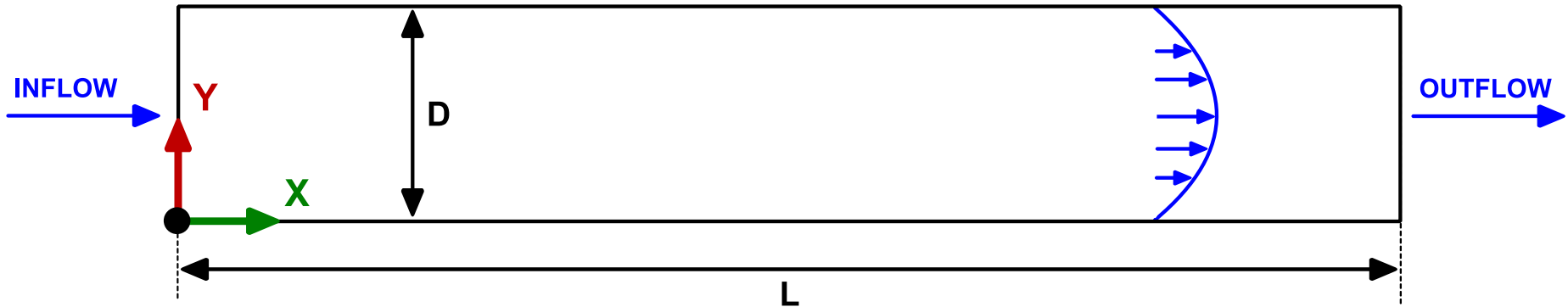


# A simple validation case – Hagen-Poiseuille solution

## Hagen-Poiseuille solution – $Re = 100$ Incompressible flow



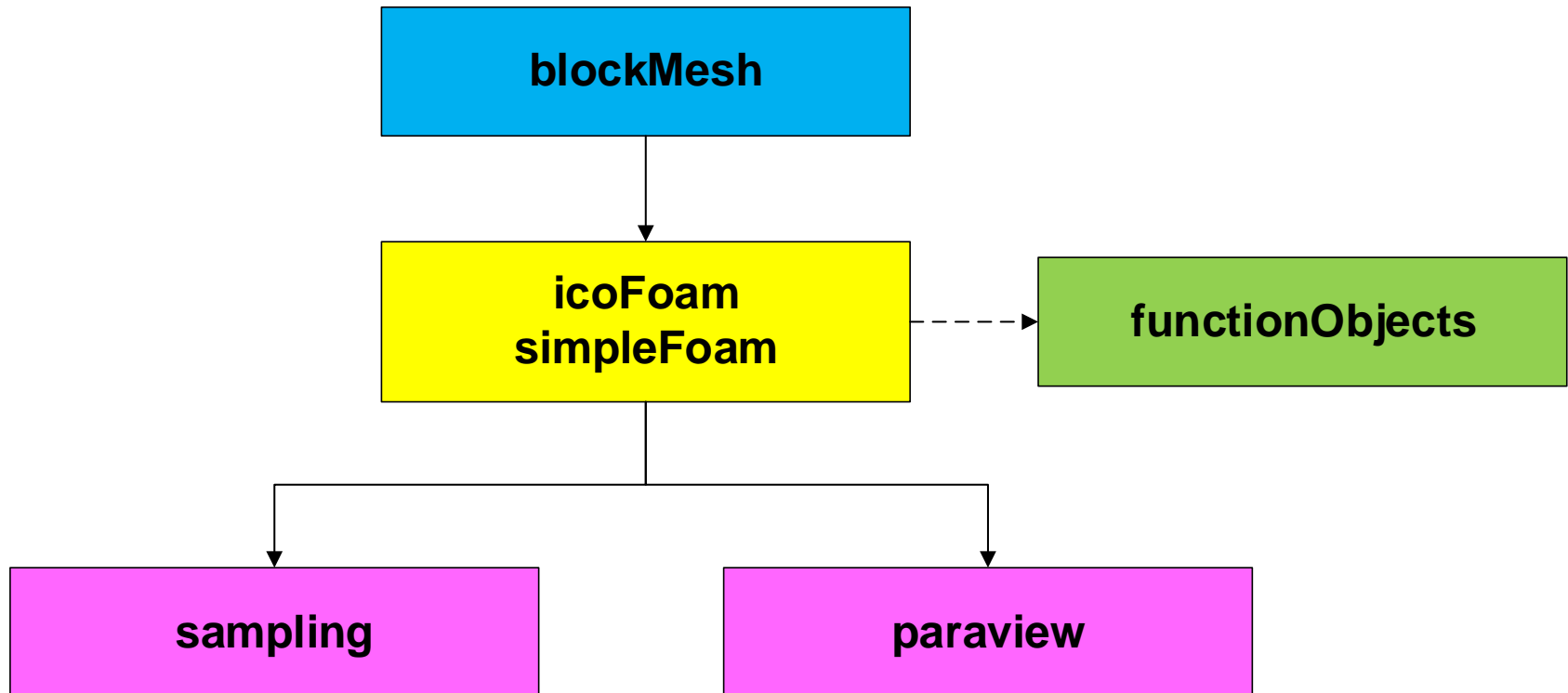
### Physical and numerical side of the problem:

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

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

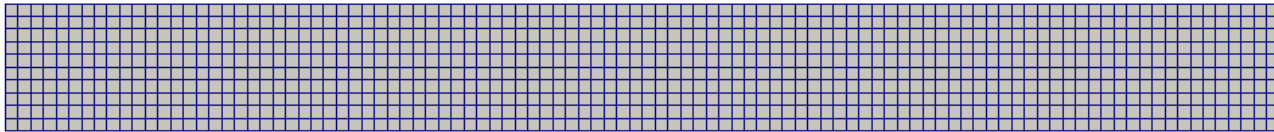
# A simple validation case – Hagen-Poiseuille solution

## Workflow of the case



# A simple validation case – Hagen-Poiseuille solution

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



**Mesh (coarse add 2D)**

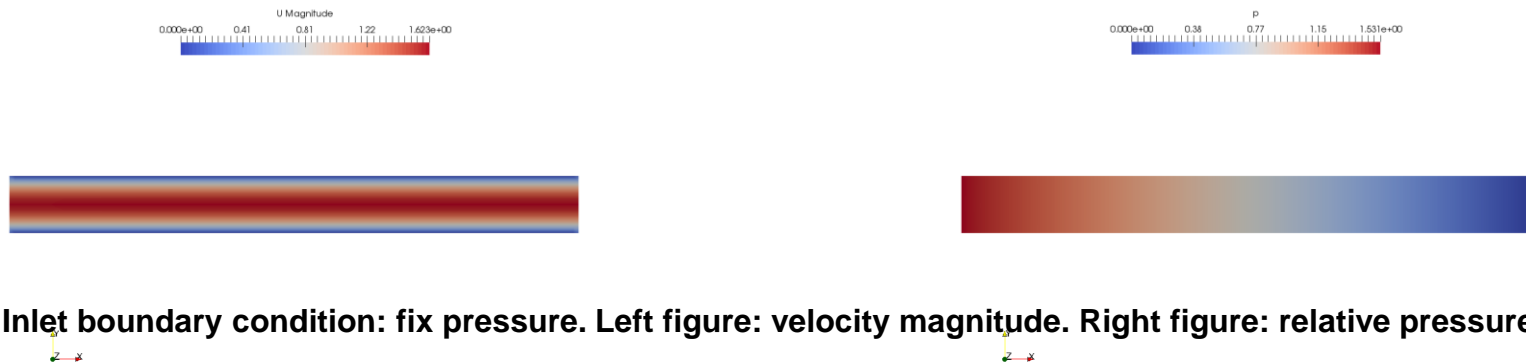
- This mesh is very coarse but for the physics involved it works fine.
- You can try to do successive refinements of this mesh in order to do a mesh independency study.
- If you deal with turbulence, you will need to refine the mesh close to the walls in order to resolve the boundary layers.

# A simple validation case – Hagen-Poiseuille solution

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



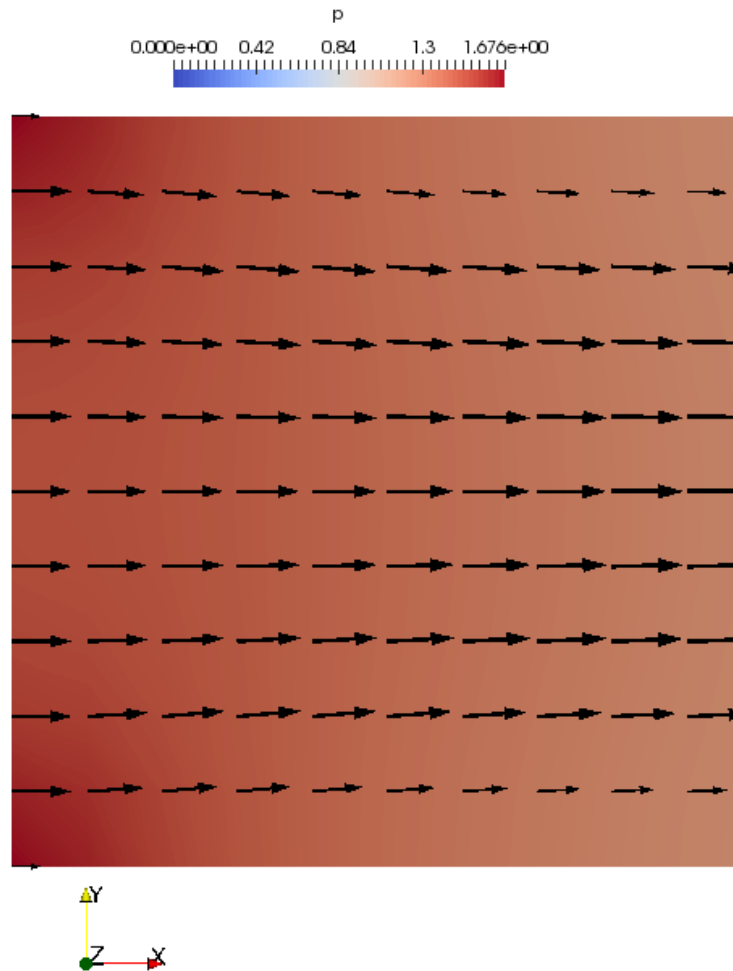
Inlet boundary condition: fix uniform velocity. Left figure: velocity magnitude. Right figure: relative pressure



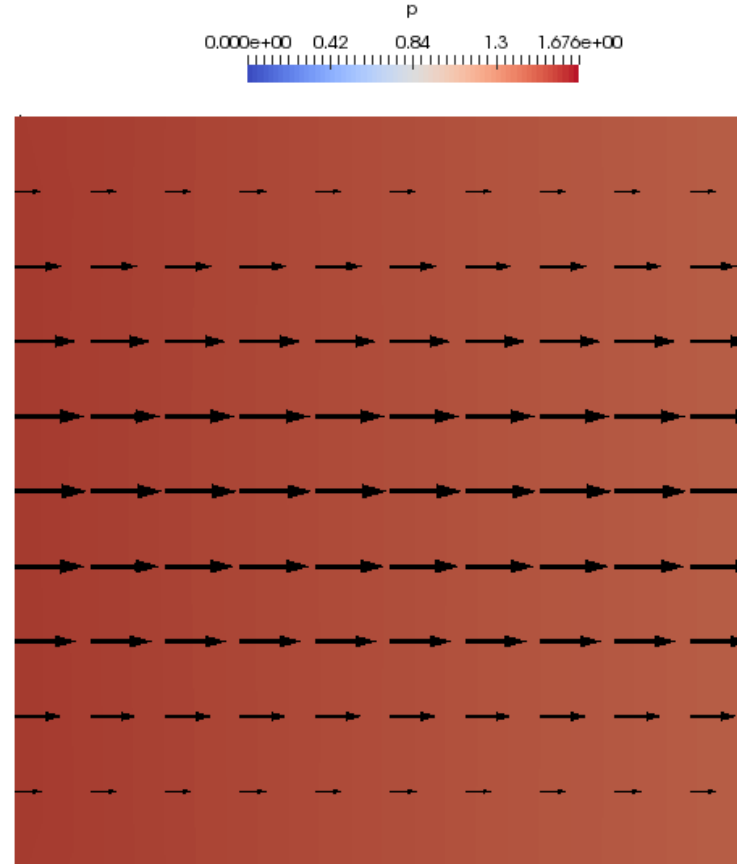
Inlet boundary condition: fix pressure. Left figure: velocity magnitude. Right figure: relative pressure

# A simple validation case – Hagen-Poiseuille solution

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



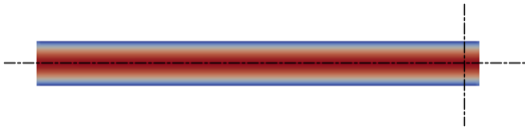
Velocity profile at the inlet  
BC: fix uniform velocity



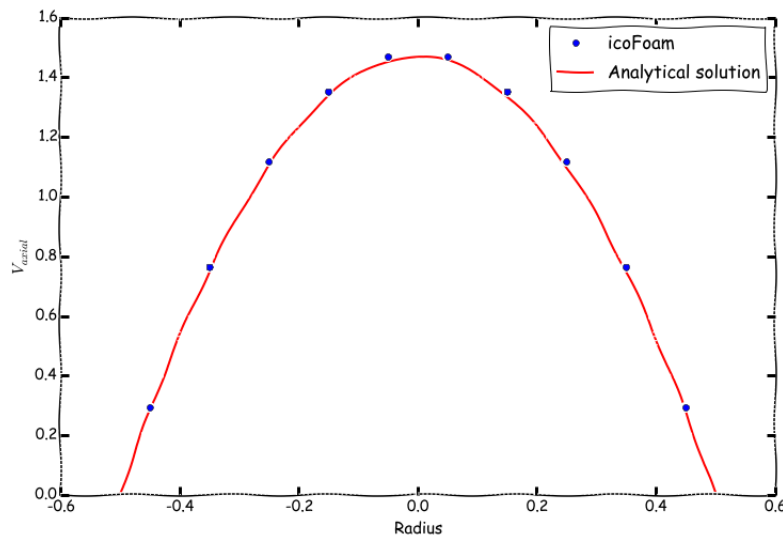
Velocity profile at the inlet  
BC: fix pressure

# A simple validation case – Hagen-Poiseuille solution

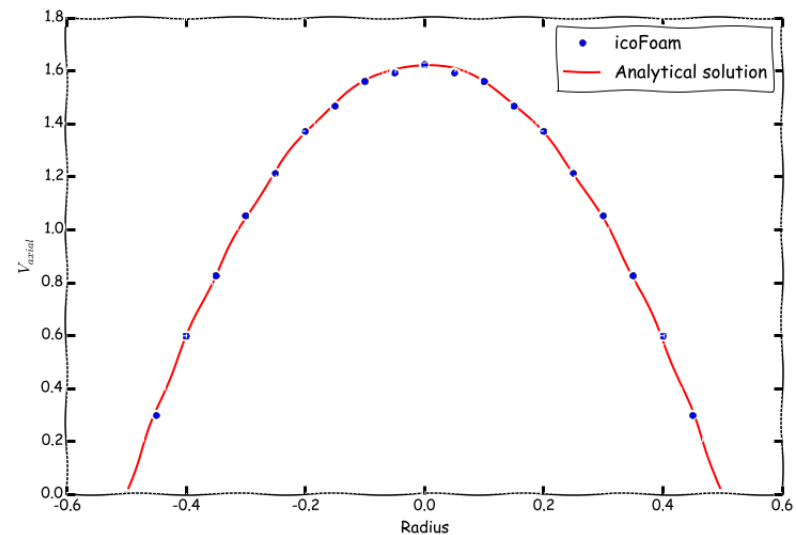
At the end of the day you should get something like this



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



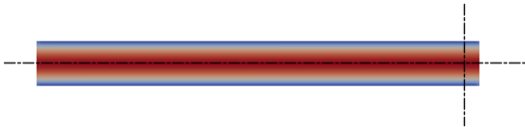
Velocity profile at the outlet  
BC: fix uniform velocity



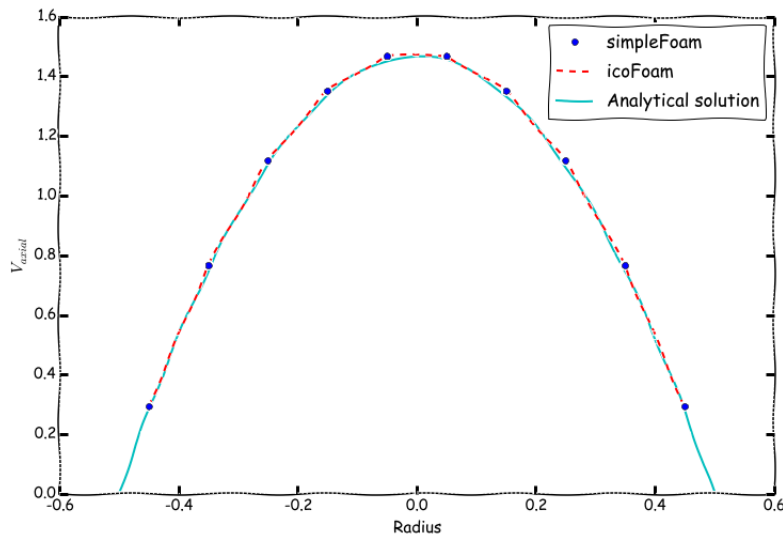
Velocity profile at the outlet  
BC: fix pressure

# A simple validation case – Hagen-Poiseuille solution

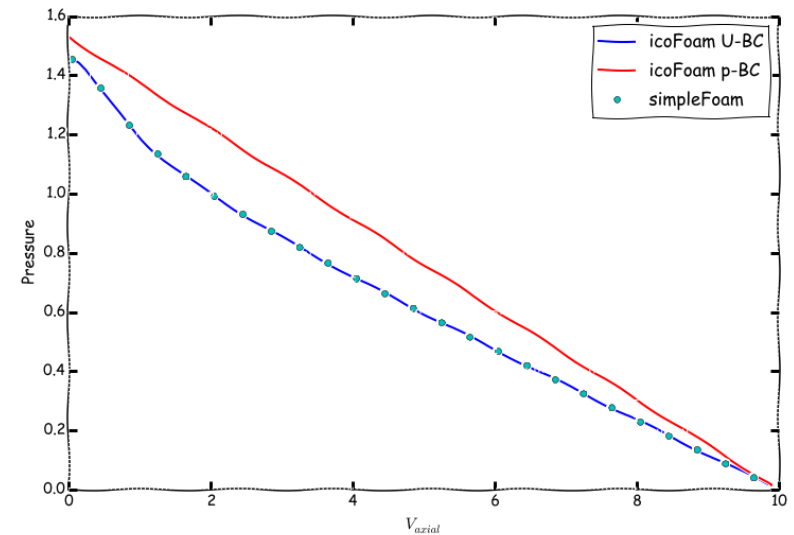
At the end of the day you should get something like this



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



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



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

# A simple validation case – Hagen-Poiseuille solution

## Loading OpenFOAM® environment

- If you are using our virtual machine or using the lab workstations, you will need to source OpenFOAM® (load OpenFOAM® environment).
- To source OpenFOAM®, type in the terminal:
  - `$> of4x`
- To use PyFoam you will need to source it. Type in the terminal:
  - `$> anaconda2`                      **or**                      `anaconda3`
- Remember, every time you open a new terminal window you need to source OpenFOAM® and PyFoam.
- By default, when installing OpenFOAM® and PyFoam you do not need to do this. This is our choice as we have many things installed and we want to avoid conflicts between applications.



# A simple validation case – Hagen-Poiseuille solution

## What are we going to do?

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

# A simple validation case – Hagen-Poiseuille solution

**Let us explore the case directory**

# A simple validation case – Hagen-Poiseuille solution

## The *blockMeshDict* dictionary file

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *blockMeshDict* dictionary file

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *blockMeshDict* dictionary file

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

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

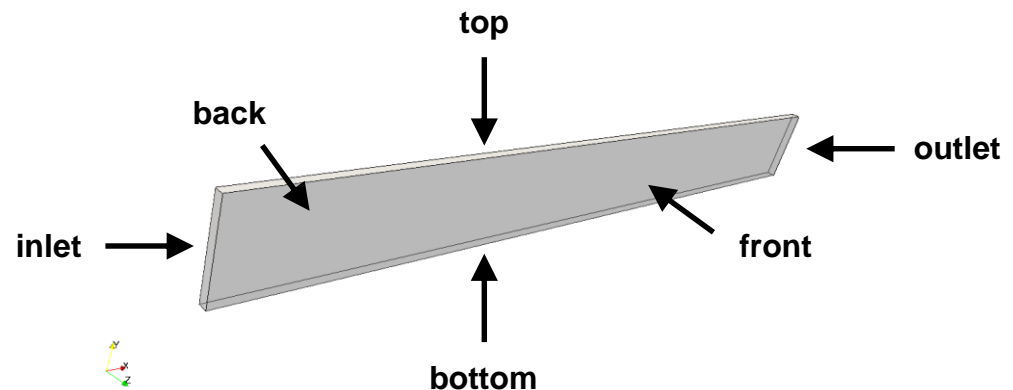
# A simple validation case – Hagen-Poiseuille solution




## The *boundary* dictionary file

```
18 6
19 (
20     top
21     {
22         type            wall;
23         nFaces          100;
24         startFace       1890;
25     }
26     inlet
27     {
28         type            patch;
29         nFaces          10;
30         startFace       1990;
31     }
32     outlet
33     {
34         type            patch;
35         nFaces          10;
36         startFace       2000;
37     }
38     bottom
39     {
40         type            wall;
41         nFaces          100;
42         startFace       2010;
43     }
44     back
45     {
46         type            empty;
47         nFaces          1000;
48         startFace       2110;
49     }
50     front
51     {
52         type            empty;
53         nFaces          1000;
54         startFace       3110;
55     }
56 )
```

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



# A simple validation case – Hagen-Poiseuille solution

 The *transportProperties* dictionary file

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

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

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

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

# A simple validation case – Hagen-Poiseuille solution



## The 0 directory

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



# A simple validation case – Hagen-Poiseuille solution

 The file *0/p*

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

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

# A simple validation case – Hagen-Poiseuille solution

 The file *0/p*

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

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

# A simple validation case – Hagen-Poiseuille solution

 The file  $0/U$

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

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

# A simple validation case – Hagen-Poiseuille solution

 The file  $0/U$

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

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

# A simple validation case – Hagen-Poiseuille solution

## The **system** directory

- The **system** directory consists of the following compulsory dictionary files:
  - *controlDict*
  - *fvSchemes*
  - *fvSolution*
- *controlDict* contains general instructions on how to run the case.
- *fvSchemes* contains instructions for the discretization schemes that will be used for the different terms in the equations.
- *fvSolution* contains instructions on how to solve each discretized linear equation system.

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

```
49  functions
50  {

    name_of_the_functionObject_dictionary
    {
        Dictionary with the functionObject entries
    }

204
207  };
```

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

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

```
49 functions
50 {
51
52 minmaxdomain
53 {
54     type fieldMinMax;
55
56     functionObjectLibs ("libfieldFunctionObjects.so");
57
58     enabled true; //true or false
59
60     mode component;
61
62     writeControl timeStep;
63     writeInterval 1;
64
65     log true;
66
67     fields ( p U );
68 }
69
70
71
```

- **fieldMinMax functionObject**

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

**postProcessing/minmaxdomain/0**



# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

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

- **faceSource functionObject**

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

**postProcessing/inMassFlow/0**

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

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

- **faceSource functionObject**

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

**postProcessing/outMassFlow/0**

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

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

- **faceSource functionObject**

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

**postProcessing/inPre/0**

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

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

- **faceSource functionObject**

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

**postProcessing/outMax/0**

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

# A simple validation case – Hagen-Poiseuille solution



## The *fvSchemes* dictionary

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *fvSolution* dictionary

```
17 solvers
18 {
19     p
20     {
21         solver      GAMG;
22         tolerance    1e-6;
23         relTol       0.01;
24         smoother     GaussSeidel;
25         nPreSweeps    0;
26         nPostSweeps   2;
27         cacheAgglomeration on;
28         agglomerator  faceAreaPair;
29         nCellsInCoarsestLevel 100;
30         mergeLevels   1;
31     }
32
33     pFinal
34     {
35         $p;
36         relTol       0;
37     }
38
39     U
40     {
41         solver      PBiCG;
42         preconditioner DILU;
43         tolerance    1e-08;
44         relTol       0;
45     }
46 }
47
48 PISO
49 {
50     nCorrectors      1;
51     nNonOrthogonalCorrectors 0;
52 }
```

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

# A simple validation case – Hagen-Poiseuille solution

## The *fvSolution* dictionary

```
17 solvers
18 {
19     p
20     {
21         solver      GAMG;
22         tolerance    1e-6;
23         relTol       0.01;
24         smoother     GaussSeidel;
25         nPreSweeps    0;
26         nPostSweeps   2;
27         cacheAgglomeration on;
28         agglomerator   faceAreaPair;
29         nCellsInCoarsestLevel 100;
30         mergeLevels    1;
31     }
32
33     pFinal
34     {
35         $p;
36         relTol       0;
37     }
38
39     U
40     {
41         solver      PBiCG;
42         preconditioner DILU;
43         tolerance    1e-08;
44         relTol       0;
45     }
46 }
47
48 PISO
49 {
50     nCorrectors      1;
51     nNonOrthogonalCorrectors 0;
52 }
```

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

# A simple validation case – Hagen-Poiseuille solution

## The **system** directory

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



# A simple validation case – Hagen-Poiseuille solution

## The *sampleDict* dictionary

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

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

# A simple validation case – Hagen-Poiseuille solution

## Running the case

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

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

# A simple validation case – Hagen-Poiseuille solution

## Running the case

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

# A simple validation case – Hagen-Poiseuille solution

## Let us use different boundary conditions

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

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

# A simple validation case – Hagen-Poiseuille solution

 The file *0/p*

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

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

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

# A simple validation case – Hagen-Poiseuille solution

 The file  $0/U$

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

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

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

# A simple validation case – Hagen-Poiseuille solution

## Running with a steady state solver


- At  $Re = 100$  nothing is happening. No vortex shedding, no detached flow, no flow instabilities, no turbulence, and no shock waves (this is kind of a boring case). Therefore is safe to say that this is a steady flow.
- In this case, we can use a steady solver. Steady solvers are way much faster than unsteady solvers but they violate a lot of principles, this is a trick that CFDers use to speed-up things. If you are happy with this approximation use steady solvers with no remorse.
- In an ideal world, steady solvers should converge in one iteration. But due to the non-linearities in the governing equations we need to proceed in an iterative way, until we satisfy a convergence criteria.
- Let us run this case using `simpleFoam` (which is an incompressible steady solver).
- As we are using a new solver we need to do some changes in the dictionaries files.
- This case is already setup in the directory `$PTOFC/1010F/laminar_pipe/case2`
- At this point, let us explore the case directory.

# A simple validation case – Hagen-Poiseuille solution

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




# A simple validation case – Hagen-Poiseuille solution

 The *turbulenceProperties* dictionary file

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

```
17      simulationType      laminar;
```

# A simple validation case – Hagen-Poiseuille solution

 The *transportProperties* dictionary file

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

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *controlDict* dictionary

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *fvSchemes* dictionary

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *fvSolution* dictionary

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

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

# A simple validation case – Hagen-Poiseuille solution

## The *fvSolution* dictionary

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

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

<b>p</b>	<b>0.3;</b>
<b>U</b>	<b>0.7;</b>
<b>k</b>	<b>0.7;</b>
<b>omega</b>	<b>0.7;</b>
<b>epsilon</b>	<b>0.7;</b>

# A simple validation case – Hagen-Poiseuille solution

## Running the case

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

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

# A simple validation case – Hagen-Poiseuille solution

## Running the case

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



# A simple validation case – Hagen-Poiseuille solution

## What about mesh quality?

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

Checking geometry...

Overall domain bounding box (0 -0.5 0) (10 0.5 0.1)

Mesh has 2 geometric (non-empty/wedge) directions (1 1 0)

Mesh has 2 solution (non-empty) directions (1 1 0)

All edges aligned with or perpendicular to non-empty directions.

Boundary openness (7.8140697e-20 1.4221607e-17 5.4393739e-16) OK.

Max cell openness = 8.6736174e-17 OK.

Max aspect ratio = 1 OK.

Minimum face area = 0.01. Maximum face area = 0.01. Face area magnitudes OK.

Min volume = 0.001. Max volume = 0.001. Total volume = 1. Cell volumes OK.

Mesh non-orthogonality Max: 0 average: 0

Non-orthogonality check OK.

Face pyramids OK.

Max skewness = 1.0658141e-13 OK.

Coupled point location match (average 0) OK.

Non-orthogonality

Skewness

Mesh OK.

# A simple validation case – Hagen-Poiseuille solution

## What about mesh quality?

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

Checking geometry...

Overall domain bounding box (0 -0.5 0) (10 0.5 0.1)

Mesh has 2 geometric (non-empty/wedge) directions (1 1 0)

Mesh has 2 solution (non-empty) directions (1 1 0)

All edges aligned with or perpendicular to non-empty directions.

Boundary openness (7.8140697e-20 1.4221607e-17 5.539394e-16) OK.

Max cell openness = 9.3768837e-17 OK.

Max aspect ratio = 1.98 OK.

Minimum face area = 0.00085. Maximum face area = 0.02154739. Face area magnitudes OK.

Min volume = 8.5e-05. Max volume = 0.001915. Total volume = 1. Cell volumes OK.

Mesh non-orthogonality Max: 86.473612 average: 2.5674993

\*Number of severely non-orthogonal (> 70 degrees) faces: 1.

Non-orthogonality check OK.

<<Writing 1 non-orthogonal faces to set nonOrthoFaces

\*\*\*Error in face pyramids: 2 faces are incorrectly oriented.

<<Writing 2 faces with incorrect orientation to set wrongOrientedFaces

\*\*\*Max skewness = 11.305066, 1 highly skew faces detected which may impair the quality of the results

<<Writing 1 skew faces to set skewFaces

Coupled point location match (average 0) OK.

Failed 2 mesh checks.

Too high non-orthogonality  
Acceptable values are less than 80

Failed sets can be  
visualized in paraFoam

Too high skewness  
Acceptable values are less than 6

This does not mean that you can  
not run the simulation