Supplement

More on sampling

Sampling on point clouds, lines and surfaces

Supplement – More on sampling

- OpenFOAM® provides the `postProcess` utility to sample field data for plotting.
- The sampling locations are specified in the dictionary *`sampleDict`* located in the case **`system`** directory.
- You can give any name to the input dictionary, hereafter we are going to name it *`sampleDict`*.
- During the sampling, inside the case directory, a new directory named **`postProcessing`** will be created. In this directory, the sampled values are stored.
- This utility can sample points, lines, and surfaces.
- Data can be written in many formats, including well-known plotting packages such as: grace/xmgr, gnuplot and jPlot.
- The sampling can be executed by running the utility `postProcess` in the case directory and according to the application syntax.
- A final word, this utility does not do the sampling while the solver is running. It does the sampling after you finish the simulation.

Supplement – More on sampling

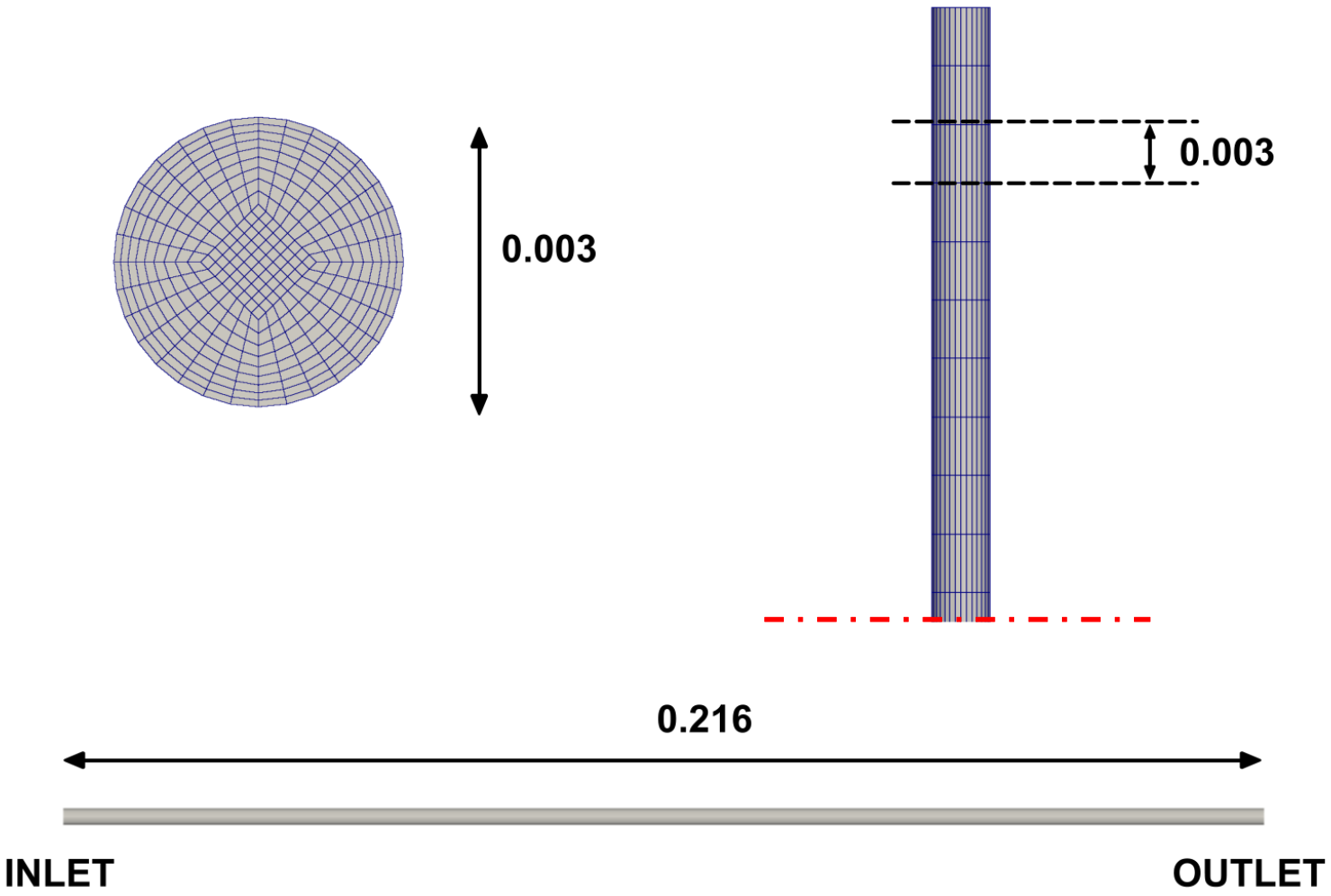
- Let us do some sampling.
- For this we will use the 3D pipe case, which you will find in the directory:

`$PTOFC/advanced_postprocessing/pipe/`

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Supplement – More on sampling

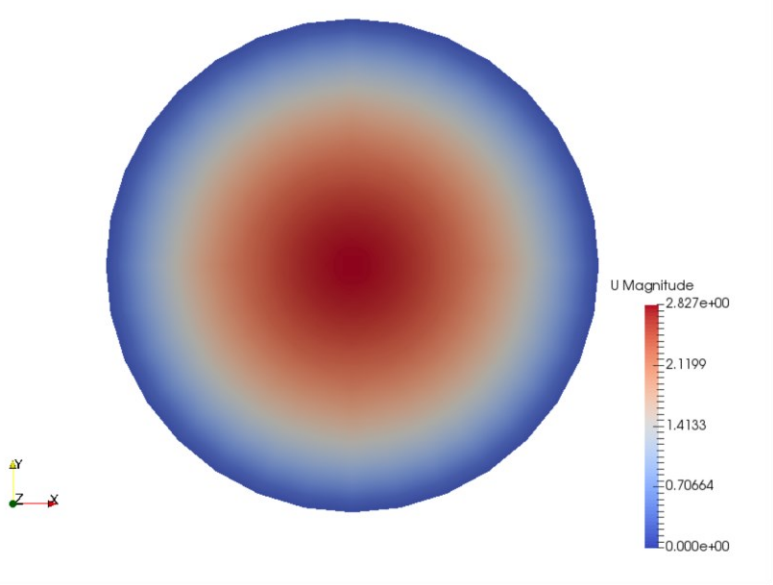
Laminar flow in a straight pipe – Re = 600



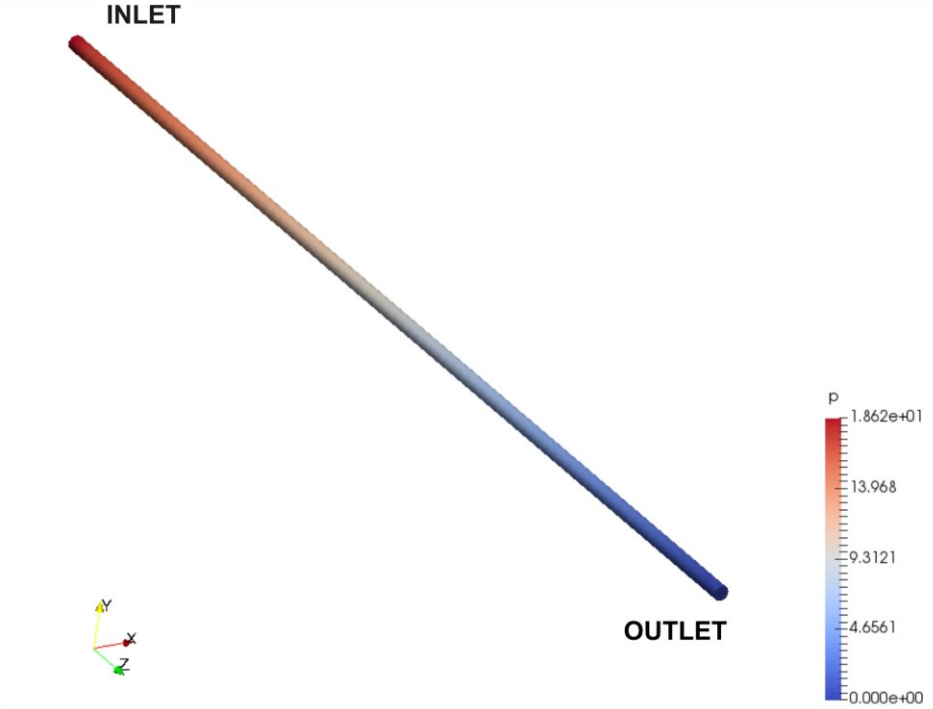
Mesh and domain

Supplement – More on sampling

Laminar flow in a straight pipe – Re = 600



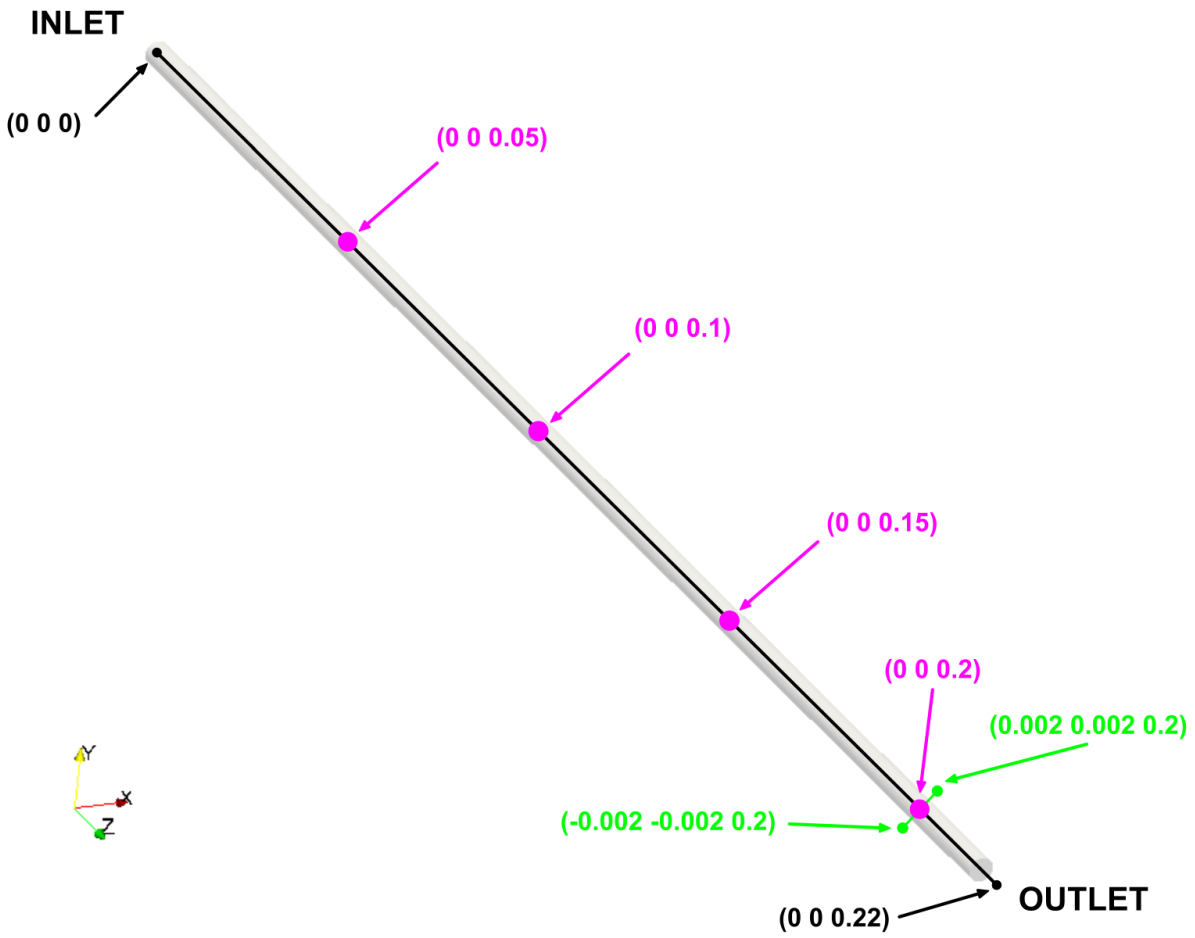
Velocity magnitude at the outlet



Pressure contours at the wall

Supplement – More on sampling

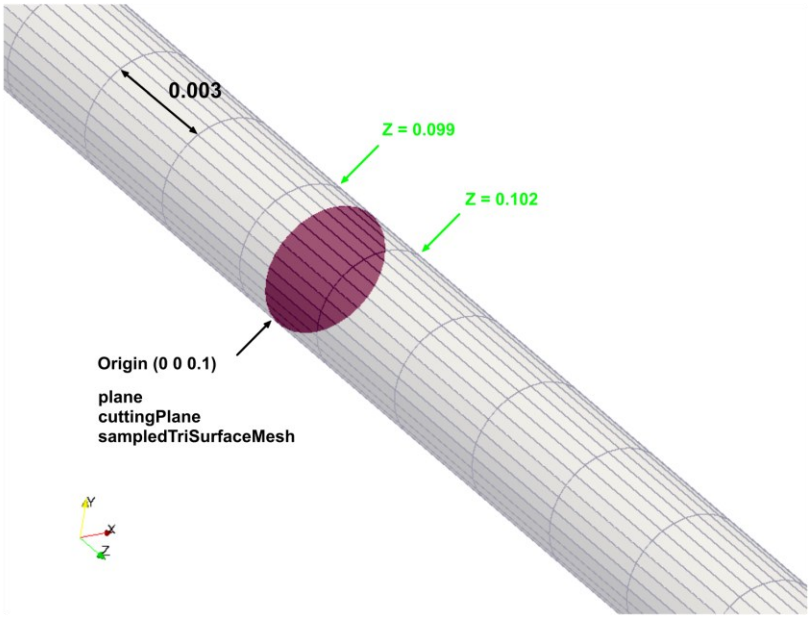
Laminar flow in a straight pipe – Re = 600



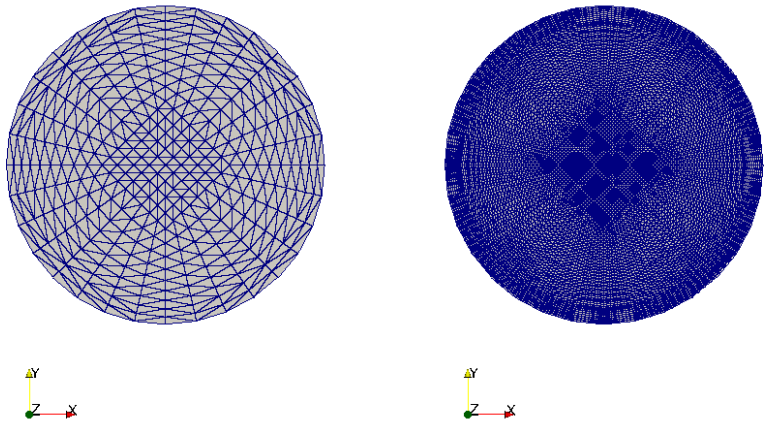
Point and lines where we want to sample

Supplement – More on sampling

Laminar flow in a straight pipe – Re = 600



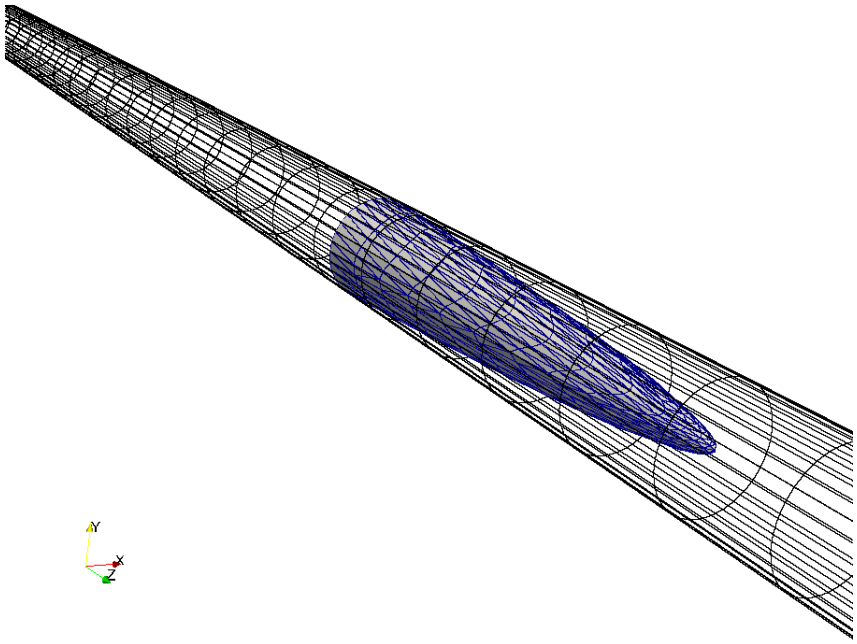
Surface type and location

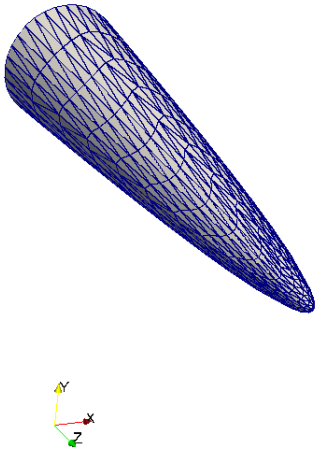


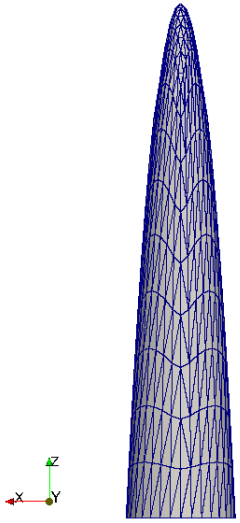
Coarse and fine surfaces

Supplement – More on sampling

Laminar flow in a straight pipe – Re = 600







We can also sample in arbitrary surface

Supplement – More on sampling

What are we going to do?

- We will simulate a laminar flow in a straight pipe (Re = 600).
- We will use this case to introduce the sampling utility `postProcess`.
- We will introduce the utility `topoSet` used to do topological modifications on the mesh.
- We will use this utility to run **functionObjects** a-posteriori.
- We will compare the numerical solution with the analytical solution.
- To find the numerical solution we will use the solver `pisoFoam`.
- After finding the numerical solution we will do some sampling.
- At the end, we will do some plotting (using gnuplot or Python) and scientific visualization.

Supplement – More on sampling

Running the case

- Let us run the simulation and do some sampling. In the terminal type:

```
1. $> foamCleanTutorials
2. $> blockMesh
3. $> checkMesh
4. $> topoSet
5. $> pisoFoam | tee log.solver
6. $> postProcess -func sampleDict1 -latestTime
7. $> postProcess -func sampleDict2 -latestTime
8. $> gnuplot gnuplot/gnuplot_script
9. $> paraFoam
```

- Do not erase the solution, we are going to use it in the next sections.

Supplement – More on sampling

Running the case

- In step 4 we use the utility `topoSet` to do mesh topological manipulation. This utility will read the dictionary *`topoSetDict`* located in the **`system`** directory. Later on, we will talk about what are we doing in this step.
- In step 5 we run the simulation and save the log file.
- In step 6 we use the `postProcess` utility. By using the option `-func` we specify to do the sampling according to the dictionary *`system/sampleDict1`*. We sample the latest saved solution.
- In step 7 we use the `postProcess` utility. By using the option `-func` we specify to do the sampling according to the read dictionary *`system/sampleDict2`*. We sample the latest saved solution.
- In step 8 we use the gnuplot script *`gnuplot/gnuplot_script`* to plot the sampled fields. Feel free to take a look at the script and to reuse it.
- Finally, in step 9 we visualize the solution.

Supplement – More on sampling

The *sampleDict* dictionary

- Let us visit the *sampleDict* dictionaries.
- This dictionary is located in the directory **system**.
- The *sampleDict* file contains several entries to be set according to the user needs.
- You can set the following entries,
    - The choice of the interpolationScheme.
    - The format of the line data output.
    - The format of the surface data output.
    - The fields to be sample.
    - The sub-dictionaries that controls each sampling operation.
    - In these sub-dictionaries you can set the name, type and geometrical information of the sampling operation.
- In this case, in the dictionary *sampleDict1* we are sampling points and lines, and in the dictionary *sampleDict2* we are sampling surfaces.

Supplement – More on sampling

The *sampleDict1* dictionary

```
17    type sets;

19    setFormat raw;

21    interpolationScheme cellPointFace;

24    fields
25    (
26        U
27        p
28    );

30    sets
31    (
33        s1
34        {
35            type            lineCellFace;
40            axis            z;
41            start           (0 0 0);
42            end             (0 0 0.22);
43        }
44
45        s2
46        {
47            type            lineCellFace;
49            axis            x;
50            start           (-0.002  -0.002 0.2);
51            end             ( 0.002   0.002 0.2);
52        }
53
```

Sample sets (points and lines).

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

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

Fields to sample.

Name of the output file

Sample method from the solution to the line.

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

Name of the output file

Sample method from the solution to the line.

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

Note:
Use the banana method to know all the options available.

Supplement – More on sampling

The *sampleDict1* dictionary

```
54
55      somePoints

56      {
57          type     points;
58          ordered  true;
59          axis     xyz;

61          points
62          (
63              (0 0 0.05)
64              (0 0 0.1)
65              (0 0 0.15)
66              (0 0 0.2)
67          );
68      }

71  );
```

Name of the output file

Sample a cloud of points

Location of the sample points.

The sampled information is always saved in the directory

postProcessing/sampleDict1

As we are sampling the latest solution (0.1), the sampled data will be located in the directory:

postProcessing/sampleDict1/0.1

The files *s1_p.xy*, *s2_p.xy*, *s1_U.xy*, *s2_U.xy*, *somePoints_p.xy*, and *somePoints_U.xy* located in the directory **postProcessing/sampleDict1/0.1** contain the sampled data. Feel free to open the output files using your favorite text editor.

Supplement – More on sampling

The *sampleDict2* dictionary

```
17  type surfaces;

19  surfaceFormat raw;

21  interpolationScheme cell;
22
26  fields
27  (
28      U
29      p
30  );
31
32  surfaces
33  (
34      surf1
35      {
36          type            plane;
37          planeType       pointAndNormal;
38          pointAndNormalDict
39          {
40              basePoint       (0 0 0.1);
41              normalVector    (0 0 1);
42          }
43      }
44
45      surf2
46      {
47          type            cuttingPlane;
48          planeType       pointAndNormal;
49          pointAndNormalDict
50          {
51              basePoint       (0 0 0.1);
52              normalVector    (0 0 1);
53          }
54
55          interpolate     true;
56      }
```

Sample surfaces.

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

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

Fields to sample.

Name of the object and output file

Surface sampling method. In this case we are using an infinite plane.

Name of the object and output file

Surface sampling method. In this case we are using a cutting plane. The interpolate option means that we interpolate the cell centered values to the surface triangulation.

Supplement – More on sampling

The *sampleDict2* dictionary

```
57
58      surf3
59      {
60          type            triSurfaceMesh;
61          surface         surface2.stl;
62          source          cells;
63
64          interpolate true;
65      }
66
67      surf4
68      {
69          type            triSurfaceMesh;
70          surface         surface3.stl;
71          source          insideCells;
72
73          interpolate false;
74      }
75  );
```

Name of the object and output file

Surface sampling method. In this case we are using a STL file, the file is always located in the directory **`constant/triSurface`**.

Name of the object and output file

Surface sampling method. In this case we are using a STL file, the file is always located in the directory **`constant/triSurface`**.

The sampled information is always saved in the directory

`postProcessing/sampleDict2`

As we are sampling the latest solution (0.1), the sampled data will be located in the directory:

`postProcessing/sampleDict2/0.1`

The files *p_surf1.raw*, *p_surf2.raw*, *p_surf3.raw*, *p_surf4.raw*, *U_surf1.raw*, *U_surf2.raw*, *U_surf3.raw*, and *U_surf4.raw*, located in the directory **`postProcessing/sampleDict2/0.1`** contain the sampled data. Feel free to open the output files using your favorite text editor.

Supplement – More on sampling

The output files

- The output format of the point sampling (cloud) is as follows:

Scalars

```
#POINT_COORDINATES (X Y Z)          SCALAR_VALUE
0      0      0.05                  13.310995
0      0      0.1                   19.293817
…
```

Vectors

```
#POINT_COORDINATES (X Y Z)          VECTOR_COMPONENTS (X Y Z)
0      0      0.05                  0      0      2.807395
0      0      0.1                   0      0      2.826176
…
```

Supplement – More on sampling

The output files

- The output format of the line sampling is as follows:

Scalars

```
#AXIS_COORDINATE        SCALAR_VALUE
0                       18.594038
0.0015                  18.249091
…
```

Vectors

```
#AXIS_COORDINATE        VECTOR_COMPONENTS (X Y Z)
0                       0      0      1.6152966
0.0015                  0      0      1.8067536
…
```

Supplement – More on sampling

The output files

- The output format of the surface sampling is as follows:

Scalars

```
#POINT_COORDINATES (X Y Z)          SCALAR_VALUE
0      0      0.05                  13.310995
0      0      0.1                   19.293817
…
```

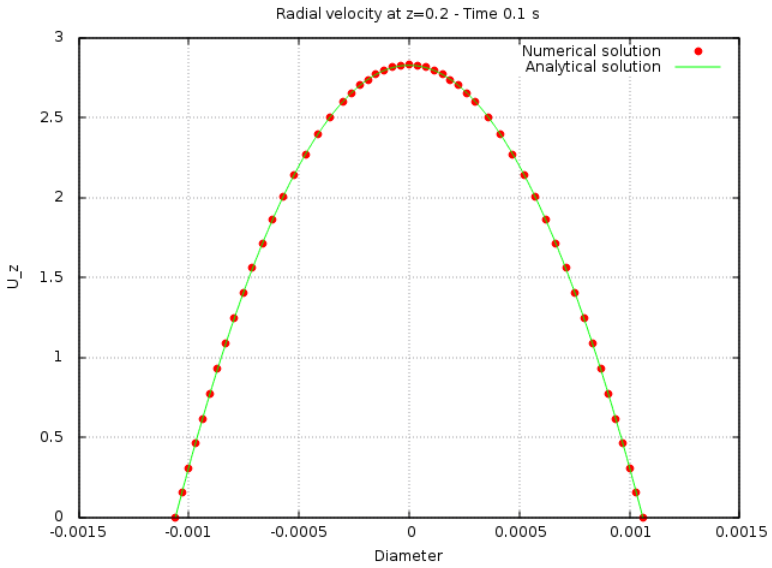
Vectors

```
#POINT_COORDINATES (X Y Z)          VECTOR_COMPONENTS (X Y Z)
0      0      0.05                  0      0      2.807395
0      0      0.1                   0      0      2.826176
…
```

Supplement – More on sampling

- To plot the sampled data using gnuplot we can proceed as follows:

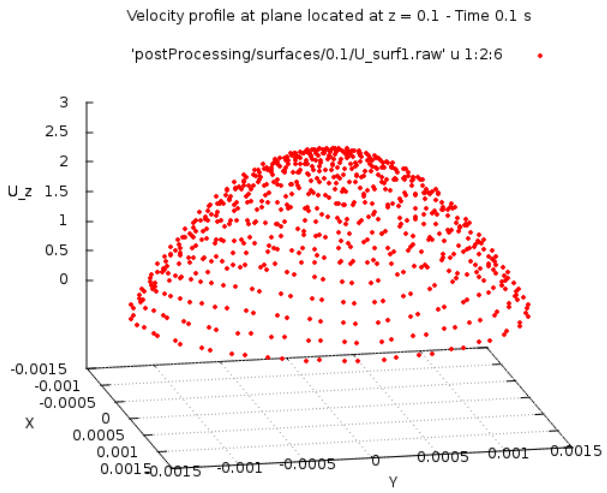
1. `gnuplot> set title 'Radial velocity at z=0.2 - Time 0.1 s'`
2. `gnuplot> set xlabel 'Diameter'`
3. `gnuplot> set ylabel 'U_z'`
4. `gnuplot> set grid`
5. `gnuplot> plot [][] 'postProcessing/sampleDict1/0.1/s2_U.xy' u 1:4 w p pt 7 title "Numerical solution", 2.8265544*(1-x**2/(0.0010606602 )**2) title "Analytical solution"`



Supplement – More on sampling

- To plot the sampled data using gnuplot we can proceed as follows:

```
1. gnuplot> set title 'Velocity profile at plane located at z = 0.1 - Time 0.1 s'
2. gnuplot> set xlabel 'X'
3. gnuplot> set ylabel 'Y'
4. gnuplot> set zlabel 'U_z'
5. gnuplot> set grid
6. gnuplot> splot [][][] 'postProcessing/sampleDict/0.1/U_surf1.raw' u 1:2:6 pt 7 ps 0.5
```



Supplement – More on sampling

Creating **faceSet** and **zoneSet**

- To create **sets** and **zones** we use the utility `topoSet`.
- This utility reads the dictionary *`topoSetDict`* located in the **`system`** directory.
- **faceSet/cellSet** and **faceZone/cellZone** can be used to do modifications to the mesh or to apply source terms.
- We can only do sampling operations on **zoneSets** made of a set of faces and/or cells, therefore, if we have a **faceSet/cellSet** we need to convert it to a **faceZone/cellZone**.
- Creating an internal **faceZone** is particularly important if we are interested in computing the mass flow in an internal surface, as **sampledSurface** does not work with **surfaceScalarFields;** therefore, we need to use a **faceZone**.
- Alternatively, we can compute the mass flow in paraFoam/paraView or we can use the **areaNormalIntegrate** operation on the **sampleSurface**.
- Let us create an internal **faceZone** and a **cellZone** and let us compute the mass flow and do some sampling on these sets.

Supplement – More on sampling

The *topoSetDict* dictionary

```
17  actions
18  (
19
20      {
21          action  new;
22          name    internalfaces;
23          type    faceSet;
24
25          source  boxToFace;
26          sourceInfo
27          {
28              box (-1 -1 0.098) (1 1 0.1);
29          }
30      }
31
32      {
33          action  new;
34          name    internalfacepatch;
35          type    faceZoneSet;
36
37          source  setToFaceZone;
38          sourceInfo
39          {
40              faceSet internalfaces;
41          }
42      }
43
```

- In this step we are creating a **new faceSet** from a **boxToFace** source.
- The name of the new **faceSet** is **internalFaces**.
- The **box** source encloses the faces we want to tag.
- We can visualize this set in paraFoam.
- Remember, we can not sample on a **faceSet**, we need to convert it to a **faceZone**.

- In this step we convert the **faceSet internalFaces**, to a **faceZone**.
- The name of new **faceZone** is **internalfacepatch**.
- At this point, we can use this **faceZone** to do sampling.
- We can visualize this zone in paraFoam.

Note:
Use the banana method to know all the options available.

Supplement – More on sampling

The *topoSetDict* dictionary

```
44        {
45            action  new;
46            name    internalcells;
47            type    cellSet;
48
49            source  boxToCell;
50            sourceInfo
51            {
52                box (-1 -1 0.096) (1 1 0.1);
53            }
54        }
55
56        {
57            action  new;
58            name    internalcells;
59            type    cellZoneSet;
60
61            source  setToCellZone;
62            sourceInfo
63            {
64                set internalcells;
65            }
66        }
67
68    );
```

- In this step we are creating a **new cellSet** from a **boxToCell** source.
- The name of the new **cellSet** is **internalCells**.
- The **box** source encloses the cells we want to tag.
- We can visualize this set in paraFoam.
- Remember, we can not sample on a **cellSet**, we need to convert it to a **cellZone**.

- In this step we convert the **cellSet internalcells**, to a **cellZone**.
- The name of new **cellZone** is **internalcells**.
- At this point, we can use this **cellZone** to do sampling.
- We can visualize this set in paraFoam.

Note:
Use the banana method to know all the options available.

Supplement – More on sampling

- Let us create an internal **faceZone** and a **cellZone** and let us compute the mass flow and do some sampling on these sets.

1. `$> topoSet`
2. `$> pisoFoam -postProcess -dict system/functionobject3 -latestTime`
3. `$> pisoFoam -postProcess -dict system/functionobject4 -latestTime`

- In step 1 we use the utility `topoSet` to create the new **faceZone** and **cellZone**.
- In step 2 we run a **functionObject** a-posteriori. In this **functionObject** we compute:
  - Mass flow in a **sampledSurface**.
  - Mass flow in a **faceZone**.
- In step 3 we run a **functionObject** a-posteriori. In this **functionObject** we compute:
  - Volume integral in a **cellZone**.

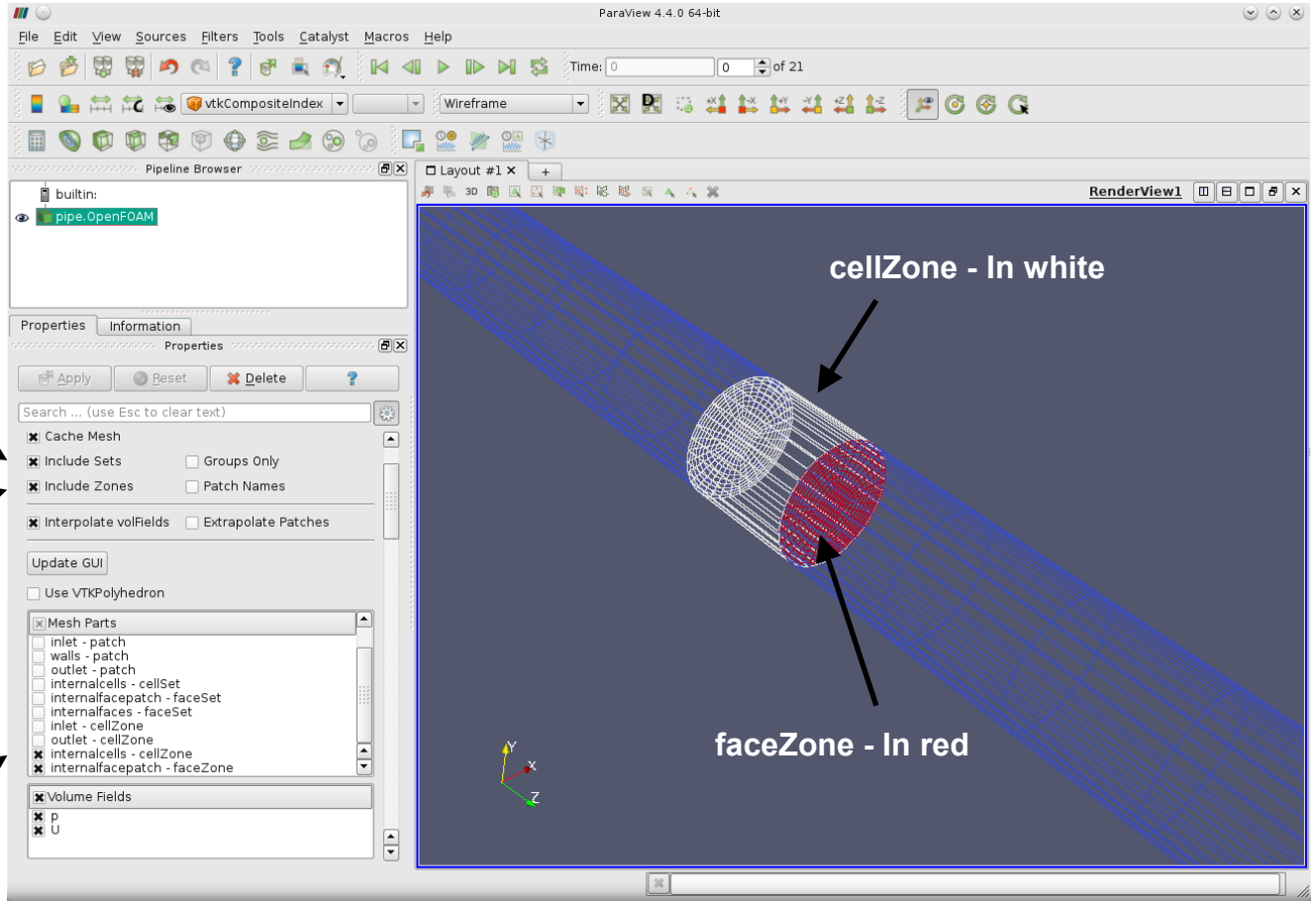
Supplement – More on sampling

Visualizing the newly created sets and zones

Check the option `Include Sets` to visualize the sets

Check the option `Include Zones` to visualize the zones

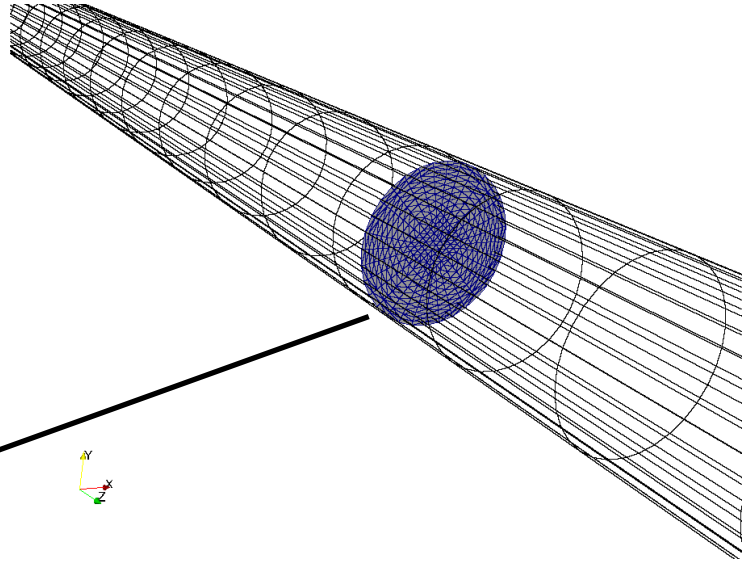
Select the sets you want to visualize (faceSet, faceZone, cellSet or cellZone)



Supplement – More on sampling

The `functionobject3` dictionary

```
17      functions
18      {
19
21          surface1_massflow
22          {
23              type            surfaceRegion;
24              functionObjectLibs ("libfieldFunctionObjects.so");
25              enabled         true;
26
27              writeControl    timeStep;
28              writeInterval   1;
29
30              log             true;
31              writeFields     false;
32
33              regionType      sampledSurface;
34              Name            dummy;
35
36              sampledSurfaceDict
37              {
38                      type sampledTriSurfaceMesh;
39                      surface surface1.stl;
40                      source cells;
41                      interpolate false;
42              }
43
54              operation       areaNormalIntegrate;
55              fields
56              (
57                  U
58              );
59
60          }
```

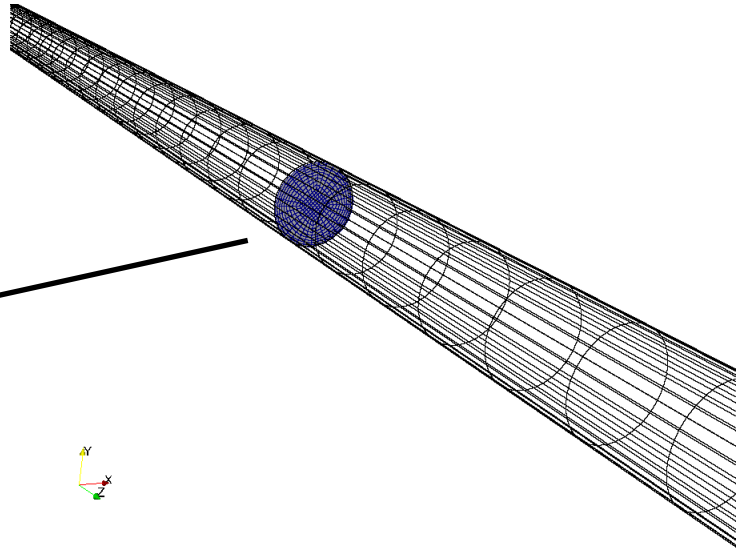


- Compute mass flow using **areaNormalIntegrate** operation with the field **U**.
- Using the operation **sum** with the field **phi**, will not work because **phi** is a **surfaceScalarField**.

Supplement – More on sampling

The `functionobject3` dictionary

```
64          surface2_massflow
65          {
66              type            surfaceRegion;
67              functionObjectLibs ("libfieldFunctionObjects.so");
68              enabled         true;
69
70              writeControl    timeStep;
71              writeInterval   1;
72
73              log             true;
74              writeFields     false;
75
76              regionType      faceZone;
77              name            internalfacepatch;
78
79              operation       sum;
80              fields
81              (
82                  phi
83              );
84
85          }
```

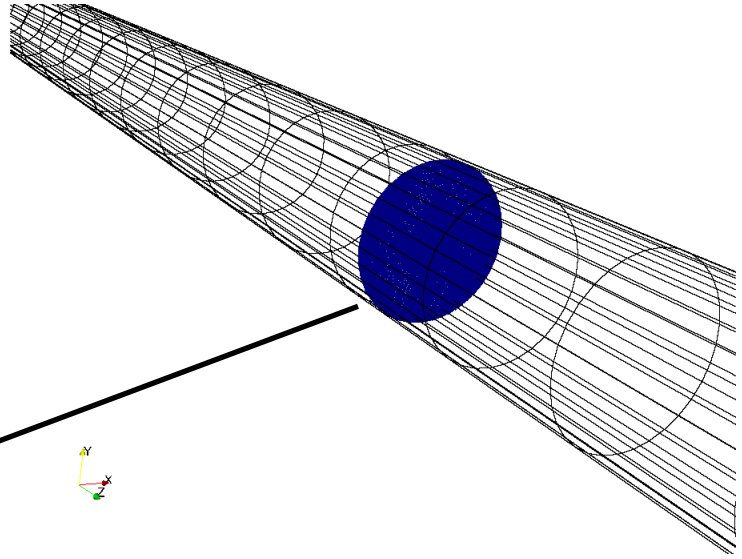


- Compute mass flow using **sum** operation with the field **phi**.
- In this case it works because we are sampling on a **faceZone**.
- The **faceZone** are internal faces of the mesh.

Supplement – More on sampling

The `functionobject3` dictionary

```
90          surface3_massflow
91          {
92              type            surfaceRegion;
93              functionObjectLibs ("libfieldFunctionObjects.so");
94              enabled         true;
95
96              writeControl    timeStep;
97              writeInterval   1;
98
99              log             true;
100             writeFields     false;
101
102             regionType      sampledSurface;
103             name            dummy;
104
105             sampledSurfaceDict
106             {
107                     type sampledTriSurfaceMesh;
108                     surface surface2.stl;
109                     source cells;
110                     interpolate true;
111             }
112
113             operation       areaNormalIntegrate;
114             fields
115             (
116                 U
117             );
118
119         }
```



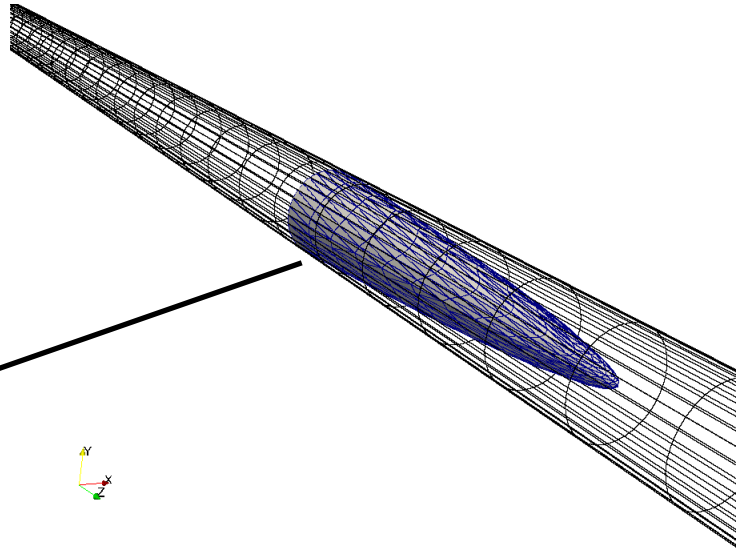
In this case we are using a finer surface therefore we enable the option interpolate.

Compute mass flow using **areaNormalIntegrate** operation with the field **U**.

Supplement – More on sampling

The `functionobject3` dictionary

```
123         surface4_massflow
124         {
125             type            surfaceRegion;
126             functionObjectLibs ("libfieldFunctionObjects.so");
127             enabled         true;
128
129             writeControl    timeStep;
130             writeInterval   1;
131
132             log             true;
133             writeFields     false;
134
135             regionType      sampledSurface;
136             name            dummy;
137
138             sampledSurfaceDict
139             {
140                     type sampledTriSurfaceMesh;
141                     surface surface3.stl;
142                     source insideCells;
143                     interpolate true;
144             }
145
146             operation       areaNormalIntegrate;
147             fields
148             (
149                 U
150             );
151
152         }
```



Compute mass flow using **areaNormalIntegrate** operation with the field **U**.

Supplement – More on sampling

The `functionobject3` dictionary

```
156         plane1_massflow
157         {
158             type            surfaceRegion;
159             functionObjectLibs ("libfieldFunctionObjects.so");
160             enabled         true;
161
163             writeControl    timeStep;
164             writeInterval   1;
165
166             log             true;
167             writeFields     false;
168
169             regionType      sampledSurface;
170             name            dummy;
171
172             sampledSurfaceDict
173             {
174                     type plane;
175                     planeType pointAndNormal;
176                     pointAndNormalDict
177                     {
178                         basePoint    (0 0 0.1);
179                         normalVector (0 0 1);
180                     }
181             }
182
184             operation       areaNormalIntegrate;
185             fields
186             (
187                 //phi
188                 U
189             );
190         }
191
193     }
```

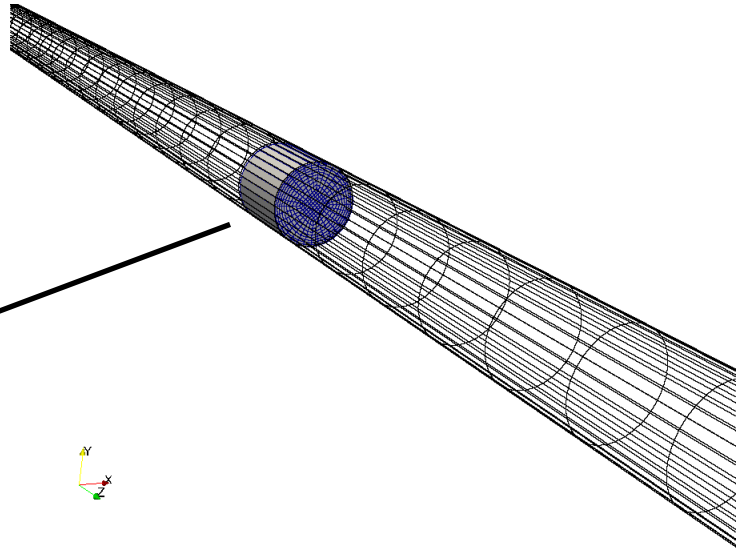
We sample in a plane.

Compute mass flow using **areaNormalIntegrate** operation with the field **U**.

Supplement – More on sampling

The `functionobject4` dictionary

```
17    functions
18    {
19
22        cells_fo1
23        {
24            type            volRegion;
25            functionObjectLibs ("libfieldFunctionObjects.so");
26            enabled         true;
27
28            writeControl    timeStep;
29            writeInterval   1;
30
31            log             true;
32
33            writeFields     false;
34
35            regionType cellZone;
36            name internalcells;
37
38            operation volIntegrate;
39
40            fields
41            (
42                  U
43            );
44
45        }
97
98    }
99
```



Compute **volIntegrate** of the field **U**.

Remember, we can do **cellSource** and **faceSource** sampling only on **cellZone/faceZone**.

Supplement – More on sampling

Exercises

- Where is located the source code of the utility `postProcess`?
- Try to do the sampling in parallel? Does it run? What about the output file? Is it the same?
- How many options are there available to do sampling in a line?
- How many options are there available to do sampling in a surface?
- Compute the mass flow at the inlet and outlet patches using the operation **sum** and **areaNormalIntegrate**. Do you get the same output?
- Compute the mass flow at the inlet and outlet patches using paraFoam/ParaView and compare with the output of the `postProcess` utility. Do you get the same results?
- Do point, line, and surface sampling using paraFoam/ParaView and compare with the output of the `postProcess` utility. Do you get the same results?

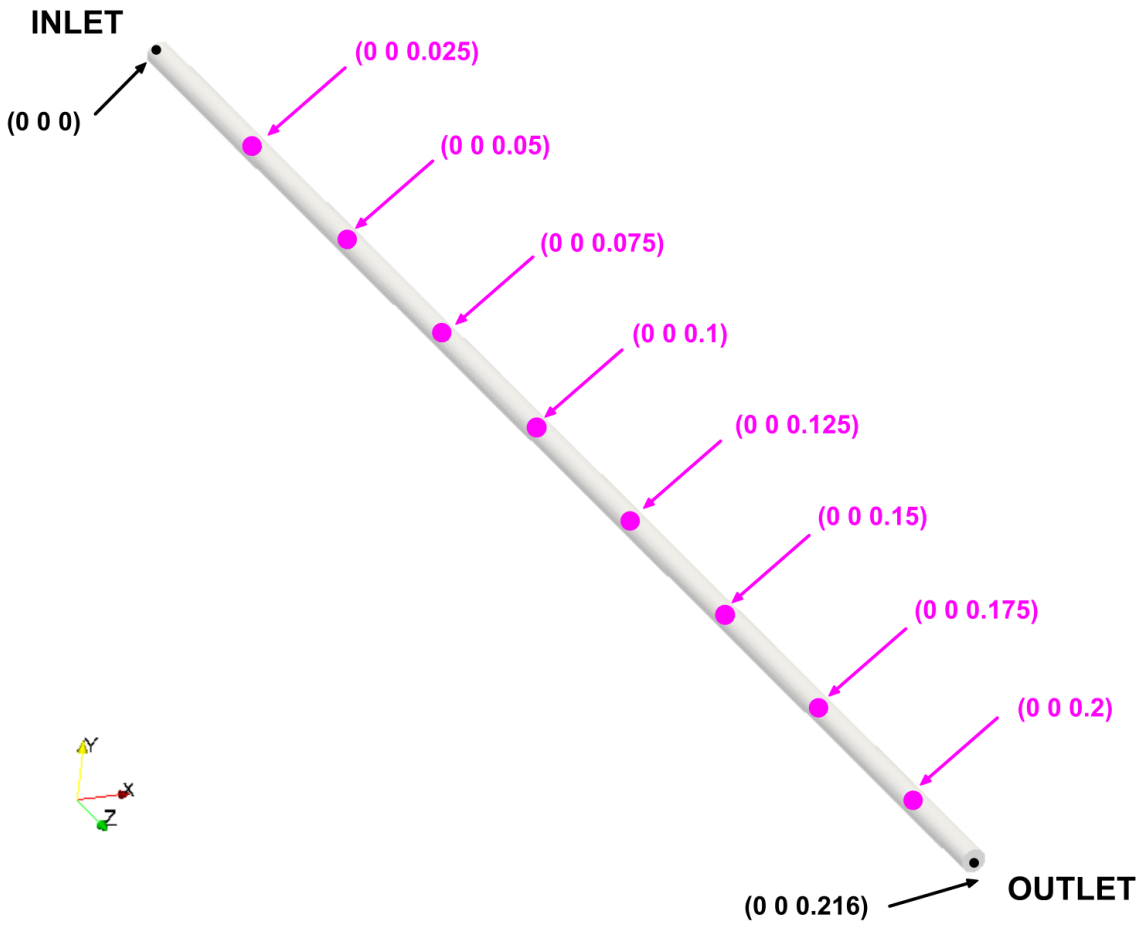
Sampling on points (probing)

Supplement – More on sampling

- OpenFOAM® provides the `postProcess` utility to probe field data for plotting.
- The probing locations are specified in the dictionary *`probesDict`* located in the case **`system`** directory.
- You can give any name to the input dictionary, hereafter we are going to name it *`probesDict`*.
- During the probing, inside the case directory a new directory named **`postProcessing`**, will be created. In this directory, the sampled values are stored.
- This utility can sample only points.
- Data can be written in many formats, including well-known plotting packages such as: grace/xmgr, gnuplot and jPlot.
- The probing can be executed by running the utility `postProcess` in the case directory and according to the application syntax.
- A final word, this utility does not do the sampling while the solver is running. It does the sampling after you finish the simulation.

Supplement – More on sampling

Laminar flow in a straight pipe – Re = 600



Probes location

Supplement – More on sampling

- We hope you did not erase the previous solution because we will use it to play around with the `probeLocations`. In the terminal type:

1. ```
   $> postProcess -func probesDict
   ```

- This will probe all the saved solutions at the specified locations. It will save time vs. quantity of interest.

- The sampled information is always saved in the directory (same name as the input file),
  - **`postProcessing/probesDict`**

- As we started to sample from time 0, the sample data is saved in the directory
  - **`postProcessing/probesDict/0`**

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

Supplement – More on sampling

The *probesDict* dictionary

```
17    type probes;
18
20    fields
21    (
22        p
23        U
24    );
25
27    probeLocations
28    (
29        (0 0 0.025)
30        (0 0 0.050)
31        (0 0 0.075)
32        (0 0 0.10)
33        (0 0 0.125)
34        (0 0 0.150)
35        (0 0 0.175)
36        (0 0 0.20)
37    );
```

Fields to sample. The output files will have the name of the fields.

Probe locations.

Supplement – More on sampling

The output files

- The output format of the probing is as follows:

Scalars

```
# Probe 0 (0 0 0.025)
# Probe 1 (0 0 0.05)
# Probe 2 (0 0 0.075)
# Probe 3 (0 0 0.1)
#       Probe           0           1           2           3
#       Time
        0               0           0           0           0
        0.005           19.1928     16.9497     14.2011     11.7580
        0.01            16.6152     14.5294     12.1733     10.0789
        …
        …
        …
```

Supplement – More on sampling

The output files

- The output format of the probing is as follows:

Vectors

```
# Probe 0 (0 0 0.025)
# Probe 1 (0 0 0.05)
# Probe 2 (0 0 0.075)
# Probe 3 (0 0 0.1)
#       Probe           0               1               2               3
#       Time
        0               (0 0 0)         (0 0 0)         (0 0 0)         (0 0 0)
        0.005           (0 0 2.1927)    (0 0 2.1927)    (0 0 2.1927)    (0 0 2.1927)
        0.01            (0 0 2.5334)    (0 0 2.5334)    (0 0 2.5334)    (0 0 2.5334)
        …
        …
        …
```

Supplement – More on sampling

Exercises

- Try to do the sampling in parallel? Does it run? What about the output file? Is it the same?
- Do the same sampling using paraFoam/paraview and compare with the output of the `postProcess` utility. Do you get the same results?
- Compute the descriptive statistics of each column of the output files using gnuplot. Be careful with the parentheses of the vector files.

  (Hint: you can use sed within gnuplot)

On-the-fly postprocessing functionObjects and the postProcess utility

Supplement – More on sampling

- It is possible to perform data extraction/manipulation operations while the simulation is running by using the **functionObjects**.
- **functionObjects** are small pieces of code executed at a regular interval without explicitly being linked to the application.
- When using **functionObjects**, files of sampled data can be written for plotting and post processing.
- **functionObjects** are specified in the **controlDict** dictionary and executed every time step or pre-defined intervals.
- All **functionObjects** are runtime modifiable.
- All the information related to the **functionObject** is saved in the directory **postProcessing** or in the solution directory.
- It is also possible to execute **functionObject** after simulation is over, we will call this running **functionObject** a-posteriori.

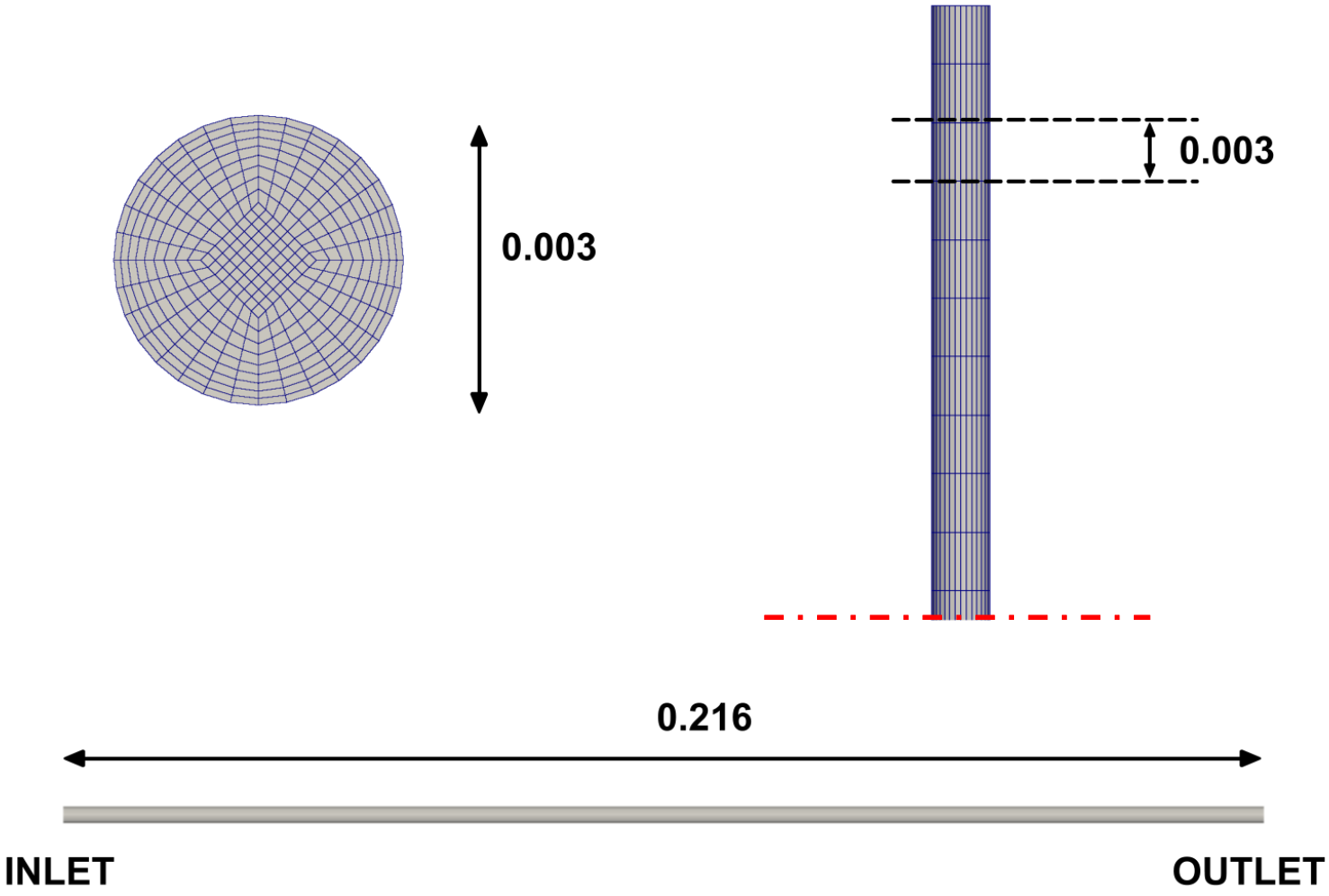
Supplement – More on sampling

- Let us do some on-the-fly postprocessing.
- Let us revisit the pipe case. Go to the directory:

`$PTOFC/advanced_postprocessing/pipe`

- In the case directory, you will find the `README.FIRST` file. In this file, you will find the general instructions of how to run the case. In this file, you might also find some additional comments.
- You will also find a few additional files (or scripts) with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on. These files can be used to run the case automatically by typing in the terminal, for example, `sh run_solver`.
- We highly recommend to open the `README.FIRST` file and type the commands in the terminal, in this way you will get used with the command line interface and OpenFOAM® commands.
- If you are already comfortable with OpenFOAM®, use the automatic scripts to run the cases.

Supplement – More on sampling



Mesh and domain

Supplement – More on sampling

What are we going to do?

- We will use this case to introduce **functionObjects**.
- We will use the utility `postprocess` to run **functionObjects** a-posteriori.
- We will also use the utility `postprocess` to compute some quantities in patches.
- We will compare the numerical solution with the analytical solution.
- We will do data cleaning and data analytics using shell scripting.
- To find the numerical solution we will use the solver `pisoFoam`.
- After finding the numerical solution we will do some sampling.
- At the end, we will do some plotting (using gnuplot or Python) and scientific visualization.

Supplement – More on sampling

Running the case

- Let us run the simulation. In the terminal type:

1. `$> foamCleanTutorials`
2. `$> blockMesh`
3. `$> checkMesh`
4. `$> topoSet`
5. `$> pisoFoam | tee log.solver`
7. `$> paraFoam`

- In step 4 we use the utility `topoSet` to do mesh topological manipulation. This utility will read the dictionary *`topoSetDict`* located in the **`system`** directory. Later on, we will talk about what are we doing in this step.
- In step 5 we run the simulation and save the log file. In step 6 we use `pyFoamPlotWatcher.py` to plot the residuals on-the-fly. As the job is running in background, we can launch this utility in the same terminal tab.
- Finally, in step 7 we visualize the solution.

Supplement – More on sampling

- Let us explore the case directory.
- You will notice that we now have a new directory named **`postProcessing`**.
- Inside this directory, you will find many subdirectories pointing to the **functionObject** used.
- By the way, we are saving a large amount of information.
- This is typical of unsteady simulations, and sometimes it can be too daunting to post-process and analyze the data.
- To ease the pain of doing data analytics and post-processing, you can use shell scripting or Python scripting.
- Hereafter, we are going to address how to use shell scripting.

Supplement – More on sampling

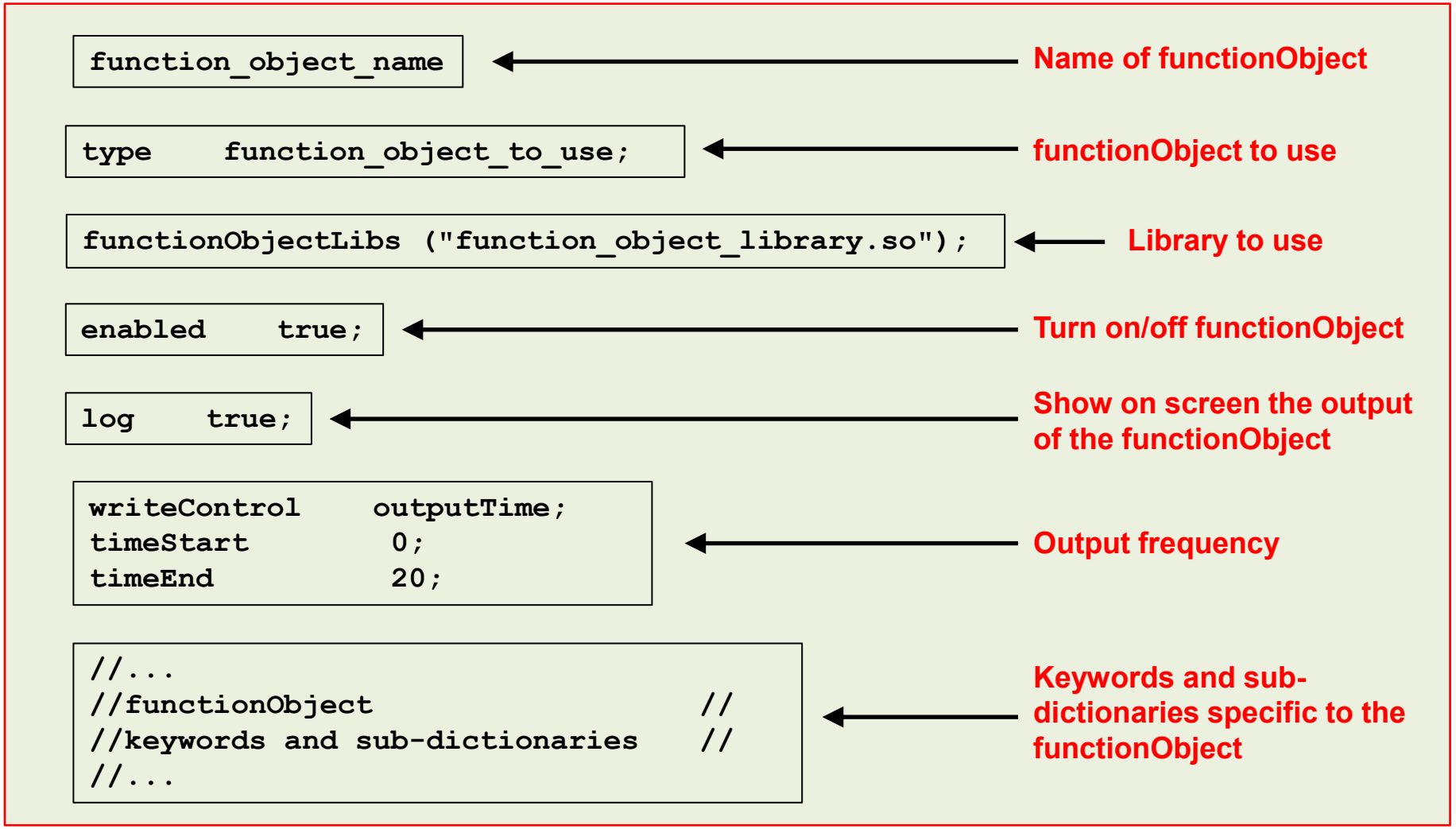
- Inside the directory **`postProcessing`**, you will find the following sub-directories:

  - `cells_fo1`
  - `cells_fo2`
  - `field_fo4`
  - `field_fo4.region1`
  - `field_fo4.region2`
  - `forces_object`
  - `inlet_average`
  - `inlet_massflow`
  - `inlet_massflow_posteriori`
  - `innerpatch_massflow`
  - `minmaxdomain`
  - `outlet_areaNormalIntegrate`
  - `outlet_massflow`
  - `outlet_massflow_posteriori`
  - `outlet_max`
  - `plane1_massflow`
  - `pressureDrop`
  - `pressureDrop.region1`
  - `pressureDrop.region2`
  - `probesDict`
  - `probes_fo1`
  - `probes_online`
  - `sampleDict1`
  - `sampleDict2`
  - `sets_fo2`
  - `sets_online`
  - `surface1_massflow`
  - `surface2_massflow`
  - `surface3_massflow`
  - `surface4_massflow`
  - `surfaces_fo3`

- Inside each sub-directory you will find the time directory `0`, this is the time directory from which we started to sample using **`functionObject`**. Inside this directory you will find the sampled data.
- If you start to sample from time `50`, you will find the time directory `50`.
- If you stop the simulation and restart it, let us say from time 10, you will find the time directories `0` and `10`.
- For line and surface sampling, you will find all the time directories corresponding to the saving frequency. Inside each directory you will find the sampled data.
- Let us take a look at the general organization of a **functionObject**.

Supplement – More on sampling

- The **functionObject** entry in the ***controlDict*** dictionary, contains at least the following information:



Supplement – More on sampling

- There are many **functionObjects** implemented in OpenFOAM®, and sometimes is not very straightforward how to use a specific **functionObject**.
- Also, **functionObjects** can have many options and some limitations.
- Our best advice is to read the doxygen documentation or the source code to learn how to use **functionObjects**.
- The source code of the **functionObjects** is located in the directory:

```
$WM_PROJECT_DIR/src/postProcessing/functionObjects
```

- Here after we are going to study a few commonly used **functionObjects**.

Supplement – More on sampling

The *controlDict* dictionary

```
51      functions
52      {


            name_of_the_functionObject_dictionary
            {
                Dictionary with the functionObject entries
            }


113     #include "functionObject0"
114
115     }
```

- Let us take a look at the bottom of the *controlDict* dictionary file.
- Here we define the **functionObjects**, which are functions that will do a computation while the simulation is running.
- In this case, we define the **functionObjects** in the sub-dictionary **functions** (lines 51-115).
- Each **functionObject** we define, has its own name and its compulsory keywords and entries.
- Notice that in line 113 we use the directive include to call an external dictionary with the **functionObjects** definition.
- If you use the include directive, you will need to update the *controlDict* dictionary in order to read any modification done in the included dictionary files.
- By the way, you can give any name to the input files defined in line 113.

Supplement – More on sampling

The *controlDict* dictionary

```
56      minmaxdomain
57      {
58          type fieldMinMax;
59
60          functionObjectLibs ("libfieldFunctionObjects.so");
61
62          enabled true;
63
64          mode component;
65
66          writeControl timeStep;
67          writeInterval 1;
68
69          log true;
70
71          fields (p U);
72      }
```

- **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

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 56).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *controlDict* dictionary

```
78      field_averages
79      {
80          type            fieldAverage;
81          functionObjectLibs ("libfieldFunctionObjects.so");
82          enabled         true;
83
84          writeControl    outputTime;
85          //writeControl    timeStep;
86          //writeInterval   100;
87
88          //cleanRestart true;
89
90          timeStart       0.05;
91          timeEnd         0.1;
92
93          fields
94          (
95              U
96              {
97                  mean        on;
98                  prime2Mean  on;
99                  base        time;
100             }
101
102             p
103             {
104                 mean        on;
105                 prime2Mean  on;
106                 base        time;
107             }
108         );
109     }
```

- **fieldAverage functionObject**
  - This **functionObject** is used to compute the average values of the field variables.
  - The output of this **functionObject** is saved in the time solution directories.

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
19      inlet_massflow
20      {
21
22          type            surfaceRegion;
23          functionObjectLibs ("libfieldFunctionObjects.so");
24          enabled         true;
25
26           //writeControl     outputTime;
27          writeControl    timeStep;
28          writesInterval  1;
29
30           log             true;
31
32           writeFields     false;
33
34           regionType      patch;
35           name            inlet;
36
37          operation       sum;
38           fields
39           (
40               phi
41           );
42       }
```

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

    postProcessing/inlet_massflow/0

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 19).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
48      outlet_massflow
49      {
50          type            surfaceRegion;
51          functionObjectLibs ("libfieldFunctionObjects.so");
52          enabled         true;
53
54          //writeControl     outputTime;
55          writeControl    timeStep;
56          writeInterval   1;
57
58          log             true;
59
60          writeFields     false;
61
62          regionTyp       patch;
63          Name            outlet;
64
65          operation       sum;
66          fields
67          (
68              phi
69          );
70      }
```

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

    postProcessing/outlet_massflow/0

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 48).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
76      probes_online
77      {
78          type            probes;
79          functionObjectLibs ("libfieldFunctionObjects.so");
80          enabled         true;
81          writeControl    outputTime;
82
83          probeLocations
84          (
85                      (0 0 0)
86                      (0 0 0.1)
87                      (0 0 0.2)
88          );
89
90          fields
91          (
92              U
93              p
94          );
95
96      }
```

- **probes functionObject**
  - This **functionObject** is used to probe field data at the given locations.
  - The output of this **functionObject** is saved in ascii format in the files *p* and *U* located in the directory

    postProcessing/probes_online/0

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 76).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
131    pressureDrop
132    {
133        type            fieldValueDelta;
134        functionObjectLibs ("libfieldFunctionObjects.so");
135        enabled         true;
136
137        region1
138        {
139            writeFields off;
140            type surfaceRegion;
141            regionType patch;
142            name inlet;
143
144            operation sum;
145
146            fields
147            (
148                phi
149            );
150        }
151
152        region2
153        {
154            writeFields off;
155            type surfaceRegion;
156            regionType patch;
157            name outlet;
158
159            operation sum;
160
161            fields
162            (
163                phi
164            );
165        }
166
167        operation subtract;
168    }
```

- **fieldValueDelta functionObject**
  - This **functionObject** is used to compute the difference/average/min/max of two field values.
  - We are using the operation sum with the field **phi**. This is equivalent to compute the mass flow.
  - We are using the operation subtract between the two field values.
  - The output of this **functionObject** is saved in ascii format in the file *fieldValueDelta.dat* located in the directory

    postProcessing/pressureDrop/0

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 131).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
174   innerpatch_massflow
175   {
176       type            surfaceRegion;
177       functionObjectLibs ("libfieldFunctionObjects.so");
178       enabled         true;
179
180       writeControl    timeStep;
181       writeInterval   1;
182
183       log             true;
184       writeFields     false;
185
186       regionType      faceZone;
187       Name            internalfacepatch;
188
189       operation       sum;
190       fields
191       (
192           phi
193       );
194
195   }
```

- **faceSource functionObject**
  - This **functionObject** is used to compute the mass flow in an inner patch.
  - In this case, we are sampling the faceZone **internalfacepatch**.
  - We are using the operation sum with the field **phi**.
  - The output of this **functionObject** is saved in ascii format in the file *faceSource.dat* located in the directory

postProcessing/innerpatch_massflow/0

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 174).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
201  sets_online
202  {
203      type            sets;
204      functionObjectLibs ("libfieldFunctionObjects.so");
205      enabled         true;
206      writeControl    outputTime;
207
208      interpolationScheme cellPointFace;
209      setFormat raw;
210
211      sets
212      (
213
214          set1
215          {
216              type            lineCellFace;
217
218              axis            x;
219              start           (-0.002  -0.002 0.2);
220              end             ( 0.002   0.002 0.2);
221          }
222
223      );
224
225      fields
226      (
227              U
228              p
229      );
230
231  }
```

- **sets functionObject**
  - This **functionObject** is used to sample field data in a line.
  - The output of this **functionObject** is saved in ascii format in the files *set1_p.xy* ***and*** *set1_U.xy* located in the time directories inside the folder

    postProcessing/sets_online

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 201).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

The *functionObject0* dictionary

```
237    forces_object
238    {
239        type forces;
240        functionObjectLibs ("libforces.so");
241        Enabled true;
242
243        //writeControl outputTime;
244        writeControl   timeStep;
245        writeInterval  1;
246
247        //// Patches to sample
248        patches ("walls");
249
250        //// Name of fields
251        pName p;
252        Uname U;
253
254        //// Density
255        rho    rhoInf;
256        rhoInf 1.;
257
258        //// Centre of rotation
259        CofR (0 0 0);
260    }
```

- **forces functionObject**
  - This **functionObject** is used to compute the forces on a patch.
  - In this case, we are sampling the patch **walls**.
  - The output of this **functionObject** is saved in ascii format in the file *forces.dat* located in the time directories inside the folder

    postProcessing/forces_object/0

- Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (line 237).

Note:
Use the banana method to know all the options available for each entry.

Supplement – More on sampling

Running functionObjects a-posteriori

- Sometimes, it can happen that you forget to use a **functionObject** or you want to execute a **functionObject** a-posteriori (when the simulation is over).
- The solution to this problem is to use the solver with the option `-postProcess`.
- This will only compute the **functionObject**, it will not rerun the simulation.
- For instance, let us say that you forgot to use a given **functionObject**.
- Open the dictionary *`controlDict`*, add the new **functionObject**, and type in the terminal,

  - `$> name_of_the_solver -postProcess -dict dictionary_location`

- By proceeding in this way you do not need to rerun the simulation, you just compute the new **functionObject**.

Supplement – More on sampling

- In the directory **`system`**, you will find the following **functionObjects** dictionaries: *`functionObject1`* , *`functionObject2`* , *`functionObject3`* , *`functionObject4`*, *`functionObject5`* , *`functionObject6.`*
- Try to figure out what we are doing in every **functionObject** dictionary.
- At this point, let us run each the **functionObject** a-posteriori. In the terminal type:

1. `$> pisoFoam -postProcess -dict system/functionObject1 | tee log_fo1`
2. `$> pisoFoam -postProcess -dict system/functionObject2`
3. `$> pisoFoam -postProcess -dict system/functionObject3 -time 0.05:0.1`
4. `$> pisoFoam -postProcess -dict system/functionObject4 -noZero`
5. `$> pisoFoam -postProcess -dict system/functionObject5 -latestTime`
6. `$> pisoFoam -postProcess -dict system/functionObject6 -latestTime`

Supplement – More on sampling

- In this step 1, we are reading the dictionary `functionObject1` located in the directory **`system`** (the dictionary file can be located anywhere), and we are doing the computation for all the saved solutions. Notice that we are redirecting the output to a log file (`| tee log_fo1`).
- In this step 2, we are reading the dictionary `functionObject2` and we are doing the computation for all saved solutions.
- In this step 3, we are reading the dictionary `functionObject3` and we are doing the computation for the time range 0.05 to 0.1 (`-time 0.05:0.1`).
- In this step 4, we are reading the dictionary `functionObject4` and we are doing the computation for all the saved solutions, except time zero (`-noZero`).
- In this step 5, we are reading the dictionary `functionObject5` and we are doing the computation only for he latest save solution (`-latestTime`).
- In this step 6, we are reading the dictionary `functionObject6` and we are doing the computation only for he latest save solution (`-latestTime`).

Supplement – More on sampling

Some shell and awk scripting

- Let us do some shell and awk scripting on the sampled data.
- Let us go to the directory **`postprocesssing/minmaxdomain/0`**
- To erase the parentheses in the file *`fieldMinMax.dat`* and save the output in the file *`out.txt`*, type in the terminal:
    - `$> cat fieldMinMax.dat | tr -d "()"  > out.txt`
- To extract the velocity from the file *`out.txt`*, and save the output in the file *`vel_minmax.txt`*, type in the terminal:
    - `$> awk '0 == NR % 2' out.txt > vel_minmax.txt`
- To extract the pressure from the file *`out.txt`*, and save the output in the file *`pre_minmax.txt`*, type in the terminal:
    - `$> awk '1 == NR % 2' out.txt > pre_minmax.txt`

Supplement – More on sampling

Some shell and awk scripting

- To erase the header of the file *pre_minmax.txt*, type in the terminal:
  - `$> awk '{if (NR!=1) {print}}' pre_minmax.txt > tmp`
  - `$> mv tmp pre_minmax.txt`
- To erase the header of the file *vel_minmax.txt*, type in the terminal:
  - `$> awk '{if (NR!=1) {print}}' vel_minmax.txt > tmp`
  - `$> mv tmp vel_minmax.txt`
- To compute the mean value of the seventh column of the file *pre_minmax.txt* (maximum pressure in this case), type in the terminal:
  - `$> awk 'NR>=50 && NR { total += $7; count++} END { print "Mean_value "  total/count}' pre_minmax.txt`

    We are computing the mean value of column 7 starting from row 50
- To compute the mean value of the seventh column of the file *vel_minmax.txt* (minimum Z velocity component in this case), type in the terminal:
  - `$> awk 'NR>=50 && NR { total += $5; count++} END { print "Mean_value "  total/count}' vel_minmax.txt`

    We are computing the mean value of column 7 starting from row 50

Supplement – More on sampling

Some shell and awk scripting

- In the directory **`scripts`** (located in the top level case directory) you will find the following scripts that will do some data processing automatically,
  - *`script_cleanfile`*
  - *`script_cleanforce`*
  - *`script_extractfields`*
  - *`script_force_coe`*
  - *`script_forces`*
  - *`script_meanvalues`*
  - *`script_minmax`*
  - *`script_probes`*

- To run the scripts you need to be inside the **`scripts`** directory. In the terminal type,
  - `cd scripts`
  - `sh script_name`

- Feel free to reuse them and adapt these scripts according to your needs.

Supplement – More on sampling

Some shell and awk scripting

- The script *script_force_coe* will compute the mean value and standard deviation of the lift and drag coefficients (in this case they are not computed).
- The script *script_forces* will extract the force components and saved in a clean file.
- The script *script_minmax* will extract the minimum and maximum values of the field variables.
- The script *script_probes* will extract the information of the probes.

We just showed how to compute the average and standard deviation and do some manipulation of the information saved in the output files, but you can do many things in an automatic way.

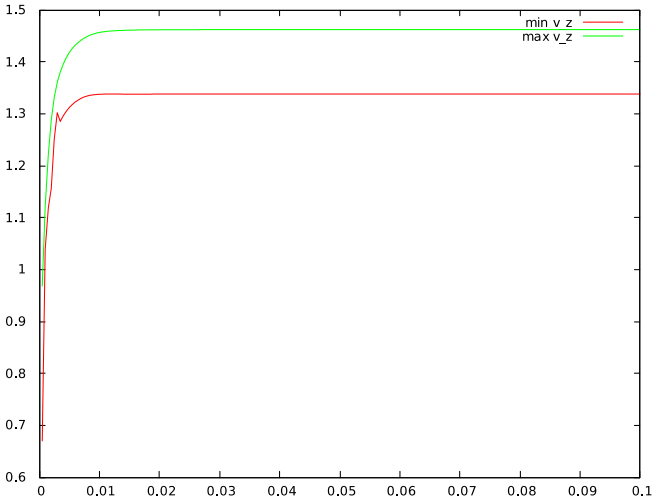
The power of scripting!!!

Supplement – More on sampling

Plotting in gnuplot

- Let us do some plotting using gnuplot. Type in the terminal (you must be inside the **`scripts`** directory):

1. ```
   $> sh script_minmax
   ```
2. ```
   $> gnuplot
   ```
3. ```
   gnuplot> plot [][] 'vel_minmax.txt' u 1:5 w l title "min v_z",
                      '' u 1:11 w l title "max v_z"
   ```


   ```

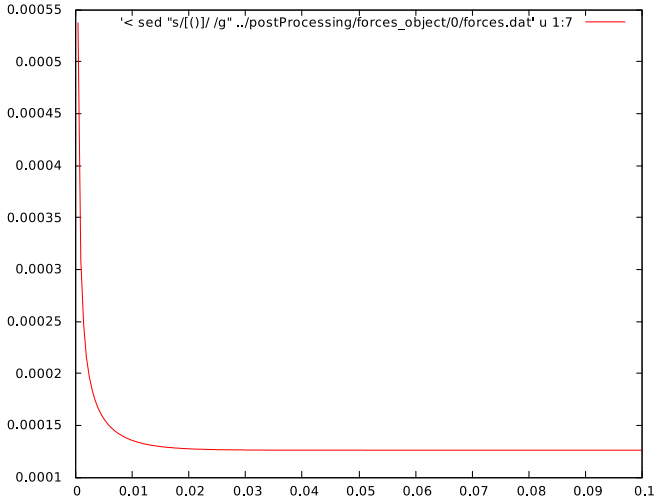
Supplement – More on sampling

Plotting in gnuplot

- Let us do some plotting using gnuplot. Type in the terminal (you must be inside the **`scripts`** directory):

1.
```
gnuplot> plot [][] '< sed "s/[()]/ /g"
                   ../postProcessing/forces_object/0/forces.dat' u 1:7 w l
```

- In this step we are using sed inside gnuplot to clean the parentheses. If you do not erase the parentheses in the input file, gnuplot will complain.



Supplement – More on sampling

Exercises

- Where is located the source code of the **functionObjects**?
- Try to run in parallel? Do all **functionObjects** work properly?
- Try to run **functionObjects** a-posteriori in parallel? Does it work? Do all **functionObjects** work properly?
- Compute the Courant number using **functionObjects**.
- Compute the total pressure and velocity gradient using **functionObjects** (on-the-fly and a-posteriori).
- Sample data (points, lines and surfaces) using **functionObjects** (a-posteriori).
- Is it possible to do system calls using **functionObjects**? If so what **functionObjects** will you use and how do you use it? Setup a sample case.
- Is it possible to update dictionaries using **functionObjects**? If so what **functionObjects** will you use and how do you use it? Setup a sample case.
- What are the compulsory entries of the **functionObjects**.