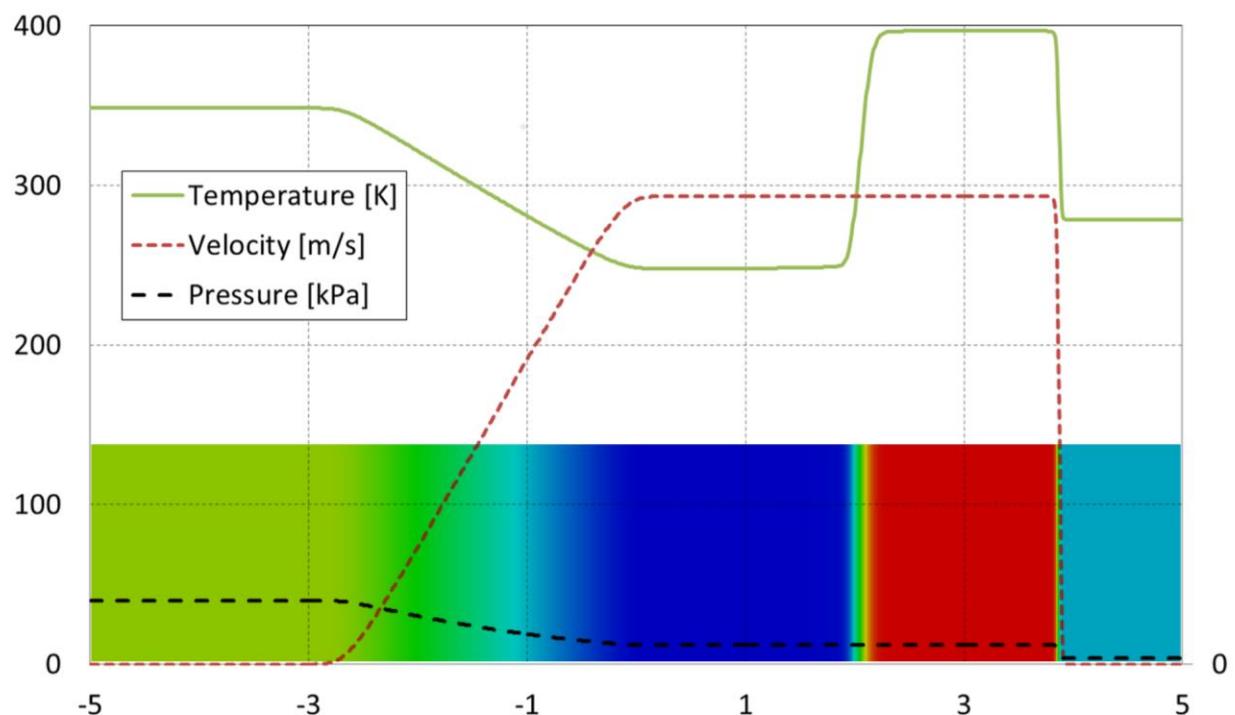


Tutorial Fourteen

Sampling



Bahram Haddadi



7th edition, March 2025

Except where otherwise noted, this work is licensed under
<http://creativecommons.org/licenses/by-nc-sa/3.0/>

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Benjamin Piribauer
- Yitong Chen



Technische Universität Wien
Institute of Chemical, Environmental
& Bioscience Engineering



Except where otherwise noted, this work is licensed under
<http://creativecommons.org/licenses/by-nc-sa/3.0/>

Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0)

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

- Attribution — you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that, they endorse you or your use of the work).
- Noncommercial — you may not use this work for commercial purposes.
- Share Alike — if you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

With the understanding that:

- Waiver — any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain — where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights — In no way are any of the following rights affected by the license:
- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- Notice — for any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

This offering is not approved or endorsed by ESI® Group, ESI-OpenCFD® or the OpenFOAM® Foundation, the producer of the OpenFOAM® software and owner of the OpenFOAM® trademark.

Available from: www.fluiddynamics.at

Background

1. Importance of Sampling in CFD Simulations

In computational fluid dynamics, it is crucial to analyze simulation results effectively to ensure accurate predictions and correct numerical behavior. Sampling provides:

- Enhanced Debugging: Identifying issues such as unexpected flow behaviors, numerical instability, or divergence at an early stage.
- Real-time Monitoring: Observing the evolution of flow variables without waiting for the simulation to finish.
- Efficient Data Management: Extracting only necessary data instead of storing large amounts of full-domain output, thereby reducing storage requirements.
- Post-processing Flexibility: Enabling in-depth analysis and visualization of selected flow regions using external tools such as ParaView .

2. Sampling in OpenFOAM®

This tutorial serves as a introduction to the sampling utility available in OpenFOAM®. The sampling utility is a powerful feature that allows users to extract data from specific surfaces or points within a simulation domain. This extracted data can then be analyzed to understand the behavior of the simulated flow, validate numerical results, or visualize specific regions of interest.

Sampling in OpenFOAM® can be performed in two primary ways:

- Post-processing sampling – Data is extracted after the simulation has completed.
- In-situ sampling – Data is collected during the simulation runtime, allowing for real-time monitoring and debugging.

By using the sampling utility, users can examine critical parameters such as velocity, pressure, turbulence properties, and other field variables at selected locations, which help in gaining insights into the correctness and stability of the numerical solution.

fluid – shockTube

Tutorial outline

Simulate the flow along a shock tube for 0.007 s and use OpenFOAM® sampling utility for extracting the data along a line during the simulation and after the simulation.

Objectives

- Understand the function of sampling and how to use the sampling utility

Data processing

Import your simulation to ParaView to visualize it and analyze the extracted data with sampling tool.

1. Pre-processing

1.1. Copying tutorial

To test the sampling feature, we will use the shockTube tutorial covered in Tutorial Three and extract data over a line between (-5 0 0) and (5 0 0).

```
$FOAM_TUTORIALS/compressible/fluid/shockTube
```

1.2. system directory

1.2.1. sample dictionary

The *sample* file can be found in the system directory.

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

type sets;
libs          ("libsampling.so")

interpolationScheme      cellPoint;

setFormat    raw;

sets
(
    data
    {
        type          lineUniform;
        axis          x;
        start         (-4.995 0 0);
        end           (4.995 0 0);
        nPoints       1000;
    }
);

fields          (T mag(U) p);

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

In the *type* the type of data to be sampled is defined, e.g. *sets* or *surfaces*. The different options for *interpolationScheme* and *setFormat* will be discussed in a later section.

In the *sets* sub-dictionary each set of data should be given a name, which is freely chosen by the user, in this case the name is simply '*data*'. In the bracket for the set of data, we need to specify the following criteria:

- *type*: specifies the method of sampling. Here uniform was chosen to make a sample on a line with equally distributed number of points.
- *axis*: to define how the point coordinates are written. In this case, *x* means that only the *x* coordinate for each point will be written.
- *Start/end*: the endpoints of the line-sample are defined
- *nPoints*: number of points on our line

Outside of the *data* and *sets* bracket in the *fields* we have to define which fields we want to sample.

1.2.2. controlDict

To have the option to sample for each time step instead of each write-interval or sampling while the solver is running; instead of the *sample* dictionary additions in the *functions* file (it can be also integrated into the *controlDict*) are needed.

In this part one will change the *functions* file of the shockTube tutorial so that our line- sampling from before will be executed while running, and per time step.

Modify the *functions* file as following:

```
// * * * * *
* * * * */
...
functions
{
    #includeFunc mag(U)

    linesample
    {
        type           sets;
        functionObjectLibs ("libsampling.so");
        writeControl   timeStep;
        outputInterval 1;

        interpolationScheme cellPoint;

        setFormat raw;

        sets
        (
            data
            {
                type      lineUniform;
                axis      x;
                start    (-4.995 0 0);
                end      (4.995 0 0);
                nPoints  1000;
            }
        );
        fields          (T mag(U) p);
    }
}
// * * * * */
* * * * */
```

`linesample` sub-dictionary includes the settings for the sampling tool. Any arbitrary name can be chosen instead of `linesample`. The chosen name will be the name of the folder in the *postProcessing* directory after running the solver.

Inside our `linesample` sub-dictionary:

- `type`: `sets` or `surfaces` can be chosen. More types will be covered in a later section.
- `functionObjectLibs`: provides the operations needed for the sampling tasks.
- `writeControl`: specifies the intervals in which sampling data should be collected in the case of `timeStep`, depending on the `outputInterval`, sampling data will get saved in dependence of the `timeStep`. In the case

of `outputInterval` being equal to 1, every time step, sampling data will be saved. Changing the interval to 2 means that data will be saved every 2 time steps.

2. Running simulation

To run the Tutorial go to your case directory in the terminal and use the following commands:

```
>blockMesh  

>setFields  

>foamRun -solver fluid
```

3. Post-processing

After fluid solver finishes running, based on your sampling approach the following steps should be performed:

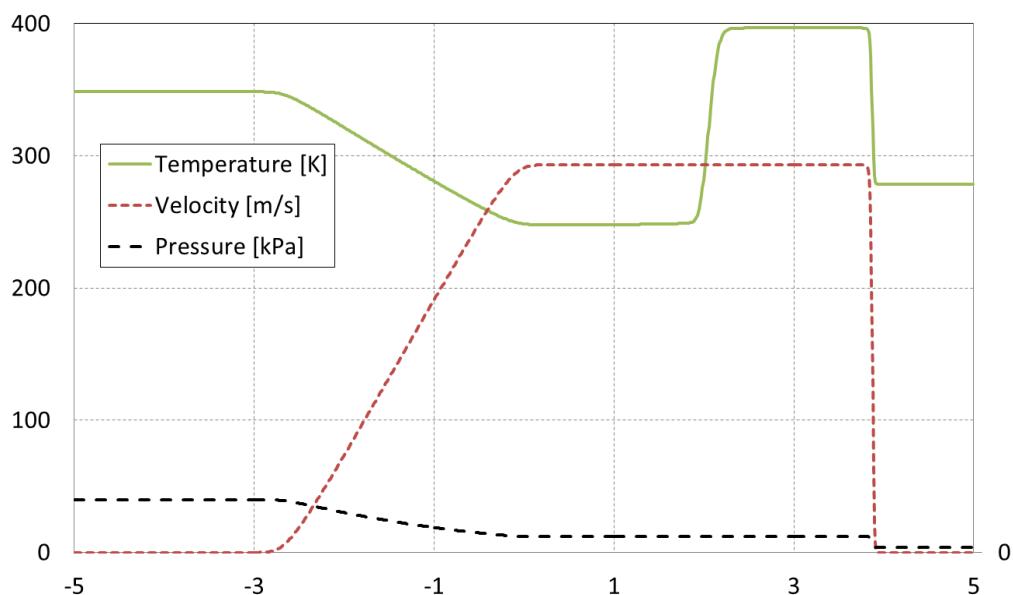
3.1. sample dictionary

It is also possible to use the `sample` command to extract your sample-data.

```
>postProcess -func sample
```

A new folder will appear in your case directory named `postProcessing` and in it a folder named `sample`. In this folder all the sampling data will be stored in separate folders for each write-interval.

The `postProcessing` directory and all its subdirectories have been generated after the first time step. Now it can be seen that for every time step a folder is generated instead of only every write interval.



Extracted data using sampling tool after 0.007 s

4. Types of sampling

There are 2 main types of sampling. The sets type, which was used in our example above, and the surfaces type.

In the sets type of sampling different kinds of point samplings, like the line sampling we used before, or some kind of cloud sampling are included. In the surface type whole surfaces are sampled, like near a patch, or on a plane defined by a point and a normal vector.

Let us discuss the similarities between the set and surface types. If the sampling happens in the *controlDict* the 4 entries discussed in the *controlDict* section of this tutorial need to be included for both types. On top of that, both types need an interpolation scheme. Here only two of the schemes: cell and cellPoint will be discussed. The cell scheme assumes that the cell centre value as constant in the whole cell. The cellPoint scheme will carry out linear interpolation between the cell centre and vertex values. Lastly, the field bracket looks the same for both cases.

4.1. sets

All sets need a `setFormat`, for example `csv`, which needs to be included after the `interpolationScheme`.

After that the sets sub-dictionary begins where a bracket with an arbitrary name begins in which the sets sampling type and the geometrical location of the sampling points will be chosen. In the following, a few of sampling types will be discussed.

4.1.1. lineUniform

This one was used in the above example. A line from a start point to an end point with a fixed number of points evenly distributed along it.

`axis` determines what is written for the point coordinate in the output file. `distance` means it will only write the distance between sampled point and start point in the file.

```
lineX1
{
    type          lineUniform;
    axis          distance;
    start         (0.0201  0.05101  0.00501);
    end           (0.0601  0.05101  0.00501);
    nPoints       10;
}
```

4.1.2. face

This type also samples along a line from a defined start to endpoint, but only writes in the log file for every face the line cuts.

```
lineX2
{
    type          face;
    axis          x;
    start         (0.0001  0.0525  0.00501);
    end           (0.0999  0.0525  0.00501);
```

}

4.1.3. cloud

The `cloud` type samples data at specific points defined in the point's bracket.

```
somePoints
{
    type          cloud;
    axis          xyz;
    points        ((0.049 0.049 0.00501) (0.051 0.049 0.00501));
}
```

4.1.4. patchSeed

The `patchSeed` sampling type is used for sampling patches of the type `wall`.

One can for example sample the surface adsorption on a wall with this type.

```
patchSeed
{
    type      patchSeed;
    axis      xyz;
    patches   (".*Wall.*");
    maxPoints 100;
}
```

Please note that for patches only a patch of type `wall` can be used. If you choose a wrong type, nothing will be sampled and you receive no error message.

4.2. surfaces

All surfaces need a `surfaceFormat` specified. Practical formats would be the `vtk` format, which can be opened with paraview, and the `raw` format, which can be used for gnuplots. Instead of the `sets` bracket now a `surfaces` bracket must be used and the `type` is of course `surfaces`. Inside the `surfaces` brackets one can now sample an arbitrary number of surfaces, each in its own inner brackets. The different types of surface sampling like the `plane` in the example below will be discussed in the next sections.

```
type           surfaces;
interpolationScheme  cellPoint;
surfaceFormat     vtk;

fields
(
    U
);

surfaces
(
    yoursurfacename
    {
        type      plane;
        basePoint   (0.1 0.1 0.1);
        normalVector (0.1 0 0);
        triangulate false;
    }
);
```

4.2.1. plane

The type `plane` creates a flat plane with an origin in the `basePoint` normal to the `normalVector`. This `normalvector` starts in the origin, not in the

basePoint. To turn the triangulation of the surface off one has to add triangulate false.

```
constantPlane
{
    type          plane;      // always triangulated
    basePoint     (0.0501 0.0501 0.005);
    normalVector  (0.1 0.1 1);

    // Optional: restrict to a particular zone
    // zone        zone1;

    // Optional: do not triangulate (only for surfaceFormats that support
    // polygons)
    //triangulate   false;
    //interpolate   true;
}
```

One can also set a new origin for the basePoint and normalVector with

```
coordinateSystem
{
    origin     (0.0501 0.0501 0.005);
}
```

4.2.2. patch

A sampling of type `patch` can sample data on a whole patch. Please note that while the syntax looks the same as in the `patchSeed` case, also patches that are not of type `wall` work. Default option will triangulate the surface; this can be turned off with `triangulate false`.

```
walls_interpolated
{
    type          patch;
    patches       ( ".*Wall.*" );
    //interpolate   true;
    // Optional: whether to leave as faces (=default) or triangulate
    // triangulate   false;
}
```

4.2.3. patchInternalField

Similar to the `patch` type, this type needs a patch on which it samples. It will sample data that's a certain distance away in normal direction (`offsetMode normal`). There is also the option to define the distance in other ways seen in the commented section of the code.

Note: While the sampling happens not on the patch but a distance away from it, the geometric position of the sampled values in the output file will be written as the position of the patch.

Once again the default triangulation can be turned off with `triangulate false`.

```
nearWalls_interpolated
{
    // Sample cell values off patch.
    // Does not need to be the near-wall
    // cell, can be arbitrarily far away.
    type          patchInternalField;
    patches       ( ".*Wall.*" );
    interpolate   true;

    // Optional: specify how to obtain
    // sampling points from the patch
```

```

// face centres (default is 'normal')
//
// --- Specify distance to
// offset in normal direction
offsetMode    normal;
distance      0.1;
//
// --- Specify single uniform offset
// offsetMode   uniform;
// offset       (0 0 0.0001);
//
// --- Specify offset per patch face
// offsetMode  nonuniform;
// offsets     ((0 0 0.0001) (0 0 0.0002));

// Optional: whether to leave
// as faces (=default) or triangulate
// triangulate   false;
}

```

4.2.4. triSurfaceSampling

With the `triSurfaceSampling` type data can be sampled in planes which are provided as a trisurface stl file. To create such a file one can use the command below. The command will generate a .stl file of one (or more) of your patches.

```
>surfaceMeshTriangulate name.stl -patches "(yourpatch)"
```

Here your patch needs to be replaced with the name of one of your patches defined in the constant/polyMesh/boundary file. Starting the command without the patches option will generate a stl file of your whole mesh boundary. Next make a directory in the constant folder named `triSurface` if it does not already exist and copy the .stl file there. In the code, you now have to specify your stl file as the surface. For the source, the use of `boundaryFaces` seems to be a good option of the stl file is one of your patches.

```

triSurfaceSampling
{
    // Sampling on triSurface
    type      sampledTriSurfaceMesh;
    surface   integrationPlane.stl;
    source    boundaryFaces;
    // What to sample: cells (nearest cell)
    // insideCells (only triangles inside cell)
    // boundaryFaces (nearest boundary face)
    interpolate true;
}

```

Note: Most CAD software can export the surface of 3D drawings as stl files.

4.2.5. isoSurface

The `isoSurface` sampling type is quite different to what was discussed before in this tutorial. Until now, all the sampling types had a constant position in space and changing field values at that position were extracted. With the `isoSurface` sampling, one tracks the position of a defined value in space. The example below can be copied into the shocktube tutorials `sample file` (of course, it needs all the other options needed for surface type sampling).

Using `vtk` for the `surfaceFormat` one can visualize the moving shockwave in space. Note that both the `vtk` of the sampling and the whole shocktube case can be opened together in paraview to compare the results.

Note that the `isoField` needs to be a scalarfield.

```
interpolatedIso
{
    // Iso surface for interpolated values only
    type           isoSurface;
    // always triangulated
    isoField       p;
    isoValue       9e4;
    interpolate    true;

    //zone          ABC;
    // Optional: zone only
    //exposedPatchName fixedWalls;
    // Optional: zone only

    // regularise    false;
    // Optional: do not simplify

    // mergeTol      1e-10;
    // Optional: fraction of mesh bounding box
    // to merge points (default=1e-6)
}
```

4.2.6. isoSurfaceCell

The `isoSurfaceCell` type is very similar to the one we discussed before, but this one does not cross any cell with its surface and does not interpolate values.

```
constantIso
{
    // Iso surface for constant values.
    // Triangles guaranteed not to cross cells.
    type           isoSurfaceCell;
    // always triangulated
    isoField       rho;
    isoValue       0.5;
    interpolate    false;
    regularise    false;
    // do not simplify
    // mergeTol      1e-10;
    // Optional: fraction of mesh bounding box
    // to merge points (default=1e-6)
}
```