NTNU

Norwegian University of
Science and Technology

# Modelling a tracer injection and sensor manifold

Abel Tenu Mekonnen

Chemical Engineering
Submission date: July 2016
Supervisor: Heinz A. Preisig, IKP

Norwegian University of Science and Technology
Department of Chemical Engineering

# Summary

Residence Time Distribution (RTD) of a rector system is the probability function that describes the time a fluid element spends inside the system. RTD is used by most chemical engineers to describe the hydraulics within a system and also to compare the behaviour of the real equipment with their respective ideal models. The technique is used both for design and troubleshooting. Experimentally, the non-reacting tracer is introduced into the inlet of the system and the concentration is measure at the inlet and the outlet. The signals are then used to compute the RTD. The ideal excitation signal contains all frequencies this is either a pulse or its integral, the step. The quality of the two signals thus determines directly the quality of the computed.

This thesis focuses on the design of a tracer injection and sensor manifold. The entire thesis work is conducted in two parts: first, to validate the design of the in-house conductivity sensor and second, to design the tracer injection using a CFD simulation tool, for which OpenFOAM is used.

To validate the design of the in-house conductivity sensor, several simulations using the simpleFoam solver were performed utilising tetrahedral and hexahedral meshes of the liquid body in the sensor. The simulations were used to visualize the flow behavior in the form of the velocity and pressure fields. The objective of performing the simulation for the two types of mesh is to find how the results are affected by the choice.

To design the tracer injection part, several simulations using the simpleFoam and the scalarTransportFoam solvers were performed. These were used to visualize the tracer cloud formation and the distribution of the tracer throughout the sensor for a given injector design. Two types of a 90 degree bent cylinder injector design, one with large length and a needle shape at one end of the geometry and the other is circular shape in both ends with small length, were studied. For both injector designs, a counter-current injection flow direction was used.

In terms of the execution time and RAM usage, the hexahedral mesh performed better than the tetrahedral mesh of the sensor. But in terms of the quality of the mesh and the accuracy of the result of simulations, the tetrahedral mesh of the sensor was better than the hexahedral mesh. The result of the simulation for the tetrahedral mesh agrees well with the experiment showing only 4.8% error. Results of the simulations showed that the conductivity sensor has a high pressure drop due to different cross sectional areas in some parts; the main effects are due to the rozita and cone part. Hence, there is a need for an improvement of either the rozita and cone parts or design completely a new sensor.

Evolution and distribution of the tracer cloud passing through the sensor is affected by the shape and size of the injection tube and the tracer flow direction. The 90 degree bent cylinder with needle shape at one end of the injector gives unexpected result, as the tracer cloud is formed at the top of the pipe and not as expected at the bottom. This result also contradicts result from previous work of Oscar Pujol, which gives a tracer cloud formation at the center of the pipe near the tip of the injector for the same injector design. The second injector design

gives a good mixing and distribution of the tracer. Based on the simulation I recommended to adding a mixer after the injector and before the sensor inlet in order to improve the mixing and distribution of the tracer concentration.

Salome is used to generate all the geometries and tetrahedral meshes and ”snappyHexMesh” is used to generate the hexahedral meshes. All simulations were running on one of NTNU’s high performance computer.

# Acknowledgements

First of all I like to sincerely appreciate my supervisor, Prof Heinz for his immense support, mentoring and especially for giving me a chance to do this thesis. I started this thesis with out any previous experience or knowledge about CFD, OpenFOAM and Linux but his massive tolerance helped me to developed this master thesis. I would like also to thank Mikael Hammer for his support to get all necessary tools to setup the experiment.

At last, my appreciation goes to Ph.D. student sigve karolius, friends and colleagues who at one time or the other offer word of advise and encouragement all through this thesis.

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations

| | | |
|---|---|---|
| CFD | = | Computational Fluid Dynamics |
| DILU | = | Diagonal Incomplete-LU |
| GAMG | = | Generalized Geometric-Algebraic Multi-Grid |
| MPI | = | Message Passing Interface |
| NTNU | = | Norwegian University of Science and Technology |
| PBiCG | = | Preconditioned Bi-Conjugate Gradient |
| RAS | = | Reynolds-averaged stress |
| RTD | = | Residence Time Distribution |
| STL | = | Stereolithography |
| 2D | = | Two Dimensional |
| 3D | = | Three Dimensional |

Chapter 1

# Introduction

## 1.1 Scope

RTD of a rector systems is the probability distribution function that describes the length of the time a fluid element spends inside the given system. RTD is used by most chemical engineers to describe the hydraulics within a system and also to compare the behavior of the real equipment with their respective ideal models. The technique is used both for design and troubleshooting.RTD is measured experimentally by introducing non-reacting tracer into the system at the inlet and measuring the concentration at the inlet and the outlet[3, 4]. Thus in order to measure the RTD, one requires an injector to inject the tracer at the inlet of the system and two sensors(one at the inlet and the second at the outlet of the system) to measure the trace concentration. This thesis is on the design of the conductivity sensor and tracer injection.

Model simulations are important tool to validate the feasibility of the design and to optimize the design before the apparatus is manufactured. Because of the growth of computational power and storage capacity, the development of validated and fast numerical procures; Computational Fluid Dynamics (CFD) is used to solve fluid flow related problems.

Nowadays, there are a lot of programs to solve CFD problems which are either developed by commercial companies or by open source communities. FLUENT, CFX and COMSOL are some of the best known programs which developed by commercial companies; OpenFOAM, SU2 and Code_saturn are the most known programs in the open source community.

The aim of this thesis is to use CFD simulations to validate the design of the in-house conductivity sensor and to design the tracer injection which yields the best distribution of tracer concentration throughout the sensor. A CFD simulations tool, called OpenFOAM is used for performing the simulation and software called Salome is used to generate the geometry and a mesh throughout this thesis work. This thesis was done in two parts namely:

- To validate the in-house conductivity sensor by performed simulation using simpleFoam solver. The simulation is used to visualize the flow behavior in the form of the velocity and pressure fields. A simple experiment was done to measure the pressure drop of the real conductivity sensor, and the result compared with the simulation result to check the viability of the simulation. The simulation was done utilising structured and unstructured

meshes of the liquid body in the sensor to see how the simulation result is affected by the mesh type and to get the most suited mesh type for the sensor.

- To design the injection system. Simulation were performed using the simpleFoam and the scalarTransportFoam solvers. This is used to visualize the distribution of the tracer concentration throughout the sensor.

All simulations were performed in parallel with 16 mpi processor using a supercomputer at NTNU, called Vilje.

## 1.2 Previous Work

This thesis is the continuation of my specialization course project on modeling of the Conductivity sensor using CFD simulation, which was conducted in fall 2015. The project focused on to validate the design of the in-house conductivity sensor using a CFD simulation tool, called OpenFOAM. The simpleFoam simulation was performed using a tetrahedral mesh of the sensor geometry. Without considering the design of the injection system and assuming the tracer being injected at the inlet of the sensor, simulation was performed using scalarTransportFoam solver. At the time, the simulations did not converge, which lead to this continuation project.

## 1.3 Structure of the report

Including this introduction section, the thesis report is arranged in five chapters as indicated below:
The over view of computational fluid dynamics including the basic theory of the three basic governing equations of CFD and an introduction about used software in this thesis outlined in **Chapter 2**.
**Chapter 3** briefly describes the three basic procedures of CFD used for modelling the conductivity sensor and injection system. This includes the procedure and basic theory to generate the geometry and mesh of the sensor and injector, defining boundary and initial conditions and the choice of the discretisation schemes and linear-solver settings. In addition a description on how to execute the OpenFOAM job on NTNU's supercomputer is included.
**Chapter 4** summarizes the result obtained and the discussion of the results.
Finally **Chapter 5** summarises the findings of the project and recommendations future work are given.

Chapter 2

# Theory

## 2.1 Introduction to CFD

CFD is a multidisciplinary topic which uses applied mathematics, physics and computational software to solve and analyse a problem that involves fluid flows as well as to visualize how an object affected when the fluids passed on it. A naiver-Stokes equation is the fundamental basis of CFD problems, which describe how the pressure, temperature, density, and velocity of a moving fluid are related. The required calculation to simulate the interaction of fluids with surface defined by boundary conditions performed, using a computer[2].

The main concept of CFD methods is to find the flow quantities value in the system at a large number of connected points which is called numerical grid or mesh. CFD methods used three basic procedures, called pre-processing, simulation, and post-processing to get the solution for a given case.

- Pre-processing: First the geometry of a given problem is defined. Then its volume which is occupied by the fluid is divided into discrete cells, called mesh or grid, then the physical modeling and boundary conditions are defined.
- Simulations: iteratively equations are solved.
- Post-processing: the solution result is analysed and visualized using post-processor tool.

## 2.2 The Governing Equations of CFD

Continuity, momentum, and energy equations are the three fundamental governing equations of fluid dynamics that talks about physics. All this three fundamental governing equations of fluid dynamics are the base for CFD.

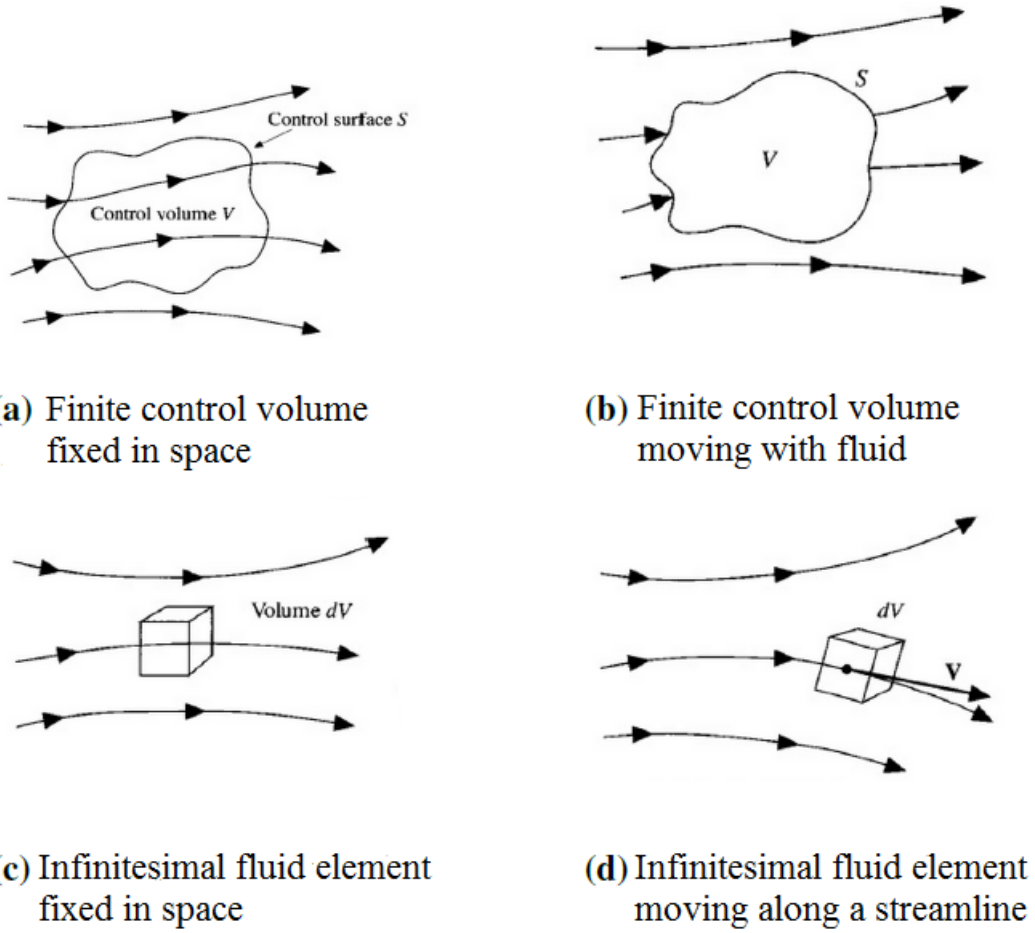
All of the fundamental governing equations of fluid dynamics are based on the three fundamental physical principles, which are:

- Conservation of mass
- Newton's second law, and

- Conservation of energy

These fundamental physical principles applied to a suitable model of the flow to produce mathematical statement of the governing equations, either in conservation or non-conservation form. The form of the governing equation is depending on which flow model is used to derive it.

For a continuum fluid, there are four basic model of flow which is used to driving the governing equation. Two of the four model have a finite control volume either control volume fixed in space with the fluid moving through it or the control volume moving with the fluid such that the same fluid particles are always in the same control volume(Figure 2.1(a) and 2.1(b)). The other two of the four flow model have infinitesimal small volume either it fixed in space, where the fluid is moving through it or the volume moving alongside with a streamline (Figure 2.1(c) and 2.1(d)).




**(a)** Finite control volume fixed in space

**(b)** Finite control volume moving with fluid

**(c)** Infinitesimal fluid element fixed in space

**(d)** Infinitesimal fluid element moving along a streamline

**Figure 2.1:** Models of a flow[1]

### 2.2.1 The continuity equation

The continuity equation is the one of the governing flow equation which is obtained by applying the conservation of mass principle to a suitable model among the four model of flow. There are four forms of continuity equation depend on the four model of flow, but mathematically all are the same. For this report the finite control volume fixed in space model of flow is used to derive the continuity equation.

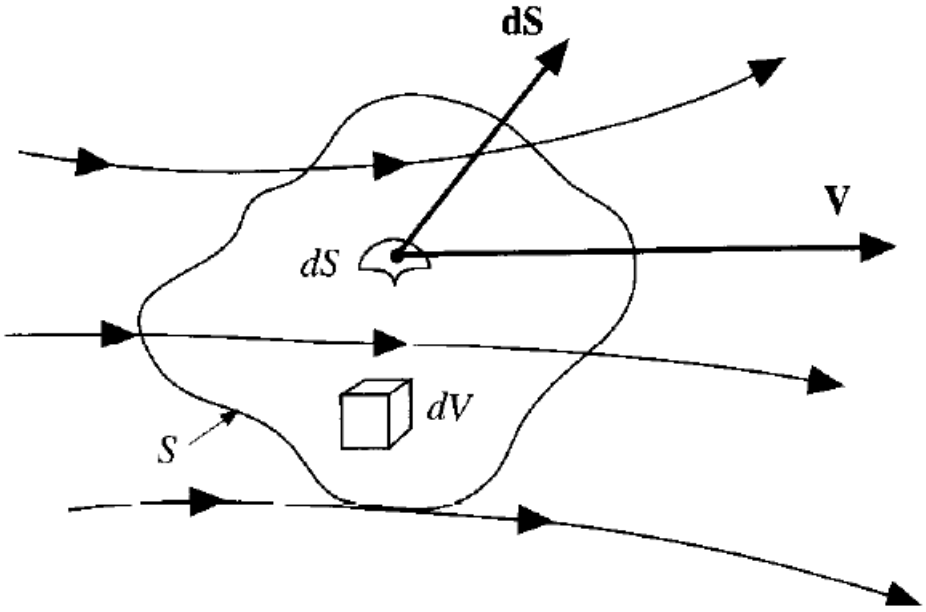



**Figure 2.2:** Finite control volume fixed in space[1]

This model shown at Figure 2.1(a).The control volume is fixed in space and it bounds by the surface which is called the control surface. The detail of the flow model is shown in Figure 2.2, where **V** is the flow velocity, **dS** is the vector elemental surface area, and dV is an elemental volume inside the finite control volume. Applied the conservation of mass principle to this control volume,which is expressed as follows

$$X = Y \tag{2.1}$$

Where, X is the net mass flow out of control volume through surface S and Y is the time rate of decrease of mass inside control volume.
First find an expression for the left side of Eq.2.1. Across any fixed surface the mass flow of a moving fluid is equal to the product of density and volumetric flow rate of the fluid, which is itself the product of area of surface and component of velocity perpendicular to the surface. Therefore the elemental mass flow across the area **dS** will be:

$$\rho V n dS = \rho \mathbf{V}.\mathbf{dS} \tag{2.2}$$

As shown from the Figure 2.2, **dS** always points out of the control volume, therefore the sign of $\rho$**V**.**dS** depend on the direction of **V**. It will be either positive (means outflow) when **V** points out of the control volume or negative (means inflow) when **V** points into the control volume. The summation over S of the elemental mass flow which expressed in Eq.2.2 gives the net mass flow which out from the entire control volume through the control surface. Hence, the left side of Eq.2.1 becomes

$$X = \iint_S \rho \mathbf{V}.\,\mathbf{dS} \tag{2.3}$$

Now similarly find an expression for the right side of Eq.2.1. In the elemental volume dV, mass will be product of density and dV. Therefore the total mass inside the control volume will be:

$$\iiint_V \rho \, dV \tag{2.4}$$

The rate of mass increase with time inside V is then

$$\partial \iiint_V \rho \, dV \partial t \tag{2.5}$$

The above equation can be express as follow

$$\frac{\partial \rho}{\partial t}(dxdydz) \tag{2.6}$$

And it is equal to the rate of mass increase with time inside the element. The right side of Eq.2.1 becomes the negative of Eq.2.6.

The conservation mass principle is said that the net mass flow out from the given system must equal to the rate of mass decrease with time inside the given system. Applied this principle to the fixed element in Fig.2.2, which gives the expression as follows:

$$[\frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} + \frac{\partial \rho w}{\partial z}]dxdydz = -\frac{\partial \rho}{\partial t}(dxdydz) \tag{2.7}$$

by doing simple rearrangement it becomes:

$$\frac{\partial \rho}{\partial t} + [\frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} + \frac{\partial \rho w}{\partial z}] = 0 \tag{2.8}$$

The term in brackets in Eq.2.8, is the divergence of $\rho\mathbf{V}$ which is expressed with $\nabla .(\rho\mathbf{V})$. Therefor Equation 2.8 becomes:

$$\frac{\partial \rho}{\partial t} + \nabla.(\rho\mathbf{V}) = 0 \tag{2.9}$$

This is the continuity equation in the form of a partial differential equation. Equation 2.9 derived based on an infinitesimally small element that's why the equation is in the form of partial differential equation and also the element is fixed in space such that the equation is in the form of conservation.

### 2.2.2 The momentum equation

To derive the momentum equation, newton's second law will apply to a model of a flow. Newton's second low stated that: the net force on the fluid element is the product of its mass and acceleration of the fluid element. Newton's second low mathematically written as follow:

$$\mathbf{F} = m\mathbf{a} \tag{2.10}$$

Both the four models of fluid can be used to derive the momentum equation, but the infinitesimal fluid element moving along a streamline, which is shown in the Figure 2.1(b) is chosen to derive the momentum equation in this report.

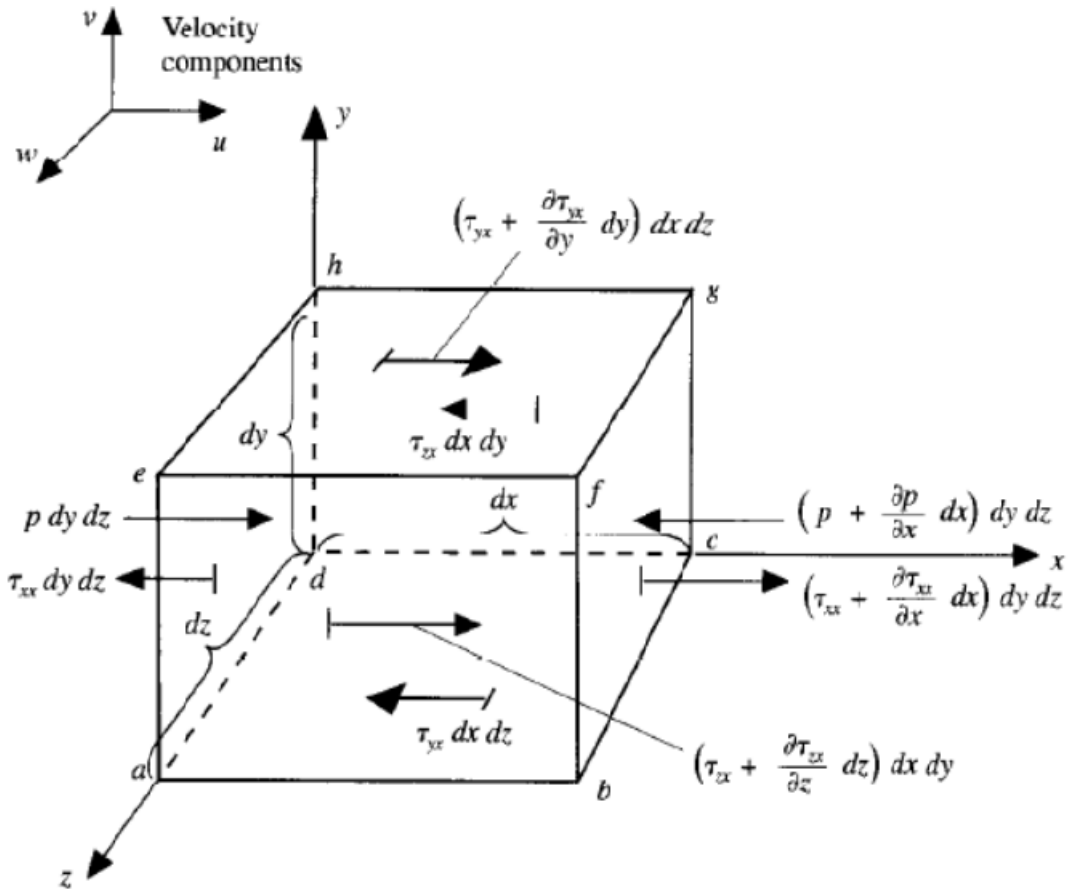



**Figure 2.3:** Forces in the x-direction acting on infinitesimally small, moving fluid element[1].

Figure 2.3 shows the surface force in the x-direction which acting on the infinitesimally small, moving fluid element and it used to derive the x component of the momentum equation. Newton's law which expressed mathematically in Eq.2.10 only show the vector relation, and it can be express in three direction, i.e in x, y, and z direction as scalar form. So Eq.2.10 written as the scalar form only in the x direction as follows:

$$\mathbf{F}_x = m\mathbf{a}_x \tag{2.11}$$

The left side of Eq.2.11 tells us the moving fluid element exposed by forces in the x direction and there are two kinds of this force namely:

- The body force: the force that act directly on the mass of the fluid element volume. Gravitational, centrifugal, and electromagnetic forces are some of the body force.
- The surface forces: the force that act directly on the surface of the fluid element. These force is the result of two sources: the pressure distribution acting on the surface of the fluid element, introduced by the outside fluid which surrounding the volume of the fluid element; and the shear and normal stresses distributions acting on the surface of the fluid element, it is the result of the friction between the outside fluid and the surface of the fluid element.

The x component of the body force acting on the fluid element denote by B express as follow:

$$B = \rho f_x(dx\,dy\,dz) \tag{2.12}$$

Where, $f_x$ is the x component of the body force per unit mass acting on the fluid element and (dx dy dz) is the fluid element volume.

The shear and normal stresses are related to the time change of the shearing deformation and the volume of the fluid element, respectively. Both denote by $\tau$ with different directions.
The net surface force acting on the fluid element in the x direction express mathematically as follow:

$$\begin{aligned} D = [p - (p + \frac{\partial p}{\partial x})]dy\,dz + [(\tau_{xx} + \frac{\partial \tau_{xx}}{\partial x}dx) - \tau_{xx}]dy\,dz \\ +[(\tau_{yx} + \frac{\partial \tau_{yx}}{\partial y}dy) - \tau_{yx}]dx\,dz + [(\tau_{zx} + \frac{\partial \tau_{zx}}{\partial z} - \tau_{zx})]dx\,dy \end{aligned} \tag{2.13}$$

Where, D is the net surface force acting on the fluid element in the x direction.

The total force acting on the fluid element is the sum of the body force and surface force acting on the fluid element. Therefore, the x component of the total force found by adding Eq.2.12 and Eq.2.14 and cancelling some terms, it denote by $F_x$ and express as follow:

$$\mathbf{F}_x = [-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z}]dx\,dy\,dz + \rho f_x dx\,dy\,dz \tag{2.14}$$

This equation, Eq.2.14 is the expression of the force term in Eq.2.11.
The right-side of Eq.2.11 is the product of mass and the x component of acceleration of the fluid element.
Mass of the fluid element which denote by m is the product of the density and volume of the fluid element and express as follow:

$$m = \rho\,dx\,dy\,dz \tag{2.15}$$

The acceleration of the fluid element is the time rate of change of its velocity, and the x component of the acceleration is denote by $a_x$ which mathematically expressed as follow:

$$a_x = \frac{Du}{Dt} \tag{2.16}$$

The combinations of Eq.2.14, Eq.2.15 and Eq.2.16 give the x component of the momentum equation for a viscous flow. And by using similar procedure the y and z component of the momentum equation can be found. Both the expression of the momentum equations, that means x, y, and z components shown below:

$$\begin{aligned} \rho\frac{Du}{Dt} &= -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \quad (a) \\ \rho\frac{Dv}{Dt} &= -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \quad (b) \\ \rho\frac{Dw}{Dt} &= -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z \quad (c) \end{aligned} \tag{2.17}$$

Eq.2.17 (a) to 2.17(c) is the x, y, and z component of the momentum equations in the form of non-conservation, respectively. Eq.2.17 is called the Navier-Stokes equations which is the name

of the founders of the equation. The Navier-stokes equations can be also obtained in the form of conservation by applying the definition of the substantial derivative only in the left-hand side of Eq.2.17.

Let consider the x component of the Navier-Stokes equation and the left-hand side of the equation re-write as follow:

$$\rho\frac{Du}{Dt} = \rho\frac{\partial u}{\partial t} + \rho\mathbf{V}.\nabla u \tag{2.18}$$

By expanding and rearranging the derivative term in the right-hand side of Eq.(2.18), the derivative term express as follow:

$$\rho\frac{\partial u}{\partial t} = \frac{\partial(\rho u)}{\partial t} - u\frac{\partial \rho}{\partial t} \tag{2.19}$$

Similarly, by doing the vector identity for the divergence of the product of a scalar and a vector and rearranging the divergence term in the right-hand side of Eq.2.18 re-wright as follow:

$$\rho\mathbf{V}.\nabla u = \nabla.(\rho u\mathbf{V}) - u\nabla.(\rho\mathbf{V}) \tag{2.20}$$

By substitute Eq.2.19 and Eq.2.20 into Eq.2.18 and rearranging, the following equation obtained:

$$\rho\frac{Du}{Dt} = \frac{\partial(\rho u)}{\partial t} - u[\frac{\partial \rho}{\partial t} + \nabla.(\rho\mathbf{V})] + \nabla.(\rho u\mathbf{V}) \tag{2.21}$$

The term in brackets in Eq.2.21 is the same with the left-hand side of Eq.2.9 which is the continuity equation and it is equal to zero. So Eq.2.21 reduces to:

$$\rho\frac{Du}{Dt} = \frac{\partial(\rho u)}{\partial t} + \nabla.(\rho u\mathbf{V}) \tag{2.22}$$

By substitute Eq.2.22 into Eq.2.17 (a), the x component of the Navier-Stokes equation in the form of conservation is obtained. And with similar procedure the y and z component of the Navier-stokes equation in the form of conservation also found. Both the Navier-Stokes equations in the conservation form shown below:

$$\frac{\partial(\rho u)}{\partial t} + \nabla.(\rho u\mathbf{V}) = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \quad (a)$$

$$\frac{\partial(\rho v)}{\partial t} + \nabla.(\rho v\mathbf{V}) = -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \quad (b) \tag{2.23}$$

$$\frac{\partial(\rho w)}{\partial t} + \nabla.(\rho w\mathbf{V}) = -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z \quad (c)$$

### 2.2.3 The energy equation

The third physical principle, which is the conservation of energy, is applied to a model of a flow in order to derive the energy equation. The same with the momentum equation derivation the flow model of an infinitesimal fluid element moving along a streamline is used for deriving the energy equation. The conservation of energy principle, which is the first law of thermodynamics, stated that the rate of change of energy inside fluid element is the sum of the net heat flux

into the fluid element and the rate of work done on the fluid element due to body and surface forces.

The rate of change of energy inside fluid element is the combination of two energies: Internal energy due to the random motion molecules; and Kinetic energy due to translational fluid element motion.

The Energy equations which is deriving from using the flow model of an infinitesimal moving fluid element and applying the first law of thermodynamics is in the non-conservation form and mathematically written as follow:

$$\begin{aligned}\rho\frac{D}{Dt}(e+\frac{V2}{2}) = \rho q + \frac{\partial}{\partial x}(k\frac{\partial T}{\partial x}) + \frac{\partial}{\partial y}(k\frac{\partial T}{\partial y}) + \frac{\partial}{\partial z}(k\frac{\partial T}{\partial z}) \\ -\frac{\partial up}{\partial x} - \frac{\partial vp}{\partial y} - \frac{\partial wp}{\partial z} + \frac{\partial u\tau xx}{\partial x} + \frac{\partial u\tau yx}{\partial y} + \frac{\partial u\tau zx}{\partial z} \\ +\frac{\partial v\tau xy}{\partial x} + \frac{\partial v\tau yy}{\partial y} + \frac{\partial v\tau zy}{\partial z} + \frac{\partial w\tau xz}{\partial x} + \frac{\partial w\tau yz}{\partial y} + \frac{\partial w\tau zz}{\partial z} + \tau\mathbf{f}.\mathbf{V}\end{aligned} \tag{2.24}$$

In sum up, it is very difficult to solve all the three governing equations analytically as both equations are a coupled system of nonlinear partial differential equations, so numerical solutions is required. Numerical solutions only give the value of the flow quantities at connected points in the system, called numerical grid or mesh. Therefore, the decomposition of the domain into a number of smaller, non-overlapping sub-domains is very important. Besides, the decomposition of the domain, the differential equations that representing the flow must be converting into a set of algebraic equation by using a procedure, called discretization. The resulting algebraic equations can be linear or non-linear and it is usually large, so a digital computer is used to solve it.

## 2.3 Numerical grid or Mesh

As mentioned in the previous section, CFD requires the subdivision of the geometry into a number of smaller, non-overlapping sub-domains which contains points, nodes, faces, cells, volumes (in the case of three dimensional (3D)) to solve the flow physics, and the network of these sub-domains are called grid or mesh. The quality of the mesh is very important to get a better solution, since solver use the previous node solution as initial conditions for solving the next node. The time needed to solve the problem, the speed of convergence, usage of the RAM memory and the accuracy of the solution highly dependent on the granularity of the mesh. In general, the finer mesh gives more accurate solution but it consuming more time and RAM to generate the solution.

The mesh can be uniform or non-uniform in terms of size of the mesh element: Uniform mesh has approximately equal size elements, while non-uniform mesh has non equal size elements. Both kind of mesh have their own advantages, for example non-uniform mesh is advantageous than uniform mesh to capture nonlinearities and gradients in spatial locations where they are strong with minimal numerical error, while in some case where numerical error is dependent to a certain degree on the size difference between adjacent elements using uniform mesh is more advantageous over non-uniform mesh.

The mesh can be create using a different kinds of element shape such as triangles, quadrilaterals in 2D and tetrahedral, hexahedral, and prisms in 3D. The numbers of nodes in an element depends on the shape of the element. Three dimensional elements requires more computational time and memory required than two dimensional elements, since three dimensional elements have a larger number of nodes than two dimensional elements. Therefore, the most suited element shape for a particular application must be used when generating 3D meshes to produces the least practicable computational effort.

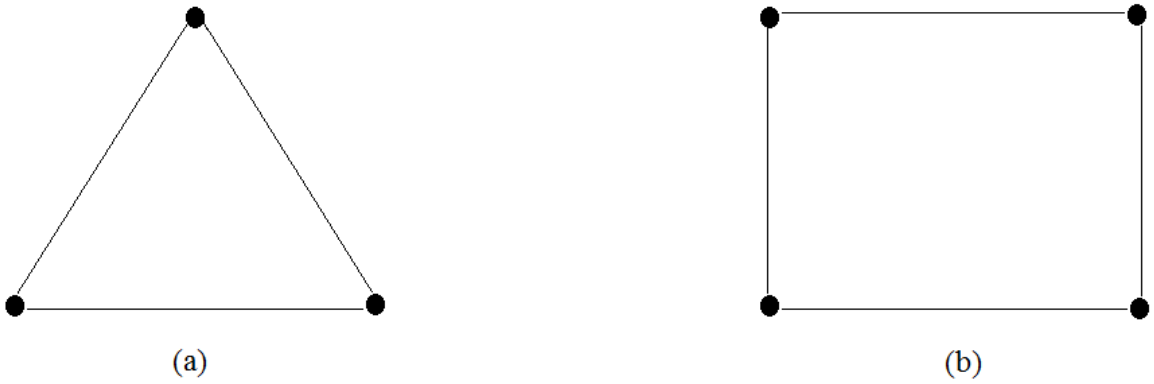



**Figure 2.4:** Two dimensional elements: (a) Triangle element has 3 nodes and (b) Quadrilateral element has 4 nodes

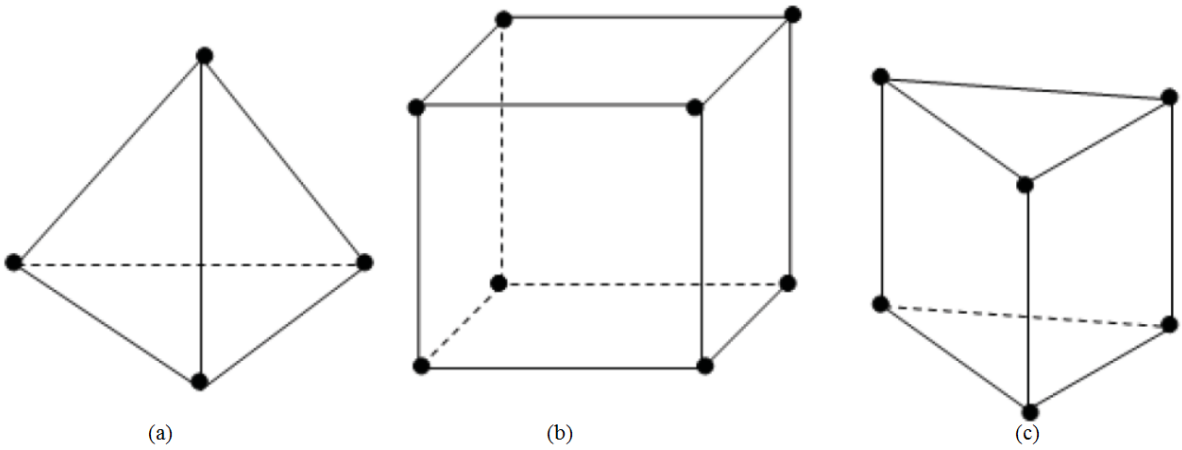


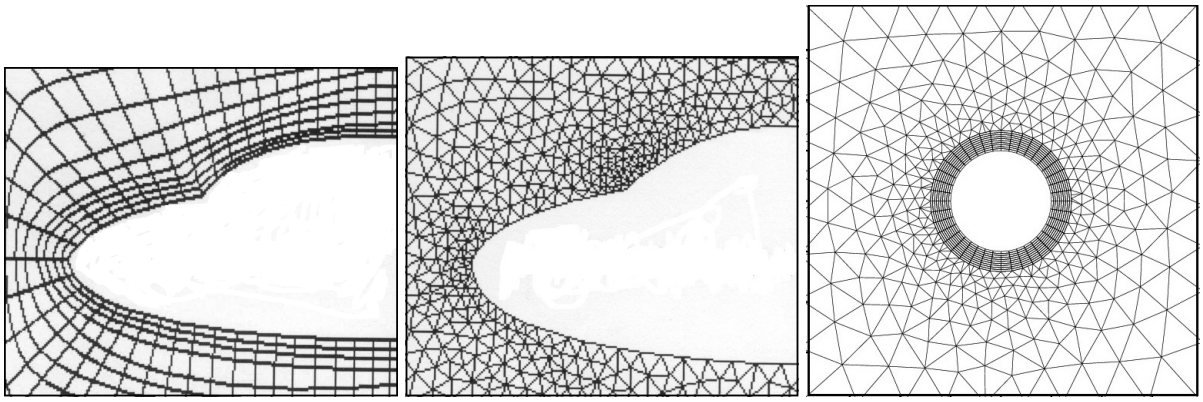

**Figure 2.5:** Three dimensional elements[5]: (a) Tetrahedral element has 4 nodes, (b) Hexahedral element has 8 nodes, and (c) Prismatic element has 6 nodes

Generally, based on the connectivity of points, there are three types of meshes: Structured, Unstructured, and Hybrid mesh.

- Structured mesh: in this type of mesh each point has the same number of neighbors, in other word structured meshes has regular connectivity. Quadrilateral and hexahedral elements are the possible element choice to produce structured mesh in 2D and 3D, respectively. This kind of mesh has a better convergence and higher resolution.
- Unstructured mesh: each point can have different number of neighbors, which means the mesh have irregular connectivity. Triangles and tetrahedral elements are the possible

choices to produce unstructured mesh in 2D and 3D, respectively. This kind of mesh is easy to make but it utilise high memory storage and needs longer time to compute.

- Hybrid mesh: this mesh is a combination of structured and unstructured mesh. This kind of mesh is very useful for a very complex geometry, which is difficult to make a structured mesh.



**(a)** Structured Mesh[11] **(b)** Unstructured Mesh[11] **(c)** Hybrid Mesh[12]

**Figure 2.6:** Types of Meshes based on the connectivity of points

The mesh can be created in the CFD package itself or in the third party software. The OpenFOAM mesh generator utility, called snappyHexMesh and the third-party mesh generator tool, called Salome are used to crate pure hexahedral and tetrahedral mesh of the conductivity sensor, respectively, in this thesis. In the next subsections the overview of the theory behind snappyHexMesh and Salome will be outline.

### 2.3.1 Salome

"SALOME is a free software that provides a generic platform for pre and post-processing for numerical simulation. It is based on an open and flexible architecture made of reusable components available as free software[7]."

Salome is free software and it is compatible with OpenFOAM. In Salome, the user can either make both the geometry and generates the mesh or import the geometry from other CAD software and generate the mesh only. It is easy to use, as it contains various drawing operations and a lot of different kinds of shapes to draw the geometry and a lot of algorithms for generate the mesh. Since Salome has a lot of different kind of algorithms and hypothesis, it is possible to generate unstructured, structured and hybrid mesh. The creation of fine mesh, especially tetrahedral mesh (unstructured mesh) is computationally heavy and as such requires powerful computer. Salome has several options for exporting the generated mesh, so the user can export the mesh in different file format depends on the software used for the CFD simulation later. The **.UNV** file format is used for importing the mesh into OpenFOAM.

### 2.3.2 SnappyHexMesh

SnappyHexMesh is the mesh generation utility supplied with the OpenFOAM. It generates a hexahedral mesh from a triangulated surface geometry in the Stereolithography (STL) format. It is imported from external CAD software and located in a constant/triSurface sub-directory of the case directory[8]. The geometry with the STL file format must be surround by a background mesh of hexahedral cells which is created using blockMesh utility of OpenFOAM before the snappyHexMesh is executed. The objective of the snappyHexMesh is generating the structured mesh( pure hexahedral mesh) of an object by cutting and refining of the background mesh which is intersect with STL surface and snapping it to the surface of an object. And finally it crates the boundary layers. The entire process is controlled by snappyHexMeshDict which is located in the ”system” sub-directory. The snappyHexMesh contains three main libraries or programs called:

- ”autoRefineDriver”: responsible for the cutting and refining of the mesh
- ”autoSnapDriver”: responsible for snapping of the mesh on to the surface of an object
- ”autoLayerDriver”: responsible for the Creation of the boundary layers

These three programs activates by setting true for castellatedMesh, snap, and addLayeres command which are found in the snappyHexMeshDict files, respectively.

SnappyHexMesh is suitable and effective, probably the quicker to generate a complex structured mesh. But for the very complex geometry it’s too difficult to make a high quality mesh, especially in the boundary layer regions, since it generates its own mesh by cutting the background mesh through the surface.

## 2.4 OpenFOAM

OpenFOAM is a free Linux-based CFD Toolbox distributed by the OpenFOAM foundation which includes Pre-processing, solving (simulation), and Post-processing. OpenFOAM is based on a C++ library, which the user can create a new solvers and utilities using some pre-requisite knowledge of C++ programing language[8]. It does not have any kind of graphical user interface, so every information or inputs needs to run the program is feeding through specific text files using Linux terminal.

OpenFOAM are organised into different set of directories according to the type of flow, for example combustion, compressible, electromagnetics, incompressible and then different sub-directories according to types of solver, for example icoFoam, pisoFoam, simpleFoam. To use OpenFOAM, the user must create a case inside a directory and sub-directory depends on what kind of flow and solver is going to be used for solving the problem. The user also must assign a name for the case which becomes a name of a directory and it contains the minimum set of files required to run an application. Every OpenFOAM cases must contains at least three directory folders which are ”system”, ”constant” and ”time” directories.

A ”system” directory contains three main files which are used to set parameters for the application. The **controlDict** files contains a general information about start/end time for simulation,

time step of solution, written time interval in which the simulation results are written, parameter for data output, and etc; The **fvSchemes** file is the file which the user define the discretiastion schemes for derivatives, divergences, laplacian, interpolation, etc terms at run-time used in the solution; and the **fvSolution** file contains the equation solvers, tolerances, algorithm and converergence criteria, which are used to run the case.

A ”constant” directory contains a files used to specifying physical properties such as transportation properties and RASproperties (properties for Reynolds averaging) for turbulent modeling needed to solve the case. There is also a sub-directory polyMesh which contains full information about a Mesh.

The ”time” directories containing individual files of data, it can be initial and boundary conditions which is set by the user in the ”0” folder as the simulation start at time t=0 or the results written by OpenFOAM itself, for particular fields, for example velocity and pressure fields. The names of each time directories are based on written interval value of the result of simulation which the user defines it in the controlDict directory.

Figure 2.7 shows the basic directory of an OpenFOAM case structure, which contains the minimum set of files required to run an application.

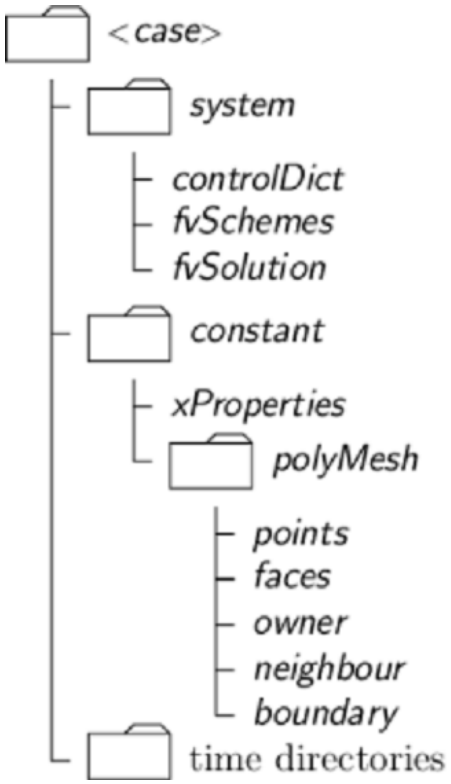



**Figure 2.7:** OpenFOAM case structure[8]

When once put every necessary properties and parameters for each three folder in the case directory, the user must use a serious of commands in terminal in order to execute the case.The command used to execute the solution is same as the solver used, for examples ”icoFoam” for icoFoam solver and ”simpleFoam” for the simpleFoam solver, from the case directory.

Chapter 3

# Modeling

As it was mentioned before, CFD methods use three general procedures for solving the given case. The brief insight into the three general procedures of CFD used for modeling the conductivity sensor and injection system will be outline in this chapter.

## 3.1 Pre-processing

### 3.1.1 Mesh generation

As mentioned in the introduction, this thesis has two main objectives: validate the in-house conductivity sensor and design the injection system. For the first task, simpleFoam simulations were performed to compute for the sensor the flow behaviour in the form of the velocity and pressure field. For the second task, both simpleFoam and scalarTransportFoam simulations performed using a geometry which consists the injection system part together with the conductivity sensor, in order to visualize the distribution of a tracer (passive scalar) concentration throughout the conductivity sensor for a given injector design.

The fluid part of the geometries for both tasks was constructed in Salome. Because the velocity and pressure profile at the entrance is not known, one extends the pipe at the entrance with the dimension of approximately five times the diameter of the pipe at the inlet. This is used to get the laminar flow behaviour of the velocity filed and avoid large velocity gradients in the direction normal to the boundaries near the original inlet of the geometries. For each geometry, first different parts of the geometry define using different drawing operation in geometry module of Salome, such as points, lines, 2d sketch, cylinder, faces, solid, rotation, cut, mirror image, etc, and then all parts put together using the operation called partition. Group of solids and faces of the geometry was generated in the geometry module. Group of solids used later in the mesh module to mesh the geometry with different level of refinement using sub-mesh, while group of faces used to create group of boundary patch for the mesh and used to define the boundary condition later in the OpenFOAM case. Among the several options of algorithms and hypothesis in Salome, NETGEN 3D was used for both geometry to create the pure tetrahedral mesh (unstructured mesh).

Beside the tetrahedral mesh which was generated in Salome, the hexahedral mesh (structured mesh) of the conductivity sensor is created in OpenFOAM mesh generation utility, called snap-

pyHexMesh. To generate the hexahedra mesh of the conductivity sensor in snappyHexMesh, first the geometry of the sensor which was created in Salome is imported as **.stl** file into a snappyHexMesh directory and located in "triSurface" sub-folders inside "constant" sub-directory. The **snappyHexMeshDict** which is located in the "system" sub-directory is used to control the entire process, and this files located in Appendix**D**. This kind of mesh gives lower number of cells and nodes compared to the tetrahedral mesh which was created in Salome for the same geometry.

The mesh of the conductivity sensor geometry has three different groups of patches, called:

- inlet,
- outlet, and
- walls

While the mesh of the geometry which contains the injection part together with the conductivity sensor has four different groups of patches, called:

- inlet
- injection-inlet
- outlet
- walls

After the mesh created in Salome, then it exported in **.UNV** file format and later it imported to OpenFOAM. In Salome the mesh is created in mm and converted to m when imported to OpenFOAM, since all the units in OpenFOAM are SI Units. The mesh of the given geometry located in "polyMesh" sub-folder inside the "constant" folder under subdirectory incompressible/simpleFoam for pressure and velocity simulation, where incompressible indicates the type of flow; and a subdirectory basic/scalarTransportFoam for tracer concentration simulation.

### 3.1.2 Boundary and Initial Conditions

After the mesh of the geometry is created and imported into OpenFOAM, the next and most important step in CFD simulation are the definition of the initial and boundary conditions of the different fields. All this files are located in the "0" directory of the case, as the case start up at time t=0 sec. The initial conditions define the initial state of the case and the boundary conditions define what is going on the boundaries of the domain.

For simpleFoam, the "0" directory contains two files namely one for the pressure field and one for the velocity field. And for scalarTransportFoam, the "0" directory contains three files namely one for the volumetric flow rate ($\phi$), one for the velocity field and the other for the concentration/scalar field. But the initial and boundary conditions was defined only for scalar field while the $\phi$ and velocity files are copy from the simpleFoam case at the simulation latest time.

Pressure field (p): for the incompressible solvers in OpenFOAM, pressure is defined as kinematic pressure, which is the "reduced" pressure divided by density and it has the Unit of $m^2/s^2$.

As the pressure absolute value is not relevant, the initial field is set to be ”uniform” with the value of zero, which means initially the pressure inside the geometry was the same as atmospheric pressure. The boundary conditions are defined for all the boundary patches of the geometries, which are described in the previous sub-section. For inlet, walls and injection-inlet boundary patches, a ”zeroGradient” boundary conditions is used, which means that pressure gradient is zero in the direction perpendicular to the boundary and it extrapolates the values from the domain. For the outlet boundary patch, the fixedValue condition with a value of ”uniform 0” is used, it means initially the pressure at the outlet was the same as the atmospheric pressure and throughout the simulation this value will be changing.

Velocity field (U): the initial field is set to be ”uniform” with the value of zero, but the velocity must be expressed with three dimensions, such as x, y and z directions, so the internal field expressed as the form of ”uniform (0 0 0)”. The boundary conditions (inlet, walls, and injection-inlet) are set to be uniform and fixed (”fixedValue”). In order to visualize the tracer concentration distribution throughout the sensor geometry, both simpleFoam and scalarTransportFoam simulations are performed two times. Hence, the boundary condition for the injection-inlet patches is defined two times with different initial velocity values. The first simpleFoam simulation was performed when the tracer was injected while the second simpleFoam simulation was performed when the tracer injection was finished. The value of ”uniform (0 0 0)” is used at the walls boundary patches, which means the velocity of the fluid at the wall of the geometry in both direction is zero throughout the simulation. For the outlet boundary patch, a ”pressureInletOutletVelocity” condition with the value of ”uniform (0 0 0)” is used, it is the combination of pressureInletvelocity and InletOutlet conditions. This boundary condition is a good choice when the direction of the fluid flow is unknown and there is a back flow at the outlet. Since the value is set as uniform (0 0 0), the pressureInletOutletVelocity condition works like the zeroGradient condition, that means both conditions gives the same result for our cases.

Concentration/scalar transport field (T): the initial and boundary condition for this field is defined two times at the injection-inlet boundary patch, as the simulation was performed two times in scalarTransortFoam solver. For the first simulation, the simulation when the tracer was injected, the initial field set to be uniform with the value of uniform zero, which means no tracer concentration initially inside the entire geometry. And ”fixedValue” condition with the value of uniform 1 is used as a boundary condition at injection-inlet boundary patch. And for the second simulation, the simulation when the tracer injection was finished, the initial field set to be ”nonuniform” with value of T which is found from the first scalarTransportFoam simulation at the latest time (the latest time of the simulation is the same with the tracer injection time, which is 1 sec in this case). And the boundary condition change from ”fixedValue” with the value of uniform 1 to ”zeroGradient” condition at the injection-inlet patch.
For both scalarTransportFoam simulations the same boundary condition is defined for the inlet, outlet and walls boundary patches. For inlet, ”fixedValue” conditions with the value of uniform zero is used, meaning throughout the simulation there is no scalar concentration at the inlet of the geometry. For the outlet and walls boundary patches, ”zeroGradient” conditions is used, meaning the concentration gradient is zero in the direction perpendicular to the boundary and it extrapolates the values from the domain throughout the simulation.

In order to perform the scalarTransportFoam simulation, the two files beside the T file, such as $\phi$ and velocity files are copied from the simpleFoam simulation in to the ”0” subdirectory of

the scalarTransportFoam simulation case.
For the first simulation of scalarTransportFoam, the $\phi$ and velocity files are copied from the first simpleFoam simulation at the latest time into the "0" subdirectory of the scalarTransportFoam case, as the case start up at time t=0 sec.
And for the second simulation of scalarTransportFoam, the $\phi$ and velocity files are copied from the second simpleFoam at the latest time into the "1" subdirectory of the scalarTransportFoam case, as the case start up at time t=1 sec.

Summaries of the boundary conditions used for simpleFoam and scalarTransportFoam simulations in the case of both modeling of the conductivity sensor and injection system design are presents in Table 3.1 to Table 3.3.

| Variable | inlet | outlet | walls |
|---|---|---|---|
| Velocity(U) | fixedValue | zeroGradient | fixedValue |
| Pressure(p) | zeroGradient | fixedValue | zeroGradient |

**Table 3.1:** Boundary Conditions for simpleFoam simulation in the case of the conductivity sensor modeling

| Variable | inlet | injection-inlet | outlet | walls |
|---|---|---|---|---|
| Velocity(U) | fixedValue | fixedValue | zeroGradient | fixedValue |
| Pressure(p) | zeroGradient | zeroGradient | fixedValue | zeroGradient |

**Table 3.2:** Boundary Conditions for simpleFoam simulation in the case of injection system design

| Variable | inlet | injection-inlet | outlet | walls |
|---|---|---|---|---|
| concentration(T) | fixedValue | fixedValue/zeroGradient | zeroGradient | zeroGradient |

**Table 3.3:** Boundary Conditions for scalarTransportFoam simulation in the case of injection system design

The boundary conditions files for pressure and velocity fields are located in Appendix**A** and for Concentration field is found in Appendix**B**

### 3.1.3 Physical Properties

The physical properties for the case stored in the "control" directory of the OpenFOAM directory tree and each physical properties file has a suffix name of Properties. For a simpleFoam case, two dictionaries must be defined: transportProperties dictionary is the dictionary which the kinematic viscosity of the case is stored, water at a temperature of $18^oc$ with the value of

$1.06 * 10^6$[9] was used for this thesis; and RASProperties dictionary which contains information about RASModel and flow properties. SimpleFoam is used for steady-state turbulence flow but it is possible to use it also for steady-state laminar flow simply by putting laminar as RASModel and switched off turbulence in the RASProperties dictionary[8]. For the scalarTransportFoam case, the only property must that specified is the diffusion coefficient of the tracer which is stored in the transportProperties dictionary.

All the physical Properties files are located in the Appendix**A** for simpleFoam case and in the Appendix**B** for the scalarTransportFoam case.

### 3.1.4 Control, Discretisation and Linear Solver Settings

Files which contain information about control, discretisation and linear-solver settings are stored in the controlDict, fvSchemes and fvSolution dictionary, respectively, and both these three dictionaries are located in the "system" directory.

In controlDict dictionary: start/stop time for the simulation, time step for solution, and reading and writing of the solution data were sets.

In fvSchemes dictionary: the choices of finite volume discretisation schemes for terms in equations were specified.

In fvSolution dictionary: the choice of the linear-solver, solution tolerances and algorithm for the fields which are calculating in the simulation were sets. OpenFOAM have several options of linear-solver but in this thesis, GAMG: generalized geometric-algebraic multi-grid solver with GaussSeidel smother, tolerance of $1 * 10^7$ , and relative tolerance of 0.1 were used for pressure equation; PBiCG: preconditioned bi-conjugate gradient solver with diagonal incomplete-LU(DILU) preconditioner, tolerance of $1 * 10^8$ and relative tolerance of 0.01 were used for velocity equation; and PBiCG solver with DILU preconditioner, tolerance of $1 * 10^5$ and relative tolerance of 0 were used for tracer transport(T) equation. As the relative tolerance value set to 0 for the tracer transport, the solution forced to converge at each time step to the solver tolerance.

Both the solver tolerance, tolerance and relative tolerance, relTol determines the convergence of the solver. The solver tolerance should represent the minimum value of the residual and the relative tolerance specifies the ratio of current to initial residuals value which solver uses as convergence criteria. The solver converges if either the residuals is less than the solver tolerance or the value of the ratio of current residual to final residual falls below the relative tolerance[8].

All linear-solver settings for simpleFoam and scalarTransportFoam simulations are summarise in Table 3.4.

| Variable | Solver | Preconditioner | Smoother | Tolerance | relTol |
|---|---|---|---|---|---|
| Velocity(U) | PBiCG | DILU | - | $1 * 10^8$ | 0.01 |
| Pressure(p) | GAMG | - | GaussSeidel | $1 * 10^7$ | 0.1 |
| concentration(T) | PBiCG | DILU | - | $1 * 10^5$ | 0 |

**Table 3.4:** Linear-solver settings for both simpleFoam and scalarTransportFoam simulations

All this text files are located in Appendix**A** for simpleFoam case and in Appendix**B** for the scalarTransportFoam case.

## 3.2 Running the Simulation

After setting every information and data in the three directory, such as ”0”, ”constant”, and ”systems” directory, then the next step will be running the simulation. The simple way of running the OpenFOAM application is to run the case in the foreground. For example to run simpleFoam, first entering the case directory and typing simpleFoam at the terminal/command prompt.

For this thesis every OpenFOAM cases running in parallel on multiprocessor using NTNU's supercomputer: Vilje. In order to run the simulation in the supercomputer, all files of the case must first be transferred from the host computer to the supercomputer. And in addition to all OpenFOAM case files two additional files are needed: decomposeParDict which is located in the system directory and the job script file which contains all information used to run the case in the supercomputer. The more detailed description about how to run the OpenFOAM case in parallel using supper computer is outlined in section 3.4.

Summary: For the case of modeling the conductivity sensor, simulations were performed with the simpleFoam solver using hexahedral and tetrahedral meshes of the sensor. And for the case of the injection system design, several simulations were performed for each of the injector design with the simpleFoam and the scalarTransportFoam solvers using the following procedures:

For simpleFoam:

- First, the pressure and velocity fields were calculated at a time when the tracer was injected.
- Second, the pressure and velocity fields were calculated at a time when the tracer injection was stopped/finished.

For scalarTransportFoam:

- First, the initial distribution of the tracer concentration inside the sensor was analysed when the tracer was injected using the value of $\phi$ and velocity fields obtained from the first simpleFoam simulation.

- Second, the final distribution of the tracer concentration was analysed when the tracer injection was finished using the value of $\phi$ and velocity fields obtained from the second simpleFoam simulation and the value of concentration field obtained from the first scalarTransportFoam simulation as initial conditions for the T field.

## 3.3 Post-processing

OpenFOAM has a post-processing tool to visualize the result of the simulation, called paraFoam. It is also used to view the mesh before the case run to check any error in the mesh. To start the paraFoam post-processing, typing paraFoam in the terminal window from within the case directory or from another directory. This opens the paraView window showing the solution of the given case graphically. There is a panel in the left side of the paraview window which is used to control the given case. This panel contains the following: "Pipeline browser" which lists all opened modules in paraView; "Properties panel" which contains the input selections, such as times, regions and fields for the given case; "Display panel" which used to control the selected module visual representation; and "Information panel" which gives information about mesh geometry and size[8].

In this thesis, the final result of pressure and velocity fields are shown graphically throughout the entire geometry for the simpleFoam simulation and the distribution of the tracer concentration throughout the geometry are shown graphically for scalarTransportFoam simulation.

## 3.4 Running OpenFoam on a Supercomputer

Running OpenFOAM using a single processor sometimes takes long time to generate a solution, especially for complex geometry, very fine and unstructured mesh. So to generate a solution in the short period of time it is better to run OpenFOAM in parallel on distributed processor using a supercomputer. To run OpenFOAM case in parallel, the geometry and associated fields must be broken into pieces and distributed to a separate processor using a method, called domain decomposition. The standard message passing interface (MPI) is used to running OpenFOAM in parallel.

The OpenFOAM utility, called "decomposePar" is used to decompose the mesh and fields. There is a file in the system folder which contains the parameters used for the decomposition of the mesh and fields, and it is called "decomposeParDict". OpenFOAM has three methods of decomposition which specified in the following method keyword: "Simple", "hierchical" and "scotch".

The supercomputer which is found at the NTNU is used to run all the simulation cases for this thesis. It is called "Vilje" and it has 1404 nodes, 2 eight-core processor per node and 16 cores per node. There is a lot of software installed on it including OpenFOAM[10].

All simulations used 16 mpi processors, so the mesh and associated fields must be broken in to pieces and distributed into these 16 processors. "Simple" method keyword is chosen to decompose the mesh and fields for both simpleFoam and scalarTransportFoam simulations. The same decomposeParDict file is used for both solvers, and it is located in Appendix**C.1**.

To run OpenFoam on Vilje, a simple job script is needed. This job script contains name of the job, Vilje account name, the wall clock time limit of the job, number of cpus, number of mpi processor and some other information needed to run the case. First it must be made executable by typing "**chmod +x** $< jobscriptname >$ in terminal, since it written in the host computer, then it has to be submit by typing "**qsub** $< jobscriptname >$". The job scripts used for both simpleFoam and scalarTransportFoam in this thesis are located in Appendix**C.2**.

Each file used for running a case in OpenFOAM including the job script can be written either in the supercomputer itself or in the host computer and transfer into the supercomputer home directory using a command "**scp**" in the terminal. This command used for transferring files and folder from host computer to the supercomputer and vice versa.

After the simulation is done, the solution must be gather together and put into the specific time folders in the "0" directory of the OpenFOAM case, since the solution is distributed into each mpi processor as the case running using 16 mpi processor. "ReconstructPar" command which is specified in the job script is responsible for this task.

## 3.5 Experimental Work

A simple experiment was performed to measure the pressure drop of the real conductivity sensor using various pump speed. The experiment setup consists of U-tube, conductivity sensor, pump, plastic pipe and T-junction. The setup is shown in figure 3.1.

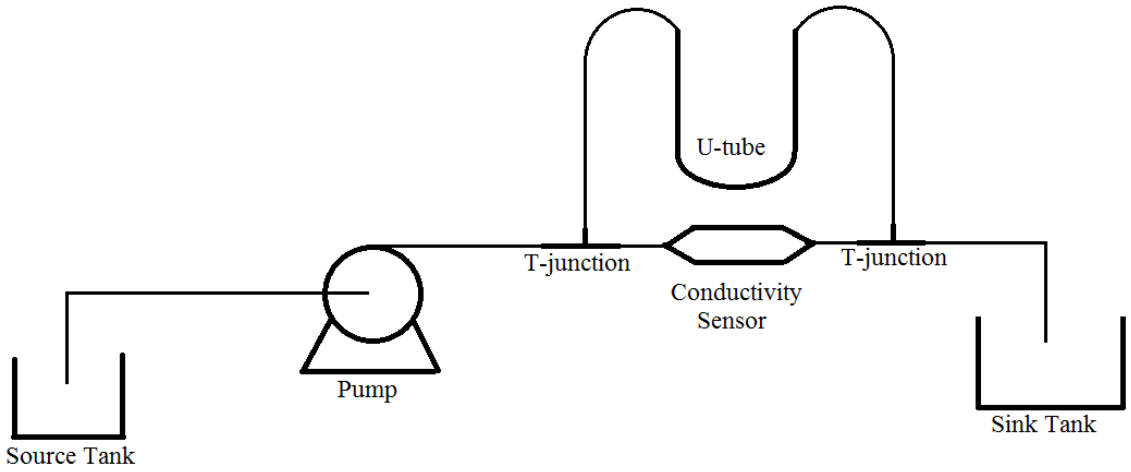



**Figure 3.1:** Experiment setup

Distilled water is pumped from source tank to sink tank through the conductivity sensor with various pump speeds and the pressure drop over the conductivity sensor is read from the U-tube.
The objective of performing this experiment is to check the viability of the simulation results. Less or equal to 5% error between the simulation and the experiment result is acceptable. All the experiment results for the various pump speeds are summarised in Table E.1 which is located in Appendix**E**.

Chapter 4

# Result and Discussion

As mentioned in chapter one, this thesis has two main objectives. To fulfill the two objectives several simulations were performed using simpleFoam and scalarTransportFoam solvers.

To validate the design of the in-house conductivity sensor and to get a best suited mesh type for the sensor geometry, several simpleFoam simulations were performed utilising tetrahedral and hexahedral meshes of the liquid body in the sensor.

To design the injection system, two types of injector geometries were studied. To visualize the trace cloud formation and distribution throughout the geometry for a given injection design, simulations were performed with the simpleFoam and scalarTransportFoam solvers.

Two different names were used for the same conductivity sensor in this thesis namely old and modified sensor. Old sensor is denoted for the conductivity sensor geometry which was generated using a measurement from the technical drawing and direct measurement from the physical sensor, while modified sensor is denoted for the conductivity sensor geometry which some part of the sensor geometry changed with in expected error bounds.

Figure 4.1 and 4.2 summaries the main results from the simulations for the modeling of the conductivity sensor and injection system design cases. The more detailed discussions on all obtained results are outline in the next sub-sections.

| Sensor geometry | Simulation Performed | Results |
|---|---|---|
| Old sensor (Hexahedral Mesh) | simpleFoam | The result of simulation is not in good agreement with the experiment, 55% error was found. The mesh quality is also not good enough. |
| Old sensor (Tetrahedral Mesh) | simpleFoam | The result of simulation is not in a good agreement with the experiment, 38.7% error was obtained. But the mesh has a good quality and granularity. |
| modified sensor (Hexahedral Mesh) | simpleFoam | Yields a better result of simulation than the above two cases, but still not in a good agreement with the experiment, 23.7% error was found. The mesh has a granularity but does not have a best quality. |
| Modified sensor (Tetrahedral Mesh) | simpleFoam | The result of simulation is in a good agreement with the experiment showing only 4.8% error. A pressure drop of 63.49cmH2O was obtained. The mesh has a good quality but it needs to be more fine (dense). |

**Table 4.1:** Summaries of the results of simulations for a conductivity sensor modelling case

| Injector geometry | Simulation Performed | Results |
|---|---|---|
| 90 degree bent cylinder with a needle shape at one end | simpleFoam(x2) and scalarTransportFoam(x2) | The tracer cloud was formed in unexpected direction and most of the tracer goes through in the top of the sensor. This is not a desired occurrence. |
| 90 degree bent cylinder with circular shape at both end | simpleFoam(x2) and scalarTransportFoam(x2) | This design gives a better cloud formation and tracer distribution throughout the sensor geometry. |

**Table 4.2:** Summaries of the results of simulations for injection system design case

## 4.1 Modeling of conductivity sensor

The aim of this section is to perform the simpleFoam simulation of the conductivity sensor to visualize the flow behavior of the fluid inside the sensor. This is used to validate the design of the in-house conductivity sensor. Simulations were performed utilising tetrahedral and hexahedral meshes. The objective of performing the simulation with different types of meshes of the sensor is to find how the results of simulations are affected by the choice. For this purpose only the fluid part of the conductivity sensor was created in Salome, as shown in figure 4.1 and figure 4.2.

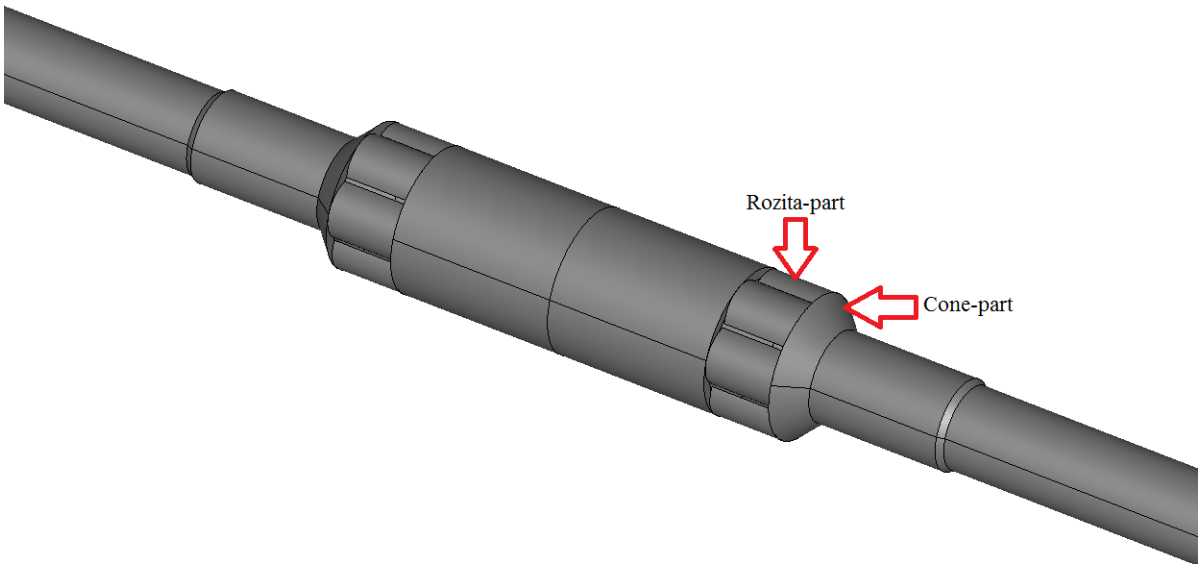
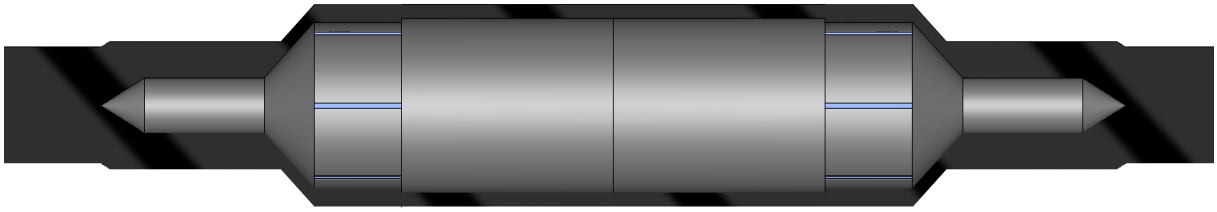
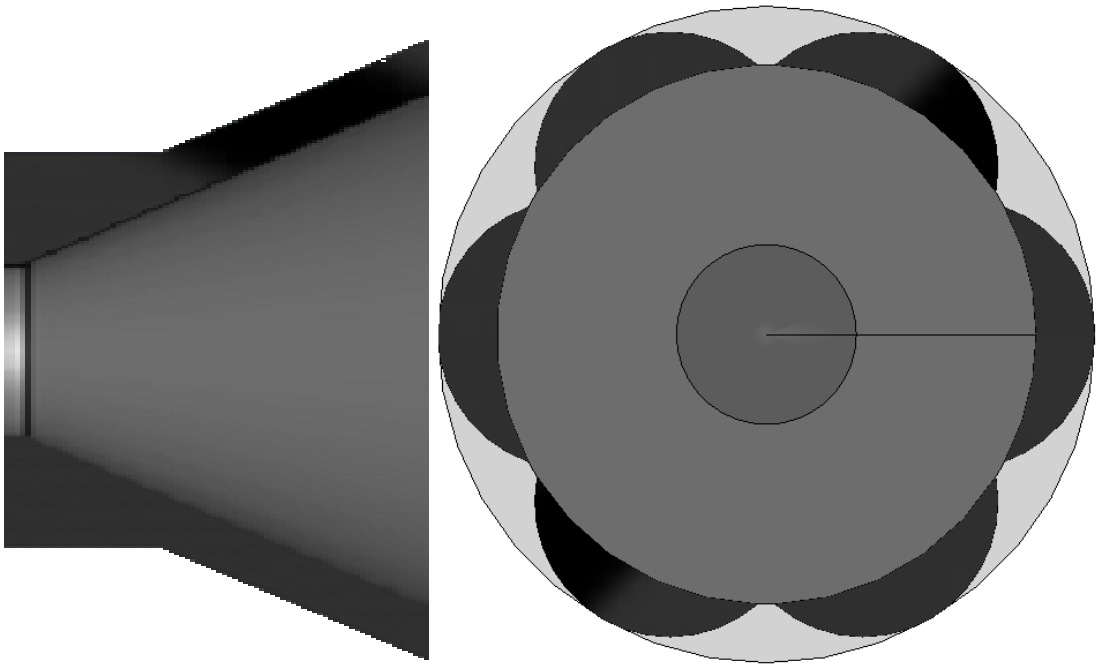



**Figure 4.1:** The fluid part of the conductivity sensor



**(a)** section view of the fluid part



**(b)** cone-part **(c)** rozita-part

**Figure 4.2:** Section view of the fluid part of the conductivity sensor

As shown in figure 4.2, all the black color parts are the fluid part and all the gray color parts are the solid parts and the solid part consider like a wall in the simulation. But when created the geometry in Salome, the fluid part created as a solid and a solid part was created as empty. This is because of the mesh only generated on the fluid.

### 4.1.1 Hexahedral Mesh

The hexahedral mesh of the sensor fluid part was created using the OpenFOAM mesh generator utility, called snappyHexMesh. The fluid part which was created in Salome used to generate this mesh. The geometry was imported to snappyHexMesh case directory as a **.stl** text format. This mesh has 1,933,973 of hexahedra cells and 26,039 of polyhedra cells and it shown in figure 4.3.

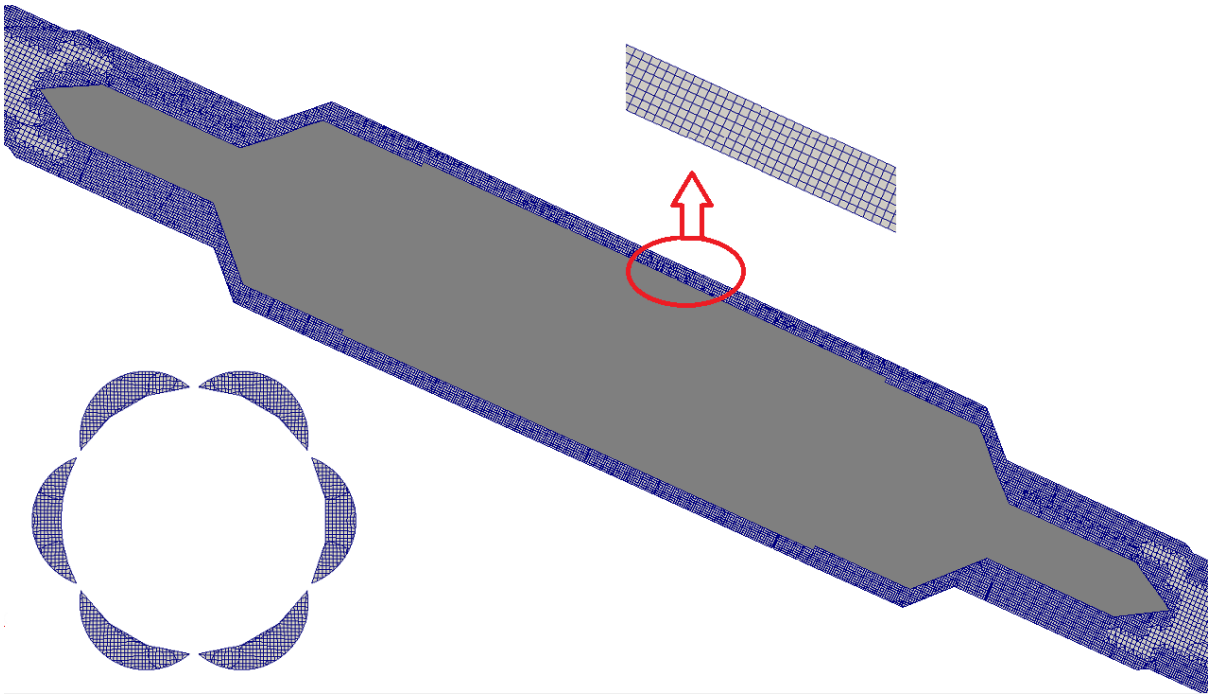


**Figure 4.3:** Hexahedral mesh of the sensor fluid part

As seen from figure 4.3, the quality of the mesh was not good enough, especially around the pin part and at the tip of rozita part. This is due to the Sharpe edge both at the tip of the pin and rozita. It is difficult to get six faces around the pin part and at the tip of the rozita, so instead of hexahedra cells polyhedra cells was created, since six faces are needed to generate hexahedral element shape. Generally, the mesh has six non-orthogonal faces and two highly skewed faces which may affect the quality of the result of simulation later in the simulation. The effect of the non-orthogonal faces can be removed by putting non-orthogonal corrector in the ”fvSolution” subdirectory. As seen from figure 4.3 which is indicated by the red circle, the mesh has a uniform six layers in the narrowest part of the sensor and it is a good number of layers for laminar flow.

After the mesh was generated, the simulation was done in OpenFOAM with the simpleFoam solver to visualize the flow behaviour in the form of the velocity and pressure fields. Simulations were performed using various pump flow rates. The result of simulation for a maximum

pump flow rate, which gives 0.488m/s of sensor inlet velocity is shown in figure 4.4 and 4.5 and the rest of the results are summarise in Table **E.1**.

As seen from figure 4.4, the fluid passing through the sensor with similar velocity which is equal to the initial sensor inlet velocity, except in the rozita and cone part of the sensor. There was a high velocity in these two parts. This result indicates that the flow cross sectional area in the rozita and cone part is not the same with the flow cross sectional area of the rest of the sensor parts.

The pressure decreases along the fluid flow direction as shown in figure 4.5, this is mainly due to the cone and rozita parts of the sensor which creates a higher reduction in pressure. The pressure drop of 29.98cmH2O was found for sensor inlet velocity of 0.488m/s and it is very lower than the pressure drop of 66.7cmH20 which was found from experiment for the same sensor inlet velocity. There was approximately 55% of error between the simulation and experiment.

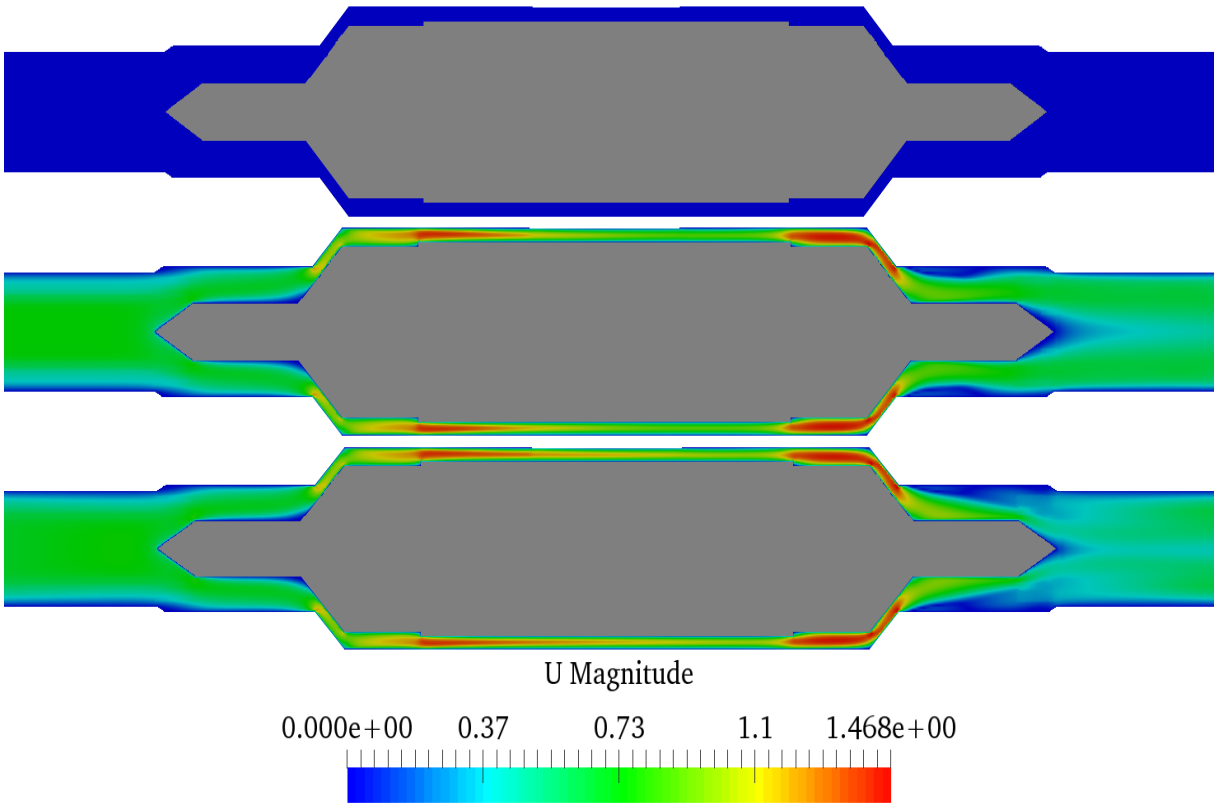



**Figure 4.4:** Velocity Field: top at t=0, middle at t=150 and bottom at t=300 sec

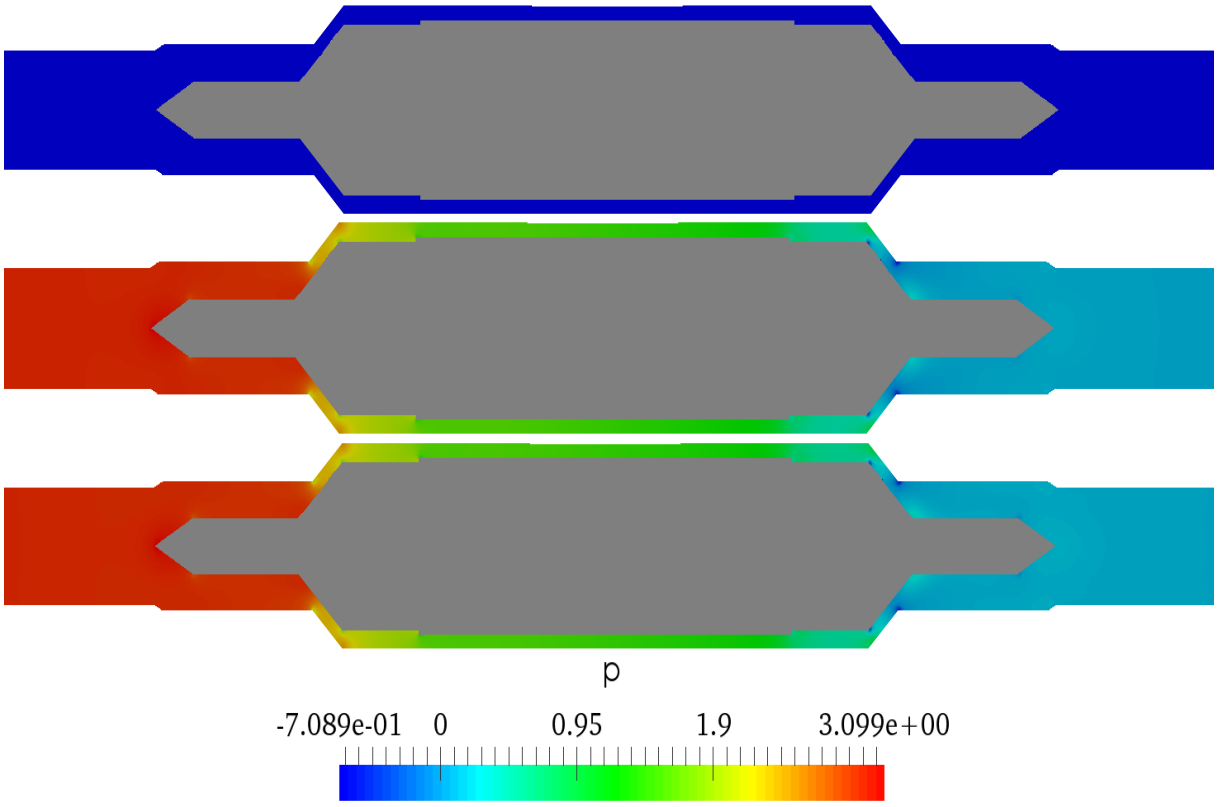



**Figure 4.5:** Pressure Field: top at t=0, middle at t=150 and bottom at t=300 sec

### 4.1.2 Tetrahedral Mesh

The tetrahedral mesh of the sensor fluid part was created in Salome using the same geometry in the above case.The sensor geometry was meshed with different refinement level using a sub-mesh. The mesh is more fine in the rozita and cone part of the sensor geometry, since this two parts are the most critical part for the pressure drop as seen from the simulation result of the above hexahedral mesh case. This mesh is pure tetrahedra mesh with 6,608,075 tetrahedra cells and it shown in figure 4.6.

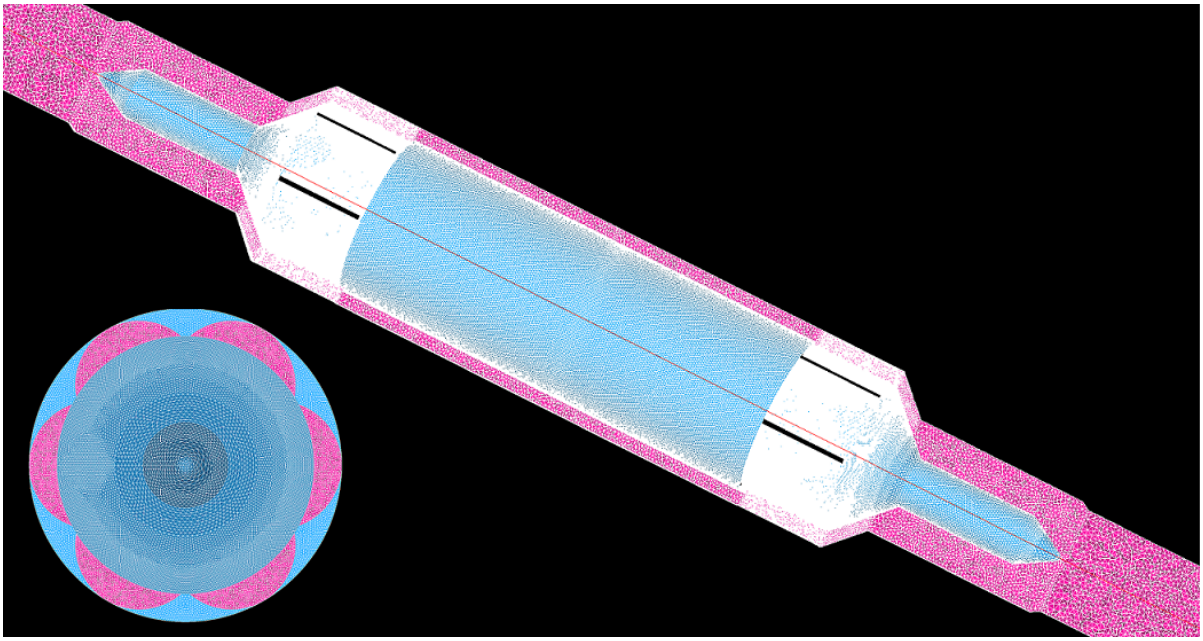


**Figure 4.6:** Tetrahedral mesh of the sensor

The mesh has 334 non-orthogonal faces and no skewed faces. Similarly, the effect of these non-orthogonal faces were removed by sitting non-orthogonal corrector in ”fvSolution” subdirectory. The mesh has more than six layer in the narrowest part of the sensor. But the layer is not uniform like the hexahedral mesh, since tetrahedra element shape gives unstructured mesh type. This mesh has a better quality than the hexahedral mesh in the above case, as the mesh has a good uniformity and granularity throughout the entire sensor geometry.

Similarly, simulations were performed in OpenFOAM using the simpleFoam solver with various pump flow rates. The results of simulations are shown in figure 4.7 and figure 4.8 for sensor inlet fluid velocity of 0.488m/s and the rest of the results of the remaining sensor inlet velocities are summarise in Table **E.1**.

As seen from figure 4.7 and 4.8, the behaviour of the result of simulation for the tetrahedral mesh is almost the same with the result from the hexahedral mesh. As shown clearly in figure 4.7, there is a zero fluid velocity around the tip of the pin in the outlet side. The pressure drop for the tetrahedral mesh which is 40.9cmH2O is much higher than which is found from the hexahedral mesh for the same sensor inlet velocity of 0.488m/s. But still the result of simulation for the tetrahedral mesh was not the same or nearly the same with the experimental result, 38.7% of error was found.

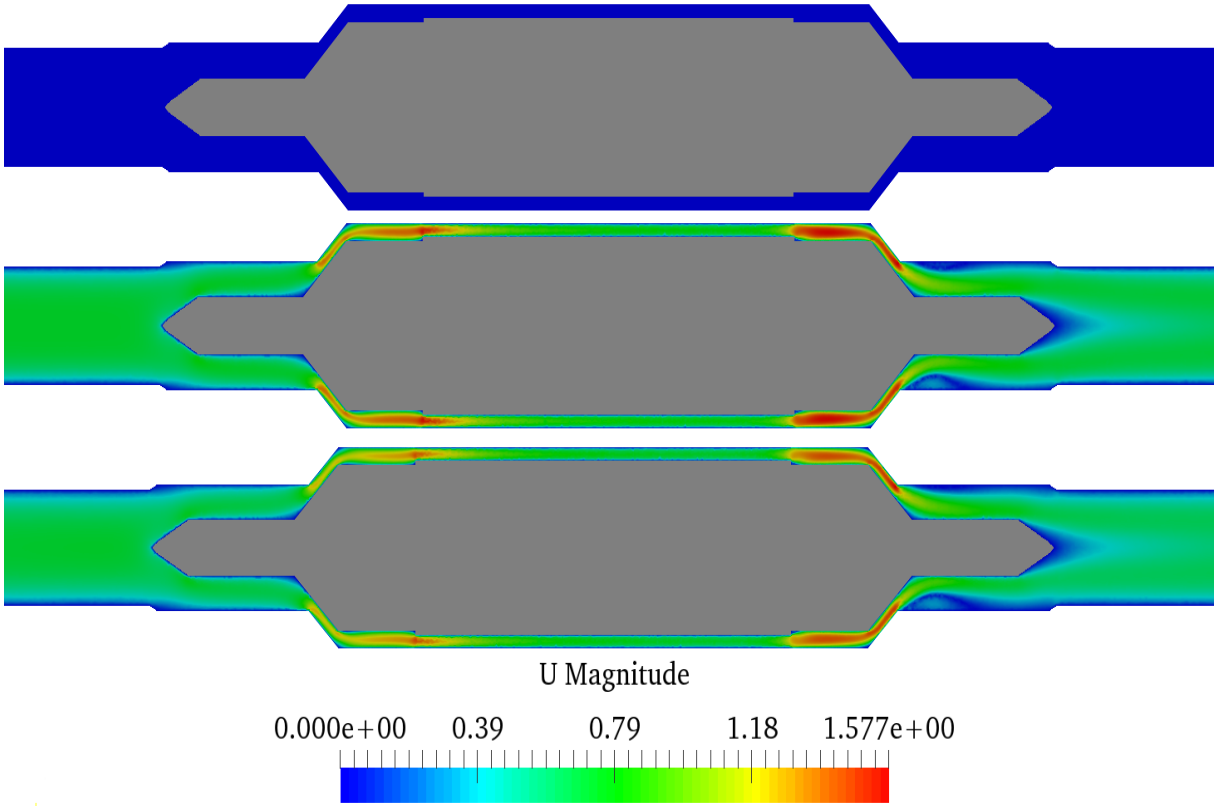



**Figure 4.7:** Velocity Field: top at t=0 sec, middle at t=150 sec and bottom at t=300 sec

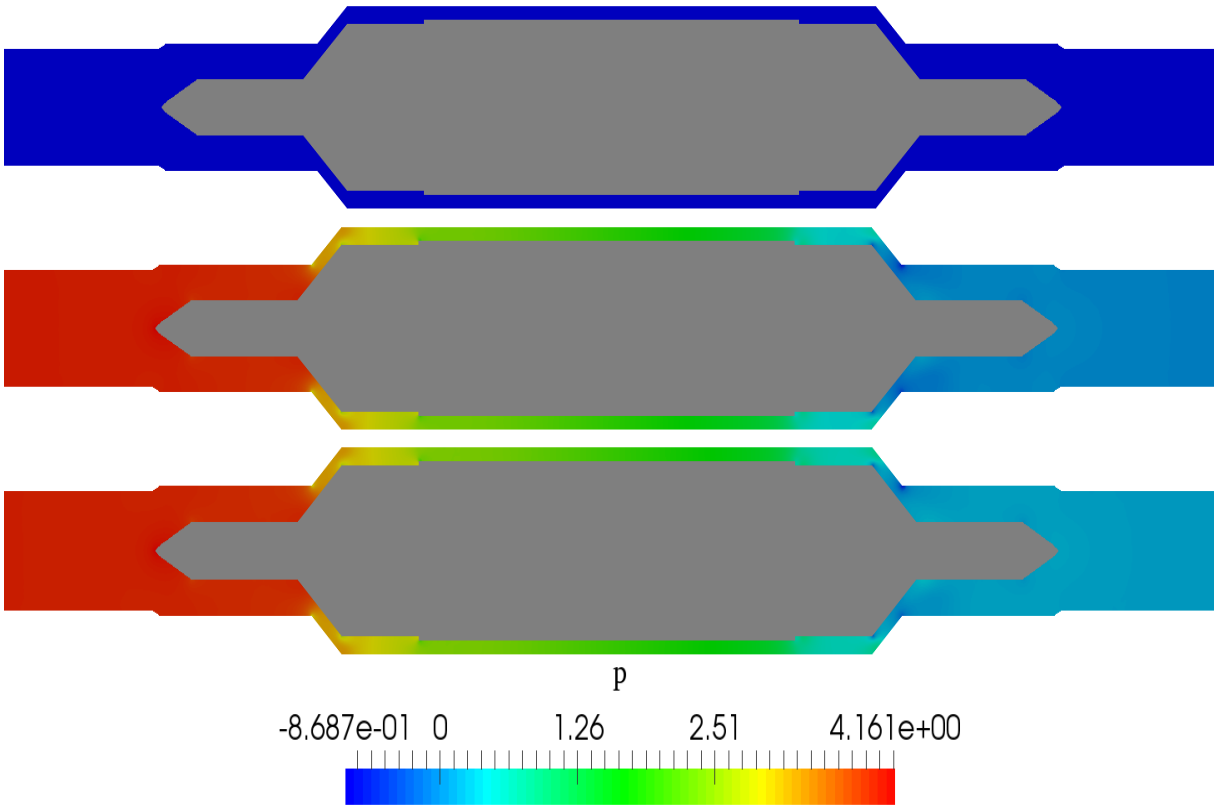



**Figure 4.8:** Pressure Field: top at t=0 sec, middle at t=150 sec and bottom at t=300 sec

Generally, as seen from the result of both types of meshes, the results of simulations were not in a good agreement with the experiment result. As mentioned in section 3.5, a good agreement between the result of simulation and experiment with only less or equal to 5% error is very important to be sure of the accuracy of the result of simulation. So, before continue with the next task of the thesis some investigation on the fluid property, the choice of the boundary conditions, solvers and the dimensions of the sensor fluid part was performed.

The Reynold numbers were calculated for various pump flows rates at sensor inlet diameter and all the values are less than 2000 as seen from Table **E.1**. This value tells that the flow has a laminar behavior. All simulations assume a laminar flow, so no need have changed to turbulence flow. Friction is already considered in the calculations for the laminar flow simulations and no problem in the boundary conditions and solvers. So this shows that the problem is not in the simulations, but is more likely in the dimensions of the sensor fluid part geometry, since the drawing of the fluid part was created using the dimensions which are measured directly from the physical sensor and the sensor has some parts which are difficult to measure exactly. Sensitivity analyse was performed on the sensor geometry to get the exact dimensions of sensor fluid part. The result obtained from the sensitivity analyse was discussed in the next sub-section.

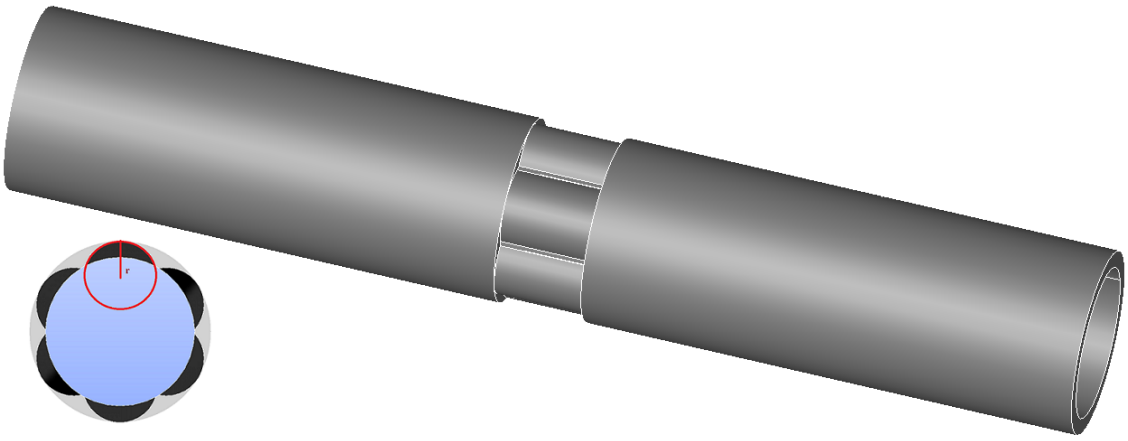
### 4.1.3 Sensitivity Analyse

As seen from the result of simulation for both hexahedral and tetrahedral mesh in sub-section 4.1.1 and 4.1.2, the cone and rozita parts are the most critical part for the increment in pressure drop and in addition to that these two parts are also the most difficult parts to measure the dimensions exactly from the real sensor using a measurement tool. Therefore, the sensitivity analyse focused on this two parts.

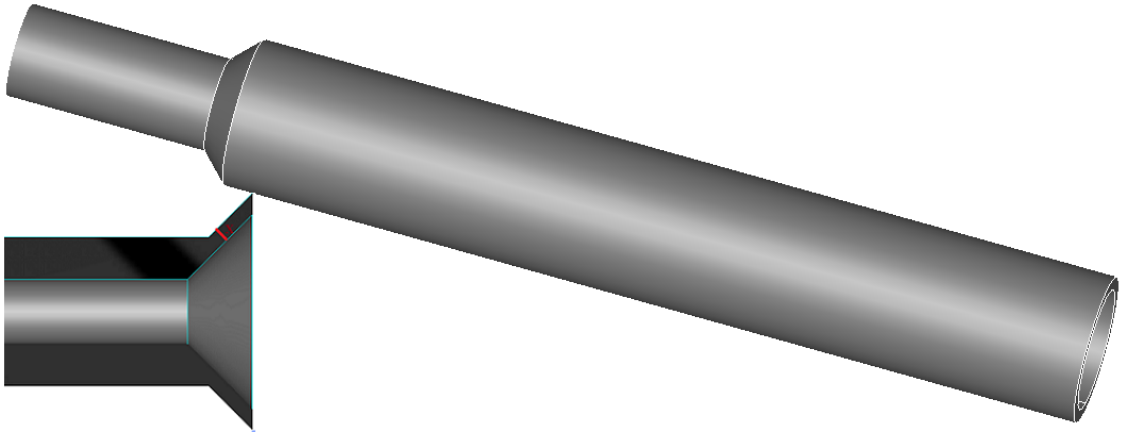
First, two concentric cylinder with rozita in the middle was created, as shown in figure 4.9(a) and mesh it with tetrahedral element shape in Salome. Then the simulation was performed with three different size of rozita, for **r** values of 1.5, 1.45 and 1.4 mm using simpleFoam solver. Where, **r** is the radius of the single rozita cylinder which is shown with red circle in the bottom left side of figure 4.9(a). The pressure drop of 5.42, 6.09 and 7.09 $m^2/s^2$ was found, respectively. The result shows that the reduction of the rozita radius with 0.05 and 0.1mm increase the pressure drop by 12.36% and 30.81%, respectively.

Second, two concentric cylinder with cone at the inlet side was created, as shown in figure 4.9(b) and mesh it with tetrahedral element shape in Salome. The simulation was performed with three different values of **x**: 0.46, 0.44 and 0.42 mm. Where, **x** is the perpendicular distance between the outer and inner cone which is shown with red line in the bottom left side of figure 4.9(b). The pressure drop of 5.36, 5.47 and 5.59 $m^2/s^2$ was obtained, respectively. The simulation result shows that the reduction of **x** with 0.02 and 0.04mm increase the pressure drop by approximately 2 and 4.3%, respectively.

The sensor fluid part geometry was modified based on the sensitivity analyse results. The radius of the rozita decreased by 0.01mm while the perpendicular distance between the outer and inner cone decreased by 0.06mm. The modified sensor fluid part was created in Salome and it shown in figure 4.10.



**(a)** Two concentric cylinder with rozita in the middle



**(b)** Two concentric cylinder with cone at the inlet side

**Figure 4.9:** Simple geometries using for sensitivity analyse

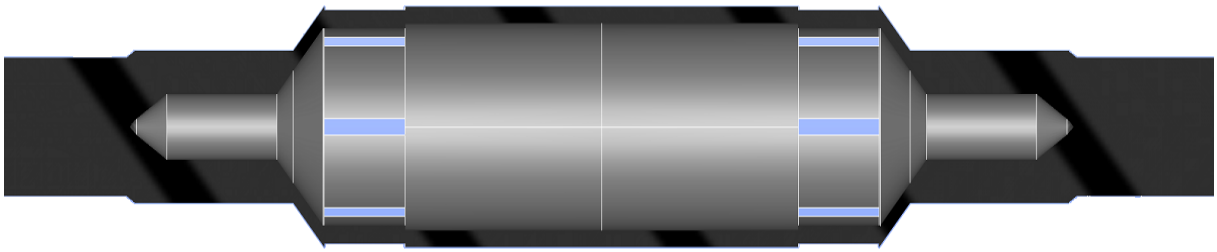


**Figure 4.10:** Section view of the modified sensor fluid part

Similarly, simulations with simpleFoam solver were performed for the modified sensor using both hexahedral and tetrahedral mesh and results were discussed in the next two sub-sections.

### 4.1.4 Hexahedral mesh of the modified sensor

This mesh was generated in snappyHexMesh using the same parameters and procedures as the first case. The mesh has 4,962,578 of hexahedra cells and 581,578 of polyhedra cells. This mesh is shown in figure 4.11.

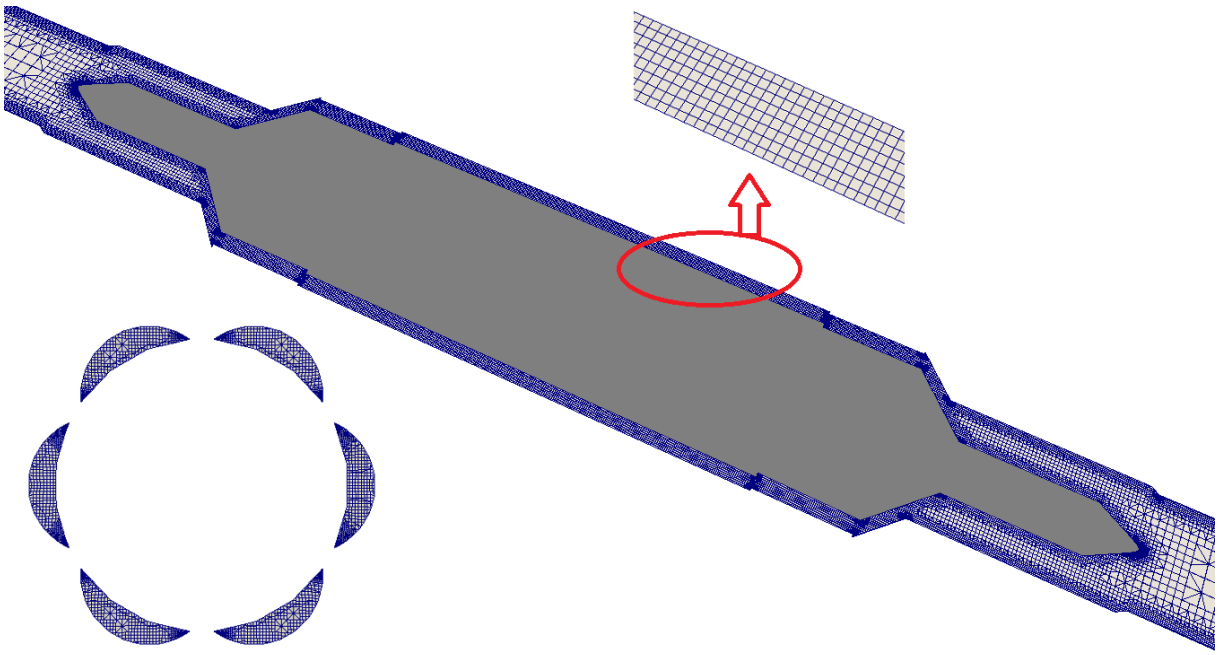


**Figure 4.11:** Hexahedral mesh of the modified sensor

This mesh is approximately three times finer than the hexahedral mesh of the old sensor which is illustrated in sub-section 4.1.1. Figure 4.11 shows that the quality of the mesh around the rozita part was not good enough. This is due to the sharp edge around it, which is created because of the reduction of the rozita radius. The mesh has three non-orthogonal faces, seventy skewed faces and eight layers. This numbers indicated that the mesh has a good quality in terms of the number of non-orthogonal faces and layers, while in terms of number of skewed faces the mesh quality is not good enough.

After the mesh was created, simulations were performed using simpleFoam solver for various pump flow rates. The result of simulation for the sensor inlet velocity of 0.488m/s is illustrated in figures 4.12 and 4.13. Results for the others pump flow rates are illustrated quantitatively in Table **E.1**.

Figure 4.12 clearly shows that the reduction of the rozita radius increase the fluid velocity inside the rozita part. This is due to the effect that, for the size reduction of any part of the sensor, makes the reduction of the flow cross sectional area. In addition to that the figure shows unexpected result of simulation behaviour around the pin part at the outlet side, which indicated zero fluid velocity in the flow direction. This is might be due to the poor mesh quality around the pin part and the presence of high number of skewed faces in the mesh.

As seen from figure 4.13, the pressure decreases along the flow direction and the reduction of the size of rozita and cone increase the pressure drop as expected. The pressure drop of 50.9cmH2O was found but this value is still very lower than the real pressure drop of the sensor which is found from the experiment. There was around 23.7% of error between the simulation and experiment.

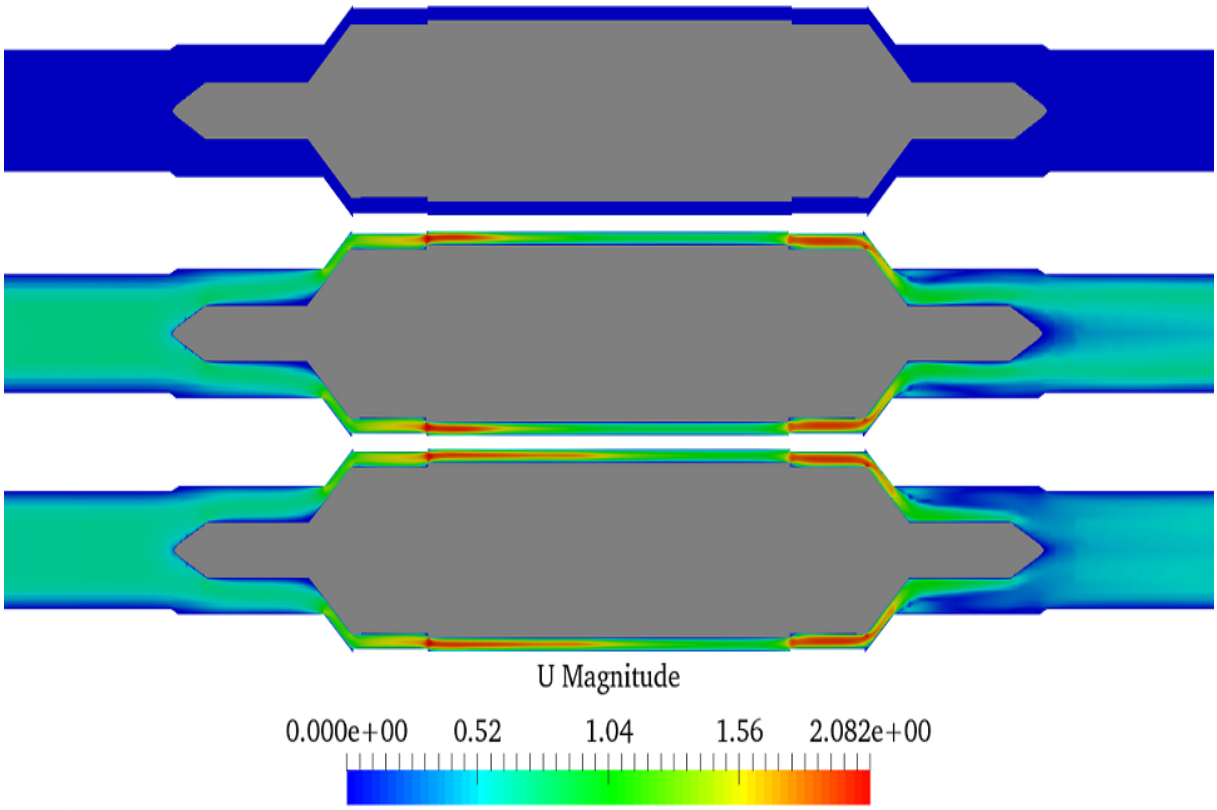



**Figure 4.12:** Velocity Field: top at t=0, middle at t=150 and bottom at t=300 sec

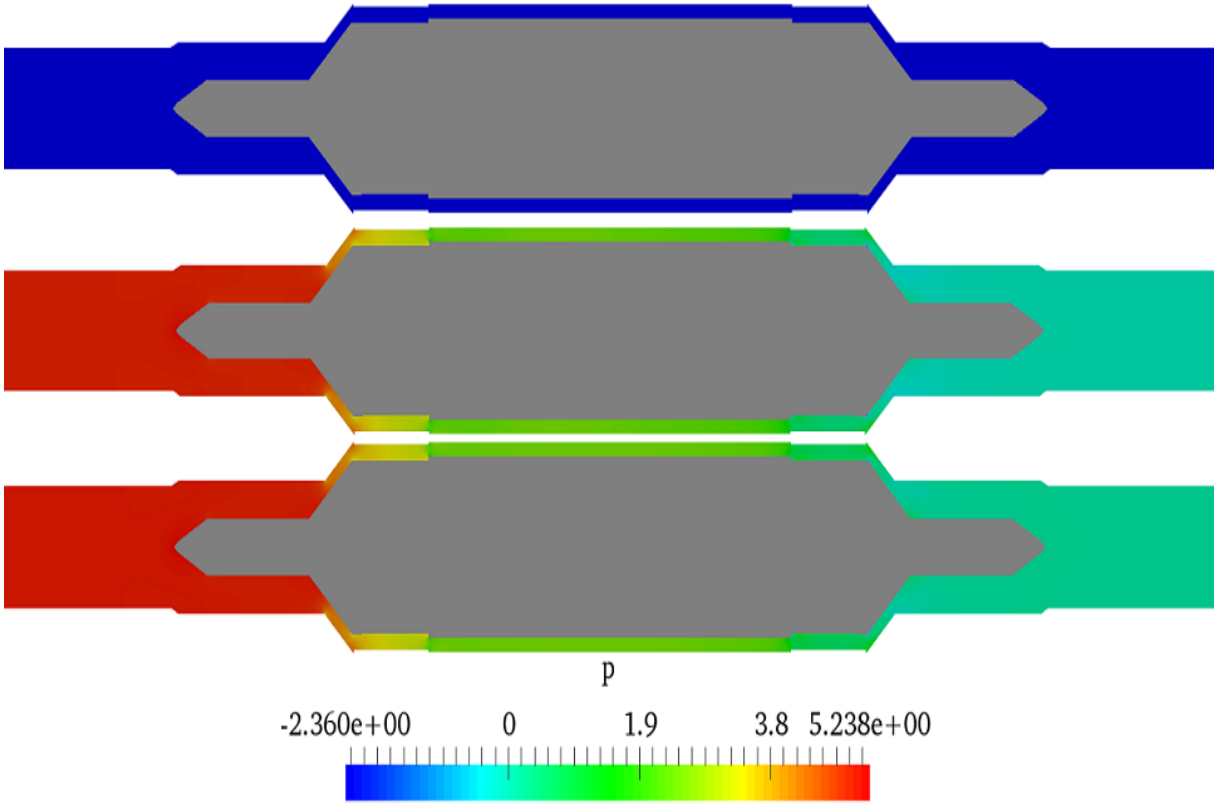



**Figure 4.13:** Pressure Field: top at t=0, middle at t=150 and bottom at t=300 sec

### 4.1.5 Tetrahedral Mesh of the modified sensor

The tetrahedral mesh of the modified sensor was created in Salome. Similarly, the sensor was meshed with different refinement level, which means that the mesh is finer in the critical part of the sensor: cone and rozita part. The mesh is pure tetrahedral mesh with 7,499,090 of cells. This mesh is illustrated in figure 4.14.

As seen from figure 4.14, the mesh has a good quality and granularity, especially in the critical part of the sensor geometry. The mesh has 394 non-orthogonal faces while no skewed faces and it needs to be finer in the narrowest part of the sensor geometry in order to increase the number of layers, as it has only five layers.

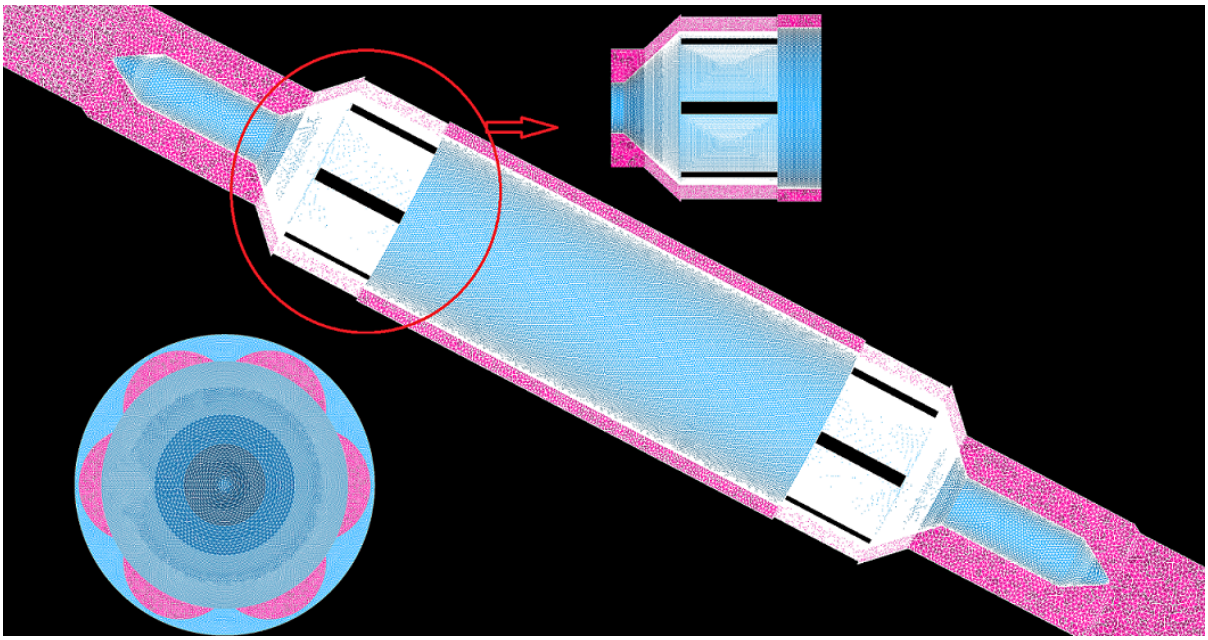


**Figure 4.14:** Tetrahedral mesh of the modified sensor

Similarly, simulations were performed with the simpleFoam solver for various pump flow rates. Here only the results of simulation for the fluid velocity of 0.488m/s at the sensor inlet are illustrated in figure 4.15 and 4.16, while the results of simulations for others various pump flow rates are summarise quantitatively in Table **E.1**.

The velocity of the fluid was the same for the entire sensor geometry except in the rozita and cone part, which gives the higher fluid velocity, as shown clearly in figure 4.15. As seen from figure 4.16, the pressure decreases in the flow direction and there is a more pressure reduction in the cone and rozita part. There is also a negative pressure at the few Sharpe edge near the cone and rozita part. This problem might be solved by making the mesh very fine (more dense) at the edges. A pressure drop of 63.49cmH2O is obtained from this simulation. This value is nearly the same with the experimental result, there was only approximately 4.8% error.

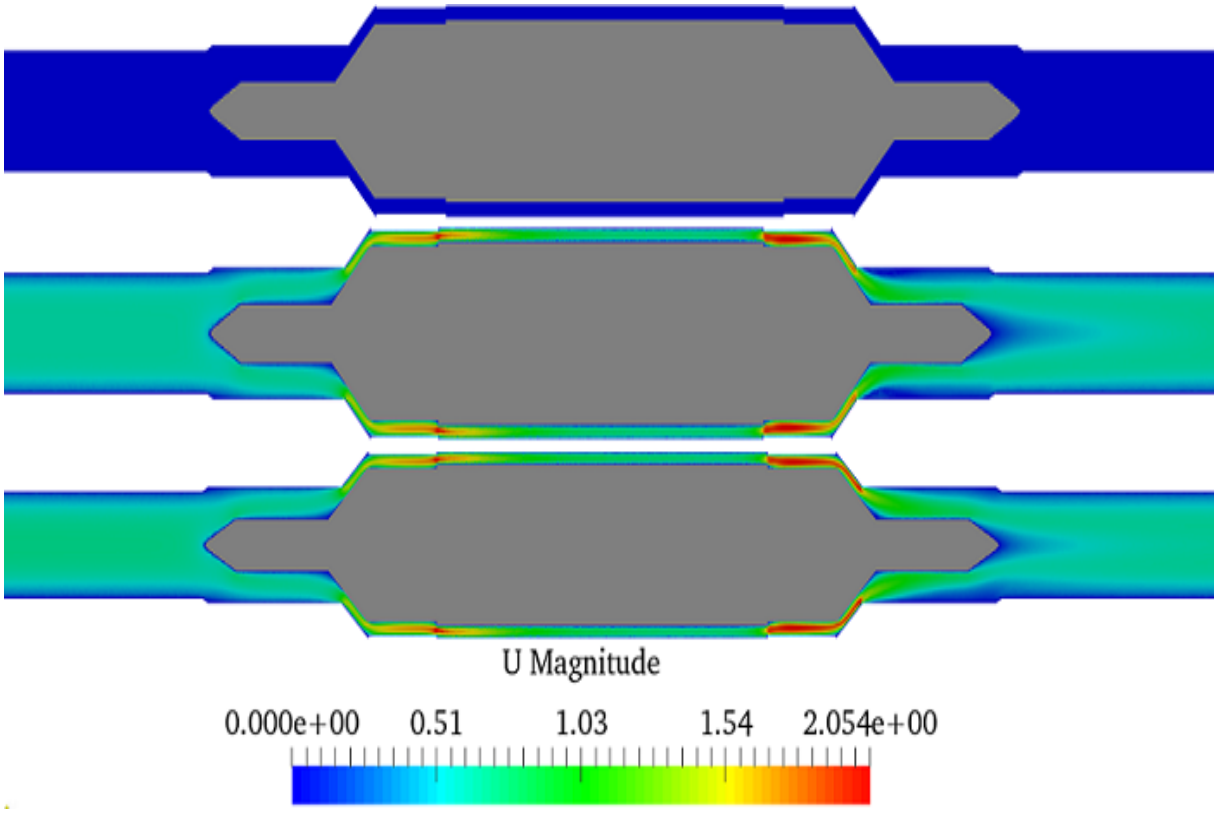



**Figure 4.15:** Velocity Field:top at t=0, middle at t=150 and bottom at t=300 sec

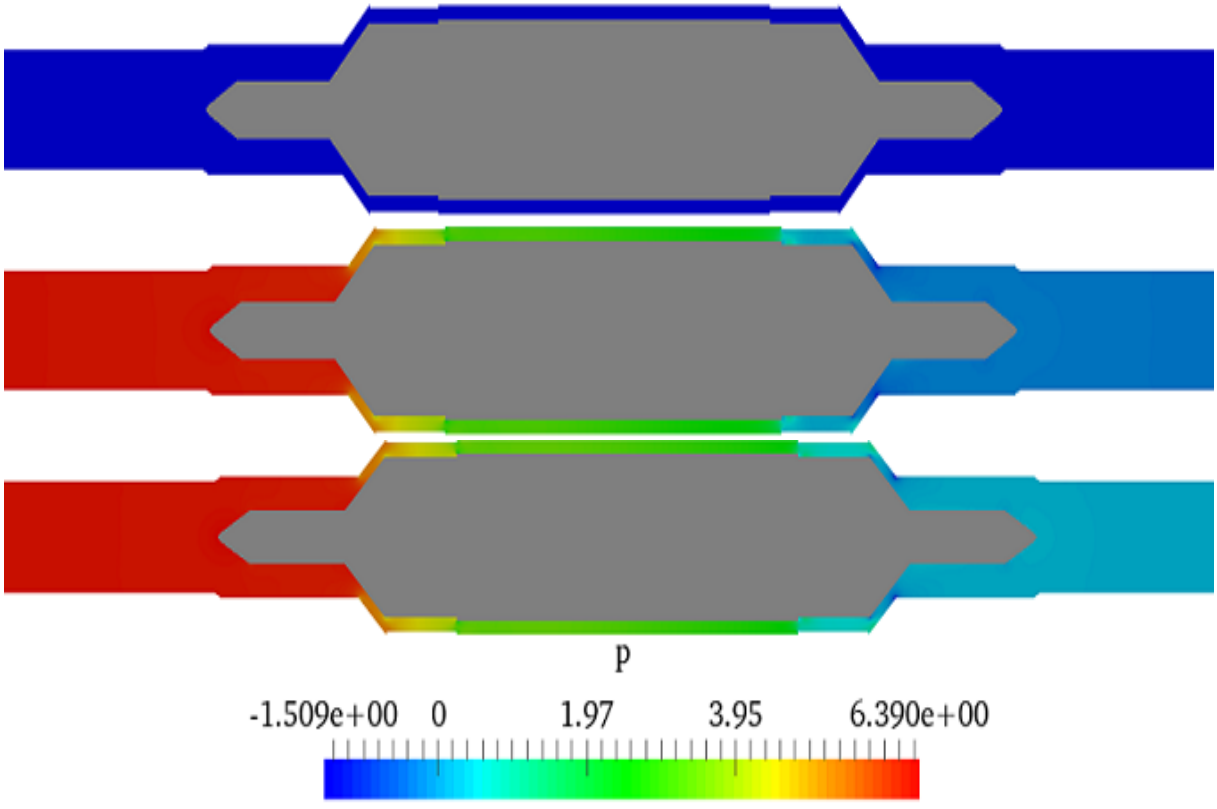



**Figure 4.16:** Pressure Field: top at t=0, middle at t=150 and bottom at t=300 sec

For a comparison, all the results of simulations from the four cases above and the experimental result for eleven pump flow rates are plotted in a single figure, as shown in figure 4.17. All the source dates used for generated this graph are found in Table **E.1**.

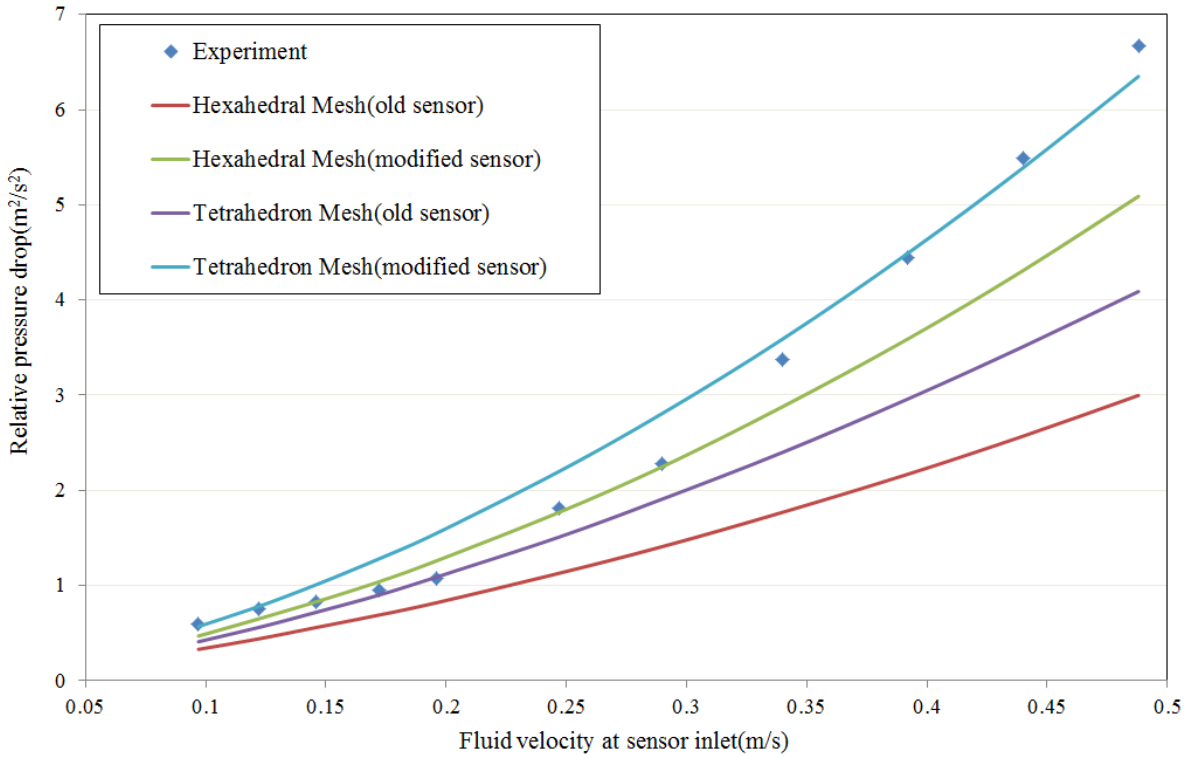



**Figure 4.17:** comparison of simulation results with the experiment result

All the results of simulations and figure 5.17 clearly show that tetrahedral mesh gives a better result of simulations over the hexahedral mesh.The result of the simulation for the tetrahedral mesh of the modified sensor agrees well with the experiment showing only 4.8% error. But in terms of the simulation execution time and RAM usage, the hexahedral mesh is better than the tetrahedral mesh. In conclusion, the conductivity sensor has a high pressure drop due to different cross sectional areas in some parts; the main effects are due to the rozita and cone parts.

The modified sensor with a tetrahedral mesh type gives a valid simulation result and it used for the next task of this thesis: Design of the injection system which yields the best distribution of tracer concentration throughout the sensor. All the results from this task are discussed in the next sections.

## 4.2 Injection System Design

The objective of this task is to design the tracer injector which can gives the best distribution of tracer concentration. Simulations were performed to visualize how the shape and length of the injector geometry affect the formation of the injection cloud and distribution throughout the sensor, besides the flow direction of the tracer injection. The counter-current injection fluid flow direction, the tracer injected in opposite direction of the fluid flow, is used for this task. This injection fluid flow direction studied by Oscar Pujol[6] previously and his work showed that it has a better mixing effect. So, the focused of this thesis is only on the shape and size of the injector geometry. For this purpose two kinds of injector geometries were studied. Both geometries are 90 degree bent cylinder type with different face shape at the one end of the geometry.

Geometries and meshes of the two injector types together with the sensor were created in Salome. For both injector design the entire geometry was meshed with different refinement level. The injection cylinder together with injector, cone and rozita part of sensor are denser than the other part of the geometry. Because of the size of the entire geometry, only the geometry and mesh of the injection cylinder together with half of the sensor part was shown in this report. But all simulations were done using the whole geometry. Several simulations were performed for each of the two cases using the simpleFoam and the scalarTransportFoam solvers. The initial fluid velocity of 0.488 m/s at the inlet boundary patch of the geometry which is delivered by the pump maximum speed and the injection velocity of 5m/s (which is calculated from the injection time of 1 sec, tracer volume of 1ml and 1mm diameter of injector inlet) was used for all simulations. Results from both cases are discussed in the next two sub-sections.

### 4.2.1 Bent cylinder injector with needle shape at one end of the geometry

The Geometry and mesh of injector together with the sensor are shown in figure 4.18 and 4.19, respectively.

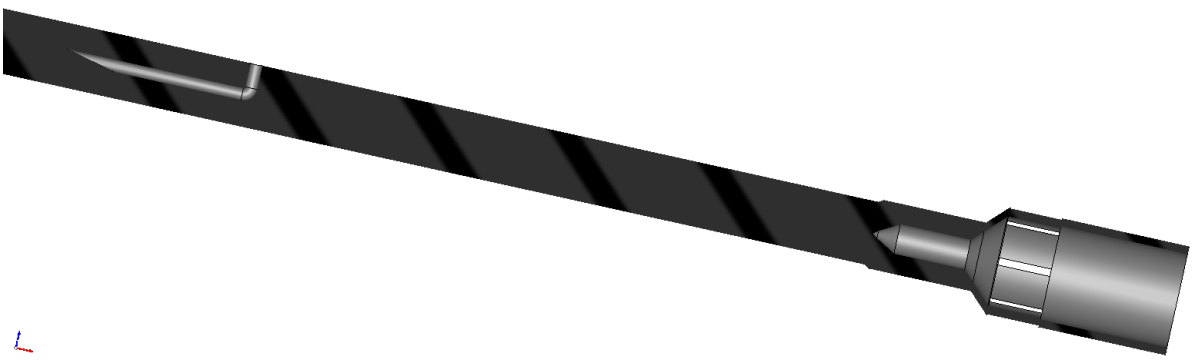


**Figure 4.18:** Geometry of the injector part together with half of the sensor

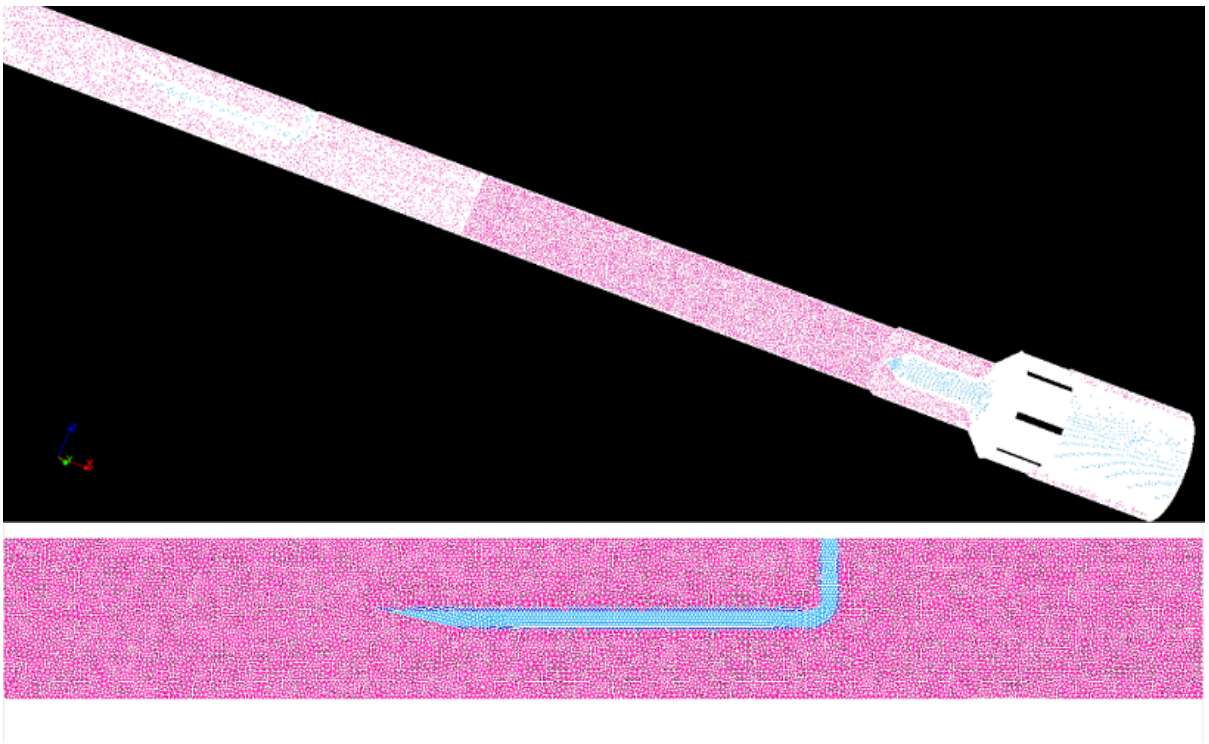


**Figure 4.19:** Mesh: top, injector part together with half of the sensor and bottom, injector part only

This mesh has 11,094,604 of tetrahedral cells without non-orthogonal and skewed faces. This indicates that the mesh has a good quality and granularity. As seen from figure 4.19, the geometry of the injector was not meshed, since it is not consider as a fluid body and only the needle face of the injector was used as an inlet boundary patch for the tracer injection, which means the direction of the injection is in the opposite direction of the fluid flow. This implicated that the shape and size of geometry of the injector does not have any effect on the formation of the injection cloud. Hence, the formation of the cloud only depends on the needle shape of the injector inlet boundary patch. But this is not the case in reality, as the tracer injects at the top of the injector and flow through the injector geometry before it goes out and mixed with the fluid. Using this mesh, simulations were performed and the results of simulations are discussed as follow:

First, the simpleFoam simulation was done at the time when the tracer injection taken place to evaluate the pressure and velocity fields. For simplicity, this simulation is denoted by simpleFoam simulation1. The result obtained from the simulation is shown in figure 4.20 and figure 4.21 for velocity and pressure fields, respectively

Because of the tracer, pressure build up at the top of the injection cylinder part near the injector tip, as clearly shown in figure 4.21. Due to this, most of the fluid passes through the bottom. And as seen from figure 4.20, the tracer cloud was formed at the top side which is unexpected, as the tracer was injected in downward direction.

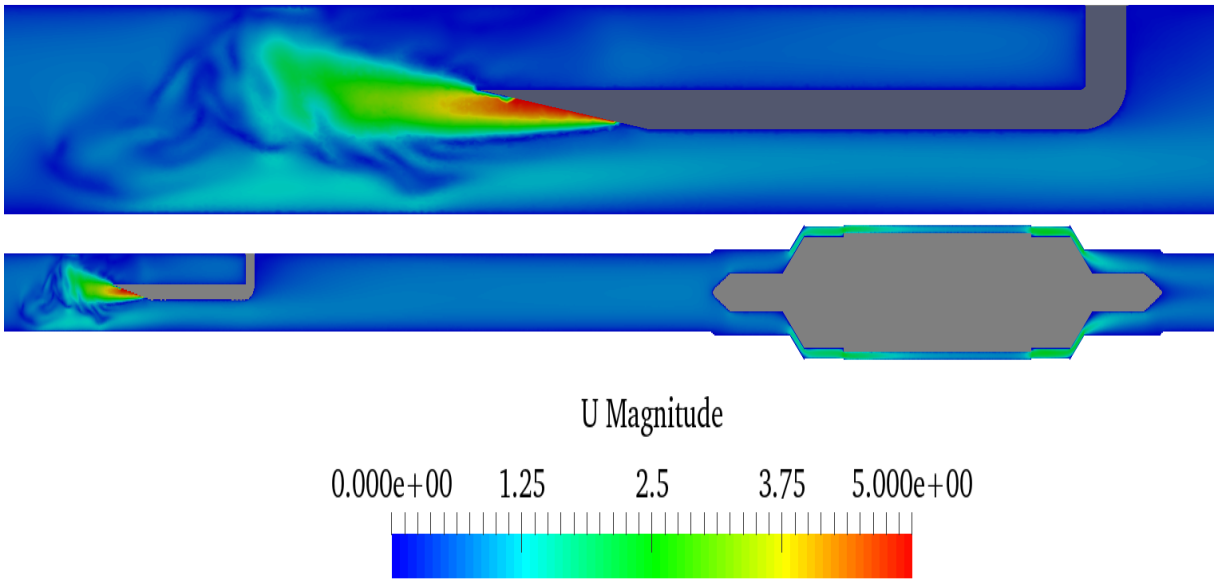



**Figure 4.20:** Velocity field during injection at t=300 sec. top shows at injection system part only while bottom shows for the full geometry

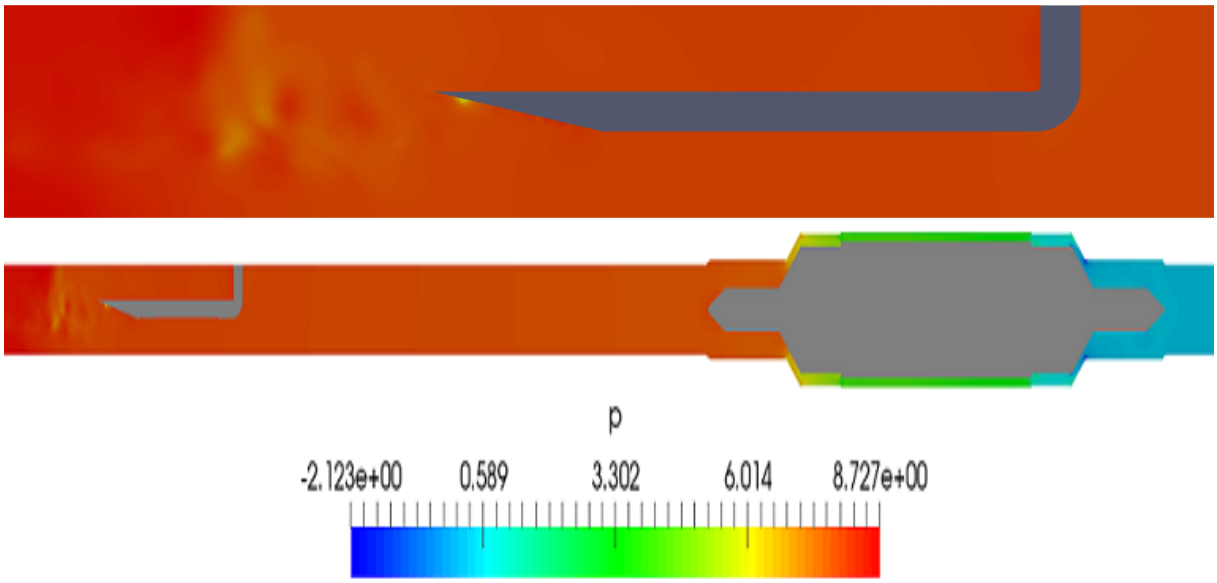



**Figure 4.21:** Pressure field during injection at t=300 sec. top shows at injection system part only while bottom shows for the full geometry

Using the velocity and $\phi$ fields obtained from simpleFoam simulation1, the first scalarTransportFoam simulation was performed to evaluate the tracer scalar concentration distribution around the sensor during the tracer was injected. Similarly, for simplicity this simulation is denoted by scalarTransportFoam simulation 1. Figure 4.22 shows the evolution of the injection cloud and distribution through the geometry.

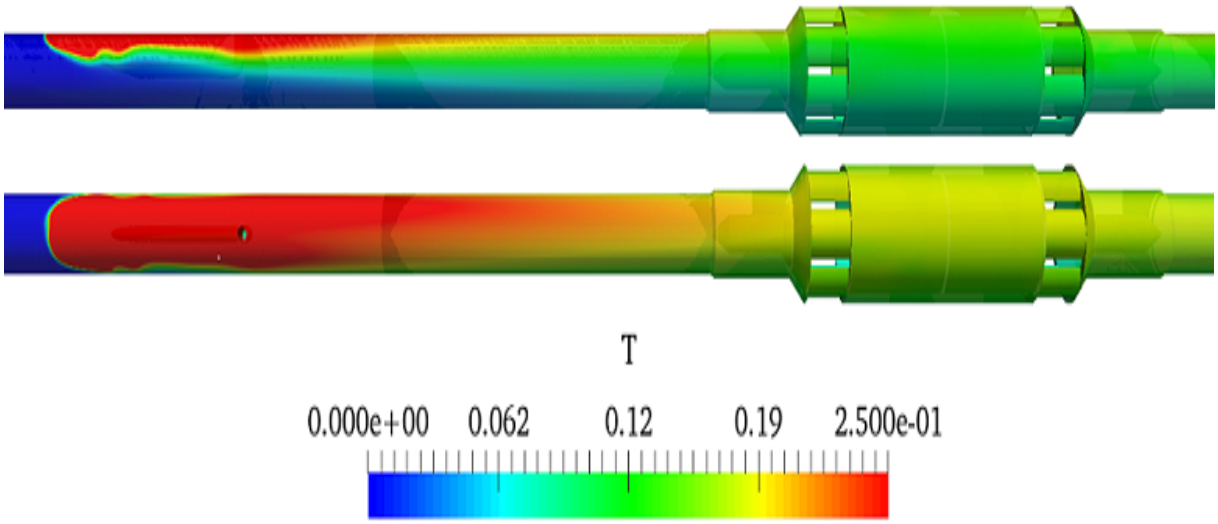



**Figure 4.22:** Evolution of the injection cloud during injection at t=1 sec. Top figure shows evolution of cloud in y axis while bottom figure shows it in the z axis

Second, simpleFoam simulation was performed during the tracer injection was stopped/finished to evaluate the pressure and velocity fields. This simulation is denoted by simpleFoam simulation 2. Figure 4.23 is illustrated the result obtained from this simulation.

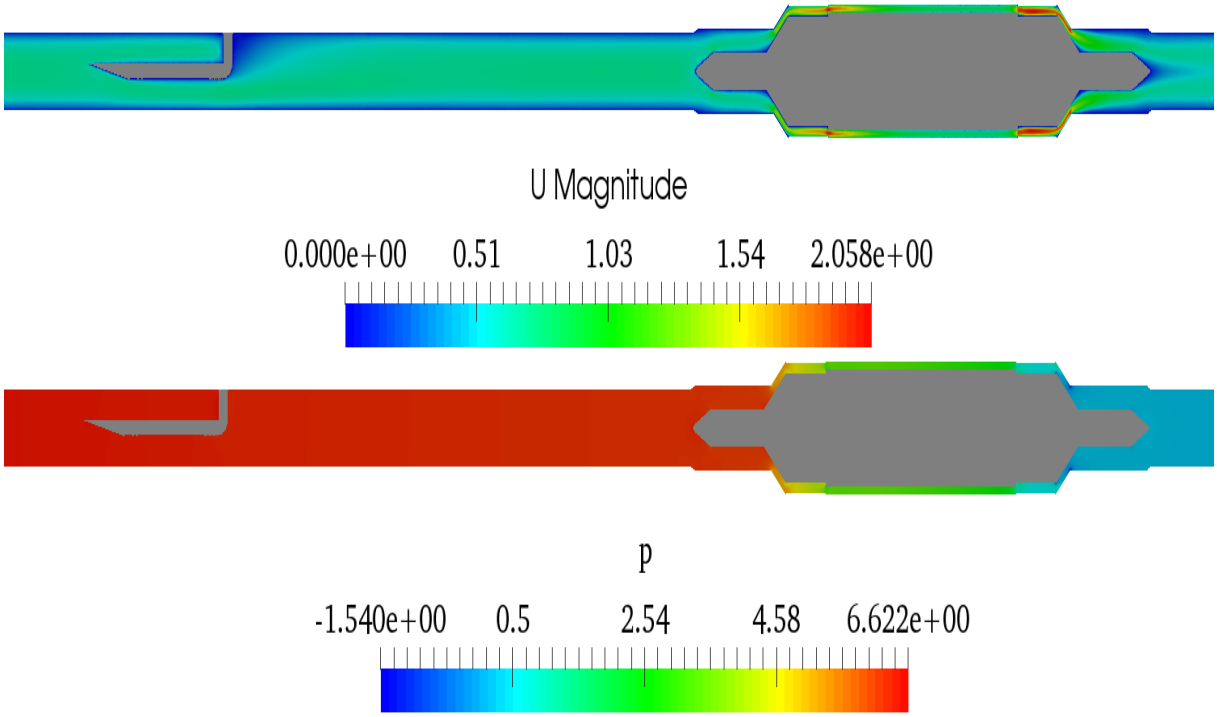



**Figure 4.23:** simpleFoam simulation result during injection stopped at t=300 sec. top, velocity while bottom, pressure

As seen from Figure 4.23 for velocity field, the presence of the injector geometry affects the fluid flow to some extent. Because of the vertical cylinder part of the injector geometry some void space was created, as the fluid is pumped continuously with a given velocity. This is

clearly shown in figure 4.23 with a 0 velocity near the wall of the vertical cylinder part of the injector.

Finally, using the velocity and $\phi$ fields from simpleFoam simulation 2 and the scalar field from scalarTransportFoam simulation 1 at the end time of the simulation, which are 300 sec for velocity and $\phi$ fields and 1 sec for scalar field, the second scalarTransportFoam simulation was performed. This is used to visualize the distribution of the tracer concentration throughout the sensor. The scalar field value obtained from scalarTransportFoam simulation 1 used as initial condition for the scalar field in this simulation. Figures 4.24 and 4.25 show the distribution of the injection cloud since injection is finished until it has passed through the sensor in y and z axis, respectively.

Figures 4.22, 4.24 and 4.25 show that most of the tracer concentration goes through the top of the sensor. This is not the expected phenomena from the given injection design. The expected phenomena for this injector design was either tracer distributed throughout the sensor or most of the tracer concentration goes through bottom of the sensor geometry, Since the tracer is injected in downward direction.

Generally, this injector design did not yield a good mixing as well as a better distribution of tracer concentration throughout the sensor geometry. This is not a desired phenomenon for measuring the conductivity later. For measuring conductivity, well mixed and evenly distributed tracer concentration throughout the whole sensor geometry is important.

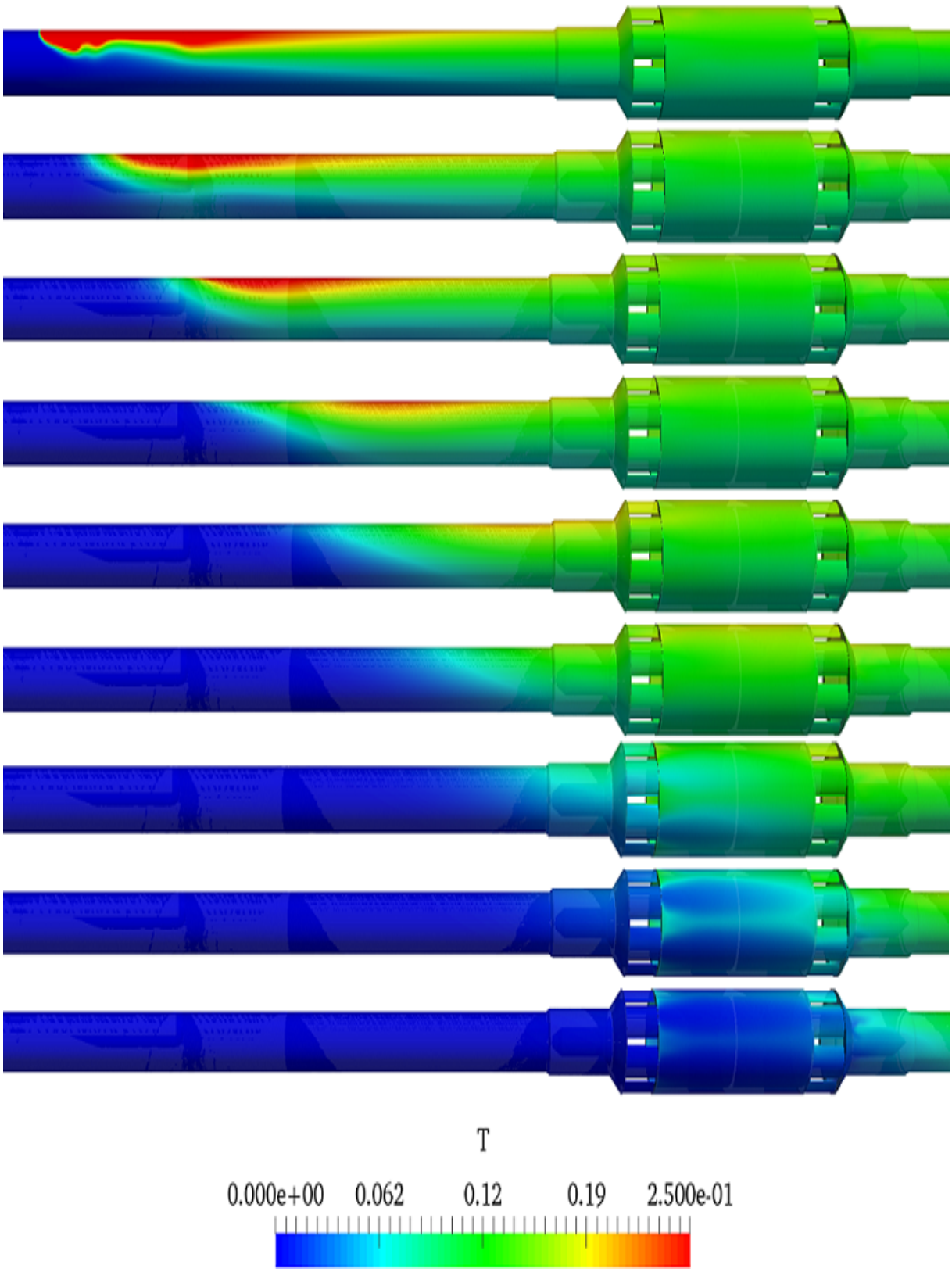



**Figure 4.24:** Evolution of the injection cloud showed in the y axis, side view

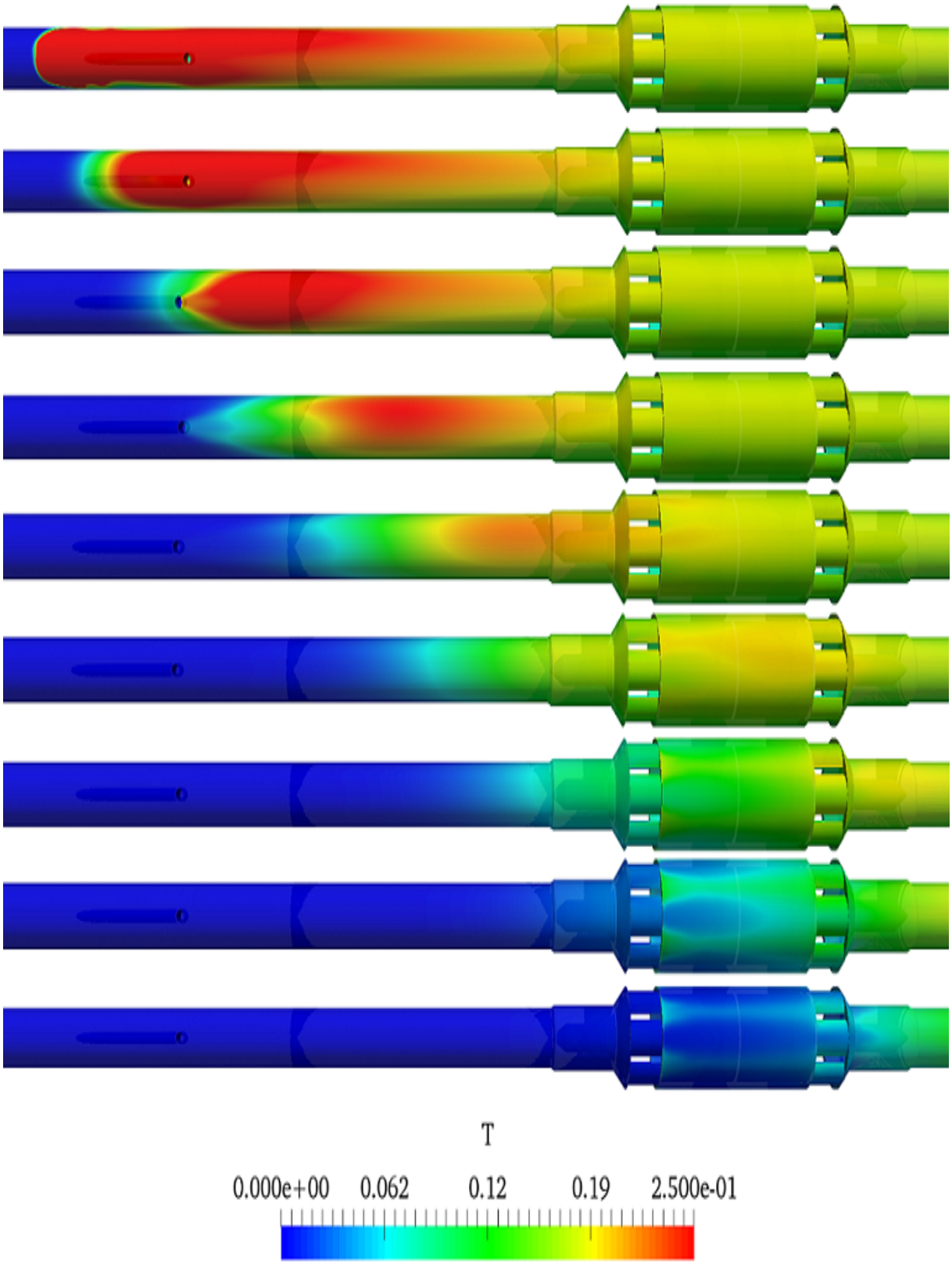



**Figure 4.25:** Evolution of the injection cloud showed in the z axis, top view

### 4.2.2 Bent cylinder injector geometry with circular shape at both end

Now the injector has circular shape in both end of the geometry with small length. Figures 4.25 and 4.26 show the geometry and mesh of the injector part together with the sensor, respectively.

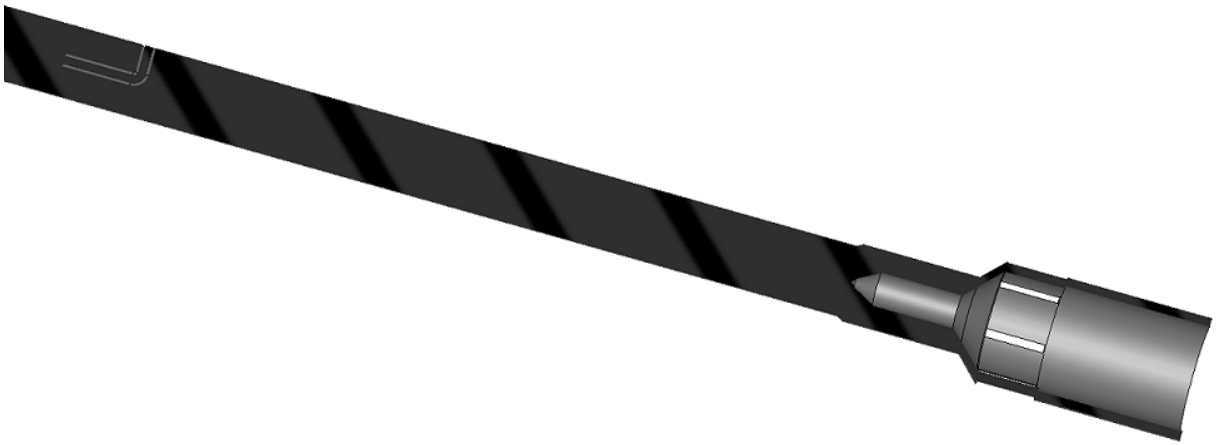


**Figure 4.26:** Geometry of the injector part together with half of the sensor

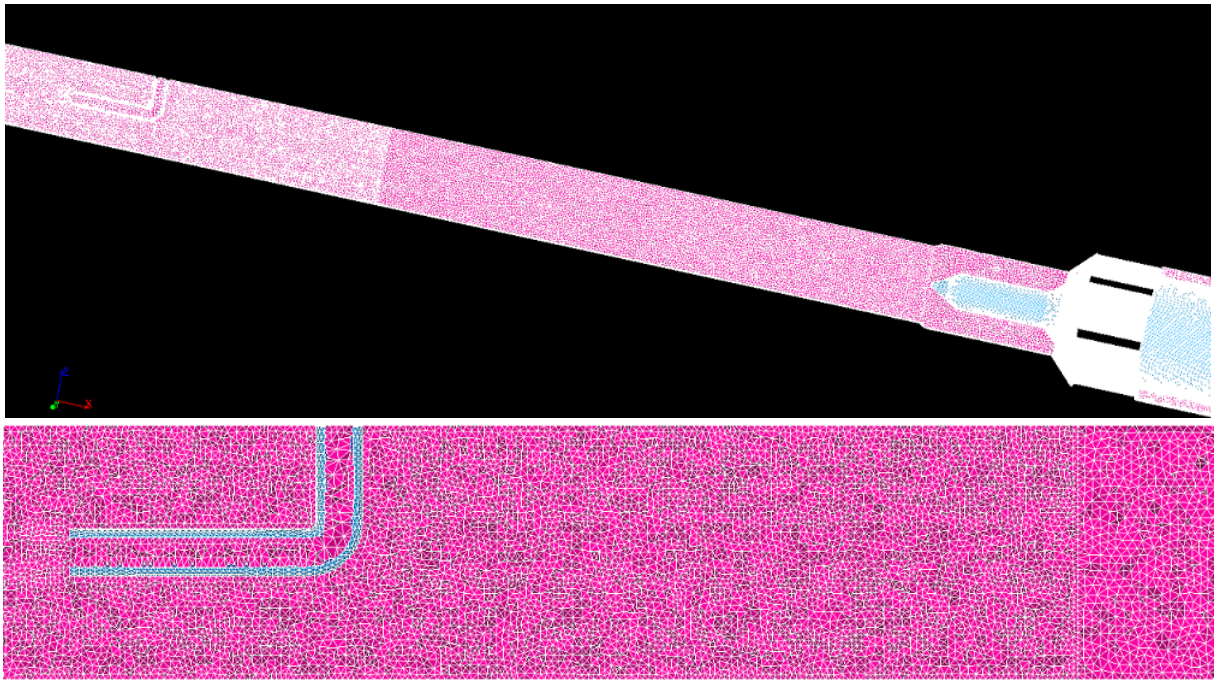


**Figure 4.27:** Mesh: top, mesh of the injector together with half of the sensor and bottom, mesh of injector part only

Figure 4.27 clearly shows that the inside cylinder of the injector is consider as a fluid part while the outer cylinder of the injector consider as a wall. Now the top faces of the injector is the inlet boundary patch for the injection, which means the tracer is injected in the top-down direction. But due to the bent part of the injector geometry the tracer flow direction is changed into the opposite direction of the fluid flow. The mesh has 10,707,076 tetrahedral cells and it has a good quality and granularity without non-orthogonal and skewed faces.

The same simulation procedure, boundary conditions and parametrise as the above case were used also for this case. Hence, only the results of simulations are discussed here.

Results obtained from simpleFoam simulation1, simpleFoam simulations during the tracer was injected, at the end time of the simulation are shown in figure 4.28 and 4.29 for velocity and pressure fields, respectively.

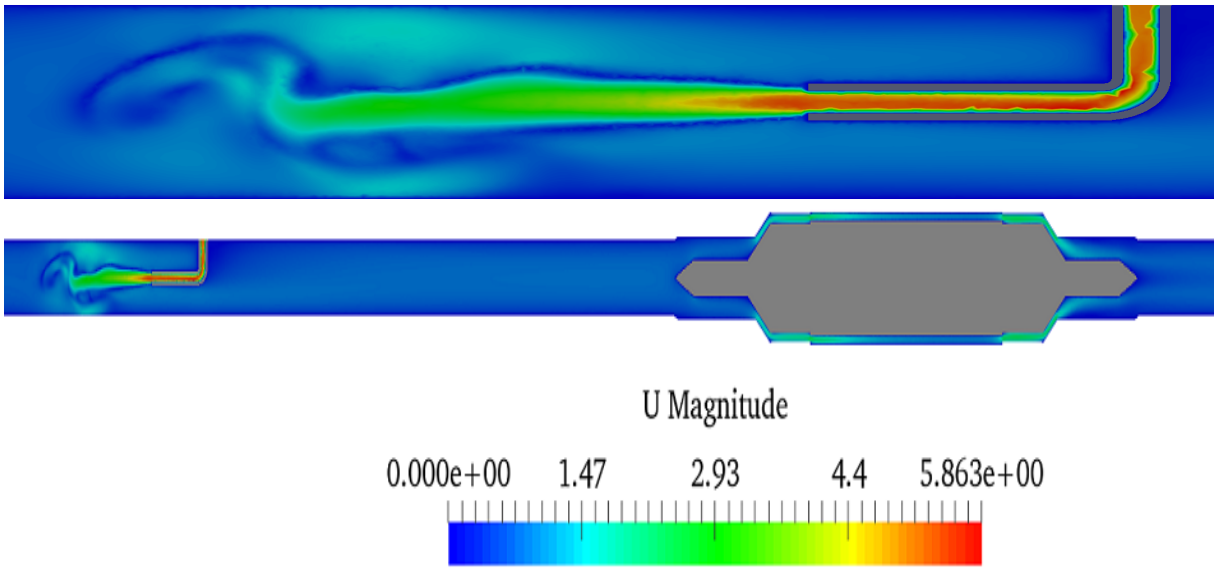



**Figure 4.28:** Velocity field during injection at t=300 sec. top shows at injection system part only while bottom shows for the full geometry

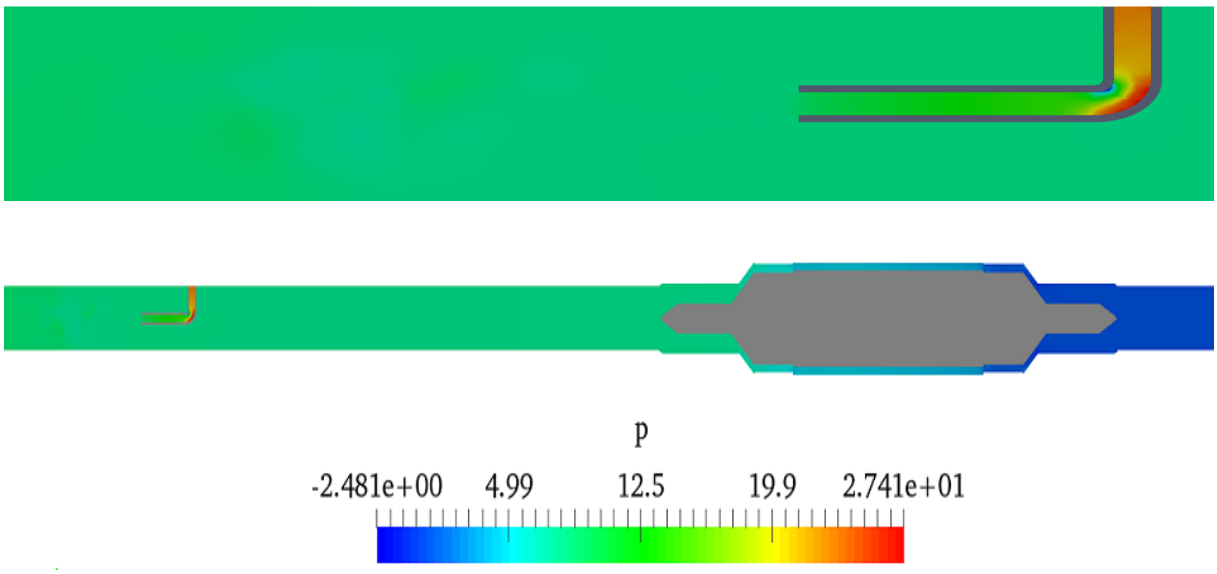



**Figure 4.29:** Pressure field during injection at t=300 sec. top shows at injection system part only while bottom shows for the full geometry

As shown clearly in figure 4.29 of pressure filed, there is a high pressure inside the injector near the bent. This is due to change of flow direction of the tracer. The pressure decreases

rapidly after the bent part and finally approaches the pressure as it meets the main stream.

The tracer is penetrated into the fluid in the center of the geometry and makes a uniform and thin cloud, as clearly shown in figure 4.28. This situation gives a better mixing effect.

Using $\phi$ and velocity fields obtained from simpleFoam simulation1, the first scalarTransport-Foam simulation was done. Results obtained from the scalarTransportFoam simulation1 at the end time of the simulation are shown in figure 4.30.

Figure 4.30 clearly shows that the concentration distribution in both directions looked almost the same, which indicates a good mixing and distribution of tracer concentration throughout the sensor during tracer was injected.

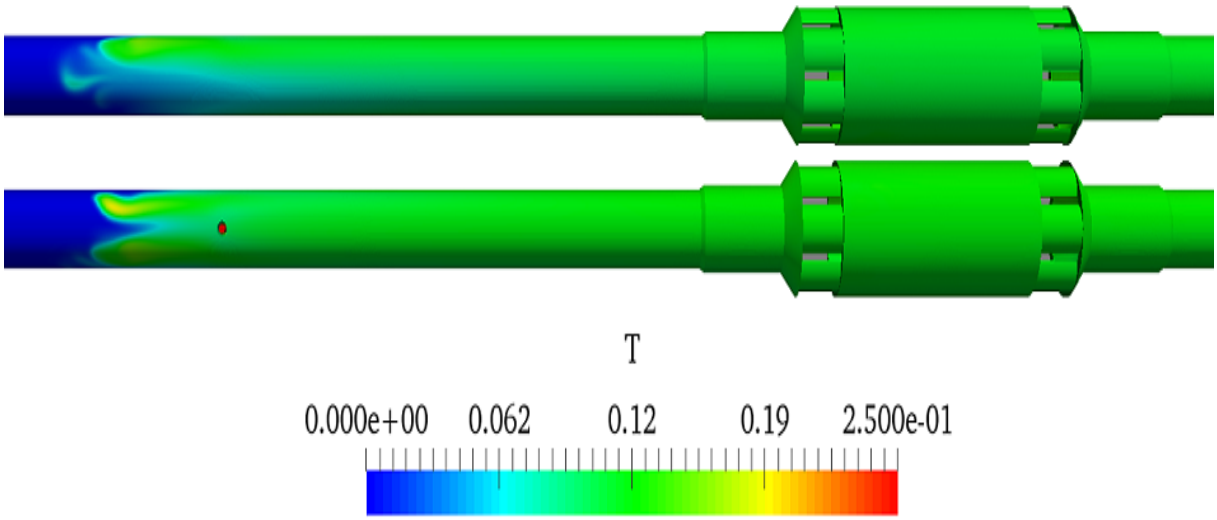



**Figure 4.30:** Evolution of the injection cloud during injection at t= 1sec. Top figure shows evolution of cloud in y axis while bottom figure shows in z axis

Results obtained from the second simpleFoam simulation, simpleFoam simulation during the tracer injection was stopped, at the end time of the simulation are shown in figure 4.31.

As seen from figure 4.31 of velocity field, there is zero velocity inside the entire injector; this is because of the presence of stationary fluid inside the injector. Similarly with the above case, the presence of the injector geometry gives a 0 velocity near the wall of the vertical cylinder of the injector.

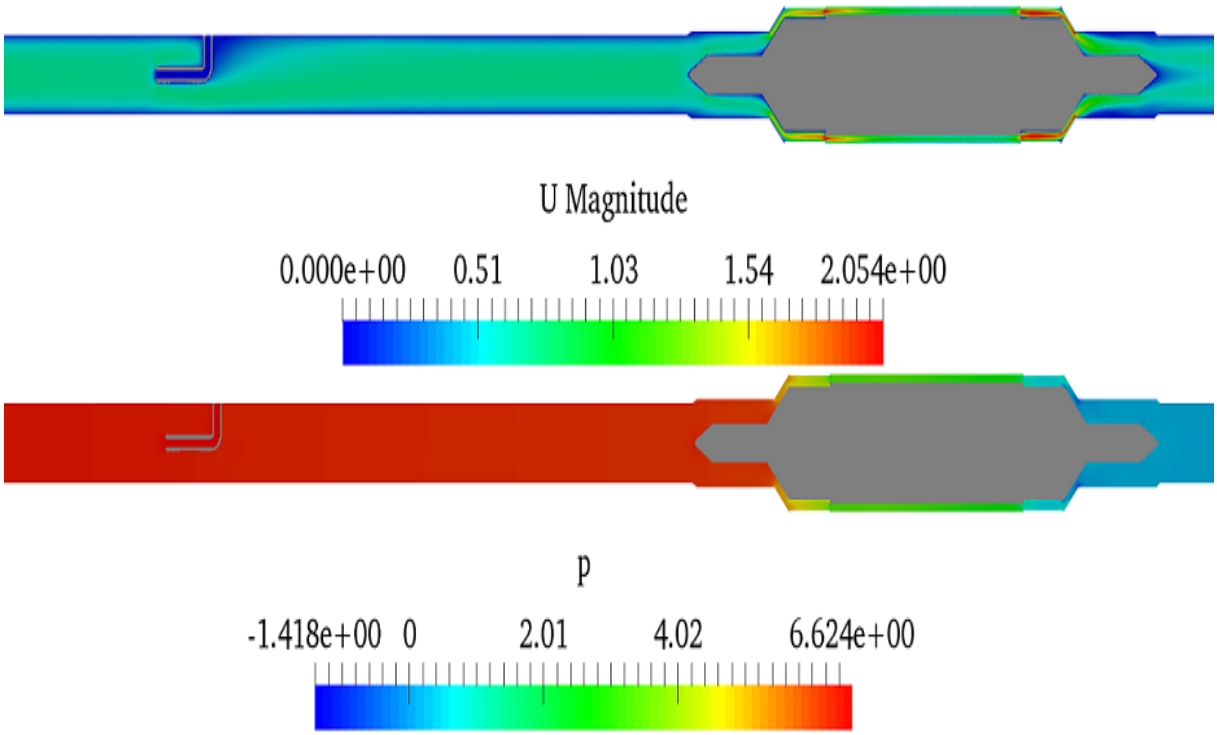



**Figure 4.31:** simpleFoam simulation result during injection stopped at t=300 sec. top, velocity while bottom, pressure

Now, using the $\phi$ and velocity fields obtained from the second simpleFoam simulation and scalar concentration filed obtained from the first scalarTransportFoam simulation at the end time of the simulations, the second scalarTransportFoam simulation was performed. Results from this simulation are illustrated in figures 4.32 and 4.33.

A very small amount of tracer remains inside injector held by the stationary fluid, since the injection was stopped and it could not get enough force in order to go out from the injector, as seen from figure 4.31 with small red dot in the inlet part of the injector. After some time it will be totally dilute with the fluid, which is come into the injector through the bottom opening of the injector.

The evolution of the injection cloud throughout the sensor part is almost looked the same in both side of the geometry, as seen from figures 4.32 and 4.33. Result indicated that this design of the tracer injector and the counter-current injection direction gives a good mixing and distribution of the tracer concentration throughout the entire sensor.

Results from both injector design show that the formation of the tracer cloud and its distribution throughout the sensor depends on the shape and size of the injector tube in addition to the tracer flow direction.

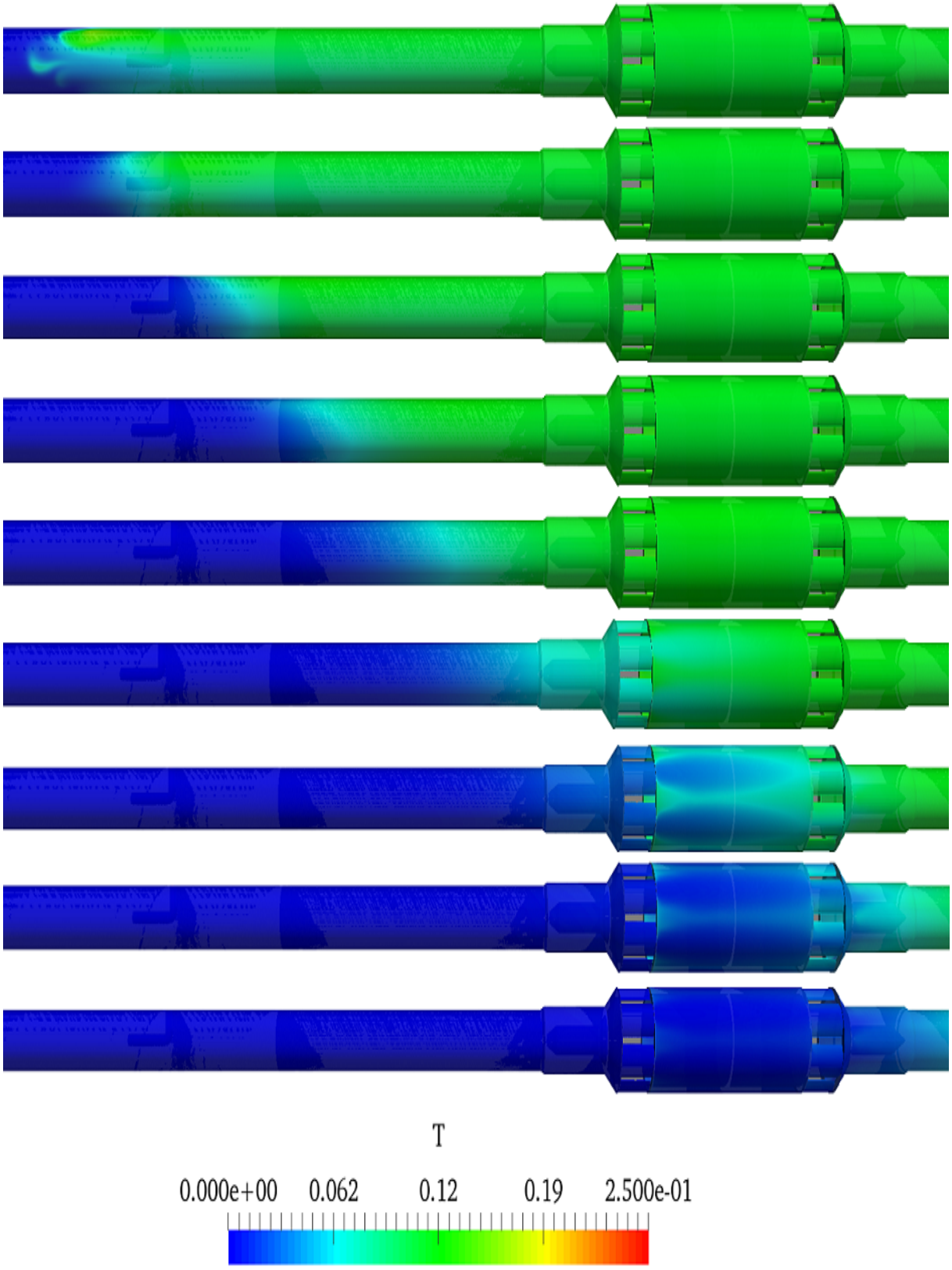



**Figure 4.32:** Evolution of the injection cloud showed in the y axis, side view

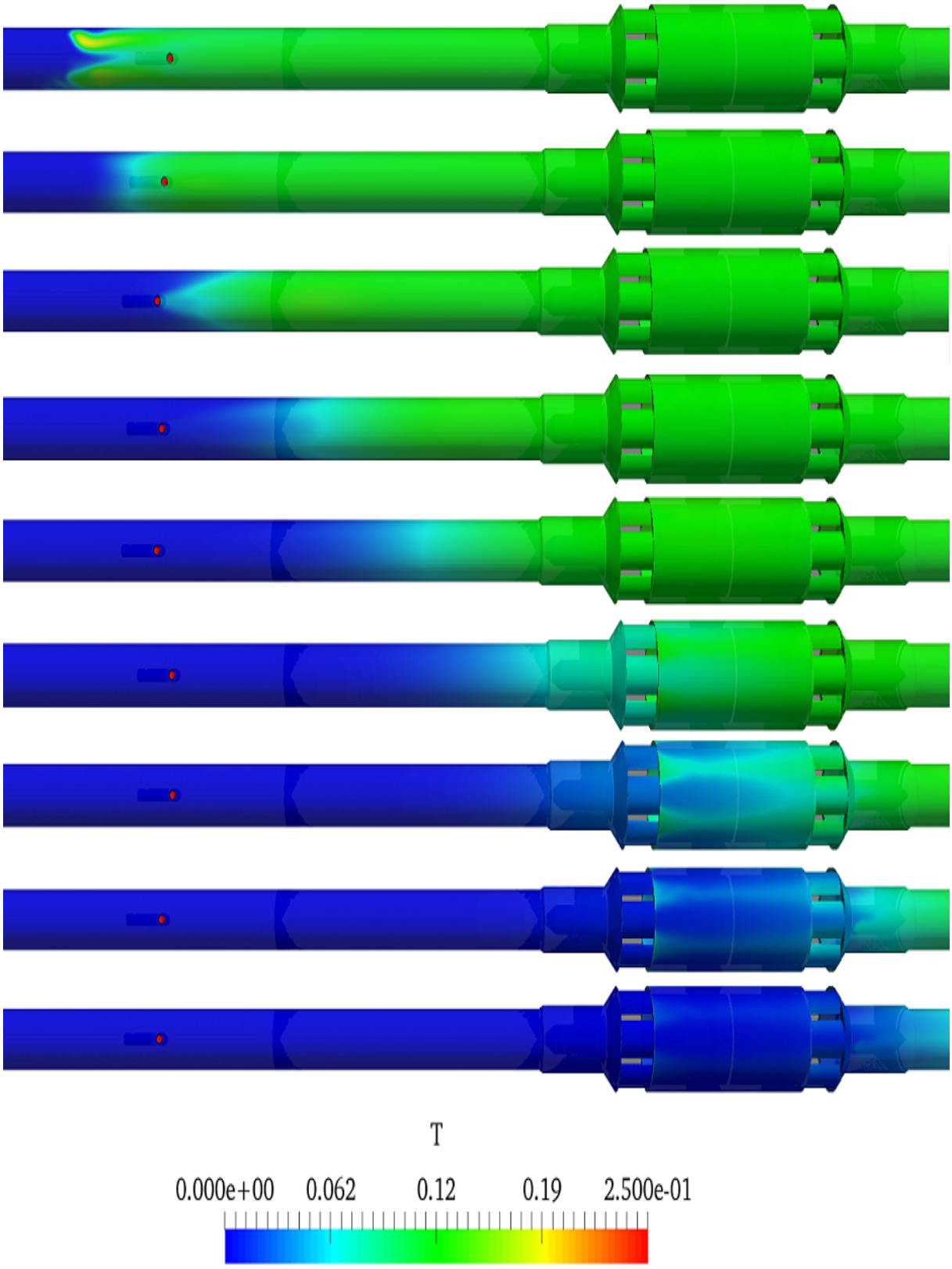



**Figure 4.33:** Evolution of the injection cloud showed in the z axis, top view

## 4.3 Structured mesh of the conductivity sensor

Structured mesh has a lot of advantages over unstructured mesh, for example it utilise less memory storage and it is computationally faster. But creating structured mesh is more difficult than creating unstructured mesh, since a lot of trade offs must be considered.

In this section, some work was performed to generate the structured mesh (pure hexahedral mesh) of the conductivity sensor in Salome.

The geometry of the sensor was divided in to different small parts and generated using a block to get sex face everywhere in the sensor geometry. Then using partition and glue face operation all parts put together. Half part the sensor geometry shown in figure 4.34.

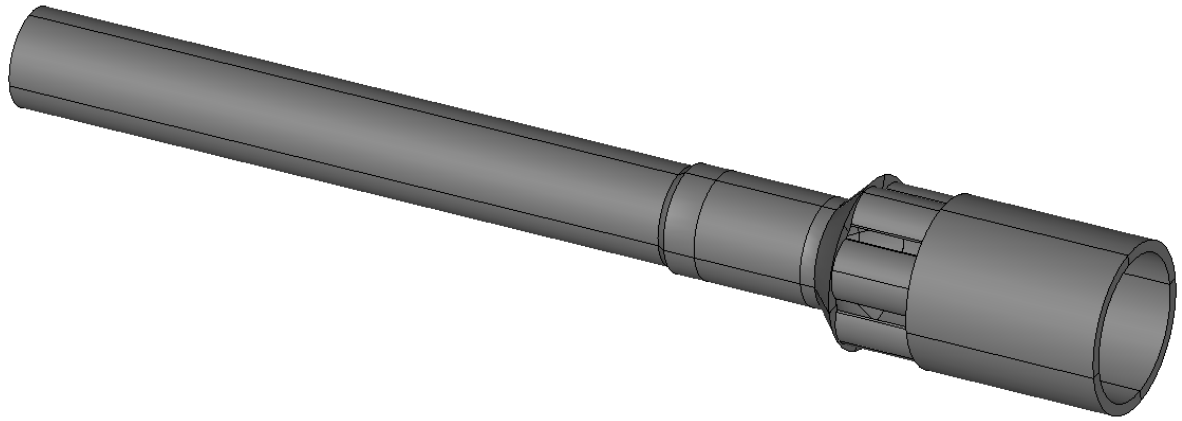


**Figure 4.34:** Half part of the sensor geometry

The sensor geometry was meshed with hexahedral element shape (structured mesh) using a 3D automatic hexahedralization hypothesis without defined the boundary patches. And it was imported to OpenFOAM successfully. This mesh is shown in figure 4.35.

As seen from figure 4.35, the mesh has good and uniform quality and large number of uniform layers in the narrowest part of the sensor.

The problem was happened when tried to import the mesh with the boundary patches, the process of importing the mesh into OpenFOAM is terminated with some errors.

To investigate the problem, only some parts of the sensor was meshed and the boundary patches was defined. Then the mesh was imported to OpenFOAM successfully. Mesh of the pin_part and sensor_inlet part shows in figures 4.36 and 4.37.

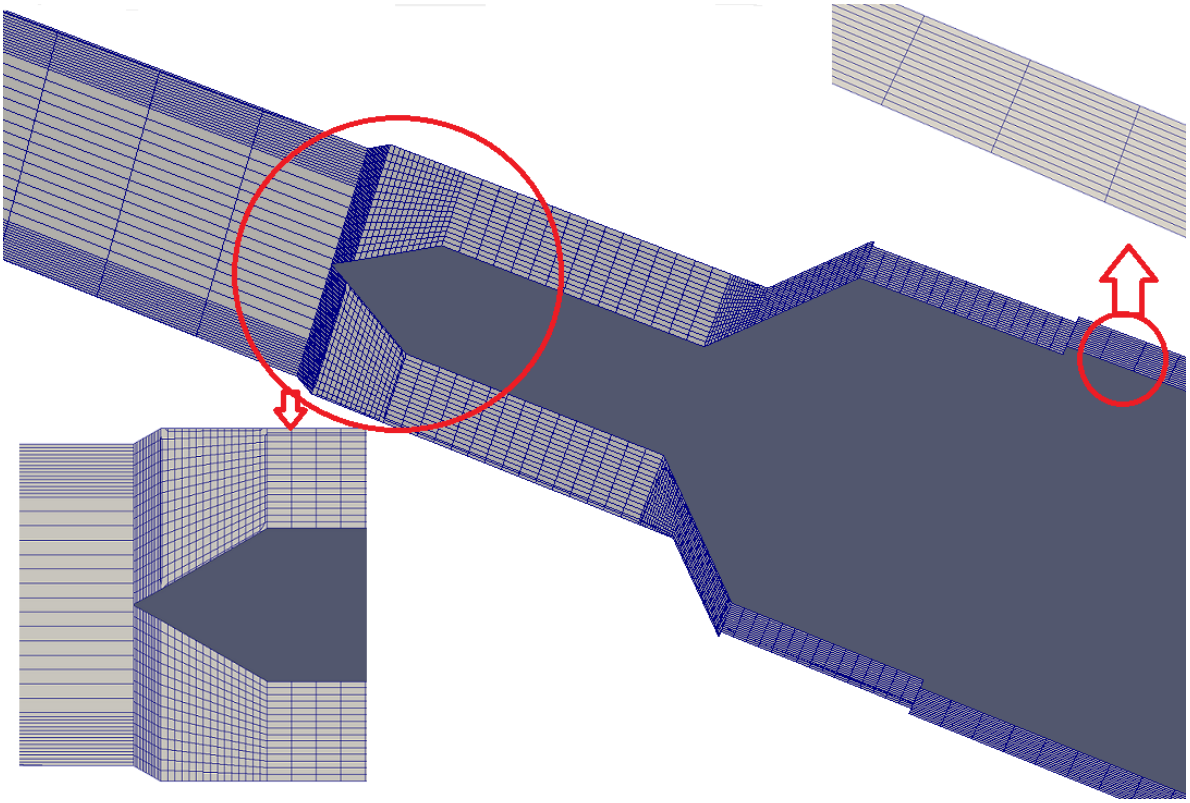


**Figure 4.35:** Structured mesh of the sensor half part

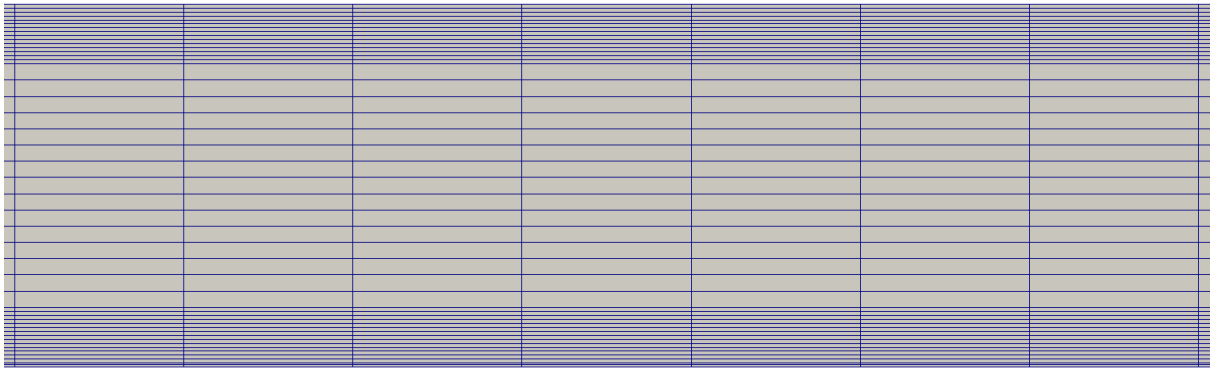


**Figure 4.36:** Structured Mesh of the sensor_inlet part

Since the individual parts of the sensor geometry was meshed and imported to OpenFOAM with the boundary patches without error, the problem is not because of geometry. The problem might be on the operation used to put all parts together. As mentioned earlier in this section, all parts generated using blocks and there were a face difference between parts of the sensor at the contact point/place. In order to put all parts together with some kind of operation might be requires the same face at the contact place of the parts, since most of the operation creates a common face from two faces. To overcome this problem requires a detail investigation.

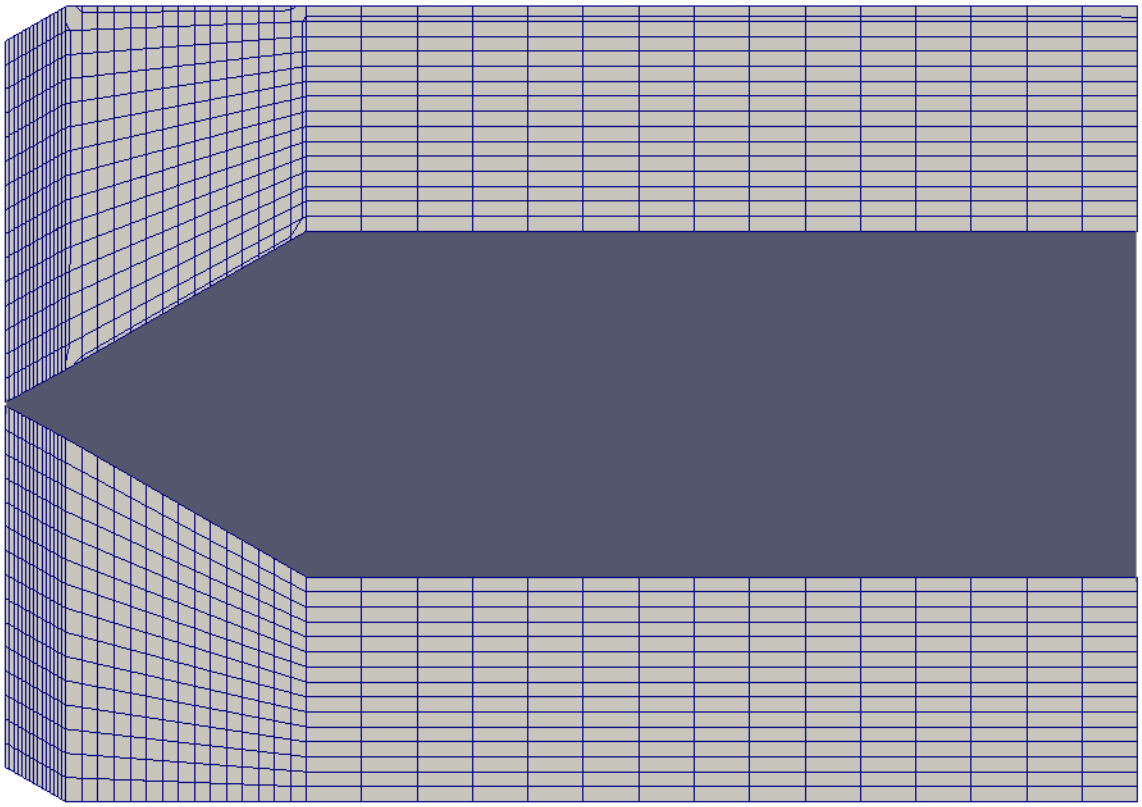


**Figure 4.37:** Structured Mesh of the pin_part

Chapter 5

# Conclusion and scope for future work

## 5.1 Conclusion

The objectives of this thesis is to use CFD simulations to validate the design of the in-house conductivity sensor and to design the injection system which yields the best distribution of tracer concentration throughout the sensor.

A validation of the in-house conductivity sensor was conducted by performing a simulations with the simpleFoam solver. Two types of mesh of the sensor, a Hexahedral mesh generated in snappyHexMesh and a tetrahedral mesh generated in Salome, were used. In terms of the execution time and RAM usage the hexahedral mesh performed better than the tetrahedral mesh of the sensor. But in terms of the quality of the mesh and the accuracy of the result of simulations the tetrahedral mesh of the modified sensor was better than the hexahedral mesh; the result of the simulation for the tetrahedral mesh agrees well with the experiment showing only 4.8% error. Results showed that snappyHexMesh was not a good candidate to generate the mesh for the conductivity sensor, since it has a lot of sharp edges. The conductivity sensor has a high pressure drop due to different cross sectional area in some parts; the main effects are due to in the rozita and cone parts. Therefore, either design improvement in rozita and cone part only or completely a new design of the conductivity sensor is necessary which can gives low pressure drop and uniform fluid velocity throughout the sensor.

A design of the tracer injection was conducted by computation of fluid dynamics modeling. The model contained two parts, first velocity and $\phi$ fields were calculating using the simpleFoam solver and then transport of the tracer (passive scalar) was evaluated using the scalarTransportFoam solver. Two types of a 90 degree bent cylinder injector design, one with large length and a needle shape at one end of the geometry and the other is circular shape in both ends with small length, were studied. For both injector designs, a counter-current injection flow direction was used. Results showed that the evolution and distribution of the tracer cloud passing through the sensor is affected by the shape and size of the injection tube and the tracer flow direction. The result of simulation for a 90 degree bent cylinder with needle shape at the one end of the injector gives unexpected phenomenon, as the tracer cloud is formed at the top of the pipe and not as expected at the bottom. This result also contradicts result from previous work of Oscar Pujol[6], which gives a tracer cloud formation at the center of the pipe near the tip of the injector for the same injector design.The second injector design gives a good mixing and

distribution of the tracer. Based on the simulation, I recommended to adding a mixer after the injector and before the sensor inlet in order to improve the mixing and distribution of the tracer concentration.

## 5.2 Scope for future work

The aim of the future works are to consider the conductivity equation in the simulation and generate a structured, pure hexahedral mesh and perform all the simulations for the manifold design. Particularly for the conductivity measurement it is better to use a structured mesh, since the conductivity equation is linear.

Some work was done adding the conductivity equation in the scalarTransportFoam solver. This work is not included in the report because it could not be completed. It is not an easy task but certainly possible to include this equation in the OpenFOAM simulations. The scalarTransport-Foam solver was modified first, by added the conductivity equation after the scalar Transport equation in the **scalartTransportFoam.C** file, because the conductivity depends on the tracer concentration. And second, the conductivity is defined as a scalar variable in the **createFields.H** file. It compiled and calculates the conductivity without any error. But the problem was it calculated the conductivity for the entire cell of the geometry which is unnecessary, since we only want to calculate the conductivity in specific part of the conductivity sensor. This problem might be solved by extracting the entire cell id and applying the conductivity equation for those specific cells of the sensor part only for the reconstructions of the conductivity measurement. The modified **scalartTransportFoam.C** and **createFields.H** are located in Appendix**F**.

# Bibliography


[1] J.D. Anderson. Computational fluid dynamics. The basics with applications. McGraw-Hill, 1995

[2] J. Blazek. Computational Fluid Dynamics. Principles and Applications. ELSEVIER. 1st ed., 2001

[3] O. levenspiel. Chemical Reaction Engineering. John Wiley & Sons. 3rd ed., 1999

[4] G.Hill Charles. Introduction to Chemical Engineering Kinetics & Reactor Design. John Wiley & Sons. 2nd ed., 2014

[5] Abdulnaser Sayma. Computational fluid dynamics. Ventus publishing Aps, 2009

[6] Oscar Pujol. Design of injector and conductivity sensor. July 2013

[7] SALOME official webpage (www.salome-platform.org). May 2016

[8] OpenCFD Ltd. OpenFOAM -user guide, version 2.4.0. 21st may 2016

[9] http://www.engineeringtoolbox.com/water-dynamic-kinematic-viscosity-d_596.html. May 2016

[10] https://www.hpc.ntnu.no/display/hpc/NTNU+HPC+GROUP. 2016

[11] http://ntl.bts.gov/DOCS/ch5.html. June 2016

[12] http://perso.usthb.dz/ nhannoun/research.html. June 2016

Appendix A

# simpleFoam Files

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p; //Units are Pascals divided by the density
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            fixedValue;
        value           uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }
// only the above three bondary pathes used for the conductivity sensor modeling case
// The last boundart patch added for the injection system design case
    injection-inlet
    {
        type            zeroGradient;
    }


}

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

**Figure A.1:** Pressure Boundary Condition

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.4.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;  // This is the fluid velocity at the various boundaries
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (0.488 0 0);
    }
    outlet
    {
        type            zeroGradient;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
// only the above three bondary pathes used for the conductivity sensor modeling case
// The last boundart patch added for the injection system design case
    injection-inlet
    {
        type            fixedValue;
        value           uniform (0 0 -5);  // This value change to uniform (0 0 0)
    }                                      // for the second simulation

}

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

**Figure A.2:** Velocity Boundary Condition

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel  Newtonian;

nu              nu [ 0 2 -1 0 0 0 0 ] 1.06e-06;


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

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

RASModel        laminar;

turbulence      off;

printCoeffs     off;


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

**Figure A.4:** RASProperties file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     simpleFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         300;

deltaT          1;

writeControl    timeStep;

writeInterval   30;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable false;



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

**Figure A.5:** controlDict file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         steadyState;
}

gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
    grad(U)         Gauss linear;
}

divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         none;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
    interpolate(U)  linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p               ;
}


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

**Figure A.6:** fvSchemes files

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
   p
    {
        solver          GAMG;
        tolerance       1e-07;
        relTol          0.1;
        smoother        GaussSeidel;
        nPreSweeps      0;
        nPostSweeps     2;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 10;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
   U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0.01;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-7;
        U               1e-7;
    }
}

relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
    }
}
cache
{
      grad(U);
}

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

**Figure A.7:** fvSolution file

Appendix B

# scalarTransportFoam Files

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;//For the second simulation, This value change to
//nonuniform list<scalar> with the value of T from the first simulation
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 0;
    }

    outlet
    {
        type            zeroGradient;
    }

   walls
    {
        type            zeroGradient;
    }

    injection-inlet
    {
        type            fixedValue;
        value           uniform 1;//This bondary condition also change to
                        //zeroGradient for the second simulation
    }
}

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

**Figure B.1:** Scalar field Boundary Condition file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

DT              DT [0 2 -1 0 0 0 0] 1e-5;


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

**Figure B.2:** transportProperties file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     scalarTransportFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         1;

deltaT          0.01;

writeControl    timeStep;

writeInterval   10;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable false;

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

**Figure B.3:** controlDict file for the first scalarTransportFoam simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     scalarTransportFoam;

startFrom       startTime;

startTime       1;

stopAt          endTime;

endTime         1.5;

deltaT          0.001;

writeControl    timeStep;

writeInterval   25;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable false;

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

**Figure B.4:** controlDict file for the second scalarTransportFoam simulation

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      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;
}

fluxRequired
{
    default         no;
    T;
}

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

**Figure B.5:** fvSchemes files

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    T
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
}


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

**Figure B.6:** fvSolution file

Appendix C

# Files used for running OpenFoam and snappyHexMesh in parallel using supercomputer

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

numberOfSubdomains 16; // # of CPU cores

method          simple;

simpleCoeffs
{
    n               ( 4 2 2 ); // needs to multiply to = # cores
    delta           0.001;
}

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

manualCoeffs
{
    dataFile        "cellDecomposition";
}


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

**Figure C.1:** decomposeParDict

```
#!/bin/bash -l
#PBS -N abels_job
#PBS -S /bin/bash
#PBS -A ntnu947
#PBS -l select=2:ncpus=32:mpiprocs=16
#PBS -l walltime=01:45:00

cd /home/ntnu/abelm/<casename>

# Load modules
module load mpt/2.11 gcc/4.9.1
module load openfoam/2.4.0
case=<casename>

# Run OpenFOAM
decomposePar -force
mpiexec_mpt -n 16 renumberMesh -overwrite -parallel
mpiexec_mpt -n 16 <solvername> -parallel>logfile
reconstructPar
```

**Figure C.2:** sample job script used to run OpenFoam case on supercomputer with parallel

```
#!/bin/bash -l
#PBS -N abels_job
#PBS -S /bin/bash
#PBS -A ntnu947
#PBS -l select=2:ncpus=32:mpiprocs=16
#PBS -l walltime=00:30:00

cd /home/ntnu/abelm/<casename>

# Load modules
module load mpt/2.11 gcc/4.9.1
module load openfoam/2.4.0
case=<casename>

# Run OpenFOAM
blockMesh
surfaceFeatureExtract
decomposePar -force
mpirun -np 16 snappyHexMesh -overwrite -parallel
reconstructParMesh -constant
```

**Figure C.3:** Job script used to run snappyHexMesh case on supercomputer with parallel

Appendix D

# snappyHexMesh Files

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      surfaceFeatureExtractDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

inlet.stl
{
    extractionMethod    extractFromSurface;//extractFromFile or extractFromSurface
    extractFromSurfaceCoeffs
    {includedAngle   150;}
    writeObj                yes;    // Write options
}

outlet.stl
{
    extractionMethod    extractFromSurface;//extractFromFile or extractFromSurface
    extractFromSurfaceCoeffs
    {includedAngle   150;}
    writeObj                yes;    // Write options
}

walls.stl
{
    extractionMethod    extractFromSurface;//extractFromFile or extractFromSurface
    extractFromSurfaceCoeffs
    {includedAngle   150;}
    writeObj                yes;    // Write options
}


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

**Figure D.1:** surfaceFeatureExtractDict file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      snappyHexMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       true;

// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface
geometry
{
        inlet.stl {type triSurfaceMesh; name inlet;}
        outlet.stl {type triSurfaceMesh; name outlet;}
        walls.stl {type triSurfaceMesh; name walls;}
        volume.stl {type triSurfaceMesh; name volume;}
};

// Settings for the castellatedMesh generation.
castellatedMeshControls
{

    // Refinement parameters
    // ~~~~~~~~~~~~~~~~~~~~~
    // If local number of cells is >= maxLocalCells on any processor
    // switches from refinement followed by balancing
    // (current method) to (weighted) balancing before refinement.
    maxLocalCells 100000;

    // Overall cell limit (approximately). Refinement will stop immediately
    // upon reaching this number so a refinement level might not complete.
    // Note that this is the number of cells before removing the part which
    // is not 'visible' from the keepPoint. The final number of cells might
    // actually be a lot less.
    maxGlobalCells 2000000;

    // The surface refinement loop might spend lots of iterations refining just a
    // few cells. This setting will cause refinement to stop if <= minimumRefine
    // are selected for refinement. Note: it will at least do one iteration
    // (unless the number of cells to refine is 0)
    minRefinementCells 10;

    // Allow a certain level of imbalance during refining
    // (since balancing is quite expensive)
    // Expressed as fraction of perfect balance (= overall number of cells /
    // nProcs). 0=balance always.
    maxLoadUnbalance 0.10;

    // Number of buffer layers between different levels.
    // 1 means normal 2:1 refinement restriction, larger means slower
    // refinement.
    nCellsBetweenLevels 3;
```

```
// Explicit feature edge refinement
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Specifies a level for any cell intersected by its edges.
// This is a featureEdgeMesh, read from constant/triSurface for now.
features
(

    {file "inlet.eMesh"; level 3;}
    {file "outlet.eMesh"; level 3;}
    {file "walls.eMesh"; level 3;}

);

// Surface based refinement
// ~~~~~~~~~~~~~~~~~~~~~~~~
// Specifies two levels for every surface. The first is the minimum level,
// every cell intersecting a surface gets refined up to the minimum level.
// The second level is the maximum level. Cells that 'see' multiple
// intersections where the intersections make an
// angle > resolveFeatureAngle get refined up to the maximum level.

refinementSurfaces
{
    inlet {level (0 0);}
    outlet {level (0 0);}
    walls {level (3 7);}

}

// Resolve sharp angles
resolveFeatureAngle 30;


// Region-wise refinement
// ~~~~~~~~~~~~~~~~~~~~~~

// Specifies refinement level for cells in relation to a surface. One of
// three modes
// - distance. 'levels' specifies per distance to the surface the
//   wanted refinement level. The distances need to be specified in
//   descending order.
// - inside. 'levels' is only one entry and only the level is used. All
//   cells inside the surface get refined up to the level. The surface
//   needs to be closed for this to be possible.
// - outside. Same but cells outside.

refinementRegions
{
    volume
    {
        mode inside;
        levels ((1.0 3));
    }
}


// Mesh selection
// ~~~~~~~~~~~~~~

// After refinement patches get added for all refinementSurfaces and
// all cells intersecting the surfaces get put into these patches. The
// section reachable from the locationInMesh is kept.
// NOTE: This point should never be on a face, always inside a cell, even
// after refinement.
```

```
    locationInMesh (0.018 0.00001 0.00001);

    // Whether any faceZones (as specified in the refinementSurfaces)
    // are only on the boundary of corresponding cellZones or also allow
    // free-standing zone faces. Not used if there are no faceZones.
    allowFreeStandingZoneFaces true;
}


// Settings for the snapping.
snapControls
{
    //- Number of patch smoothing iterations before finding correspondence
    //  to surface
    nSmoothPatch 3;

    //- Relative distance for points to be attracted by surface feature point
    //  or edge. True distance is this factor times local
    //  maximum edge length.
    tolerance 2.0;

    //- Number of mesh displacement relaxation iterations.
    nSolveIter 30;

    //- Maximum number of snapping relaxation iterations. Should stop
    //  before upon reaching a correct mesh.
    nRelaxIter 5;

    // Feature snapping

        //- Number of feature edge snapping iterations.
        //  Leave out altogether to disable.
        nFeatureSnapIter 10;

        //- Detect (geometric only) features by sampling the surface
        //  (default=false).
        implicitFeatureSnap false;

        //- Use castellatedMeshControls::features (default = true)
        explicitFeatureSnap true;

        //- Detect points on multiple surfaces (only for explicitFeatureSnap)
        multiRegionFeatureSnap false;
}


// Settings for the layer addition.
addLayersControls
{
    // Are the thickness parameters below relative to the undistorted
    // size of the refined cell outside layer (true) or absolute sizes (false).
    relativeSizes true;

    // Per final patch (so not geometry!) the layer information
    layers
    {
        walls
        {
            nSurfaceLayers 3;
        }
    }

    // Expansion factor for layer mesh
    expansionRatio 1.0;
```

```
    // Wanted thickness of final added cell layer. If multiple layers
    // is the thickness of the layer furthest away from the wall.
    // Relative to undistorted size of cell outside layer.
    // See relativeSizes parameter.
    finalLayerThickness 0.003;

    // Minimum thickness of cell layer. If for any reason layer
    // cannot be above minThickness do not add layer.
    // Relative to undistorted size of cell outside layer.
    minThickness 0.001;

    // If points get not extruded do nGrow layers of connected faces that are
    // also not grown. This helps convergence of the layer addition process
    // close to features.
    // Note: changed(corrected) w.r.t 17x! (didn't do anything in 17x)
    nGrow 0;

    // Advanced settings

    // When not to extrude surface. 0 is flat surface, 90 is when two faces
    // are perpendicular
    featureAngle 80;

    // At non-patched sides allow mesh to slip if extrusion direction makes
    // angle larger than slipFeatureAngle.
    slipFeatureAngle 30;

    // Maximum number of snapping relaxation iterations. Should stop
    // before upon reaching a correct mesh.
    nRelaxIter 3;

    // Number of smoothing iterations of surface normals
    nSmoothSurfaceNormals 1;

    // Number of smoothing iterations of interior mesh movement direction
    nSmoothNormals 3;

    // Smooth layer thickness over surface patches
    nSmoothThickness 10;

    // Stop layer growth on highly warped cells
    maxFaceThicknessRatio 0.5;

    // Reduce layer growth where ratio thickness to medial
    // distance is large
    maxThicknessToMedialRatio 0.3;

    // Angle used to pick up medial axis points
    // Note: changed(corrected) w.r.t 17x! 90 degrees corresponds to 130 in 17x.
    minMedianAxisAngle 90;


    // Create buffer region for new layer terminations
    nBufferCellsNoExtrude 0;


    // Overall max number of layer addition iterations. The mesher will exit
    // if it reaches this number of iterations; possibly with an illegal
    // mesh.
    nLayerIter 50;
}
```

```
// Generic mesh quality settings. At any undoable phase these determine
// where to undo.
meshQualityControls
{
    maxNonOrtho 65;
    maxBoundarySkewness 20;
    maxInternalSkewness 4;
    maxConcave 80;
    minFlatness 0.5;
    minVol 1e-13;
    minTetQuality 1e-9;
    minArea -1;
    minTwist 0.02;
    minDeterminant 0.001;
    minFaceWeight 0.02;
    minVolRatio 0.01;
    minTriangleTwist -1;

    // Advanced

    //- Number of error distribution iterations
    nSmoothScale 4;
    //- Amount to scale back displacement at error points
    errorReduction 0.75;
}

// Advanced

debug 0;


// Merge tolerance. Is fraction of overall bounding box of initial mesh.
// Note: the write tolerance needs to be higher than this.
mergeTolerance 1e-6;


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

**Figure D.2:** snappyHexMeshDict file

Appendix E

# Quantitative results of simulation and experiment

| Pump speed | Velocity | Reynolds | relative Pressure Drop ($m^2/s^2$) | | | | |
|---|---|---|---|---|---|---|---|
| | | | Experiment Result | Simulation result | | | |
| | | | | Hexadron Mesh | | Tetrahedron Mesh | |
| | | | | old sensor | modified sensor | old sensor | modified sensor |
| 4000 | 0.488 | 1933.59 | 6.67 | 2.998 | 5.09 | 4.09 | 6.35 |
| 3600 | 0.44 | 1743.4 | 5.49 | 2.57 | 4.31 | 3.51 | 5.39 |
| 3200 | 0.392 | 1553.2 | 4.44 | 2.17 | 3.59 | 2.96 | 4.49 |
| 2800 | 0.34 | 1347.17 | 3.38 | 1.77 | 2.88 | 2.4 | 3.59 |
| 2400 | 0.29 | 1149.06 | 2.28 | 1.41 | 2.25 | 1.91 | 2.81 |
| 2000 | 0.247 | 978.68 | 1.81 | 1.13 | 1.77 | 1.51 | 2.2 |
| 1600 | 0.196 | 776.6 | 1.08 | 0.82 | 1.26 | 1.09 | 1.55 |
| 1400 | 0.172 | 681.51 | 0.95 | 0.69 | 1.04 | 0.9 | 1.28 |
| 1200 | 0.146 | 578.49 | 0.83 | 0.56 | 0.83 | 0.72 | 1.01 |
| 1000 | 0.122 | 483.4 | 0.75 | 0.44 | 0.65 | 0.56 | 0.78 |
| 800 | 0.097 | 384.34 | 0.59 | 0.33 | 0.47 | 0.41 | 0.57 |

**Table E.1:** summery of all results of simulations and experiment from eleven pump flow rates, pump speed in **rpm** and velocity in **m/s**

Appendix F

# Modified scalarTransportFoam files

```
Info<< "Reading transportProperties\n" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);


Info<< "Reading diffusivity DT\n" << endl;

dimensionedScalar DT
(
    transportProperties.lookup("DT")
);

Info<< "Reading limiting molar conductivity Am\n" << endl;

dimensionedScalar Am
(
    transportProperties.lookup("Am")
);

 Info<< "Reading emperical constant K\n" << endl;

dimensionedScalar K
(
    transportProperties.lookup("K")
);

Info<< "Reading field T\n" << endl;

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field U\n" << endl;

volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);


Info<< "calculating field A\n" << endl;

    volScalarField A
    (
         IOobject
         (
             "A",
             runTime.timeName(),
             mesh,
             IOobject::NO_READ,
             IOobject::AUTO_WRITE
         ),
        | Am.value()-K.value()*sqrt(T)  // This is equation of the conductivity of a solution
     );                                 // for a strong electrolyte at low concentration


    #include "createPhi.H"
```

**Figure F.1:** createFields.H file

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2015 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    scalarTransportFoam

Description
    Solves a transport equation for a passive scalar

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "fvIOoptionList.H"
#include "simpleControl.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#include "fvCFD.H"
#include "fvIOoptionList.H"
#include "simpleControl.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"

    simpleControl simple(mesh);

    #include "createFields.H"
    #include "createFvOptions.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nCalculating scalar transport\n" << endl;

    #include "CourantNo.H"

  while (simple.loop())
  {
      Info<< "Time = " << runTime.timeName() << nl << endl;

      while (simple.correctNonOrthogonal())
      {
          solve
          (
              fvm::ddt(T)
            + fvm::div(phi, T)
            - fvm::laplacian(DT, T)
           ==
              fvOptions(T)
          );
           Am.value()-K.value()*sqrt(T);
      }

      runTime.write();
  }

  Info<< "End\n" << endl;

  return 0;
}


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

**Figure F.2:** scalarTransportFoam.C file