Introductory level L1 – Tutorial 3

Flow around a cylinder

Revision 1.
December 2014.

This page intentionally left in blank

Copyright and Disclaimer

This page intentionally left in blank

1. Overview and learning goals

This is an introductory level tutorial. To follow this tutorial you need minimum working skills using OpenFOAM®. Basic knowledge using Linux and shell scripting is also beneficial.

This tutorial describes the setup and solution of a typical external aerodynamics configuration using and incompressible solver. In this tutorial we will solve a two-dimensional incompressible flow around a cylinder at a given Reynolds number.

In this tutorial you will learn:

- Generate the mesh using OpenFOAM® meshing tools.
- Assess mesh quality.
- Set physical properties.
- Set boundary conditions and initial conditions.
- Choose the solution method and numerical schemes.
- Solution monitoring.
- Compute the aerodynamic forces and aerodynamic coefficients on the body.
- Visualization using `paraFoam`.
- Plotting using `gnuplot`.
- Data manipulation and plotting using `Python`.
- Some scripting.

NOTE:

All the dictionaries and files to be used in this tutorial have already been generated and are located within the case directory. The case is ready to go.

2. Prerequisites

This tutorial assumes that you have little or no experience with OpenFOAM®, therefore each step will be explicitly described.

This tutorial also assumes that you have a minimum working knowledge of Linux shell.

To follow this tutorial you will need the following applications:

- OpenFOAM® version `2.3.0` or `2.3.x`
- `Gnuplot 4.6`
- Any text editor (`gedit`, `kate`, `vi`, `emacs`, *etc*.). For convenience, we will use `gedit`.
- `pyFoam` (optional).
- `Python 2.7`
- `scipy 0.14.0` or newer.
- `numpy 1.9.0` or newer.
- `pandas 0.15.0` or newer.
- `matplotlib 1.4.0` or newer.

You will also need the training material and course tutorials that you will find in the USB key.

You can copy the training material and course tutorials anywhere you like, there is no restriction on this matter.

This tutorial is located in the directory:

```
$PTOFC/first_tutorial/vortex_shedding/c1/
```

NOTE:

`$PTOFC` points to the absolute path where you unpacked the tutorials.

3. Problem description

In this tutorial we are going to solve a two-dimensional incompressible flow around a cylinder at a Reynolds number of 200. For this value of Reynolds number we expect the development of the von Karman vortex street behind the cylinder; hence, a transient solver will be used. This is a typical external aerodynamics configuration.

All the dimensions, fluid properties, boundary conditions, and initial conditions values are given in SI units.

WARNING:

Remember, OpenFOAM® is fully dimensional.

The computational domain is shown in figure 1. The external boundaries are located far enough the cylinder surface in order to avoid large gradients between the cylinder and the external boundaries.

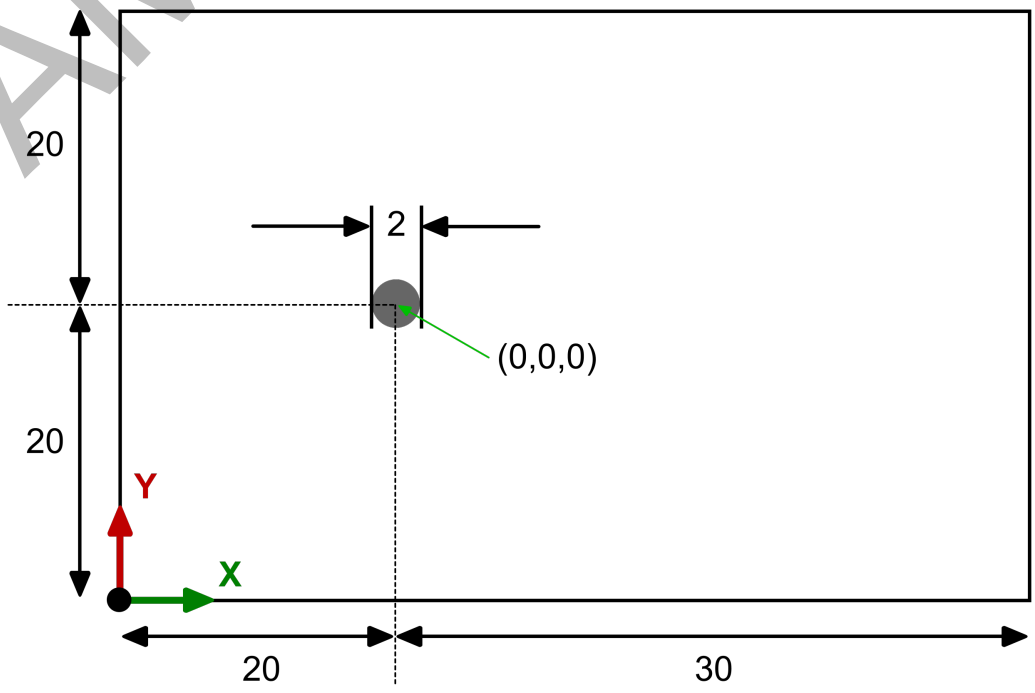

Figure 1. Computational domain (not to scale) and dimensions.

A layout of the boundary conditions and initial conditions is illustrated in figure 2.

In this case, the flow is entering the domain at the boundary face BC1.

At the boundary face BC2 the flow is going out of the domain. As the method is conservative, the mass flow entering the domain and the mass flow going out the domain must balance.

The boundaries BC3 and BC4 are defined as slip walls (there is no boundary layer). To model a slip wall, you can use slip wall or symmetry boundary conditions. Remember, symmetry boundary conditions only apply to planar surfaces.

The boundary BC5 is a no-slip wall, we are interested in resolving the boundary layer on this boundary patch. We are also interested in computing the aerodynamic forces on this wall.

The initial conditions IC1 correspond to a uniform flow in the whole domain.

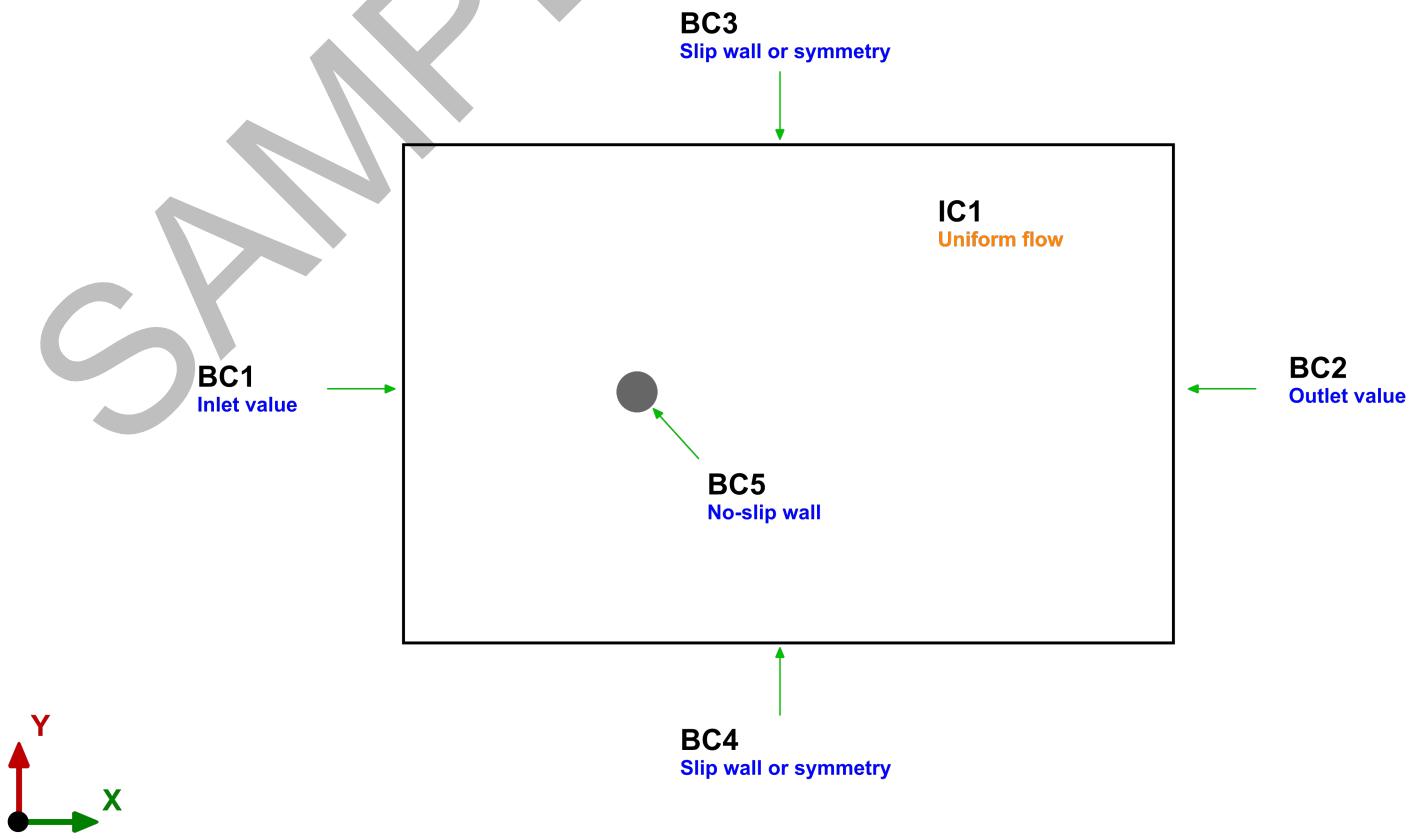

Figure 2. Boundary conditions and initial conditions layout.

In figure 3 we show the expected flow regimes according to the Reynolds number. As we are targeting for a Reynolds number of 200 we will be simulating a laminar case; therefore, we do not need to use any turbulence model.

As illustrated in figure 3, we also expect the onset of the von Karman vortex street behind the cylinder. Henceforth, this is an unsteady aerodynamics case and we need to choose the time step of the simulation in such a way to get an accurate and stable solution.

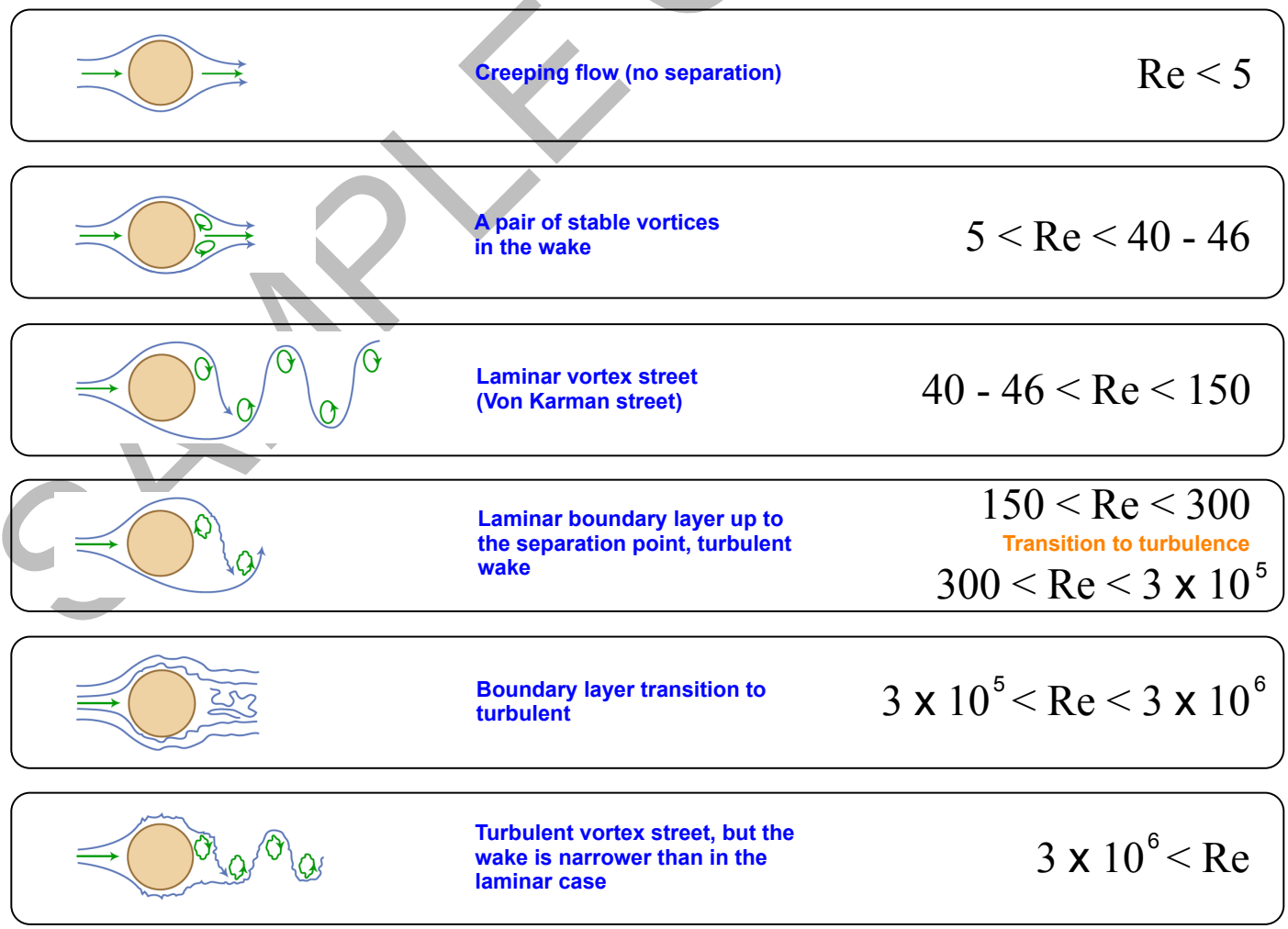

Figure 3. Flow regime according to the Reynolds number.

As this is an incompressible case and a single-phase simulation, we only need to set the kinematic viscosity for the working fluid. Therefore, the inlet velocity needs to be set in such a way to get a Reynolds number of `200`. Let us recall that,

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

where

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

Finally, it is always a good idea to take a look at the governing equations underlying the physics involved. In this case, we are going to solve the incompressible Navier-Stokes equations, which are given as follow,

$$\nabla \cdot (\mathbf{u}) = 0,$$
$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{uu}) = \frac{-\nabla p}{\rho} + \nu \nabla^2 \mathbf{u},$$

To run the simulation, we need to define the physical properties appearing in the governing equations. We also need to define the boundary conditions and initial conditions of the primitive variables appearing in the governing equations.

WARNING:

When using incompressible solvers, the pressure value saved by OpenFOAM® is the density pressure, that is, pressure divided density. Hence, when doing post-processing and data analysis it is essential to multiply the saved pressure value by the density value of the working fluid, this is done to obtain the right pressure units.

4. Geometry generation

No need to use any CAD or solid modeling application to generate the geometry.

The geometry is defined in the `blockMeshDict` dictionary, which is located in the case directory:

```
constant/polyMesh/
```

5. Meshing

The mesh is generated using `blockMesh` meshing utility. The meshing utility `blockMesh` is a three dimensional multi-block mesh generator. The mesh is generated from the dictionary `blockMeshDict`, this dictionary is located in the case directory:

```
constant/polyMesh/
```

In the dictionary `blockMeshDict` we define the geometry (vertices and edges), the block topology (how vertices are connected, the number of cells in each direction and the cell expansion ratio), and the boundary patches (the patches where we are going to assign boundary conditions values). Let us open the `blockMeshDict` dictionary, type in the terminal:

```
gedit constant/polyMesh/blockMeshDict &
```

The coordinates of the vertices that define the geometry are entered in the block list `vertices` of the dictionary `blockMeshDict`,

```
19        vertices
20        (
21             //back up
22             (1   0 -0.5)                    //0
23             (3   0 -0.5)                    //1
24             (30  0 -0.5)                    //2
25             (30 2.12132 -0.5)               //3
26             (2.12132 2.12132 -0.5)          //4

               ...
               ...
               ...

93        );
```

NOTE:

In the listings, the numbers colored in magenta represents the line number in the dictionary. The blue characters are comments.

NOTE:

As in C++, in OpenFOAM® every index label starts from `0`.
For example, the vertices label numbering starts from `0`.

Each edge joining 2 vertex points is assumed to be straight by default. However, any edge may be specified to be curved by entries in the block list named `edges`. This list is optional; if the geometry contains no curved edges, it may be omitted. In this case, we define the arcs that made up the cylinder geometry in the block list `edges`,

```
122         edges
123         (
124             //up
125             arc 0  5   (0.939693  0.34202  -0.5)
126             arc 5  10  (0.34202   0.939693 -0.5)
127             arc 19 24  (0.939693  0.34202   0.5)
128             arc 24 29  (0.34202   0.939693  0.5)
129             arc 11 16  (-0.939693 0.34202  -0.5)
130             arc 16 10  (-0.34202  0.939693 -0.5)

                ...
                ...
                ...

161         );
```

To define the `edges`, interpolation points must be specified through which the edge passes. For an edge type `arc`, a single interpolation point is required; the circular arc will intersect this interpolation point.

To get more information about the curved edge types available and how to use them, please refer to the OpenFOAM® user guide [1].

The next block list in the dictionary `blockMeshDict`, contains the block topology of the multi-block mesh. Each block definition is a compound entry consisting of a block type identifier, a list of vertex labels, a vector giving the number of cells required in each direction, and the type and list of cell expansion ratio in each direction. For this case, the blocks are defined as follows:

```
95          blocks
96          (
97                //up
98                hex (5 4 9 10 24 23 28 29) (20 10 1) simpleGrading (2 1 1) //0
99                hex (0 1 4 5 19 20 23 24)  (20 10 1) simpleGrading (2 1 1) //1
100               hex (1 2 3 4 20 21 22 23)  (50 10 1) simpleGrading (6 1 1) //2
101               hex (4 3 6 7 23 22 25 26)  (50 30 1) simpleGrading (6 7 1) //3
102               hex (9 4 7 8 28 23 26 27)  (10 30 1) simpleGrading (1 7 1) //4

                  ...
                  ...
                  ...

120         );
```

At this point, if you want to change the dimensions of the domain, the diameter of the cylinder, the number of cells, or the expansion ratio, feel free to modify any of the aforementioned block lists.

The computational domain boundaries are broken into patches (regions) that are defined in the block list `patches`. In this block list, we define the type and name of every boundary patch. The name of the boundary patch is a user choice, whereas the type of the patch has to be a valid base type (for a detailed description of the base type patches available, refer to the OpenFOAM® user guide [1] or the `doxygen` documentation [2]).

As `blockMesh` is a three dimensional multi-block mesh generator, for two-dimensional cases we need to use one cell in the direction normal to the faces where we do not want to compute the solution, in this case, the *z-direction*. These faces need to be defined as `empty` patches. An `empty` patch is a patch where we do not want to compute the solution.

In the block list `patches`, each boundary patch is defined by a list of `4` vertex numbers. The name of the boundary patch is used as an identifier for setting boundary conditions in the field data files. The patch information is described as follows:

```
163       patches
164       (
165
166            patch out
167            (
168                 //up
169                 (2 3 22 21)
170                 (3 6 25 22)
171
172                 //down
173                 (38 2 21 51)
174                 (41 38 51 54)
175
176            )
177            symmetryPlane sym1
178            (
179                 (7  8  27 26)
180                 (6  7  26 25)
181                 (8  18 37 27)
182                 (18 17 36 37)
183            )

               ...
               ...
               ...

245       );
```

For example, in lines `166-176` we define the boundary patch `out`, which has a base type `patch`, and is made up by the faces `(2 3 22 21)`, `(3 6 25 22)`, `(38 2 21 51)`, and `(41 38 51 54)`. Then, the face `(2 3 22 21)` is made up by connecting the vertices `2`, `3`, `22` and `21` (the vertices must be contiguous). The rest of the boundary patches are constructed in a similar way.

The meshing utility `blockMesh` collects all boundary faces from any boundary patch that is omitted from the boundary list and assigns them to a default patch named `defaultFaces` of type `empty`. In this case, as we did not omit any boundary face, there is no `defaultFaces` patch.

The patch type `empty` is used to define two-dimensional cases. Notice that for two-dimensional cases, we need to use one cell in the direction normal to the `empty` faces or the third dimension (take a look at the block list `blocks`). You have the option to omit the block faces that lies in the `2D` plane, knowing that they will be grouped into an `empty` patch. However, in this case we explicitly define the `empty` patches,

```
163         patches
164         (

                ...
                ...
                ...


220             empty back
221             (
222                 //up
223                 (5 10 9 4)
224                 (0 5 4 1)

                ...
                ...
                ...

247             empty front
248             (
249                 //up
250                 (24 23 28 29)
251                 (20 23 24 19)

                ...
                ...
                ...

272             )
273
274         );
```

If you want to learn more about `blockMesh`, refer to the OpenFOAM® user guide [1].

At this point, we can generate the mesh. From within the case directory, type in the terminal:

```
blockMesh
```

The running status of `blockMesh` is reported in the terminal window. There should be no error messages at this stage.

If you modified the `blockMeshDict` dictionary and you are getting errors, in the terminal you will get a message telling you what is the problem and the line where the problem occurred.

Let us now check the mesh quality, to do this we use the utility `checkMesh`. From within the case directory, type in the terminal:

```
checkMesh
```

The utility `checkMesh` gives the mesh statistics, will look for topological errors and will report the mesh quality.

Topological errors must be repaired. If a mesh quality check fails we still can run the simulation, this simply means that we are using a low quality mesh; however, have in mind that this will affect the stability and the accuracy of the solution.

At this point, take a look at the `checkMesh` report.

No single standard mesh quality metric exists that can effectively assess the quality of the mesh. The most important quality metrics are: `Max aspect ratio`, `Mesh non-orthogonality` and `Max skewness`. After running `checkMesh` and if you did not modify the `blockMeshDict` dictionary, you should get the following values:

```
Max aspect ratio = 6.88512 OK.

Mesh non-orthogonality Max: 43.434 average: 10.3505
Non-orthogonality check OK.

Max skewness = 0.455112 OK.
```

An acceptable `Max aspect ratio` value should be less than `1000`. Higher values are acceptable, but have in mind that high aspect ratio values will smear the gradients in the direction of the longest cell side.

Values of `Mesh non-orthogonality Max` up to `70` are acceptable. For higher values you should adjust the numerical scheme and solution method to take care for the mesh non-orthogonality. Remember, the higher the mesh non-orthogonality the more diffusion you will be adding to the solution. If you get values of `Mesh non-orthogonality Max` of more than `85`, you should consider remeshing the domain.

Mesh skewness is highly related to the mesh non-orthogonality, most of the time if you have high skew cells you will get high non-orthogonal cells, and vice-versa. You can safely run a simulation with values of `Max skewness` up to `4`. You can run with higher values of `Max skewness` but have in mind that this will affect the accuracy and stability of the solution.

If you want to visualize the mesh, from within the case directory type in the terminal:

```
paraFoam
```

If you are interested in visualizing the mesh block topology, from within the case directory type in the terminal:

```
paraFoam -block
```

If you are new to `paraFoam`, in section 12 we briefly explain how to use it.

6. Physical properties

The physical properties are defined in the dictionary `transportProperties`, this dictionary is located in the case directory:

```
constant/
```

As this is an incompressible case, we only need to define the kinematic viscosity `nu`.

Remember, the cylinder diameter is `2.0 m` and the inlet velocity is `1.0 m/s`, hence to get a Reynolds number of `200` the kinematic viscosity `nu` needs to be equal to `0.01`.

Let us open the `transportProperties` dictionary, from within the case directory type in the terminal:

```
gedit constant/transportProperties &
```

In the dictionary, you will find the dimensions and the value of the kinematic viscosity `nu`,

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

The numbers between the square brackets represent the kinematic viscosity units. The input format for the dimensional units is given by the seven scalars delimited by the square brackets, the properties and units of the seven scalars are shown in table 1.

No.	Property	SI unit	
1	Mass	kilogram	(kg)
2	Length	meter	(m)
3	Time	second	(s)
4	Temperature	Kelvin	(K)
5	Quantity	kilogram-mole	(kgmol)
6	Current	Ampere	(A)
7	Luminous intensity	Candela	(cd)

Table 1. Base units

When defining the dimensional units, each of the values of the seven scalars corresponds to the power of each of the base units of measurement as listed in table 1. For example, the units of the kinematic viscosity `nu` are:

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

The physical value of the kinematic viscosity `nu` is defined after the square brackets. In this case, the value of `nu` is equal to `0.01`.

The base units of any scalar, vector, or tensor field are defined in a similar way. In this case if you change the dimensional units of `nu` (you give the wrong dimensions), OpenFOAM® will complain.

WARNING:

Remember, OpenFOAM® is fully dimensional.
If you use the wrong dimensions, OpenFOAM® will complain.

If you want to use a different Reynolds number, you can change the value of `nu`.

Alternatively, you can change the value of the free stream velocity or the diameter of the cylinder.

7. Turbulence modeling

No turbulence model is used in this tutorial.

8. Advanced physical models

No advanced physical models are used in this tutorial.

9. Boundary conditions and initial conditions

The boundary conditions and initial conditions are defined in the field dictionaries located in the directory `0`. The field dictionaries contain three block entries:

`dimensions`: this block specifies the dimensions of the field variable.

`internalField`: in this block we define the initial conditions.

`boundaryField`: in this block we choose the numerical type of each boundary patch and its physical value.

Remember, for each single variable to be solved we must define the boundary and initial conditions.

As we are solving the Navier-Stokes equations with no turbulence modeling, we only need to define boundary and initial conditions for the pressure (scalar field `p`) and the velocity (vector field `U`).

Let us take a look at the field dictionary `p`, from within the case directory type in the terminal:

```
gedit 0/p &
```

The first block in the field dictionary is the `dimensions` block. Usually you do not need to modify this block entry. In this case, for the scalar field `p` the dimensions are:

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

In the `internalField` block you will set the desired initial conditions. Most of the times you will use a uniform field initial condition of `uniform` type. In this case, for the scalar field `p` the initial conditions are:

```
20   internalField   uniform 0;
```

This means that we are initializing a uniform field with a value of `0` in the whole domain. In this case, the pressure value is relative to a reference pressure (*e.g.* atmospheric pressure).

In this case there are seven boundary patches, namely: `in`, `out`, `cylinder`, `sym1`, `sym2`, `back`, and `front`. For the scalar field `p`, the numerical type and value of each boundary patch is defined as follows:

```
22   boundaryField
23   {
24        in
25        {
26             type            zeroGradient;
27        }
28
29        out
30        {
31             type            fixedValue;
32             value           uniform 0;
33        }
34
35        cylinder
36        {
37             type            zeroGradient;
38        }
39
40        sym1
41        {
42             type            symmetryPlane;
43        }
44
45        sym2
46        {
47             type            symmetryPlane;
48        }
49
50        back
51        {
52             type            empty;
53        }
54
55        front
56        {
57             type            empty;
58        }
59   }
```

Let us now take a look at the field dictionary `U`, from within the case directory type in the terminal:

```
gedit 0/U &
```

For the vector field `U` the dimensions are:

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

For the same field variable, the initial conditions are:

```
20   internalField   uniform (1 0 0);
```

Notice that as the velocity is a vector, we define the vector components between parentheses, where `u = 1`, `v = 0` and `w = 0`.

We now define the numerical type and value of each boundary patch for the vector field `U`,

```
22   boundaryField
23   {
24        in
25        {
26   /*
27             type            uniformFixedValue;
28             uniformValue     table
29             (
30                  (0   (1 0 0))
31                  (20  (2 0 0))
32                  (40  (1 0 0))
33             );
34   */
35
36             type            fixedValue;
37             value           uniform (1 0 0);
38        }
```

```
40          out
41          {
42             //type            zeroGradient;
43
44               type            inletOutlet;
45               phi             phi;
46               inletValue      uniform (0 0 0);
47               value           uniform (1 0 0);
48          }
49
50          cylinder
51          {
52               type            fixedValue;
53               value           uniform (0 0 0);
54          }
55
56          sym1
57          {
58               type            symmetryPlane;
59          }
60
61          sym2
62          {
63               type            symmetryPlane;
64          }
65
66          back
67          {
68               type            empty;
69          }
70
71          front
72          {
73               type            empty;
74          }
75     }
```

In the field dictionaries, the `fixedValue` numerical type boundary condition is a *Dirichlet* boundary condition, and we need to set its value, *e.g.*,

```
36          type            fixedValue;
37          value           uniform (1 0 0);
```

The `zeroGradient` numerical type boundary condition is a *Neumann* boundary condition. When we use this boundary condition we are basically extrapolating the interior values to the boundaries.

The `symmetryPlane` numerical type boundary condition is used to define a slip wall, notice that symmetrical boundary conditions are only valid for planar walls. Alternatively, you can use a `slip` numerical type boundary condition.

The `inletOutlet` numerical type boundary condition is used to define an outflow condition. If the flow is going out the domain it will use a *Neumann* boundary condition, and if the flow is coming back into the domain it will use a *Dirichlet* boundary condition to force the flow to go out. The value of the *Dirichlet* boundary condition is set by the keyword `value`.

The `empty` numerical type boundary condition is used to define a two-dimensional case. No solution is required in the direction normal to these two planes.

The cylinder wall is set using a *Dirichlet* boundary condition as follows,

```
52          type            fixedValue;
53          value           uniform (0 0 0);
```

Remember, you need to use the same naming convention between the field dictionaries and the file containing the boundary patches information (base type and patch name), this information is located in the `boundary` dictionary. The `boundary` dictionary is automatically created when the mesh is generated. This dictionary is located in the directory:

```
constant/polyMesh/
```

At this point, if you open the file `constant/polyMesh/boundary` and the files `0/U` and `0/p`, you will see that the naming convention of the boundary patches between the files is consistent.

If you misspell the name of a boundary patch or change the name of a boundary patch in a file, OpenFOAM® will complain and will tell you that there is a boundary patch naming mismatch between files. Read carefully the terminal, because OpenFOAM® will tell you where is the error.

Also, the numerical type (or the patch type defined in the field dictionaries), and the base type (or the patch type defined in the `boundary` dictionary), they need to be consistent. That is, if in the `boundary` file you define a patch as `empty`, it must be of the numerical type `empty` in the field dictionaries `U` and `p`. If you define the base type as `symmetryPlane`, the numerical type must be `symmetryPlane`. If you define the base type as `patch`, the numerical type can be any of the *Dirichlet*, *Neumann*, or *Robin* numerical types available in OpenFOAM®.

Have in mind that to define no-slip walls you use a numerical type boundary condition of the type `fixedValue` with a value entry `uniform (0 0 0)`. However, in the `boundary` file the base type must be of the type `wall`, as follows,

```
48          type      wall;
```

NOTE:

Numerical type boundary conditions are defined in the field dictionaries located in the directory `0/`
Base type boundary conditions are defined in the `boundary` dictionary located in the directory `constant/polyMesh/`

For a detailed description of the numerical type and base type boundary conditions available, refer to the OpenFOAM® user guide [1] or the `doxygen` documentation [2].

At this point, if you want to modify a boundary condition you can simply change its type or value. For example, to change the value of the velocity from `(1 0 0)` to `(2 0 0)` in the boundary patch `in`, modify the block entry as follows:

```
24          in
25          {
26      /*
27                  type            uniformFixedValue;
28                  uniformValue     table
29                  (
30                       (0    (1 0 0))
31                       (20   (2 0 0))
32                       (40   (1 0 0))
33                  );
34      */
35
36                  type            fixedValue;
37                  value           uniform (2 0 0);
38          }
```

You can also modify the type and value of the initial conditions. For example, to change the value of the initial pressure in the whole domain, modify the block entry `internalField` as follows:

```
20    internalField   uniform 101325;
```

If you decide to change the inlet velocity from `(1 0 0)` to `(2 0 0)`, it is also reasonable to change the value of the initial conditions for the velocity field `U` as follows:

```
20    internalField   uniform (2 0 0);
```

WARNING:

Boundary conditions and initial conditions need to be physically realistic. Poorly chosen boundary or initial conditions can slow down the convergence rate, affect the accuracy of the solution, generate misleading results, or can cause divergence.

10. Solution method and solution monitoring

Before running the solver we need to choose the discretization method, the linear solvers and the solution control parameters.

The discretization method is set in the dictionary `fvSchemes`, the linear solvers are set in the dictionary `fvSolution`, and the solution control parameters are set in the dictionary `controlDict`. These dictionaries are located in the directory:

```
system/
```

Let us now take a look at the dictionary `fvSchemes`, from within the case directory type in the terminal:

```
gedit system/fvSchemes &
```

In this dictionary we set the numerical schemes to be used to discretize each term appearing in the governing equations. This is done in a term-by-term basis.

WARNING:

If you do not set the numerical scheme entry for a given term, OpenFOAM® will complain and it will tell you where is the error and what is the missing keyword in the `fvSchemes` dictionary.

For the time derivative `ddtSchemes`, we use the following discretization method:

```
18  ddtSchemes
19  {
20      //default         Euler;
21      default         backward;
22      //default     CrankNicolson 0.5;
23  }
```

The `backward` method is a second order accurate temporal discretization method. The `default` keyword means that we are using this method for every single time derivative appearing in the governing equations.

For computing the gradients `gradSchemes`, we use the following discretization method:

```
25   gradSchemes
26   {
27   /*
28       default         Gauss linear;
29       grad(p)         Gauss linear;
30   */
31
32        default        leastSquares;
33
34        //default           cellMDLimited leastSquares 0.5;
35        //default           cellMDLimited Gauss linear 1;
36
37        //grad(U)           cellMDLimited Gauss linear 1;
38   }
```

The `leastSquares` method is a second order accurate discretization method and gives better accuracy and more stability over the `Gauss linear` method.

The `default` keyword means that we are using the method for every single gradient to be computed. If for any reason you would like to use different discretization methods to compute the gradient of `p` and the gradient of `U`, you can proceed as follows:

```
25   gradSchemes
26   {
27
28        //default         Gauss linear;
29        grad(p)         Gauss linear;
30
31
32        //default           leastSquares;
33
34        //default           cellMDLimited leastSquares 0.5;
35        //default           cellMDLimited Gauss linear 1;
36
37        grad(U)         cellMDLimited Gauss linear 1;
38   }
```

In this case to compute `grad(p)` we use the `Gauss linear` discretization method, which is second order accurate but it might produce under-shoots and over-shoots (oscillations).

To compute `grad(U)` we use the `cellMDLimited Gauss linear 1` discretization method (fully limited). The `cellMDLimited` keyword means that we are using slope or gradient limiter, which help in preventing spurious oscillations.

When using slope limiters we need to define a blending factor. The blending factor can vary between `0` and `1`. In this case, we are using a blending factor of `1`, which means that the limiter is always on. This translates in a very stable but highly diffusive numerical solution.

By setting the blending factor to `0`, we are turning the slope limiter off. This is equivalent to using a pure `Gauss linear` method.

If you set the blending factor to `0.5`, you are getting a compromise between the pure `Gauss linear` method and the fully limited `cellMDLimited Gauss linear` method. This numerical scheme is more accurate than the fully limited `cellMDLimited Gauss linear` method, however it might suffer of spurious oscillations. Compare to the pure `Gauss linear`, it is a little bit more diffusive but more stable.

For the computation of the convective terms `divSchemes`, we use the following discretization method:

```
40   divSchemes
41   {
42        default         none;
43        //div(phi,U)      Gauss linear;
44        //div(phi,U)      Gauss limitedLinearV 1;
45
46        div(phi,U)      Gauss linearUpwind default;
47        //div(phi,U)      Gauss linearUpwind grad(U);
48
49        //div(phi,U)      Gauss upwind;
50   }
```

In this case we are using the second order linear upwind (SOU) discretization scheme or `linearUpwind`. This numerical scheme is second order accurate.

You can also use a more stable and still second order accurate version of the SOU scheme, by using a gradient or slope limiter. The slope limiters avoid under-shoots and over-shoots.

To use this version of the SOU scheme, you will need to modify the `divSchemes` as follows,

```
40   divSchemes
41   {
42        default          none;
43        //div(phi,U)       Gauss linear;
44        //div(phi,U)       Gauss limitedLinearV 1;
45
46        //div(phi,U)       Gauss linearUpwind default;
47        div(phi,U)       Gauss linearUpwind grad(U);
48
49        //div(phi,U)       Gauss upwind;
50   }
```

Remember, you also need to define how to compute `grad(U)` in `gradSchemes`. If you want to use a very aggressive slope limiter for `grad(U)`, you can proceed as follows,

```
     grad(U)         cellLimited Gauss linear 1;
```

You can also use a pure linear numerical scheme (`Gauss linear`). The `Gauss linear` numerical scheme is second order accurate and has higher accuracy than the SOU method, but for highly convective flows can give oscillatory solutions.

At this point, feel free to choose between `Gauss linear` or `Gauss linearUpwind`. Also, try to use different slope limiters for the SOU method.

For the computation of the laplacian terms `laplacianSchemes`, we use the following discretization method:

```
52   laplacianSchemes
53   {
54   /*
55       default          none;
56       laplacian(nu,U) Gauss linear corrected;
57       laplacian((1|A(U)),p) Gauss linear corrected;
58   */
59
60       default          Gauss linear limited 1;
61   }
```

Hereafter, we are using the `Gauss linear limited` discretization method. The `limited` keyword means that we are using a numerical scheme with limited non-orthogonal corrections.

When using this method we need to define a blending factor. The blending factor can vary between `0` and `1`. In this case we are using a blending factor of `1`.

Using `Gauss linear limited 1` is equivalent to use `Gauss linear corrected`.

The value of the blending factor depends of the quality of the mesh. For low quality meshes you can use a blending factor of 0.5. A low quality mesh can be considered a mesh with a value of `Mesh non-orthogonality Max` of more than `70`.

In this case, as the value of `Mesh non-orthogonality Max` reported by the `checkMesh` utility is `43.434`, using a blending factor of `1` is acceptable.

The next block entry to define in the `fvSchemes` dictionary is the way in which the surface normal gradients `snGradSchemes` are computed. This entry is related to the `laplacianSchemes` and is set in the following way:

```
69    snGradSchemes
70    {
71         //default         corrected;
72         default         limited 1;
73    }
```

When setting the `snGradSchemes`, we use the same method as the one used to compute `laplacianSchemes`.

In this case, to compute `laplacianSchemes` we used the method `limited 1`. Hence, to compute `snGradSchemes` we need to use the same method, namely, `limited 1`.

The `interpolationSchemes` entry in the `fvSchemes` dictionary defines how the cell-centered values are interpolated to the face centers. This entry is set in the following way:

```
63    interpolationSchemes
64    {
65         default           linear;
66         //interpolate(HbyA) linear;
67    }
```

In general, linear interpolation is the common choice and is second order accurate.

NOTE:

If you want to know more about all the numerical schemes available in OpenFOAM®, refer to the user guide [1] or the `doxygen` documentation [2].

Let us now take a look at the dictionary `fvSolution`, from within the case directory type in the terminal:

```
gedit system/fvSolution &
```

In this dictionary we select the linear solver to use to solve the linear system arising from the discretization of the equations. For each variable to be solved, we need to define the linear solver to be used.

WARNING:

If you do not set the linear solver entry for a given primitive variable, OpenFOAM® will complain and it will tell you where is the error and what is the missing keyword in the `fvSolution` dictionary.

For the primitive variable `p`, we use the following linear solver:

```
32  p
33  {
34      solver          GAMG;
35      tolerance       1e-6;
36      relTol          0;
37      smoother        GaussSeidel;
38      nPreSweeps      0;
39      nPostSweeps     2;
40      cacheAgglomeration on;
41      agglomerator    faceAreaPair;
42      nCellsInCoarsestLevel 100;
43      mergeLevels     1;
44  }
```

Hereafter, we area using the `GAMG` solver with an absolute tolerance of `1e-6` and a relative tolerance of `0`. The linear solver will iterate until reaching the desired tolerance.

For the primitive variable `U`, we use the following linear solver:

```
46    U
47    {
48        solver          PBiCG;
49        preconditioner  DILU;
50        tolerance       1e-08;
51        relTol          0;
52    }
```

Hereafter, we are using the `PBiCG` solver with the matrix preconditioner `DILU`. The absolute tolerance is set to `1e-8` and a relative tolerance of `0`. The linear solver will iterate until reaching the desired tolerance.

It worth mentioning that the solvers distinguish between symmetric matrices and asymmetric matrices. The symmetry of the matrix depends on the structure of the equation being solved. Diffusion equations are always symmetric (*e.g.* pressure equation and heat diffusion), while convective equations are asymmetric.

If you select the wrong solver for the equation to be solved, OpenFOAM® will produce an error message and it will show a message with the valid solvers.

Finally, we need to set the parameters for the pressure-velocity coupling method. Hereafter, we are using the `PISO` method. The options concerning to the `PISO` method are set as follows:

```
68    PISO
69    {
70        nCorrectors     2;
71        nNonOrthogonalCorrectors 2;
72        //pRefCell        0;
73        //pRefValue       0;
74    }
```

In this case, we are using `2` `PISO` correctors (`nCorrectors`).

The entry `nNonOrthogonalCorrectors` or the number of non-orthogonal corrections is related to the mesh non-orthogonality. This option is available for both `PISO` and `SIMPLE` pressure-velocity coupling methods.

The number of `nNonOrthogonalCorrectors` corrections to be used depends on the quality of the mesh. The lower the mesh quality, the more corrections we need to use to approximate better the gradients. In this case, we use `2` non-orthogonal corrections.

For non-orthogonal meshes, it is always a good idea to use at least one correction.

For perfect orthogonal and uniform meshes we can set to `0` the number non-orthogonal corrections. However, these kinds of meshes are the exception rather than the rule.

The entries `pRefCell` and `pRefValue` are used to set the location and the value of the relative pressure. These values need to be set when working with closed incompressible systems.

NOTE:

If you want to know more about all the linear solvers available in OpenFOAM®, refer to the user guide [1] or the `doxygen` documentation [2].

Let us now take a look at the dictionary `controlDict`, from within the case directory type in the terminal:

```
gedit system/ controlDict &
```

In this dictionary we set time control, data input/output control parameters and function objects. Information such as: time-step, end-time, solution saving frequency, and forces computation; are set in this dictionary. The dictionary `controlDict` is organized as follows:

```
18   application     icoFoam;
19
20   //startFrom       startTime;
21   startFrom       latestTime;
22
23   startTime       0;
24
25   stopAt          endTime;
26   //stopAt          writeNow;
27
28   endTime         350;
29   //endTime         500;
30
31   //deltaT          0.005;
32   //deltaT          0.01;
33   //deltaT          0.02;
34   deltaT          0.05;
35
36   writeControl    runTime;
37   /*
38   adjustableRunTime
39   clockTime
40   cpuTime
41   runTime
42   timeStep
43   */
44
45   writeInterval   1;
```

If we take a look at the previous listing, we first notice that the simulation starts from the latest saved solution. This is set in the entry `startFrom latestTime`. If you want to start the simulation from the starting time (`0` in this case), you can change this keyword to `startTime`.

We also notice that the simulation starting time is `0`, this is specified in the entry `startTime 0`. Hence, we need to set the boundary and initial conditions in the time directory `0`. If you use a different value, *e.g.* `50`, you will need to set the boundary and initial conditions in the time directory `50`. If you already have a solution, the solver will start from this time directory.

The simulation will stop when it reaches the ending time, which is defined using the entry `stopAt endTime`. In this case the ending time is equal to `endTime 350`.

If while running the simulation you want to stop the simulation and save the latest time-step, you can change the keyword `stopAt endTime` to `stopAt writeNow`.

As this is an unsteady simulation, we need to set the time-step of the simulation. Using the keyword `deltaT` sets the time step. In this case, the time-step is equal to `0.05`.

The time step of the simulation needs to be wisely chosen, very large time-step will give inaccurate results or oscillatory solutions, while very small time-step will translate in very long computation time. In this case, the time-step is chosen in such a way that the CFL number is less than `1`.

The CFL information can be seen while running the simulation, we are going to address this issue in the next section.

Next, we choose the saving frequency and saving method. In this case, we are saving the solution every second of simulation physical time. The way the solution is saved is set by the keyword `writeControl`, and the saving frequency is set by the keyword `writeInterval`.

The rest of the dictionary `controlDict`, is organized as follows:

```
55   purgeWrite      0;
56
57   writeFormat     ascii;
58
59   writePrecision  6;
60
61   writeCompression off;
62
63   timeFormat      general;
64
65   timePrecision   6;
66
67   runTimeModifiable true;
```

The `purgeWrite` control is used to define the number of solutions to keep. In this case is set to `0`, meaning that we want to keep all the solutions. If you use a different value, *e.g.* `10`, it will keep the latest `10` solutions.

The `writeFormat` control specifies the format of the data files. You can choose between `ascii` and `binary` format. For large meshes it is convenient to use `binary` format.

The `writePrecision` is used together with the `writeFormat` control. It defines the precision of the solution written by the solver. Be careful, this refers to the quantity of digits to be saved and not to the number of significant decimal digits to be used.

The `writeCompression` control specifies if we want to save the data in compressed format. In this case we are not using compression.

The `timeFormat` entry is used to define the format of the time reported. You can choose between `fixed`, `scientific` and `general`. Most of the time you will use the `general` format.

The `timePrecision` control is used together with the entry `timeFormat`, it defines the precision of the time reported by the solver.

Remember, you can change all these parameters and most of the parameters of all the dictionaries while the simulation is running. To do this, you will need to enable the option to re-read the dictionaries at the beginning of each time-step. This is done as follows:

```
67    runTimeModifiable true;
```

At the end of the dictionary `controlDict`, you will find the block list `functions`.

```
73    functions
74    {

          ...
          ...
          ...
          ...
          ...
          ...

270   };
```

This block list corresponds to the function objects entries. Functions objects are small programs used to do some operations while the simulation is running.

For instance, function objects can be used to compute the aerodynamic coefficients. This is shown in the following listing:

```
208  forceCoeffs_object
209  {
210       // rhoInf - reference density
211       // CofR - Centre of rotation
212       // dragDir - Direction of drag coefficient
213       // liftDir - Direction of lift coefficient
214       // pitchAxis - Pitching moment axis
215       // magUinf - free stream velocity magnitude
216       // lRef - reference length
217       // Aref - reference area
218       type forceCoeffs;
219       functionObjectLibs ("libforces.so");
220       //patches ("body1" "body2" "body3");
221       patches (cylinder);
222
223       pName p;
224       Uname U;
225       rhoName rhoInf;
226       rhoInf 1.0;
227
228       //// Dump to file
229       log true;
230
231       CofR (0.0 0 0);
232       liftDir (0 1 0);
233       dragDir (1 0 0);
234       pitchAxis (0 0 1);
235       magUInf 1.0;
236       lRef 1;            // reference length for moments!!!
237       Aref 2.0;          // reference area 1 for 2d
238
239       outputControl   timeStep;
240       outputInterval  1;
241  }
```

This function object will compute the aerodynamic coefficient in the boundary patch `cylinder`. The values are normalized using `rhoInf` and `Aref` reference values.

In this entry, we also set the saving frequency, which can be different from the saving frequency of the solution.

The output of all function objects is saved in the directory `postProcessing`, this directory is automatically created by the solver. You will find the values of this specific function object in the directory `postProcessing/forceCoeffs_object`.

An extremely useful function object is `fieldMinMax`. This function object can be used as follows:

```
249  minmaxdomain
250  {
251       type fieldMinMax;
252       //type banana;
253
254       functionObjectLibs ("libfieldFunctionObjects.so");
255
256       enabled true;
257
258       mode component;
259
260       outputControl timeStep;
261       outputInterval 1;
262
263       log true;
264
265       fields (p U);
266  }
```

This function object will compute the maximum and minimum values of the primitive variables, and will show the information on-screen.

By monitoring the maximum and minimum values you can assess if your solution is stable or converging to a realistic value. If at any point of the simulation you observe unphysical values or oscillatory values, you know that the solver is having problems, so you can try to reduce the time step or change the numerical scheme.

Besides showing the information on-screen, this function object also saves the information in an `ascii` file. You will find the values of this specific function object in the directory `postProcessing/minmaxdomain`.

NOTE:

If you want to know more about all the options available in the `controlDict` dictionary, including the function objects available, refer to the user guide [1] or the `doxygen` documentation [2].

---

11. Running the simulation

This simulation takes approximately `520` seconds running in serial on a `Intel X5670 @2.93 GHz` processor.

WARNING:

Before running the simulation check that you have a valid mesh and that all the dictionaries are properly configured.

To run this simulation, from within the case directory type in the terminal:

```
icoFoam
```

The application running status is reported in the terminal window.

A lot of information is printed in the terminal window. By default, the application running status is not saved in a file. If you stop the simulation or close the terminal window, you will lose this information.

To save the application running status you should redirect the standard output stream (`stdout`) to a log file. To do so, type in the terminal:

```
icoFoam > log 2>&1 | tail –f log
```

This will run the solver `icoFoam`, it will redirect the standard output stream (`stdout`) to the file `log` (`> log`), it will also redirect the standard error stream (`stderr`) to the file `log` (`2>&1`), and then it will print on screen the information that is being saved in the file `log` (`tail –f log`).

You can use this method to save the running status of any solver or utility.

NOTE:

It is highly recommended to always save the application running status in a log file. This information can be used to assess the convergence rate or for troubleshooting.

Let us study one snapshot of the terminal window:

```
Time = 1

Courant Number mean: 0.10582 max: 0.92348
DILUPBiCG:  Solving for Ux, Initial residual = 0.00204804, Final residual = 6.71544e-10, No Iterations 4
DILUPBiCG:  Solving for Uy, Initial residual = 0.00465345, Final residual = 2.62731e-09, No Iterations 4
GAMG:  Solving for p, Initial residual = 0.0233984, Final residual = 5.02867e-07, No Iterations 11
GAMG:  Solving for p, Initial residual = 0.000365225, Final residual = 4.21373e-07, No Iterations 6
GAMG:  Solving for p, Initial residual = 8.72216e-05, Final residual = 7.90181e-07, No Iterations 5
time step continuity errors : sum local = 1.45052e-11, global = 2.81186e-13, cumulative = 2.38061e-10
GAMG:  Solving for p, Initial residual = 0.00622222, Final residual = 6.1394e-07, No Iterations 10
GAMG:  Solving for p, Initial residual = 8.78599e-05, Final residual = 7.77053e-07, No Iterations 4
GAMG:  Solving for p, Initial residual = 2.04239e-05, Final residual = 8.22001e-07, No Iterations 3
time step continuity errors : sum local = 1.51599e-11, global = -1.10055e-13, cumulative = 2.37951e-10
ExecutionTime = 2.04 s  ClockTime = 2 s

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

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

fieldAverage fieldAverage output:
    Calculating averages

    Writing average fields

forceCoeffs forceCoeffs_object output:
    Cm    = -1.54914e-09
    Cd    = 1.18473
    Cl    = -1.54697e-07
    Cl(f) = -7.88974e-08
    Cl(r) = -7.57992e-08

fieldMinMax minmaxdomain output:
    min(p) = -1.1127 at position (0.0406193 -1.03341 0)
    max(p) = 0.580324 at position (-1.03341 0.0406193 0)
    min(U) = (-0.0332043 0.0452098 -6.73375e-19) at position (0.702008 0.759452 0)
    max(U) = (1.61028 0.255946 8.29498e-19) at position (-0.147462 1.24529 0)

Time = 1.05
```

The keyword `Time` is the simulation or physical time. From this terminal window snapshot, we can see that the time begins at `Time = 1` and it ends at `Time = 1.05`, clearly the time-step size is `0.05`.

The keyword `Courant Number`, gives information about the mean and maximum CFL number. In this case and for `Time = 1`, the mean CFL number is equal to `0.10582` and the maximum CFL number is equal to `0.92348`. If you change the time-step, then the new mean and maximum values of the CFL number will be reported on the terminal window.

The next entry in the terminal window snapshot is the linear solvers information and the convergence rate information.

First we get the information of the convergence rate for the velocity vector `U`, and as this is a two-dimensional case only `Ux` and `Uy` are reported.

Then we get the convergence rate information for the pressure `p`. From this output, we can see that we are using `2` non-orthogonal corrections (the two iterations after the initial pressure correction), and `2 PISO` correctors (the second loop of the pressure correction).

The solver running status also reports the continuity errors or `time step continuity errors`. As the method is fully conservative, the continuity errors need to be small. It does not matter if it is are positive or negative. The continuity errors reported in the second `PISO` loop are `sum local = 1.51599e-11, global = -1.10055e-13`, and `cumulative = 2.37951e-10`.

We also get the information of the execution time, this is the actual computing time. For this case the computing time at `Time = 1` is `ExecutionTime = 2.04 s  ClockTime = 2 s`.

The `ExecutionTime` is the time used to compute the solution, whereas the `ClockTime` is the time used to compute the solution plus the added overhead due to data input/output, network traffic and other factors.

The final entry in the terminal window snapshot corresponds to the information of the function objects. In this case we are computing the inlet mass flow `faceSource inMassFlow output:` and the outlet mass flow `faceSource outMassFlow output:`, and as the method is conservative they must balance.

Next, we compute the field average values `fieldAverage fieldAverage output:`. Remember, this information is saved in the time directories, they are part of the solution.

Then we get the information of the aerodynamic coefficients `forceCoeffs forceCoeffs_object output:`.

Finally we get the maximum and minimum values of the field variables `fieldMinMax minmaxdomain output:`. The maximum and minimum values information is extremely useful to judge the stability and convergence of the solution.

In this case, except for the field average all the information of the function objects is saved in the directory `postProcessing`.

At this point and while the simulation is running, let us do some scripting and plotting.

In order to assess the convergence and stability of the simulation, plotting the aerodynamics coefficient as the simulation runs is extremely useful. To do so, you can use the `gnuplot` script `plot_coeffs`, this script is located in the directory:

```
scripts0/
```

To use the script `plot_coeffs`, open a new terminal window and from within the case directory type in the terminal:

```
gnuplot scripts0/plot_coeffs
```

You should get the following window showing the lift coefficient and drag coefficient as the simulation runs.

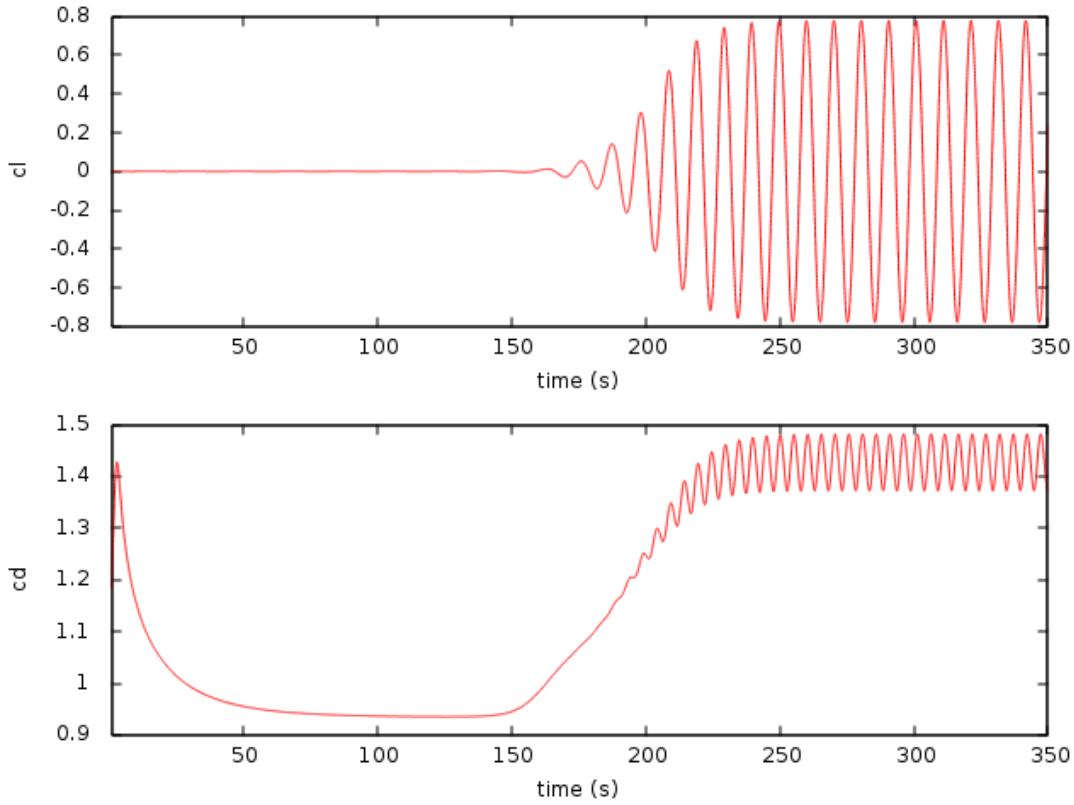

Figure 4. Lift coefficient and drag coefficient plots.

As you can observe in this plot, up to approximately `150` seconds the simulation has a steady behavior, then after `150` seconds the simulation shows a very unsteady behavior and this is reflected on the oscillating behavior of the forces.

The oscillating behavior of the forces is a direct indication of the onset of the von Karman street.

You can also plot the velocity and pressure residuals, to do so you need to save the solver running status in a log file.

To plot the residuals we first need to manipulate the log file. Open a new terminal window and from within the case directory type in the terminal:

```
sh scripts0/edit_log.sh
```

The script `edit_log.sh` will manipulate the log file and it will create a new file named `log_clean.txt`, this is the file that we are going to use to plot the residuals.

Open a new terminal window and from within the case directory type in the terminal:

```
gnuplot scripts0/plot_res1
```

You should get the following window showing the residuals as the simulation runs.

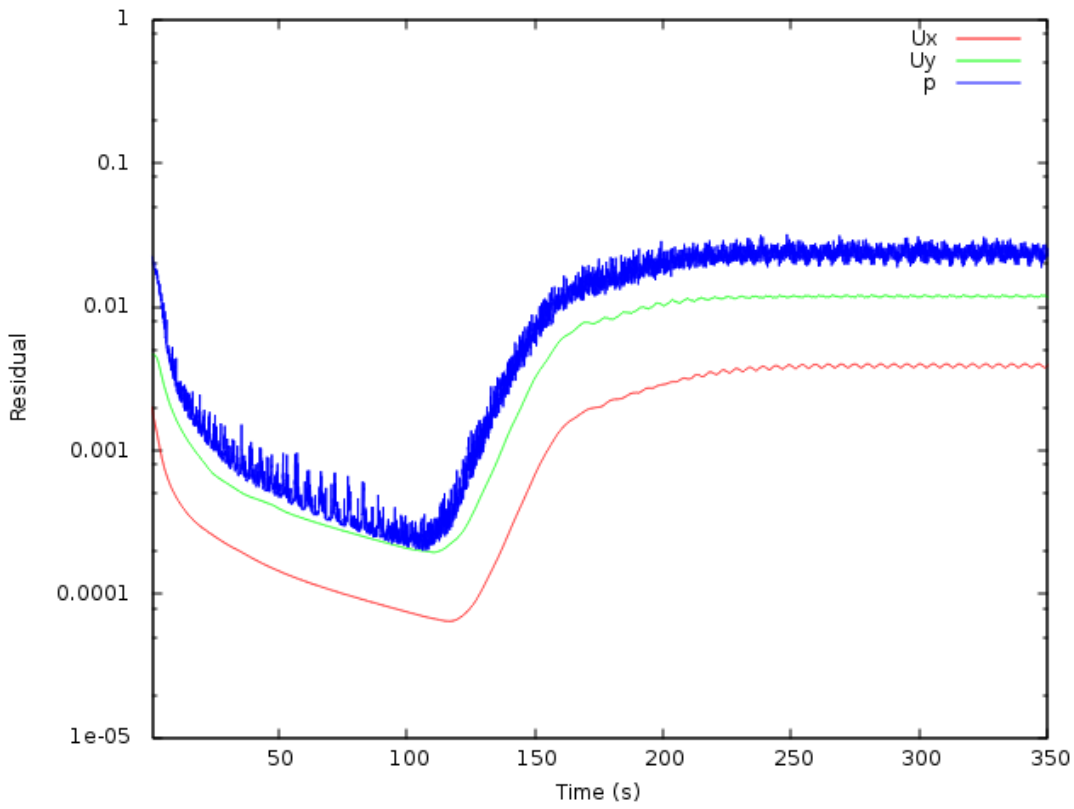

Figure 5. Velocity and pressure residuals.

As you can observe in this plot, in the first `110` seconds the residuals drops monolithically and then they start to increase.

This behavior does not mean that the simulation is diverging, it is a direct indication of the unsteadiness of the solution. If you compare figure 4 with figure 5, you will observe that roughly at the moment when the residuals start to increase, correspond to the instant when the von Karman street is onset.

One word of caution, the script `edit_log.sh` will only work with the default settings used in this tutorial.

For instance, if you change the number of `PISO` corrector steps or the number on non-orthogonal corrections, or you add new equations to the simulation (*e.g.*, turbulence modeling), you will need to modify the script `edit_log.sh`.

Alternatively, you can use the `pyFoam` library to plot the residuals on the fly. To use the `pyFoam` library you will need to install it, as it is not distributed with OpenFOAM®.

In you have `pyFoam`, open a new terminal window and from within the case directory type in the terminal:

```
pyFoamPlotWatcher.py log
```

Finally, in the directory `scripts0` you will find many scripts that you might find useful. Take a look at them and feel free to adapt these scripts to suit your needs.

12. Post-processing

To visualize the solution, from within the case directory type in the terminal:

```
paraFoam
```

You should get a `paraFoam` window similar to the one shown in figure 6.

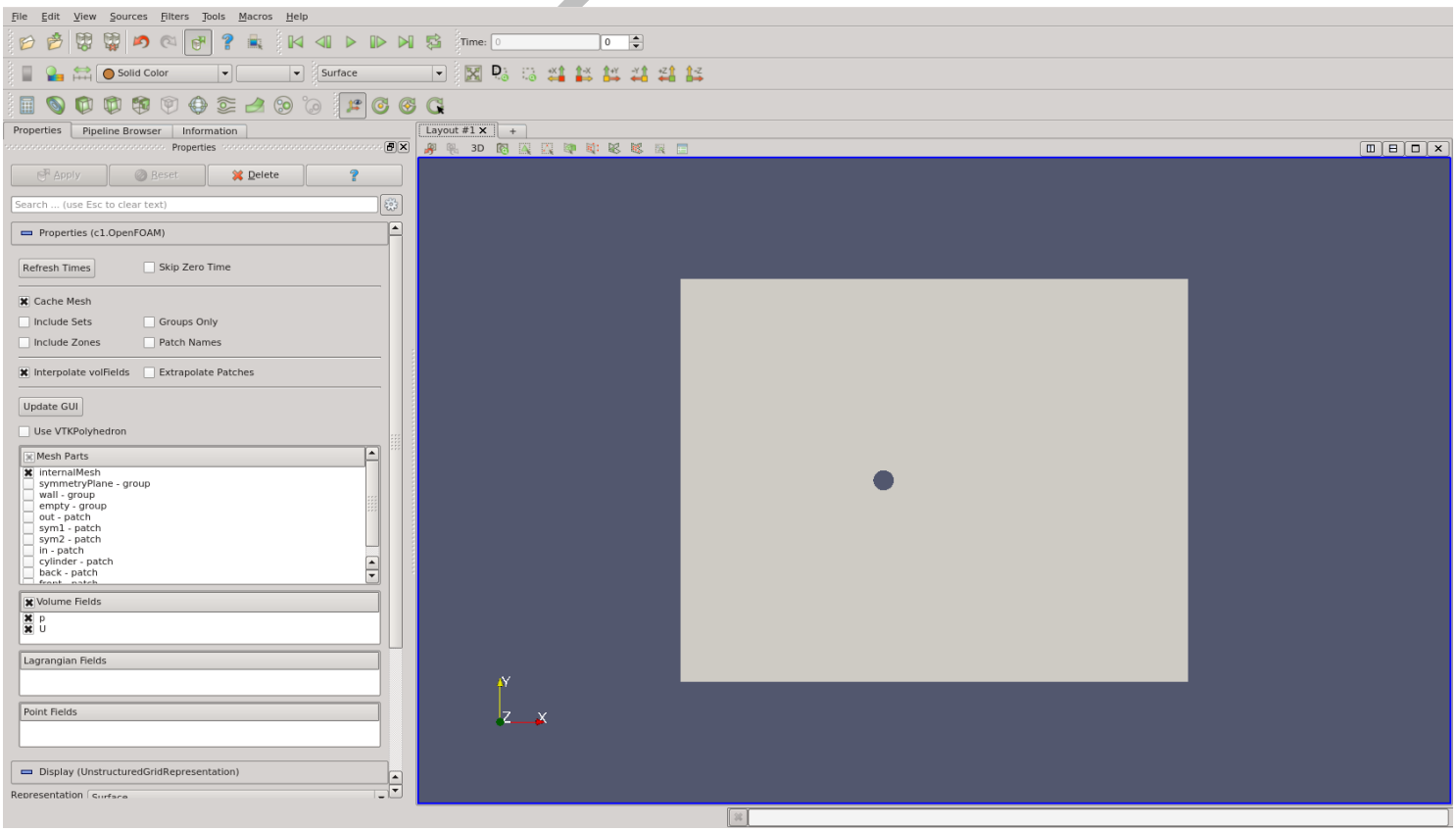

Figure 6. `paraFoam` **window.**

If you do not see anything, make sure that the case `c1.OpenFOAM` is selected in the `Pipeline Browser` tab (highlighted in blue), and that the eye button next to it is switch on, as shown in figure 7.

Before the solution can be viewed, the appropriate fields must be selected in the `Properties` tab.

From the `Volume Fields` menu of the `Properties` tab, we get access to all field variables saved during the simulation, this is illustrated in figure 8. Select those that you want to visualize. By default, only the primitive variables are selected (`p` and `U` in this case).

In the `Properties` tab, you can also select the mesh parts you want to visualize. In the `Mesh Parts` menu the option `internalMesh` is selected by default, as shown in figure 8.

As we are not interested in viewing the solution at time equal to `0` (initial conditions), select the box `Skip Zero Time` in the `Properties` tab, as shown in figure 8. Notice that new field variables will become available in the `Volume Fields` menu of the `Properties` tab.

If you are interested in visualizing the mesh, from the `representation toolbar` select `Wireframe` by clicking the pull-down menu that initially reads `Surface` as illustrated in figure 9. You can also select the option `Surface With Edges`. To return to the surface representation, select `Surface` from the pull-down menu.

From the `active variable toolbar` select the variable you want to visualize by clicking the pull-down menu that initially reads `Solid Color`, as illustrated in figure 10.

Let us select the velocity vector `U`. As this is a vector, we have access to the components and magnitude information, as seen in the pull-down menu shown in figure 11.

To visualize the different time-steps saved, use the `VCR controls` as illustrated in figure 12. The current time is shown in the `current time controls toolbar`, as shown in figure 12.

At this point and if you go to the latest time, that is, `350` seconds, you should get something similar to what is shown in figure 13.

By default, `paraFoam` do not use the color map shown in figure 13. If you want to use this color map, you can select it by using the `Edit Color Map` option available in the `Active variable toolbar`, as shown in figure 14.

By clicking in `Edit Color Map` button will open the `Color Map Editor`, select `Choose Preset` and select the desired color map. From this editor you can also change the color scale.

To enable the color legend visibility, use the legend visibility button in the `Active variable toolbar`, as illustrated in figure 14.

By default only the primitive variables are saved, in this case the pressure field (scalar field `p`) and the velocity field (vector field `U`).

OpenFOAM® comes with many utilities that can be used to compute derived fields. For instance, if you are interested in computing the vorticity field, from within the case directory type in the terminal:

```
vorticity -noZero
```

This will compute the vorticity field except for time `0`.

After computing the vorticity, you can access the new field in `paraFoam` as you usually do. The vorticity field is available in the `Properties` tab, as illustrated in figure 15.

If you want to visualize the newly created field, from the `active variable toolbar` select the vorticity field by clicking the pull-down menu that initially reads `Solid Color`, as illustrated in figure 16. Remember, as this is a vector we have access to the components and magnitude information. In figure 17, we show a plot of the z-component of vorticity.

At this point, try to use any of the filters available in the `Common` and `DataAnalysis` toolbars or in the `Filters` drop-down menu, as shown in figures 18 and 19.

Remember, the visibility of individual regions and filters can be turned on or off using the eye icon next to the labels in the `Pipeline Browser`, as seen in figure 20.

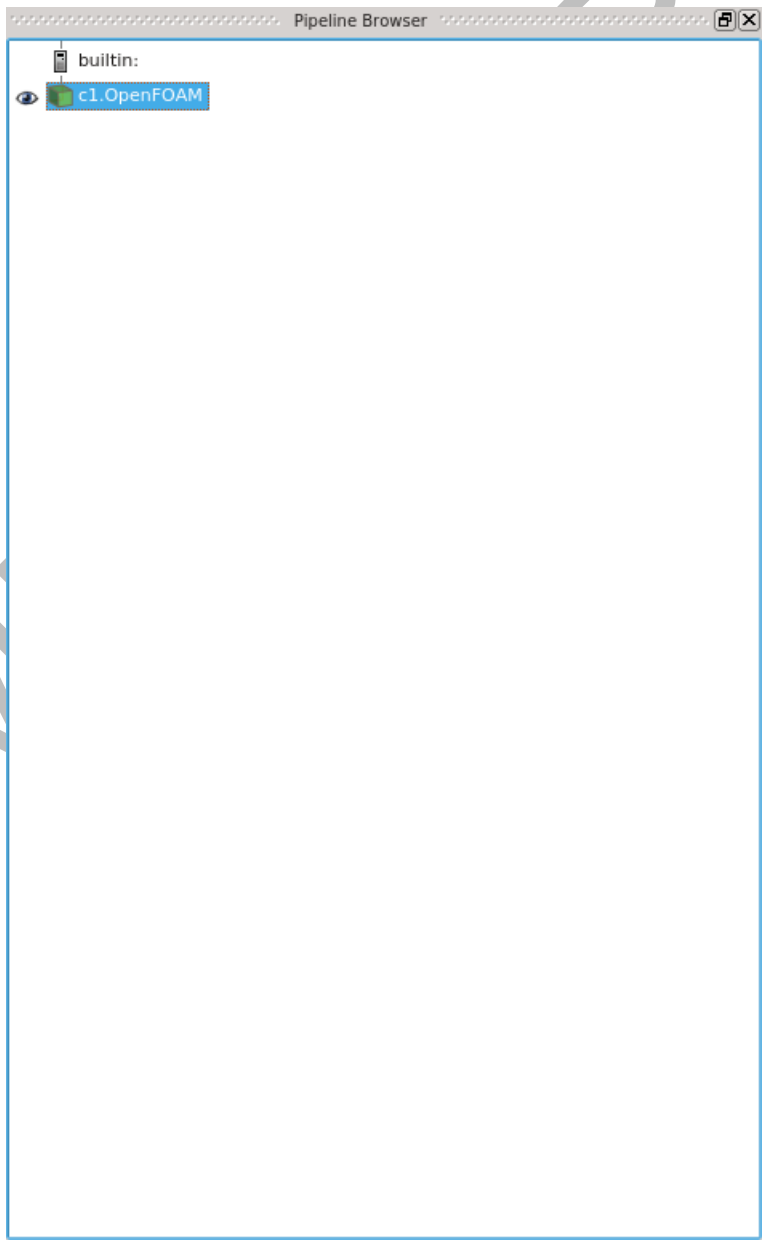

Figure 7. Pipeline browser.

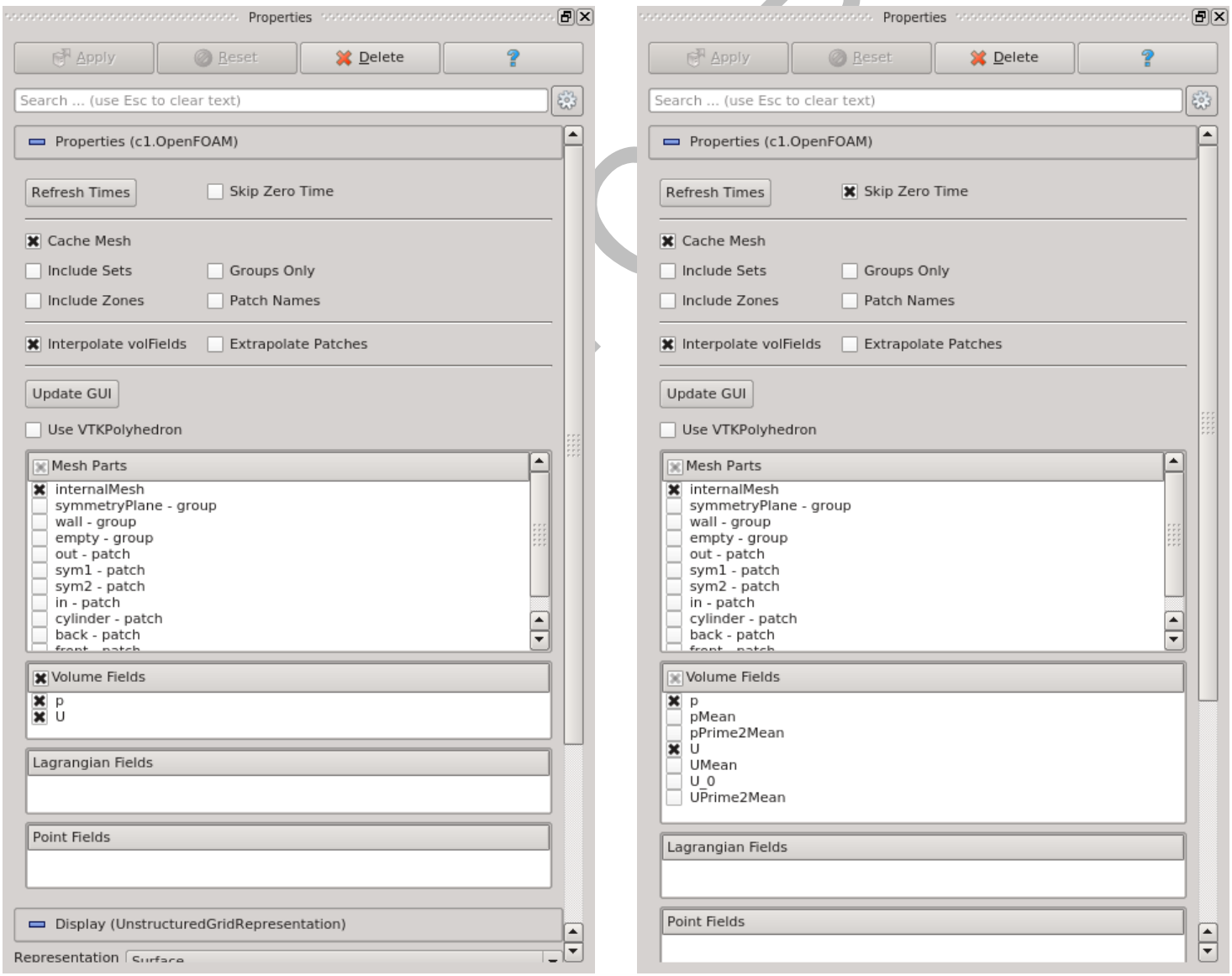

Figure 8. Properties tab.

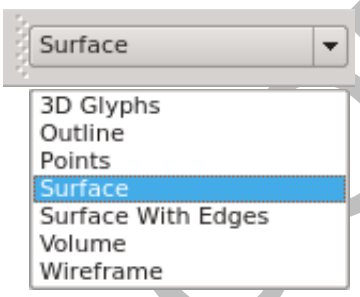

Figure 9. Representation toolbar.

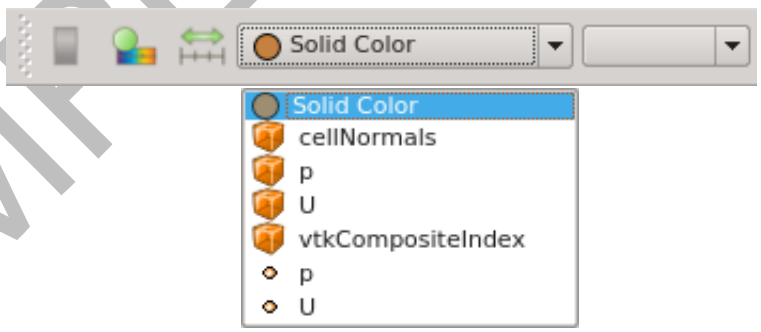

Figure 10. Active variable toolbar.

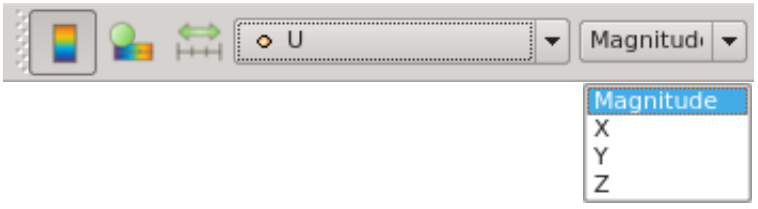

Figure 11. Active variable toolbar.

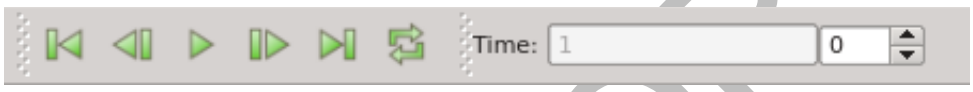

Figure 12. VCR controls and current time controls.

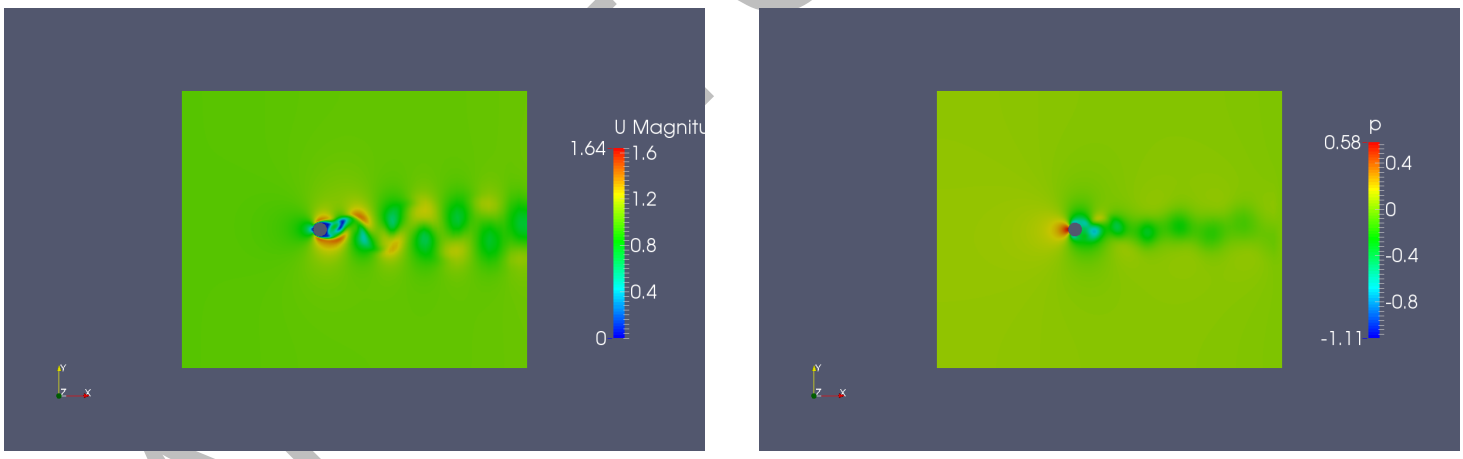

Figure 13. Contours of velocity magnitude and relative pressure.

Figure 14. Toggle color legend visibility (left button) and edit color map (right button) options in the Active variable toolbar.

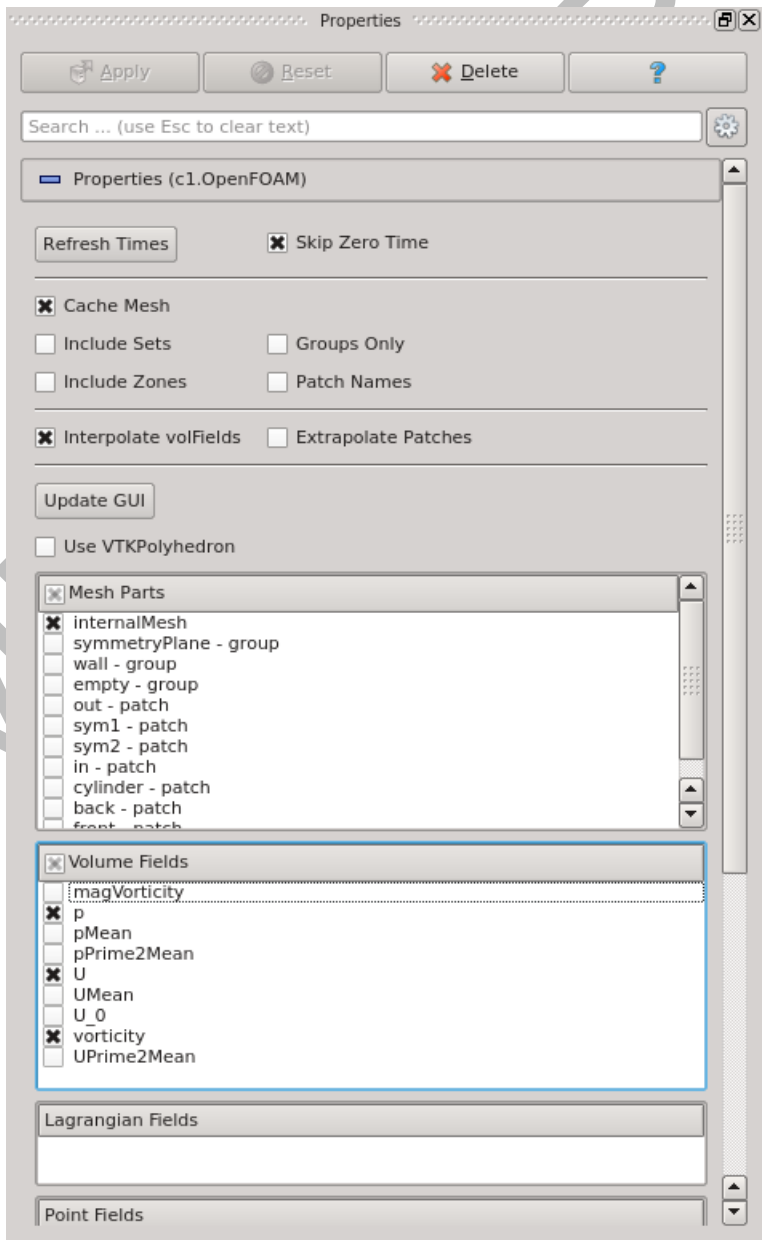

Figure 15. Properties tab.

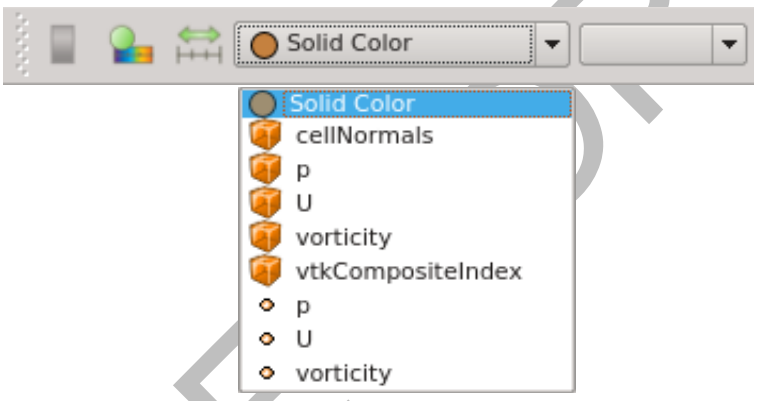

Figure 16. Active variable toolbar.

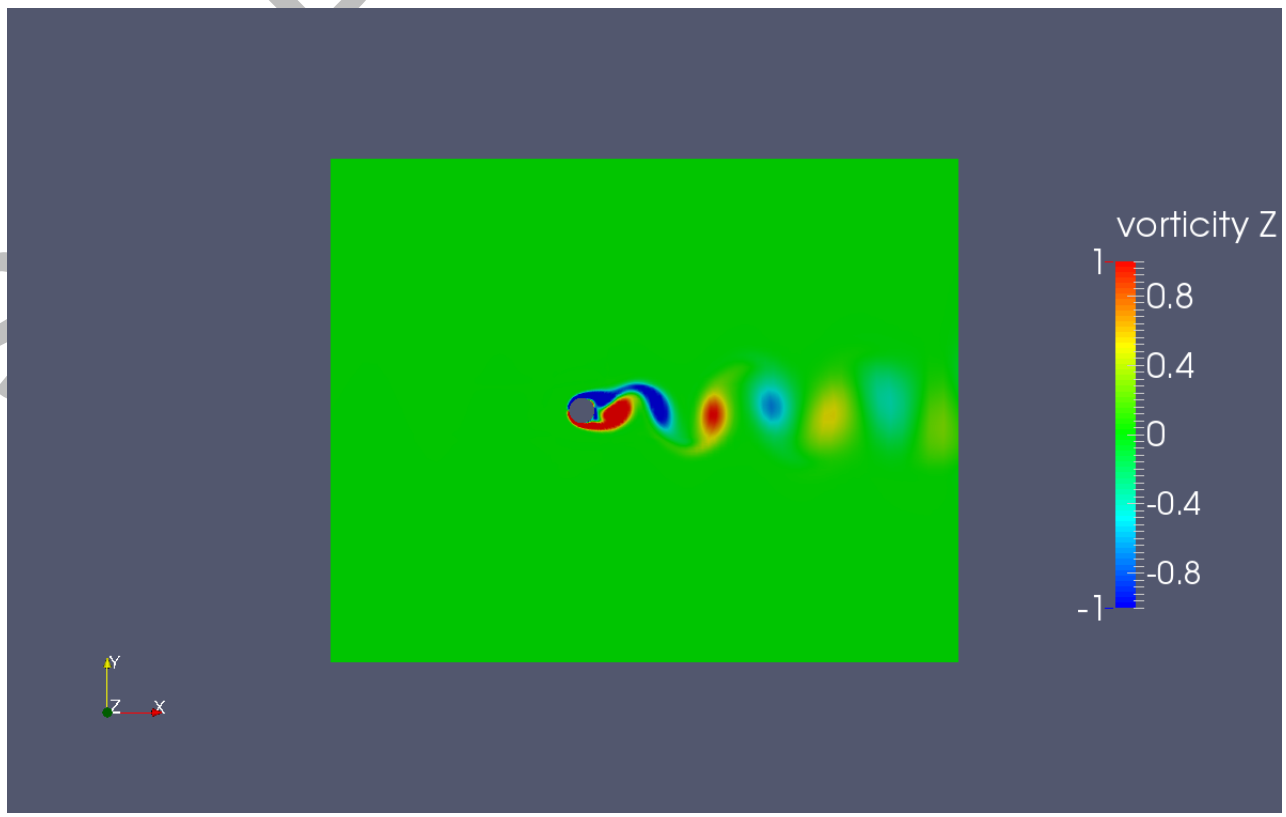

Figure 17. Contours of vorticity (*z*-component).

Figure 18. Common and DataAnalysis toolbars.

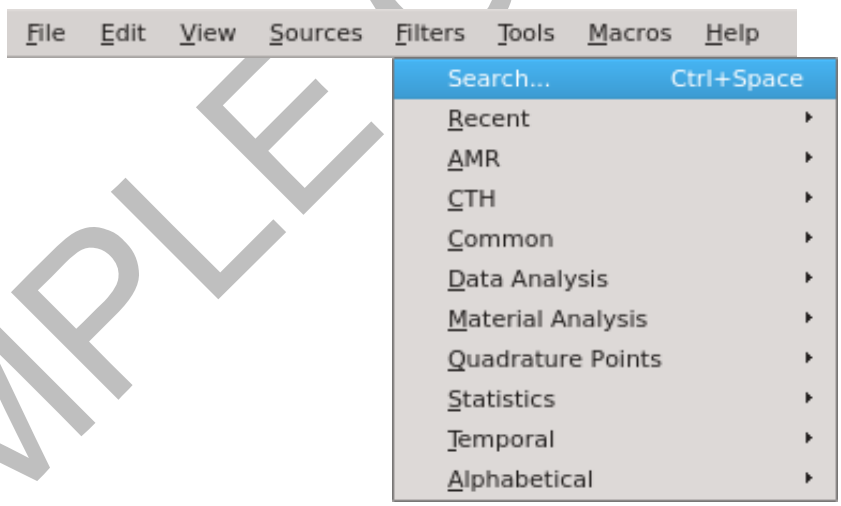

Figure 19. Filters drop-down menu.

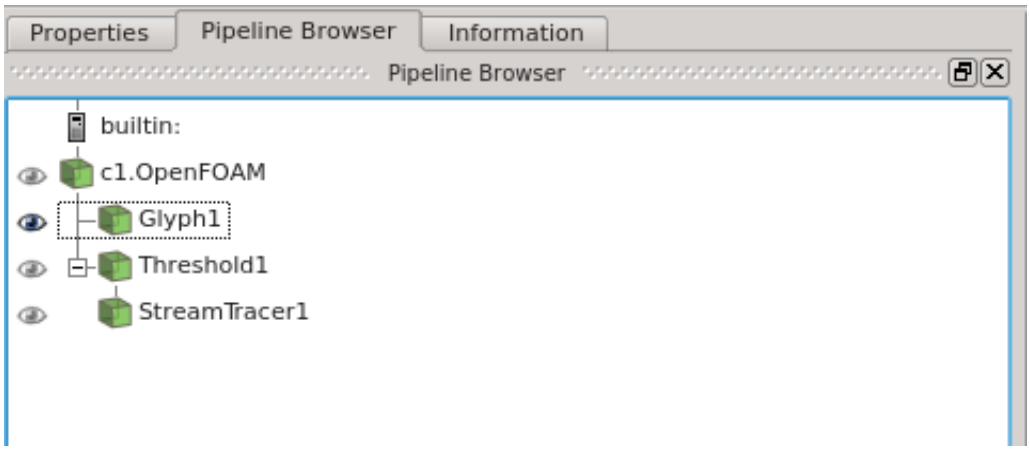

Figure 20. Pipeline browser. Only the object Glyph1 is visible.

In the directory:

```
scripts1/
```

You will find a few `gnuplot` and `Python` scripts that you can use to do post-processing after the simulation is over.

From within the case directory type in the terminal:

```
python scripts1/plot_coe_cl.py
```

This script will save the output in the file `fig_cl.png`. The output plot consists of the lift coefficient in function of time and iteration number, as shown in figure 21.

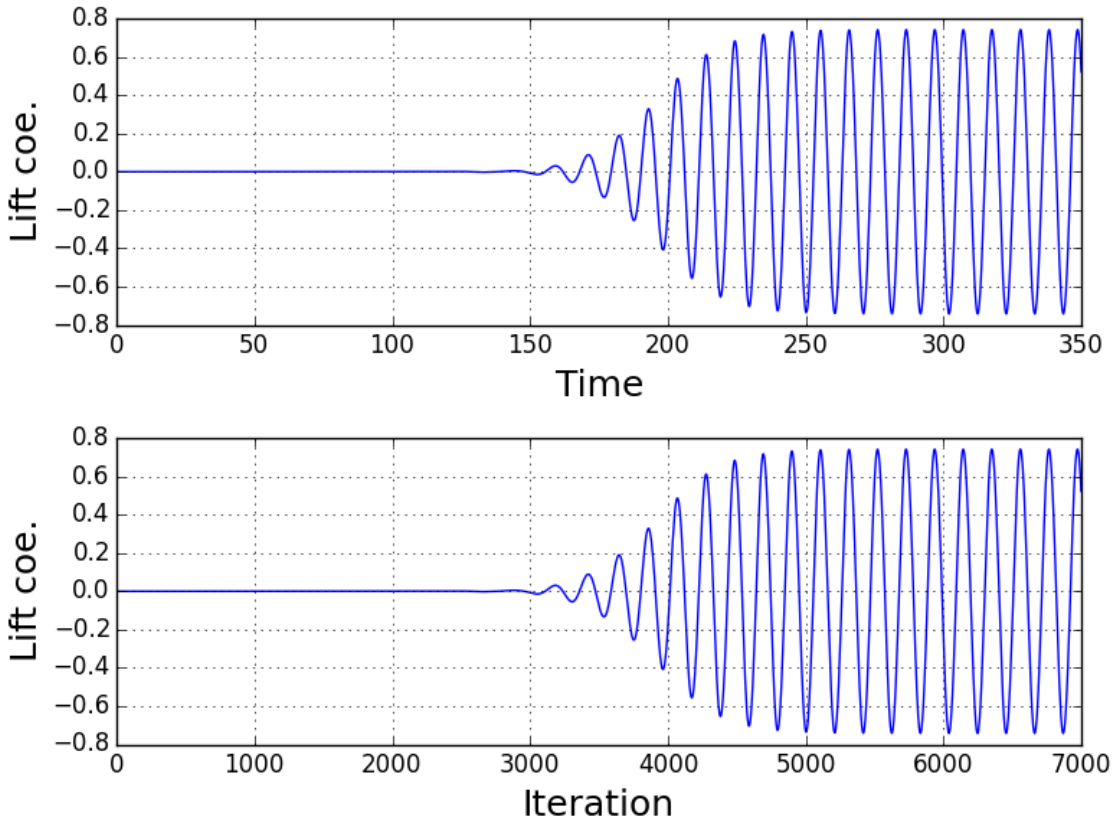

Figure 21. Lift coefficient plot in function of time and iteration number.

To visualize the plot, from within the case directory type in the terminal:

```
gwenview fig_cl.png
```

To compute the vortex shedding frequency using the lift coefficient signal, from within the case directory type in the terminal:

```
python scripts1/psd_cl.py
```

This python script will compute the power spectral density (PSD).

The output of this script is saved in the file `fig_psd_cl.png`. To visualize the PSD plot, from within the case directory type in the terminal:

```
gwenview fig_psd_cl.png
```

The plot of the PSD is shown in figure 22.

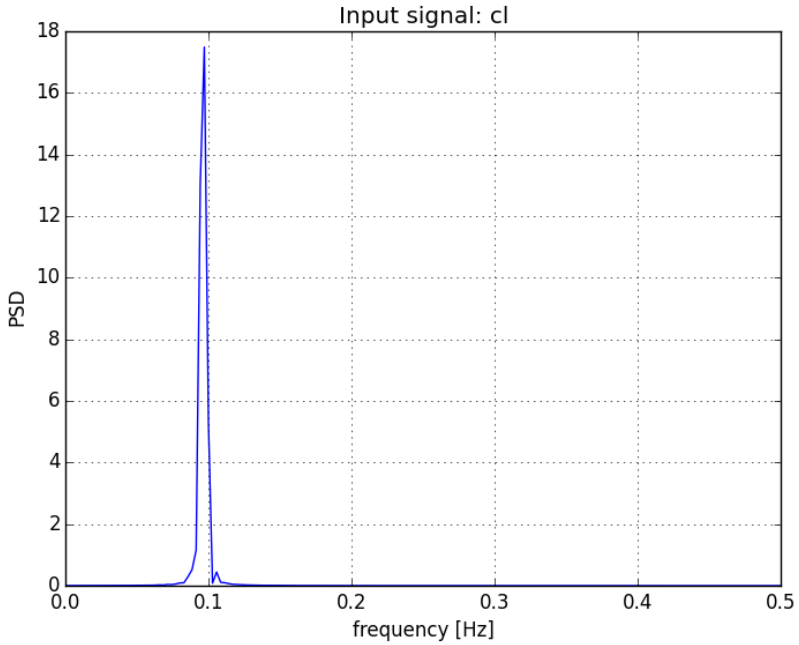

Figure 22. PSD plot of lift coefficient signal.

As it can be seen in figure 22, the dominant frequency is close to 0.1 Hz, this is the vortex shedding frequency.

Using the vortex shedding frequency, we can now compute the Strouhal number as follows,

$$St = \frac{f \times L}{U} \approx 0.2$$

It is worth mentioning that the lift coefficient signal gives the actual vortex shedding frequency. The drag coefficient signal gives twice the frequency obtained using the lift coefficient.

You can compute the PSD of the drag coefficient signal as follows,

```
python scripts1/psd_cd.py
```

These python scripts are an alternative to plotting in `gnuplot`.

The advantage of using python is that we can do advanced plotting, matrix manipulation, statistics and scientific computing with the data.

Feel free to adapt any of these scripts to fit your needs.

You can compute basic statistics of the drag and lift coefficients signal using `gnuplot`.

To do so, from within the case directory type in the terminal:

```
gnuplot scripts1/gstats_cl
```

This script will compute the basic statistics of the lift coefficient signal.

In the terminal window you will get the following output,

```
* FILE:
  Records:       6000
  Out of range:     0
  Invalid:          0
  Blank:            0
  Data Blocks:      1

* COLUMN:
  Mean:          -0.0040
  Std Dev:        0.3742
  Sum:          -24.2511
  Sum Sq.:      840.3415

  Minimum:       -0.7789 [5733]
  Maximum:        0.7790 [5835]
  Quartile:      -0.0888
  Median:        -0.0000
  Quartile:       0.0799
```

In this case, we computed the basic statistics for the last 6000 iterations (records) of the lift coefficient signal. In figure 23, we plot the lift coefficient signal in function of the iteration number.

To compute the basic statistics of the drag coefficient signal, from within the case directory type in the terminal:

```
gnuplot scripts1/gstats_cd
```

In the terminal window you will get the following output,

```
* FILE:
  Records:       2000
  Out of range:     0
  Invalid:          0
  Blank:            0
  Data Blocks:      1

* COLUMN:
  Mean:           1.4282
  Std Dev:        0.0386
  Sum:         2856.3136
  Sum Sq.:     4082.2455

  Minimum:        1.3712 [  58]
  Maximum:        1.4820 [1435]
  Quartile:       1.3896
  Median:         1.4293
  Quartile:       1.4669
```

In this case, we computed the basic statistics of the last 2000 iterations (records). We used the last 2000 records, because after 5000 iterations we get a periodic behavior of the signal, as shown in figure 23.

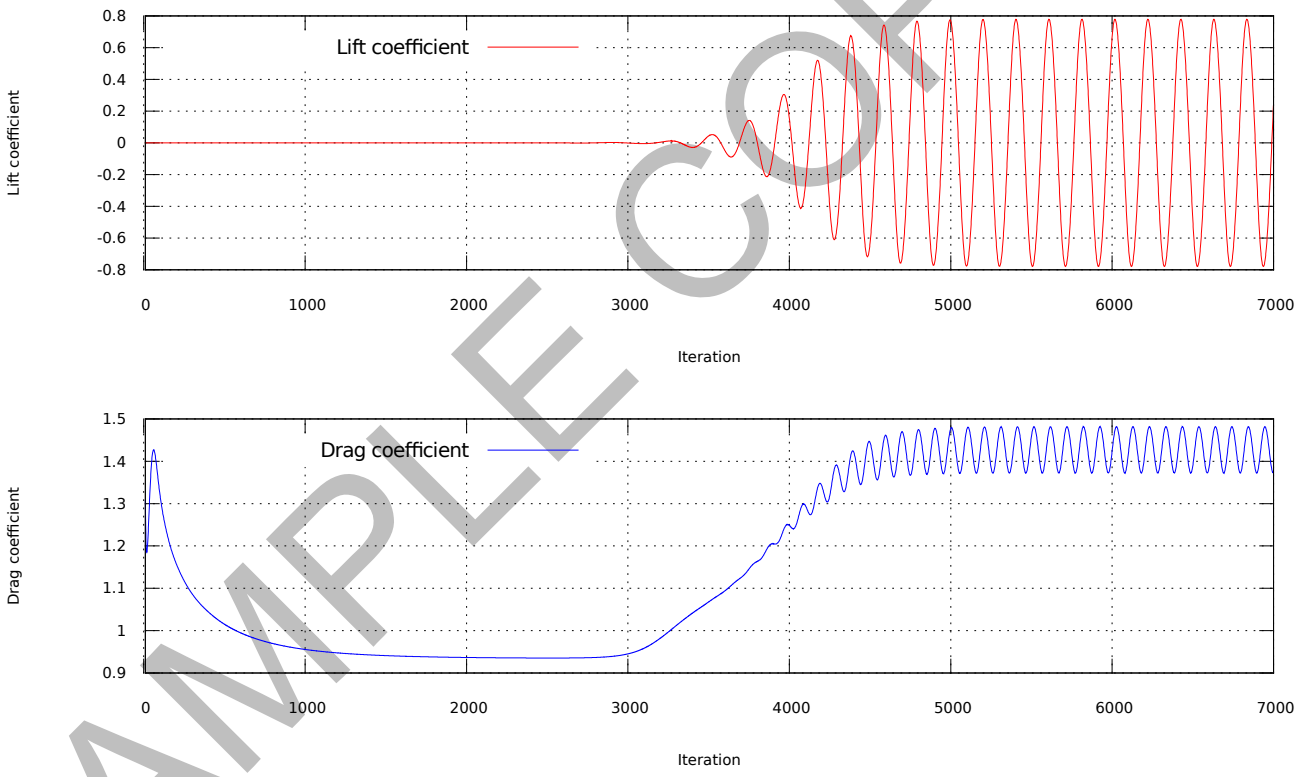

Figure 23. Lift coefficient and drag coefficient plot in function of iteration number (record entry).

Feel free to adapt any of these scripts to fit your needs.

13. Summary

To run this tutorial, go to the case directory:

```
cd $PTOFC/first_tutorial/vortex_shedding/c1/
```

To run in serial, within the case directory type in the terminal:

```
blockMesh
checkMesh
icoFoam > log 2>&1 | tail –f log
paraFoam
```

To run in parallel and assuming that you are using 4 processors, within the case directory type in the terminal:

```
blockMesh
checkMesh
decomposePar
mpirun –np 4 icoFoam –parallel > log 2>&1 | tail –f log
reconstructPar
paraFoam
```

- The utility `blockMesh` generates the mesh. This utility does not run in parallel.

- The utility `checkMesh` reports mesh statistics, checks for topological errors and reports the mesh quality. To run this utility in parallel, from within the case directory type in the terminal:

```
mpirun –np 4 checkMesh –parallel
```

- In this tutorial, we use the following solver:

```
Application
    icoFoam

Description
     Transient solver for incompressible, laminar flow of Newtonian
     fluids.
```

- The utility `paraFoam` is used for post-processing and visualization.

- The utility `decomposePar` decompose the domain for running in parallel. It reads the information contained in the dictionary `decomposeParDict` located in the directory `system`. This utility does not run in parallel.

- The utility `reconstructPar` reconstruct the solution after running in parallel. This utility does not run in parallel.

- To run in parallel we use the MPI library. To use a solver or utility in parallel, you can proceed as follows:

```
mpirun –np <PROCS> <utility/solver> –parallel
```

Where `<PROCS>` is the number of processors to use, this number needs to be the same as the number of domain partitions defined in the dictionary `decomposeParDict`.

The keyword `<utility/solver>` is the name of the utility or solver to use.

When running in parallel, do not forget to always add the flag `–parallel`.

14. Verification & Validation

Some numerical results of the flow past a circular cylinder at various Re = 200.

Reference		$\bar{c}_d$ – Re = 200	$\bar{c}_l$ – Re = 200
[3]	Russell and Wang	1.29 ± 0.022	± 0.050
[4]	Calhoun and Wang	1.17 ± 0.058	± 0.067
[5]	Braza *et al*.	1.40 ± 0.05	± 0.075
[6]	Choi *et al*.	1.36 ± 0.048	± 0.064
[7]	Liu *et al*.	1.31 ± 0.049	± 0.069
[8]	Guerrero	1.40 ± 0.048	± 0.072

15. Exercises and further improvements

- In this tutorial, we used `gnuplot` to compute the basic statistics of the drag and lift coefficients. Try to write a `Python` script to compute the basic statistics.
- Compare the solution obtained using a first order method (upwind) and a second order method (linear and second order upwind).
- Compare your solution with the results reported in section 14. Can you quantify the error?

16. References

[1] OpenFOAM® user guide. Version 2.3.0.

[2] http://www.openfoam.org/docs/cpp/

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

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

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

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

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

[8] J. Guerrero. *Numerical Simulation of the unsteady aerodynamics of flapping flight*. PhD Thesis, University of Genova, 2009.