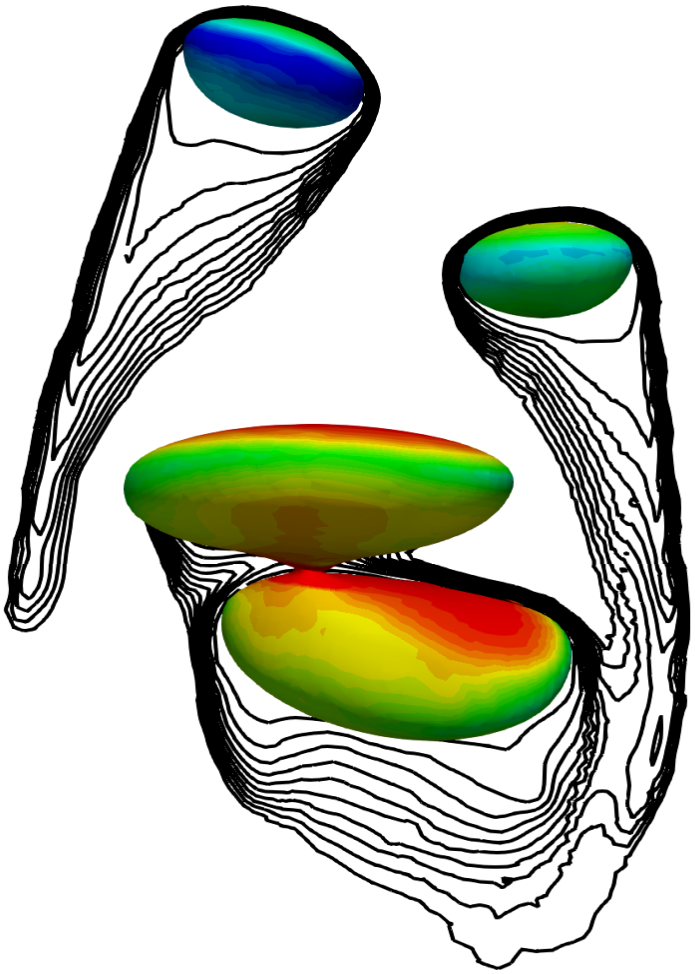
OpenFOAM® Basic Training



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Editor:

- Bahram Haddadi

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien
Institute of Chemical, Environmental & Bioscience Engineering

Compatibility:

- OpenFOAM® v12

Available from: www.fluiddynamics.at

This tutorial series introduces OpenFOAM®, a widely used open-source software for Computational Fluid Dynamics (CFD). Fourteen case examples help the users to learn essential OpenFOAM® tools and functions. These examples cover different aspects such as mesh generation, multiphase modeling, turbulence modeling, parallel processing and reaction modeling.

Where to Find Tutorial Cases?
The base tutorial cases can be accessed directly from OpenFOAM® installation directory or will be compiled in step-by-step approach.

Compatibility with OpenFOAM® Versions
These tutorials are designed primarily for OpenFOAM® v12 (Foundation version – www.openfoam.org). However, they can also be adapted for other OpenFOAM® versions, such as:

- ESI-OpenFOAM® (maintained by OpenCFD)
- Foam-extend (a community-driven fork with additional features)

Tutorial Structure
Each case example follows a structured learning approach:

0. **Background**: overview of the key concepts covered in the tutorial and the CFD principles related to the case.
1. **Pre-processing**: step-by-step setup of the case, including directory structure, essential input files (dictionaries), and necessary modifications.
2. **Running the simulation**: instructions on executing the solver, running necessary commands, and monitoring the progress of the simulation.
3. **Post-processing**: analyzing the simulation results using OpenFOAM®'s built-in tools and the visualization software ParaView v5.x.

By following these tutorials, users will gain hands-on experience in setting up, running, and analyzing CFD simulations in OpenFOAM®.

Tutorial One: **Basic Case Setup**

Solver: icoFoam
Geometry: 2-dimensional
Tutorial: elbow

Tutorial Two: **Built in Mesh**

Solver: fluid
Geometry: 2-dimensional
Tutorial: forwardStep

Tutorial Three: **Patching Fields**

Solver: fluid
Geometry: 1-dimensional
Tutorial: shockTube

Tutorial Four: **Discretization – Part 1**

Solver: functions
Geometry: 1-dimensional
Tutorial: shockTube

Tutorial Five: **Discretization – Part 2**

Solver: functions
Geometry: 2-dimensional
Tutorial: circle

Tutorial Six: **Turbulence, Steady state**

Solver: incompressibleFluid
Geometry: 2-dimensional
Tutorial: pitzDaily

Tutorial Seven: **Turbulence, Transient**

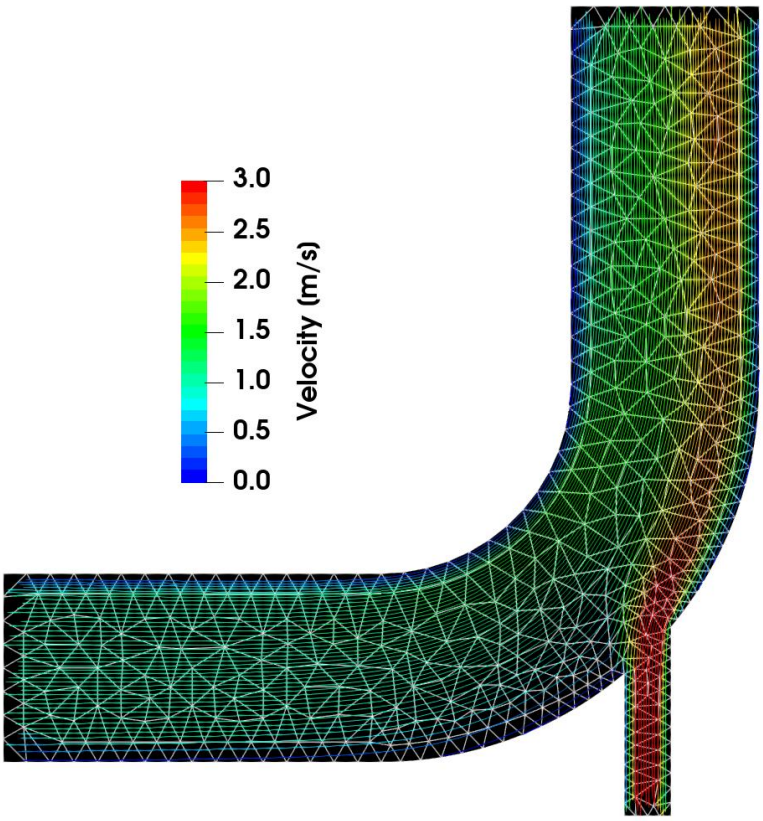
Solver: incompressibleFluid
Geometry: 2-dimensional
Tutorial: pitzDaily

Tutorial Eight: **Multiphase - VoF**

Solver: incompressibleVoF
Geometry: 2-dimensional
Tutorial: damBreak

Tutorial One

Basic Case Setup



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. What is CFD?

Computational Fluid Dynamics (CFD) is a method used to analyze systems involving fluid flow, heat transfer, and related phenomena such as heat and mass transfer. This analysis is performed through computer-based simulations, which help in understanding how fluids behave under different conditions. CFD is a powerful tool used in a variety of fields, including aerospace, automotive, chemical engineering, environmental studies, and biomedical applications.

The goal of CFD development is to create tools that are as reliable as other computer-aided engineering (CAE) methods like stress analysis. However, CFD is trickier due to the complex nature of fluid flow, which involves turbulence, variable properties, and nonlinear behavior. The mathematical foundation of CFD is based on the Navier-Stokes and continuity equations, which describe the motion of fluid substances and are derived from fundamental conservation laws (mass and momentum). These equations are partial differential equations that represent how fluid velocity, pressure, and density change over time and space.

While CFD offers many advantages, such as cost reduction in experimental setups and the ability to simulate complex scenarios, it is not fully automated. A good understanding of the underlying physics is necessary to set up a reliable simulation and interpret results correctly. Additionally, even with advanced computational resources, real-time simulations are still challenging due to the intensive calculations required. CFD is typically used alongside experimental methods like wind tunnel testing to validate and improve results.

CFD software comes in two main types:

- **Open-source and free (e.g., OpenFOAM®):** Offers flexibility for modification and customization, making it popular in academic and research environments.
- **Commercial and closed source (e.g., ANSYS Fluent, COMSOL):** Provides user-friendly interfaces, technical support, and advanced features, making it suitable for industrial applications.

In this guide, the focus will be on OpenFOAM®, an open-source CFD software written in C++. It allows users to access, modify, and even develop custom solvers to meet specific research or industrial needs. OpenFOAM® is widely used due to its flexibility and extensive documentation, although it requires a good understanding of both CFD principles and programming basics.

For beginners, it's helpful to think of CFD as a "virtual wind tunnel" where you can simulate fluid flow without physically building models or conducting real-world experiments. This makes it a cost-effective and versatile tool, especially during the design and testing phases of engineering projects.

CFD is not only limited to air and water flow simulations, and it is extensively used in modeling different sophisticated processes such as: weather patterns (meteorology), blood flow in arteries (biomedical engineering), combustion

processes in engines (mechanical engineering), pollution dispersion in the atmosphere (environmental engineering) and many more!

2. Workflow of CFD

A typical CFD workflow consists of three main stages:

2.1 Pre-processing

This stage involves setting up the simulation, including:

- **Geometry Definition:** Creating the computational domain that represents the physical system. This can be done using CAD (Computer-Aided Design) software or built-in geometry tools in CFD software. The accuracy of geometry affects how well simulation represents the real-world scenario. Think of geometry as the "shape" or "structure" through which the fluid will flow. Beginners can start with simple geometries like pipes, ducts, or channels before progressing to complex designs.
- **Mesh Generation:** Dividing the domain into smaller, non-overlapping elements (cells) to form a grid. The quality and density of the mesh significantly affect the accuracy of the simulation. Finer meshes are used in regions with high gradients (e.g., near walls, sharp edges, or around obstacles), while coarser meshes suffice for uniform flow areas. Meshes can be structured (regular grids) or unstructured (irregular shapes), depending on the complexity of the geometry. A structured mesh is easier to generate and solve but less flexible for complex geometries, while an unstructured mesh can fit intricate shapes better. Imagine the mesh as a net spread over your geometry. The tighter the net (finer mesh), the more detailed your simulation results will be. However, this also increases computational effort, so there's a balance to be found.
- **Model Selection:** Choosing physical models to represent phenomena such as turbulence (e.g., k-epsilon, k-omega models), heat transfer, and chemical reactions. The selection depends on the flow regime (laminar or turbulent) and the specific application.
- **Fluid Properties:** Defining parameters like density, viscosity, thermal conductivity, and specific heat capacity. These properties vary with temperature, pressure, or composition in complex simulations. Incompressible flow assumes constant density, while compressible flow accounts for changes in density due to pressure and temperature variations.
- **Boundary and Initial Conditions:** Setting conditions at the domain's boundaries (e.g., velocity at an inlet, pressure at an outlet, wall conditions) and initial conditions for transient simulations. Proper boundary conditions are crucial for accurate results.

The solution variables (e.g., velocity, pressure, temperature) are calculated at specific points within each cell. The mesh's resolution influences the

simulation's accuracy and computational cost. A mesh independence study is often performed to ensure that the results are not sensitive to the mesh size. This involves running simulations with progressively finer meshes until the changes in results become negligible.

Tip for beginners: Start with a coarser mesh to get quick results, then gradually refine the mesh to see how it affects accuracy. This helps you learn how sensitive your simulation is to mesh density.

2.2 Solver

In this stage, numerical methods are applied to solve the governing equations of fluid flow, including:

- **Conservation Equations:** Mass, momentum, and energy conservation laws are integrated over each control volume. These equations are often coupled, meaning changes in one variable affect others. For example, changes in velocity can influence pressure and vice versa**.**
- **Discretization:** The continuous equations are converted into algebraic forms using methods like the finite volume method (FVM), which ensures conservation principles are maintained within each cell. Other methods include the finite difference method (FDM) and finite element method (FEM), though FVM is most common in CFD. Discretization involves approximating derivatives with algebraic expressions, allowing the equations to be solved numerically.
- **Solution Techniques:** The resulting algebraic equations are solved iteratively until convergence is achieved. Common iterative solvers include the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) and PISO (Pressure-Implicit with Splitting of Operators) algorithms. Convergence is determined when changes in the solution between iterations fall below a predefined threshold.

The finite volume method is widely used because it ensures the conservation of physical quantities within each control volume, making it both accurate and robust. It is also flexible for handling complex geometries and boundary conditions.

Think of the solver as the "engine" of CFD—it's where all the heavy lifting happens to calculate how the fluid moves. Understanding how the solver works helps in troubleshooting and optimizing simulations.

2.3 Post-processing

This stage involves analyzing and visualizing the simulation results. Key tasks include:

- **Visualization:** Using cutting planes, contour plots, vector fields, streamlines, and line plots to represent flow variables such as velocity, pressure, and temperature distributions. Visualization helps in identifying flow patterns, vortices, and areas of interest like high-pressure zones.

- **Data Analysis:** Evaluating physical quantities like forces (drag, lift), heat transfer rates, pressure drops, and flow rates. Quantitative analysis helps validate the simulation results against experimental data or theoretical predictions.
- **Validation:** Comparing simulation results with experimental data or theoretical models to ensure accuracy. Sensitivity analysis may be conducted to understand the influence of different parameters.

Popular post-processing tools include commercial software like TecPlot and Ensight, as well as open-source tools such as ParaView and SALOME. These tools allow for advanced visualization techniques, including 3D rendering and time-dependent animations, making it easier to interpret complex flow behaviors.

Tip: Post-processing is not just about making pretty pictures. It helps you understand the flow physics and detect any errors or inconsistencies in your simulation.

3. icoFoam Solver

icoFoam is an OpenFOAM® solver suitable for analyzing incompressible, laminar flow of Newtonian fluids. It is based on the PISO algorithm (pressure-implicit split-operator), which is essentially a pressure-velocity iterative procedure for transient problems. In each iterative step, PISO solves the momentum equation using one predictor step, with two further corrector steps for both velocity and pressure.

icoFoam – elbow

Tutorial outline

Using icoFoam solver, simulate 75 s of flow in an elbow for the following GAMBIT® meshes:

- Tri-mesh (comes with OpenFOAM® tutorial)
- Hex-mesh coarse (check GAMBIT® “elbow 2D” tutorial)
- 2 times finer hex-mesh (refined previous step mesh)

Objectives

- Basic case setup in OpenFOAM®
- Setting up initial values of p and U
- Ensuring proper boundary definitions (imported boundaries from GAMBIT®, additional surfaces during conversion and boundaries definition in OpenFOAM®)

Data processing

Import your simulation to ParaView, extract data to make two diagrams (using spreadsheet calculators) of pressure and velocity magnitude along a line between two tubes, do the same for all three simulations.

1. Pre-processing

1.1. Setting system environment

Make sure your system environment is set correctly under the chosen version of OpenFOAM® (v12), check Appendix B Part A.

1.2. Copying tutorial

Open a terminal and copy the elbow tutorial from the following path to your working directory (see Appendix A for running a terminal in Linux):

```
$FOAM_TUTORIALS/legacy/incompressible/icoFoam/elbow
```

Note: The '$FOAM_TUTORIALS' allows the tutorial to be extracted from the tutorial folder in the installation directory of OpenFOAM® under the current system environment.

Note: The tutorial can also be simply copied from the mentioned directory using your file explorer.

1.3. Converting mesh

The mesh, which is produced by GAMBIT®, is not directly compatible with OpenFOAM®. First, the mesh needs to be converted to an OpenFOAM® mesh, using the following tool:

```
>fluentMeshToFoam elbow.msh
```

Note: the '>' sign is not part of the command. It is only used to show that the command should be typed inside a terminal.

If the mesh was created in mm and is converted using the mentioned command it will convert the mesh with wrong dimensions, since all the units in OpenFOAM® are SI Units (International System of Units).

There are different flags included with most of OpenFOAM® tools, for checking them use the flag `-help` after the command, e.g.:

```
>fluentMeshToFoam -help
```

The output gives an overview of available options of the tool and a short description on how to use it:

```
Usage: fluentMeshToFoam [OPTIONS] <Fluent mesh file>
options:
  -2D <thickness>   use when converting a 2-D mesh (applied before scale)
  -case <dir>       specify alternate case directory, default is the cwd
  -fileHandler <handler>
                    override the fileHandler
  -libs <(lib1 .. libN)>
                    pre-load libraries
  -noFunctionObjects
                    do not execute functionObjects
  -scale <factor>   geometry scaling factor - default is 1
  -writeSets        write cell zones and patches as sets
  -writeZones       write cell zones as zones
```

```
  -srcDoc           display source code in browser
  -doc              display application documentation in browser
  -help             print the usage
Using: OpenFOAM-10 (see https://openfoam.org)
Build: 10
```

The `-scale` flag is used for converting the mesh dimensions from other units to SI units, e.g. if the mesh was created in mm it will be converted to meter by using `-scale 0.001` (which is not the case in this tutorial):

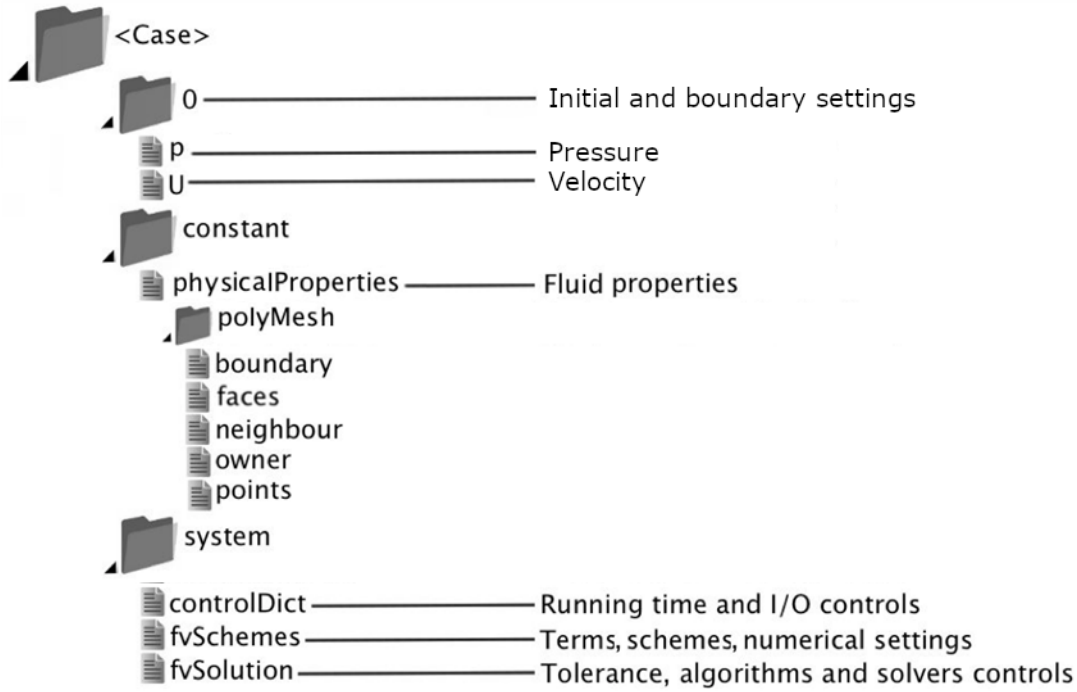
```
>fluentMeshToFoam elbow.msh -scale 0.001
```

Note: The mesh which is imported to OpenFOAM® should be a three-dimensional mesh. For carrying out 2D (also 1D) simulations, a three-dimensional mesh should be created with just one cell in the third dimension (for 1D, one cell in the second and one cell in the third direction).

Note: If there are internal boundaries in the mesh, there is another tool, fluent3DMeshToFoam. Using this tool, the internal boundaries will be kept during conversion.

1.4. Case structure

Most of the cases in OpenFOAM® have the following basic case structure (directory tree):



There are three main directories (0, constant, system) in each case folder:

1.4.1. 0 directory

The 0 directory includes the initial and boundary conditions for running the simulation. In each file in this folder, the initial conditions for one property can be set. The files are named after the property they are standing for, e.g. usually

p file includes pressure initial and boundary conditions. In the elbow example, there are only two files inside the 0 directory, p and U. p stands for pressure and U stands for velocity. Checking p:

```
>nano  p
```

Note: nano is the command line based text editor, which comes by default with Ubuntu. You can use any other text editor (also graphical ones) for opening and editing the files.

Note: You can use ctrl+x for closing and exiting the nano.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Website:  https://openfoam.org                  |
|   \\  /    A nd           | Version:  12                                    |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    wall-4
    {
        type            zeroGradient;
    }

    velocity-inlet-5
    {
        type            zeroGradient;
    }

    velocity-inlet-6
    {
        type            zeroGradient;
    }

    pressure-outlet-7
    {
        type            fixedValue;
        value           uniform 0;
    }

    wall-8
    {
        type            zeroGradient;
    }

    frontAndBackPlanes
    {
        type            empty;
    }
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

In `dimensions`, the physical dimension according to SI base units of the quantity is defined, for example here it shows that the p dimension is $(m/s)^2$.

Note: In the dimension matrix the first number represents mass (kilogram), the second one the length (meter), the third one time (second), the fourth one the temperature (Kelvin), the fifth one the quantity (mole), the sixth one current (ampere) and the last one luminous intensity (candela).

Note: As you can see the p unit is not the pressure unit (Pa). It is because in incompressible solvers in OpenFOAM® p is defined as pressure divided by density.

The `internalField` sets the initial field of a specific quantity in the solution domain. There are two types: uniform and non-uniform. Uniform field assigns a single value to all cells, whereas non-uniform field specifies a unique value to each field element.

The type of each of our boundaries as well as the value of this quantity on the boundaries is defined in the `boundaryField`. There are many different types of boundary conditions in OpenFOAM®, a few very common ones:

- `zeroGradient`: Applies a zero gradient boundary type to this boundary (Neumann boundary condition).
- `fixedValue`: Applies a fixed value to this boundary (Dirichlet boundary condition).
- `empty`: It is for sides, which are vertical to the direction that is not going to be considered (e.g. in 2D simulations these boundaries are vertical to the third dimension). In this boundary type both sides vertical to one dimension should be selected together and named as one boundary.

Note: If a `fixedValue` boundary condition with `value` equals `$internalField` is used, it is equal to using `zeroGradient`, except `zeroGradient` applies the boundary condition implicitly, but `fixedValue` with `$internalField` value applies the boundary condition explicitly.

The U file has to be defined via three components (since velocity is a vector): first one stands for the x component, second one for the y component, and the third one for the z component of the velocity. For this case setup the z component is always zero because it is a 2D simulation and no calculations will be done in the z direction. The boundaries vertical to z direction have been already set to empty.

1.4.2. constant directory

The constant directory usually consists of the mesh subdirectory and some files. In the sub-directory “polyMesh” the mesh data are stored (in this case the data for imported mesh). Among the files in the polyMesh directory, the boundary file is relevant for users and includes the mesh boundary data, e.g. name, type and the patch group which can be modified by the user for changing the boundary type or name for a created or imported mesh (for the sake of space, the dictionary headers will not be included in this scope anymore):

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

6
(
    wall-4
    {
        type            wall;
        inGroups        List<word> 1(wall)
        nFaces          100;
        startFace       1300;
    }
    velocity-inlet-5
    {
        type            patch;
        nFaces          8;
        startFace       1400;
    }
…
    frontAndBackPlanes
    {
        type            empty;
        inGroups        List<word> 1(empty);
        nFaces          1836;
        startFace       1454;
    }
)

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

Comparing the boundary names and types with the ones set in GAMBIT®, they should be the same.

Note: *However, in terms of boundary type, empty boundary condition does not exist in GAMBIT®. All the faces perpendicular to the direction, which is not going to be considered, are defined as a new boundary with type wall. After importing the mesh to OpenFOAM®, modify that boundary in the file constant/polyMesh/boundary, and change its type from `wall` to `empty`, and change `inGroups` from `wall` to `empty`. In this case, after converting the mesh, the face `frontAndBackPlanes` needs to be modified for both hex-mesh and finer hex-mesh.*

The files in the constant directory (usually) include material properties, simulation physics and chemistry, e.g. by opening the physicalProperties file, properties dimensions and the property value can be found and edited, e.g.:

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

`nu` is the fluid kinematic viscosity, which is 0.01 m^2/s for this example.

1.4.3. system directory

Solver and finite volume methods settings can be found and changed in this directory. There are three main files in this directory:

- **fvSchemes**: The discretization scheme used for each term of the equations are set in this file (it will be discussed in more detail in the next tutorials).
- **fvSolution**: Contains the settings to the coupling method of pressure and velocity, the numerical methods, which are used for solving different quantities, and the final tolerance for convergence of that quantity.

- **controlDict**: The time from where simulation starts (`startFrom`), the time when the simulation finishes (`stopAt`), the time step (`deltaT`), the data saving interval (`writeInterval`), the saved data file format (`writeFormat`), the saved file data precision (`writePrecision`), and also if changing the files during the run can affect the run or not (`runTimeModifiable`) are set in this file.

Note: If the write format is `ascii`, then the simulation data which is written to the file can be opened and read using any text editor. If the format is `binary`, the data will be written in binary style and is not readable by text editors. The advantage of binary over ascii is the smaller file size, and consequently faster conversion and writing to disk, for big simulations.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
* * * * * *//
application      icoFoam;

startFrom        latestTime;

startTime        0;

stopAt           endTime;

endTime          75;

deltaT           0.05;

writeControl     timeStep;

writeInterval    20;

purgeWrite       0;

writeFormat      ascii;

writePrecision   6;

writeCompression off;

timeFormat       general;

timePrecision    6;

runTimeModifiable true;

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

Note: This simulation continues from the last time step data, which is saved (`latestTime`). If there was no saved data, it will start from start time (`startTime`), which is zero in this case.

Note: Our first modification in the simulation is changing the `endTime` from the original value of 10s to 75s, for running the simulation up to 75s.

2. Running simulation

The simulation can be run by typing the solver's name and executing it:

```
>icoFoam
```

Note: For running the simulation, the solver command (e.g. icoFoam) should be executed inside the copy of the tutorial main folder. For example: The command should be executed in the elbow folder, if it was run at some subfolders or somewhere else, the simulation will fail.

3. Post-processing

3.1. Exporting simulation data

The data files created by OpenFOAM® should be exported (converted) by the appropriate tools, to the post processing tools data format. For ParaView:

```
>foamToVTK
```

where VTK is the ParaView data format. This command should be also executed in the case main directory, e.g. elbow. Here, ParaView is used as the post-processing tool, for running it

```
>paraview &
```

Note: Another option to open the OpenFOAM® simulation results with ParaView without converting them to VTK; Create an empty text file in the main case directory, name it <someName>.foam (e.g. foam.foam), and execute the following command. This method is good for fast evaluation of the data in the middle of the simulation or with a decomposed case in parallel simulations:

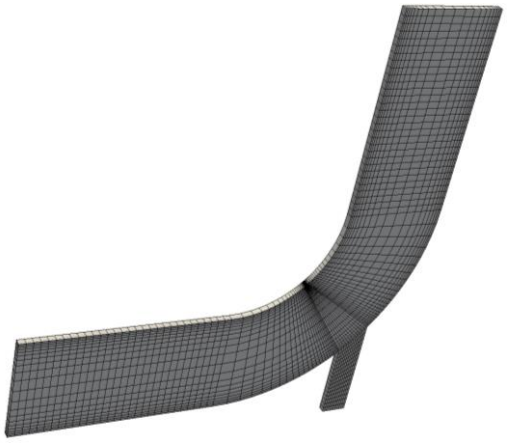
```
>paraview foam.foam &
```

Note: By putting & at the end of command, the command line will remain active and ready for further inputs while that program is running.

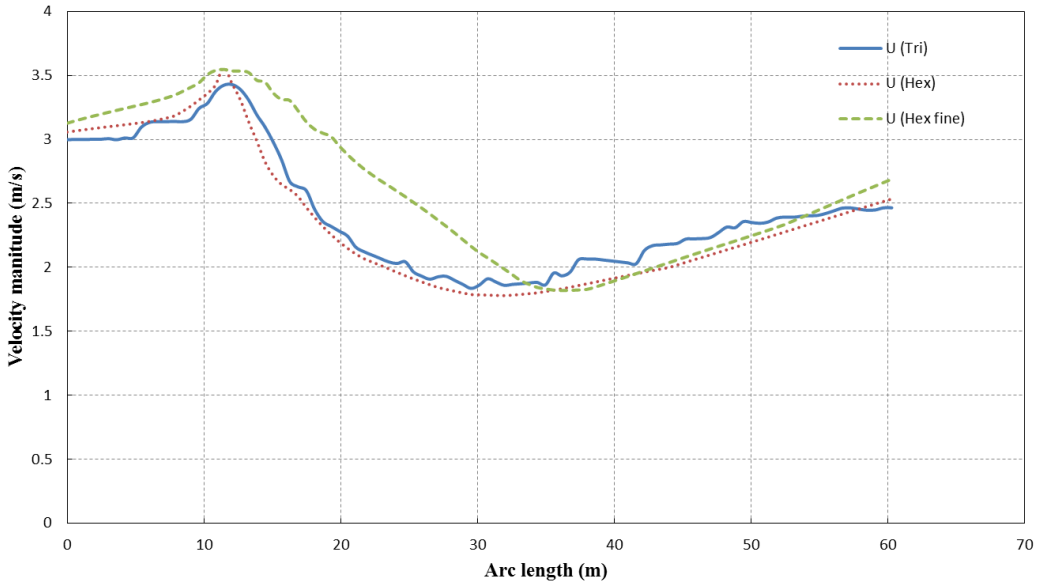
3.2. Examining different meshes

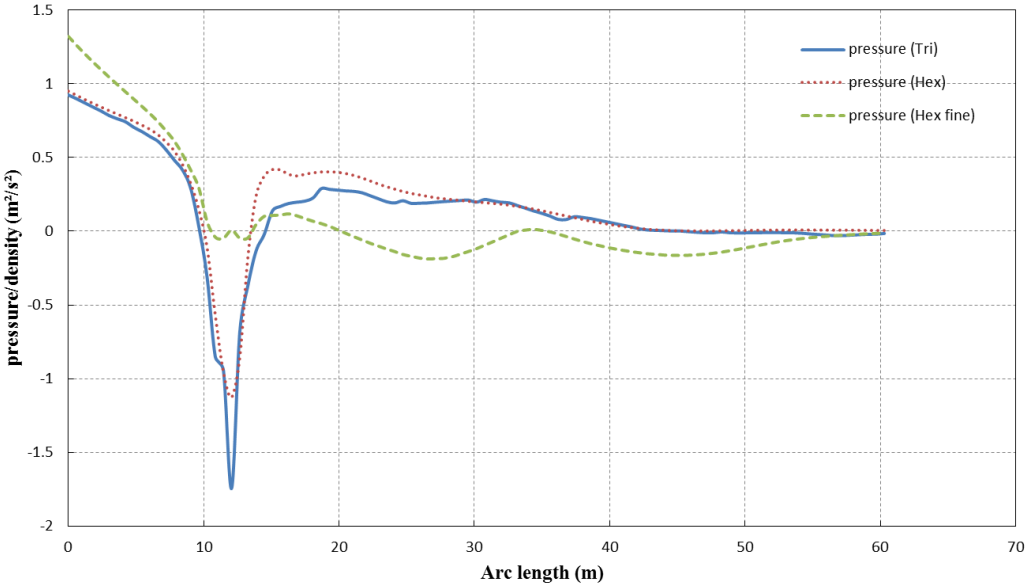
Do the same for the other two meshes. Only the mesh for the first simulation is included in the elbow example of OpenFOAM®. For the other two simulations, the mesh should be provided by the user. For finding the tutorials on how to create the geometry and the mesh, search the internet for "GAMBIT® elbow mesh 2D". The dimensions and the mesh info are provided in that tutorial. Try to create it by using GAMBIT® (or any other similar mesh creation tools). When you are done, you have to convert it into a 3D mesh with one cell in the z-direction.

The comparisons of all three case results and charts are shown below.



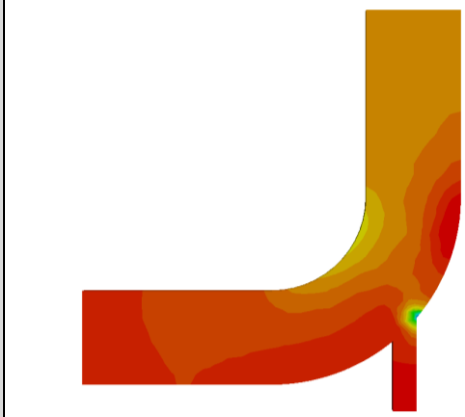
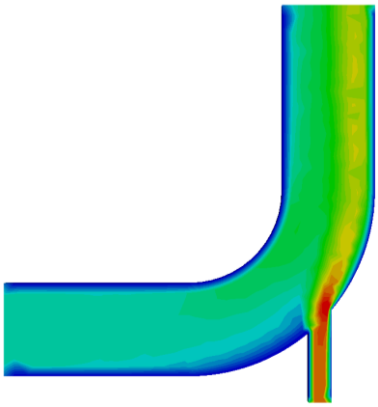
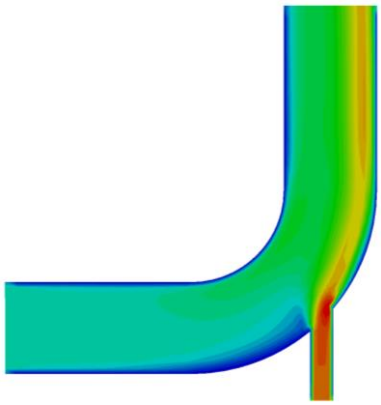
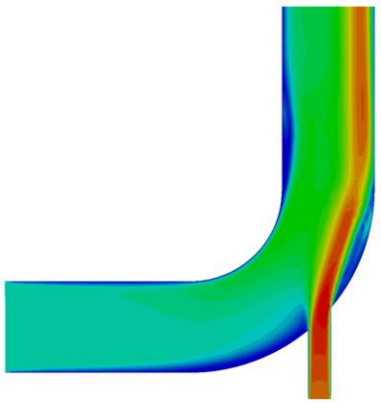
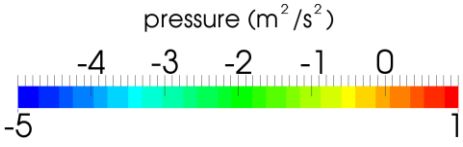
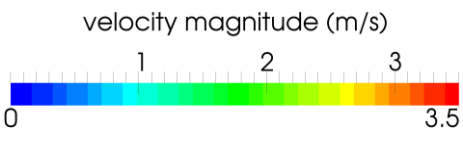
The Hex Fine mesh





Pressure and velocity for different meshes at t=75 s, along the arc shown

The comparison plots are along the line between points A (54 0 0) at the small tube entrance and B (60 60 0) at the large tube exit part (length units are in meter) for Tri-mesh, for other two meshes created using GAMBIT® the points are A (22 -33 0) and B (27 30 0).

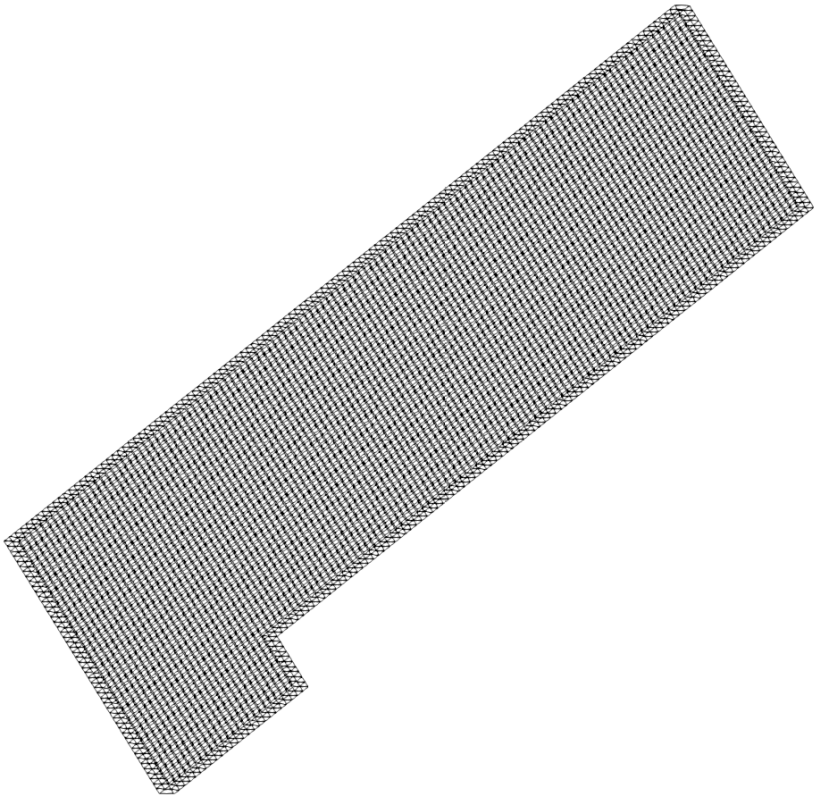
Mesh	Pressure	Velocity
Tri		
Hex		
Hex Fine		
		

Comparison of different mesh type results at t = 75 s

Note: For extracting data over a line, the line should be defined in ParaView using "Plot Over Line", then the data over this line can be exported by choosing Save Data from File menu in ParaView.

Tutorial Two

Built in Mesh



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. What is Mesh?

When studying fluid flow and heat transfer, mathematical equations known as partial differential equations (PDEs) describe how physical properties such as mass, energy, and momentum change over space and time. However, solving these equations directly (analytically) is extremely difficult unless the problem is very simple.

To solve PDEs numerically, these equations are discretized and converted from a set of PDEs to a set of algebraic equations. This involves breaking the entire fluid domain into many smaller, manageable sections. These small sections are called grid cells, and together they form a mesh.

A mesh is like a net or grid that covers the entire area where fluid behavior is analyzed. The finer (smaller) the mesh, the more accurately flow details can be captured, but at the cost of increased computational demand. Choosing the right mesh ensures a balance between accuracy and efficiency in simulations.

One of the most common numerical methods for solving these equations is the finite volume method (FVM), which is explained below.

2. The Finite Volume Method (FVM)

OpenFOAM® applies the finite volume method (FVM) to simulate fluid flow. This method works by applying a key equation called the transport equation, which describes how physical property (such as velocity, temperature, or pressure) moves through a fluid domain over time. The general transport equation is:

$$\frac{\partial(\rho\varphi)}{\partial t} + \nabla \cdot (\rho\varphi \boldsymbol{u}) = \nabla \cdot (\Gamma \nabla\varphi) + S_\varphi$$

Rate of change of φ inside fluid element	**+**	**Net rate of flow of φ out of fluid element**	**=**	**Rate of change of φ due to diffusion**	**+**	**Rate of change of φ due to sources**

The finite volume method works by applying and integrating this equation over a control volume (CV), which is a small section of the mesh. A mathematical technique called the Gauss divergence theorem helps converting volume integral terms in the equation into surface integrals. This allows for the calculation of the amount of a property entering and exiting each grid cell, ensuring that all properties are conserved throughout the simulation.

$$\int_{\Delta t} \frac{\partial}{\partial t}\left(\int_{CV} \rho\varphi \, dV\right) dt + \int_{\Delta t}\int_{A} \boldsymbol{n} \cdot (\rho\varphi \boldsymbol{u}) \, dA dt = \int_{\Delta t}\int_{A} \boldsymbol{n} \cdot (\Gamma \nabla\varphi) \, dA dt + \int_{\Delta t}\int_{CV} S_\varphi dV \, dt$$

For time-dependent problems, the equation must also be integrated over a small time step (Δt\Delta t) to account for changes in properties over time. This

step-by-step approach is essential for accurately capturing transient behaviors, such as turbulence or shock waves.

3. Discretization of Transport Equations

Discretization of the transport equations is critical to the finite volume method and is done using the mesh, which involves dividing the domain into smaller regions.

In CFD, the meshes can be divided into two main categories:

- **Structured meshes**: These are arranged in a regular, grid-like pattern, often using Cartesian coordinates (X, Y, Z directions). They are simple to use but may not work well for complex geometries.
- **Unstructured meshes**: These use irregularly shaped grid cells and can represent complex shapes, such as curved surfaces and complex objects, more accurately.

Mesh generation in OpenFOAM® is done using built-in tools such as blockMesh (for structured meshes) and snappyHexMesh (for unstructured meshes). External software like GAMBIT® can also be used for creating meshes. This tutorial focuses on using blockMesh, which provides a simple way to generate structured grids. More advanced mesh generation using snappyHexMesh is covered in Tutorial Twelve.

4. foamRun Solver – fluid module

In OpenFOAM® 12, the foamRun application serves as a versatile tool for executing various solver modules. Unlike traditional/legacy solvers (e.g. icoFoam) that are specific to certain types of simulations, foamRun dynamically loads and runs a solver module which can be either defined in the simulation setup or as a command-line argument. This modular approach enhances flexibility, allowing users to select appropriate solver modules for their specific simulation needs.

“fluid” is the solver module for steady or transient turbulent flow of compressible fluids with heat-transfer with optional mesh motion and change.

fluid Solver – forwardStep

Tutorial outline

Using foamRun and fluid solver, simulate 10 s of flow over a forward step.

Objectives

- Understand blockMesh
- Define vertices via coordinates as well as surfaces and volumes via vertices.

Data processing

Import your simulation into ParaView, and examine the mesh and the results in detail.

1. Pre-processing

1.1. Copying tutorial

Copy the tutorial from the following folder to your working directory:

`$FOAM_TUTORIALS/fluid/forwardStep`

1.2. Case structure

1.2.1. 0 directory

There are two new files in the 0 folder, T and Ma. File T includes the initial temperature values and Ma is the Mach number values which are calculated using the OpenFOAM® function objects (this can be ignored for this tutorial). Internal pressure and temperature fields are set to 1, and the initial velocity in the domain as well as the inlet boundary is set to (3 0 0).

Note: As it can be seen, the p unit is the same as the pressure unit (kg m^{-1} s^{-2}), because fluid module is for compressible fluids.

Note: Do not forget that, this example is a purely numeric example (you might have noticed this from the pressure values).

1.2.2. constant directory

By checking *physicalProperties* file, different properties of a compressible gas can be set:

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * */ /
thermoType
{
    type            hePsiThermo;
    mixture         pureMixture;
    transport       const;
    thermo          hConst;
    equationOfState perfectGas;
    specie          specie;
    energy          sensibleInternalEnergy;
}
// Note: these are the properties for a "normalized" inviscid gas
//       for which the speed of sound is 1 m/s at a temperature of 1K
//       and gamma = 7/5
mixture
{
    specie
    {
        molWeight       11640.3;
    }
    thermodynamics
    {
        Cp              2.5;
        Hf              0;
    }
    transport
    {
        mu              0;
        Pr              1;
    }
}

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

In the `thermoType`, the models for calculating thermo physical properties of gas are set:

- `type`: Specifies the underlying thermos-physical model, which in this case is enthalpy based thermodynamics while incorporating the equation of state using psi (compressibility)
- `mixture`: Is the model, which is used for the mixture, whether it is a pure mixture, a homogeneous mixture, a reacting mixture or ….
- `transport`: Defines the transport model used. In this example a constant value is used for viscosity.
- `thermo`: It defines the method for calculating heat capacities, e.g. in this example constant heat capacities are used.
- `equationOfState`: Shows the relation which is used for the compressibility of gases. Here ideal gas model is applied by selecting `perfectGas`.
- `energy`: This key word lets the solver decide which type of energy equation it should solve enthalpy or internal energy.

After defining the models for different thermos-physical properties of gas, the constants and coefficients of each model are defined in the sub-dictionary `mixture`. E.g. `molWeight` shows the molecular weight of gas, `Cp` stands for heat capacity, `Hf` is the heat of fusion, `mu` is the dynamic viscosity and `Pr` shows the Prandtl number.

By opening the momentumTransport the appropriate turbulent mode can be set (in this case it is laminar):

```
simulationType    laminar;
```

1.2.3. system directory

In this tutorial the mesh is not imported from other programs (e.g. GAMBIT®). It will be created inside OpenFOAM®. For this purpose the blockMesh tool is used. blockMesh reads the geometry and mesh properties from the *blockMeshDict* file (found in the system directory):

```
>nano blockMeshDict
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * *//
convertToMeters 1;
vertices
(
    (0 0 -0.05)
    (0.6 0 -0.05)
    (0 0.2 -0.05)
    (0.6 0.2 -0.05)
    (3 0.2 -0.05)
    (0 1 -0.05)
    (0.6 1 -0.05)
    (3 1 -0.05)
    (0 0 0.05)
    (0.6 0 0.05)
    (0 0.2 0.05)
    (0.6 0.2 0.05)
    (3 0.2 0.05)
```

```
    (0 1 0.05)
    (0.6 1 0.05)
    (3 1 0.05)
);
blocks
(
    hex (0 1 3 2 8 9 11 10) (25 10 1) simpleGrading (1 1 1)
    hex (2 3 6 5 10 11 14 13) (25 40 1) simpleGrading (1 1 1)
    hex (3 4 7 6 11 12 15 14) (100 40 1) simpleGrading (1 1 1)
);
defaultPatch
{
    type  empty;
}
boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 8 10 2)
            (2 10 13 5)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (4 7 15 12)
        );
    }
    bottom
    {
        type symmetryPlane;
        faces
        (
            (0 1 9 8)
        );
    }
    top
    {
        type symmetryPlane;
        faces
        (
            (5 13 14 6)
            (6 14 15 7)
        );
    }
    obstacle
    {
        type patch;
        faces
        (
            (1 3 11 9)
            (3 4 12 11)
        );
    }
);

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

As noted before units in OpenFOAM® are SI units. If the vertex coordinates differ from SI, they can be converted with the `convertToMeters` command. The number in the front of `convertToMeters` shows the constant, which should be

multiplied with the dimensions to change them to meter (SI unit of length). For example:

```
convertToMeters     0.001;
```

shows that the dimensions are in millimeter, and by multiplying them by 0.001 they are converted into meters.

In the `vertices` part, the coordinates of the geometry vertices are defined, the vertices are stored and numbered from zero, e.g. vertex `(0 0 -0.05)` is numbered zero, and vertex `(0.6 1 -0.05)` points to number 6.

Note: In OpenFOAM® (and C++) counting starts from 0 and not 1!

In the `block` part, blocks are defined. The array of numbers in front each block shows the block building vertices, e.g. the first block is made of vertices `(0 1 3 2 8 9 11 10)`.

After each block, the mesh is defined in every direction. e.g. `(25 10 1)` shows that this block is divided into:

- 25 parts in x direction
- 10 parts in y direction
- 1 part in z direction

As was explained in tutorial 1, even for 2D simulations the mesh and geometry should be 3D, but with one cell in the direction, which is not going to be solved, e.g. here number of cells in z direction is one and it's because of that it's a 2D simulation in x-y plane.

The last part, `simpleGrading(1 1 1)` shows the size function, in this case 1 means there is no change in the cell size from one cell to another

In the `boundary` part, each boundary is defined by the vertices it is made of, and its `type` and `name` are defined.

Note: For creating a face, the vertices should be chosen clockwise when looking at the face from inside of the geometry.

2. Running simulation

Before running the simulation, the mesh has to be created. In the previous step, the mesh and the geometry data were set. For creating it, the following command should be executed from the case main directory (e.g. forwardStep):

```
>blockMesh
```

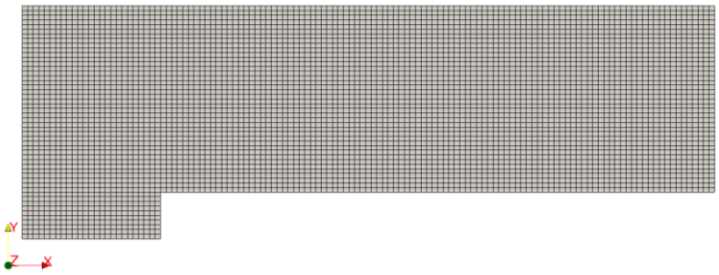
After that, the mesh is created in the constant/polyMesh folder. For running the simulation, type the solver name form case directory and execute it:

```
>foamRun -solver fluid
```

Note: The solver can be also defined in the controlDict (which is the case here) and then the simulation can be performed simply using foamRun command without the solver flag.

3. Post-processing

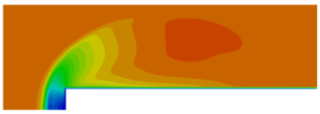
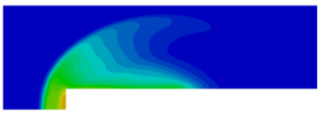
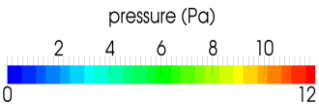
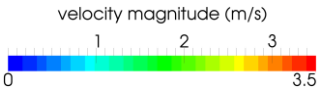
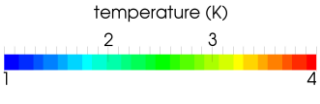
The mesh is presented in the following way in ParaView, and you can easily see the three blocks, which were created.



Mesh generated by blockMesh

Note: When a cut is created by default in ParaView, the program shows the mesh on that plane as a triangular mesh even if it is a hex mesh. In fact, ParaView changes the mesh to a triangular mesh for visualization, where every square is represented by two triangles. For avoiding this when creating a cut in ParaView in the Slice properties window, uncheck "Triangulate the Slice".

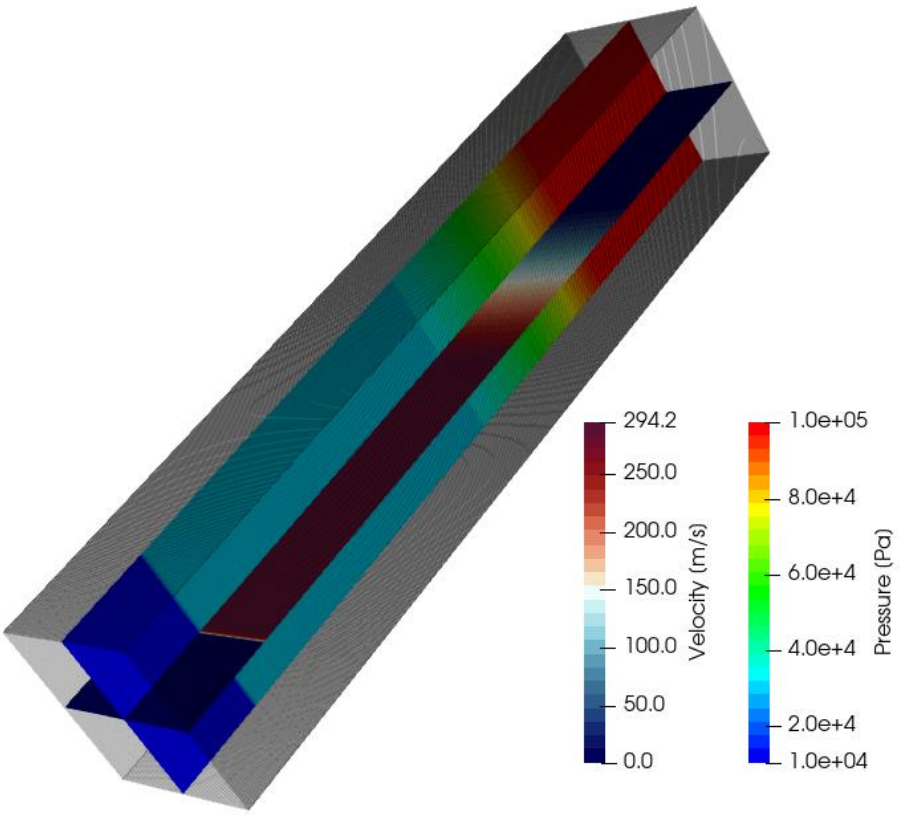
The simulation results are as follows:

Time	Pressure	Velocity	Temperature
0.5 s			
1 s			
10 s			
			

Pressure, velocity and temperature contours at different time steps

Tutorial Three

Patching Fields



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Initial and Boundary Conditions

Before running a numerical simulation, it is crucial to correctly define initial and boundary conditions for the problem. Poorly defined boundary conditions can lead to non-convergence or incorrect results. Understanding and applying these conditions properly ensures that the simulation behaves realistically and produces accurate and reliable results.

Initial conditions define the starting state of the simulation. These values are (usually) assigned to the center of every cell in the computational domain before the solver begins calculations. As the simulation progresses, the solver updates these values at each iteration based on the governing equations.

Key Points about Initial Conditions:

- **Importance:** They provide a reference point for the solver and influence how quickly the solution converges.
- **Transient problems:** The solution evolves over time from the specified initial state.
- **Steady-state problems:** The solver will iterate until a stable solution is found, regardless of the initial values, but the initial value might affect the speed of convergence and solution stability.

In OpenFOAM®, non-uniform initial conditions can be specified using the setFields utility, which allows defining non-uniform distributions of properties in the computational domain. This approach is especially useful when different regions of the domain require different starting values.

Boundary conditions define how the simulation domain interacts with its surroundings. They specify fixed values or behavior at the domain's boundaries, ensuring that the flow variables (such as velocity, pressure, or temperature) remain well-defined at these locations and represent the real physics at these boundaries. Types of Boundary Conditions:

- **Dirichlet Boundary Conditions (Fixed Value):** the variable (e.g., velocity or temperature) is assigned a fixed value at the boundary, e.g. specifying a constant temperature at a heated wall.
- **Neumann Boundary Conditions (Fixed Gradient):** instead of a fixed value, the gradient of the variable is specified at the boundary, e.g. a heat flux condition at a surface where the temperature gradient is controlled.
- **Mixed Boundary Conditions:** a combination of both Dirichlet and Neumann conditions, often used for heat transfer and fluid dynamics problems.
- **Periodic Boundary Conditions:** the solution at one boundary is linked to the opposite boundary, creating a repeating or cyclic condition useful for modeling infinite domains.
- **Symmetry**: used when a boundary behaves like a mirror, preventing flow normal to it.

In OpenFOAM®, boundary and initial conditions are defined in configuration files located in the 0 directory, where users can specify different types of conditions depending on the physical problem being modeled.

Setting appropriate boundary and initial conditions ensures that the simulation correctly represents the physical problem and achieves accurate, stable, and realistic results.

2. Courant-Friedrichs-Lewy (CFL) Condition

When performing numerical simulations, stability is a key concern. The Courant-Friedrichs-Lewy (CFL) condition is an essential mathematical criterion that ensures numerical schemes remain stable and that information propagates correctly within the computational domain.

Numerical solvers work by advancing the solution through a series of small time steps. However, for the solution to be physically meaningful, the information carried by waves or particles within the fluid must not move faster than the numerical grid can capture it. If this condition is violated, the simulation may become unstable, leading to numerical errors or divergence. To achieve this CFL condition is used which is mathematically expressed as:

$$Co = \frac{u\Delta t}{\Delta x} \leq 1$$

Where:

- u = Velocity magnitude in the considered direction
- Δt = Simulation time step size
- Δx = Mesh cell size in the corresponding direction

How the CFL Condition Affects Simulations:

- **Co > 1**: The simulation becomes unstable because the numerical scheme cannot properly capture the flow information moving between cells.
- **Co ≤ 1**: The information is correctly captured within the computational grid, ensuring stability and accuracy.

A finer mesh (smaller Δx) requires a smaller time step (Δt) to maintain stability, while higher velocity (larger u) also requires a smaller Δt to satisfy the CFL condition.

One of the most effective ways to determine a suitable Δt is to maintain the Courant number close to 1 (to have the biggest stable time step), using the maximum velocity in the domain and the smallest cell size. By using this approach, the solver will run efficiently while maintaining stability.

fluid Solver – shockTube

Tutorial outline

Use the “fluid” solver; simulate 0.007 s of flow inside a shock tube, with a mesh with 100, 1000 and 10000 cells in one dimension, for initial values 1 bar/0.1 bar and 10 bar/0.1 bar.

Objectives

- To understand the setFields utility
- Learn how to specify initial and boundary conditions
- Investigate effect of grid resolution.

Data processing

Import your simulation into ParaView, and compare results.

1. Pre-processing

1.1. Copying tutorial

Copy the tutorial from the following directory to your working directory

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

1.2. Mesh and setting fields

Looking at the *blockMeshDict* file (in system directory), it is obvious that it is a 1D mesh, because of the number of mesh cells in y and z directions is one, and also in `boundary` section, plates vertical to these directions are defined as `empty`. The mesh density can be set in the `blocks` part by changing x direction mesh size (e.g. change it from `1000` to `100` or `10000`).

Another important file is *setFieldsDict* (in the system directory), which is used by the tool `setFields` for patching (assigning an amount to a region) in the simulation. For example, here a pressure of 10^5 Pa is set as the default value for the entire region (in the `defaultFieldValues`), then half of the region (from 0 to 5) is patched with a pressure of 10^4 Pa.

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

defaultFieldValues ( volVectorFieldValue U ( 0 0 0 ) volScalarFieldValue T
348.432 volScalarFieldValue p 100000 );

regions         ( boxToCell { box ( 0 -1 -1 ) ( 5 1 1 ) ; fieldValues (
volScalarFieldValue T 278.746 volScalarFieldValue p 10000 ) ; } );

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

In the `defaultFieldValues`, a value is assigned to the whole domain, for example here, the velocity has been set everywhere to zero, the temperature to 348.432 K, and the pressure to 100000 Pa. In the `regions`, a specific value is patched to a certain region of the domain. In this example the region is defined as a cube, by the coordinates of one of its diagonals in `boxToCell`.

After choosing the region, the new values are assigned to the parameters (e.g. temperature at 278.746 K and pressure at 10000 Pa).

2. Running simulation

First the mesh needs to be created:

```
>blockMesh
```

In order to assign the values which were set in the setFieldDict:

```
>setFields
```

Then run:

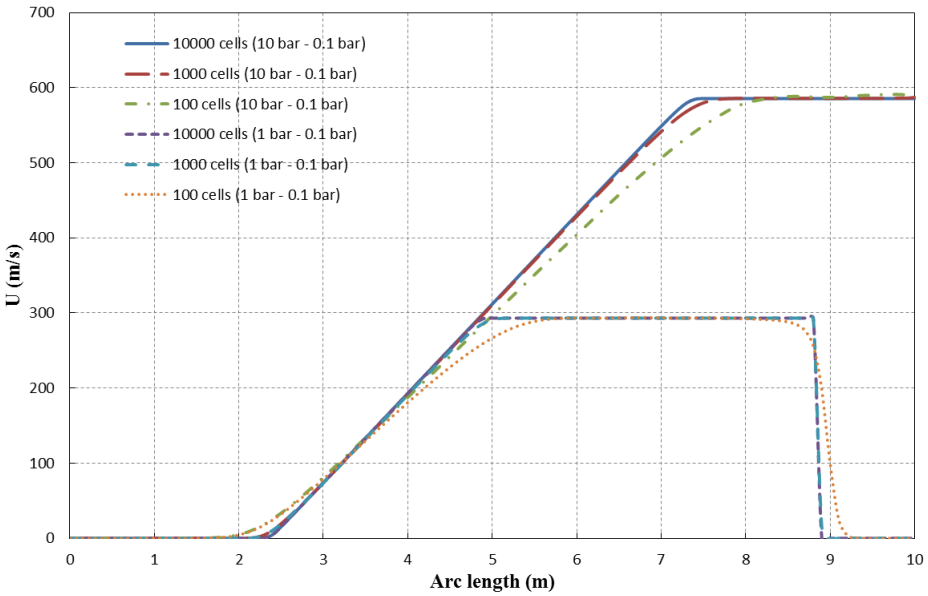
```
>foamRun -solver fluid
```

Note: Checking the information printed to the terminal (or log file) you can see how by decreasing the cell size (e.g. by increasing the number of cells) or by increasing the velocity (changing the pressure values) the Co number is increasing at a constant time step. In case the Co is getting bigger than one, the `deltaT` needs to be adopted accordingly to keep the Co below one.

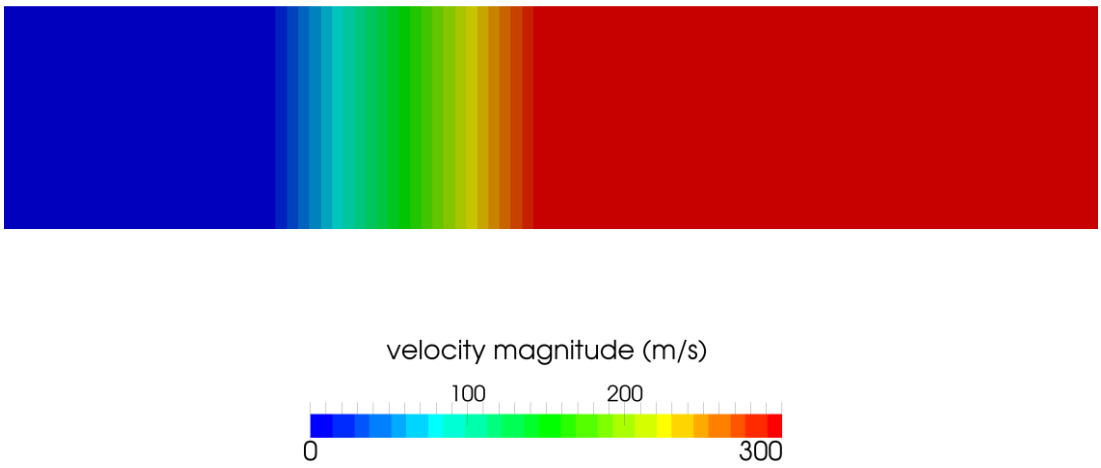
Note: After running setFields for the first time, the files in the 0 directory are overwritten. If the mesh is changed these files are not compatible with the new mesh and the simulation will fail. To solve this problem replace the files in the 0 directory with the files in the 0.orig or the files with suffix ".orig", e.g. p.orig in the 0 directory. In the OpenFOAM® files or directories with suffix ".orig" ("original") usually contain the backup files. If a command changes the original files these files can be replaced.

3. Post-processing

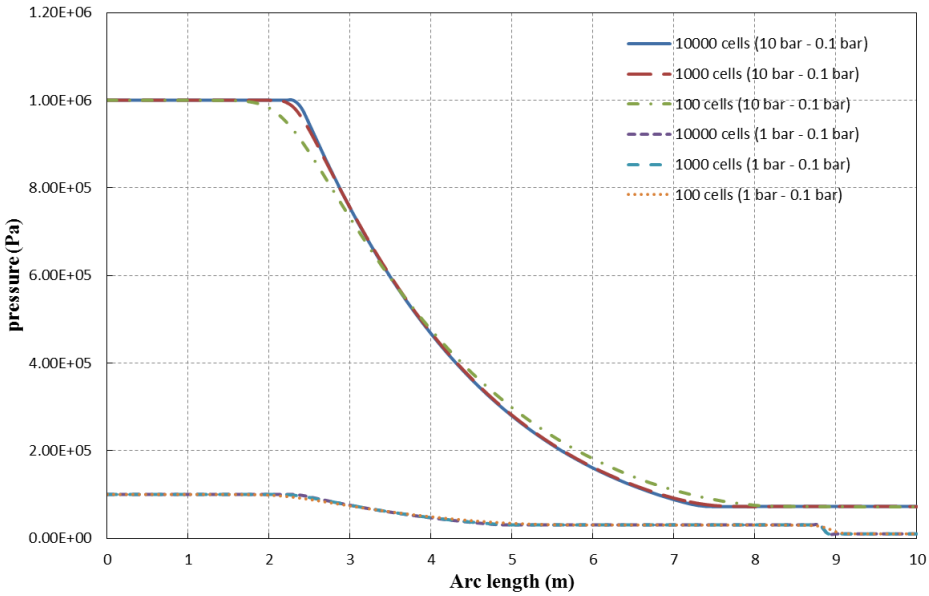
The simulation results are as follows:



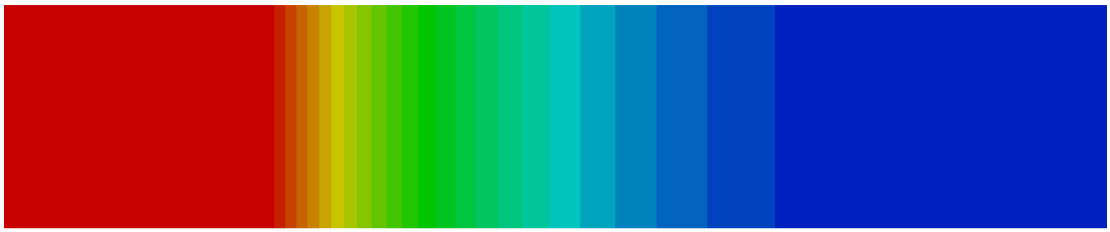
Velocities for different configurations along tube at t = 0.007 s

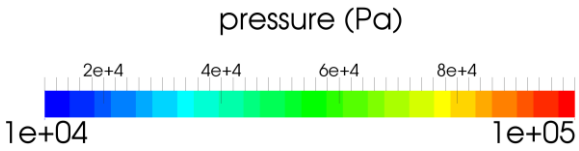


Velocity along tube axis for 10 bar/0.1bar and 10000 cells case at t = 0.007s

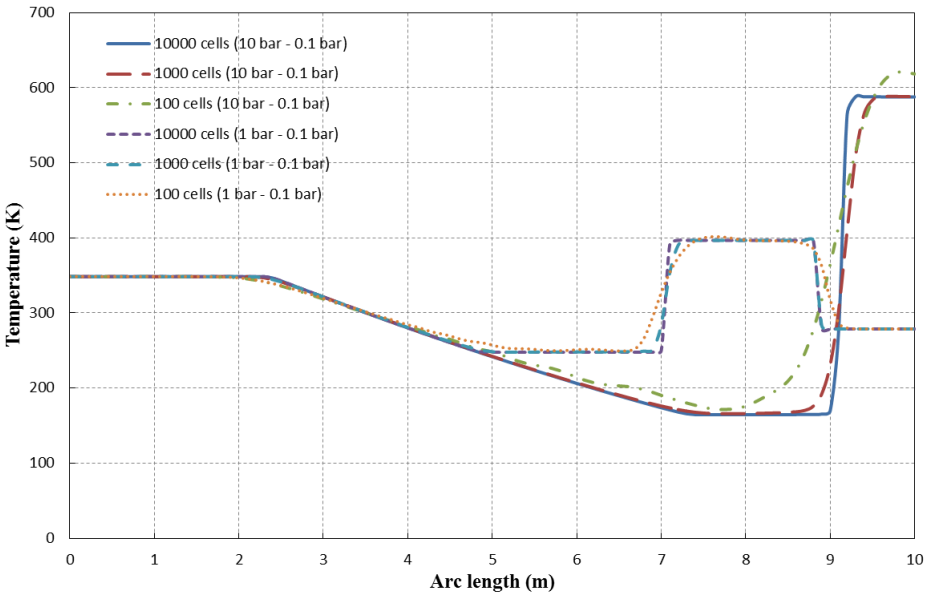


Pressures for different configurations along tube at t = 0.007 s

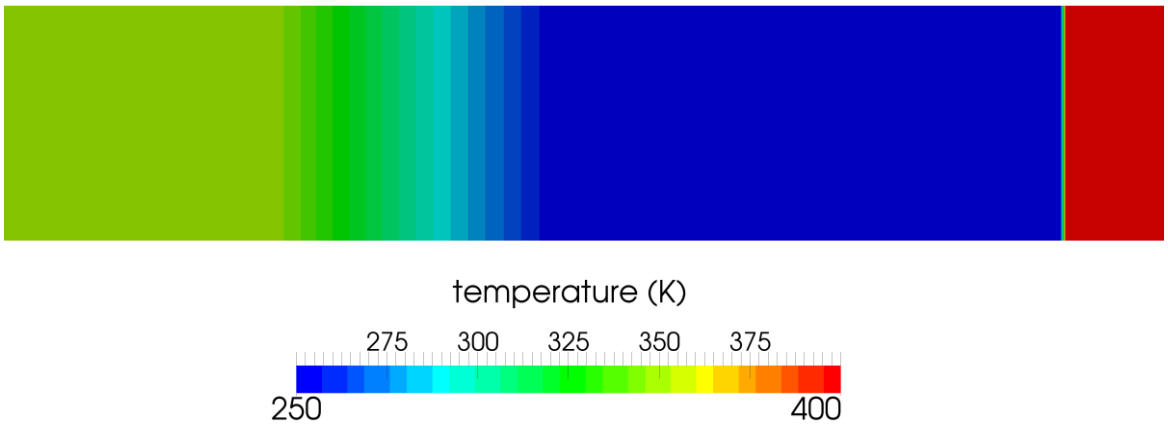




Pressure along tube axis for 10 bar/0.1bar and 10000 cells case at t = 0.007s



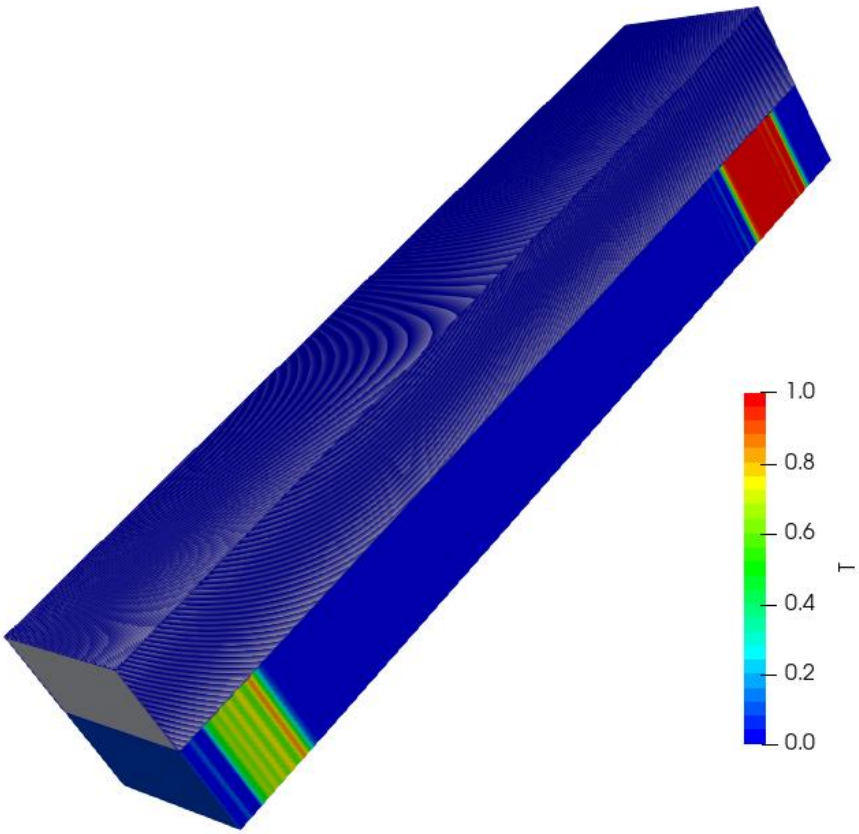
Temperature for different configurations along tube at t = 0.007 s



Temperature along tube axis for 10 bar/0.1bar and 10000 cells case at t = 0.007 s

Tutorial Four

Discretization – Part 1



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Discretizing general transport equation terms

Understanding the process of discretization is essential in Computational Fluid Dynamics (CFD). Discretization involves breaking down continuous differential equations into algebraic equations that can be solved numerically. In OpenFOAM®, various discretization schemes are used to approximate different terms in the transport equation, which describes how physical quantities (e.g., velocity, temperature, or concentration) change over space and time. Below is a detailed explanation of how each term in the transport equation is discretized.

1.1. Time derivative

The time derivative term represents how a variable evolves over time. This term is crucial for transient simulations, where the solution changes over time. Discretization of the time derivative such as $\frac{\partial \rho\varphi}{\partial t}$ of the transport equation is performed by integrating it over the control volume of a grid cell. Here, the Euler implicit time differencing scheme is explained. It is unconditionally stable, but only first order accurate in time. Assuming linear variation of φ within a time step gives:

$$\int_V \frac{\partial \rho\varphi}{\partial t} dV \approx \frac{\rho_P^n \varphi_P^n - \rho_P^0 \varphi_P^0}{\Delta t} V_P$$

Where $\varphi^n \equiv \varphi\,(t + \Delta t)$ stands for the new value at the time step we are solving for and $\varphi^0 \equiv \varphi(t)$ denotes old values from the previous time step.

Higher-order schemes, such as Crank-Nicolson, offer improved accuracy but may introduce oscillations if not applied carefully.

1.2. Convection term

The convection term describes the transport of a property due to the motion of the fluid. Convection plays a significant role in CFD since it governs how momentum, heat, and mass are transported within the fluid domain.

Discretization of convection terms is performed by integrating over a control volume and transforming the volume integral into a surface integral using the Gauss's theorem as follows:

$$\int_A \boldsymbol{n} \cdot (\rho\varphi\boldsymbol{u})\, dA \approx \sum_f \boldsymbol{n} \cdot (A\rho\boldsymbol{u})_f \varphi_f = \sum_f F\varphi_f$$

Where F is the mass flux through the face f defined as $F = \boldsymbol{n} \cdot (A\rho\boldsymbol{u})_f$. The value φ_f on face f can be evaluated in a variety of ways, which will be covered later in section 2. The subscript f refers to a given face.

Choosing the right numerical scheme is essential for balancing accuracy and stability in convection-dominated problems.

1.3. Diffusion term

The diffusion term represents the spread of the property due to molecular effects such as viscosity or heat conduction. The diffusion term is a second-order derivative term that requires careful discretization. Discretization of diffusion terms is done in a similar way to the convection terms. After integration over the control volume, the term is converted into a surface integral:

$$\int_A \boldsymbol{n} \cdot (\Gamma \nabla \varphi)\, dA = \sum_f \Gamma_f (\boldsymbol{n} \cdot \nabla_f \varphi) A_f$$

Note that the above approximation is only valid if Γ is a scalar. Here, $\nabla_f \varphi$ denotes the gradient at the face A denotes the surface area of the control volume and A_f denotes the area of a face for the control volume. However, it does not imply a specific discretization technique. The face normal gradient can be approximated using the scheme:

$$\boldsymbol{n} \cdot \nabla_f \varphi = \frac{\varphi_N - \varphi_P}{|\boldsymbol{d}|}$$

This approximation is second order accurate when the vector $\boldsymbol{d}$ between the center of the cell of interest P and the center of a neighboring cell N is orthogonal to the face plane, i.e. parallel to A. In the case of non-orthogonal meshes, a correction term could be introduced which is evaluated by interpolating cell centered gradients obtained from Gauss integration.

1.4. Source term

Source terms, such as $S_\varphi$of the transport equation, can be a general function of φ. Before discretization, the term is linearized:

$$S_\varphi = \varphi S_I + S_E$$

where S_E and S_I may depend on φ. The term is then integrated over a control volume as follows:

$$\int_V S_\varphi dV = S_I V_P \varphi_P + S_E V_P$$

There is some freedom on exactly how a particular source term is linearized. When deciding on the form of discretization (e.g. linear, upwind), its interaction with other terms in the equation and its influence on boundedness and accuracy should be examined.

2. Discretization Schemes

Discretization schemes determine how values are interpolated between cell centers and faces to compute fluxes accurately. The choice of scheme affects solution accuracy, numerical diffusion, and computational stability. Below are commonly used schemes and their respective advantages and limitations.

In general, interpolation needs a flux F through a general face f, and in some cases, one or more parameters γ. The face value φ_f can be evaluated from the

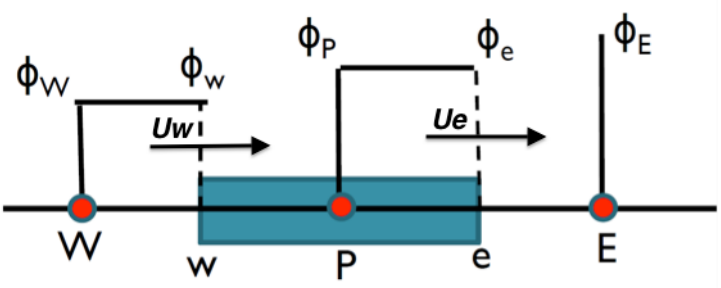
values in the neighboring cells using a variety of schemes. The flux satisfies continuity constraints, which is prerequisite to obtaining the results.

2.1. First Order Upwind Scheme

In first order upwind scheme we define φ as follows:

Note: Here we define two faces, e and w. To obtain flux through faces e and w, we need to look its neighbouring values at P/E and W/P respectively. The subscripts denote the face at which the face value φ or the flux F is located at.

$$\varphi_e = \varphi_P \qquad if, F_e > 0$$
$$\varphi_e = \varphi_E \qquad if, F_e < 0$$



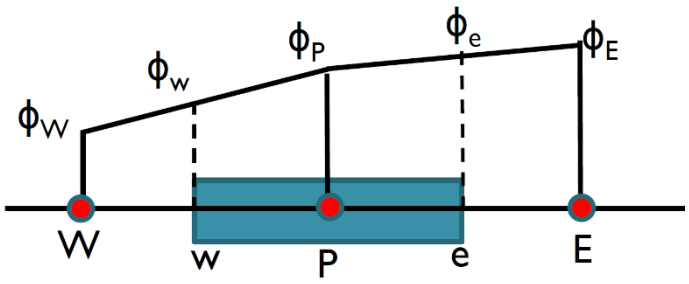
First Order Upwind Scheme

φ_w is also defined similarly (Positive direction is from W to E).

2.2. Central Differencing Scheme

Here, we use linear interpolation for computing the cell face values.

$$\varphi_e = \frac{\varphi_E + \varphi_P}{2}, \qquad \varphi_w = \frac{\varphi_P + \varphi_W}{2}$$



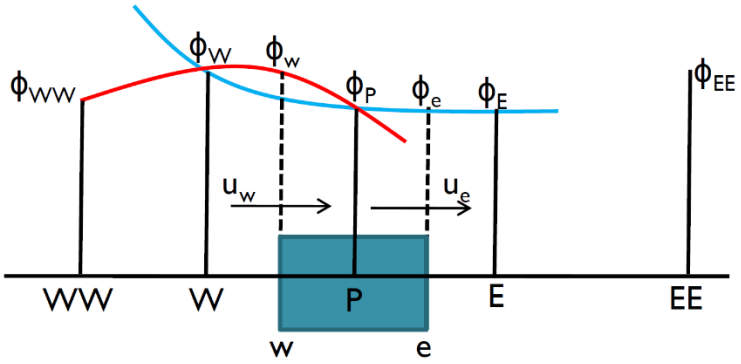
Central Differencing Scheme

2.3. QUICK

QUICK stands for Quadratic Upwind Interpolation for Convective Kinetics. In the QUICK scheme 3 point upstream-weighted quadratic interpolation are used for cell face values.

$$When\ F_e > 0, \qquad \varphi_e = \frac{6}{8}\varphi_P + \frac{3}{8}\varphi_E - \frac{1}{8}\varphi_W$$

$When\ F_w > 0, \qquad \varphi_w = \frac{6}{8}\varphi_W + \frac{3}{8}\varphi_P - \frac{1}{8}\varphi_{WW}$



QUICK scheme

Similar expressions can be obtained for $F_e < 0$ and $F_w < 0$.

Now that you know a bit more about discretization schemes, we can move on to the tutorial. In this tutorial, the scalarTransportFoam solver is used. More explanation of this solver can be found below.

3. functions solver

Among *foamRun* solver modules *functions* solver, which is specifically designed to execute function objects as defined in the *system/controlDict* or *system/functions* files. Function objects are utilities within OpenFOAM that facilitate workflow configurations and enhance simulations by generating additional data during runtime or post-processing. By utilizing the *functions* solver module with *foamRun*, users can automate the execution of these function objects, streamlining processes such as data logging, field calculations, and custom analyses without the need to run a full simulation. This approach optimizes computational resources and simplifies the integration of auxiliary calculations into the simulation workflow.

One of these functions is scalarTransport which resolves a transport equation for a passive scalar. The velocity field and boundary condition need to be provided by the user. It works by setting the source term in the transport equation to zero (see equation below), and then solving the equation.

$$\frac{\partial(\rho\varphi)}{\partial t} + \nabla \cdot (\rho\varphi \boldsymbol{u}) - \nabla \cdot (\Gamma \nabla\varphi) = 0$$

functions Solver – shockTube

Tutorial outline

Use the functions solver, simulate 5 s of flow inside a shock tube, with 1D mesh of 1000 cells (10 m long geometry from -5 m to 5 m). Patch with a scalar of 1 from -0.5 to 0.5. Simulate following cases:

- Set U to uniform (0 0 0). Vary diffusion coefficient (low, medium and high value).
- Set the diffusion coefficient to zero and also U to (1 0 0) and run the simulation in the case of pure advection using following discretization schemes:
  - upwind
  - linear
  - linearUpwind
  - QUICK
  - cubic

Objectives

- Understanding different discretization schemes.

Data processing

Import your simulation into ParaView, and plot temperature along tube length.

1. Pre-processing

1.1. Compile tutorial

Create a folder in your working directory:

```
>mkdir shockTube
```

Copy the following case to the created directory:

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

In the 0 directory, create a copy of T.orig and U.orig and rename them to T and U respectively. In the constant directory delete *physicalProperties* file, and in the system directory delete all the files except for *blockMeshDict* and *setFieldsDict* files.

From the following case:

```
$FOAM_TUTORIALS/incompressibleFluid/pitzDailyScalarTransp
ort
```

Copy *physicalProperties* file to the constant folder in the newly created case constant folder. Copy *controlDict*, *fvSchemes, fvSolution* and *functions* files from the above case system directory to the created case system directory.

1.2. system directory

Edit the *setFieldsDict*, to patch the T field to 1.0 between -0.5 m and 0.5 m and to set the U to (0 0 0) for the whole domain. For setting U in the whole domain to (1 0 0), just change `(0 0 0)` to `(1 0 0)`:

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
defaultFieldValues
(
        volVectorFieldValue U ( 0 0 0 )
        volScalarFieldValue T 0.0
);
regions
(
        boxToCell
        {
                box ( -0.5 -1 -1 ) ( 0.5 1 1 );

                fieldValues
                (
                        volScalarFieldValue T 1.0
                );
        }
);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

In the *controlDict*, update the `endTime` to 5 for 5s of simulation. As it was mentioned before, the discretization scheme for each operator of the governing equations can be set in *fvSchemes*.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,T)      Gauss linearUpwind grad(T);
}

laplacianSchemes
{
    default         none;
    laplacian(DT,T) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

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

For each type of operation a default scheme can be set (e.g. for `divSchemes` is set to `none`, it means no default scheme is set). Also a special type of discretization for each element can be assigned (e.g. `div(phi,T)` it is set to `linearUpwind`). For each element, where a discretization method has not been set, the default method will be applied. If the default setting is `none`, no scheme is set for that element and the simulation will crash.

Note: *In fvSchemes, the schemes for the time term of the general transport equation are set in* `ddtSchemes` *sub-dictionary.* `divSchemes` *are responsible for the advection term schemes and* `laplacianSchemes` *set the diffusion term schemes.*

Note: `divSchemes` *should be applied like this: Gauss + scheme. The* `Gauss` *keyword specifies the standard finite volume discretization of Gaussian integration which requires the interpolation of values from cell centers to face centers. Therefore, the* `Gauss` *entry must be followed by the choice of interpolation scheme (www.openfoam.org).*

In the *fvSolution* file add pressure reference cell number and value to the PIMPLE sub-dictionary, it should look like the following:

```
PIMPLE
{
    nNonOrthogonalCorrectors 0;
    pRefCell 0;
    pRefValue 0;
}
```

Note: pRefCell and pRefValue are dummy values that solver can start the calculations, since there is no pressure field available.

In the *functions* file, just keep the line for activating the scalar transport function. In the functions file, different functions can be called, in this case the scalar transport function is called with using "T" as the property (scalar) to be solved, it uses a constant diffusivity model and set the value of it by setting D (in this case it is 0.01).

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#includeFunc scalarTransport(T, diffusivity=constant, D = 0.01)
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Note: By setting the diffusion coefficient "D" to zero, the case will be switched to a pure advection simulation with no diffusion.

For part two:

- Set the diffusivity to 0, by setting the `D` in the *functions* file
- Set the velocity field to (1 0 0), either by using setFields utility or simply in the *0/U* file change the `internalField` to `(1 0 0)`
- Set different schemes in the *fvSchemes* file, for the `div(phi, T)`

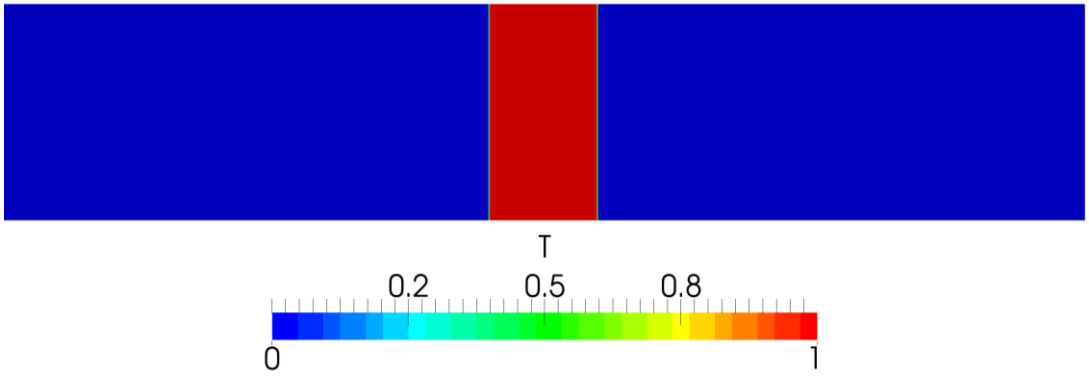
2. Running simulation

```
>blockMesh

>setFields

>foamRun -solver functions
```

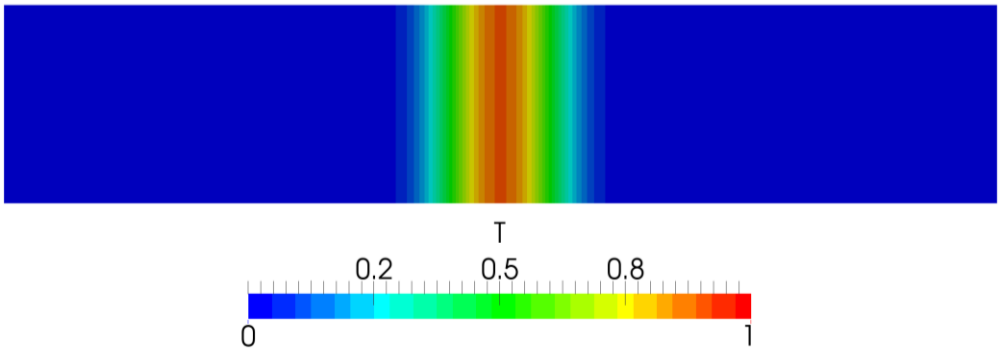
3. Post-processing

The simulation results are as follows.

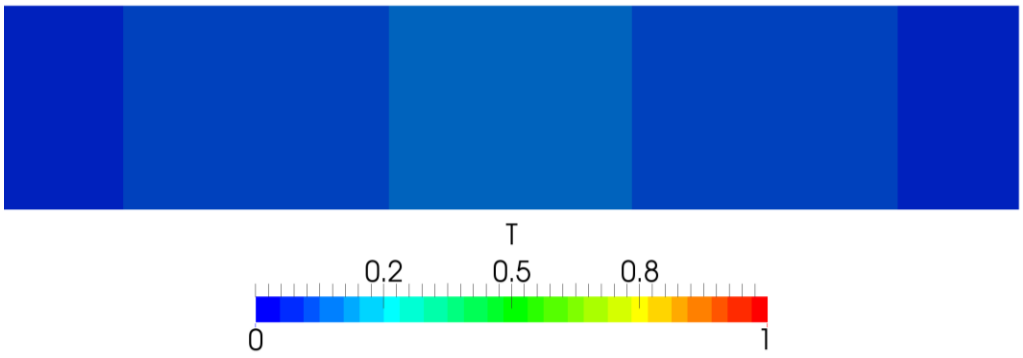
A. Case with zero velocity (pure diffusion):



Pure diffusion with low diffusivity (0.00001) at t = 5 s

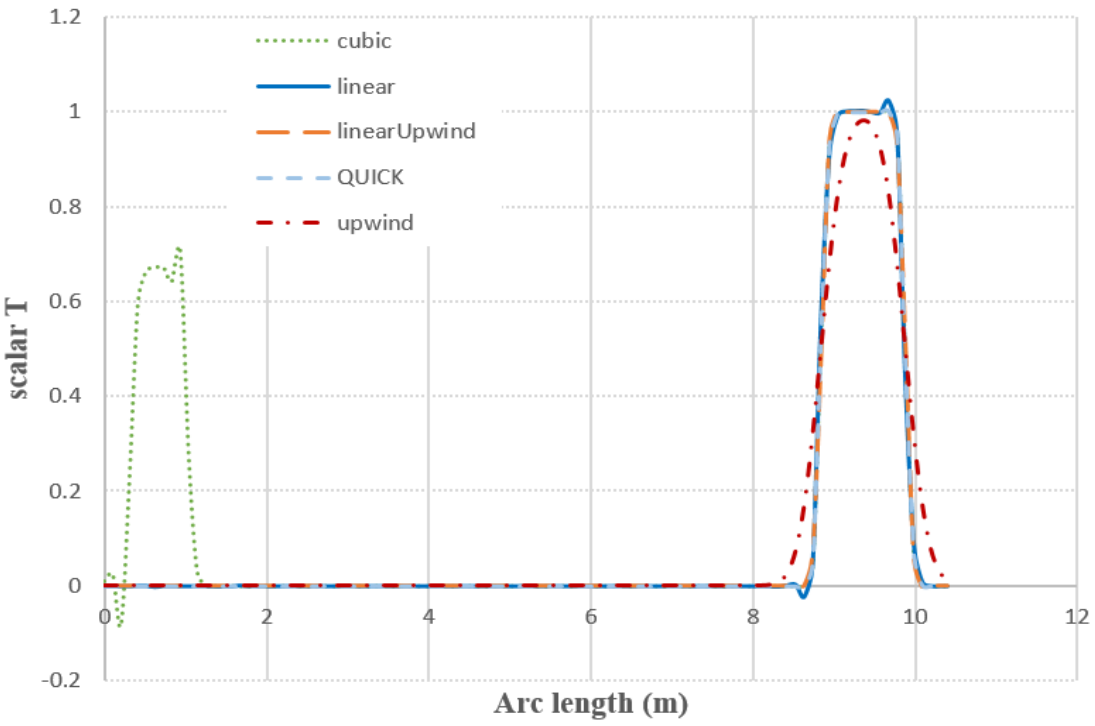


Pure diffusion with medium diffusivity (0.01) at t = 5 s



Pure diffusion with high diffusivity (1) at t = 5 s

B. Case with pure advection (diffusion coefficient = 0):

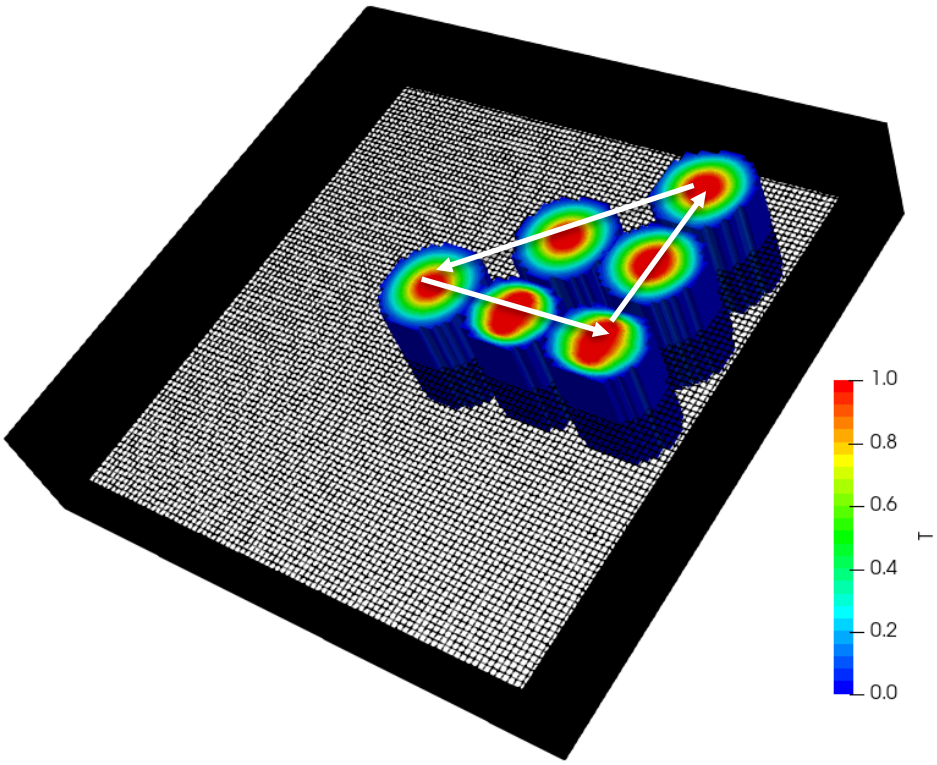


Scalar T along tube at t = 4 s

The cubic scheme predicted an unexpected rise in temperature between around 0 to 1 m, which differs hugely from the other schemes. This can be explained by looking at the numerical behavior of the cubic scheme. It is operated in fourth order accuracy with unbounded solutions, which caused another false root solution to be found. Therefore, higher order accuracy does not always generate better results!

Tutorial Five

Discretization – part 2



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Properties of discretization schemes

When performing numerical simulations, it is crucial to choose the right discretization scheme to ensure physically realistic results. The effectiveness of a discretization scheme depends on several key properties, including conservativeness, boundedness, and transportiveness. Understanding these properties helps in selecting the appropriate scheme for a given problem in Computational Fluid Dynamics (CFD). These properties also influence the numerical accuracy, stability, and efficiency of the simulation.

1.1. Conservativeness

A discretization scheme is conservative if it ensures that the total amount of a transported quantity (e.g., mass, momentum, energy) is preserved within the solution domain. This property is fundamental for obtaining physically meaningful results in fluid dynamics and preventing artificial gain or loss of the transported variable.

To achieve conservativeness, the flux balance across each control volume must be maintained. Mathematically, this means:

- The flux of φ leaving a control volume across a certain face must equal the flux entering the adjacent control volume through the same face.
- The discretization scheme should represent the flux through a common face consistently across adjacent control volumes.

A scheme that violates conservativeness can lead to unphysical results, such as artificial creation or loss of mass or energy. Finite volume methods naturally ensure conservation by integrating the governing equations over control volumes, ensuring that what exits one control volume enters the next.

1.2. Boundedness

Most numerical solvers use iterative techniques to obtain the solution at each node. The solver starts with an initial guess and updates the values until convergence is achieved. To ensure a stable and physically meaningful solution, the discretization scheme must satisfy boundedness criteria.

A bounded solution means that the numerical values of φ remain within reasonable limits, avoiding unrealistic oscillations or negative concentrations, which would be non-physical.

The sufficient condition for condition for boundedness is:

$$\frac{\sum|a_{nb}|}{|a'_P|} \begin{cases} \leq 1 \; at \; all \; nodes \\ < 1 \; at \; one \; node \; at \; least \end{cases}$$

Here a'_P is the net coefficient of the central node P (i.e. $a'_P - S_P$), a_{nb} are the coefficient of the neighbouring nodes. If the condition is satisfied, the resulting matrix of coefficients is diagonally dominant. We need the net coefficients to be

as large as possible; this means that S_P should be always negative. If this is the case, S_P becomes positive due to the modulus sign and adds to a_P.

1.3. Transportiveness

Transportiveness refers to the ability of a discretization scheme to correctly account for the dominant transport mechanism in a problem. This is assessed using the Peclet number (Pe), which measures the relative strength of convection versus diffusion:

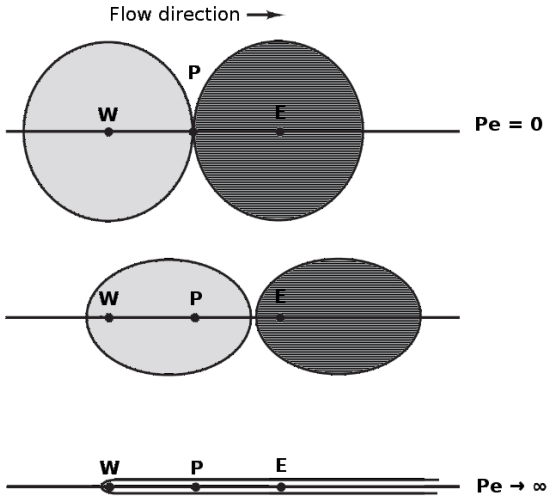
$$Pe = \frac{N_{conv}}{N_{diff}} = \frac{LU}{D}$$

Note: L is a characteristic length scale, U is the velocity magnitude, D is a characteristic diffusion coefficient.

The primary goal is to ensure that the transportiveness is borne out of the discretization scheme.

Let us consider the effect at a point P due to two constant sources of φ at nearby points *W* and *E* on either side, in three cases.

1. When Pe = 0 (pure diffusion), the contours of φ are circles, as φ is spread out evenly in all directions
2. As Pe increases, the contours become elliptical, as the values of φ are influenced by convection
3. When Pe→∞, the countours become straight lines, since φ are stretched out completely and affected only by upstream conditions



4. Transportiveness property

2. Assessing the general discretization schemes

It is useful to compare the different types of general discretization schemes covered in Tutorial Four based on their conservativeness, boundedness and transportiveness properties.

Different discretizing schemes assessment

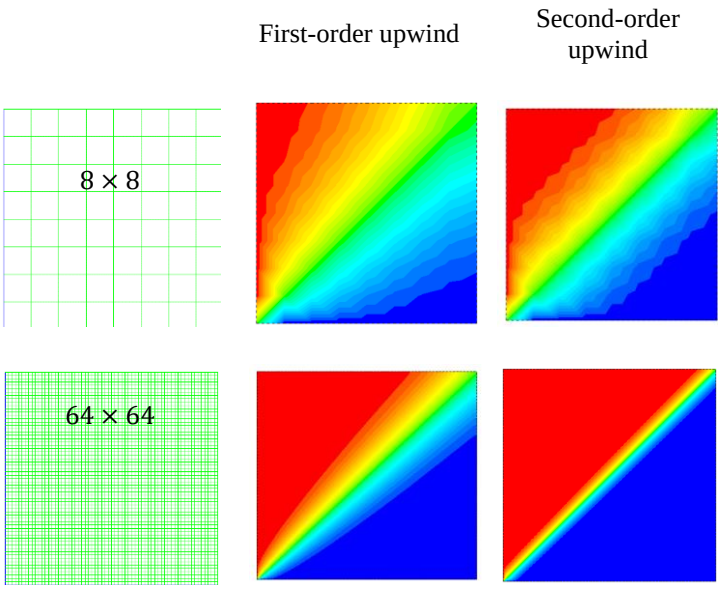
Scheme	Conser-vative	Bounded	Accuracy	Trans-portive	Remarks
Upwind	Yes	Unconditionally bounded	First order	Yes	Include false diffusion if the velocity vector is not parallel to one of the coordinate directions
Central Differencing	Yes	Conditionally bounded*	Second order	No	Unrealistic solutions at large Pe number
QUICK	Yes	Unconditionally bounded	Third order	Yes	Less computationally stable. Can give small undershoots and overshoots

** **Pe should be less than 2.***

3. Numerical (false) diffusion

Numerical diffusion is an artificial diffusion effect that occurs when the flow direction is not aligned with the computational grid. It is a numerical artifact that introduces additional diffusion into the system and primarily affects convection-dominated flows with high Peclet numbers (Pe).

False diffusion is more prominent when using first-order upwind schemes. It decreases with finer grids, but using higher-order schemes (e.g., QUICK) is a more effective way to reduce it. False diffusion can distort flow structures, leading to non-physical results, especially in high-speed flows. Using a high-resolution grid or aligning the mesh with the flow direction can help mitigate numerical diffusion.



Numerical diffusion

4. Numerical behavior of OpenFOAM® discretization schemes

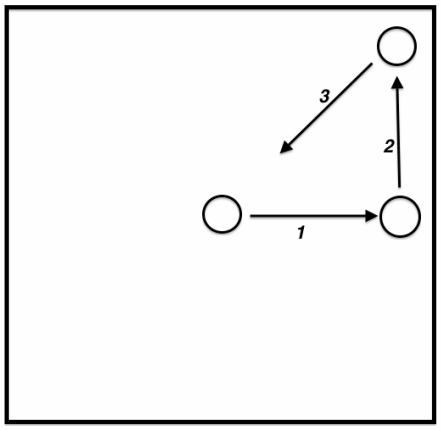
The choice of discretization scheme for this tutorial should depend critically on the numerical behavior of the scheme. Using higher order schemes, numerical diffusion errors can be reduced, however it requires higher computational efforts.

Scheme	Numerical behavior
upwind	First order, bounded
linear	Second order, unbounded
linearUpwind	First/second order, bounded
QUICK	Second order or higher, bounded
cubic	Fourth order, unbounded

functions Solver – circle

Tutorial outline

Use the functions solver, do simulate the movement of a circular scalar spot region (radius = 1 m) at the middle of a 100 × 100 cell mesh (10 m × 10 m), then move it to the right (3 m), to the top (3 m) and diagonally.



Schematic sketch of the problem

Objectives

- Choosing the best discretization scheme.

Data processing

Examine your simulation in ParaView.

1. Pre-processing

1.1. Compile tutorial

Create the new case in your working directory like in tutorial four.

1.2. 0 directory

To move the circle to right change the `internalField` to `(1 0 0)` in the U file for setting the velocity field towards the right.

1.3. system directory

Modify the *blockMeshDict* for creating a 2D geometry with 100 × 100 cells mesh.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
convertToMeters 1;

vertices
(
    (-5 -5 -0.01)
    (5 -5 -0.01)
    (5 5 -0.01)
    (-5 5 -0.01)
    (-5 -5 0.01)
    (5 -5 0.01)
    (5 5 0.01)
    (-5 5 0.01)
);
blocks
(
    hex (0 1 2 3 4 5 6 7) (100 100 1) simpleGrading (1 1 1)
);
edges
(
);
boundary
(
    sides
    {
        type patch;
        faces
        (
            (1 2 6 5)
            (0 4 7 3)
            (3 7 6 2)
            (0 1 5 4)
        );
    }
    empty
    {
        type empty;
        faces
        (
            (5 6 7 4)
            (0 3 2 1)
        );
    }
);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

Choose a discretization scheme based on the results from the previous example and set it in the *fvSchemes*.

In the *setFieldsDict* patch a circle to the middle of the geometry using the following lines.

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

defaultFieldValues (volScalarFieldValue T 0 );

regions
(
cylinderToCell
{
       p1 ( 0 0  -1 );
p2 ( 0 0  1 );
       radius 0.5;
       fieldValues
(
volScalarFieldValue T 1
) ;
}
);

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

`cylinderToCell` command is used to patch a cylinder to the region, `p1` and `p2` show the two ends of cylinder center line, in the `radius` the radius is set.

Check *controlDict*, in the first part of simulation, where the circle should move to the right set the `startFrom` to `startTime` and `startTime` to `0`. By a simple calculation, it can be seen that the `endTime` should be 3s (to move the circle from center to the right side). Similar calculations need to be done for the two other parts, except the `startTime` is set to the `endTime` of previous part, and new `endTime` should be that part "simulation time" plus `endTime` of the previous part.

Note: In the functions file set D to zero (no diffusion!).

2. Running Simulation

```
>blockMesh
```

```
>setFields
```

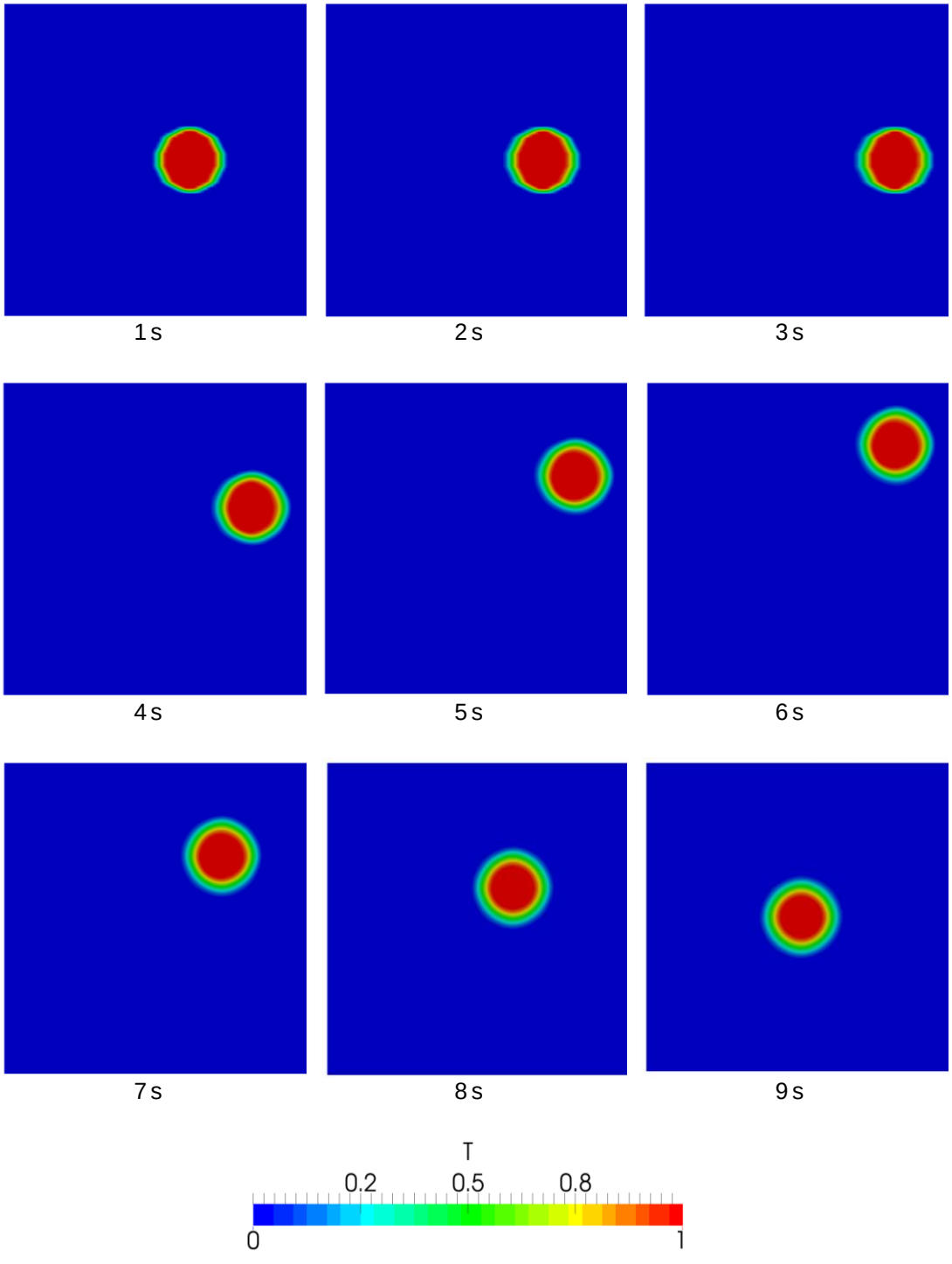
```
>foamRun -solver functions
```

For running further parts (moving the circle to top, and then diagonally), in the 0 folder in the U file change the `internalFiled` velocity to (0 1 0) so the circle moves up, and to (-1 -1 0) to move the circle diagonally back to the original position.

Note: In the controdDict file, subSolverTime is set to 0 and therefore even if the startTime is set to latestTime, the simulation will read the U file from time 0!

3. Post-processing

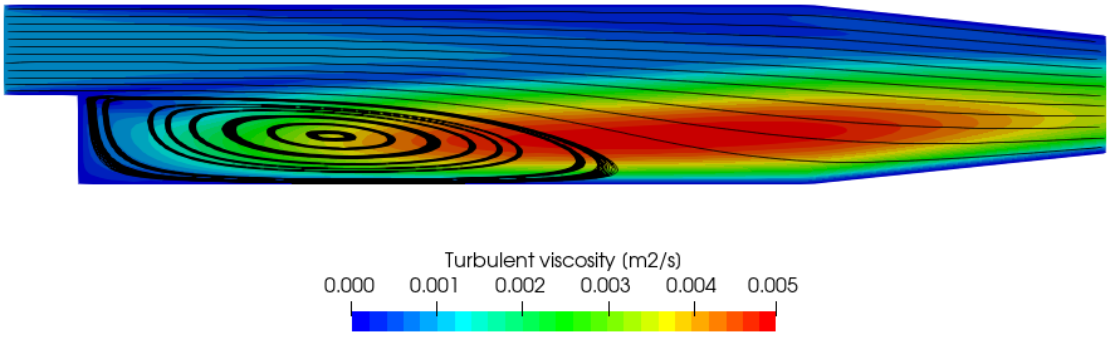
The simulation results are as follows:



Position of the circle at different time steps

Tutorial Six

Turbulence – Steady State



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Why turbulence modeling?

Many real-world engineering applications involve turbulent flow, which is a highly unsteady and chaotic phenomenon characterized by a wide range of swirling motions, called eddies, that exist at different scales. Accurately solving turbulent flows requires resolving all these eddies, which would demand an enormous amount of computational power and memory. In practical applications, such a Direct Numerical Simulation (DNS) is infeasible due to its excessive computational cost.

To overcome this challenge, turbulence models are used to approximate the effects of turbulent eddies without resolving them explicitly. These models simplify the governing equations of fluid flow while still capturing the essential characteristics of turbulence.

An important principle in turbulence modeling is averaging, which simplifies the governing equations of turbulent motion. Due to computational limitations, it is not always possible to model turbulent flow at fine spatial and temporal resolutions. Turbulence models compensate for this limitation by representing the unresolved scales of motion.

There are different types of turbulence models:

- RANS-based models:
  - Linear eddy-viscosity models
    - Algebraic models
    - One and two equation models
  - Non-linear eddy viscosity models and algebraic stress models
  - Reynolds stress transport models
- Large eddy simulations
- Detached eddy simulations and other hybrid models

In this tutorial, RANS-based model is explained in detail. In the next tutorial, large eddy simulations (LES) and Smagorinsky-Lilly model will be covered.

2. RANS-based models

The governing equations for a Newtonian fluid are:

- Conservation of mass

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \tilde{\boldsymbol{u}}) = 0$$

- Conservation of momentum (Navier-Stokes equation)

$$\frac{\partial (\rho \tilde{u}_i)}{\partial t} + \nabla \cdot (\rho \tilde{u}_i \tilde{\boldsymbol{u}}) = -\frac{\partial \tilde{p}}{\partial x_i} + \nabla \cdot (\mu \nabla \tilde{u}_i) + \tilde{S}_{Mi}$$

- Conservation of passive scalars (given a scalar $\tilde{e}$)

$$\frac{\partial(\rho\tilde{e})}{\partial t} + \nabla\cdot(\rho\tilde{e}\tilde{\boldsymbol{u}}) = \nabla\cdot\left(k\nabla\tilde{T}\right) + \tilde{S}_e$$

Note: suffix notation is used in the conservation of momentum equation for simplicity, with $i = 1$ corresponding to the x-direction, $i = 2$ the y-direction and $i = 3$ the z-direction.

One of the solutions to the problem is to reduce the number of scales (from infinity to 1 or 2) by using the Reynolds decomposition. Any property (whether a vector or a scalar) can be written as the sum of an average and a fluctuation, i.e. $\tilde{\varphi} = \Phi + \varphi$ where the capital letter denotes the average and the lower case letter denotes the fluctuation of the property. Using the Reynolds decomposition in the Navier-Stokes equations, we obtain RANS or Reynolds Averaged Navier Stokes Equations.

- Average conservation of mass

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{U}) = 0$$

- Average conservation of momentum

$$\frac{\partial(\rho\mathrm{U}_i)}{\partial t} + \nabla\cdot(\rho\mathrm{U}_i\mathbf{U}) = -\frac{\partial P}{\partial x_i} + \nabla\cdot(\mu\mathrm{U}_i) - \left(\frac{\partial(\rho\overline{uu_i})}{\partial x} + \frac{\partial(\rho\overline{vu_i})}{\partial y} + \frac{\partial(\rho\overline{wu_i})}{\partial z}\right) + \mathrm{S}_{Mi}$$

- Average conservation of passive scalars (given a scalar $\tilde{e}$)

$$\frac{\partial(\rho\mathrm{E})}{\partial t} + \nabla\cdot(\rho\mathrm{EU}) = \nabla\cdot(k\nabla\mathrm{T}) - \left(\frac{\partial(\rho\overline{ue})}{\partial x} + \frac{\partial(\rho\overline{ve})}{\partial y} + \frac{\partial(\rho\overline{we})}{\partial z}\right) + \mathrm{S}_e$$

Note: a special property of the Reynolds decomposition is that the average of the fluctuating component is identically zero, a fact that is used in the derivation of the above equations.

However, by using the Reynolds decomposition, there are new unknowns that were introduced such as the turbulent stresses ($\rho\overline{uu}$, $\rho\overline{vu}$, $\rho\overline{wu}$, $\rho\overline{uv}$, $\rho\overline{vv}$, $\rho\overline{wv}$, $\rho\overline{uw}$, $\rho\overline{vw}$, $\rho\overline{ww}$) and turbulent fluxes ($\rho\overline{ue}$, $\rho\overline{ve}$, $\rho\overline{we}$) and therefore, the RANS equations describe an open set of equations (where the over bar denotes an average). The need for additional equations to model the new unknowns is called Turbulence Modeling.

We now have 9 additional unknowns (6 Reynolds stresses and 3 turbulent fluxes). In total, for the simplest turbulent flow (including the transport of a scalar passive scalar, e.g. temperature when heat transfer is involved) there are 14 unknowns (include *u, v, w, p, T*)!

One possible approach to model the additional unknowns is to use the PDEs for the turbulent stresses and fluxes as a guide to modeling. The turbulent models are as follows, in order of increasing complexity:

- Algebraic (zero equation) models: mixing length (first order model)
- One equation models: k-model, μ_t-model (first order model)
- Two equation models: k-ε, k-kl, k-ω, low Re k-ε (first order model)
- Algebraic stress models: ASM (second order model)

- Reynolds stress models: RSM (second order model)
- Zero-Equation Models

In OpenFOAM®, there are two simulation types for turbulence flow, RAS and LES. As the name suggest, the RAS simulation is based on the RANS-based models covered above and will be the sole focus of this tutorial. In the next tutorial, we will move on to LES modeling and compare the results generated from these two modeling types.

incompressibleFluid – pitzDaily

Tutorial outline

Use incompressibleFluid solver, run a steady state simulation with following turbulence models:

- kEpsilon (RAS)
- kOmega (RAS)

Objectives

- Understanding turbulence modeling
- Understanding steady state simulation

Data processing

Show the results of U and the turbulent viscosity in two separate contour plots.

1. Pre-processing

1.1. Copying tutorial

Copy the following tutorial to your working directory:

$FOAM_TUTORIALS/incompressibleFluid/pitzDaily

Replace the system directory with the system directory from the following tutorial:

$FOAM_TUTORIALS/incompressibleFluid/pitzDailySteadyExperimentalInlet

Copy the *pitzDaily* file for the pitzDaily geometry from following directory to your system directory:

$FOAM_TUTORIALS/resources/blockMesh

1.2. 0 directory

When a turbulent model is chosen, the value of its constants and its boundary values should be set in the appropriate files. For example in kEpsilon model the k and epsilon files should be edited. See below for the epsilon file (in the 0 folder):

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

dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 14.855;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 14.855;
    }
    outlet
    {
        type            zeroGradient;
    }
    upperWall
    {
        type            epsilonWallFunction;
        value           uniform 14.855;
    }
    lowerWall
    {
        type            epsilonWallFunction;
        value           uniform 14.855;
    }
    frontAndBack
    {
        type            empty;
    }
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

Note: Here is a list of files, which should be available at 0 directory and need to be modified for each turbulence model:

- laminar: no file
- kEpsilon (RAS): k and epsilon
- kOmega (RAS): k and omega
- LRR (RAS): k, epsilon and R
- Smagorinsky (LES): s
- kEqn (LES): k and s
- SpalartAllmaras (LES): nuSgs and nuTilda

Some files are available, e.g. epsilon, k and nuTilda, some files should be created by the user, e.g. R, nuSgs. Templates for these files can be also found in the examples of older versions of OpenFOAM®, e.g. 1.7.1.

1.3. constant directory

In the *momentumTransport* file, the `simulationType` can be set as either `RAS`, `LES` or `laminar`. Then the corresponding sub-dictionary of the chosen simulation type needs to be defined. In this case, the sub-dictionary for RAS contains information about the chosen RAS model (kEpsilon), and the status of `turbulence` and `printCoeffs` are turned to on. Setting the `turbulence` to `off/false` will turn the turbulence model off and perform a laminar simulation.

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

simulationType        RAS;

RAS
{
     model          kEpsilon;

     turbulence        on;

     printCoeffs       on;
}


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

Note: For Boolean inputs in OpenFOAM either on/off, 1/0 or true/false can be used.

1.4. system directory

Running simulations in steady state mode, the `endTime` in the *controlDict* file corresponds to maximum number of iterations (e.g. 1000) instead of time, `deltaT` is the iterator and should be 1, because it is the amount of increase in the iteration number and `writeInterval` will show the frequency of saving data (e.g. 50 means each 50 iterations a saving will be done).

In the *fvSchemes* files the `ddtSchemes` is set to `steadyState` which will set the time derivative part of conservation equations to zero which is compatible with the steady state assumption.

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

ddtSchemes
{
      default          steadyState;
}

gradSchemes
{
      default          Gauss linear;
}

divSchemes
{
      default          none;
      div(phi,U)       bounded Gauss Upwind;
      div(phi,k)       bounded Gauss Upwind;
      div(phi,epsilon) bounded Gauss Upwind;
      div(phi,R)       bounded Gauss Upwind;
      div(R)           Gauss linear;
      div(phi,nuTilda) bounded Gauss Upwind;

      div((nuEff*dev2(T(grad(U))))  Gauss linear;
}

laplacianSchemes
{
      default      Gauss linear corrected;
}

interpolationSchemes
{
      default      linear;
}

snGradSchemes
{
      default      corrected;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

In case we are solving for property such as *omega* and it is not defined in the schemes and there is no default schemes defined, it should be add to the relevant schemes section, e.g. to the divSchemes (div(phi,omega)). Also, *fvSolution* needs to be adopted based on the new parameters, e.g. omega.

Note: In the fvSolution the solver type and settings need to be defined or be added to the others, e.g. for omega `"(U|k|epsilon|R|nuTilda|omega)"`*.*

2. Running simulation

```
>blockMesh -dict system/pitzDaily
```

Note: The default dictionary file for blockMesh is blockMeshDict in the system directory. It is possible to use a different dictionary file by using the flag "-dict" and the address of the file (dictionary), in this case system/pitzDaily.

```
>foamRun -solver incompressibleFluid
```

Note: When the solution converges, "`SIMPLE solution converged in … iterations`" message will be displayed in the Shell window. If nothing happens and you do not see a message after a while (this is not the case in here, it converges after a short time), then you should check the residuals which are displayed in the Shell window manually (you should check `initial residual` values, it shows the difference between this iteration and the last one), if all of the `Initial residual` (see below) values are close to amounts you have set in the fvSolution then you can stop simulation (ctrl+c).

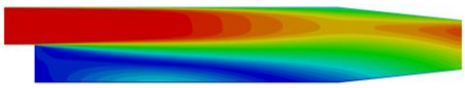
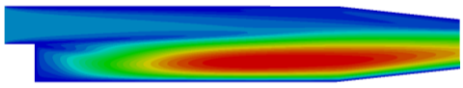
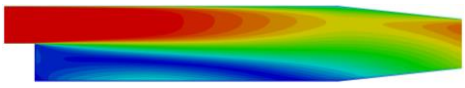
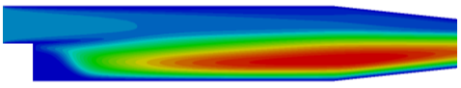
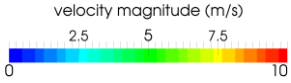
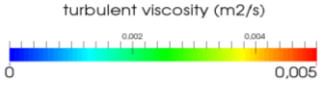
Note: You can use bash script and gnuPlot for extracting the residual data from log files and plotting them. For saving the simulation output to a log file use the following command for running the simulation and the terminal output will be saved to the log file (log file can be viewed using less - check Appendix A).

```
>foamRun -solver incompressibleFluid > log
```

```
Time = 795s

smoothSolver:  Solving for Ux, Initial residual = 0.00013831, Final residual =
9.28001e-06, No Iterations 6
smoothSolver:  Solving for Uy, Initial residual = 0.000977894, Final residual =
6.73868e-05, No Iterations 6
GAMG:  Solving for p, Initial residual = 0.00192871, Final residual =
0.000174838, No Iterations 7
time step continuity errors : sum local = 0.000840075, global = 6.13868e-05,
cumulative = -0.193739
smoothSolver:  Solving for epsilon, Initial residual = 0.000175322, Final
residual = 1.138e-05, No Iterations 2
smoothSolver:  Solving for k, Initial residual = 0.000404928, Final residual =
2.99083e-05, No Iterations 2
ExecutionTime = 56.7 s  ClockTime = 57 s

SIMPLE solution converged in 795 iterations
```

3. Post-processing

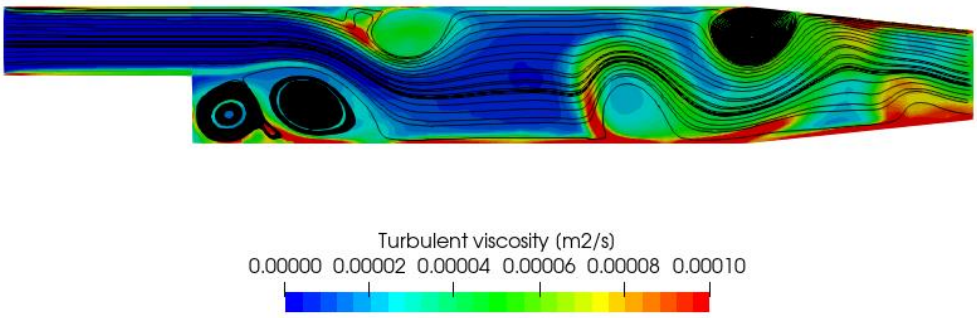
The simulation results are as follows (all simulations scaled to the same range):

RAS model	Velocity magnitude	Turbulent viscosity
kEpsilon		
kOmega		
		

Velocity magnitude and turbulent viscosity for different RAS models

Tutorial Seven

Turbulence - Transient



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Large eddy simulation (LES)

In Large Eddy Simulation (LES), turbulence is modeled by distinguishing between large-scale eddies and small-scale eddies within a fluid flow. The fundamental idea behind LES is that large eddies are dependent on the geometry and flow conditions, whereas small eddies exhibit more universal behavior. This assumption allows for a computationally efficient approach to turbulence modeling by resolving only the large eddies while modeling the small-scale eddies using Sub-Grid Scale (SGS) models.

Compared to Reynolds-Averaged Navier-Stokes (RANS) models, which completely model turbulence effects, LES provides a higher-fidelity simulation since large eddies are explicitly resolved rather than approximated. However, LES requires higher computational resources than RANS but significantly less than Direct Numerical Simulation (DNS), making it an effective trade-off between accuracy and computational feasibility.

Mathematically, it is like separating the velocity field into a resolved and sub-grid part using a filter function. The resolved part of the field represents the large eddies, while the sub grid part of the velocity represents the small eddies whose effect on the resolved field is included through the sub grid-scale model. Formally, one may think of filtering as the convolution of a function with a filtering kernel G:

$$\bar{u}_i(\vec{x}) = \int G(\vec{x} - \vec{\xi}) u(\vec{\xi}) d\vec{\xi}$$

resulting in

$$u_i = \bar{u}_i + u'_i$$

Where $\bar{u}_i$ is the resolvable scale part and u'_i is the subgrid-scale part. However, most practical (and commercial) implementations of LES use the grid itself as the filter and perform no explicit filtering. The filtered equations are developed from the incompressible Navier-Stokes equations of motion:

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\upsilon \frac{\partial u_i}{\partial x_j}\right)$$

Substituting in the decomposition $u_i = \bar{u}_i + u'_i$ and $\mathrm{p} = \bar{\mathrm{p}} + \mathrm{p}'$ and then filtering the resulting equation gives the equations of motion for the resolved field:

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\upsilon \frac{\partial \bar{u}_i}{\partial x_j}\right) + \frac{1}{\rho}\frac{\partial \tau_{ij}}{\partial x_j}$$

We have assumed that the filtering operation and the differentiation operation commute, which is not generally the case. It is thought that the errors associated with this assumption are usually small, though filters that commute with differentiation have been developed. The extra term $\partial \tau_{ij}/\partial x_j$ arises from the non-linear advection terms, because:

$$u_j \frac{\partial u_i}{\partial x_j} \neq \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j}$$

and hence

$$\tau_{ij} = \bar{u}_i\bar{u}_j - \overline{u_i u_j}$$

Similar equations can be derived for the sub grid-scale field. Sub grid-scale turbulence models usually employ the Boussinesq hypothesis, and seek to calculate (the deviatoric part of) the SGS stress using:

$$\tau_{ij} - \frac{1}{3}\tau_{kk}\delta_{ij} = -2\mu_t\bar{S}_{ij}$$

where $\bar{S}_{ij}$ is the rate-of-strain tensor for the resolved scale defined by

$$\bar{S}_{ij} = \frac{1}{2}\left(\frac{\partial\bar{u}_i}{\partial x_j} + \frac{\partial\bar{u}_j}{\partial x_i}\right)$$

and μ_t is the subgrid-scale turbulent viscosity. Substituting into the filtered Navier-Stokes equations, we then have:

$$\frac{\partial\bar{u}_i}{\partial t} + \bar{u}_j\frac{\partial\bar{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial\bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left([\upsilon + \upsilon_t]\frac{\partial\bar{u}_i}{\partial x_j}\right)$$

where we have used the incompressibility constraint to simplify the equation and the pressure is now modified to include the trace term $\tau_{kk}\delta_{ij}/3$.

2. k-Eqn model

The k-equation (kEqn) Large Eddy Simulation (LES) turbulence model is designed to capture sub-grid scale (SGS) turbulence effects by solving a transport equation for the SGS turbulence kinetic energy (k). This approach enhances the accuracy of simulations involving complex turbulent structures.

The governing equation for the SGS turbulence kinetic energy k in the kEqn LES model is:

$$\frac{\partial\rho k}{\partial t} + \nabla\cdot(\rho\, u\, k) = \nabla\cdot(\rho\, D_k\nabla k) + \rho G - \frac{2}{3}\rho k\,(\nabla\cdot u) - \frac{C_e\,\rho\, k^{1.5}}{\Delta} + S_k$$

Where G is the production term of turbulence kinetic energy, C_e is the model coefficient (default value: 1.048), Δ is the filter width and S_k is the source term.

This equation accounts for the transport, production, and dissipation of SGS turbulence kinetic energy, providing a comprehensive representation of turbulent flow dynamics.

3. Smagorinsky-Lilly model

One of the most widely used SGS models is the Smagorinsky-Lilly model, which provides a simple way to estimate the sub-grid scale eddy viscosity:

$$\tau_{ij} - \frac{1}{3}\tau_{kk}\delta_{ij} = -2(C_s\Delta)^2|\bar{S}|S_{ij}$$

In the Smagorinsky-Lilly model, the eddy viscosity is modeled by

$$\mu_{sgs} = \rho(C_s\Delta)^2|\bar{S}|$$

Where the filter width is usually taken to be

$$\Delta = (Volume)^{1/3}$$

and

$$\bar{S} = \sqrt{2S_{ij}S_{ij}}$$

The effective viscosity is calculated from

$$\mu_{eff} = \mu_{mol} + \mu_{sgs}$$

The Smagorinsky constant usually has the value: $C_s = 0.1 - 0.2$

Physical Interpretation

- The Smagorinsky model assumes that the energy cascade in turbulence is local, meaning small eddies interact mostly with nearby structures.
- The filter width Δ\Delta determines the size of the smallest resolved structures.
- The Smagorinsky constant C_s is a tunable parameter that affects model accuracy.
    - Higher C_s leads to stronger damping of small eddies.
    - A lower C_s may lead to unresolved turbulence effects.

Smagorinsky model is simple and computationally efficient, while providing reasonable approximations for turbulent energy dissipation, it works well for high-Reynolds-number flows. On the other hand, the model does not account for near-wall effects accurately, leading to overdamping of turbulence in boundary layers and the constant C_s needs tuning for different flow conditions.

incompressibleFluid – pitzDaily

Tutorial outline

Use the incompressibleFluid solver, run a backward facing step case for 0.2 s with different turbulence models:

- Smagorinsky (LES)
- kEqn (LES)
- kEpsilon (RAS)

Objectives

- Understanding turbulence models
- Transient vs steady state simulation
- Finding appropriate turbulence model

Data processing

Display the results of U and the turbulent viscosity in two separate contour plots at three different time steps. Compare with steady state simulation (Tutorial Six).

1. Pre-processing

1.1. Copying tutorial

Copy the tutorial from the following directory to your working directory:

```
$FOAM_TUTORIALS/incompressibleFluid/pitzDailyLESDeveloped
Inlet
```

Replace 0 directory with 0 directory from the following tutorial:

```
$FOAM_TUTORIALS/incompressibleFluid/pitzDaily
```

1.2. 0 directory

Set the proper turbulence model initial and boundary conditions and values.

Note: For different turbulent models, different files should be modified (check Tutorial Six).

For kEpsilon model, the epsilon file need to be added and on the walls, for all three properties: k, epsilon and nut, the wall-functions should be applied (based on the y^+ value) and proper initial values to be set. For more information: https://www.openfoam.com/documentation/guides/latest/doc/guide-turbulence.html

1.3. constant directory

As mentioned in Tutorial Six, in *momentumTransport* the turbulent model type has to be set. The `simulationType` can be changed to `LES` or `RAS`. Depending on which type is selected, the corresponding sub-dictionary needs to be specified. Below is the *momentumTransport* file for the kEqn model, which is an LES model.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
simulationType  LES;

LES
{
LESModel        kEqn;
turbulence      on;
printCoeffs     on;
delta           cubeRootVol;

dynamicKEqnCoeffs
{
    filter      simple;
}

cubeRootVolCoeffs
{
    deltaCoeff      1;
}

PrandtlCoeffs
{
    delta           cubeRootVol;
    cubeRootVolCoeffs
    {
```

```
        deltaCoeff      1;
    }

    smoothCoeffs
    {
        delta           cubeRootVol;
        cubeRootVolCoeffs
        {
            deltaCoeff      1;
        }

        maxDeltaRatio   1.1;
    }

    Cdelta          0.158;
}

vanDriestCoeffs
{
    delta           cubeRootVol;
    cubeRootVolCoeffs
    {
        deltaCoeff      1;
    }

    smoothCoeffs
    {
        delta           cubeRootVol;
        cubeRootVolCoeffs
        {
            deltaCoeff      1;
        }

        maxDeltaRatio   1.1;
    }

    Aplus           26;
    Cdelta          0.158;
}

smoothCoeffs
{
    delta           cubeRootVol;
    cubeRootVolCoeffs
    {
        deltaCoeff      1;
    }

    maxDeltaRatio   1.1;
}
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
* * * * * * *//
```

Note: For Smagorinsky, you can find the sample dictionary, including the relevant settings in the following:

`$FOAM_TUTORIALS/multiphaseEuler/LBend/constant/momentumTrasport.gas`

2. Running simulation

```
>blockMesh
```

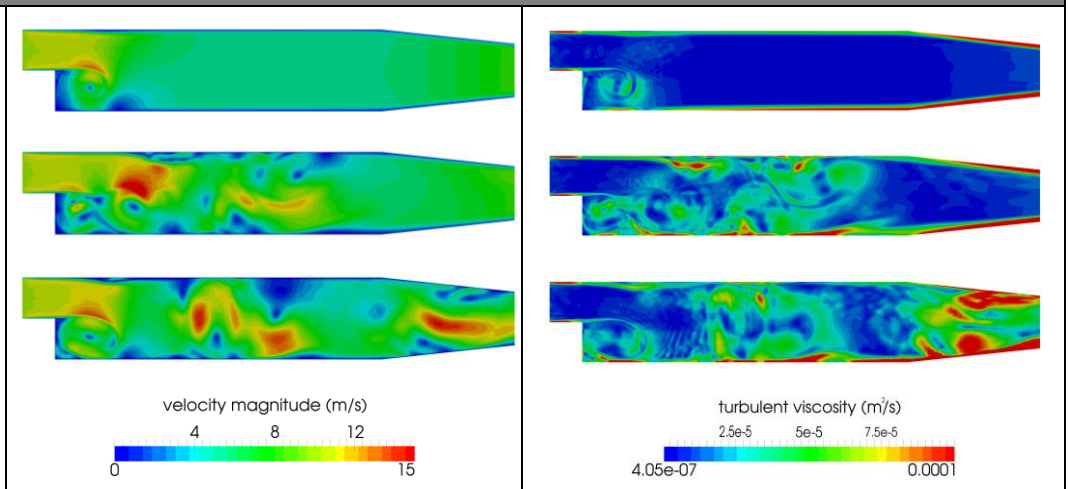
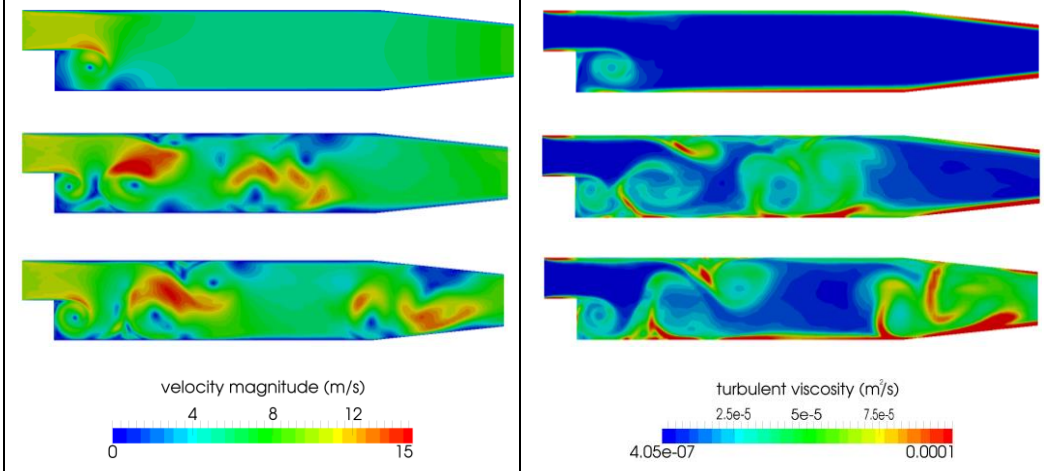
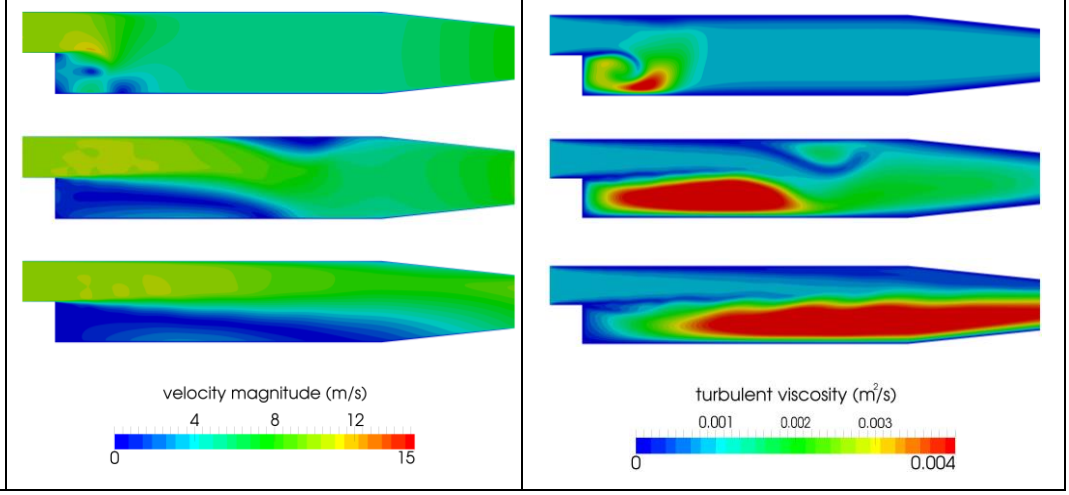
```
>foamRun -solver incompressibleFluid
```

3. Post-processing

The simulation results are as follows:

For the kEpsilon model after 0.2 s the results are similar to the steady state simulation. Therefore, it can be assumed it has reached the steady state. Other models do not have a steady situation and are fluctuating all the time, so they require averaging for obtaining steady state results.

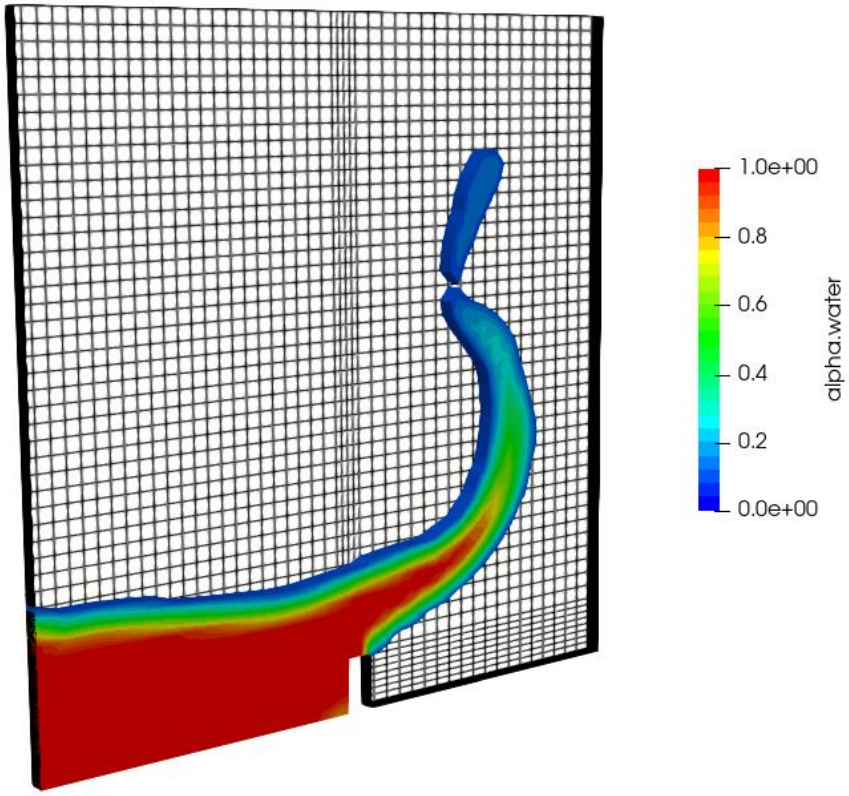
kEpsilon and other RAS models use averaging to obtain the turbulence values, but LES does not include any averaging by default. Therefore, LES simulations should use a higher grid resolution (smaller cells) and smaller time steps (for reasonable Co number). Contour plots or other LES results should be presented time averaged over reasonable number of time steps (not done in this tutorial).

	Velocity magnitude	Turbulent viscosity
Smagorinsky		
0.01 s		
0.05 s		
0.2 s		
kEqn		
0.01 s		
0.05 s		
0.2 s		
kEpsilon		
0.01 s		
0.05 s		
0.2 s		

Comparison of different turbulent models for transient simulation.

Tutorial Eight

Multiphase



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Multiphase flow

Multiphase flow refers to the simultaneous movement of two or more different phases (solid, liquid, or gas). Each phase may contain one or more chemical components. The common types of multiphase flows include:

- Gas-liquid flows (e.g., air bubbles in water, boiling liquids)
- Gas-solid flows (e.g., pneumatic transport, fluidized beds)
- Liquid-solid flows (e.g., sediment transport, slurry flows)
- Liquid-liquid flows (e.g., oil-water mixtures)
- Three-phase flows (e.g., water, gas, and solid mixtures in oil pipelines)

Multiphase flow can be further classified based on the flow regime:

- Separated flow: Distinct interfaces between phases (e.g., water flowing under oil).
- Dispersed flow: One phase exists as discrete particles or droplets within another continuous phase (e.g., bubbles in water, oil droplets in water).
- Mixed flow: A combination of separated and dispersed flow regions.

Multiphase flow is found in numerous applications, including chemical reactors (bubble columns, distillation towers), oil and gas industries (flow through pipelines, separation processes), environmental science (rain formation, erosion due to sediment transport) and power plants (boiling in nuclear reactors, cooling systems). Understanding and modeling multiphase flow is crucial for improving efficiency, design optimization, and operational safety in these applications.

2. Modeling approaches

Modeling of multiphase flow can be extremely complex, due to possible flow regime transitions. To simplify the matter, different modeling approaches can be adopted and they generally fall into two categories: lagrangian and Eulerian. In the case of dispersed configuration, Lagrangian approach is more suitable. This involves tracking individual point particles during its movement. The other approach is the Eulerian approach, which observes fluid behavior in a given control volume. Below we will cover some common modeling approaches of multiphase flow.

2.1. Euler-Euler approach (Multi-fluid model)

All phases are treated as continuous in the Euler-Euler approach. This approach is suitable for separated flows where each phase behaves as a continuum, rather than being discrete. The phases interact through the drag and lift forces acting between them, as well as through heat and mass transfer. The Euler-Euler approach is also capable of modeling dispersed flow, where

we are interested in the overall motion of particles rather than tracking individual particles.

In the Euler-Euler approach, we introduce the concept of phasic volume fractions. These fractions are assumed continuous functions of space and time, with their sum equal to one. For each phase, a set of conservation equations for mass, momentum and energy is solved individually; in addition, a transport equation for the volume fraction is solved. Coupling between the phases is achieved through shared pressure and interphase exchange coefficients.

2.2. Eddy Interaction Model

In the Eddy Interaction Model, each particle interacts with a succession of eddies. The fluid motion of the particle is characterized by three parameters: i) eddy velocity, ii) eddy lifetime, iii) eddy length. It follows the particle-tracking Lagrangian approach.

The eddy lifetime (t_e) and eddy length scale (l_e) are estimated from the local turbulence properties. From the length scale and the particle velocity, one can calculate the eddy transit time (t_c), i.e. the time taken for a particle to cross the eddy. The particle is then assumed to interact with the eddy for a time, which is the minimum of the eddy lifetime and the eddy transit time.

$$t_{int} = \min(t_e, t_c)$$

During that interaction, the fluid fluctuating velocity is kept constant and the discrete particle is moved with respect to its equation of motion. Then a new fluctuating fluid velocity is sampled and the process is repeated.

2.3. Volume of Fluid (VOF) method

VOF method belongs to the Eulerian class of modeling approach. It is based on the idea of **fraction function C**. Fraction function indicates whether a chosen phase is present inside the control volume. If C=1, the control volume is completely filled with the chosen phase; if C=0, the control volume is filled with a different phase. A value between 0 and 1 indicates that the interface between phases is present inside the control volume. It is important in VOF method that the flow domain is modeled on a fine grid, since the interface should be resolved.

The focus of the VOF method is to track the interface between phases. To do this, the transport equations are solved for mixture properties, assuming that all field variables are shared between the phases. Then an advection equation for the fraction function C is solved. The discretization of the fraction function equation is crucial for obtaining a sharp interface.

incompressibleVoF – damBreak

Tutorial outline

Use the incompressibleVoF solver to simulate breaking of a dam for 2s.

Objectives

- Understanding how to set viscosity, surface tension and density for two phases

Data processing

See the results in ParaView.

1. Pre-processing

1.1. Copying tutorial

Copy tutorial from the following folder to your working directory:

`$FOAM_TUTORIALS/incompressibleVoF/damBreak`

1.2. 0 directory

In the 0 directory, in the alpha.water.orig and p_rgh files, the initial values and boundary conditions for water phase and pressure are set. Copy alpha.water.orig to alpha.water (remember: the *.orig files are back up files, and solvers do not use them). E.g. in alpha.water:

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

dimensions      [];

internalField   uniform 0;

boundaryField
{
    #includeEtc  "caseDicts/setConstrainTypes"

    wall
    {
        type            zeroGradient;
    }

    atmosphere
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * *//
```

`#includeEtc` includes the *etc* folder in the OpenFOAM installation directory and the path `"caseDicts/setConstrainTypes"` instruct it to include the boundaries in this file also here. In this case the empty boundary is used for this tutorial.
Checking the *blockMeshDict* file there is no boundary named wall, the `wall` in the `boundaryField` section of files in the *0* is for using the same boundary condition for the boundaries which have type wall.

Note: If in the files in 0 directory some not used boundaries are defined, as far as their syntax is correct, OpenFOAM will ignore them, so you don't need to remove them!

Note: In the `dimensions` section `[]` is equal to `[0 0 0 0 0 0 0]` and it means it is a dimensionless parameter.

Note: The `inletOutlet` and the `outletInlet` boundary conditions are used when the flow direction is not known. In fact, these are derived types and are a combination of two different boundary types.

- `inletOutlet`: When the flux direction is toward the outside of the domain, it works like a zeroGradient boundary condition and when the flux is toward inside the domain it is like a fixedValue boundary condition.
- `outletInlet`: This is the other way around, if the flux direction is toward outside the domain, it works like a fixedValue boundary condition and when the flux is toward inside the domain, it is like a zeroGradient boundary condition.

E.g. if the velocity field outlet is set as `inletOutlet` and the `inletValue` is set to `(0 0 0)`, it avoids backflow at the outlet! The “`inletValue`” or “`outletValue`” are values for `fixedValue` type of these boundary conditions and “`value`” is a dummy entery for OpenFOAM® for finding the variable type. Using `(0 0 0)`, OpenFOAM® understands that the variable is a vector.

1.3. constant directory

In the file phaseProperties, there is the list of phases in the simulation (in this case air and water):

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

Phases          (water air);

sigma           0.07;

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

and `sigma` is the surface tension between two phases, in this example it is the surface tension between air and water.

For each phase, there is a dedicated *physicalProperties.fileName* file, in which the properties of the relevant phase can be set. E.g. the *physicalProperties.water* file looks as following:

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

viscosityModel  constant;
nu              1e-06;
rho             1000;

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

In both phases the coefficients for different models of viscosity are given, e.g. `nu` and `rho`. Depending on which model is selected, the corresponding coefficients are read. In this simulation, the selected model is `constant` (representing Newtonian model), therefore only the `nu` coefficient is needed.

Checking the g file, the gravitational field and its direction are defined, it is `9.81` m/s^2 in the negative y direction.

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

dimensions      [0 1 -2 0 0 0 0];
value           ( 0 -9.81 0 );
```

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

1.4. system directory

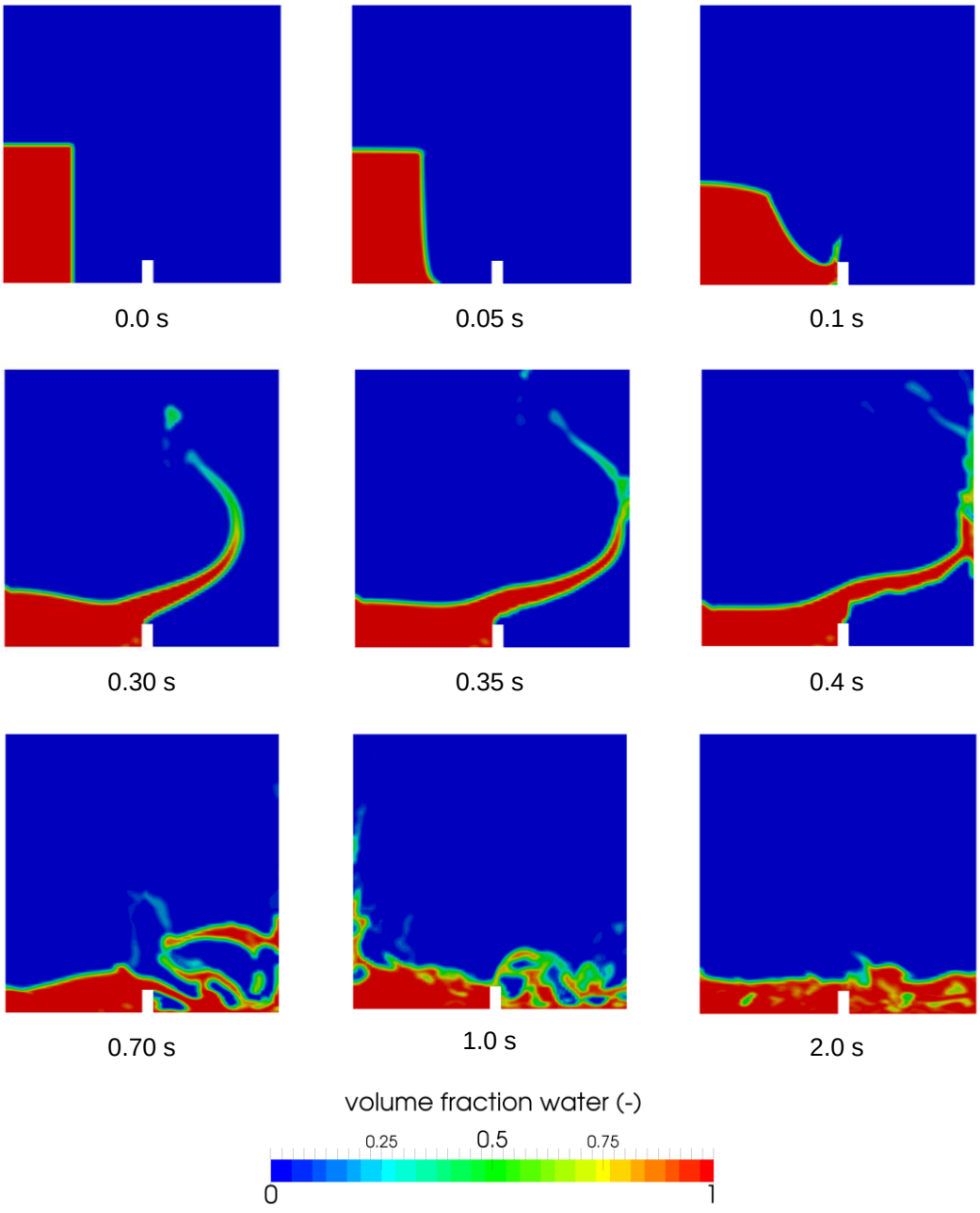
In the *controlDict* change the `endTime` to `2`, for 2s of simulation.

2. Running simulation

```
>blockMesh

>setFields

>foamRun -solver incompressibleVoF
```

3. Post-processing

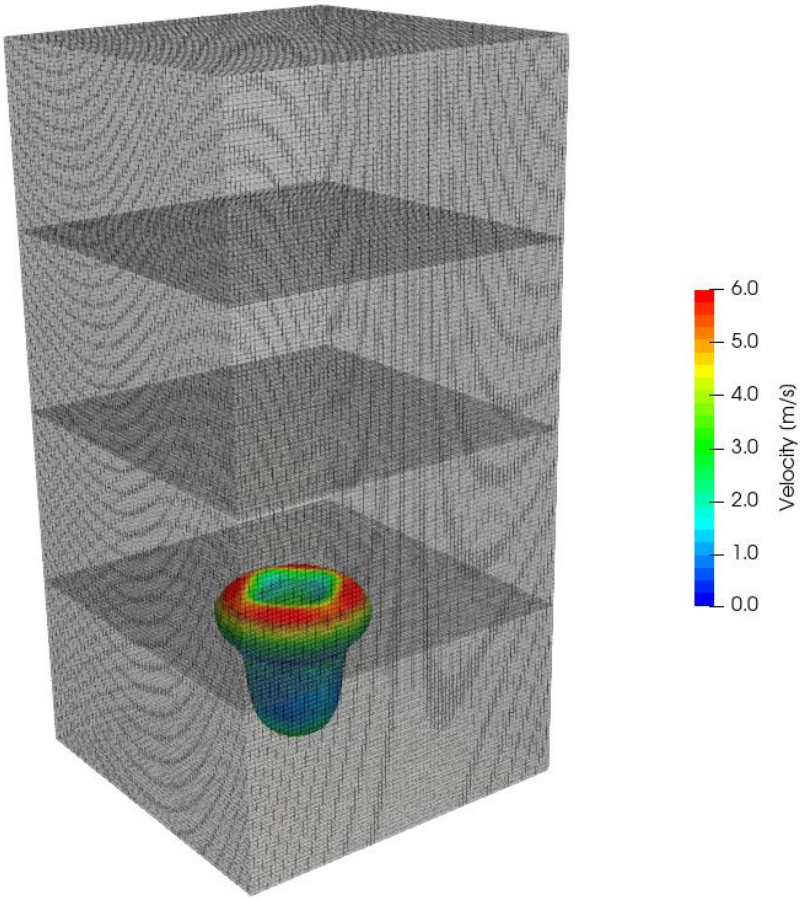
The simulation results are as follows (these are not the results for the original mesh, but a 2x refined mesh):



Contours of the water volume fraction at different time steps

Tutorial Nine

Parallel Processing



Bahram Haddadi

ENGINEERING
FLUID
DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

1. Parallel computing

Modern Computational Fluid Dynamics (CFD) problems often involve complex geometries, turbulence, multiphase flows, and chemical reactions, making simulations computationally expensive. Parallel computing reduces computation time by distributing the workload across multiple processors, enabling simulations that would otherwise be infeasible on a single machine.

Parallel computing involves dividing the computational domain into smaller sub-domains, which are assigned to different processors. This step is known as **domain decomposition**. These processors simultaneously perform calculations and communicate with each other to synchronize data exchange. This approach significantly reduces computational time, making it essential for large-scale simulations in engineering and scientific research.

Parallel computing can be carried out in two ways. One is done on a single computer with multiple internal processors, known as a **Shared Memory Multiprocessor**. The other way is achieved through a series of computers interconnected by a network, known as a **Distributed Memory Multicomputer**.

1.1. Shared versus distributed memory

	Shared Memory Multiprocessor	***Distributed Memory Multicomputer***
Memory	Data is saved in a global memory that can be accessed by all processors	Each computer has a local memory and a processor can only access its local memory
Data transfer between processors	The sender processor simply needs to write the data in a global variable and the receiver can read it	Message is sent explicitly from one computer to another using a message passing library, e.g. Message Passing Interface (MPI)

In OpenFOAM® the application of parallel computing can be executed using the *decomposePar* command. This allows the solver to be run on multiple processors. The workflow of parallel computation in OpenFOAM® is summarized below:

- Division of the mesh into sub-domains
- Running of the solver in parallel
- Reconstruction of the meshes and connecting the results.

2. Introduction to compressible flow

Until now, we have primarily considered incompressible fluid flows, where the density remains constant. However, in many real-world scenarios, fluid density varies significantly due to changes in pressure and temperature. This variation

in density makes the flow compressible, requiring specialized solvers and numerical methods.

A classic example of compressible flow is the flow through a converging-diverging nozzle, where fluid accelerates to supersonic speeds. Compressibility becomes dominant in flows when the Mach number is greater than about 0.3. The Mach number is defined as follows:

$$Ma = \frac{u}{c} = \frac{local\ velocity}{speed\ of\ sound}$$

Flow Classification Based on Mach Number:

- **Subsonic Flow (Ma < 0.3):** Flow is nearly incompressible, and density changes are negligible.
- **Transonic Flow (0.3 < Ma < 1.0):** Density variations begin to affect flow behavior.
- **Supersonic Flow (Ma > 1.0):** Shock waves and expansion waves appear, requiring advanced solvers.
- **Hypersonic Flow (Ma > 5.0):** Extremely high-speed flow, where thermal and chemical effects become significant.

When a fluid flow is compressible, temperature and pressure are affected strongly by variations in density. It is therefore important to consider the linkage between pressure, temperature and density in compressible flow, usually by applying an equation of state from thermodynamics, e.g. the ideal gas equation. More complex real-gas equations of state (e.g., Peng-Robinson, Van der Waals) may be required for high-pressure or reactive flows.

3. Compressible flow solvers

There are two general types of solution schemes for compressible flow: pressure-based and density-based.

3.1. Pressure-based solvers

This type of solver was historically derived from the solution approach used on incompressible flows. They solve for the primitive variables. The discretized momentum and energy equations are used to update velocities and energy. The pressure is obtained by applying a pressure-correction algorithm on the continuity and momentum equations. Density is then calculated from the equation of state.

3.2. Density-based solvers

Density-based solvers are suitable for solving the conserved variables. Similar to pressure-based solvers, the conversed velocity and energy terms are updated from the discretized momentum and energy equations. We can then solve for density from the continuity equation, afterwards we use the equation of state to update the pressure.

In general, density based solvers are more suitable for high-speed compressible flows with shocks. This is because density based solvers solve for conserved quantities across the shock, so the discontinuities will not affect the results.

compressibleVoF – depthCharge3D

Tutorial outline

Use the compressibleVoF solver, simulate the example case for 0.5 s.

Objectives

- Understanding the difference between incompressible and compressible solvers
- Understanding parallel processing and different discretization methods

Data processing

Investigate the results in ParaView.

1. Pre-processing

1.1. Copying tutorial

Copy the tutorial from following directory to your working directory:

`$FOAM_TUTORIALS/compressibleVoF/depthCharge3D`

1.2. 0 directory

In the 0 directory copy the *alpha.water.orig*, *p.orig, p_rgh.orig* and *T.orig* files to *alpha.water*, *p*, *p_rgh* and *T* respectively.

*Note: You can also skip copying the *.orig files, since running the setFields will do it for you.*

1.3. constant directory

Phases and common physical properties of the two phases are set in the *phaseProperties* file.

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

phases (water air);

sigma           0.07;

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

Individual phase properties are set in physicalProperties.phase files, e.g. physicalProperties.air.

1.4. system directory

The decomposeParDict file includes the parallel settings, such as the number of domains (partitions) and also how the domain is going to be divided into these subdomains for parallel processing.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//
numberOfSubdomains 4;

method          hierarchical;

simpleCoeffs
{
    n               ( 1 4 1 );
}

hierarchicalCoeffs
{
    n               ( 1 4 1 );
    order           xyz;
}

manualCoeffs
{
    dataFile        "";
}
```

```
distributed     no;

roots           ( );
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

`numberOfSubdomains` should be equal to the number of cores used. `method` should show the method to be used. In the above example, the case is simulated with the `hierarchical` method and `4` processors.

If the `simple` method is being used, the parameter `n` must be changed accordingly. The three numbers `(1 4 1)` indicate the number of pieces the mesh is split into in the x, y and z directions, respectively. Their multiplication result should be equal to `numberOfSubdomains`.

If the `hierarchical` method is being used, these parameters and the order in which the mesh should be split up in each direction should be provided.

If the `scotch` method is being used, then no user-supplied parameters are necessary except for the number of subdomains.

Note: In order to check the quality of the mesh, the checkMesh tool can be used (run it from main case directory). If the message "`Mesh OK`" is displayed – the mesh is fine and no corrections need to be done. If the mesh fails in one or more tests, try to recreate or refine the mesh for a better mesh quality (less non-orthogonally and skewness).

If non-orthogonal cells exist in a mesh, another option is using non-orthogonal corrections in the fvSolution file in the algorithm sub-dictionary (e.g. `PIMPLE` or `PISO`). Usually using `1` or `2` as `nNonOrthogonalCorrectors` is enough.

2. Running simulation

```
>blockMesh
```

```
>setFields
```

For running the simulation in parallel mode the computing domain needs to be divided into subdomains and a core should be assigned to each subdomain. This is done by following command:

```
>decomposePar
```

This decomposes the mesh according to the supplied instructions. One possible source of error is the product of the parameters in `n` does not match up to the number of the subdomains. This appears for the simple and hierarchical methods.

After executing this command four new directories will be made in the simulation directory (processor0, processor1, processor2 processor3), and each subdomain calculation will be saved in the respective processor directory.

Note: When the domain is divided to subdomains in parallel processing new boundaries are defined. The data should be exchanged with the neighbor boundary, which it is connected to in the main domain.

```
>mpirun -np <No of cores> solver -parallel > log
```

`<No of cores>` is the number of cores being used. `solver` is the solver for this simulation. For example, if `4` cores are desired, and the solver is compressibleInterFoam following command is used:

```
>mpirun -np 4 foamRun -solver compressibleVoF -parallel > log
```

`> log` is the filename for saving the simulation status data, instead of printing them to the screen. For checking the last information which is written to this file the following command can be used during the simulation running:

```
>tail -f  log
```

Note: Before running any simulation, it is important to run the top command (type the 'top' command in the terminal), to check the number of cores currently used on the machine. Check the load average. This is on the first line and shows the average number of cores being used. There are three numbers displayed, showing the load averages across three different time scales (one, five and 15 minute respectively).

Add the number of cores you plan to use to this number – and you will get the expected load average during your simulation. This number should be less than the total number of cores in the machine – or the simulation will be slowed or the machine will crash (if you run out of memory). If you are running on a multi user server it is recommended to leave at least a few cores free, to allow for any fluctuations in the machine load.

Note: top command execution can be interrupted by typing q (or ctrl+c)

The simulation can take several hours, depending on the size of the mesh and time step size.

3. Post-processing

For exporting data for post processing, at first all the processors data should be put together and a single combined directory for each time step was created. By executing the following command all the cores data will be combined and new directories for each time step will be created in the simulation main directory:

```
>reconstructPar
```

Convert the data to ParaView format:

```
>foamToVTK
```

Note: To do the reconstruction or foamToVTK conversion from a start time until an end time the following flags can be used:

```
>reconstructPar -time [start time name, e.g. 016]:[end time name, e.g. 020]
```

```
>foamToVTK -time [start time name, e.g. 016]:[end time name, e.g. 020]
```

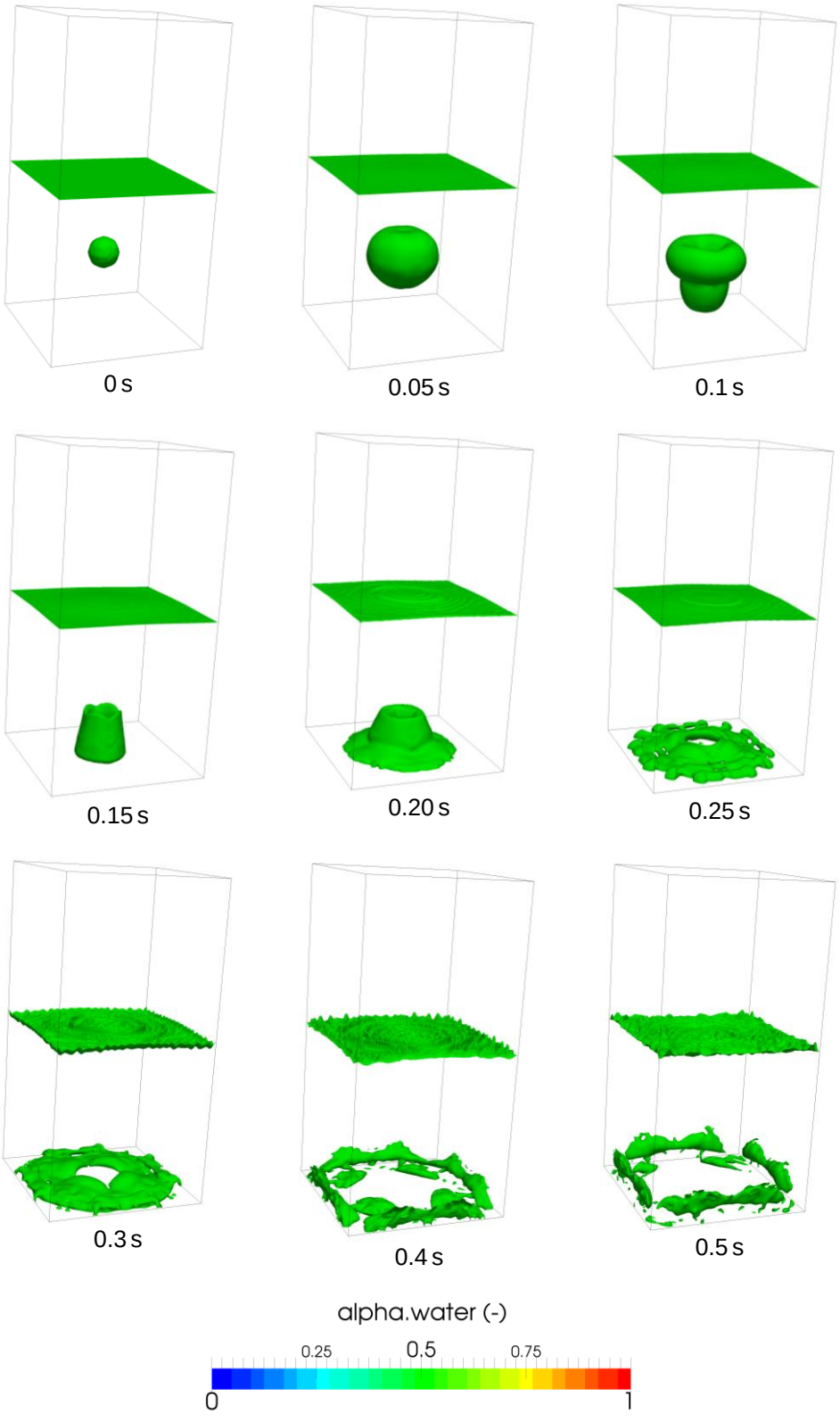
Using above commands without entering end time will do the reconstruction or conversion from start time to the end of available data:

```
>reconstructPar -time [start time name, e.g. 016]:

>foamToVTK -time [start time name, e.g. 016]:
```

For reconstructing or converting only one time step the commands should be used without end time and ":":

```
>reconstructPar -time [time name, e.g. 016]

>foamToVTK -time [time name, e.g. 016]
```

The simulation results are as follows:



3D depth charge, alpha = 0.5 iso-surfaces, parallel simulation

4. Manual method

4.1. Case set-up and running simulation

The manual method for decomposition is slightly different from the other three. In order to use it:

After running the blockMesh and setFields utilities, set the *decomposeParDict* file as any other simulation. For decomposition method, choose either simple, hierarchical or scotch. Set the number of cores to the same number which is going to be used for manual.

```
>decomposePar -cellDist
```

Once the decomposition is done, check the *cellDecomposition* file in the constant directory. It should have a format similar to:

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

1024000
(
0
0
0
0
0
0
0
0
0
0
0
0
1
1
1
1
1
1
1
1
1
1
1
1
1 ...)
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * *//
```

Note: If the above output is not displayed, but a stream of NUL characters, your text editor is probably printing binary. To fix this, open system/controlDict, and change the `writeFormat` field from `binary` to `asci` and rerun the previous command.

The first number `n` after the header, but before the opening brackets, `1024000` in this example, refers to the number of points in the mesh. Within the brackets, `n` lines follow. Each line contains one number between `0` and `n-1`, where `n` is the number of cores to be used for the computation. This number refers to the core being used to compute the corresponding cell in the points file in the constant directory. For example, if the second line in the points file brackets

reads (0.125 0 0) and the second line in the cellDecomposition directoy reads 0, this means that the cell (0.125 0 0) will be processed by processor 0.

This cellDecomposition file can now be edited. Although this can be done manually, it is probably not feasible for any sufficiently large mesh. The process must thus be automated by writing a script to populate the *cellDecomposition* file according to the desired processor breakdown.

When the new file is ready, save it under a different name:

```
>cp cellDecomposition manFile
```

Now, edit the decomposeParDict file. Select decomposition method manual, and for the dataFile field in the manual coeffs range, specify the path to the file which contains the manual decomposition. Note that OpenFOAM® searches in the constant directory by default, in case relative paths are being used:

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

numberOfSubdomains 4;

method          manual;

simpleCoeffs
{
    n               ( 1 4 1 );
    delta           0.001;
}

hierarchicalCoeffs
{
    n               ( 1 4 1 );
    delta           0.001;
    order           xyz;
}

manualCoeffs
{
    dataFile        "manFile";
}

distributed     no;

roots           ( );

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

Delete the old processor directories, decompose the case with the new decomposition settings and run the simulation.

4.2. Visualizing the processor breakdown

It may be interesting to visualize how exactly OpenFOAM® breaks down the mesh. This can be easily visualized using ParaView. After running the simulation, but before running the reconstructPar command, repeat the following for each of the processor directories:

```
>cd processor<n>
```

where `n` is the processor number

```
>foamToVTK
```

convert the individual processor files to VTK, next, open ParaView:

```
>paraview &
```

For each of the processor directories, perform the following steps:

- Open the VTK files in the relevant processor directory
- Double click them to open them and click on “Apply”
- The part of the mesh decomposed by that core will appear, in grey.
- Change the color in the drop-down menus in the toolbar. This is to ensure that each individual part can be easily seen.

Once this is done for all processors, the entire mesh will appear. However, the processor regions can now easily be seen in a different color.

In order to save this, there are two options. The first option is to take a screenshot:

File > Save a screenshot

The second option is to save the settings and modifications as a ParaView state file.

File > Save State

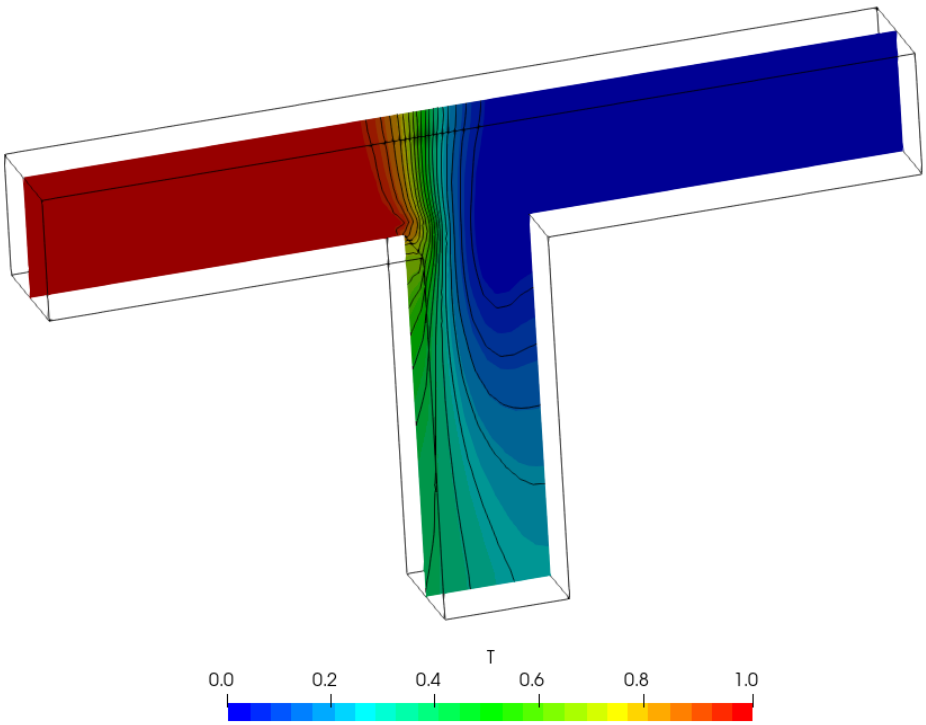
The current settings and modifications can then be easily recovered by:

File > Load State

Saving the state allows changes to be made afterwards. Saving a screenshot keeps only a picture, while losing the ability to make changes after exiting ParaView. Doing both is recommended.

Tutorial Ten

Residence Time Distribution



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

In this tutorial, we will carry out Residence Time Distribution (RTD) analysis of fluid flow through a T-junction pipe.

1. Residence Time Distribution (RTD)

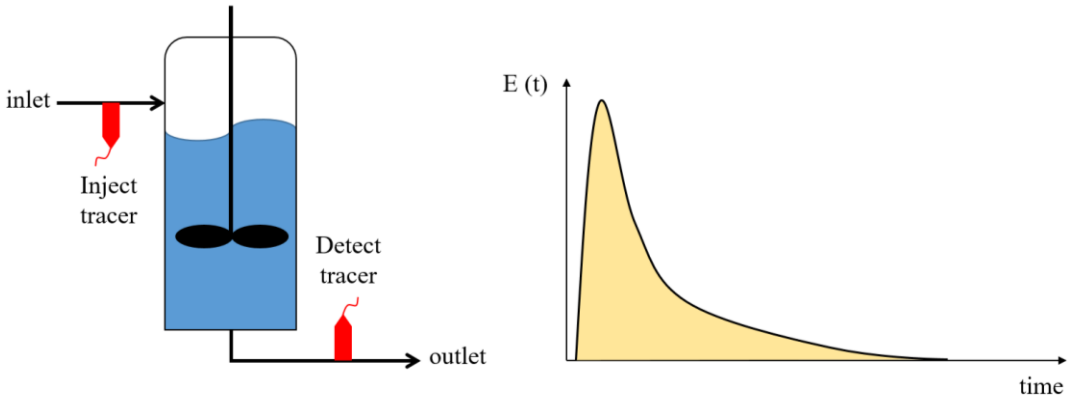
Residence Time Distribution (RTD) is a probability distribution function that describes the amount of time a fluid element spends in a process unit, such as a reactor, column, or pipe. Understanding RTD is crucial for analyzing the performance, efficiency, and mixing characteristics of industrial systems. Unlike ideal flow assumptions, most real-world fluid flows involve recirculation, bypassing, and dispersion, which RTD helps quantify. A few RTD applications:

- Optimizing reactor design by ensuring effective residence time for reactions.
- Identifying flow inefficiencies such as dead zones and short-circuiting.
- Enhancing mixing performance in industrial processes.
- Improving scale-up accuracy for chemical, pharmaceutical, and wastewater applications.

By understanding RTD, engineers can optimize designs, improve product yield, and enhance process reliability.

2. Tracer Analysis

Tracer analysis is a widely used technique for RTD measurement, where a tracer substance (such as dye, salt, or radioactive material) is injected into the system, and its concentration at the outlet is monitored over time. This provides insight into how fluid elements move through the process unit and allows engineers to quantify the RTD function.



Tracer analysis and RTD distribution of an ideal process

Based on the above diagram, first the tracer is injected into the inlet, and then the exit tracer concentration, $C(t)$, is measured at regular time intervals. This allows the exit age distribution, $E(t)$, to be calculated.

$$E(t) = \frac{C_T(t)}{\int_0^\infty C_T(t)\,dt} = \frac{Tracer\ concentration\ at\ time\ t}{Total\ tracer\ concentration}$$

It is clear from the above equation that the fraction of tracer molecules exiting the reactor that have spent a time between t and $t + dt$ in the process unit is $E(t)dt$. Since all tracer elements will leave the unit at some point, RTD satisfies the following relationship:

$$\int_0^\infty E(t)\,dt = 1$$

Types of Flow Patterns Identified by RTD

- Ideal Plug Flow: All fluid particles have the same residence time, resulting in a sharp, narrow RTD peak.
- Perfectly Mixed Flow (CSTR - Continuous Stirred Tank Reactor): Fluid particles experience a wide range of residence times, leading to a broad RTD distribution.
- Dead Zones and Recirculation: Cause multiple peaks in the RTD curve, indicating poor mixing and stagnation.
- Bypassing Flow: Results in a steep initial RTD rise, meaning some fluid exits much earlier than expected.

incompressibleFluid & functions – TJunction

Tutorial outline

Use the incompressibleFluid and functions to simulate the flow through a square cross section T pipe and calculate RTD (Residence Time Distribution) for both inlets using a step function injection:

- Inlet and outlet cross-sections: 1×1 m^2
- Gas in the system: air at ambient conditions
- Operating pressure: 10^5 Pa
- Inlet 1: 0.1 m/s
- Inlet 2: 0.2 m/s

Objectives

- Understanding RTD calculation using OpenFOAM®
- Using multiple solvers for a simulation

Data processing

Plot the step response function and the RTD curve.

1. Pre-processing

1.1. Copying tutorial

Copy the following tutorial to your working directory as a base case:

$FOAM_TUTORIALS/incompressibleFluid/pitzDaily

Replace the system directory with the system directory from the following tutorial:

$FOAM_TUTORIALS/incompressibleFluid/pitzDailySteadyExperimentalInlet

Copy the *pitzDaily* file for the pitzDaily geometry from following directory to your system directory:

$FOAM_TUTORIALS/resources/blockMesh

1.2. 0 directory

Update p, U, nut, nuTilda, k and epsilon files with the new boundary conditions (in this simulation the following boundaries should be set inlet_one, inlet_two, oulet and walls), e.g. for file *U*:

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

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    inlet_one
    {
        type            fixedValue;
        value           uniform (0.1 0 0)
    }
    inlet_two
    {
        type            fixedValue;
        value           uniform (-0.2 0 0)
    }
    outlet
    {
        type            zeroGradient;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0)

    }
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

1.3. constant directory

Check momentumTransport file for the turbulence model (`kEpsilon`).

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
simulationType  RAS
RAS
{
    model        kEpsilon;

    turbulence      on;

    printCoeffs     on;

}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

1.4. system directory

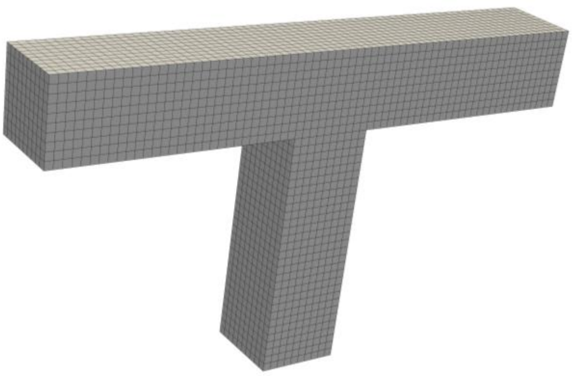
Rename the pitzDaily file to blockMeshDict and edit it to create the geometry.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
convertToMeters 1.0;

vertices
(
    (0 4 0) // 0
    (0 3 0) // 1
    (3 3 0) // 2
    (3 0 0) // 3
    (4 0 0) // 4
    (4 3 0) // 5
    (7 3 0) // 6
    (7 4 0) // 7
    (4 4 0) // 8
    (3 4 0) // 9
    (0 4 1) // 10
    (0 3 1) // 11
    (3 3 1) // 12
    (3 0 1) // 13
    (4 0 1) // 14
    (4 3 1) // 15
    (7 3 1) // 16
    (7 4 1) // 17
    (4 4 1) // 18
    (3 4 1) // 19

);
blocks
(
    hex (0 1 2 9 10 11 12 19) (10 30 10) simpleGrading (1 1 1)
    hex (9 2 5 8 19 12 15 18) (10 10 10) simpleGrading (1 1 1)
    hex (8 5 6 7 18 15 16 17) (10 30 10) simpleGrading (1 1 1)
    hex (2 3 4 5 12 13 14 15) (30 10 10) simpleGrading (1 1 1)
);
boundary
(
    inlet_one
    {
        type patch;
        faces
        (
            (0 10 11 1)
        );
    }
    inlet_two
    {
        type patch;
        faces
        (
            (7 6 16 17)
```

```
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (4 3 13 14)
        );
    }
    walls
    {
        type wall;
        faces
        (
            (0 1 2 9)
            (2 5 8 9)
            (5 6 7 8)
            (2 3 4 5)
            (10 19 12 11)
            (19 18 15 12)
            (18 17 16 15)
            (15 14 13 12)
            (0 9 19 10)
            (9 8 18 19)
            (8 7 17 18)
            (2 1 11 12)
            (3 2 12 13)
            (5 4 14 15)
            (6 5 15 16)
        );
    }
);

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

2. Running simulation

```
>blockMesh
```



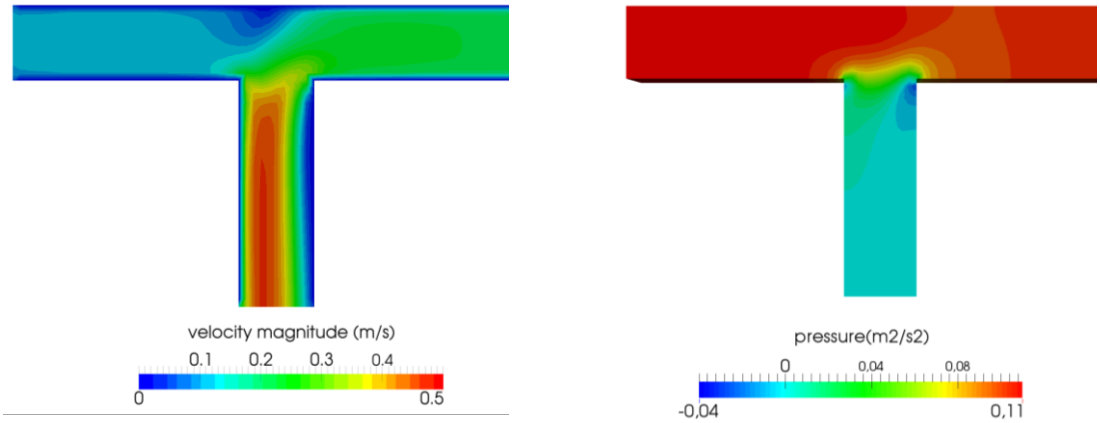
Mesh created using blockMesh

```
>foamRun -solver incompressibleFluid
```

Wait for simulation to converge. After convergence, check the results to make sure about physical convergence of the solution.

```
>foamToVTK
```

The simulation results are as follows (results are on the cut plane in the middle):



Simulation results after convergence (~65 iterations)

3. RTD calculation

3.1. Copy tutorial

Copy following tutorial to your working directory:

```
$FOAM_TUTORIALS/incompressibleFluid/pitzDailyScalarTransport
```

In the 0 directory, just keep the *T* file and delete all other files.

3.2. 0 directory

Copy and paste the *U* and *p* files from the latest time step of the simulation in the first part of the tutorial (use the latest time step velocity field from previous part of simulation to calculate RTD for this geometry). There is no need to modify or change it. The solver will use this field to calculate the scalar transportation.

Update *T* (T will be used as an inert scalar in this simulation) file boundary conditions to match new simulation boundaries, to calculate RTD of the `inlet_one` set the `internalField` value to `0`, T value for `inlet_one` to `1.0` and T value for `inlet_two` to `0`.

3.3. constant directory

In the *momentumTransport* file set the `simulationType` to `laminar`.

3.4. system directory

Copy the blockMeshDict file from the first part of tutorial.

In the controlDict file change the `endTime` from `0.2` to `120` (approximately two times ideal resistance time) and `deltaT` from `0.0001` to `0.1` (Courant number approximately 0.4).

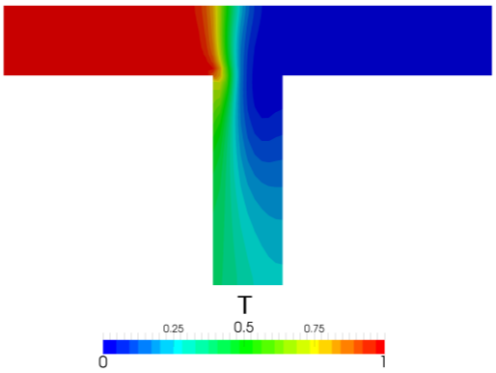
4. Running Simulation

```
>blockMesh
```

```
>foamRun -solver functions
```

```
>foamToVTK
```

5. Post-processing



Contour plots scalar T at 120 s for inlet 1

5.1. Calculating RTD

To calculate RTD the average T value at the outlets should be calculated first. The “integrate variables function” of ParaView can be used for this purpose.

```
>foamToVTK
```

Load the outlet VTK file into paraview using following path:

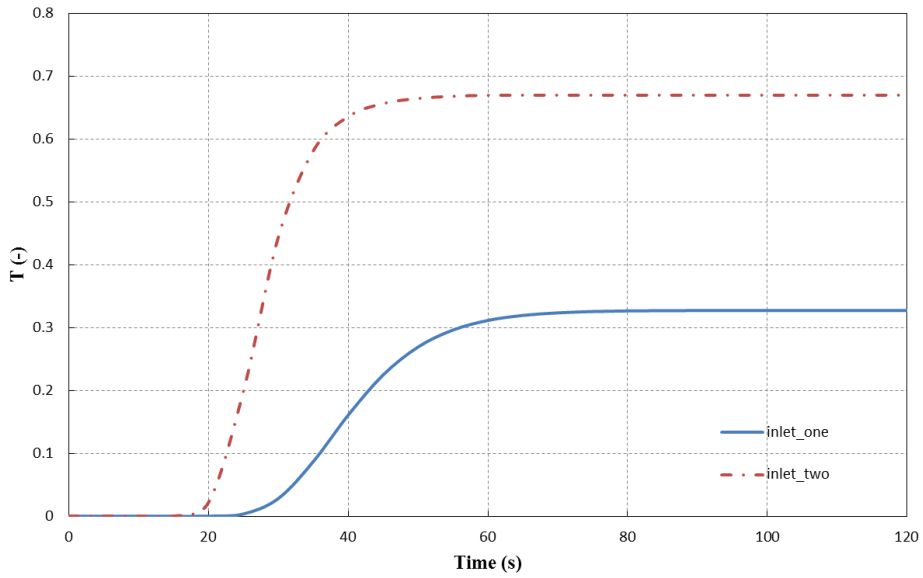
File > Open > VTK > outlet > outlet_..vtk > OK > Apply

Select T from variables menu, and then integrate the variables on the outlet:

Filters > Data Analysis > Integrate Variables > Apply

The values given in the opened window are integrated values in this specific time step. By changing the time step values for different time steps are displayed. As mentioned before, the average value of the property is needed. Therefore, these values should be divided by outlet area to get average values (1m × 1m).

After finishing the RTD calculations for `inlet_one`, the same procedure should be followed for calculating RTD of `inlet_two`, except T value for `inlet_one` should be `0` and for `inlet_two` it should be `1.0`.

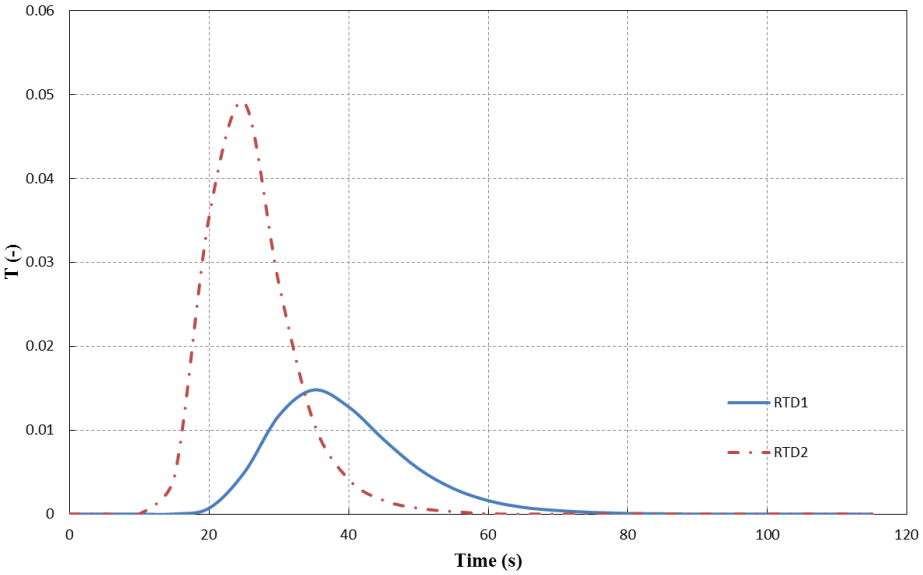


Average value of T on the outlet for two inlets versus time

The average value of T for each outlet approaches a certain constant value, which is the ratio of that scalar mass inlet to the whole mass inlet. For plotting data over time “Plot Selection Over Time” option in ParaView can be used, in the opened SpreadSheetView window (IntegrateVariables) select the set of data which you want to plot over time and then:

Filters > Data Analysis > Plot Selection Over Time > Apply

Next, to obtain the RTD plots, export the data to a spreadsheet program (e.g. Excel), calculate and plot the gradient of changes in average value of T on the outlet from time 0 to 120s for both inlets.



RTD of two inlets

Tutorial Eleven

Reaction

Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Background

In computational fluid dynamics (CFD) and chemical reaction modeling, two commonly used approaches to simulate reactive flows are the Partially Stirred Reactor (PaSR) Model and the Eddy Dissipation Concept (EDC) Model. These models help in understanding and predicting how chemical reactions occur in different flow environments, such as combustion, industrial chemical processing, pollutant dispersion, and atmospheric chemistry.

1. Partially stirred reactor (PaSR) Model

Partially stirred rector (PaSR) model is used to model thermodynamic and chemical reactions numerically, for example, combustion. In the PaSR approach, a computational cell is split into two different zones: a reacting zone and a non-reacting zone. The reacting zone is modeled as a perfectly stirred reactor (PSR), and all reactants are assumed to be perfectly mixed with each other.

For the reactor, we are interested in three concentrations, 1) mean concentration of key component in the feed, c_{in}; 2) mixture concentration in the reacting zone, c; 3) concentration at the reactor exitc_{exit}.

In the reacting zone, reaction occurs for a duration of τ_c, so the concentration of mixture changes from c_{in} to c. In the non-reacting zone, the reacted mixture is getting mixed up with the non-reacted mixture for a duration of τ_{mix}, resulting in the final exit concentration, c_{exit}.

A key parameter to be calculated in this model would be the reaction rate, and it is clear that the reaction rate is proportional to the ratio of the chemical reaction time to the total conversion time in the reactor (i.e. sum of reacting and mixing time), κ_k:

$$\kappa_k = \frac{\tau_c}{\tau_c + \tau_{mix}}$$

2. Eddy dissipation concept (EDC) Model

The Eddy Dissipation Concept (EDC) model looks at the interaction between reaction and turbulence, where the overall reaction rate is controlled by turbulent mixing. It is widely used for combustion modeling for a great variety of combustion environments with great success.

It is assumed in the model that most reaction takes place within fine turbulence structures, which are modeled as perfectly-mixed reactors. We need to know the reaction mass fraction and the mass transfer rate between the fine structures and its surrounding fluid.

The mass fraction occupied by the fine structures, γ^*, is expressed as:

$$\gamma^* = \left\{\frac{u^*}{u'}\right\}^2$$

Where u^* is the mass average fine structure velocity. The fine structures are in regions with nearly constant turbulent kinetic energy given by u'^2.

The mass transfer rate between fine structure and surrounding fluid per unit of fluid and per unit of time is modeled as:

$$\dot{m} = 2 \cdot \frac{u^*}{L^*} \cdot \gamma^*$$

where L^* is the characteristic length of the fine structure.

multicomponentFluid – reactingElbow

Tutorial outline

Use the multicomponentFluid solver; simulate combustion of CH_4 and O_2 in a mixing elbow:

- Use the two times finer Hex mesh from Example One
- Domain initially filled with N_2
- velocity-inlet-5:
  - Velocity: 1 m/s
  - Mass fractions: 23 % O_2, 77 % N_2
  - Temperature: 800 K
- velocity-inlet-6:
  - Velocity: 3 m/s
  - Mass fractions: 50 % CH_4, 50 % N_2
  - Temperature: 293 K
- Operating pressure: 10^5 Pa
- Operating temperature: 298 K
- Isolated walls

Objectives

- Understanding multi-species and reaction modeling in OpenFOAM®

Data processing

Evaluate your results in ParaView.

1. Pre-processing

1.1. Copying tutorial

Copy the following tutorial to your working directory:

$FOAM_TUTORIALS/multicomponentFluid/counterFlowFlame2D

Copy the GAMBIT® mesh from Tutorial One (two times finer mesh) to the case main directory.

1.2. 0 directory

Update all the files in 0 directory with new boundary conditions, e.g. U:

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

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    wall-4
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    velocity-inlet-5
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }

    velocity-inlet-6
    {
        type            fixedValue;
        value           uniform (0 3 0);
    }

    pressure-outlet-7
    {
        type            zeroGradient;
    }

    wall-8
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    frontAndBackPlanes
    {
        type            empty;
    }
}

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

The reaction taking place in this simulation CH_4 combusting with O_2 creating CO_2 and H_2O. N_2 is the non-reacting species. The boundary conditions and

initial value of all species should be defined in the 0 directory. These values are mass fractions (between 0 and 1) and dimension less, e.g. CH_4:

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

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0.0;

boundaryField
{
    wall-4
    {
        type            zeroGradient;
    }

    velocity-inlet-5
    {
        type            fixedValue;
        value           uniform 0; //no CH4 at this inlet
    }

    velocity-inlet-6
    {
        type            fixedValue;
        value           uniform 0.5; //50% CH4 mass fraction at this inlet
    }

    pressure-outlet-7
    {
        type            zeroGradient;
    }

    wall-8
    {
        type            zeroGradient;
    }

    frontAndBackPlanes
    {
        type            empty;
    }
}

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

Note: If the file for a species does not exist in the 0 directory, the values from Ydefault will be used for that species.

Note: For the `pressure-outlet-7` set the species boundary conditions to `zeroGradient`.

1.3. constant directory

In the *physicalProperties* file the physical properties of the species can be set:

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
thermoType
{
    type            hePsiThermo;
    mixture         coefficientWilkeMultiComponentMixture;
    transport       sutherland;
    thermo          janaf;
    energy          sensibleEnthalpy;
    equationOfState perfectGas;
    specie          specie;
}
```

```
defaultSpecie N2;

#include "thermo.compressibleGas"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * */
```

The mixture type is set to a multi-component mixture for calculating the mixture properties and the heat capacities are calculated using "janaf polynomials".

N_2 is defines as `defaultSpecie`. In reaction solvers in OpenFOAM® the default specie is calculated explicitly using the mass balance equation (to satisfy mass conservation):

$$mass\ fraction\ of\ inert\ specie = 1 - \sum mass\ fraction\ of all\ other\ species$$

Involved species are listed in the *thermo.compressibleGas* file, which was included at the end of *physicalProperties* file. The species in this simulation are O_2, H_2O, CH_4, CO_2 and N_2. They are defined in the `species` sub-dictionary:

```
species
(
    O2
    H2O
    CH4
    CO2
    N2
);
```

The reactions are addressed in the *reactions* file:

```
reactions
{
    methaneReaction
    {
        type     irreversibleArrhenius;
        reaction "CH4 + 2O2 = CO2 + 2H2O";
        A        5.2e16;
        beta     0;
        Ta       14906;
    }
}
```

in the `reactions` sub-dictionary. The reaction of methane combustion is defined and it is of `type` irreversible Arrhenius reaction, `irreversibleArrhenius`.

In the Tutorial Two it was explained that the coefficients for calculating gas mixture properties are defined in the `mixture` sub-dictionary because it was a homogeneous mixture. However, in this example the mixture is not homogenous so coefficients for calculating properties of each species are needed separately to calculate mixture properties based on each cell composition. The coefficients of each species are defined in the *thermo.compressibleGas* file from the constant directory. For example, the `O2` coefficients for each model are shown below:

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

O2
{
    specie
    {
        molWeight       31.9988;
```

```
    }
    thermodynamics
    {
        Tlow            200;
        Thigh           5000;
        Tcommon         1000;
        highCpCoeffs    ( 3.69758 0.00061352 -1.25884e-07 1.77528e-11 -
                          1.13644e-15 -1233.93 3.18917 );
        lowCpCoeffs     ( 3.21294 0.00112749 -5.75615e-07 1.31388e-09 -
                          8.76855e-13 -1005.25 6.03474 );
    }
    transport
    {
        As              1.753e-06;
        Ts              139;
    }
}
…

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

In the `thermodynamics` sub-dictionary, the janaf polynomial model coefficients for calculating the heat capacity can be found and in `transport` the sutherland model coefficients for viscosity are stored.

1.4. system directory

By setting the `adjustTimeStep` to `yes` in the *controlDict*, the solver automatically ignores `deltaT`, and calculates the `deltaT` based on the maximum Courant number `maxCo` defined for it. Change the `endTime` to `120` (approximately one time the volumetric residence time based on velocity-inlet-5) and `writeInterval` to `10`, to write every 10 s to case directory.

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

application     foamRun;

solver          multicomponentFluid

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         120;

deltaT          1e-6;

writeControl    adjustableRunTime;

writeInterval   10;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;
```

```
runTimeModifiable true;

adjustTimeStep  yes;

maxCo           0.4;

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

2. Running simulation

```
>fluentMeshToFoam fineHex.msh
```

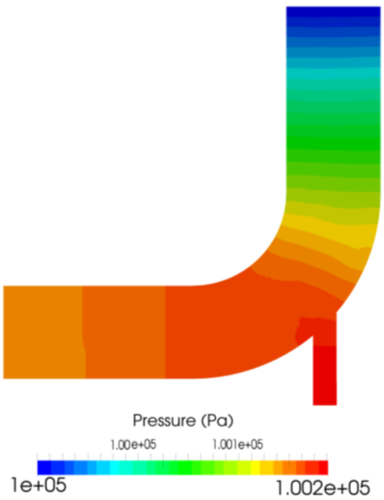
After converting the mesh, check the boundary file in the constant/polyMesh directory and change the `type` and `inGroups` of boundary `frontAndBackPlanes` from `wall` to `empty` (it is a 2D simulation).

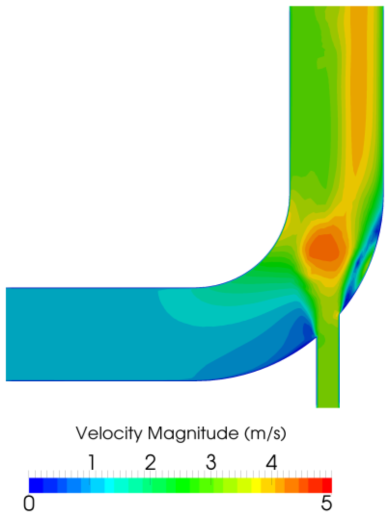
```
>foamRun -solver multicomponentFluid
```

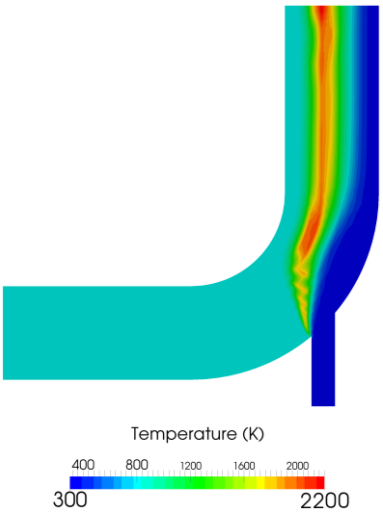
```
>foamToVTK
```

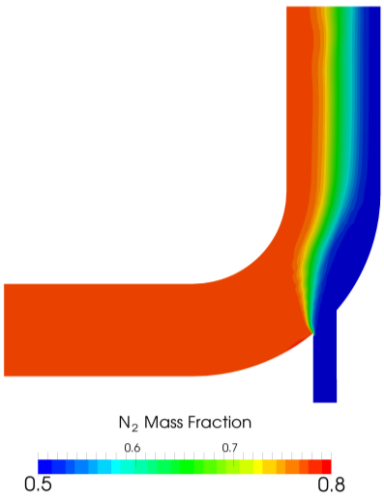
3. Post-processing

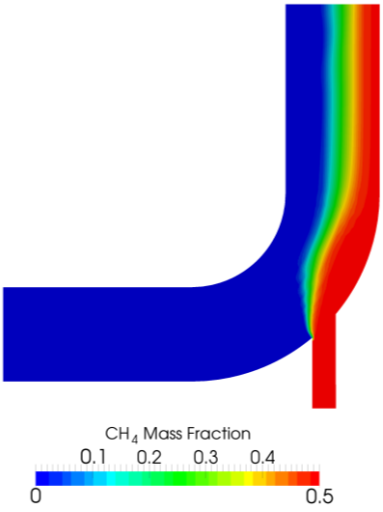
The simulation results at 120 s are as follows:

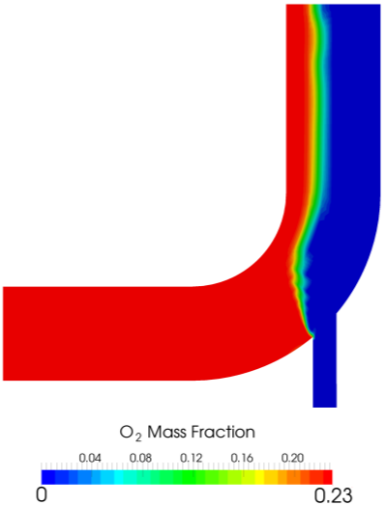


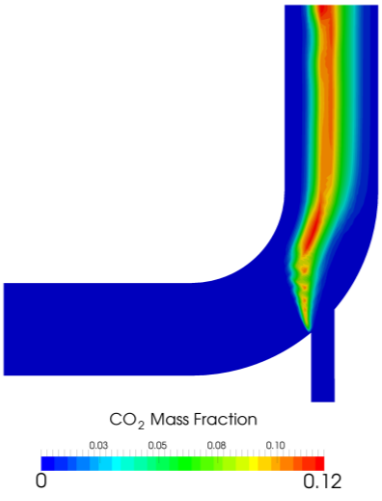


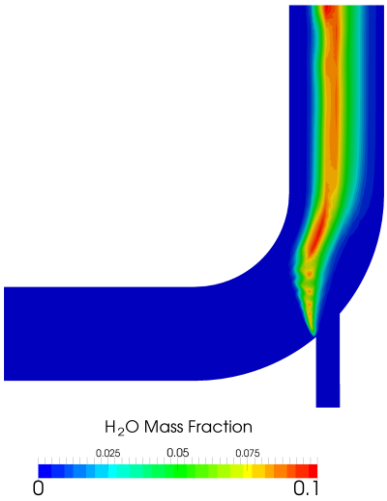








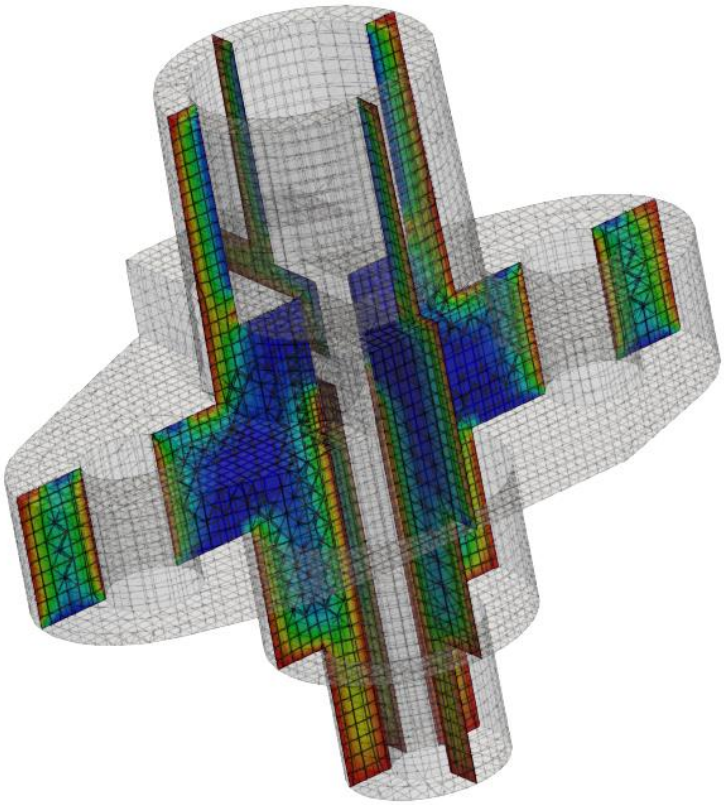




Simulation results after 120 s

Tutorial Twelve

snappyHexMesh – Single Region



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Philipp Schretter
- Yitong Chen

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

In this tutorial, we will familiarize ourselves with the *snappyHexMesh* tool in OpenFOAM®. This utility generates 3D meshes containing hexahedra and split-hexahedra. We will also introduce different types of meshes with complex geometries and compare the *snappyHexMesh* tool with other mesh generation tools.

1. Meshes with complex geometries

So far we have only worked with meshes in Cartesian co-ordinates, however, many engineering problems involve complex geometries that do not fit exactly in Cartesian co-ordinates. In such cases, it would be much more advantageous to work with grids that can handle curvature and geometric complexity more naturally.

CFD methods for complex geometries are classified into two groups:

1. structured curvilinear grid arrangements
2. unstructured grid arrangements

In a **structured grid** arrangements:

- Cells center points are placed at the intersections of co-ordinates lines
- Cells have a fixed number of neighboring cells
- Cells center points can be mapped into a matrix based on their location in the grid
- Structure and position in the matrix is given by indices (*I*, *J* in two dimensions and *I*, *J*, *K* in three dimensions)

For the most complex geometries it may be necessary to sub-divide the flow domain into several different blocks, where each mesh cell is a block, this is known as **block-structured grids**. The next level of complexity is the **unstructured grids**. It gives unlimited geometric flexibility, here the limitations of structured grids do not apply – but at the cost of higher programming and computational efforts. Unstructured grids also allow the most efficient use of computing resources for complex flows, so this technique is now widely used in industrial CFD.

2. Mesh generation tools

There are a number of advanced meshing tools available, both commercial and free source. The major mesh generators are ANSYS GAMBIT®, ICEM, Salome, snappyHexMesh and cfMesh. Here we will learn about GAMBIT®, snappyHexMesh and cfMesh tools in detail.

2.1. GAMBIT®

GAMBIT® is a 3D unstructured tool, to specify the meshing scheme in it, two parameters must be specified:

- Elements
- Type

The Elements parameter defines the shape(s) of the elements that are used to mesh the object. The Type parameter defines the pattern of mesh elements on the object. It has a single graphical user interface which brings geometry creation and meshing together in one environment.

2.2. snappyHexMesh

In contrast to GAMBIT®, which incorporates both mesh generation and refinement, the *snappyHexMesh* tool built within OpenFOAM® requires an existing geometry base mesh to work with. The base mesh usually comes from the blockMesh tool. This utility has the following key features:

- allow parallel execution to speed up the process
- supports geometry data from STL/OBJ files
- addition of internal and wall layers
- zonal meshing

The key steps involved when running snappyHexMesh are:

- **Castellation**: The cells which are beyond a region set by a predefined point are deleted
- **Snapping**: Reconstructs the cells to move the edges from inside the region to the required boundary
- **Layering**: Creates additional layers in the boundary region.

The advantages of snappyHexMesh over the other mesh generation tools are as follows:

- No commercial software package is ultimately necessary. For the meshing, the OpenFOAM® environment is sufficient and no further software is necessary.
- The geometry can be created with any CAD program like CATIA®, FreeCAD, etc. As the geometry is to be only surface data, the files need to be in .stl, .nas or .obj. format.
- The meshing process can be run in parallel mode. If high computational capabilities are available, high quality meshes can be generated in little time.

2.3. cfMesh

cfMesh is an open-source library for mesh generation implemented within the OpenFOAM® framework (like *snappyHexMesh*). Currently cfMesh is capable of producing mesh of Cartesian type in both 2D and 3D, tetrahedral and polyhedral types.

The fundamental work-flow of the tool starts from a mesh template, then followed by a mesh modifier. The modifier allows for efficient parallelization using shared memory parallelization (SMP) and distributed memory parallelization using MPI.

snappyHexMesh – flange

Tutorial outline

The procedure described in this tutorial is structured in the following order:

- Creation of the geometry data
- Meshing a geometry with one single region
- Run an OpenFOAM® simulation with the generated mesh using functions solver

Objectives

- The aim of the tutorial is to introduce single region meshing with the meshing tool snappyHexMesh
- Understanding the advantages of snappyHexMesh
- Understanding the three basic steps of snappyHexMesh

Data processing

Import your simulation to ParaView and analyze the heat distribution in the flange.

1. Pre-processing

1.1. Copying tutorial

Copy the following tutorial to your working directory.

```
$FOAM_TUTORIALS/mesh/snappyHexMesh/flange
```

Normally the .stl files are created using CAD software, such as CATIA® and freeCAD. stl files contain information about the solid geometry. However, in this tutorial the stl files are available to be copied from the OpenFOAM® tutorials folder. To do this, copy the stl files from the below location to the constant/geometry of your running case directory.

```
$FOAM_TUTORIALS/resources/geometry/flange.stl.gz
```

1.2. constant directory

The **geometry** folder in the constant directory contains the geometry files to be meshed (stl, nas, obj). The files names is to be used as a reference in later stages.

Note: The stl file should be in ascii format. All the stl files (different boundaries stl files) should form a closed geometry together otherwise it is not possible to differentiate between inside and outside of the geometry in the meshing process.

1.3. system directory

For creating a mesh using snappyHexMesh, the following files should be present in system directory:

- **blockMeshDict**: For meshing using snappyHexMesh a background mesh is needed, which should surround the geometry surface (e.g. stl file) file. The background mesh will be refined based on the settings in the snappyHexMeshDict and the extra parts will be removed. Usually, the background mesh is created using blockMesh. Here we define a base mesh.

Note: To ensure that the sharp edges are refined properly, it is very important to create perfect cube cells in the background mesh using blockMesh utility.

- **decomposeParDict***:* The meshing using snappyHexMesh can be also performed in parallel mode, in this case the parameters for distributed processors are set in this file.
- **meshQualityDict***:* Parameters to be checked for mesh quality and their values are defined in this file (the default values are usually good).
- **surfaceFeatureExtractDict**: Using surfaceFeatures utility prior to meshing with snappyHexMesh helps to extract the sharp edges and have a better mesh with snappyHexMesh on these edges. All edges are marked, whose adjacent surfaces normal are at an angle less than the angle specified in `includedAngle` in the *surfaceFeaturesDict*. The

extracted edges are written to "*.extendedFeatureEdgeMesh" files in constant/extendedFeatureEdgeMesh folder to be treated later in the meshing process.

```
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
Surfaces ("flange.stl");

includedAngle    150;
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
```

- **snappyHexMeshDict**: This file includes the settings for running the snappyHexMesh. As mentioned in the Background section meshing using this tool has three steps:
  1) Castellating
  2) Snapping
  3) Layering

In the first section of this file, `castellatedMesh`, `snap`, `addLayers` can be set to true or false depending on the stages required. In the following setting, castellating and snapping are active and adding layers is deactivated (to activate it, set the flag to true).

```
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
          castellatedMesh          true;
          snap                     true;
          addLayers                false;
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
```

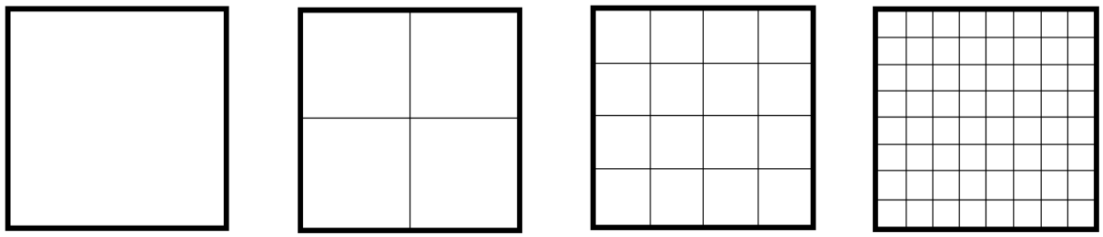
The `Geometry` sub-dictionary lists all surfaces used by *snappyHexMeshDict*, and assign them a name to be used as a reference.

It is also possible to specify regions in the domain that we want to treat them specially later, e.g. in this case we try to define a spherical region to refine it. The refined region is given an arbitrary name; in this case, it is `refineHole`, which is a sphere with its center and radius defined.

```
//  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * //
geometry
{
    flange
    {
        type triSurfaceMesh;
        file "flange.stl";
    }
    refineHole
{
    type searchableSphere;
    centre (0 0 -0.012);
    radius 0.003;
    }
};
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
```

1.3.1. CASTELLATING

In the castellating step based on the settings in the *snappyHexMeshDict* file, the created background mesh (in this case using blockMesh) cells are cut into sub-cells and the unneeded part of the mesh will be deleted (based on the internal point defined by user). The background mesh is known as mesh “level 0”, by setting the “level” to 1 the background mesh at the position of features or defined refinements will be cut into half in each direction (creating 8 sub-cells for a 3D mesh). Therefor by each level of refinement number of cells increases by factor 8!



Refinement level 0, level 1, level 2, level 3

The `castellatedMeshControls` sub-dictionary is used for user-defined mesh refinement in the castellating step.

`features` allows refinement of the “*.extendedFeatureEdgeMesh” edges to a certain level.

`refinementSurfaces` are for surface-based refinement. Every surface is specified with two levels. The first level is the minimum level and the second level is the maximum level of refinement. If the type of the surface is also defined (e.g. patch or wall) the surface will be marked as a boundary with the assigned name and type.

`resolveFeatureAngle` is an important setting. Edges, whose adjacent surfaces normal are at an angle higher than the value set, are resolved. The lower the value, the better the resolution at sharp edges.

`refinementRegions`: Volume based refinement of the regions defined in the geometry section. In this tutorial the `refinementHole` region will be refined. In the levels the first number (1E15) is the maximum number of the cells which can be reached after refinement in this region and second number (3) is the level of refinement

`locationInMesh`: Important coordinate for single region cases, to define which part of the mesh should be kept, inside or outside the geometry.

```
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
castellatedMeshControls
{
    maxLocalCells 100000;
    maxGlobalCells 2000000;
    minRefinementCells 0;
    nCellsBetweenLevels 1;

    features
    (
        {
            file "flange.extendedFeatureEdgeMesh";
            level 0;
        }
```

```
    );

    refinementSurfaces
    {
        flange
        {
                level (2 2);
        }
    }

    resolveFeatureAngle 30;

    refinementRegions
    {
       refineHole
       {
          mode inside;
          levels ((1E15 3));
       }
       locationInMesh (-9.23149e-05 -0.0025 -0.0025);
       allowFreeStandingZoneFaces true;
    }
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
```

Note: The `locationInMesh` point should never be on a face of the mesh, even after refinement. It should always be inside a cell or the meshing will fail!

In the castellated step, the background mesh will be refined based on the defined refinement levels at features, surfaces or regions and the unneeded part of the mesh will be removed.

1.3.2. SNAPPING

Important parameters are number of mesh displacement iterations, `nSolveIter` and the number of feature edge snapping iterations, `nFeatureSnapIter`. The default values are fine for most of applications.

```
//  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * //
snapControls
{
    nSmoothPatch 3;
    tolerance 1.0;
    nSolveIter 300;
    nRelaxIter 5;
        nFeatureSnapIter 10;
        implicitFeatureSnap false;
        explicitFeatureSnap true;
        multiRegionFeatureSnap true;
}
//  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * //
```

1.3.3. LAYERING

The label for the layering is equal to the labeling of the Boundary surface in the boundary file in the constant/polyMesh folder.

- `nSurfaceLayers` defines the number of surface layers
- `expansionRatio` defines the expansion ratio of the surface layers
- `finalLayerThickness` and `minThickness` define the min and the final thickness of the surface layers

- `nLayerIter`: if not snapped smoothly enough, the max number of layer addition iteration can be increased.

```
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
addLayersControls
{
    relativeSizes true;
    layers
    {
        "flange_.*"
        {
            nSurfaceLayers  3;
        }
    }
expansionRatio 1.005;

    finalLayerThickness 0.3;
    minThickness 0.25;
    nGrow 0;
    featureAngle 30;
    nRelaxIter 5;
    nSmoothSurfaceNormals 1;
    nSmoothNormals 3;
    nSmoothThickness 10;
    maxFaceThicknessRatio 0.5;
    maxThicknessToMedialRatio 0.3;
    minMedianAxisAngle 90;
    nBufferCellsNoExtrude 0;
nLayerIter 50;
nRelaxedIter 20;
}
meshQualityControls
{
    #include "meshQualityDict"
relaxed
{
    maxNonOrtho 75;
}

nSmoothScale 4;
    errorReduction 0.75;
}
writeFlags
(
    scalarLevels
    layerSets
    layerFields
);
mergeTolerance 1e-6;
   //  * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * //
```

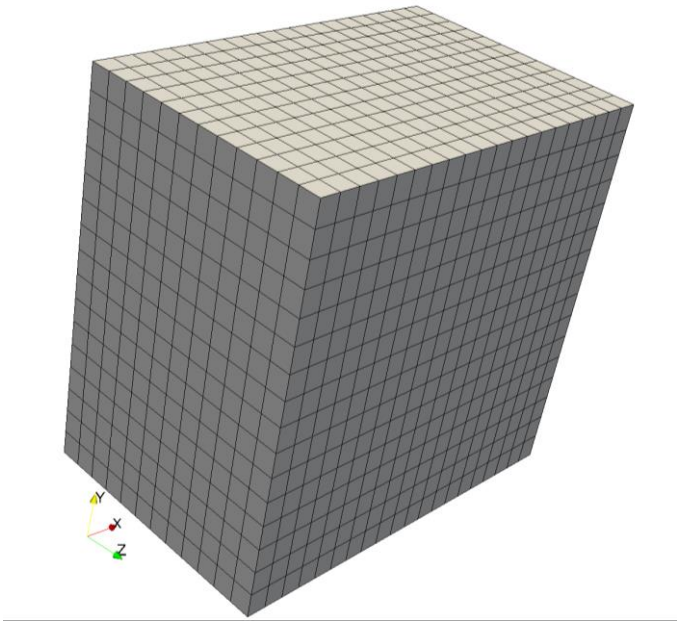
Note: Only the relevant changes, which were used in the sample flange case, are commented in the snappyHexMeshDict.

2. Running snappyHexMesh

The background mesh is created with the following command:

```
>blockMesh
```

According to the settings in the *blockMeshDict*, the mesh was created with 20 cells in x direction, 16 cells in y direction and with 12 cells in z direction.



Block mesh for flange

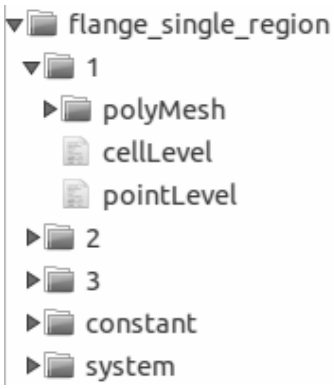
```
>surfaceFeatures
```

The command to mesh the flange geometry on one processor is

```
>snappyHexMesh
```

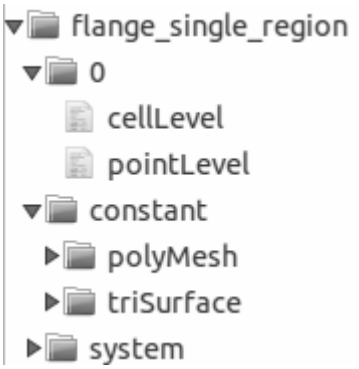
Note: The meshing process with snappyHexMesh can also be run in parallel. To run the command on several processors, refer to Tutorial Eight for more information.

The command `snappyHexMesh` creates a folder with the mesh files for each mesh step. If, for example, in the **snappyHexMeshDict**, only *castellatedMesh* is set to *true* and *snap* and *addLayers* are set to *false*, only one folder is created. If also *snap* is set to *true*, 2 folders are created and if also *addLayers* is set to *true*, 3 folders with 3 polyMesh folders are created. The folders are created based on the `deltaT` settings in the *controlDict* File (in this case it is 1, therefore folders are 1, 2 and 3).



Folders structure after running snappyHexMesh

In order to avoid the creation of these folders and only keep the final mesh (to be written directly in the constant folder), the following command can be used to overwrite the previous meshing steps. In this case, only one polyMesh folder exits in the /constant directory.



Folders structure after using -overwrite flag

```
>snappyHexMesh -overwrite
```

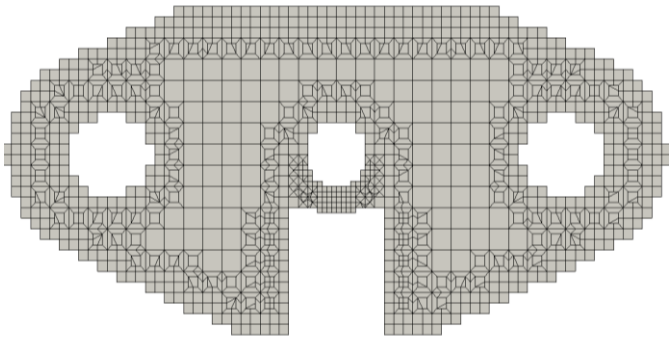
However, sometimes it is useful to run snappyHexMesh without the overwrite option, as it allows the user to make changes to a specific time step without having to run all the other steps again, thus reducing computational time.

3. Examining the meshes

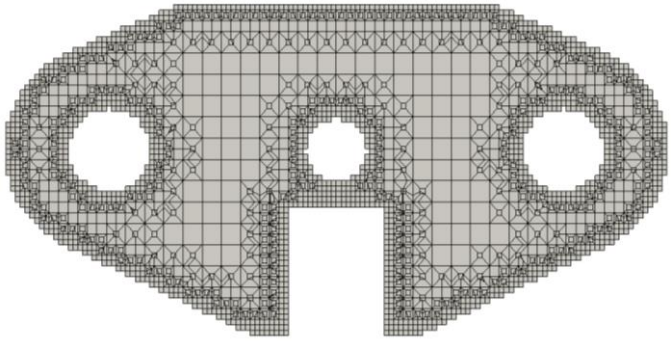
To examine, what each of the steps in the *snappyHexMeshDict* really does, we need to turn off the overwrite feature in snappyHexMesh command and generate VTK files to be opened in ParaView.

```
>foamToVTK
```

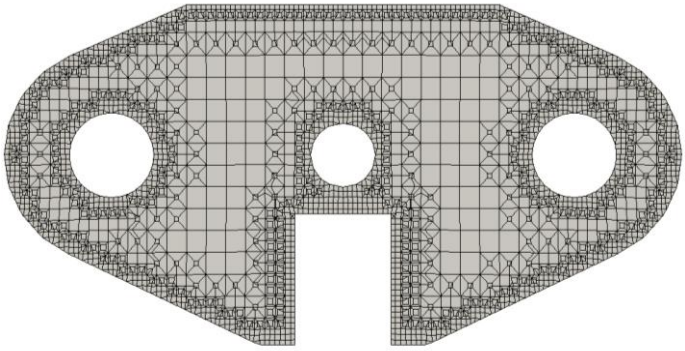
Simply change the time in Paraview to see the effect of *snappyHexMesh* steps on the mesh, i.e. time 1 corresponds to the mesh after castellating step, time 2 for the mesh after snapping step, time 3 for the mesh after the layering step.



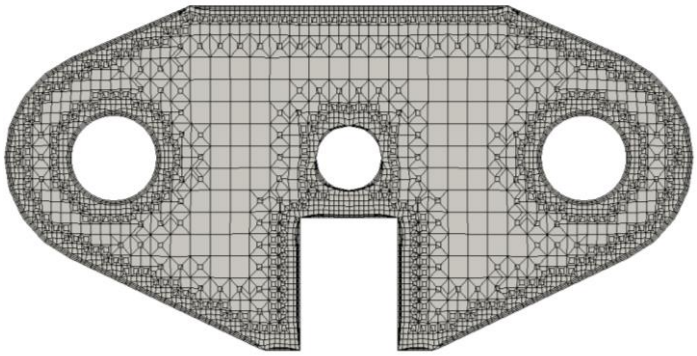
Flange mesh for step castellating with surface refinement level 2



Flange mesh for step castellating with surface refinement level 3

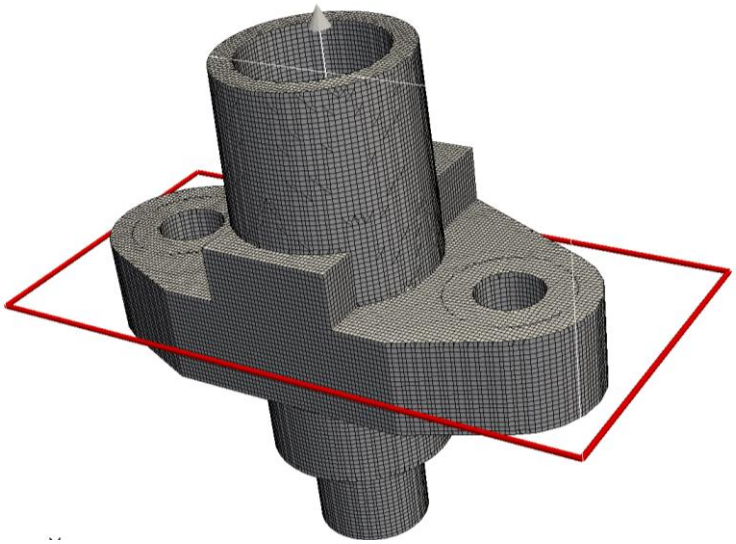


Flange mesh for step snap with surface refinement level 3



Flange mesh for step addlayers with surface refinement level 3

The slice views taken with ParaView from the center of the flange. The slices are depicted by the red plain in the following figure:



Flange with sectional plain

You can review the mesh quality with the tool *checkMesh*.

```
>checkMesh
```

4. Running simulation

4.1. Copy tutorial

Now with the new mesh ready, let's run some simulation on it! Here functions solver is chosen for the simulation. To set up the case, copy the following tutorial file into your working directory:

```
$FOAM_TUTORIALS/incompressibleFluid/pitzDailyScalarTransport
```

The flange mesh files need to be transferred to the running case directory. To achieve this, copy the polyMesh folder from the latest time step file of the flange folder into the constant directory of the pitzDaily folder. If the overwrite function is activated when running snappyHexMesh, copy the polyMesh folder from constant directory of the flange folder.

4.2. Case set-up

The following changes need to be made to set up the case:

In the 0 directory:

Remove all the files, except *p*, *T* and *U*. Update the *T* file, so that the flange has an initial temperature of 293K but is heated up from the inlet at 350K

```
dimensions      [0 0 0 1 0 0 0];

internalField   uniform 293;

boundaryField
{
   flange_patch1
```

```
    {
        type            fixedValue;
        value           uniform 350;
    }
    flange_patch2
    {
        type            fixedValue;
        value           uniform 293;
    }
    flange_patch3
    {
        type            fixedValue;
        value           uniform 293;
    }
    flange_patch4
    {
        type            fixedValue;
        value           uniform 350;
    }
}
```

Update the *U* and *p* files so that the velocity in the entire flange domain and at the boundaries is zero.

Note: to set the initial fields to zero, remove the nonuniform filed and it values and replace it with uniform field value, e.g. `uniform (0 0 0 )`.

In the constant directory

In the *momentumTransport* file, set the `simulationType` to `laminar`.

In the system directory

Update the *controlDict* file in the system directory by changing the `endTime` to 0.0005, `deltaT` to 0.000001 and `writeInterval` to 100.

4.3. Running solver

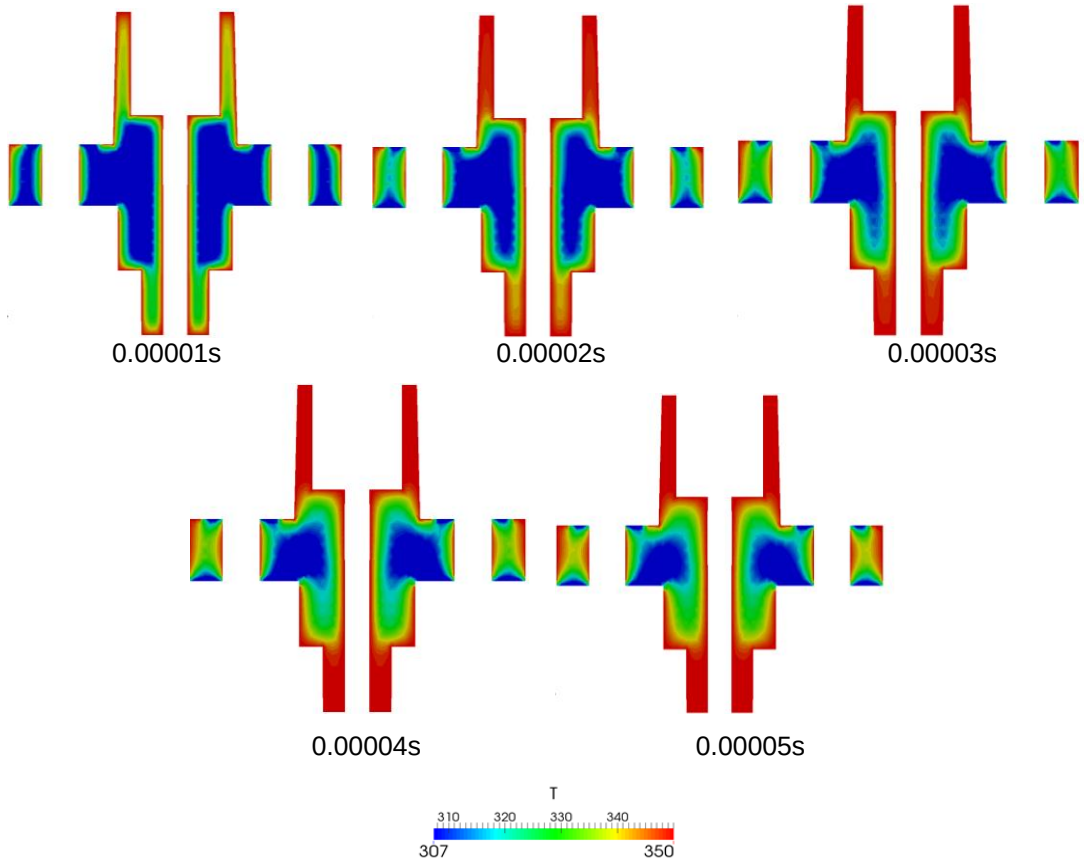
Run the solver with the command

```
>foamRun -solver functions
```

4.4. Results

Convert the results to VTK files with

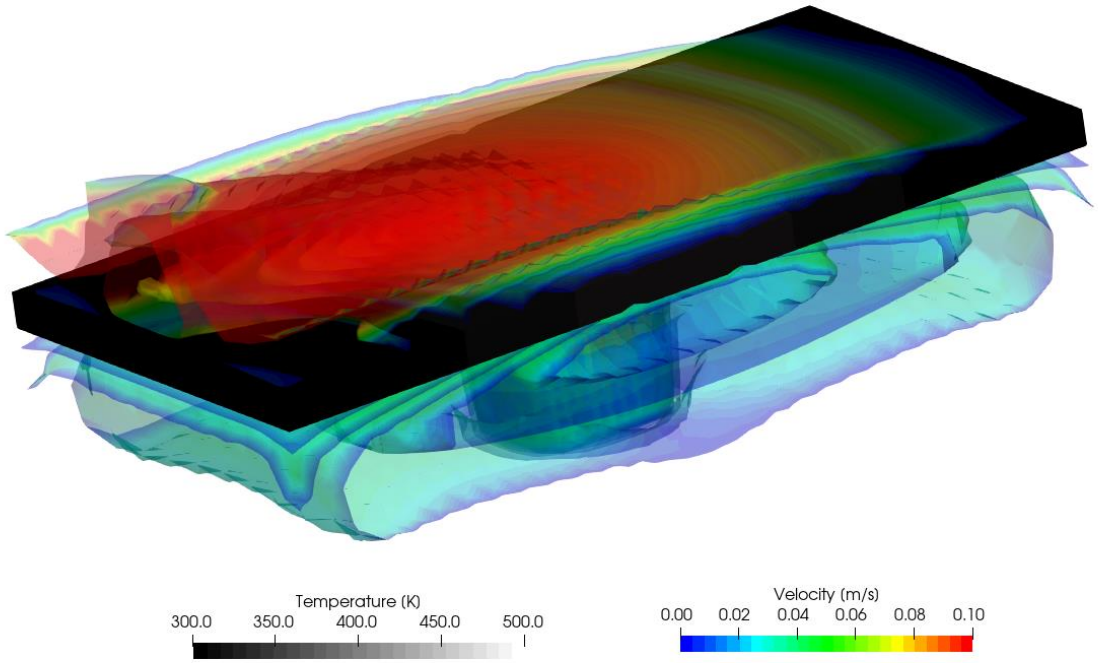
```
>foamToVTK
```



Heating of the flange from 0.01 to 0.05s

Tutorial Thirteen

snappyHexMesh – Multi-Region



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Philipp Schretter
- Yitong Chen

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Background

Multi-region modeling is a computational approach used in CFD simulations where the entire computational domain is divided into distinct regions, each representing a specific phase or material. This method is particularly useful for simulating conjugate heat transfer (CHT), fluid-structure interactions, and multiphase flow scenarios.

The key advantage of multi-region modeling is that it allows for separate governing transport equations to be solved for each region, ensuring accurate representation of different physical properties, material behaviors, and interactions at the interfaces.

Multi-region modeling is essential in cases where:

- Different materials with varying thermal properties exist (e.g., heat exchangers, cooling of electronic components, and thermal insulation studies).
- Conjugate heat transfer (CHT) needs to be modeled, where heat conduction through solids and convection in fluids occur simultaneously.
- Interfaces between different phases or substances are present (e.g., solid-liquid).
- Fluid-structure interaction (FSI) must be considered in simulations, where structural deformation affects the surrounding fluid flow.
- Energy transfer in coupled systems requires resolving different governing equations in separate domains.

Approaches to Multi-Region Modeling

Historically, two primary approaches have been used to solve multi-region problems:

1. Monolithic Approach

A single coupled matrix equation system is used to solve all regions simultaneously. Requires a strong coupling of governing equations across different regions and ensures better numerical stability but is computationally expensive. Typically used in high-accuracy simulations such as finite element-based methods.

2. Partitioned Approach (Used in OpenFOAM®)

Each region is treated as an independent subdomain, with separate governing equations solved for each. Interface conditions between regions are explicitly enforced via boundary conditions and it is computationally more efficient than

the monolithic approach. It is used in scenarios where different physics must be solved separately but interact at interfaces.

Steps for Multi-Region Modeling in OpenFOAM®

Step 1: Define the Computational Domain & Mesh Regions

The entire simulation domain is divided into multiple regions, typically a combination of fluid and solid domains. Each region is assigned specific material properties, such as: density, viscosity, thermal conductivity and specific heat capacity for fluids and thermal conductivity, heat capacity, and density for solids. The mesh is generated separately for each region using tools like blockMesh, snappyHexMesh, or external meshing software.

Step 2: Assign Field Variables for Each Region

Each region requires separate field definitions, such as temperature (T), velocity (U), and pressure (p). Thermophysical properties for each region are defined using thermophysicalModels and initial conditions for each region must be assigned in the 0 directory.

Step 3: Solve Transport Equations in Each Region

The governing equations for mass, momentum, and energy are solved for each individual region. For fluid regions, Navier-Stokes, continuity and energy equations equations are solved while in the solid region's equations such as Fourier heat conduction equation for temperature distribution are normally considred. Time-stepping is based on the smallest time-step for all the regions and numerical schemes are managed separately for each region.

Step 4: Multi-Regional Coupling at Interfaces

The interaction between different regions is enforced at interfaces by specifying boundary conditions. The interface conditions can be temperature continuity (matching temperature at the interface) and heat flux continuity (ensuring conservation of energy transfer).

Step 5: Iterative Solution for Fully Coupled Results

The solver iterates between individual region solutions until convergence is achieved. Adjustments to time-stepping, solver settings, and relaxation factors may be needed for numerical stability.

snappyHexMesh - snappyMultiRegionHeater

Tutorial outline

Try to create a multi-region geometry and mesh and run a conjugate heat transfer case using it.

Objectives

- Understanding multi region meshing with the meshing tool snappyHexMesh

Data processing

Import your simulation to ParaView. Analyze the flow field through the flange and the heat distribution in the flange.

1. Pre-processing

1.1. Copying tutorial

Download the following tutorial to your working directory:

https://github.com/OpenFOAM/OpenFOAM-5.x/tree/master/tutorials/heatTransfer/chtMultiRegionFoam/snappyMultiRegionHeater

Either by creating the folders and then downloading the files and place them in relevant folders or using following webpage:

https://download-directory.github.io/

Rename the constant/triSurface directory to constant/geometry directory. In the system directory, change the name of surfaceFeatureExtractDict to surfaceFeaturesDict.

1.2. 0 directory

Unlike the single region simulations in the 0 directory an individual folder per region exist which stores the files including initial and boundary conditions for that region (the folders can be created manually or will be generated automatically after creating and splitting the mesh). Also in the 0 directory some files exists which are just dummy files that will not be used in the simulations. The initial and boundary conditions for each region are changed and updated using the *changeDictionary* utility, which will be explained later.

1.3. constant directory

Also in the constant directory exist a folder per region; in this case, the domain is split into the following regions: bottom air, heater, left solid, right solid and top air. Within the designated folder, there are relevant dictionaries that describe the physical properties, turbulence or radiation behavior of each region, e.g. radiationProperties, momentumTransport and physicalProperties. The following changes should be applied (these changes are required, since the original tutorial belongs to OpenFOAM 5):

- In the bottomAir rename thermoPhysicalProperties file to physicalProperties
- In the bottomAir rename turbulenceProperties file to momentumTransport
- Copy the physicalProperties, momentumTransport and g file from bottomAir to topAir folder
- Update the thermoPhysicalProperties file in the constant/heater folder as following

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

thermoType
{
    type            heSolidThermo;
    mixture         pureMixture;
```

```
    transport       constIsoSolid;
    thermo          eConst;
    equationOfState rhoConst;
    specie          specie;
    energy          sensibleInternalEnergy;
}

mixture
{
    specie
    {
        molWeight   12;
    }

    transport
    {
        kappa   80;
    }

    thermodynamics
    {
        Hf      0;
        Cv      450;
    }

    equationOfState
    {
        rho     8000;
    }
}

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

- Copy the thermoPhysicalProperties file from constant/heater folder to constant/leftSolid and constant/rightSolid and replace the old files

The polyMesh directory in the constant folder (after the mesh is created) includes the original mesh while the polyMesh directories in each region folder (after the mesh is splitted) include the split mesh for that region with the new boundaries between regions.

Unlike polyMesh directories there exist just one geometry folder which stores all the stl files for mesh creation using snappyHexMesh.

In the regionProperties file, the physical phase of each region is specified. As you can see, bottom and top air regions are fluid, whereas heater, left and right solid are in solid phase.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
regions
(
        fluid                   (bottomAir   topAir)
        solid                   (heater   leftSolid   rightSolid)
);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *//
```

1.4. system directory

Like constant directory also in system directory, a folder per region can be found and all the settings for that region are stored in the corresponding folder, e.g. fvSolution, fvSchemes and decomposeParDict. The fvSchemes file in the system directory is a dummy file while the fvSolution includes the number of

outer correctors setting for PIMPLE algorithm. There is also just one controlDict file and it is in main system folder.

Note: For running the simulations in parallel, the decomposeParDict files for all the regions should have the same settings as the main one in the system directory. This is not valid for parallel meshing using snappyHexMesh while it just uses the decomposeParDict file in the main system directory.

Update the surfaceFeaturesDict file as following:

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

surfaces ("bottomAir.stl" "heater.stl" "leftSolid.stl" "rightSolid.stl"
"topAir.stl");
includedAngle   150;
writeFeatureEdgeMesh    yes;

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

Change the meshQualityDict as following:

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

// Include defaults parameters from master dictionary
#includeEtc "caseDicts/mesh/generation/meshQualityDict"

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

The files needed for creating a multi-region mesh are the same as the mesh for single-region, except for slight differences in snappyHexMeshDict file:

`locationInMesh`: In a multi-region mesh this point is not used but it should be defined just as a place holder.

`refinementSurfaces`: Different regions are defined here. E.g. for the region BottomAir all the faces and cells inside the bottomAir.stl (each region stl should be a closed volume) file are marked with `bottomAir` flag (in `faceZone` and `cellZone`).

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * *//
castellatedMeshControls
{
    maxLocalCells 100000;
    maxGlobalCells 2000000;
    minRefinementCells 10;
    nCellsBetweenLevels 2;

    features
    (
        {
            file "bottomAir.eMesh";
            level 1;
        }
…
        {
            file "topAir.eMesh";
            level 1;
        }
    );

    refinementSurfaces
    {
        bottomAir
        {
            level (1 1);
            faceZone bottomAir;
```

```
            cellZone bottomAir;
            cellZoneInside inside;
        }
…
       rightSolid
        {
            level (1 1);
            faceZone rightSolid;
            cellZone rightSolid;
            cellZoneInside inside;
        }
    }

    resolveFeatureAngle 30;

    refinementRegions
    {
    }
    locationInMesh (0.01 0.01 0.01);
    allowFreeStandingZoneFaces false;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * *//
```

After creation of the mesh and splitting to different regions, the initial and boundary conditions for each region can be manually set in the relevant region folders in 0 directory. This process can be also automated using the changeDictionary utility. The dictionary file for this utility for each region is in the relevant region folder in the system directory: *changeDictionaryDict*.

See below the *changeDictionaryDict* file for the heater region. In the `boundary` sub-dictionary type of boundaries for `minY`, `MinZ` and `maxZ` are set to patch. Then for `T` the internal field will be overwritten with `uniform 300`. In the next step all the boundaries in the T file for heater region will be set to `zeroGradient` (`“.*”` means all the boundaries with any name) and after that the bounadries with the name `“heater_to_.*”` will be changed to `turbulentTemperatureCoupledBaffleMixed` and `minY` will be changed to `fixedValue`.

```
  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * *//
boundary
{
minY
{
     type             patch;
}
minZ
{
     type             patch;
}
maxZ
{
     type             patch;
}
}

T
{
internalField          uniform 300;

boundaryField
{
    “.*”
    {
```

```
       type               zeroGradient;
       value              uniform 300;
 }
"heater_to_.*"
{
   type               compressible::turbulentTemperatureCoupledBaffleMixed;
   Tnbr               T;
   knappaMethod       solidThermo;
   value              uniform 300;
}
minY
{
   type               fixedValue;
   value              uniform 500;
}
}
    }
   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * *//
```

In the meshQualityDict file, change the following line:

```
#includeEtc "caseDicts/meshQualityDict"
```

to

```
#includeEtc "caseDicts/mesh/generation/meshQualityDict.cfg"
```

Note: Add the missing ";" to the fvSolution files for bottomAir and topAir regions:

```
"(rho|rhoFinal)"
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-7;
    relTol          0;
}
```

Update the laplacianSchemes in the the system/heater/fvSchemes file as following:

```
laplacianSchemes
{
    default              Gauss linear corrected;
    laplacian(alpha,h)   Gauss linear corrected;
}
```

In the system/heater/fvSolution change the h, $h and hFinal to e, $e and eFinal.

Copy and replace the fvScheme and fvSolution files from system/leftSolid and system/rightSolid with the ones from system/heater

Copy fvSchemes and fvSolution from bottomAir to topAir (replace the files)

2. Mesh creation and running simulation

The background mesh is created with blockMesh.

```
>blockMesh
```

Equal to the single region case, the command `surfaceFeatures` creates the **eMesh** files from the stl files with the geometry data. Also the folder *extendedFeatureEdgeMesh* is created in the constant directory. The creation of eMesh files with the command `surfaceFeatures` is not obligatory. This step is only necessary, if certain edges need to be refined.

```
>surfaceFeatures
```

For performing the meshing in parallel, the geometry needs to be decomposed prior to running *snappyHexMesh*. Depending on the number of subdomains, defined in the *decomposeParDict*, the processor folders are created accordingly.

```
>decomposePar
```

Note: *It is **recommended**, not to use the scotch method to decompose the region. Rather, the hierarchical or the simple method should be used. In case of scotch method, errors can occur while executing snappyHexMesh or while reconstructing the mesh.*

In order to prevent the creation of the folders 1, 2 (castellation and snapping features are turned on while layering is turned off) and only keep the final time step folder with the final mesh, the command *-overwrite* can be added after *snappyHexMesh.* In this case, only one folder, *0,* is created with the files **pointLevel** and **cellLevel**. The mesh data in this case is located in constant/polyMesh.

```
>mpirun -np 4 snappyHexMesh -parallel -overwrite
```

Note: If castellatedMesh and snap are set on true in the **snappyHexMeshDict**, only the snapped mesh is stored, whereas the intermediate step castellatedMesh is overwritten. If castellatedMesh, snap and addLayers are set on true in the **snappyHexMeshDict**, only the layered mesh is stored and the previous intermediate steps castellatedMesh and snap are overwritten.

In this case, only the steps *castellatedMesh* and *snap* are set to *true*, as these steps are applied to the whole mesh. The following command reconstructs the final mesh:

```
>reconstructPar -constant
```

After this step, all the regions are meshed but the meshes are connected and needs to be split. In the meshing step each region cells are marked with a flag and this flag will be used in the next step to split the mesh. Mesh regions can be split using the following command which split the mesh based on the flagged cellZones and overwrite the old meshes in the polyMesh directories in the region folders (if any exist):

```
>splitMeshRegions -cellZones -overwrite
```

With the mesh ready, the next step is to apply appropriate field values to each region, according to the *changeDictionaryDict*. This command needs to be repeated for each region, with the name of the region specified after the prefix `'-region'`.

```
>changeDictionary -region heater

>changeDictionary -region topAir
```

```
>changeDictionary -region bottomAir

>changeDictionary -region rightSolid

>changeDictionary -region leftSolid
```

Before running the simulations, the solver for each region needs to be defined, in the controlDict file add the following lines:

```
regionSolvers
{
      bottomAir   fluid;
      topAir      fluid;
      heater      solid;
      leftSolid   solid;
      rightSolid  solid;
}
```

and also remove the *rho* files from *0/topAir* and *0/bottomAir* folders, now it is ready to be run.

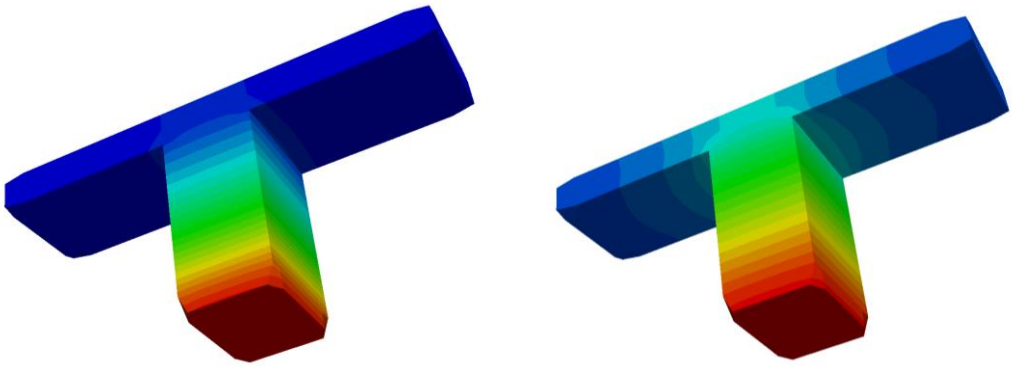
```
>foamMultiRun
```

Note: foamMultiRun can also be run on several processors.

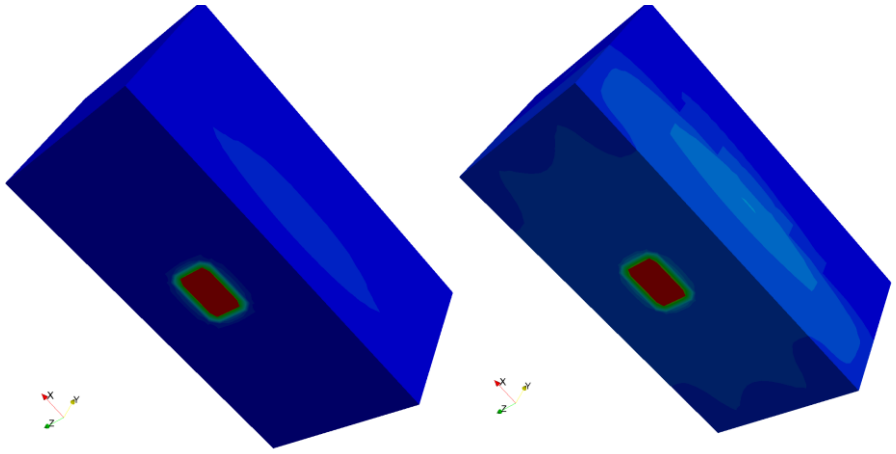
3. Post-processing

The results need to be converted to VTK files for each region with flag `-region`.

```
>foamToVTK -region heater

>foamToVTK -region topAir
```



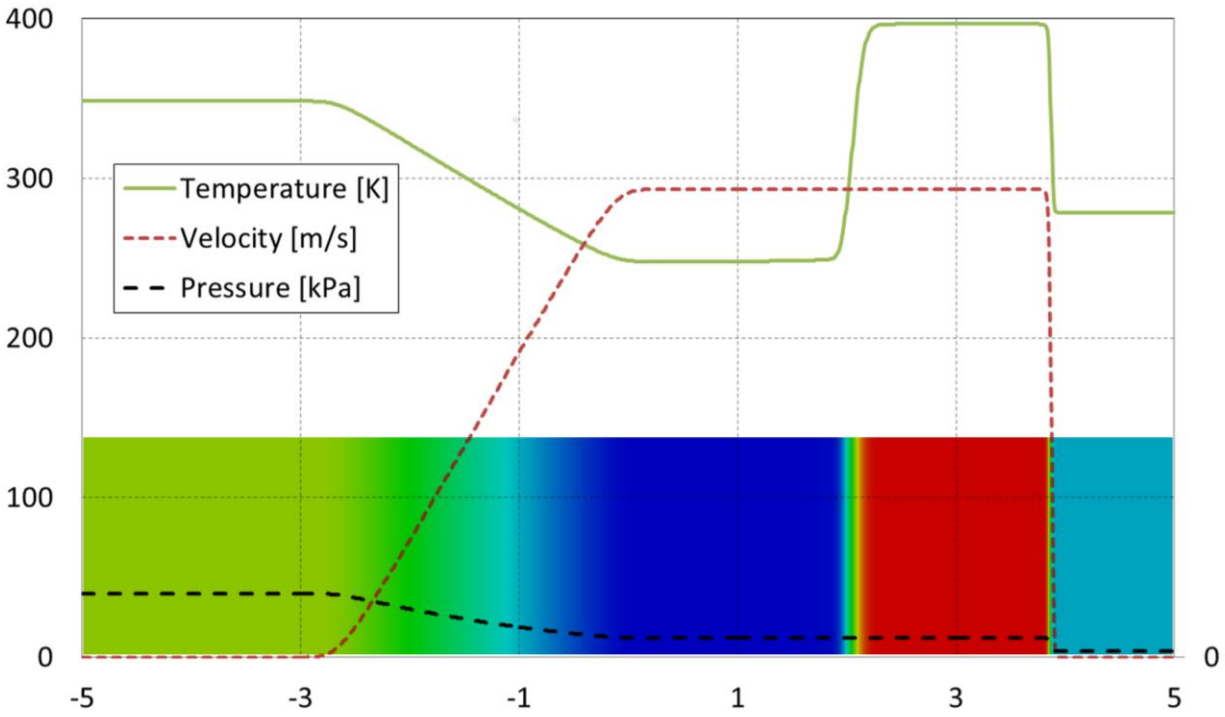
Temperature profile of heater region at time 15s and 75s



Temperature profile of entire mesh at time 15s and 75s

Tutorial Fourteen

Sampling



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

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

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

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:

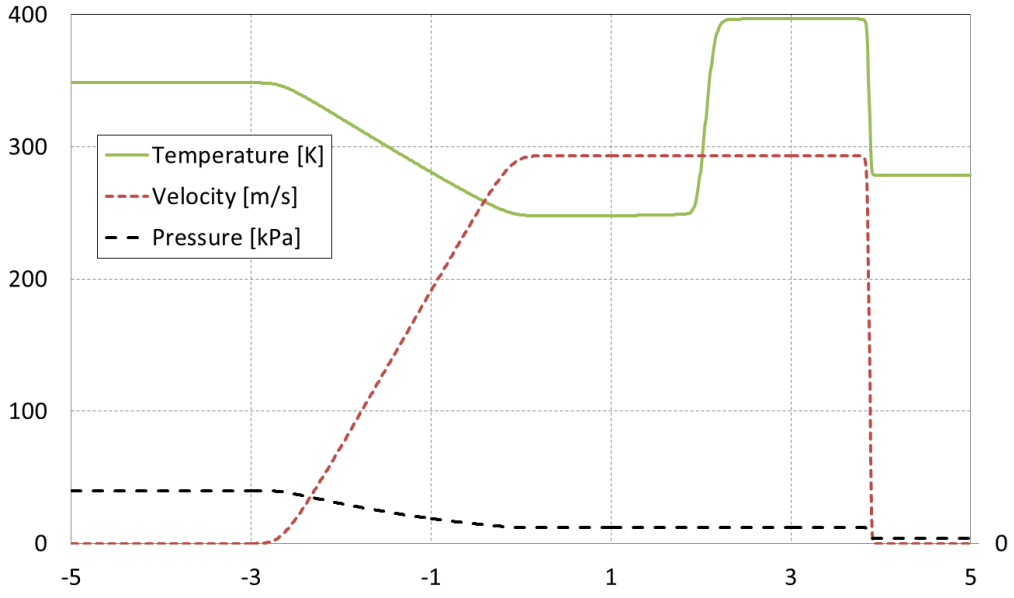
3.1. sample dictionary

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)
    }
```

Appendix A

Linux Commands

Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Clemens Gößnitzer
- Sylvia Zibuschka
- Yitong Chen
- Vikram Natarajan
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

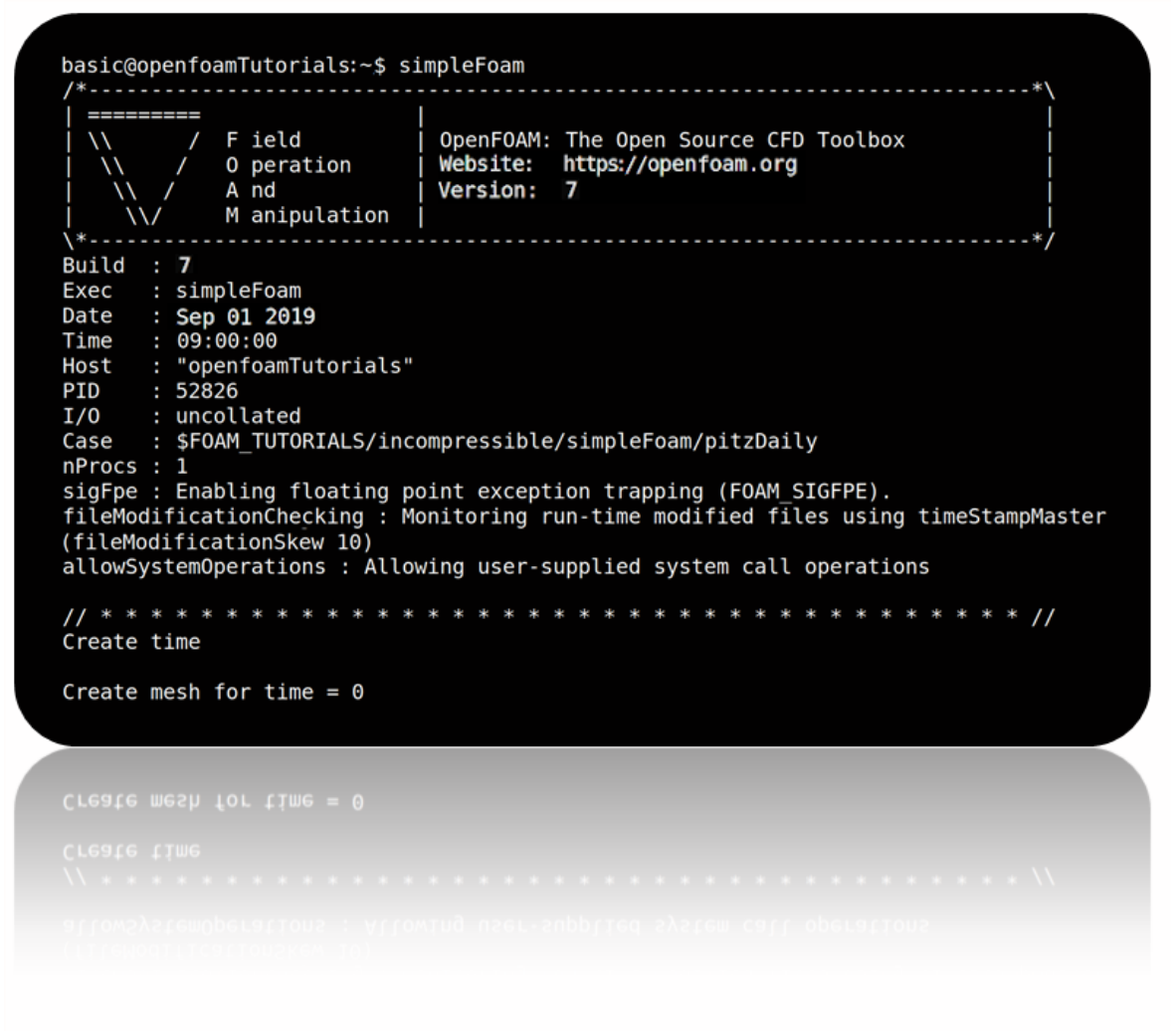
cat, more, less	File viewer with pure read function - in order of ease of operation. In ***less*** with ***pagedown***/***pageup*** you can navigate within the file, with ***/*** and ***?*** can look for strings, ***q*** can be used for closing ***less***. ***cat*** is back for universally available on Unix.
cd, cd ..	Changing the directory, ***cd ..*** goes one directory up and ***cd ~*** moves to home directory. Important to note is the space between ***cd*** and ***..*** as opposed to DOS!
cp, cp -r	Copying files or entire directory trees (with **-r** option). Caution: There is no warning or prompt when overwriting existing files! The important thing is that a target has to be always given, at least one **.** which means, copy to the current directory.
ctrl+r	Reverse search, for searching an already typed command in a terminal window.
du, du -s, du -k	Calculates the amount of space consumed in a directory. For safety reasons you should use the ***-k*** option (output in kilobytes), since some systems provide the space in blocks that include only 512 bytes ...
exit	Closing connection (terminal window).
gedit	Text editor with graphical user interface. When working with ***gedit*** some temporary files (originalFileName~) are created, they can be deleted after saving.
grep	Search command for plain-text data sets for lines matching a regular expression.
gzip, gunzip	Compression/decompression program for individual files (as opposed to ***zip***/***unzip***, this can also work on directories or file lists). The great advantage of ***gzip***: Fluent® and OpenFOAM® are able to read and write ***gz*** files directly, which saves about 30-90% space.
kill, kill -9	Stopping processes. For this the process ID is required, which can be found with ***top*** or ***ps***. The ***Exit*** is irrevocable course - but you cannot shoot processes, if you are not the "owner".
ls, ls –la	Lists the contents of a directory, with option ***-la*** also hidden files are displayed, as well as the file size and characteristics.
mc	Program that enables navigation in the text window, esc-keys, may be necessary: ***mc -c***, for navigating through mc use function keys or ***esc+[number]*** combination, e.g. ***F9*** or ***esc+9*** for moving to the menus at the top.

mkdir	Creates a new directory.
mv	Moving or renaming files and directories. Caution: There is no prompt when overwriting existing files!
Nano, pico	The command to run the ***nano*** text editor, a terminal based text editor.
passwd	The command to change the login password.
|	It is known as pipe and is used for merging two commands, redirecting one command as input to another, e.g. ***less|grep*** searches a specified word in the output of file opened with less.
ps, ps –A ps waux	Lists all the processes that were started in the respective command window with the options are all running processes on the system display.
pwd	Shows the current working directory.
rm, CAUTION: rm -fr	Deletes files. The option ***-r*** will also remove directories and files recursively and delete directories, ***f*** (force) prevents any further inquiry. - *Incorrectly applied, this command can lead to irreversible loss of all (private) data. There is no undelete or undo!*
rmdir	Deletes an empty directory.
scp	The copy command over the network - as secure FTP replacement. Also dominates the ***-r*** (recursive) option. Usage: ***scp*** source file destination file with source and the destination format can be USERNAME@ COMPUTER.DOMAIN:PATH/TO/FILE. Source or target can of course also be created locally, then (your) user name and computer are not required.
ssh	Telnet replacement with encryption. On Windows, for example, implemented with putty.
tail, tail -f	File viewer, the default outputs the last 10 lines of a file. With option ***-n XX*** can spend the last XX lines, with the ***-f*** option, the command is running from those lines, which are attached to a file. The command is therefore perfect for watching log files.

top	Displays a constantly updated list of all running processes, with process ID, memory and CPU usage. For processes of one user ***top [username]*** should be used, and for quitting ***q*** or ***ctrl+c*** should be applied.
vi, vim	File editor. For forward searching use **/**, for backward searching use ***?***. For exiting ***esc+:x***. ***nano*** or ***pico*** are recommended for beginners, which are easier to handle.

Appendix B

Running OpenFOAM®



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Sylvia Zibuschka
- Yitong Chen
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

1. Running OpenFOAM® on a Local Linux PC (or virtual machine):

- Open a terminal
- Go to the OpenFOAM® installation directory (e.g. /opt/openfoam10) in the opened terminal
- Change to the etc directory in the OpenFOAM® installation directory
- Run the following command:

```
>. ./bashrc
```

- If a new terminal is opened, the same procedure should be repeated in that in order to activate OpenFOAM® in here.

2. Running OpenFOAM® on Remote Computers via SSH (e.g. server):

2.1. Windows:

- Run PuTTY (search for: PuTTY windows).
- Set the following:

  `Category>Session`

  Host name: openhost.university.edu

  Connection type: SSH

  `Category>Connection>SSH>Tunnels`

  Source port: 5901

  Destination: localhost:59**[1]

- Do not forget to press *Add*!

  Please make sure that display is not used by others.

  `Category>Connection>Data`

  Auto-login username: openFoamUser[2]

  `Category>Session`

  Saved Sessions: openFoamUser

- Press *Save.*
- Now choose from “saved sessions” your session (*openFoamUser*) and press *Open*. In the opened Command (Prompt) window, it prompts for

[1] Display number

[2] Session ID

your password. The password is not echoed to the screen and the passwords are case sensitive.

- Immediately after entering your password, your computer will attempt to establish a connection to your server. If it is your first time connecting to that server, you will see a message asking you to confirm the identity of the machine. Make sure you entered the address properly, and type *yes*, followed by the *return* key, to proceed.
- Change to etc directory in OpenFOAM® installation directory
- Execute the following command:

```
>. ./bashrc
```

- To log out use whatever command is used to logout from the server you are logged into (typically ctrl + d).

2.2. Mac OS X and Linux:

- Open your Terminal application. You will see a window with a $ or > symbol and a blinking cursor. From here, you may issue the following command to establish the SSH connection to your server (be careful about upper case 'L' in the -gL).

```
>ssh -gL 5901:localhost:59** openFoamUser@university.edu
```

- Immediately after issuing this command, your computer will attempt to establish a connection to your server. If it is your first time connecting to that server, you will see a message asking you to confirm the identity of the machine. Make sure you have entered the address properly, and type *yes*, followed by the return key, to proceed.
- You will then be prompted to enter your password. Type or copy/paste your SSH user password into the Terminal. You will not see the cursor moving while entering your password. This is normal. Once you are finished inputting your password, press return on your keyboard. At this point, you will be connected to your server remotely through SSH.
- Change to etc directory in OpenFOAM® installation directory
- Execute the following command:

```
>. ./bashrc
```

2.2.1. Running OpenFOAM® in Graphical Interface (VNC):

- Connect to remote machine via SSH connection using part B.

- Make sure VNC Server is installed on the remote machine and it is started (ask administrator for display number, port and other information, for starting VNC Server check FAQ)
- Install the appropriate VNC Viewer and run it (search for: vnc viewer):

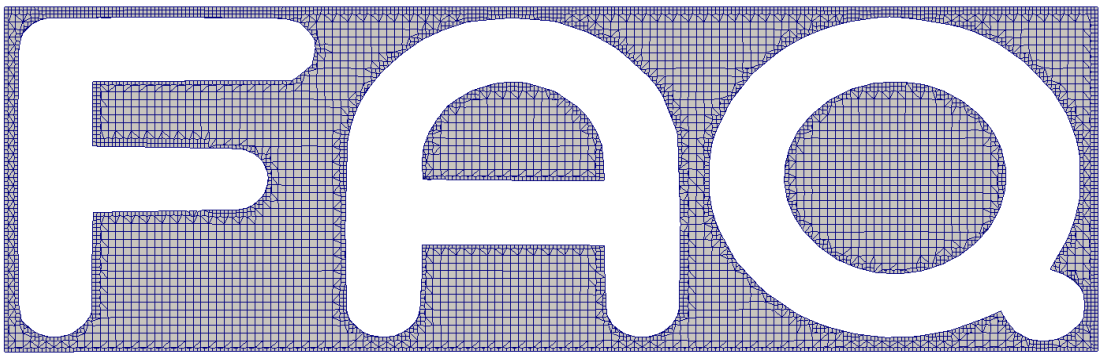
  VNC Server: localhost:01
- Press Connect
- Press Continue
- Enter your password
- Press *Ok*
- On VNC desktop open a terminal
- Change to etc directory in OpenFOAM® installation directory
- Execute the following command:

```
>. ./bashrc
```

If a new terminal in the VNC desktop is opened, the last two steps should be done in that to activate OpenFOAM® in there.

Appendix C

Frequently Asked Questions



How the mesh has been created?!

Bahram Haddadi

ENGINEERING
FLUID
DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Sylvia Zibuschka
- Yitong Chen
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

Q - What should I do in case of a GAMBIT failure?

A - e.g. Program stops responding:

- Type "ps" in the command window, search for the GAMBIT process number.
- "kill -9 PROCESS NUMBER" Enter

GAMBIT creates lock files, which must also be deleted, otherwise it is not possible to open of the affected files:

- "rm *. lok" Enter

Furthermore, "junk" (temporary files from GAMBIT) should be disposed of:

- "rm -fr GAMBIT.xxx" erases the complete directory, xxx again is the process number.
- If you have forgotten, to save before the crash, you should copy the file "jou" (it contains all the commands that have been executed and can be processed automatically in GAMBIT) from the directory, to resume its status before the crash.

Q - How can I prevent typing long commands in the terminal for couple of times?

A - Using curser keys to navigate line by line.

Type beginning of the command and use Tab (auto completion).

By using reverse search, use ctrl+r to search for previous commands typed in the terminal, e.g. typing a part of command show the suggestions and you can navigate through them.

Q - My VNC is not responding from server side?

A - First you should kill your VNC server:

>vncserver -kill :[YOUR DISPLAY NUMBER]

Restart your VNC server (according to SSH forward):

>vncserver:[YOUR DISPLAY NUMBER] -geometry 1600x800 -depth 24

Q - I have deleted some of my files accidently. What should I do?

A - Sorry, no recycling or undelete in Linux

Q - Why can I not connect to the server?

A - Check to see if you have an IP address for your network card.

Q - How can I start VNC Viewer from my Linux computer terminal?

A - Use command:

>vncviewer :[NUMBER OF LOCAL PORT, e.g. 1 or 2]

Q - Error “command not found”?

A - Make sure OpenFOAM® and ParaView are installed correctly. Check Appendix B for starting OpenFOAM®.

Q - Does foamToVTK command not work for chtMultiRegionFoam?

A - Use command:

>foamToVTK -region[REGION NAME]

Q - Is it possible to export animations from ParaView?

A - Yes, by choosing .ogv file format from “file/save animation” menu. The output will be a video file with .ogv format. In the new ParaView versions (newer than 4.3.0) the animation can also be saved using .avi format.

Q - Is there any tool in Linux to convert series of ParaView pictures to video?

A - Yes, command line tool ffmpeg:

>ffmpeg -r [FRAME PER SECOND RATE] -f image2 -i [images names, e.g. rho.%4d.jpg] [OUTPUT FILE NAME].[OUTPUT FILE FORMAT, e.g avi]

Q - How can complex geometries be patched?

A - During creating the geometry in the preprocessing software, e.g. GAMBIT, create volume zones, which you will need to patch later (see software user manual for creating regions in each software). For converting the mesh to the OpenFOAM® mesh use the appropriate tool with “-writeZones” flag to import zones to OpenFOAM®, e.g.:

>fluentMeshToFoam -writeZones <your mesh>

then in the setFieldDict change it like this:

```
regions
(
    zoneToCell
    {
        name air; // region name which you assigned in gambit
        fieldValues
        (
            volScalarFieldValue alpha.water 0 // the value of property
                                              //which you want to patch
        );
    }
);
```

Then after running setFields tool, it will assign the values to that region.

Q - How can I create a bash scripting file for executing couple of command in series?

A - Instead of typing command sequences one by one after each other and executing them. It is possible to put all those commands in a file and execute that file to run them. This is known as “bash scripting”.
Bash scripting is typically used in the cases when the same simulation should be run with identical settings a couple of times, but with a few changes. For bash scripting create an empty file (e.g. using nano editor creating text file “go”):

> nano go

Add the commands to this file (e.g. commands for running blockMesh, setFields, decomposePar, compressibleInterFoam in parallel mode and reconstructPar):

```
blockMesh
setFields
decomposePar
mpirun -np 4 compressibleInterFoam -parallel >log
reconstructPar
```

Exit the editor and save the file (ctrl+x , y, enter for nano editor).
For changing this file to an executable file, file permissions should be set. By using this command file permissions are displayed:

>ls -la go

```
-rw-r--r-- 1 e166**** E020D166 73 Aug 23 9:15 go
```

The first ‘r’ shows that this text file can be read by user, the ‘w’ shows that user has the permission to write this file, but the ‘–‘ sign shows that this file is not executable by the user. To change this permissions execute following command:

>chmod u+x go

Now this file is executable:

>ls -la go

```
-rwxr--r-- 1 e166**** E020D166 73 Aug 23 9:15 go
```

Now you can run the simulation by this executable text file:

>./go

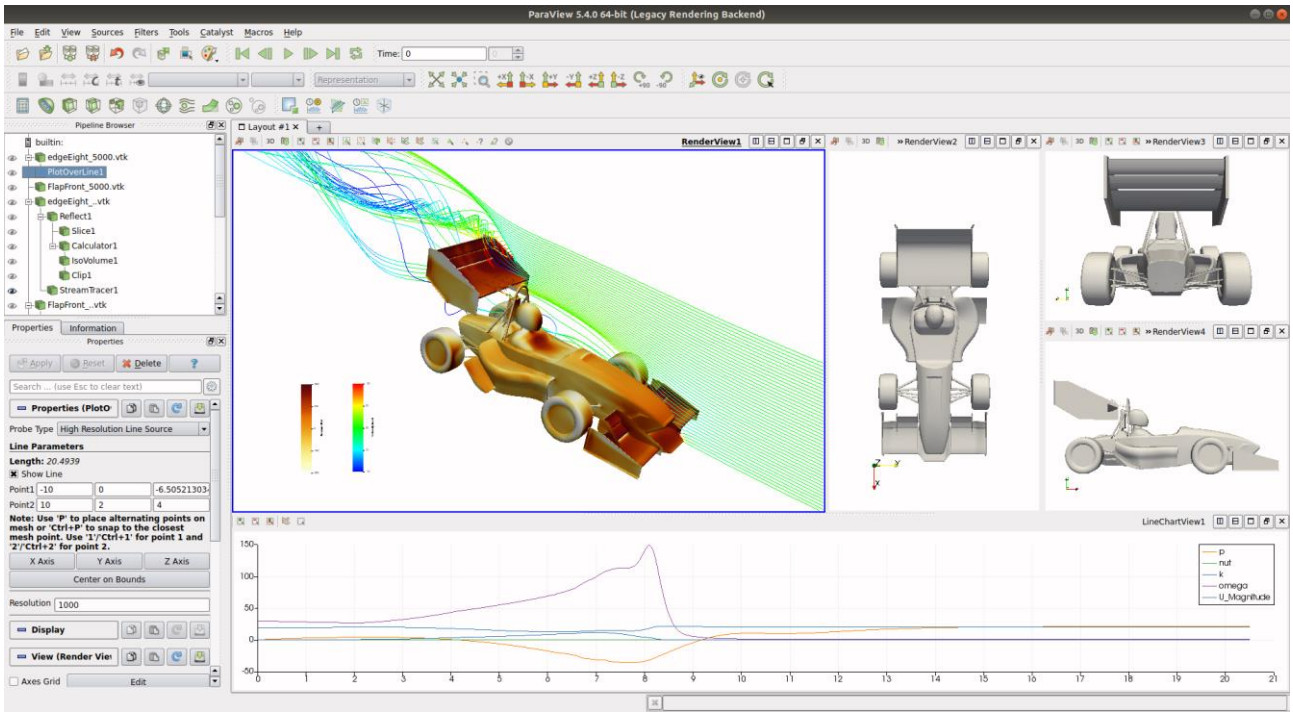
After executing the file, the commands added to the file will be executed one by one. In most of the OpenFOAM® tutorials there are **Allrun** and **Allclean** files, which are bash scripts for running the case and cleaning a case, respectively.

Q - How the cover mesh has been created?

A - Error: invalid question!

Appendix D

Paraview



Bahram Haddadi

ENGINEERING FLUID DYNAMICS

7th edition, March 2025

Contributors:

- Bahram Haddadi
- Christian Jordan
- Michael Harasek
- Sylvia Zibuschka
- Yitong Chen
- Jozsef Nagy

TU WIEN ICEBE IMAGINEERING NATURE

Technische Universität Wien

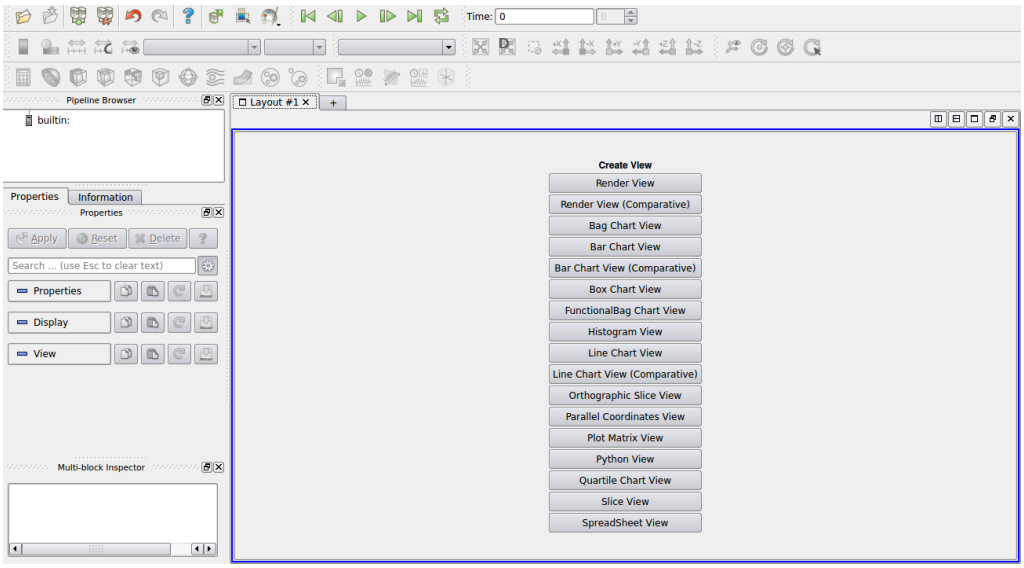
Institute of Chemical, Environmental

& Bioscience Engineering

Available from: www.fluiddynamics.at

1. Introduction to ParaView

The post-processing application for OpenFOAM® is ParaView, which is a free, open source program. In this tutorial, different features and tools available in ParaView 5.10.1 will be explored.



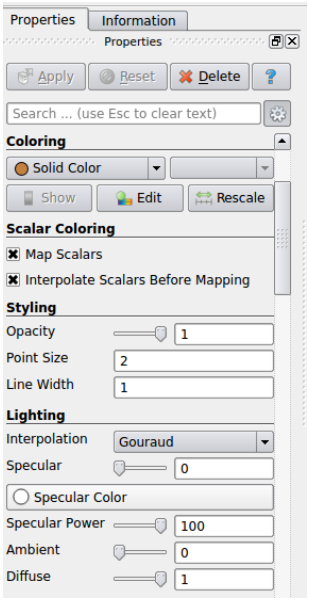
ParaView Interface

The tree structure (“pipeline”) of ParaView helps the user to easily choose and display suitable sub-models for creating the desired image or animation. Adding a mesh or velocity vectors to a contour plot of pressure is an example of this functionality.

For generation operations, use the OpenFOAM® command *foamToVTK* to convert OpenFOAM® files into readable formats for ParaView. Then open the .vtk file and press the green *Apply* button in the Properties panel. The reset button is used for resetting the window and deleting the selected operation.

2. ParaView Interface

2.1. Properties Panel



Properties panel

- **Colouring**

The drop-down menu for solid colour allows different field variables to be chosen and viewed, for example, pressure and velocity magnitude.

The Rescale button allows the data range to be adjusted to fit the data, as sometimes the max/min data range are not updated automatically.

- **Scalar Coloring**

The '*Map Scalars*' option allows the scalar values to be mapped to a specific colour using a lookup table.

Turning the option *'Interpolate Scalars Before Mapping'* on or off will affect the way the scalar data is visualized with colours. According to the ParaView documentation, if it is turned on, scalars will be interpolated within polygons and colour mapping will happen on a per-pixel basis; if off, color mapping occurs at polygon points and colors are interpolated, which is generally less accurate[1].

- **Styling**

The opacity of the image can be set (1 = solid, 0 = invisible) in the *Opacity* option.

- **Lighting**

There are two options for *Interpolation*, *Gouraud* or *Flat*. With *Gouraud* shading enabled, normals are defined only per point and no face normal is needed. If the *Interpolation* is changed to *Flat*, only the face normals will be computed and used for lighting, note that this option is not suitable for objects with smooth surfaces [2].

- **Backface Styling**

This is an advanced feature in ParaView that enables the backface style of a wire frame object to be changed.

- **Transforming**

The Transform filter allows you to translate, rotate and change the size and the origin of the data sets.

- **Miscellaneous**

By default ParaView triangulate the cells and shows them as triangles. For disabling this uncheck the "*Triangulate*" option in the Miscellaneous section of the Properties panel.

- **Glyph Parameters**

The Glyth Parameters filter generates a glyph, which can be arrow, cone, box, cylinder, line, sphere or a 2D glyph. The glyth is generated at each point in the input dataset[3]. Depending on the type of glyph chosen, different options are available to orientate, scale and size the glyph.

- **Orientation Axes**

The Orientation Axes feature controls an axes icon in the image window (e.g. to set the color of the axes labels x, y and z).

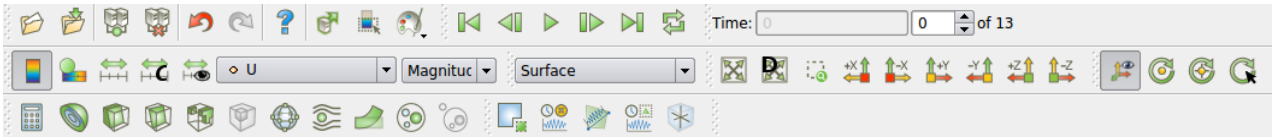
- **Lights**

The lighting controls options appear when clicking on the *Edit* button. For producing images with strong bright colors (e.g. isosurface) Headlight of strength 1 is appropriate.

- **Background**

The background color of the layout can be chosen from the drop-down menu, with types *Single color, Gradient* and *Image* available.

2.2. Button toolbars

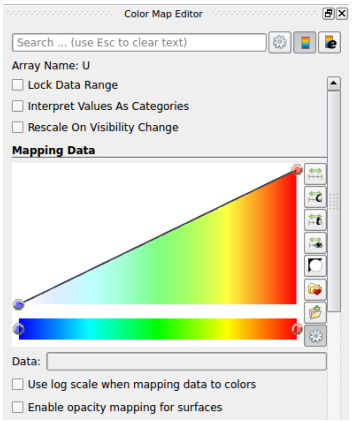


Button toolbars

Pull-down menus at the top of the main window and the major panels, in the toolbars below the main pull-down menus increase the functionality of ParaView. The function of each button can be easily understood by its icon, also any button description can be found in the Help menu (keeping the mouse over an icon without clicking on it will also give a short explanation on its functionality).

A feature worth mentioning is the drop-down menu next to the *Reset* button, this provides the options of the different ways of presenting the mesh. To see the structure of the mesh, use *Surface with Edges*; and to see both the cell structure and the interior of the mesh, use *Wireframe*.

2.3. Color Map Editor Panel



Color map editor

The *Choose preset* button allows the color scheme of the scale to be chosen, a common color scheme used is Blue to Red Rainbow. The *Rescale to custom range* button allows the maximum and minimum values of the color scale to be freely chosen by the user.

Another important feature can be used by clicking the button *Edit color legend properties* on the top right of the panel, this allows the scale title, font style to be changed.

3. Manipulating the view

3.1. Contour plots

Clicking on the *Contour* button in the Button Toolbars creates a contour plot. The contour filter operates on any type of data set, but requires the input to have at least one point-centered scalar (single-component) array. The output of this filter is polygonal.

The chosen scalar field can be selected from a pull down menu. If the case is a 3D case module, the contours will be a set of 2D surfaces that represent a constant value. The *Isosurfaces* list in the Properties panel allows the user to specify the values at which the isosurfaces are computed.

3.2. Introducing a cutting plane

Creating contour plots across a plane is more convenient than isosurfaces. Cutting planes are the tools which can be used for this purpose, to create surfaces. This can be done by clicking on the Slice button in the Button Toolbars. A cutting plate can be manipulated and repositioned. In a similar way, the contour lines can also be derived out of planes.

By default ParaView triangulate the cells and shows them as triangles. For disabling this uncheck “triangulate the slice” option in the Properties panel of the slice.

3.3. Streamlines

To create tracer lines, click on the Stream Tracer button in the Button Toolbars. Tracer points can be along a line or points, and this can be chosen in the Seed Type drop-down menu in the Seeds section of the Properties panel. Usually, some trial and error is needed for achieving the desired streamlines. The length of steps tracer takes can be changed in the Streamline Parameters section of the Properties panel. A smaller length increases calculation time but increases smoothness. For having high quality images Tubes filter can be used after tracer lines have been created. There are different types of tubes, not only cylindrical.

3.4. Vector plots

The Glyph filter is used for creating vector plots. Scale Mode menu in the properties panel is used for:

- Setting the length of a vector, weather to be proportional to vector magnitude or not, all with the same length (Vector).
- Controlling the base length of the glyphs (set Scale Factor).

4. Data Analysis

4.1. Plot over time

This option is available by clicking the *Plot Selection Over Time* button in the Button Toolbars. This allows the data at one point to be plotted over the entire time range.

4.2. Plot over line

This option allows the data points to be plotted along a line at a specific time step. Click on the *Plot Over Line* button. The Cartesian coordinates of the beginning and ending points of the line can be specified in the Properties panel. Several variables can be plotted at the same time, to turn each variable on or off and to change its legend name, use the Series Parameters section in the Properties panel.

4.3. Integrate Variables

The *Integrate Variables* option is selected from the Filters menu. This tool integrates point and cell data over lines and surfaces. It also computes length of lines, area of surface, or volume[4]. Different data types available are *Point Data, Cell Data*, or *Field Data*; this can be chosen in the *Field Association* section in the Properties panel.

5. Exporting Data

5.1. Image Output

For creating a screenshot of the graphs, the easiest way is Save Screenshot from File menu. After selecting it in the opened window, the picture resolution

can be set, and by locking the aspect ratio, changing image resolution in one direction cause change in its resolution in the other direction respectively. For high quality images, a resolution of more than 1000 pixels is a good choice.

5.2. Animation Output

Some animations can be saved in ParaView by selecting the *Save animation* option in the File menu. The resolution and number of frames per time step can be specified. You can save your animation by assigning a name and choosing the file format. The most suitable file format is *.ogv*.

5.3. Data Output

The field values of a chosen variable (e.g. temperature or pressure) can be exported into Excel using the Save Data option in the File menu. The precision of the writer can be chosen and there is an option to export data from all time steps.