Module 5

The postprocess utility – Sampling – Probing – On-the-fly postprocessing – Field manipulation – Data conversion

Roadmap

1. **On-the-fly postprocessing – functionObjects and the postProcess utility**
2. Sampling with the postProcess utility
3. Field manipulation
4. Data conversion

On-the-fly postprocessing – functionObjects

- It is possible to perform data sampling, extraction and manipulation while the simulation is running by using monitors or as they are called in OpenFOAM, **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 at pre-defined intervals.
- All **functionObjects** are runtime modifiable.
- Depending on the **functionObject** you are using, its output is saved in the directory **`postProcessing`** or in the solution directory (time directories).
- It is also possible to execute **functionObjects** after the simulation is over, we will call this running **functionObjects** a-posteriori.
- You can use **functionObjects** to compute the Mach number, the vorticity field, and to sample the velocity at given points or along a line, and everything while the simulation is running.

On-the-fly postprocessing – functionObjects

- In the directory **`$FOAM_SRC/functionObjects`** you will find the source code for the **functionObjects**.
- There are many **functionObjects**, and according to what they do, they are located in different sub-directories, namely, **`field`**, **`forces`**, **`lagrangian`**, **`solvers`**, and **`utilities`**. Just to name a few **functionObjects**:

  - **CourantNo**
  - **div**
  - **fieldAverage**
  - **fieldValues**
  - **grad**
  - **MachNo**
  - **Q**
  - **vorticity**
  - **yPlus**
  - **forceCoeffs**
  - **forces**
  - **WallShearStress**
  - **scalarTransport**
  - **codedFunctionObject**
  - **residuals**
  - **systemCall**
  - **timeActivatedFileUpdate**
  - **writeObjects**

- In addition to the **functionObjects** located in the directory **`$FOAM_SRC/functionObjects`**, you can also run the sampling and co-processing utilities on-the-fly.
- You will find the source code for the sampling and co-processing utilities in the directory **`$FOAM_SRC/sampling`**.

On-the-fly postprocessing – functionObjects

- **functionObjects** are defined in the ***controlDict*** dictionary.
- To execute a **functionObject** you need to at least define the following entries:

```
function_object_name
```
User given name

```
type     function_object_to_use;
```
functionObject to use

```
functionObjectLibs ("function_object_library.so");
```
Library to use
Instead of functionObjectLibs, you can also use libs

```
enabled    true;
```
Turn on/off functionObject

```
log        true;
```
Show on screen the output of the functionObject

```
writeControl       outputTime;
timeStart          0;
timeEnd            20;
```
Output frequency

```
// ...
// functionObject
// keywords and sub-dictionaries
// ...
```
Keywords and sub-dictionaries specific to the functionObject

On-the-fly postprocessing – functionObjects

- There are many **functionObjects** implemented in OpenFOAM®, and they can have many options, as well as limitations.
- Our best advice is to read the doxygen documentation or the source code to learn how to use the **functionObjects**.
- Remember, the source code of the **functionObjects** is located in the directory:

  `$WM_PROJECT_DIR/src/functionObjects`

- The source code of the sampling and co-processing utilities is located in the directory:

  `$WM_PROJECT_DIR/src/sampling`

- The source code of the database entries required by the **functionObjects** is located in the directory:

  `$FOAM_SRC/OpenFOAM/db/functionObjects`

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

On-the-fly postprocessing – functionObjects

- Let us do some on-the-fly postprocessing.
- For this we will use the multi-element airfoil 2D case.
- You will find this case in the directory:

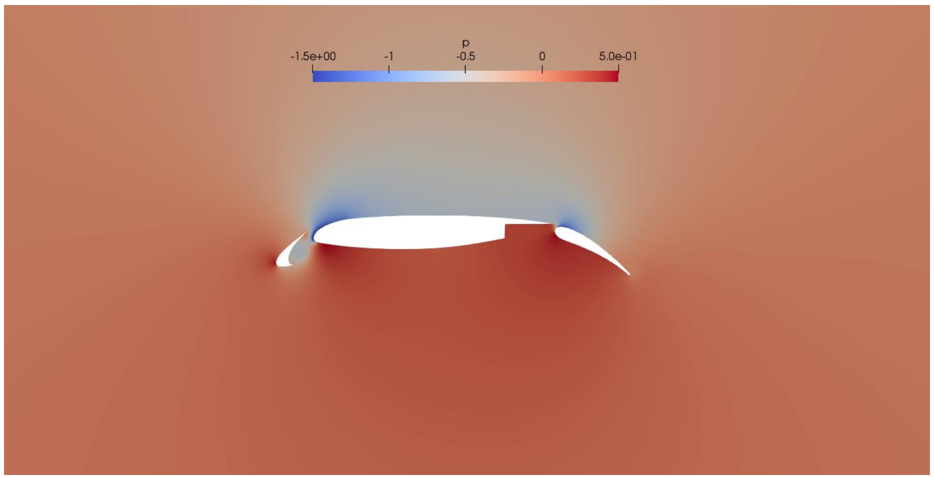
```
$PTOFC/101postprocessing/MDA_30P30N
```

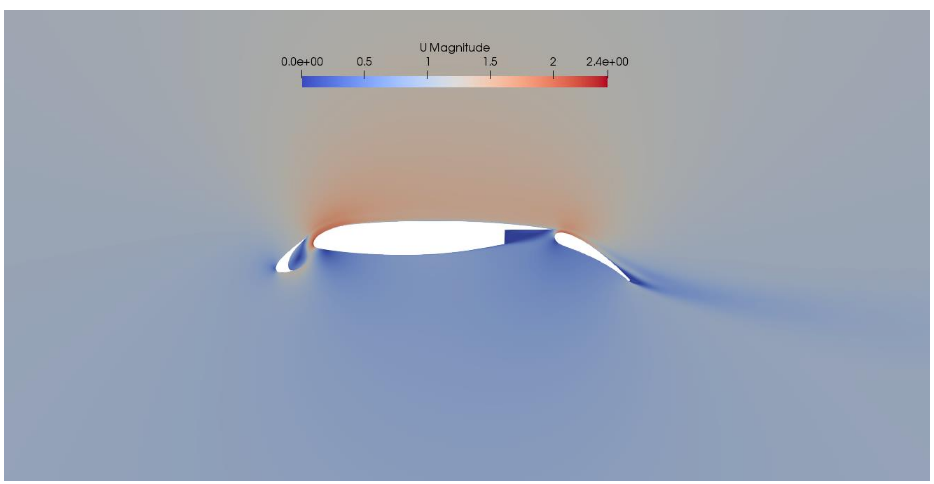
- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, 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, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

On-the-fly postprocessing – functionObjects

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

Qualitative post-processing





Quantitative post-processing

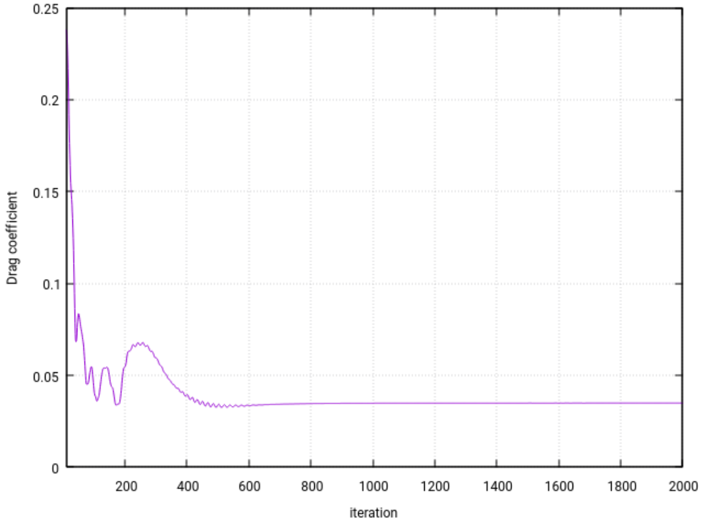
	c_d	c_l
Experimental values	0.0332	2.167
Numerical values	0.0346	2.238

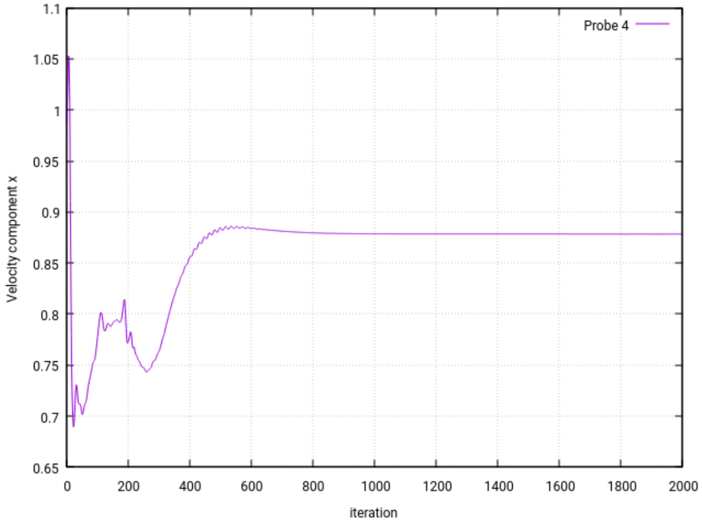
Additionally, by using **functionObjects** we will compute many derived quantities, such as,

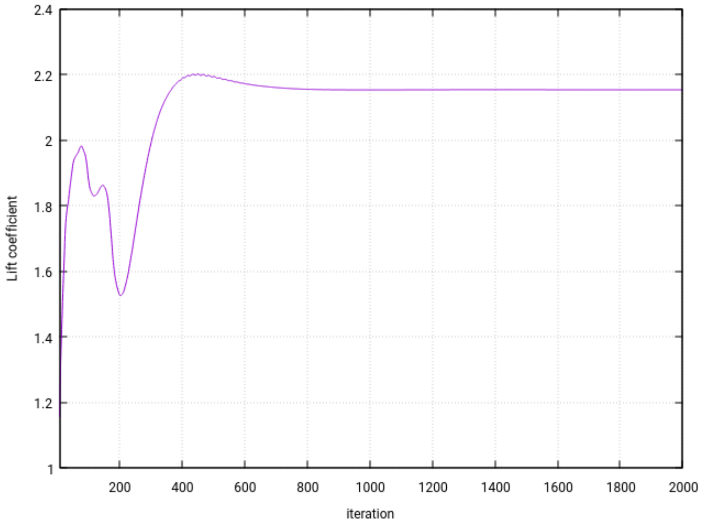
- yPlus.
- Voriticity.
- Mean values of the field variables (notice that we will compute the average of a steady solution).
- Forces.
- Force coefficients.
- Minimum and maximum values of the field variables.
- Sampling at given points.
- Mass flow at inlets and outlets.

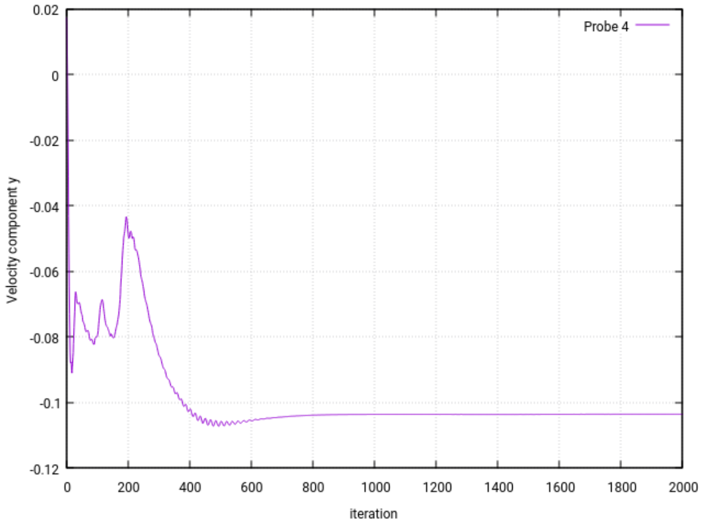
On-the-fly postprocessing – functionObjects

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





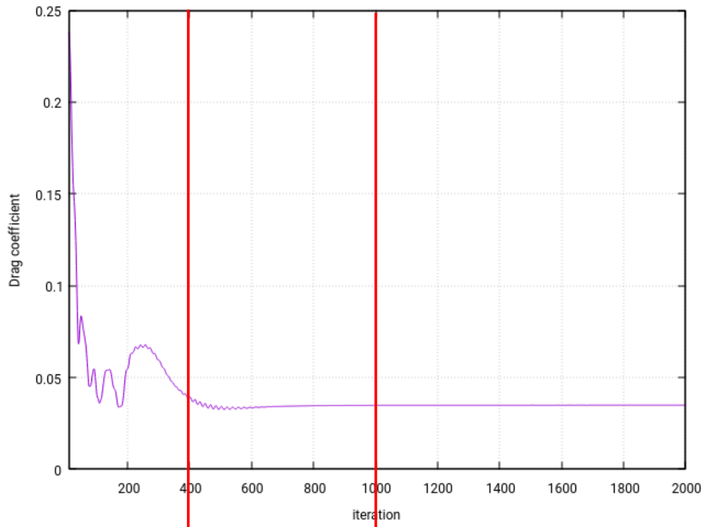


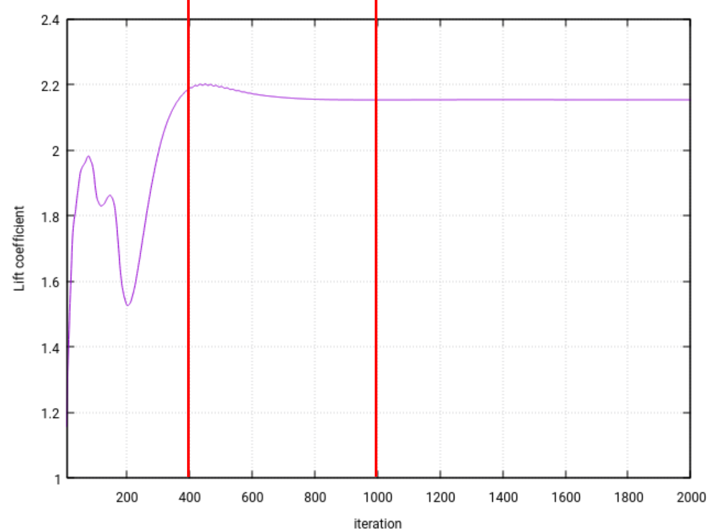


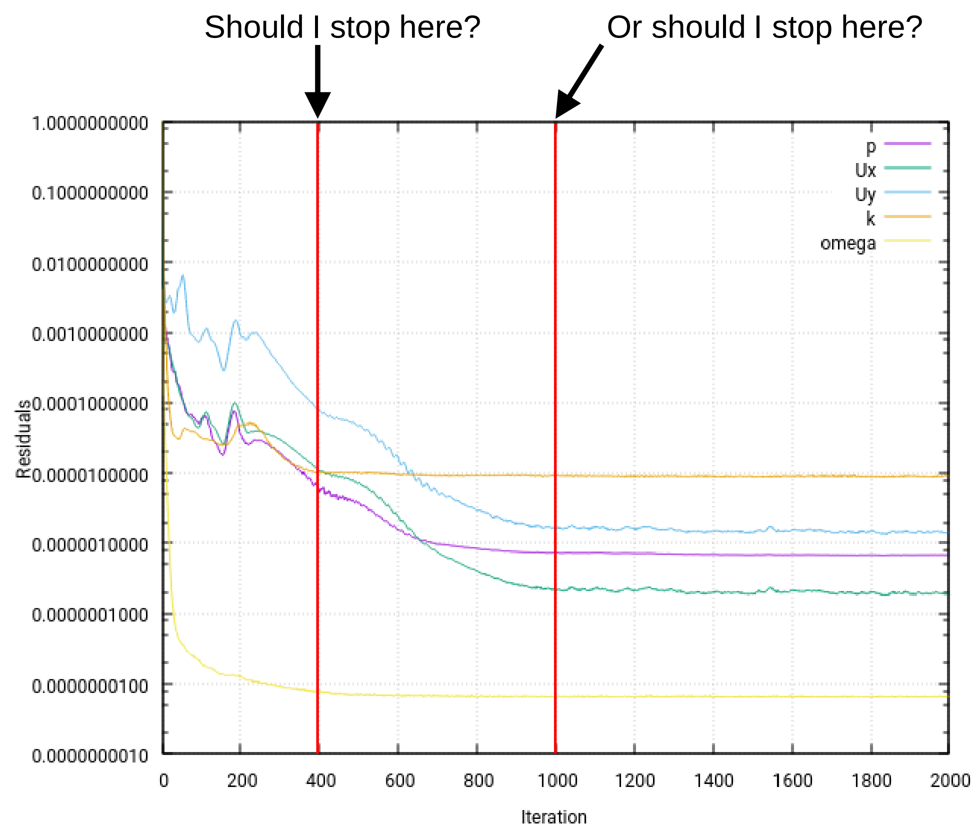
Quantitative post-processing

On-the-fly postprocessing – functionObjects

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







Quantitative post-processing – Assessing residuals

On-the-fly postprocessing – functionObjects

Running the case

- Let us run this case using the automatic scripts distributed with the tutorial. In the terminal type:

```
1. $> sh run_all.sh
```

- After the simulation is finish, you will find the decomposed directories (**`processor0`**, **`processor1`**, **`processor2`** and **`processor3`**), the **`postProcessing`** directory, and the **`2000`** directory. The solution, and output of the **functionObjects**, is saved in these directories.
- Remember, to visualize the decomposed solution you will need to launch `paraFoam` as follows,

```
1. $> paraFoam -builtin
```

- Do not erase the solution as we are going to use it in the next section.

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
48      functions
49      {

            name_of_the_functionObject_dictionary
            {
                Sub-dictionary with functionObject entries
            }

398     #include "externalFunctionObject"

402     }
```

- Let us take a look at the bottom of the `controlDict` dictionary file. In this dictionary is where we define all **functionObjects.**
- Within this dictionary, **functionObjects** are defined in the sub-dictionary **functions**, *i.e.*,

functions
{
    functionObjects definition
};

- In this case, the **functionObjects** are defined in lines 48-402 (the sub-dictionary **functions**).
- Each defined **functionObject** has its own name and its compulsory keywords and entries.
- Notice that in line 398 we use the directive **include** to call an external dictionary with the **functionObjects** definition.
- If you do not give the path of the external dictionary, the solver will look for it in the directory **`system`**.
- 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.

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
48      functions
49      {

204      forces_object
205      {
206             type forces;

207             functionObjectLibs ("libforces.so");

210             writeControl   timeStep;
211             writetInterval  1;

213             enabled true;

215             //// Patches to sample
216             patches ("wall_slat" "wall_airfoil" "wall_flap");

218             //// Name of fields
219             pName p;
220             Uname U;

222             //only for incompressible flows
223             rho rhoInf;
224             rhoInf 1.0;

226             //// Centre of rotation
227             CofR (0 0 0);

228      }

402      }
```

functionObject identifier (user given)

- Let us explain in detail how to setup a **functionObject**.
- As the names implies, this **functionObject** is used to compute the forces on a given body or set of bodies (line 204).
- You can add as many forces **functionObjects** (or any other one) as you like, but you should assign them different identifiers (line 204). Remember not to use white spaces when naming **functionObjects**.
- The output of this **functionObject** is saved in the directory **`postProcessing/forces_object`**, where the directory name is taken from line 204.
- Inside this directory, you will find the subdirectory **`0`**, which means that you started to sample data from time **`0`**.
- If you start from a different time, you will find a different subdirectory, *e.g.*, **`86.05`**
- Remember, different **functionObjects** will have different entries, to know the entries just refer to the online documentation or skim the source code, which is located in the directory,
  - **`$WM_PROJECT_DIR/src/functionObjects`**

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
functions
{

 forces_object
 {
        type forces;

        functionObjectLibs ("libforces.so");

        writeControl   timeStep;
        writetInterval  1;

        enabled true;

        //// Patches to sample
        patches ("wall_slat" "wall_airfoil" "wall_flap");

        //// Name of fields
        pName p;
        Uname U;

        //only for incompressible flows
        rho rhoInf;
        rhoInf 1.0;

        //// Centre of rotation
        CofR (0 0 0);

 }

 }
```

- Let us study all entries of the forces **functionObject**

functionObject identifier (user given)

functionObject to use

functionObject library to use

Controls for saving frequency

Turn on/off **functionObject**

Compute the forces on these patches

Name of the velocity and pressure fields. If you use different fields, *e.g.*, **pMean** and **Umean**, they need to be computed before this **functionObject**

Reference density value. It only needs to be defined for incompressible flows. For compressible flows, the computed density is used instead (you will need to define a dummy value, though)

Reference center of rotation to compute moments

Note:

- The source code of this functionObject is located in the directory **`$FOAM_SRC/functionObjects/forces/forces`**
- Use the banana method to know all the options available for each entry.

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
functions
{

 forceCoeffs_object
 {
        type forceCoeffs;
        functionObjectLibs ("libforces.so");

        enabled true;

        patches ("wall_slat" "wall_airfoil" "wall_flap");

        pName p;
        Uname U;

        rho rhoInf;
        rhoInf 1.0;

        log true;

        CofR (0.0 0 0);

        pitchAxis (0 0 1);
        magUInf 1.0;
        lRef 1;
        Aref 1;

        writeControl   timeStep;
        writeInterval  1;

        liftDir     (0 1 0);
        dragDir     (1 0 0);

 }

}
```

functionObject identifier (user given)

- Let us study now the **functionObject** used to compute the force coefficients.

- This **functionObject** computes the force coefficients.
- These entries are similar to those of the force **functionObject**

This option will output the values to a text file located in the directory **`postProcessing/forceCoeffs_object`**

Reference center of rotation to compute moments

Reference values used to compute coefficients

Controls for saving frequency

Reference axes to compute the lift and drag coefficients.

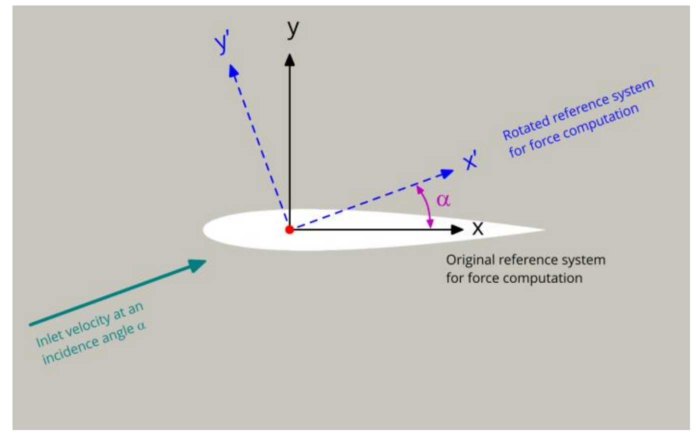
On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
functions
{

 forceCoeffs_object
 {
         type forceCoeffs;
         functionObjectLibs ("libforces.so");

         enabled true;

         patches ("wall_slat" "wall_airfoil" "wall_flap");

         pName p;
         Uname U;

         rho rhoInf;
         rhoInf 1.0;

         log true;


         CofR (0.0 0 0);


         pitchAxis (0 0 1);
         magUInf 1.0;
         lRef 1;
         Aref 1;

         writeControl   timeStep;
         writeInterval  1;

         liftDir      (0 1 0);
         dragDir      (1 0 0);

 }

 }
```

functionObject identifier (user given)

- Let us study now the **functionObject** used to compute the force coefficients.



$$\text{liftDir } (-\sin(\alpha), \cos(\alpha), 0)$$

$$\text{dragDir } (\cos(\alpha), \sin(\alpha), 0)$$

- Reference axes to compute the lift and drag coefficients.
- Remember, lift and drag are perpendicular and parallel to the incoming flow, respectively.
- So, if the inlet velocity is entering at a given angle, you should adjust the vectors **liftDir** and **dragDir** so they are aligned with the incoming flow (rotation matrix).

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
48    functions
49    {

120    minmaxdomain_scalar
121    {
122            type volFieldValue;
123            libs ("libfieldFunctionObjects.so");
125            enabled true;
126            log true;

129            writeControl timeStep;
130            writeInterval 1;

132            writeFields false;
134            writeLocation true;
138            regionType all;

140            operation none;

142            fields
143            (
144                p nuTilda nut
145            );
146    }

176    mindomain_scalar
177    {
178            $minmaxdomain_scalar
179            operation min;
180    }

188    maxdomain_scalar
189    {
190            $minmaxdomain_scalar
191            operation max;
192    }

402    }
```

- **volFieldValue functionObject**
  - This **functionObject** can be 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 `volFieldValue.dat` located in the directory

    `postProcessing/minmaxdomain_scalar/0`
  - Remember, the name of the directory where the output data is saved is the same as the name of the **functionObject** (*e.g.*, line 120).
  - In this case, we are splitting the computation of the minimum and maximum values in two parts.
    - In lines 120-146, we define the body of the **functionObject**.
    - In lines 176-180 and 188-192, we compute the minimum and maximum values of the scalar field variables (line 144), using the body of the **functionObject** (lines 120-146).
  - This particular **functionObject** definition can be use for scalar fields.

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
48    functions
49    {

148    minmaxdomain_vector
149    {
150            type volFieldValue;
151            libs ("libfieldFunctionObjects.so");
153            enabled true;
154            log true;

157            writeControl timeStep;
158            writeInterval 1;

160            writeFields false;
162            writeLocation true;
166            regionType all;

168            operation none;

170            fields
171            (
172                U
173            );
174    }

182    mindomain_vector
183    {
184            $minmaxdomain_scalar
185            operation minMag;
186    }

194    maxdomain_vector
195    {
196            $minmaxdomain_scalar
197            operation maxMag;
198    }

402    }
```

- **volFieldValue functionObject**
  - For vector fields, we can use the following **functionObject** definition.
  - For vectors we use the operations **minMag** (line 185) and **maxMag** (line 197).
  - Whereas for scalar fields, we use the operations **min** (line 179) and **max** (line 191).
  - In this case, the vector field variable is defined in line 172.
  - If the option **writeLocation** is enabled (line 134 or line 162), this **functionObject** will report the location of the minimum or maximum value.

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
48    functions
49    {


81      cellMin
82      {
83          #includeEtc "caseDicts/postProcessing/minMax/cellMin.cfg"
84          enabled       true;
85          log           true;
86          fields   (p);
87      }

89      cellMax
90      {
91          #includeEtc "caseDicts/postProcessing/minMax/cellMax.cfg"
92          enabled       true;
93          log           true;
94          fields   (p);
95      }

98      cellMinMag
99      {
100         #includeEtc "caseDicts/postProcessing/minMax/cellMinMag.cfg"
101         enabled       true;
102         log           true;
103         fields   (U);
104      }

106      cellMaxMag
107      {
108         #includeEtc "caseDicts/postProcessing/minMax/cellMaxMag.cfg"
109         enabled       true;
110         log           true;
111         fields   (U);
112      }


402   }
```

- The previous definition of the **functionObject** is expanded.
- In OpenFOAM, it is also possible to use packed **functionObjects**.
- The packed **functionObjects** are located in the directory **`$WM_PROJECT_DIR/etc/caseDicts`**
- To use packed **functionObjects** you just need to include them in the definition, *e.g.*, line 83.
- You will also need to add any optional entry, *e.g.*, lines 84-86.
- Notice that lines 81-112 are commented in the original dictionary.
- Lines 81-87, are equivalent to lines 120-146 and lines 176-180 in the expanded **functionObject**.
- Personally speaking, we prefer to use the expanded definition of the **functionObjects**.

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
functions
{

  cellMin
  {
      #includeEtc "caseDicts/postProcessing/minMax/cellMin.cfg"
      enabled       true;
      log           true;
      fields   (p);
  }

  cellMax
  {
      #includeEtc "caseDicts/postProcessing/minMax/cellMax.cfg"
      enabled       true;
      log           true;
      fields   (p);
  }

  cellMinMag
  {
      #includeEtc "caseDicts/postProcessing/minMax/cellMinMag.cfg"
      enabled       true;
      log           true;
      fields   (U);
   }

   cellMaxMag
   {
      #includeEtc "caseDicts/postProcessing/minMax/cellMaxMag.cfg"
      enabled       true;
      log           true;
      fields   (U);
   }

 }
```

- Packed **functionObject**

Name of the functionObject.
This is also the name of the folder where the output of the functionObject will be saved.

Location and type of the packed functionObject.
- cellMin – minimum value of a scalar field.
- cellMax – maximum value of a scalar field.
- cellMinMag – minimum value of a vector field (magnitude).
- cellMaxMag – maximum value of a vector field (magnitude).

Options related to the functionObject.
The functionObject can have more options than the ones shown here.
In this case we are using the following options:
- enabled = Enable/disable functionObject.
- log = Print to screen output of the functionObject

Fields to sample

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
48      functions
49      {

279      yplus
280      {
281              type yPlus;
282              functionObjectLibs ("libfieldFunctionObjects.so");
283              enabled true;
284              log     true;
285              writeControl outputTime;
286      }

402      }
```

- **yPlus functionObject**
  - This **functionObject** is used to compute the yPlus field.
  - This **functionObject** has two outputs, one output saved in the solution directories (**1**, **2**, **3**, and so on). You can visualize this output using paraview/paraFoam.
  - The second output is located in the directory

    postProcessing/yplus/0

  - In this file you will find the minimum, maximum and average values of yPlus in all patches defined as walls.
  - Remember, the name of the directory where the output data (descriptive statistics) is saved is the same as the name of the **functionObject** (line 279).

On-the-fly postprocessing – functionObjects

The `controlDict` dictionary

```
48    functions
49    {

358     fieldAverage1
359     {
360     type            fieldAverage;
361     libs ( "libfieldFunctionObjects.so" );
362     writeControl    writeTime;

366     timeStart    100;
367     //timeEnd    1000;

369     fields
370     (
371             U
372             {
373                     mean        on;
374                     prime2Mean  on;
375                     base        time;
376             }

378             p
379             {
380                     mean        on;
381                     prime2Mean  on;
382                     base        time;
383             }

385             nut
386             {
387                     mean        on;
388                     prime2Mean  on;
389                     base        time;
390             }
391     );
392     }

402     }
```

- **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.
  - In this case, we are computing the field averages of velocity (**U**), pressure (**p**), and turbulent viscosity (**nut**).
  - In line 366, we define the starting time to compute the statistics. If you do not define this value, the statistics will be computed starting from 0.
  - In line 367, we define the end time of the statistics (notice that this line is commented). If you do not define this value, the statistics will be computed until the end of the simulation.
  - In the source code you can find a description of all options for this **functionObject**. The source code is located in the directory:
    - `$WM_PROJECT_DIR/src/functionObjects/field/fieldAverage`
  - In this **functionObject**, prime2Mean is the average of the product of the fluctuations of the variable,

$$\overline{\phi'\phi'}$$

On-the-fly postprocessing – functionObjects

The *controlDict* dictionary

```
48      functions
49      {

398     #include "externalFunctionObject"

402      }
```

User given file name

- In line 398 we add a **functionObject** definition using an external file.
- In this case, the **functionObject** is located in the directory **`system`**
  - If you want to run this **functionObject** online, do not add lines 51, 52, and 211 in the file *`externalFunctionObject`*.
  - To run this functionObject a-posteriori (after the simulation is over by using the saved solution); add lines 51, 52, and 211 to the file *`externalFunctionObject`*.
  - We explain how to run **functionObjects** a posteriori later.

On-the-fly postprocessing – functionObjects

The *externalFunctionObject* dictionary

```
24      probes_online
25      {
26              type            probes;
27              functionObjectLibs ("libfieldFunctionObjects.so");
28              enabled         true;
29              writeControl timeStep;
30              writeInterval 1;
31
32              probeLocations
33              (
34                      (1   0    0)
35                      (2   0    0)
36                      (2   0.25 0)
37                      (2  -0.25 0)
38              );
39
40              fields
41              (
42                      U
43                      p
44              );
45
46      }

52      vorticity
53      {
54              type vorticity;
55              functionObjectLibs ("libfieldFunctionObjects.so");
56              enabled         true;
57              log             true;
58              writeControl    outputTime;
59      }
```

- **probes functionObject**
  - This **functionObject** is used to probe field data at the given locations.
  - In this case, we are sampling the fields **U** and **p** (lines 42-43)
  - 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 24).

- **vorticity functionObject**
  - This **functionObject** is used to compute the vorticity field.
  - The output of this **functionObject** is saved in the solution directories (**1**, **2**, **3**, and so on).
  - You can visualize this output using paraview/paraFoam.

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 new **functionObject**, it will not rerun the simulation.
- For example, 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`

- You also have the option of adding the new **functionObject** in an external file. If you chose this option, do not forget to add the **functionObject** within the **function** sub-dictionary block:

```
function
{
        //functionObject definitions here
};
```

Running functionObjects a-posteriori

- In the directory **`system`**, you will find the following **functionObject** external dictionaries: `functionObject1`
- To run this **functionObject** a-posteriori, type in the terminal:

```
1. $> simpleFoam -postProcess -dict system/functionObject1 -noZero
2. $> mpirun -np 4 simpleFoam -parallel -postProcess -dict system/functionObject1 -time 500:2000
3. $> simpleFoam -postProcess -dict system/functionObject1 -latestTime
```

- In step 1, we are reading the dictionary `system/functionObject1` and we are doing the computation for all the saved solutions, except time zero.
- In step 2, we are reading the dictionary `system/functionObject1` and we are doing the computation for the time range 500 to 2000. Notice that we are running in parallel.
- In step 3, we are reading the dictionary `system/functionObject1` and we are doing the computation only for he latest saved solution.
- If you do not give any time manipulator, the computation will be carried out on every saved solution.

On-the-fly postprocessing – functionObjects

Final remarks on functionObjects

- A **functionObject** that is very useful, but we did not use in this case:

```
inlet_massflow
{
      type                    surfaceFieldValue;
      functionObjectLibs      ("libfieldFunctionObjects.so");

      enabled          true;
      log              true;

      writeControl   timeStep;
      writeInterval  1;

      writeFields      false;

      regionType       patch;
      name             inlet;

      operation        sum;
      fields (phi);
}
```

Compute **functionObject** in a boundary patch

Compute **functionObject** in **this** boundary patch

- This **functionObject** is used to computed the mass flow across a boundary patch.
- Remember, the method is conservative so what is going in, is going out (unless you have source terms).
- So, if you want to measure the mass imbalance, setup this function object for each boundary patch where you have flow entering or going out of the domain.

On-the-fly postprocessing – functionObjects

Final remarks on functionObjects

- As you can see, there are many functionObjects implemented in OpenFOAM®.
- We just explained the most common **functionObjects**.
- You can use the banana method to know all the options available for each entry, search in the documentation, or read the source code located in the directory **`$FOAM_SRC/functionObjects`**
- In the supplement slides you will find more examples of more complex **functionObjects**.
- You will also find a deck of slides with a detailed explanation of advanced paraview features and some basic instructions for data plotting and analysis using gnuplot.
- Remember, you can also do the same postprocessing using paraview/paraFoam, but you will only work on the saved fields.
- A great advice before running your simulation, setup all your **functionObjects** and gather as much as possible quantitative data.

On-the-fly postprocessing – functionObjects

Exercises

- Where is located the source code of the **functionObjects**?
- Try to run in parallel? 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 **functionObject** 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**?

Roadmap

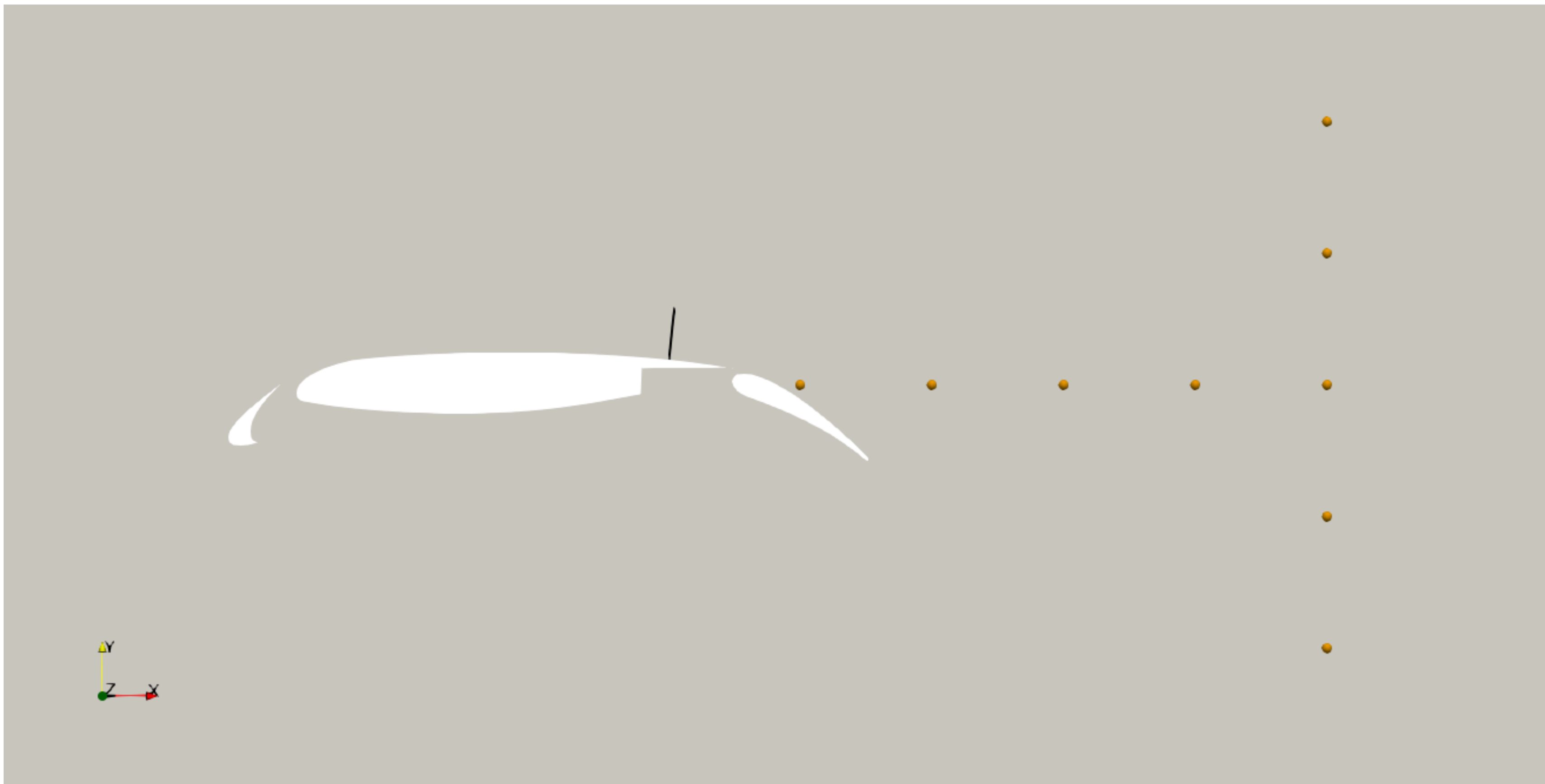
1. ~~On-the-fly postprocessing – functionObjects and the postProcess utility~~
2. **Sampling with the postProcess utility**
3. Field manipulation
4. Data conversion

Sampling with the postProcess utility

- OpenFOAM® provides the `postProcess` utility to sample field data for plotting.
- The sampling parameters are specified in a dictionary located in the case **`system`** directory.
- You can give any name to the input dictionary, hereafter we are going to name them *`sampleDict`* (to sample along a line) and *`probesDict`* (to sample in a set of probes).
- During the sampling, and inside the case directory, a new directory named **`postProcessing`** will be created. In this directory, the sampled values are stored in a sub-directory with the name of the input dictionary, in this case, **`sampleDict`** and **`probesDict`**.
- This utility can sample points, lines, and surfaces.
- Data can be written in a range of 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.

Sampling with the postProcess utility

- To do sampling, we will use the solution from the previous case.
- If you do not have the solution, follow the instructions given in the previous slides.
- Hereafter, we will sample along a line and in a few probe locations, as illustrated in the figure below.



Sampling with the postProcess utility

Running the case

- Let us do the sampling,

1. `$> postProcess -func sampleDict -time 2000`
2. `$> postProcess -func probesDict -time 2000`

- In step 1, we do some sampling using the dictionary *sampleDict*. We also do the sampling only for time 2000
- In step 2, we do some sampling using the dictionary *probesDict*. We also do the sampling only for time 2000.
- Remember, you can use different time manipulators.
- If you do not give any time manipulator option, the sampling will be computed for all saved solutions (including time directory 0).

Sampling with the postProcess utility

The `sampleDict` and `probesDict` dictionaries

- These dictionaries are located in the directory **`system`**.
- In this case, the `sampleDict` dictionary is used to sample along a line. This file contains several entries to be set according to the user needs. The following entries can be set,
    - 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, the `probesDict` is used to sample in a set of points. This file contains several entries to be set according to the user needs. The following entries,
    - The fields to be sample.
    - Location of the probes.
- The following **functionObjects** type can be used to do sampling: **patchProbes**, **probes**, **sets**, or **surfaces**.

Sampling with the postProcess utility

The *sampleDict* dictionary

```
type sets;

libs ("libsampling.so");

interpolationScheme cellPoint;

setFormat       raw;

surfaceFormat raw;

fields
(
    U
    wallShearStress
);

sets
(

    profile0
    {

        type    lineCellFace;

        axis    distance;

        start   (       0.75150 0.04767 0   );
        end     (       0.76168 0.14715 0   );

    }

);
```

Sample sets (points and lines).

Use sampling library

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

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).

Fields to sample. No need to mention that they must exist.

Sub-dictionary where we define all sampling objects (sets)

Name of the set and output file

Interpolation method (from the solution to the line).

Sample method definition

Location of the sample line. Definition if the start and end point

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

Sampling with the postProcess utility

The *sampleDict* dictionary

```
type sets;

libs ("libsampling.so");

interpolationScheme cellPoint;

setFormat       raw;

surfaceFormat raw;

fields
(
    U
    wallShearStress
);

sets
(

    profile0
    {

        type    lineCellFace;

        axis    distance;

        start   (       0.75150 0.04767 0   );
        end      (      0.76168 0.14715 0   );

    }

);
```

Name of sampled set

- Remember, the sampled data is always saved in the directory **`postProcessing`**
- Then, in the sub-directory **`sampleDict`** (whose name corresponds to the name of the input file), you will find the data sampled in a directory corresponding to the sampled time.
- For example, in this case you fill find the data in the directory **`postProcessing/sampleDict/2000`**
- Then, in the file *`profile0_U_wallShearStress.xy`* you will find the data.
- The name of the output file corresponds to the name of the sampled set, appended by the name of the sampled fields.
- Different files will be created for tensor, vector and scalar fields.
- Feel free to open the output files using your favorite text editor.

Sampling with the postProcess utility

The *probesDict* dictionary

```
type probes;



(
    p
    U
);



probeLocations
(
    (1.0    0 0)
    (1.25   0 0)
    (1.5  0 0)
    (1.75   0 0)
    (2.0    0 0)
    (2.0 -.25 0)
    (2.0 -.5 0)
    (2.0  .25 0)
    (2.0  .5 0)
);
```

Sample points.

Fields to sample. No need to mention that they must exist.

Location of the points.

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

Sampling with the postProcess utility

The *probesDict* dictionary

```
type probes;

(
    p
    U
);

probeLocations
(
    (1.0    0 0)
    (1.25   0 0)
    (1.5   0 0)
    (1.75   0 0)
    (2.0    0 0)
    (2.0 -.25 0)
    (2.0 -.5 0)
    (2.0   .25 0)
    (2.0   .5 0)
);
```

- Remember, the sampled data is always saved in the directory **`postProcessing`**
- Then, in the sub-directory **`probesDict`** (whose name corresponds to the name of the input file), you will find the data sampled in a directory corresponding to the sampled time.
- For example, in this case you fill find the data in the directory **`postProcessing/probesDict/2000`**
- Then, inside this directory, you will find several files containing the sampled data.
- The name of the output file corresponds to the name of the sampled fields, in this case, *U* and *p*.
- Feel free to open the output files using your favorite text editor.

Sampling with the postProcess utility

The output files – **functionObject** type **sets** or **surfaces**

- 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
…
```

Sampling with the postProcess utility

The output files – **functionObject** type **sets** or **surfaces**

- 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
…
```

Sampling with the postProcess utility

The output files – **functionObject** type **sets** or **surfaces**

- 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
…
```

Sampling with the postProcess utility

The output files – **functionObject** type **probes**

- 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
        …
        …
        …
```

Sampling with the postProcess utility

The output files – **functionObject** type **probes**

- 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)
        …
        …
        …
```

Sampling with the postProcess utility

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?
- How many options are there available to do sampling in a line?
- Do point, line, and surface 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)

Roadmap

1. ~~On-the-fly postprocessing – functionObjects and the postProcess utility~~
2. ~~Sampling with the postProcess utility~~
3. **Field manipulation**
4. Data conversion

Field manipulation

- Hereafter we are going to deal with field manipulation
- Field manipulation means modifying a field variable or deriving a new field variable using the primitive variables computed during the solution stage.
- We will do the post-processing using the command line interface (CLI), or non-GUI mode.
- The utility `postProcess` can be used as a single application, *e.g.*,
    - `$> postProcess –func vorticity`
- Or it can be used with a solver using the option `–postprocess`, *e.g.*,
    - `$> simpleFoam -postprocess –func vorticity`
- Running the solver with the option `–postprocess` will only execute the post-processing and it will let you access data available on the database for the particular solver (such as physical properties or turbulence model).

Field manipulation

- To get a list of what can be computed using the `postProcess` utility, type in the terminal:
    - `$> postProcess -list`

- The utility `postProcess` can take many options. To get more information on how to use the utility, type in the terminal:
    - `$> postProcess -help`
    - `$> simpleFoam -postProcess -help`

- The options of the solver using the `-postProcess` flag are the same as the options of the utility `postProcess`.
- In the sub-directory **`$FOAM_UTILITIES/postProcessing/postProcess`** you will find the utility `postProcess`.
- In the directory **`$FOAM_SRC/functionObjects`**, you will find the source code of the objects that can be used to compute a new field.

Field manipulation

- We will now do some field manipulation using the cylinder case.
- For this we will use the supersonic wedge tutorial located in the directory:

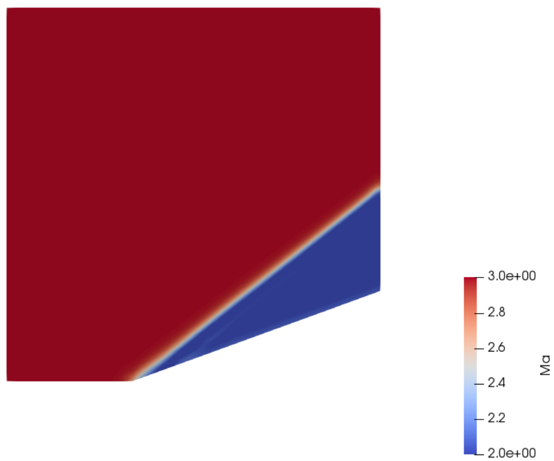
```
$PTOFC/101postprocessing/supersonic_wedge/
```

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
  - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, 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, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

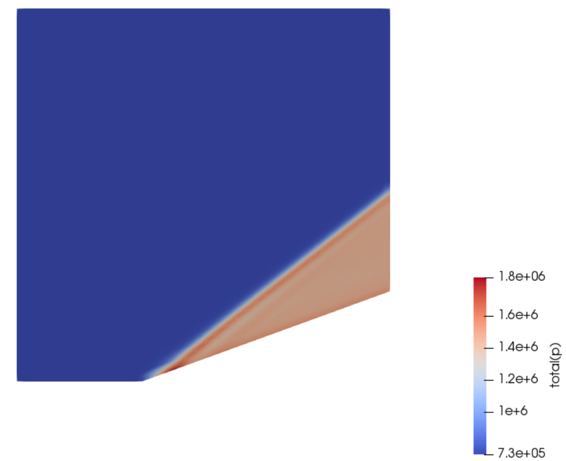
Field manipulation

After computing the solution, we can compute derived fields (*e.g.*, Mach number, density, Courant number, vorticity, and so on), using the primitive fields (U, p, T)

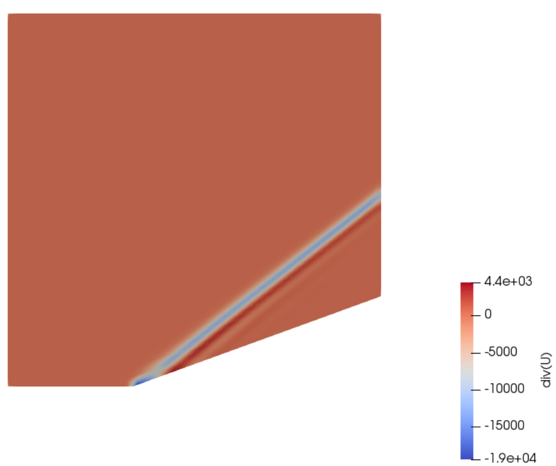
Mach number



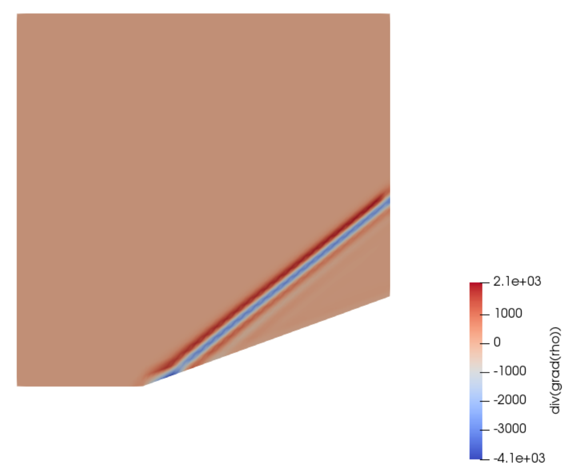
Total pressure



Divergence of U



Divergence of density gradient (numerical shadowgraph)



Field manipulation

What are we going to do?

- We will use this case to introduce the `postProcess` utility for field manipulation.
- We will also show how to run the solver with the option `-postProcess`. This will let us do only the post-processing after the solution has been computed, and it will let us access the database of the solver.
- To find the numerical solution we will use the solver `rhoPimpleFoam`.
- `rhoPimpleFoam` is a transient solver for laminar or turbulent flow of compressible gas.

Running the case

- Let us run this case using the automatic script, in the terminal type,

1. `$> sh run_solver.sh`
2. `$> paraFoam`

- Fell free to open the file *`run_solver.sh`* to know all the steps.

Field manipulation

- After finding the solution, we can compute the new field variables using the primitive variables computed during the solution stage. In the terminal type:

```
1. $> rhoPimpleFoam -postProcess -func MachNo
2. $> rhoPimpleFoam -postProcess -func CourantNo
3. $> rhoPimpleFoam -postProcess -func wallShearStress
4. $> rhoPimpleFoam -postProcess -func 'writeObjects(rho)' -time 0
5. $> rhoPimpleFoam -postProcess -func vorticity
6. $> postProcess -func vorticity
7. $> rhoPimpleFoam -postProcess -dict system/externalFunctionObject -latestTime
```

- If the new field variables require information of the simulation database (fluxes, turbulence properties, transport properties), you will need to process as in steps 1-5.
- If the new field variable only requires to use a variable that already exist in the solution folder, you can proceed as in step 6.

Field manipulation

- In step 1, we compute the Mach number.
  - To compute this value, the `postProcess` utility needs to access the `thermophysicalProperties` dictionary.
- In step 2, we compute the Courant number.
  - To compute this value, the `postProcess` utility needs to access the face fluxes (phi).
- In step 3, we compute the wall shear stress.
  - To compute this value, the `postProcess` utility needs to access the transport and turbulence properties.
- In step 4, we compute the density (**rho**) for the initial time (time = 0).
  - To compute this value, the `postProcess` utility needs to access the simulation database.
- In steps 5 and 6, we compute the vorticity field, this field is derived from the velocity field.
  - The `postProcess` utility does not need to access any particular solver information. Both options will give the same output.
- In step 7, we use an external file to compute the derived fields.
  - In this case we are computing the density gradient **grad(rho)** and the divergence of the density gradient **div(grad(rho))**.
  - Remember, in order to compute the derived field **div(grad(rho))**, you need to compute first **grad(rho)**.

Field manipulation

- After finding the solution, we can compute new field variables using the primitive variables computed during the solution stage. In the terminal type:

```
1. $> postProcess -func 'grad(U)'
2. $> postProcess -func 'components(U)'
3. $> postProcess -func 'mag(U)'
4. $> postProcess -func 'magSqr(U)'
5. $> postProcess -func 'totalPressureCompressible(rho,U,p)' -noZero
6. $> postProcess -func 'div(U)' -time 500:1000
7. $> postProcess -func 'mag(grad(U))' -latestTime
```

- We can also use the utility `postProcess` to compute the average and integral of a specified field over a patch. In the terminal type:

```
8. $> postProcess -func 'patchAverage(p,patch=inlet)' -latestTime
9. $> postProcess -func 'patchAverage(U,patch=outlet)' -latestTime
10. $> postProcess -func 'patchIntegrate(p,patch=inlet)' -latestTime
11. $> postProcess -func 'patchIntegrate(U,patch=outlet)' -latestTime
```

Field manipulation

- In steps 1-11, all the fields are derived from pre-existing fields.
    - The `postProcess` utility does not need to access any particular solver information.
- In step 1, we compute the gradient of the velocity vector **U**.
    - The field is saved as **grad(U)**.
- In step 2, we compute the components of the velocity vector **U**.
    - The components are saved as **Ux**, **Uy** and **Uz**.
- In step 3, we compute the magnitude of the velocity vector **U**.
    - The output is saved as **mag(U)**.
- In step 4, we compute the magnitude squared of the velocity vector **U**.
    - The output is saved as **magSqr(U)**.
- In step 5, we compute the total pressure.
    - The output is saved as **total(p)**. The option **–noZero** means do not compute the value for time zero.

Field manipulation

- In step 6, we compute the divergence of the velocity vector **U**.
  - The output is saved as **div(U)**. You will need to define how to interpolate **div(U)** in the *fvSchemes* dictionary. The option **–time 500:1000** means save the values between the given range (500-1000).
- In step 7, we compute the magnitude of the gradient of the velocity vector **U**.
  - The output is saved as **mag(Grad(U))**. The option **–latestTime** will compute the value only for the latest saved solution.
- In step 8, we compute the average of **p** over the patch **inlet**.
- In step 9, we compute the average of **U** over the patch **outlet.**
- In step 10, we compute the integral of **p** over the patch **inlet.**
- In step 11, we compute the integral of **U** over the patch **outlet**.

Roadmap

1. ~~On-the-fly postprocessing – functionObjects and the postProcess utility~~
2. ~~Sampling and probing with the postProcess utility~~
3. ~~Field manipulation~~
4. **Data conversion**

Data conversion

- OpenFOAM® gives users a lot of flexibility when it comes to scientific visualization.
- You are not obliged to use OpenFOAM® visualization tools (paraFoam or paraview).
- You can convert the solution obtained with OpenFOAM® to many third-party formats by using OpenFOAM® data conversion utilities.
- If you are looking for a specific format and it is not supported, you can write your own conversion tool.
- In the directory `$FOAM_UTILITIES/postProcessing/dataConversion`, you will find the source code of the following data conversion utilities:

  - **foamDataToFluent**
  - **foamToEnsight**
  - **foamToEnsightParts**
  - **foamToGMV**
  - **foamToTecplot360**
  - **foamToTetDualMesh**
  - **foamToVTK**
  - **smapToFoam**

- To get more information on how to use a data conversion utility, you can read the source code or type in the terminal:
  - `$> name_of_data_conversion_utility -help`

Data conversion

ASCII ↔ Binary conversion

```
application     icoFoam;
startFrom       startTime;
startTime       0;
stopAt          endTime;
endTime         50;
deltaT          0.01;
writeControl    runTime;
writeInterval   1;
purgeWrite      0;
writeFormat     binary;
writePrecision  8;
writeCompression off;
timeFormat      general;
timePrecision   6;
runTimeModifiable true;
```

- Another utility that might come in handy, specially when dealing with large meshes is `foamFormatConvert`.
- This utility converts the mesh and field variables into ascii or binary format.
- Working in binary format can significantly reduce data parsing and dimension of the files (specially for large meshes).
- The drawback is that the files are not human readable anymore.
- To convert ascii files into binary files, just type in the terminal:
    - `$> foamFormatConvert`
- Remember you will need to set the keyword **writeFormat** to binary in the `controlDict` dictionary.
- In the same way, if you want to convert from binary to ascii, set the keyword **writeFormat** to ascii in the `controlDict` dictionary and type in the terminal:
    - `$> foamFormatConvert`