POLITECNICO DI TORINO

Department of Mechanical, Aerospace, Automotive and Production Engineering

Master of Science in Automotive Engineering

Master Degree Thesis

3D CFD-Simulation of a working cycle of the ECN test-bed engine using OpenFOAM

Professor:
Prof. Ezio Spessa

Supervisor:
Dr. rer. nat. Dietmar Schmidt

Candidate
Vincenzo Damiano Virgilio
Student ID: 254242

Academic year 2019-2020

Abstract

Throughout the history of the IC engine, and decades before climate change concerns became prominent, researchers have striven to improve its fuel efficiency, to reduce pollutant emissions and operating costs and to ensure the optimal use of finite fuel resources for current and future generations. Over the last four decades, in response to air-quality concerns, research on engine combustion, exhaust after-treatment and controls has led to a demonstrably cleaner environment.
It is likely that future mobility will be characterized by a mix of solutions, involving battery electric and hybrid electric vehicles (BEV and HEV), fuel cell electric vehicles (FCEVs) and conventional vehicles, depending on consumer acceptance (e.g. cost), the country considered and the specific application (city, country, personal, freight, etc.). Thus, the combustion engine will still play a central role, whether used for power generation or for powering the vehicle itself, even in strongly electrified powertrain configurations. Because of this, there is great interest in improving the thermal efficiency of IC engines without significant increases in purchase and operating costs in the short-to-medium term.
One of the most powerful tools in the investigation of new solutions is the 3D-CFD simulation, which can provide a very precise and reliable reproduction of thermo-fluid-dymanical phenomena inside the engine, with consequent savings in time and financial resources.
Many companies develop software for CFD analysis for commercial use. Each of them differs from the other in cost, performance, interface and implementation of mathematical models.
However there is a free, open source CFD software called OpenFOAM. It has an extensive range of features to solve anything from complex fluid flows involving chemical reactions, turbulence and heat transfer, to acoustics, solid mechanics and electromagnetics.
The purpose of this thesis is to analyze the possible advantages of using an open source program like OpenFoam in the study of ICE. The most important step is to validate the reliability of the calculations obtained from the simulations. For this purpose, an experimental engine has been analyzed, whose data are present in the ECN (Engine Combustion Network) database.

The study will cover all the step, starting from the simple geometry, the creation of the mesh, the simulation and finally the results.

Acknowledgements

This work is the results of my activity at Institut für Verbrennungsmotoren und Kraftfahrwesen Lehrstuhl Fahrzeugantriebe (IVK), an institute of the Universität Stuttgart dedicated to the Research on Internal Combustion Engines.
I would like to thank Dr. rer. nat. Dietmar Schmidt for his patience and help during the work and also to the other thesis students for their helpfulness. At the IVK I learned how to overcome problems and help one another. I will carry this experience with me into my future personal career.
I would also like to thank Prof. Ezio Spessa for meeting my work and university needs and being available when I needed him.

Vorrei ancora una volta alla fine di quest altro ciclo di studi ringraziare i miei genitori, i quali mi hanno fornito l'appoggio e la motivazione necessaria per conseguire questo titolo. Il raggiungimento di questo risultato non é solamente frutto della mia dedizione ma anche dei sacrifici di mio padre Aldo e mia madre Marika.
Un grazie anche a tutti gli amici di una vita, i compagni di studi e i compagni di avventura gli che sono rimasti al mio fianco e grazie ai quali sono riuscito a raggiungere questo obiettivo.

Torino, 2020

Vincenzo Damiano Virgilio

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computation fluid dynamics (CFD) is an engineering tool used to simulate the action of thermo-fluids in a system. It is used by many industries in their development work to analyze, optimize and verify the performance of designs before costly prototypes and physical tests. The application of CFD is rapidly expanding with the growth in affordability of computational resources. It is becoming essential for CFD solvers to provide validation and verification. Mesh related issues play a very important role on accuracy and convergence. The means to achieve high fidelity computational simulations of fluid dynamic phenomena is analyzed by considering the various constituent parts of the simulation hierarchy including the mathematical model of the physics, the numerical model, the computational model (including the mesh), and most importantly the human in the loop.

1.1 CFD principles

Computational Fluid-Dynamics (CFD) is the development of methods and algorithms that allow you to simulate the behavior of fluids in very complex phenomena. This simulation consists in the numerical elaboration of complicated mathematical models with the aim of studying the temporal evolution of the fluid through its main fluid dynamic parameters: pressure, velocity, temperature and density. The use of numerical processing in the design phase is essential in those situations where it is necessary to study and analyze a large number of cases. For time-accurate simulations, the initial conditions should closely correspond to what would occur in nature.
The multi-dimensional models aim to solve the equations of conservation of: mass, momentum, energy and chemical species according to the spatial dimensions of the system and time, with the aim of predicting in detail (local and temporal) the fields of motion of the fluid in the cylinder and the processes of combustion and heat

exchange, which depend on those. The fluid-dynamic processes that take place inside the cylinder and through the intake and exhaust ports are typically three-dimensional. Introducing three spatial coordinates, next to the temporal one, there is a strong increase in the complexity of the model and in the necessary calculation times.
Equations and notions of CFD calculation in detail are not the subject of this thesis. Nevertheless, a summary of the theoretical basis will be presented below [8].
The governing equations of fluid flow represent mathematical statements of the **conservation laws of physics**:

- The mass of a fluid is conserved
- The rate of change of momentum equals the sum of the forces on a fluid particle (Newton's second law)
- The rate of change of energy is equal to the sum of the rate of heat addition to and the rate of work done on a fluid particle (first law of thermodynamics)

The fluid will be regarded as a continuum. For the analysis of fluid flows at macroscopic length scales (say $1\mu m$ and larger) the molecular structure of matter and molecular motions may be ignored. We describe the behavior of the fluid in terms of macroscopic properties, such as velocity, pressure, density and temperature, and their space and time derivatives. These may be thought of as averages over suitably large numbers of molecules. A fluid particle or point in a fluid is then the smallest possible element of fluid whose macroscopic properties are not influenced by individual molecules.

1.1.1 Mass conservation in three dimensions

The first step in the derivation of the mass conservation equation is to write down a mass balance for the fluid element:
Rate of increase of mass in fluid element = Net rate of flow of mass into fluid element

$$\frac{\partial \rho}{\partial t} + div(\rho u) = 0 \tag{1.1}$$

Equation 1.1 is the **unsteady, three-dimensional mass conservation or continuity equation at a point in a compressible fluid**. The first term on the left hand side is the rate of change in time of the density (mass per unit volume). The second term describes the net flow of mass out of the element across its boundaries and is called the convective term.
For an **incompressible fluid** (i.e. a liquid) the density ρ is constant and equation 1.1 becomes:

$$div(u) = 0 \tag{1.2}$$

or in longhand notation:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial y} = 0 \tag{1.3}$$

1.1.2 Momentum equation in three dimensions

Newton's second law states that the rate of change of momentum of a fluid particle equals the sum of the forces on the particle:
Rate of increase of momentum of fluid particle = Sum of forces on fluid particle
The **rates of increase of x-, y- and z-momentum** per unit volume of a fluid particle are given by:

$$\rho\frac{Du}{Dt}, \rho\frac{Dv}{Dt}, \rho\frac{Dw}{Dt} \tag{1.4}$$

We distinguish two types of **forces** on fluid particles:

- surface forces
  - pressure forces
  - viscous forces
  - gravity force
- body forces
  - centrifugal force
  - Coriolis force
  - electromagnetic force

It is common practice to highlight the contributions due to the surface forces as separate terms in the momentum equation and to include the effects of body forces as source terms. The pressure, a normal stress, is denoted by p. Viscous stresses are denoted by τ. The usual suffix notation τ_{ij} is applied to indicate the direction of the viscous stresses. The suffices i and j in τ_{ij} indicate that the stress component acts in the j- direction on a surface normal to the i-direction.
The total force per unit volume on the fluid due to the surface stresses is equal to the sum of the net force in the x-direction divided by the volume $\delta x \delta y \delta z$:

$$\frac{\partial(-p + \tau_{xx})}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} \tag{1.5}$$

Without considering the body forces in further detail their overall effect can be included by defining a source S_{M_x} of x-momentum per unit volume per unit time. The **x-component of the momentum equation** is found by setting the rate of change of x-momentum of the fluid particle 1.4 equal to the total force in the x-direction on the element due to surface stresses 1.5 plus the rate of increase of x-momentum due to sources:

$$\rho\frac{Du}{Dt} = \frac{\partial(-p+\tau_{xx})}{\partial x} + \frac{\partial\tau_{yx}}{\partial y} + \frac{\partial\tau_{zx}}{\partial z} + S_{M_x} \tag{1.6}$$

It is not too difficult to verify that the textbf$y$-component of the momentum equation is given by

$$\rho\frac{D\upsilon}{Dt} = \frac{\partial\tau_{xy}}{\partial x} + \frac{\partial(-p+\tau_{yy})}{\partial y} + \frac{\partial\tau_{zy}}{\partial z} + S_{M_y} \tag{1.7}$$

and the **z-component of the momentum equation** by

$$\rho\frac{Dw}{Dt} = \frac{\partial\tau_{xz}}{\partial x} + \frac{\partial\tau_{yz}}{\partial y} + \frac{\partial(-p+\tau_{zz})}{\partial z} + S_{M_z} \tag{1.8}$$

The sign associated with the pressure is opposite to that associated with the normal viscous stress, because the usual sign convention takes a tensile stress to be the positive normal stress so that the pressure, which is by definition a compressive normal stress, has a minus sign.
The effects of surface stresses are accounted for explicitly; the source terms S_{M_x}, S_{M_y} and S_{M_z} include contributions due to body forces only. For example, the body force due to gravity would be modeled by $S_{M_x} = 0$, $S_{M_y} = 0$ and$S_{M_z} = \rho g$.

1.1.3 Energy equation in three dimensions

The energy equation is derived from the **first law of thermodynamics**, which states that the rate of change of energy of a fluid particle is equal to the rate of heat addition to the fluid particle plus the rate of work done on the particle:
Rate of increase of energy of fluid particle = Net rate of heat added to fluid particle + Net rate of work done on fluid particle

As before, we will be deriving an equation for the **rate of increase of energy** of a fluid particle per unit volume, which is given by

$$\rho\frac{DE}{Dt} \tag{1.9}$$

Thus far we have not defined the specific energy E of a fluid. Often the energy of a fluid is defined as the sum of internal (thermal) energy i, kinetic energy $\frac{1}{2}(u^2+\upsilon^2+w^2)$ and gravitational potential energy. This definition takes the view that the fluid

element is storing gravitational potential energy. It is also possible to regard the gravitational force as a body force, which does work on the fluid element as it moves through the gravity field.
Here we will take the latter view and include the effects of potential energy changes as a source term. As before, we define a source of energy S_E per unit volume per unit time. Conservation of energy of the fluid particle is ensured by equating the rate of change of energy of the fluid particle 1.9 to the sum of the net rate of work done on the fluid particle, the net rate of heat addition to the fluid $-div(q) = div(kgradT)$ and the rate of increase of energy due to sources. If we use the Newtonian model for viscous stresses in the internal energy equation we obtain after some rearrangement :

$$\rho\frac{Di}{Dt} = -pdiv(u) + div(kgradT) + \phi + S_i \tag{1.10}$$

Where ϕ is the dissipation function.

1.1.4 Navier-Stokes equations for a Newtonian fluid

The governing equations contain as further unknowns the viscous stress components τ_{ij}. The most useful forms of the conservation equations for fluid flows are obtained by introducing a suitable model for the viscous stresses τ_{ij}. In many fluid flows the viscous stresses can be expressed as functions of the local deformation rate or strain rate. In three-dimensional flows the local rate of deformation is composed of the linear deformation rate and the volumetric deformation rate.
All gases and many liquids are isotropic. Liquids that contain significant quantities of polymer molecules may exhibit anisotropic or directional viscous stress properties as a result of the alignment of the chain-like polymer molecules with the flow. Such fluids are beyond the scope of this introductory course and we shall continue the development by assuming that the fluids are isotropic.
In a **Newtonian fluid the viscous stresses are proportional to the rates of deformation**. The three-dimensional form of Newton's law of viscosity for compressible flows involves two constants of proportionality: the first (dynamic) viscosity, μ, to relate stresses to linear deformations, and the second viscosity, λ, to relate stresses to the volumetric deformation. The nine viscous stress components, of which six are independent, are :

$$\tau_{xx} = 2\mu\frac{\partial u}{\partial x} + \lambda div(u), \tau_{yy} = 2\mu\frac{\partial v}{\partial y} + \lambda div(u), \tau_{zz} = 2\mu\frac{\partial w}{\partial z} + \lambda div(u) \tag{1.11}$$

$$\tau_{xy} = \tau_{yx} = \mu(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}), \tau_{xz} = \tau_{zx} = \mu(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}), \tau_{yz} = \tau_{zy} = \mu(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}) \tag{1.12}$$

Not much is known about the second viscosity λ, because its effect is small in practice. For gases a good working approximation can be obtained by taking the value $\lambda = -\frac{2}{3}\mu$ (Schlichting, 1979). Liquids are incompressible so the mass conservation equation is div $u = 0$ and the viscous stresses are just twice the local rate of linear deformation times the dynamic viscosity. Substitution of the above shear stresses 1.11 and 1.12 into 1.6 yields the so-called Navier–Stokes equations, named after the two nineteenth-century scientists who derived them independently.
The viscous stresses in the y- and z-component equations can be recast in a similar manner. We clearly intend to simplify the momentum equations by 'hiding' the bracketed smaller contributions to the viscous stress terms in the momentum source. The Navier–Stokes equations can be written in the most useful form for the development of the finite volume method:

$$\rho\frac{Du}{Dt} = -\frac{\partial(p)}{\partial x} + div(\mu grad(u) + S_{M_x} \tag{1.13}$$

$$\rho\frac{Dv}{Dt} = -\frac{\partial(p)}{\partial y} + div(\mu grad(v) + S_{M_y} \tag{1.14}$$

$$\rho\frac{Dw}{Dt} = -\frac{\partial(p)}{\partial z} + div(\mu grad(w) + S_{M_z} \tag{1.15}$$

1.2 Turbulence models

For turbulent flows, the choice of turbulence models is a primary choice to be made, based on the simulation needs. If a turbulence closure can be chosen just based on its suitability for modeling certain physical phenomena, it will be a very good situation indeed. However, often the practitioner may weigh in the robustness of the model in choosing one. The robustness is strongly tied to both the model itself and its numerical implementation. Depending on what model is chosen, the corresponding variables will have to be defined.
Turbulence causes the appearance in the flow of eddies with a wide range of length and time scales that interact in a dynamically complex way. Given the importance of the avoidance or promotion of turbulence in engineering applications, it is no surprise that a substantial amount of research effort is dedicated to the development of numerical methods to capture the important effects due to turbulence [8].

1.2.1 Direct Numerical Simulations (DNS)

The direct simulation DNS (acronym of the term Direct Numerical Simulations) requires that the description of the phenomenon, (behavior of a fluid) occurs through the complete numerical resolution of the Navier-Stokes equations, with all the equations necessary to close the problem and without the need to introduce sub-models of turbulence or hypotheses. In order to describe the dynamics of the turbulent flow up to the smallest space-time scales, it is necessary that the calculation domain and the time length of the phenomenon are divided into very small intervals. This is due to the fact that in a turbulent motion the diffusion and dissipation of mechanical energy are entrusted to microscopic structures with very short characteristic times, independent of the spatial and temporal scales of the average motion.
All this translates into the need for very high potential of computers to carry out direct simulation. The DNS also requires very small calculation grids (in the order of the Kolmogorov length scales: 0.01-0.05 mm) and currently this type of simulation can be performed with very low Reynolds numbers, to have calculation times that, while remaining long, are dignified. If we raise the number of Reynolds the times of calculation grow exponentially and become unsustainable, as well as the costs. For these reasons, direct simulation is confined to low Reynolds numbers and simple geometries. This is therefore not applied on an industrial level, also because industrial applications do not require such a high level of precision. The use of direct simulation for existing applications in reality, and not only at the research level, remains a long-term goal.

1.2.2 The LES (Large Eddy Simulation)

The LES (Large Eddy Simulation) simulation technique is an intermediate level between direct simulation and RANS modeling. It consists in directly simulating the larger vortex structures, through a grid that is sufficiently dense while the smaller structures are modeled. The larger structures have a convective nature at relatively high Reynolds numbers and scales comparable to those of the medium motion. They have origin and typology deriving from those of the medium motion, from which they extract energy to produce turbulent kinetic energy, are linked to the geometry and are generally anisotropic.
It is therefore necessary to simulate directly the vortices of large and medium scale and model those of smaller scales. In this way a universal methodology is built to deal with the problem, because the modeled structures are independent of the geometry. The derivation of the LES equations is similar to that of the equations of the RANS modeling with the difference that for the LES simulation with respect to the RANS one must carry out an operation of spatial filtering instead of temporal. An example of this last operation comes made in the phase of generation of the grate, in which are generated of the grids of calculation that are not in a position to simulate the smallest vortexsities, in how much too great, because in the contrary case a direct simulation would be carried out.

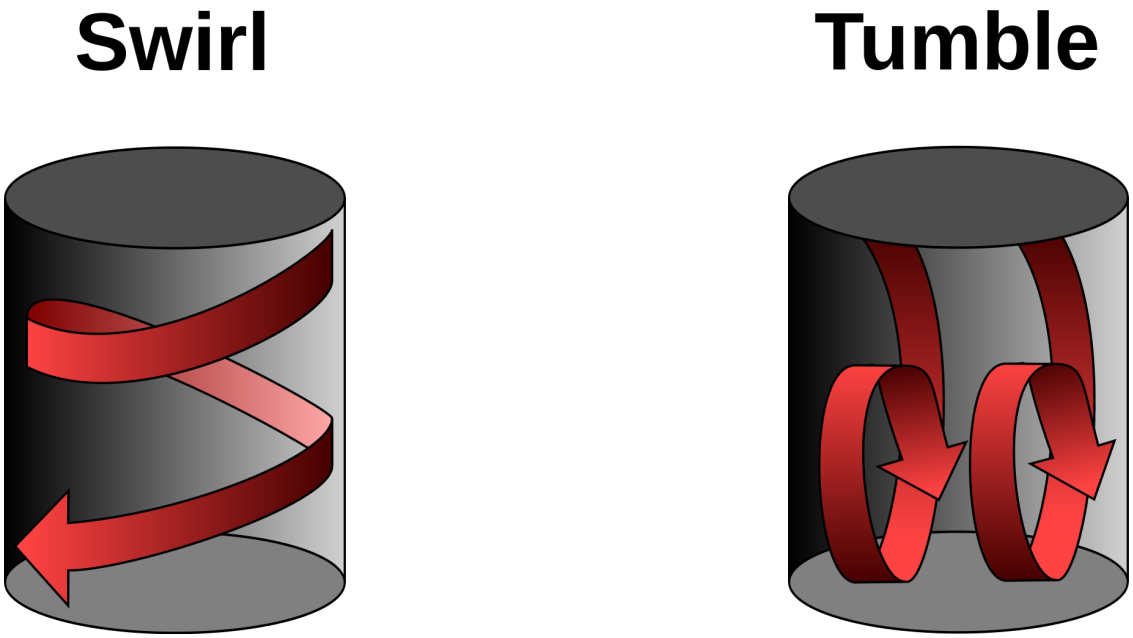
1.2.3 The Reynolds-averaged Navier–Stokes equations (RANS equations)

Attention is focused on the mean flow and the effects of turbulence on mean flow properties. Prior to the application of numerical methods the Navier–Stokes equations are time averaged (or ensemble averaged in flows with time-dependent boundary conditions). Extra terms appear in the time-averaged (or Reynolds- averaged) flow equations due to the interactions between various turbulent fluctuations. These extra terms are modeled with classical turbulence models: among the best known ones are the k–σ model and the Reynolds stress model. The computing resources required for reasonably accurate flow computations are modest, so this approach has been the mainstay of engineering flow calculations over the last three decades.
For most engineering purposes it is unnecessary to resolve the details of the turbulent fluctuations. CFD users are almost always satisfied with information about the time-averaged properties of the flow (e.g. mean velocities, mean pressures, mean stresses etc.). Therefore, the vast majority of turbulent flow computations has been and for the foreseeable future will continue to be carried out with procedures based on the Reynolds-averaged Navier–Stokes (RANS) equations.
It has become clear from our discussions of simple turbulent flows that turbulence

levels and turbulent stresses vary from point to point in a flow. Mixing length models attempt to describe the stresses by means of simple algebraic formulae for μ_t as a function of position. The k–σ model is a more sophisticated and general, but also more costly, description of turbulence which allows for the effects of transport of turbulence properties by convection and diffusion and for production and destruction of turbulence.
Two transport equations (PDEs), one for the turbulent kinetic energy k and a further one for the rate of dissipation of turbulent kinetic energy ϵ, are solved.
The underlying assumption of both these models is that the turbulent viscosity μ_t is isotropic: in other words that the ratio between Reynolds stress and mean rate of deformation is the same in all directions. This assumption fails in many complex flows where it leads to inaccurate predictions. Here it is necessary to derive and solve transport equations for the Reynolds stresses themselves. It may at first seem strange to think that a stress can be subject to transport. However, it is only necessary to remember that the Reynolds stresses initially appeared on the left hand side of the momentum equations and are physically due to convective momentum exchanges as a consequence of turbulent velocity fluctuations. Fluid momentum – mean momentum as well as fluctuating momentum – can be transported by fluid particles and therefore the Reynolds stresses can also be transported.
The six transport equations, one for each Reynolds stress, contain diffusion, pressure–strain and dissipation terms whose individual effects are unknown and cannot be measured. In Reynolds stress equation models (also known in the literature as second-order or second-moment closure models) assumptions are made about these unknown terms, and the resulting PDEs are solved in conjunction with the transport equation for the rate of dissipation of turbulent kinetic energy ϵ. The design of Reynolds stress equation models is an area of vigorous research, and the models have not been validated as widely as the mixing length and k–σ model. Solving the seven extra PDEs gives rise to a substantial increase in the cost of CFD simulations when compared with the k–σ model, so the application of Reynolds stress equation models outside the academic fraternity is relatively recent.
A much more far-reaching set of modeling assumptions reduces the PDEs describing Reynolds stress transport to algebraic equations to be solved alongside the k and σ equations of the k–σ model. This approach leads to the algebraic stress models that are the most economical form of Reynolds stress model able to introduce anisotropic turbulence effects into CFD simulations [8].

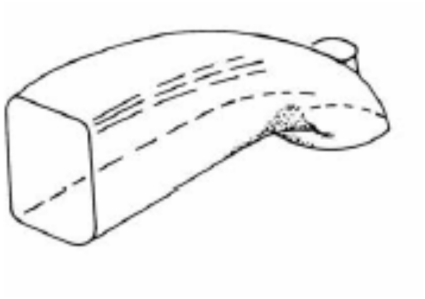
1.3 Charge motion within the cylinder

Gas motion within the engine cylinder is one of the major factors that controls the combustion process in SI engines and the fuel-air mixing and combustion processes in diesel engines. It also has a significant impact on heat transfer. In 2stroke engines the flow details in inlet and exhaust port as well as in the cylinder also govern the scavenging process. The initial in-cylinder flow pattern is set up by the intake process. It may then be substantially modified during compression.

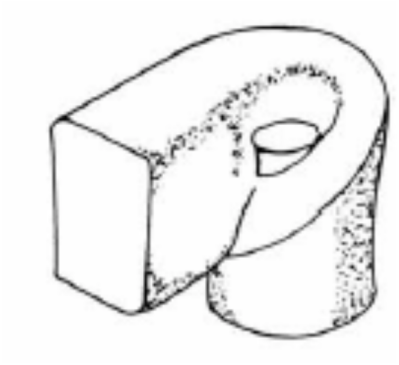


1.3.1 Swirl

The swirl vortex is an organized rotation of the charge about the cylinder axis.
Swirl is created during the induction stroke, by bringing the intake flow into the cylinder with an initial angular momentum, for instance by means of either 'directed ducts' (1.2a) or 'helical ports' (1.2b).
The directed port brings the flow toward the valve opening in the desired tangential direction. Its passage is straight, which due to other cylinder head requirements restricts the flow area and results in a relatively low discharge coefficient.
The helical port uses the port inner side wall to force the flow preferentially through the outer periphery of the valve opening, in a tangential direction. Since only one wall is used to obtain a directional effect, the port areas are less restrictive.



(a) Directed ducts



(b) Helical ports

While some decay in swirl due to friction occurs during engine cycle, intake generated swirl usually persists through the compression, combustion, and expansion processes. In engine designs with bowl-in-piston combustion chambers, the rotational motion set up during intake is substantially amplified during compression. Swirl is used in diesels and some stratified-charge engine concepts to promote more rapid mixing between the inducted air charge and the injected fuel. Swirl is also used to speed up the combustion process in SI engines and to improve scavenging in 2stroke engines.
The angular momentum of the air, which enters the cylinder at each crank angle during induction, decays throughout the rest of the intake process and during the compression process due to friction at the walls and turbulent dissipation within the fluid. Typically one-quarter to one-third of the initial moment of momentum about the cylinder axis will be lost by top-center at the end of compression. However, swirl velocities in the charge can be substantially increased during compression by suitable design of the combustion chamber. In many designs of direct-injection diesel combustion systems, air swirl is used to obtain much more rapid mixing between the fuel injected into the cylinder and the air than would occur in the absence of swirl. The tangential velocity of the swirling airflow set up inside the cylinder during induction is substantially increased by forcing most of the air into a compact bowl-in-piston combustion chamber, usually centered on the cylinder axis, as the piston approaches its top-center position. Neglecting the effects of friction, angular momentum is conserved: so as the moment of inertia of the air is decreased, its angular velocity must increase (however, the total angular momentum of the charge within the cylinder decays due to friction at the chamber walls).

1.3.2 Tumble

Tumble is an organized vortex about an axis that is perpendicular to the cylinder one. It is used mainly in SI engines in order to enhance turbulence prior to the spark discharge.
In fact, tumble is generated during expansion and, provided that it is strong enough, it is 'accelerated' during the first part of the compression stroke. Tumble is then destroyed near TDC, and converted into turbulence energy at small scales.

Chapter 2

OpenFOAM

In this section will be introduced the software used for this simulation, the difference with commercial software listing strengths and weaknesses. Understanding the structure and operation of the program is essential in order to successfully complete a reliable simulation.

Figure 2.1: OpenFOAM logo

OpenFOAM (for "Open-source Field Operation And Manipulation") is a C++ toolbox for the development of customized numerical solvers, and pre-/post-processing utilities for the solution of continuum mechanics problems, most prominently including computational fluid dynamics (CFD).
OpenFOAM is the leading free, open source software for computational fluid dynamics (CFD), owned by the OpenFOAM Foundation and distributed exclusively under the General Public Licence (GPL). The GPL gives users the freedom to modify and redistribute the software and a guarantee of continued free use, within the terms of the license.
OpenFOAM is developed and maintained by individuals who contribute their work to the project, with the support and consent of the companies that employ them. The project operates through a network of trust between the individuals, where greater authority is given to contributors who consistently produce high quality work and demonstrate long term commitment [1].

In the list below I have collected some advantages and disadvantages that I believe are relevant in the choice of a CFD software [2].

Advantages:

- Friendly syntax for partial differential equations
- Fully documented source code
- Unstructured polyhedral grid capabilities
- Automatic parallelization of applications written using OpenFOAM high-level syntax
- Wide range of applications and models ready to use
- Commercial support and training provided by the developers
- No license costs

Disadvantages:

- The development community suffers from fragmentation, giving rise to numerous forked projects.
- Absence of an integrated graphical user interface (stand-alone open-source and proprietary options are available)
- The Programmer's guide does not provide sufficient details, making the progress slow if you need to write new applications or add functionality

2.1 Get Started

A CFD simulation usually consists of the steps of pre-processing, flow field computation and post-processing. The pre-processing includes above all the preparation, meshing and definition of the flow area of the CFD model. In the next step, the calculation is carried out with the previously defined boundary conditions. In post-processing, data can be evaluated through sampling or visually evaluated through a third-party program (ParaView).

2.1.1 Pre-processing

Often the starting point for a CFD simulation is the creation of geometry using CAD software. Having a defect-free geometry is the basis for avoiding numerous errors during the Mesh phase and consequently the subsequent phases of the simulation. If the geometry is supplied by a third party or downloaded from the Internet, it is necessary to verify it. In this case I used a popular open source program: Blender. To create the Mesh I used an OpenFOAM utility: snappyHexMesh.
The preparation of the geometry and the Mesh will be explained in detail in a dedicated chapter because often from them depends the reliability of the results.

2.1.2 Case structure

OpenFOAM cases are configured using several plain text input files located across the three directories [3]:

1. **system**
    - controlDict
    - fvSchemes
    - fvSolution
    - fvOptions (optional)
    - <system dictionaries>
2. **constant**
    - polyMesh
    - <constant dictionaries>
3. **<initial time directory>**
    - <field files>
4. **Additional directories** can be generated, depending on user selections. These may include:
    - <result time directories>: field predictions as a function of iteration count or time
    - postProcessing: data typically generated by function objects
    - data conversion, e.g. VTK

2.1.3 snappyHexMesh

This section describes the mesh generation utility, *snappyHexMesh*, supplied with OpenFOAM. The *snappyHexMesh* utility generates 3-dimensional meshes containing hexahedra (hex) and split-hexahedra (split-hex) automatically from triangulated surface geometries, or tri-surfaces, in Stereolithography (STL) or Wavefront Object (OBJ) format. The mesh approximately conforms to the surface by iteratively refining a starting mesh and morphing the resulting split-hex mesh to the surface. An optional phase will shrink back the resulting mesh and insert cell layers. The specification of mesh refinement level is very flexible and the surface handling is robust with a pre-specified final mesh quality. It runs in parallel with a load balancing step every iteration [4].

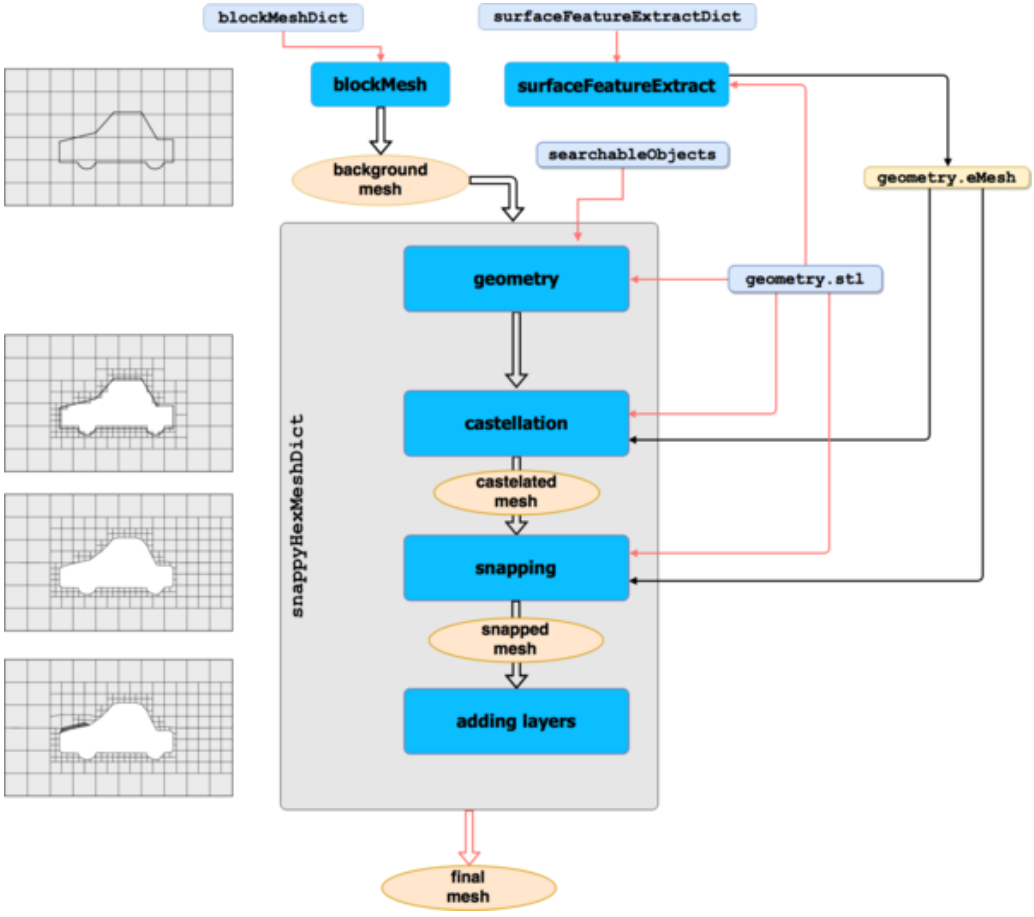


Figure 2.2: snappyHexMesh workflow

This utility includes:

- creation of the background mesh using the *blockMesh* utility (or any other hexahedral mesh generator). The following criteria must be observed:
  1. The mesh must consist purely of hexes.
  2. The cell aspect ratio should be approximately 1, at least near the STL surface.
  3. There must be at least one intersection of a cell edge with the STL surface
- extraction of features on the surfaces with *surfaceFeatureExtract* utility from the geometry files in STL format.
  1. The STL geometry can be obtained from any geometry modeling tool.
  2. The STL file can be made up of a single surface describing the geometry, or multiple surfaces that describe the geometry.
  3. In the case of a STL file with multiple surfaces, we can use local refinement in each individual surface. This gives us more control when generating the mesh.
  4. The STL geometry is always located in the directory *constant/triSurface*
- setting up the snappyHexMeshDict input dictionary
- running *snappyHexMesh* in serial or parallel

Meshing controls are set in the snappyHexMeshDict located in the case system directory. This has five main sections, described by the following:

- geometry: specification of the input surfaces
- castellatedMeshControls: starting from any pure hex mesh, refine and optionally load balance when running in parallel. The refinement is specified both according to surfaces, volumes and gaps
- snapControls: guaranteed mesh quality whilst morphing to geometric surfaces and features
- addLayersControls: prismatic layers are inserted by shrinking an existing mesh and creating an infill, subject to the same mesh quality constraints
- meshQualityControls: mesh quality settings enforced during the snapping and layer addition phases
- Global settings

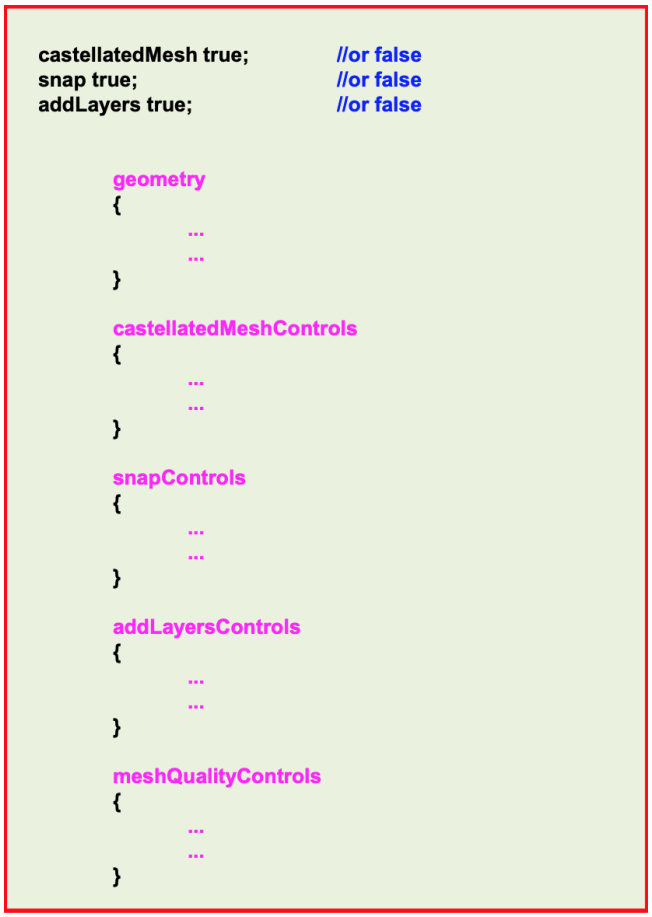


Figure 2.3: snappyHexMeshDict layout

2.2 Solvers

OpenFOAM has suitable solvers for different flows and since OpenFOAM is an open source software, you can also create your own solvers and extensions. The LibICE is such an extension, which was developed by the Polytechnic of Milan (Politecnico di Milano), for the improved network movement with the simulation of combustion engines. Above all, this extension makes it possible to perform a valve movement during the calculations. This is necessary to be able to simulate a complete working cycle including charge change. Note that this extension is compatible with the OpenFOAM versions 2.2.2 and 2.2.x. Thus the solver ivkEDCsprayEngineFoam is also a solver developed at the Institut für Verbrennungsmotoren und Kraftfahrwesen Lehrstuhl Fahrzeugantriebe (IVK) of the University of Stuttgart, which was developed as part of a final thesis. This implements the Solver SprayFoam in LibICE and enables the simulation of direct injection engines.

Chapter 3

Pre-processing

In this chapter will be listed all the necessary steps of the preparation for the actual calculation: the pre-processing. This includes the creation of the geometry, the creation of the different meshes necessary for the calculation and the definition of the boundary conditions. The creation of the Meshes in particular is directly connected to the stability of the calculation, to the time of the simulation and to the precision of the results. The ECN test-bed engine will be examined in order to reproduce the experimental results.

3.1 Geometry

3.1.1 The ECN test-bed engine

The ECN (Engine Combustion Network) is an international collaboration among experimental and computational researchers in engine combustion [5]. The objectives of this network are:

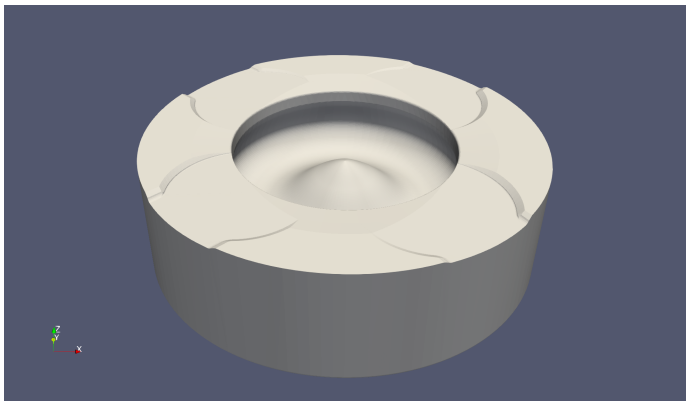
1. Establish an internet library of well-documented experiments that are appropriate for model validation and the advancement of scientific understanding of combustion at conditions specific to engines.

2. Provide a framework for collaborative comparisons of measured and modeled results.

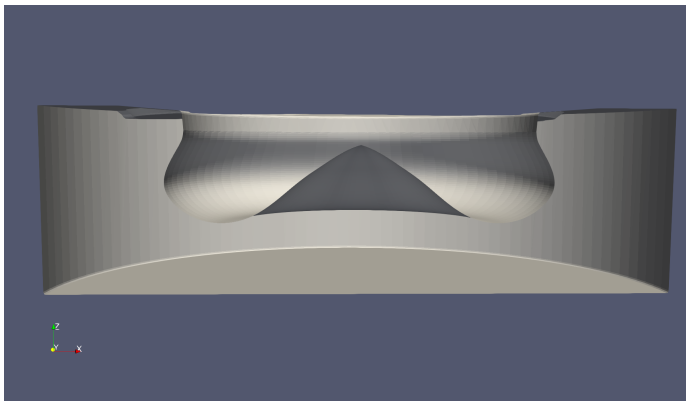
3. Identify priorities for further experimental and computational research.

The engine under investigation is small-bore, swirl-supported diesel research engine. Browsing the repository of the ECN site we can easily download engine geometry files needed for CFD simulations, as well as for experimental data used to evaluate CFD simulation results at this page: https://ecn.sandia.gov/engines/

small-bore-diesel-engine/ The cylinder head is from a GM 1.9L, light-duty diesel engine. Two piston geometries are available: re-entrant and stepped-lip. The piston used in this work will be the conventional re-entrant, shown in Figure 4.1. The engine data are listed in the Table 3.1.



(a) 3D view of the piston with valve cutouts



(b) Piston section

Figure 3.1: Conventional re-entrant piston

Property	Value	Dimension
Bore	82	[mm]
Stroke	90.4	[mm]
Rod length	166,67	[mm]
Displacement volume	0,477	[L]
Bowl volume	23442,51	$[mm^3]$
Cycle	4	[-]
Geometric compression ratio rc	16.7:1	[-]
Squish height	1,09	[mm]
Piston geometry	Re-entrant (with valve cut-outs)	[-]
Intake / Exhaust valves	2/2	[-]

Table 3.1: GM 1.9L Engine Geometry

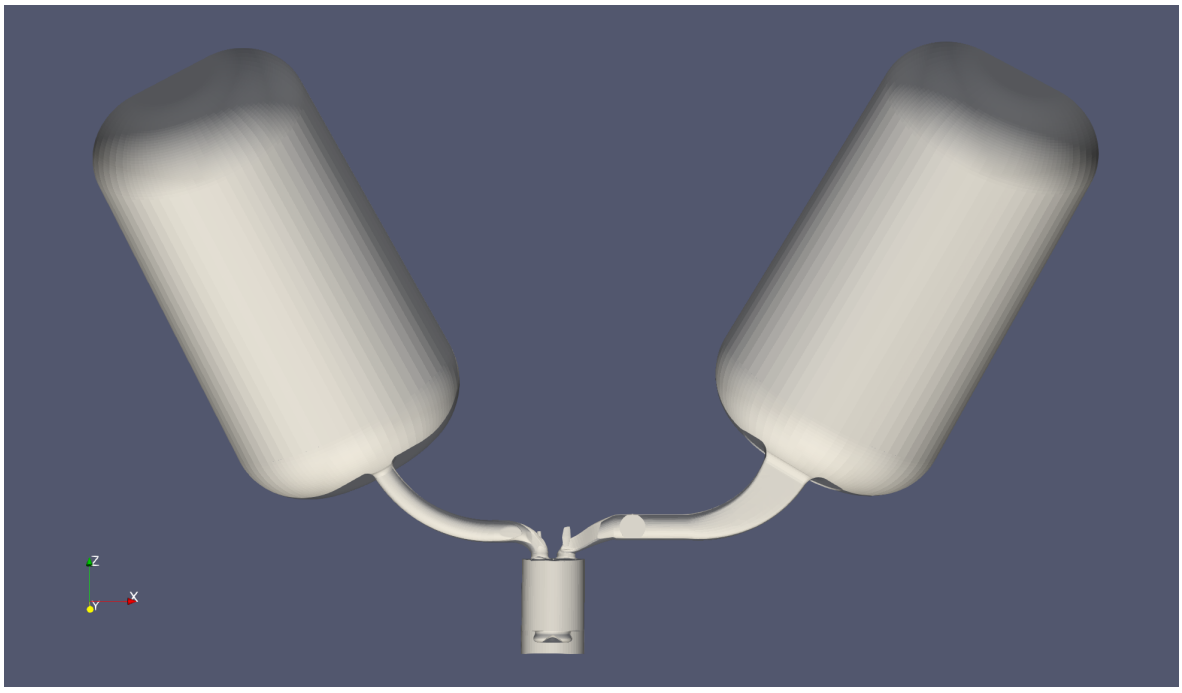


Figure 3.2: Longitudinal section of the Original STL file.

To prove the reliability of OpenFOAM, the engine geometry will not be modified or simplified. It represents a true representation of the real model.
The original geometry in STL format provided by ECN in the Figure 4.6 includes, at the intake and exhaust manifolds, large volume containers in order to simulate the external environment. Since they are not necessary for CFD calculation and increase the number of cells that will compose the Mesh, these containers will not be considered, positioning the inlet and the outlet port at the ends of the manifolds, see Figure 3.3.

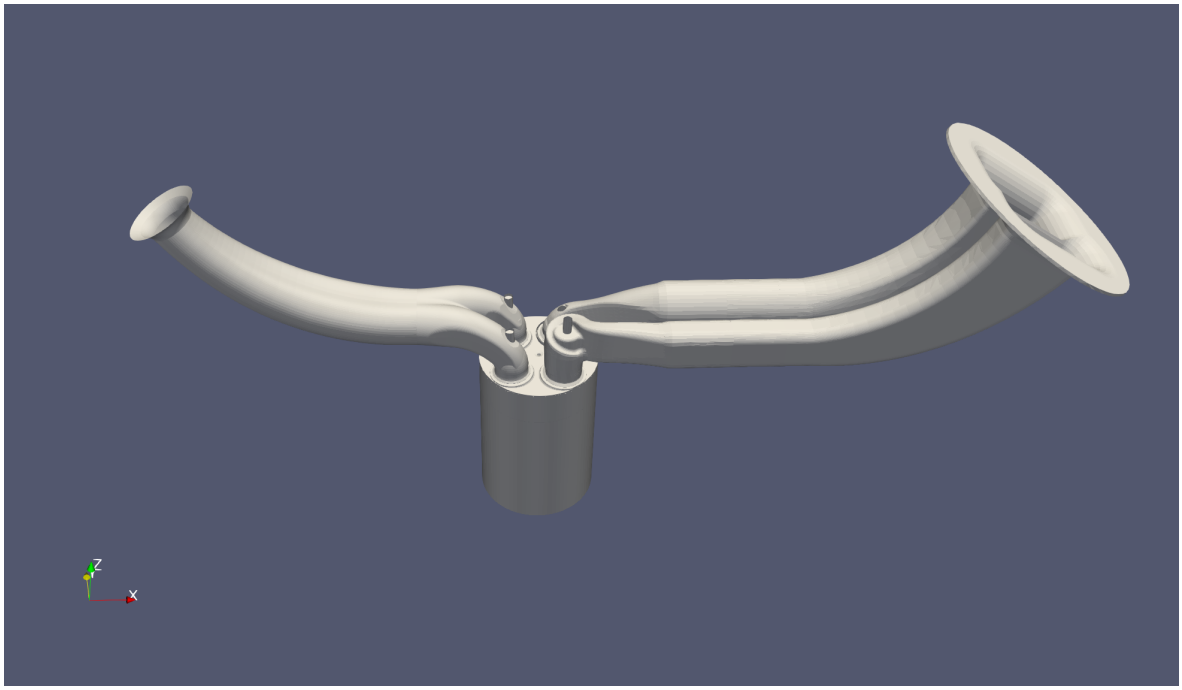


Figure 3.3: Orthogonal view of the modified CAD geometry.

As is often the case with many engines with 4 valves per cylinder, tangential ports and helical ports are combined to create swirl within the cylinder. The intake manifold of this engine is equipped with Swirl Flap to change the swirl ratio inside the cylinder according to the operating points. In this experiment we will operate the engine in wide-open flaps position as shown in the Figure 3.4.

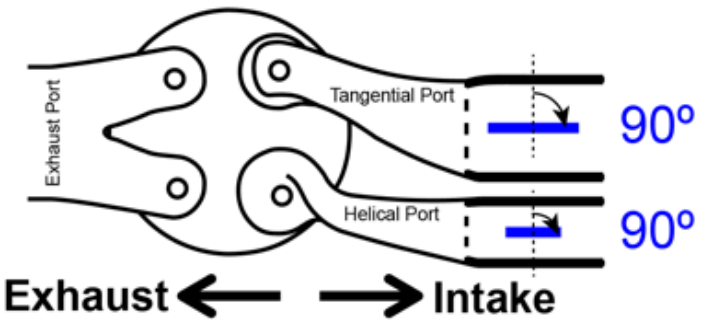
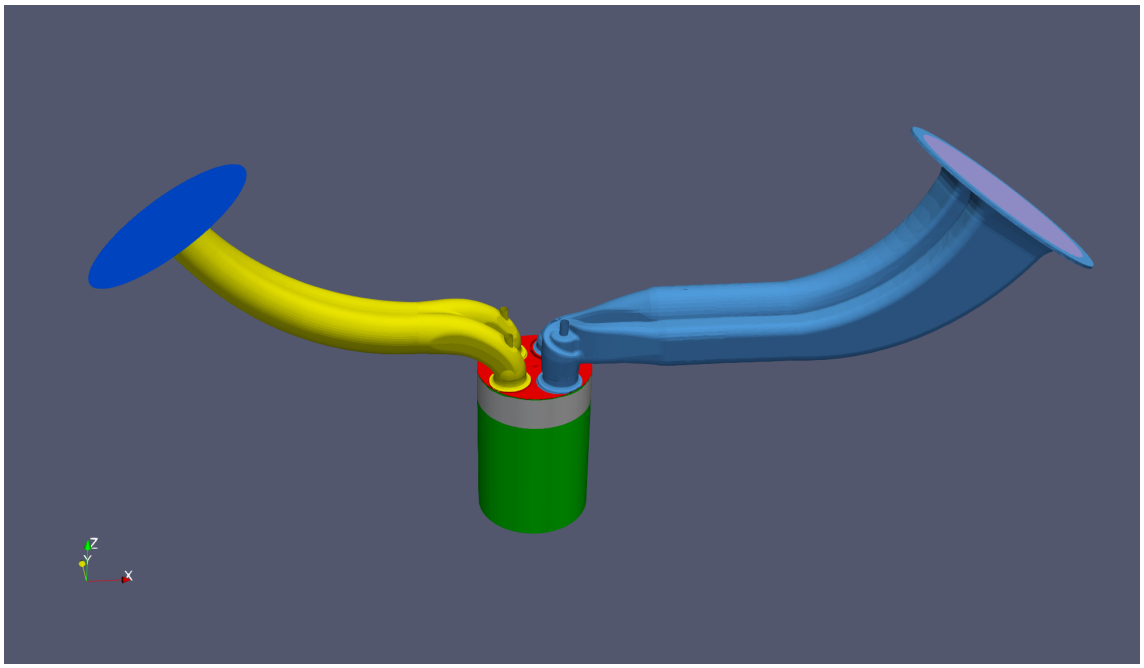


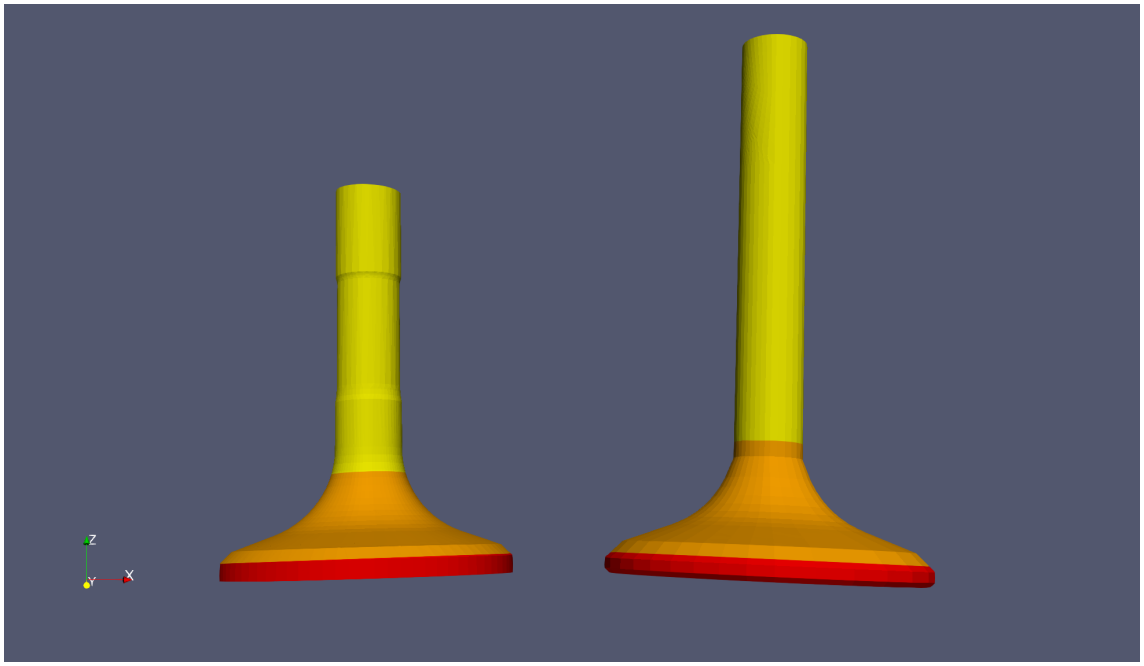
Figure 3.4: Top view of the single-cylinder light duty engine with steady-state swirl ratio of 2.2.

In order to create the part movement that occurs during a complete work cycle, it is necessary to create part files in STL format for each component. The entire assembly is divided into 13 parts listed and shown in Figure 3.5:

1. piston
2. liner
3. cylinderHead
4. Inlet
5. Outlet
6. Wall_inl
7. Wall_exh
8. Valve_stem_exh
9. Valve_stem_inl
10. Valve_top_exh
11. Valve_top_inl
12. Valve_bottom_exh
13. Valve_bottom_inl



(a) Engine assembly



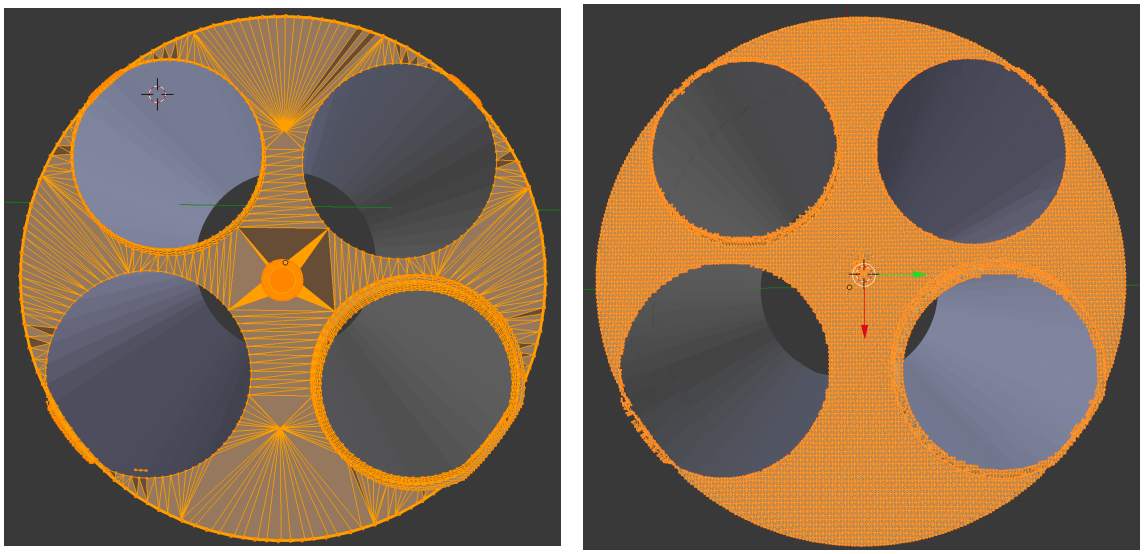
(b) Valve assembly

Figure 3.5: Components

3.1.2 Waterproof surfaces

I experienced the necessity of a good surface triangulation while using snappyHexMesh with this particular geometry downloaded from the ECN site. Based on the experience of other colleagues, this problem does not arise if the geometry is simple and is created from the scratch. Nevertheless, I have found that other users using SHM had to deal with problematic surfaces. Most people are using common CAD software packages and export single surfaces as STL files. These STL files have two big problems, if they represent a closed volume [6]:

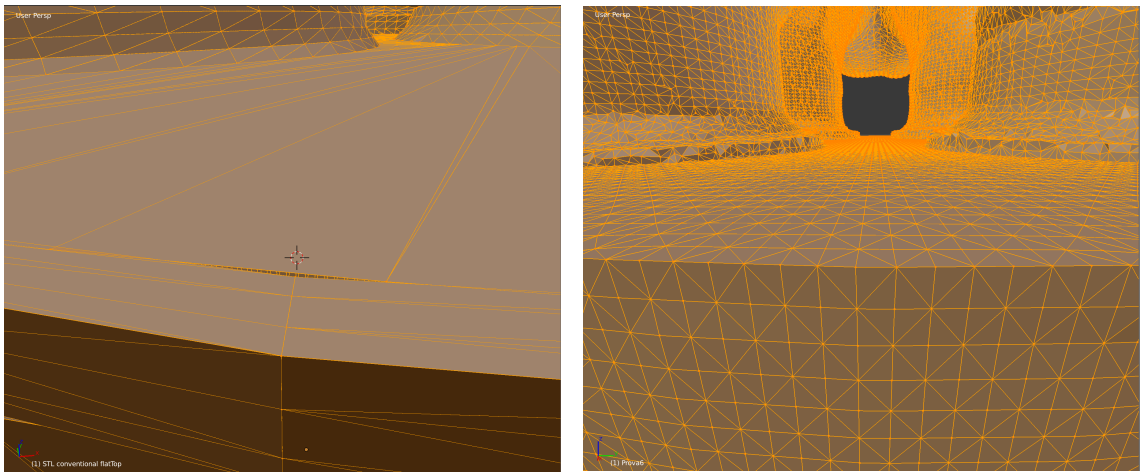
- If the whole model is exported, the volume is closed, but the triangulation is very bad (see Figure 3.6a);
- If you export the model in several steps (several STL's), the connected edges share not the same points (never be waterproof; see Figure 3.7a)



(a) STL file with bad triangulation

(b) STL file with good triangulation

Figure 3.6: Top view of *cylinderHead*



(a) Gaps between different STL files (b) Waterproof closed volume

Figure 3.7: 3D-view of the intersection between *cylinderHead* and *liner*

Both (and especially the second one) influences the mesh generation tremendously. The good thing is that in some case it does not matter as mentioned before . The bad thing is, that you could really get unexpected results. Some of them were:

- snapping problems
- due to snap problems the layer generation was influenced too
- other unexpected behavior with refinement, snapping and zones
- Finally it could happend that snappyHexMesh can not recognize which fluid domain should be used (no cells will be removed, this happen due to gaps)

To avoid this I highly recommend to use:

1. a nice triangulated surface for each patch (see Figure 3.6b).
2. a water proofed STL if it represent a closed surface (that means the shared edges have to be identical for each STL).

For this purpose I used a free CAD software called BLENDER, a valid alternative could be SALOMONE.
However, when the faces are very elaborate, fixing the surface may be too time-consuming. SnappyHexMesh can be used instead. The result is not as accurate as the triangulation using Salome but finally you will end up with a waterproofed STL. In order to do that, the non-closed STL file has to be meshed as accurate as it can

be.
Once the snapped mesh is obtained, the following command 3.1 will transform the unique surface edges extracted in a STL file resulting in a 100% waterproofed STL (as shown in Figure 3.7b) and the surfaces can be refined as much as the user wants without moving into a danger zone (cell size is small enough that snappyHexMesh will realize the gaps).

Listing 3.1: Command

```
#!/bin/sh
# Source tutorial run functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions

runApplication blockMesh
runApplication decomposePar
runApplication mpirun -np 4 snappyHexMesh -overwrite -parallel
runApplication reconstructParMesh -constant
rm -r processor*
runApplication checkMesh
surfaceMeshTriangulate myNewSurface.stl
```

3.2 Different geometries creation

Since multiple meshes are required in order to calculate a work cycle, the appropriate geometry for each mesh is also required. However, it is difficult to determine from the beginning which positions of the engine crank angle require a mesh, which is why it makes sense to create geometries at a distance of one degree of crank angle. By convention we will use 0 to 720 CA. In which the 0 and 720 correspond to the beginning of the power stroke (piston at TDC and all valve closed).
For this reason another thesisist at IVK developed a procedure that allows, with the help of Bash scripts in Linux environment, to create geometries at the distance of one degree crank angle. For this purpose the *surfaceTransformPoints* utility is used, which allows different manipulations of STL files.
Before editing STL files, the names inside the STL file must be adapted. Before proceeding, make sure that the name in the STL file matches the name assigned to the file (i.e. in Listing 3.2).

Listing 3.2: Example of STL file

```
solid cylinderHead
 facet normal -0.749118 -0.662302 -0.0133178
  outer loop
   vertex -0.0318761 -0.0258009 0.107105
   vertex -0.0318621 -0.0258091 0.106733
   vertex -0.0316401 -0.0260609 0.106759
  endloop
 endfacet

[...]

 facet normal -0.622957 0.781525 -0.0338266
  outer loop
   vertex 0.0231458 0.0334396 0.107901
   vertex 0.0232348 0.0334884 0.107391
   vertex 0.0229085 0.033601 0.107865
  endloop
 endfacet
endsolid cylinderHead
```

The next step is to check the file sizing. Since most CAD systems work in millimeters, but OpenFOAM works in meters, the geometry for the OpenFOAM application could be scaled incorrectly. If this is not evident from the STL file, the file can be opened in Paraview and the size of the loaded part is displayed in the Information tab of the Object Inspector. Paraview displays the dimensions in meters here. If a scale is required, this can be done using the *surfaceTransformPoints* command with the -scale option. With this (Listing 3.3) procedure all components can be scaled to the correct unit.

Listing 3.3: Command to scale

```
surfaceTransformPoints -scale "(0.001 0.001 0.001)" cylinderHead.stl
    cylinderHead.stl
```

Once the files have been named and scaled correctly, you can actually create the geometry. Creating geometry is scripted in a Geometry folder that contains the following folder structure (Figure 3.8):

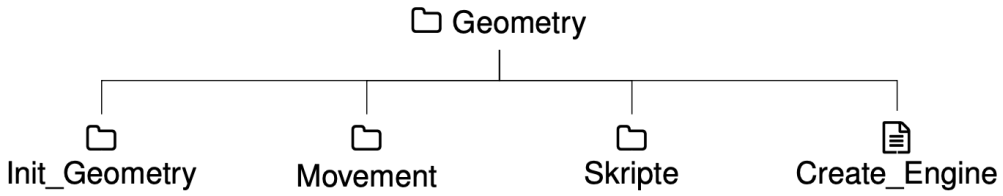


Figure 3.8: Geometry folder structure

- The 13 components of the initial geometry must be placed in the **Init_Geometry**.
- The folder **Movement** contains 3 .txt files:
  - *movePiston.txt*
  - *moveExhaustValve.txt* and *moveIntakeValve.txt*
- The *Create_Engine* script creates additional folders and copies into them the corresponding components:
  - Fixed_Geometry: contains all fixed parts such as the cylinderHead, liner, Inlet, Outlet, Wall_inl, Wall_exh.
  - Piston: contains the piston for each CA. The appropriate value is read from the corresponding file in the Movement folder and the component is moved using the surfaceTransformPoints command.
  - Valve_exh: Contains all components of the exhaust valve for each CA.
  - Valve_inl: Contains all components of the intake valve for each CA.
- The **Skripte** folder contains piston and valve displacement sequences performed by *Create_Engine* script.

The *movePiston.txt* specifies the distance of the piston from the TDC (s_α) during a complete work cycle. The value corresponding to the single CA can be determined by the equation 3.1 [7]:

$$s_\alpha = r(1 - \cos(\alpha) + \frac{r}{4l}(1 - \cos(2\alpha)) \tag{3.1}$$

A quick way to quickly calculate and directly write the .txt file is to use code 3.4 using MATLAB from which we will get the values shown in the Figure 3.9.

Listing 3.4: *movePiston.txt* creation

```
clear all
close all
clc

lk=166.6748*10^-3; %conrod lenght
ls=0.271187; %throw/rodLength
rk=lk*ls; %crank throw
alpha=0; %crank angle

for i=1:721
    s(i)=rk*(1-cos(deg2rad((alpha)))+(ls/4)*(1-cos(2*deg2rad(alpha))));
    alpha=alpha+1;
end

%write data on .txt file
fileID = fopen('movePiston.txt','w')
fprintf(fileID,'%f\n',s)
fclose(fileID)
```

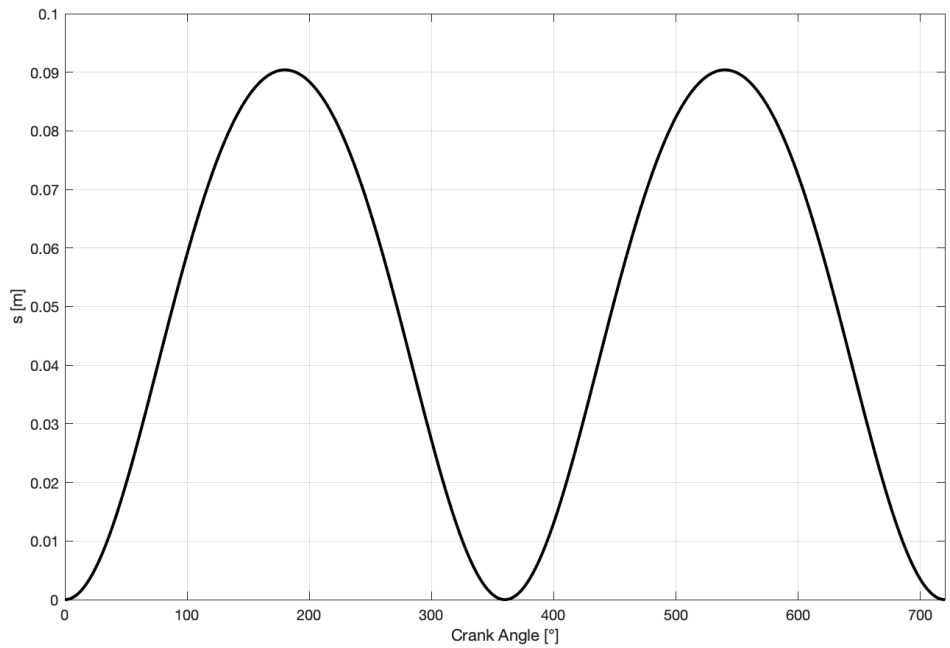


Figure 3.9: Piston distance from TDC

Data on valve lifts can be found in the documentation provided by ECN. In this specific case, to obtain a value for each CA, fitting curves have been made. In addition, the original Valve lifting involved valve overlap and too small lift values that would have required too fine mesh and therefore too long simulation time. For this reason, all values lower than 0.7 mm were discarded (Listing 3.5).

Listing 3.5: *moveIntakeValve.txt* creation

```
%intake valves

y= xlsread('Valve lift data.xlsx','Rearranged','J2:J654')*10^-3;
x= xlsread('Valve lift data.xlsx','Rearranged','I2:I654');
x_new=0:1:720;
y_new=interp1(x,y,x_new);
lift=y_new';

for i=1:721
    if  lift(i,1)<0.0007 % all the values lower than this are thrown away
        lift(i,1)=0;
    else
    end
end

lift(1,1)=0;
lift(721,1)=0;
fileID = fopen('moveIntakeValve.txt','w');
fprintf(fileID,'%f\n',lift);
A = importdata('moveIntakeValve.txt');
fclose(fileID);
```

Valve movement	Crank Angle [°]
Exhaust valves opening	144
Exhaust valves closing	348
Intake valves opening	374
Intake valves closing	556

Table 3.2: Valves intake/closing points

The piston and valve displacement sequences executed by the Create_Engine script stored in the Script folder are executed one after the other. The appropriate value is read from the corresponding file in the Movement folder and the component is moved using the *surfaceTransformPoints* command.

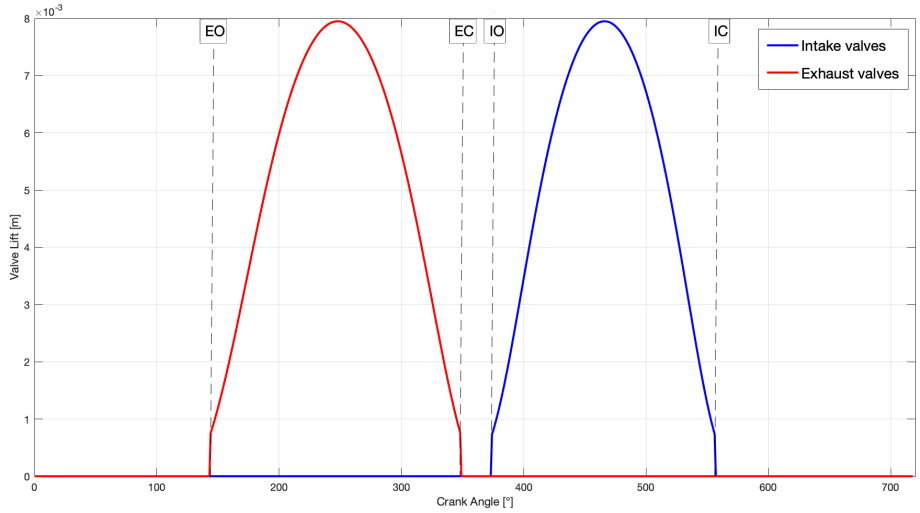


Figure 3.10: Valve lift data

The valves are rotated in the vertical direction with the -rollPitchYaw option before the translation movement, then moved along the Z axis and rotated again in the original direction (as in Listing 3.6).
Finally, a folder is created for each crank angle and the pistons and valves corresponding to the fixed components are merged into an *Engine.stl* file using the *cat* (Listing 3.7) command and stored in the corresponding folder.

Listing 3.6: Valve lifting command example

```
surfaceTransformPoints -rollPitchYaw "(0 -"$Valve_angle" 0)"
    Valve_top_inl.stl Valve_top_inl1.stl
surfaceTransformPoints -translate "(0 0 -"$displacement")"
    Valve_top_inl1.stl Valve_top_inl2.stl
surfaceTransformPoints -rollPitchYaw "(0 "$Valve_angle" 0)"
    Valve_top_inl3.stl "$CA"_Valve_top_inl.stl
```

Listing 3.7: *cat* command

```
cat Fixed_Geometry/*.stl Piston/"$CA"_piston.stl Valve_exh/"$CA"_*.stl
    Valve_inl/"$CA"_*.stl >   Engine.stl
```

Chapter 4

Mesh creation

At this point we should have a perfect geometry that allows the user to take full advantage of all the features of the reference utility: snappyHexMesh.
As mentioned in the introductory chapter of OpenFOAM, the accuracy and reliability of the calculation depend on the number of cells that make up the Mesh. On the other hand, a high number of cells significantly prolongs the calculation time. It is therefore necessary to find the right compromise between precision and calculation time, and consequently adapt the mesh to the different geometries. For this purpose we will use another script created at the IVK that, however, did not include the use of SHM and therefore was adapted by me to use this utility.

4.1 Different geometries refinement

As introduced in chapter two, the first step is to create the *blockMeshDict* (see Listing 4.1) in order to create a block of hexagonal cells that totally envelopes our Engine.stl file.
It is not important to create a refined block, as the cells will be re-finished via SHM.
It is convenient to create square cells to have as uniform a region as possible.
The boundaries will also be specified later.

Listing 4.1: *blockMeshDict* creation

```
/*--------------------------------*- C++
    -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version: 4.0                                    |
|   \\  /    A nd           | Web:     www.OpenFOAM.org                       |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    * * //

convertToMeters 1;

//These vertices define the block below. It envelopes the stl files. The
    block can be even bigger than the stl files
//Watch out if the stl files are created in mm or m!

vertices
(
    ( -0.287 -0.088 -0.02)
    (  0.380 -0.088 -0.02)
    (  0.380  0.064 -0.02)
    ( -0.287 0.064 -0.02)
    ( -0.287 -0.088 0.277)
    (  0.380 -0.088 0.277)
    (  0.380  0.064 0.277)
    ( -0.287 0.064 0.277)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (68 20 34) simpleGrading (1 1 1)
);

edges
```

```
(
);

boundary
(
    allboundary//Don't worry about these settings
    {
        type patch;
        faces
        (
            (3 7 6 2)
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
            (0 3 2 1)
            (4 5 6 7)

        );

    }

);

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

The second step is to create the surfaceFeatureExtractDict simply as shown in Listing 4.2.

Listing 4.2: *surfaceFeatureExtractDict* creation

```
/*--------------------------------*- C++
    -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version: 4.0                                    |
|   \\  /    A nd           | Web:     www.OpenFOAM.org                       |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      surfaceFeatureExtractDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    * * //

//JN: Here we define, which edges we want to use as features for the
    geometry. Usually we use all of them


Engine.stl
{
    extractionMethod  extractFromSurface;

    extractFromSurfaceCoeffs
    {
        includedAngle  180;
    }

        writeObj             yes;
}




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

The final step is to create the different *snappyhexMeshDict* because, depending on the positions of the piston and valves, the size of the cells in the Mesh must be differently shaped (see Table 4.1).

Dictionary	Condition
snappyHexMeshDictVC	All valves closed
snappyHexMeshDictFE	Fine mesh around Exhaust valve bottoms
snappyHexMeshDictCE	"Course" mesh around Exhaust valve bottoms
snappyHexMeshDictFI	Fine mesh around Intake valve bottoms
snappyHexMeshDictCI	"Course" mesh around Intake valve bottoms

Table 4.1: Different *snappyHexMeshDict*

When one of the valves (exhaust or intake) opens, the space between the valve bottom and the valve seat is very small and requires a very dense mesh in that region. The valve will then open more and more, so a narrow mesh is no longer necessary.
When both valves are closed, a refinement of the valve area is unhelpful. However, a refinement suitable for the cylinder head must be considered as when the piston is at the TDC (i.e. power stroke) the squish height is very small (about 1mm).

Now, referring to the structure shown in the Figure (2.3), the geometry must be inserted as follows:

Listing 4.3: Geometry in *snappyHexMeshDict*

```
geometry
{
    Engine.stl // name of the stl file
    {
        type triSurfaceMesh;
        regions
        {
            cylinderHead           // Named region in the STL file
            {
                name cylinderHead; // User-defined patch name
            }

[...]

        }
    }
    //refinement_box
    //{
    //    type   searchableBox;     // region defined by bounding box
    //    min    (-0.043 -0.043 0.105);
    //    max    (0.043 0.043 0.110);
    //}
};
```

4.1.1 Valve closed refinement

When setting the Dict, always consider the most critical geometry. In the case where both valves are closed, the geometry to be considered is when the piston is at the TDC and therefore, in this case, the space between the piston and the cylinder head is 1.09 mm.
For this reason we have to refine as follows:

Listing 4.4: Refinement in VC Dict

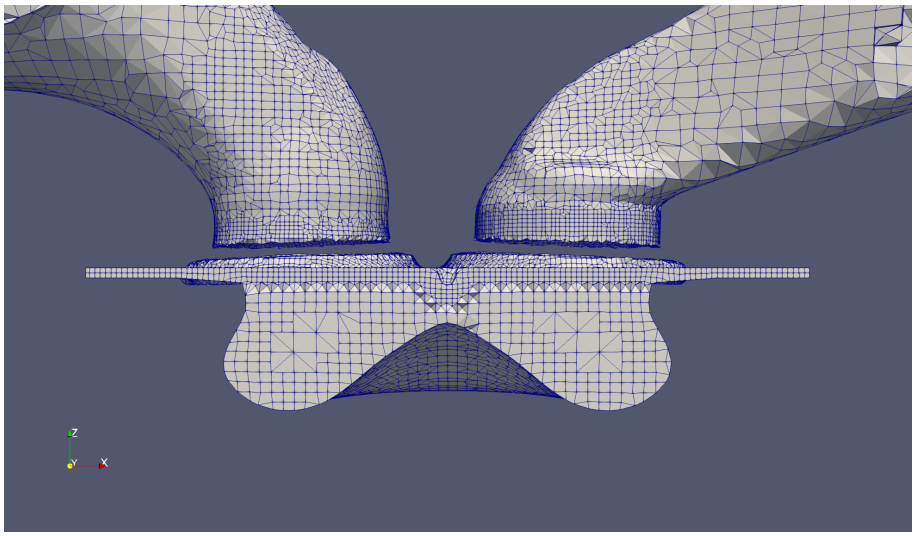
```
refinementSurfaces
    {

Engine.stl//JN: name of the stl file
    {
        level (2 2); //tutto 2
        regions
        {
            cylinderHead
            {
                level (4 4);
            }
            piston
            {
                level (3 3);
            }
            Wall_inl
            {
                level (3 3);
            }
            Wall_exh
            {
                level (3 3);
            }
            Valve_bottom_inl
            {
                level (4 4);
            }
            Valve_bottom_exh
            {
                level (4 4);
            }

        }
```

```
}

}
```

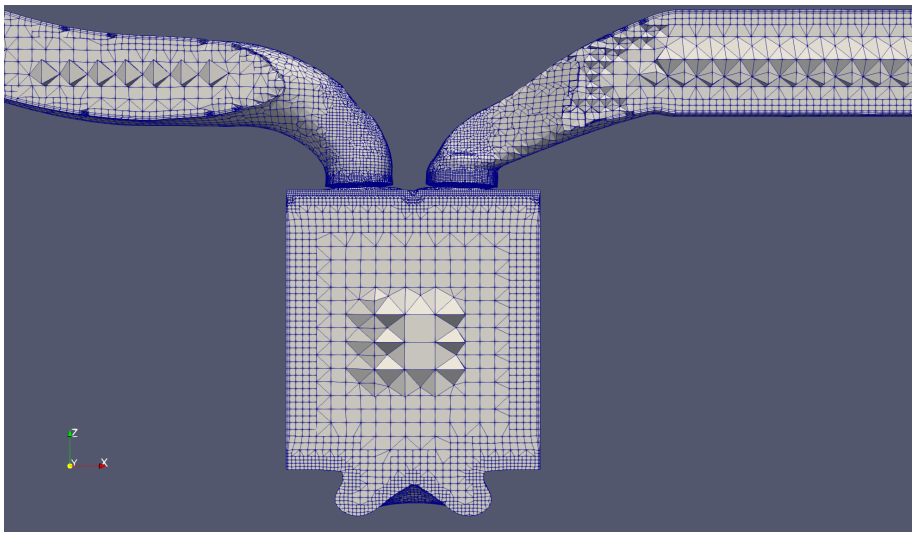
An alternative could be to introduce in the geometry a refinement box section and then enter our parameters in refinementRegions.

```
refinementRegions
{
    box_head
    {
    mode inside;
    levels ((1 4));
     }
}
```

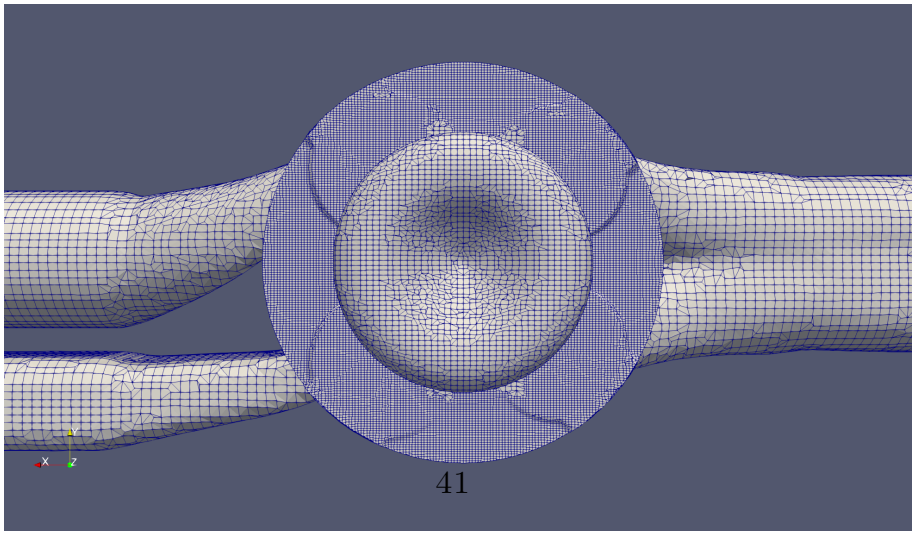
In this way we will have a Mesh without issues when the piston is at the TDC (Figure 4.1a) and also when it is close to the BDC (Figure 4.1b). As we can see from the images the cells gradually become larger when they move away from our points of interest. This allows us to decrease the total number of cells and therefore the simulation time. How gradually we set it through *nCellsBetweenLevels*.



(a) Valve closed Mesh at 0° CA



(b) Valve closed Mesh at 557° CA



(c) Valve closed Mesh bottom view

Figure 4.1: Valve closed Mesh

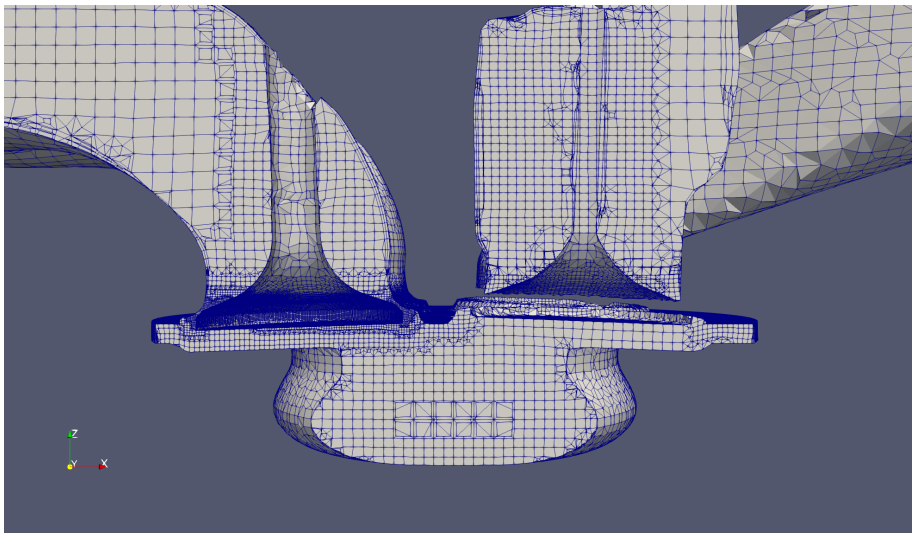
4.1.2 Valve bottoms fine refinement

This Mesh is the one that requires the highest degree of refinement, as it will include all geometries where the valve lift is between 0.7 mm and 2 mm.
Clearly the Dict for intake valves are distinct from those for exhaust valves, but they are finished in a specular manner.
Since a high level of refinement substantially increases the number of cells, it is essential to try to limit the area in question as much as possible. In addition, when the valve opens the inlet and outlet air will also bag the final part of the intake and exhaust ducts and the valve stem. For this reason it is necessary to refine the surrounding areas as well. In this case I used the box refinement: *box_valve* that includes also the valve stem and *box_special* that includes only the small gap between the valve bottom and the valve seat.

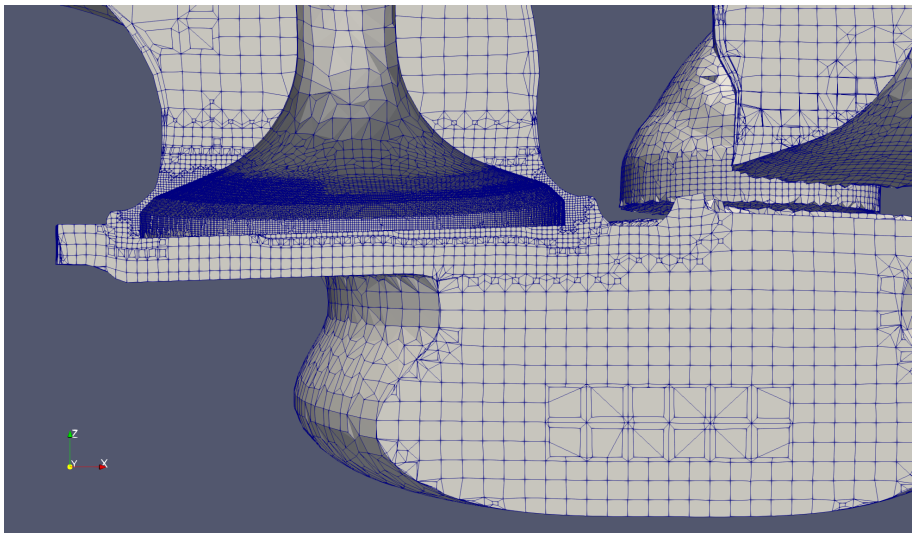
```
refinementSurfaces
    {

Engine.stl//JN: name of the stl file
    {
        level (2 2); //all the other surfaces 2
        regions
        {
            cylinderHead
            {
                level (4 4);
            }

            piston
            {
                level (3 3);
            }
        }
    }

    }
```

```
    refinementRegions
    {
        box_valve
        {
        mode inside;
        levels ((1 3));
         }
```

```
    box_special
    {
   mode inside;
    levels ((1 6));
   }
}
```

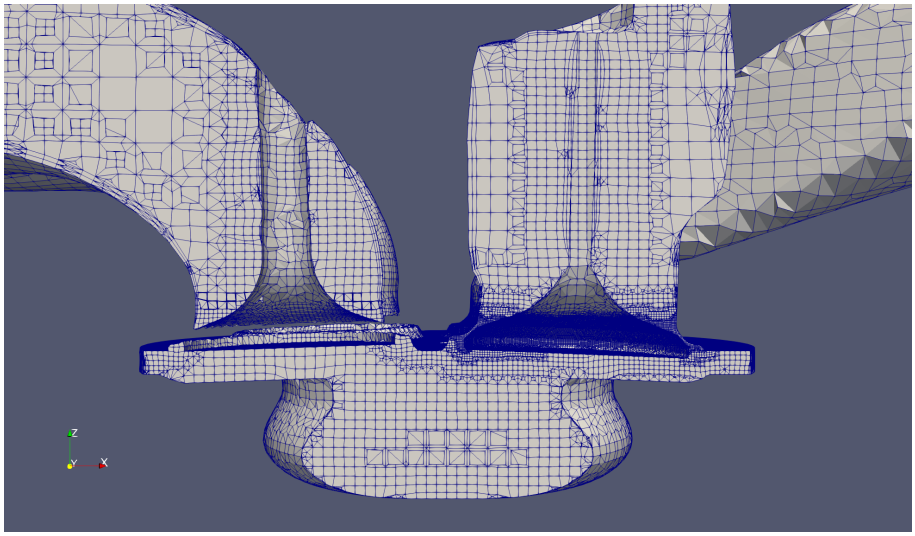


(a) Cylinder section at 348° CA

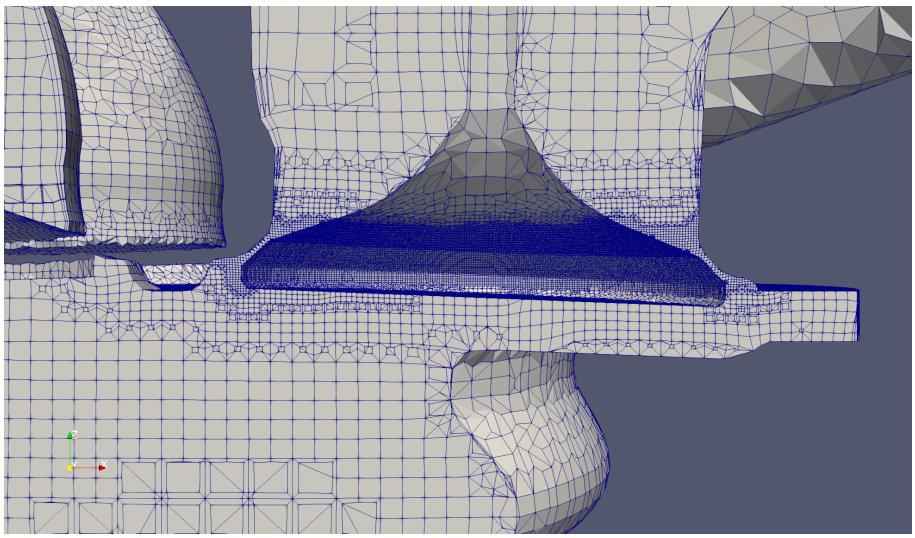


(b) Close-up of exhaust valve

Figure 4.2: Fine exhaust Mesh



(a) Cylinder section at 374° CA



(b) Close-up of intake valve

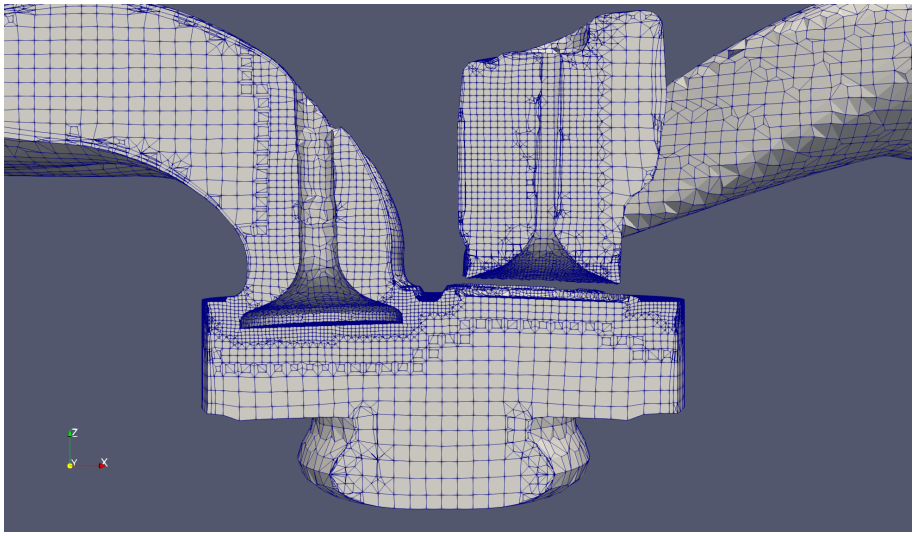
Figure 4.3: Fine intake Mesh

4.1.3 Valve bottoms "course" refinement

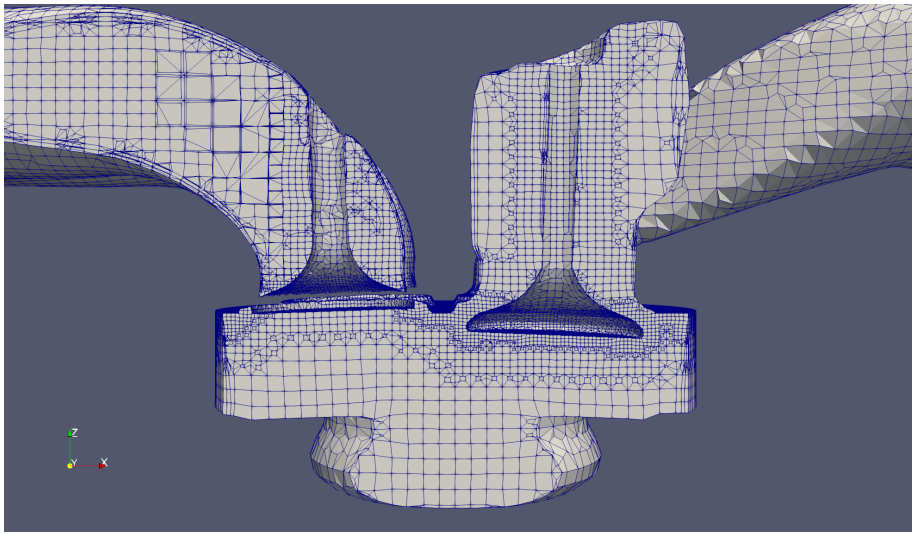
While the fine mesh will only cover a few cases in our cycle, this mesh will cover a much higher number of cases. This is why we prefer to have a lighter mesh. Setting and images are shown below.

```
refinementSurfaces
    {

Engine.stl//JN: name of the stl file
    {
        level (2 2); //tutto 2
        regions
        {
            cylinderHead
            {
                level (4 4);
            }
            Valve_bottom_exh
            {
                level (3 3);
            }

        }
    }

    }
```

```
    refinementRegions
    {
        box_valve
        {
        mode inside;
        levels ((1 3));
         }
        box_special
        {
       mode inside;
        levels ((1 3));
       }
    }
```



(a) Cylinder section at 300° CA



(b) Cylinder section at 402° CA

Figure 4.4: Course exhaust and intake Mesh

4.2 Mesh Generation script

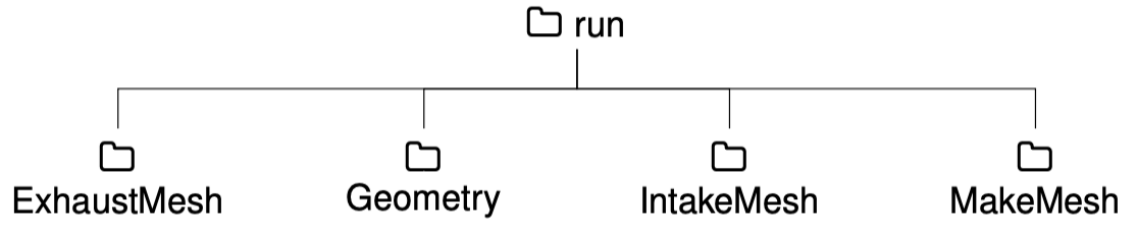


Figure 4.5: Run folder structure

In the course of a complete work cycle, LibICE requires the use of several meshes. Even if LibICE has a utility that moves the grids connected to the moving components, you should not compress or stretch the grids too much. As experienced in this case, you will need one graticule every 6 CA° when both valves are closed and a 3 CA° interval when moving the valves.
However, in some cases you may need to use mesh at specific points.
A script has been developed for this purpose: *MeshGeneration* , which executes a cycle for interlacing set by us. The operations include:

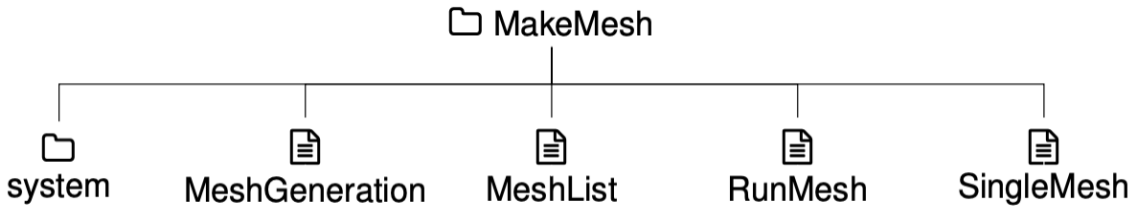


Figure 4.6: MakeMesh folder structure

1. read the .txt files (MoveIntake and MoveExhaust) in order to select the snappyHexMeshDict corresponding to our needs.
2. copy the corresponding geometry (in STL format) in the triSurface folder.
3. execute the mesh using the *RunMesh* script.
4. merge one or both the manifolds.

This last operation is necessary because in case 2 or all the valves are closed the manifold mesh is removed because it is outside the closed surface. Therefore we use the mergeMesh application to add the missing part of the engine. For this purpose

the Mesh for Intake and Exhaust must be previously created in the corresponding folders as shown in Figures 4.5.

The *RunMesh* script has been modified to allow the use of snappyHexMesh.

The *SingleMesh* script has the same function as *MeshGenaration* but used to create a single Mesh.

After all networks are created the MeshList script creates a list of all the CA for which the Mesh were created.

4.2.1 Mesh quality

At the end you have to verify that there are no errors in any of the generated meshes. This can be verified visually using Paraview. Also in the folders containing the mesh is saved the *log.checkMesh* file that reports the quality of the generated mesh and any errors. A parameter that can dramatically affect the result is Mesh non-orthogonality whose maximum value must not exceed 70. Instead the Max skewness value even if reported as warning can be tolerated if it does not exceed 10.

Chapter 5

Simulation

Once you have created and verified the meshes, it is time to set up and run the simulation.
Depending on the number of cells in each mesh and also on the characteristics of the fluid, it can take a long time. The type of machine used and the number of processors also affects the calculation time, and in some cases the stability of the calculation.
Again, a script will be used to set the parameters and run the simulation.

5.1 Setting the simulation

The parameters of the simulation are contained in the Initialisierungscase (structure shown in Figure 5.1):

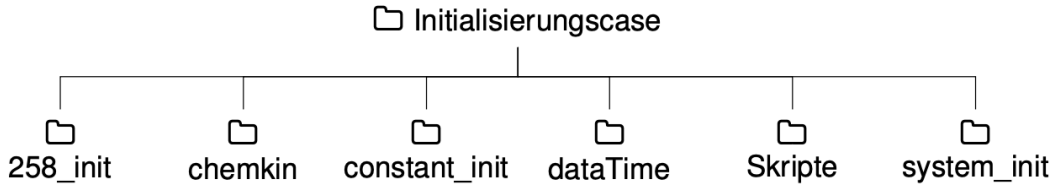


Figure 5.1: Initialization Case folder

- **258_init**: This folder is used as a template to start our calculation. This crank angle is chosen because it represents the maximum opening moment of the exhaust valves and therefore the pressure inside the cylinder is plausibly similar to the atmospheric one. Inside this folder are set the characteristics of all the components and the internal field. The initial values entered are as close as possible to reality.
- **chemkin**: define the chemical reactions during combustion, in this case neglected.
- **constant_init**: This folder contains all the data that remains constant during the entire simulation:
  - *triSurface*: CylinderZone.stl, ExhaustZone.stl and IntakeZone.stl files are stored in this folder. These are necessary to recognize which cells belong to the respective zones. In order to obtain results that relate to a specific zone, i.e. pressure exclusively in the cylinder.
  - *combustionProperties*: here you can activate combustion (in this case disabled) and insert the combustion model to be used.
  - *engineGeometry*: here are the engine characteristics (conRod lenght, bore, stroke, clearance, rpm). In order for the mesh to be moved, here are the coordinates of the moving parts, the axis and direction of movement. For the piston the origin is in TDC position and the axis is the Z axis. For valves, the origin must be considered when the valves are closed and, even if the type of coordinates entered is cylindrical, in the axis entry it is sufficient to enter the coordinates of a point belonging to the valve axis.
  - *exhaustValve.txt / intakeValve.txt*: previously reviewed files that contain the Valve Lift data.
  - *RASProperties*: If the RAS method is chosen, the turbulence model used is entered here: kEpsilon.
  - *LESProperties*: If the LES method is chosen, the turbulence model used is entered here: Smagorinsky.
  - *turbulenceProperties*: Here is selected the dict to use: LES or RAS.
- **system_init**: here are the dict that control the various parts of the simulation.
  - *controlDict*: This dict is used to control the start, end and step time of the simulation. We can also set the maximum value of the Courant number. This represents the base dict that will be manipulated by the skripts as shown below.

  - *decomposeParDict*: In order to conduct a parallel simulation it is first necessary to divide the cells into various subdomains. In this Dict we set the number of subdomains and the method of subdivision.

- **dataTime**: The files containing the pressure data within the intake (p_in.t) and exhaust manifolds (p_ex.t), extracted from the experimental tests (Figure 5.2)), must be entered here.

- **skripte**: Here are the skripts that will be later explained to prepare and start the simulation.

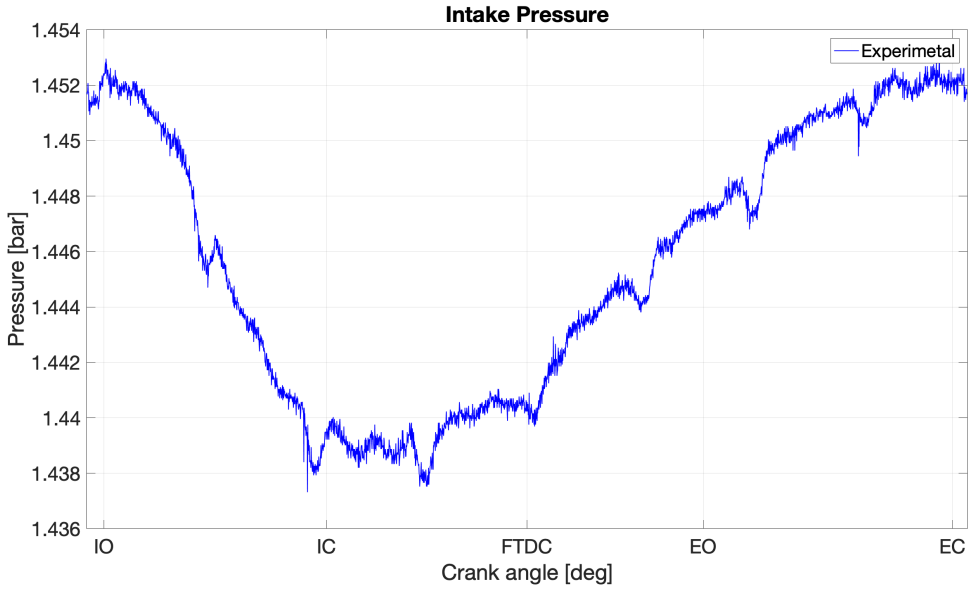


Figure 5.2: ECN experimental intake pressure

After properly setting up our default Initialisierungscase, we have to create the folder where all the Mesh will be inside and where the simulation will take place. The *RechnungVorbereiten.sh* skript creates a folder (shown in Figure 5.3) numbered by us with the option STROKE and will copy inside it the Mesh we indicated, then it will use other scripts to modify the startTime and endTime in the controlDict for each of the selected CA.

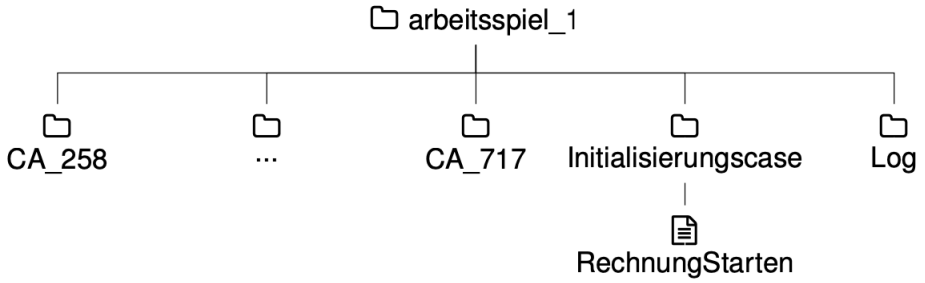


Figure 5.3: Work cycle folder

5.1.1 Parallel running and decomposePar

A parameter that can greatly influence the time and reliability of the simulation is not only the number of subdomains but also the method of subdivision. In order to obtain a compromise between speed and accuracy I investigated the difference between the various methodologies [bibliografia 4]:

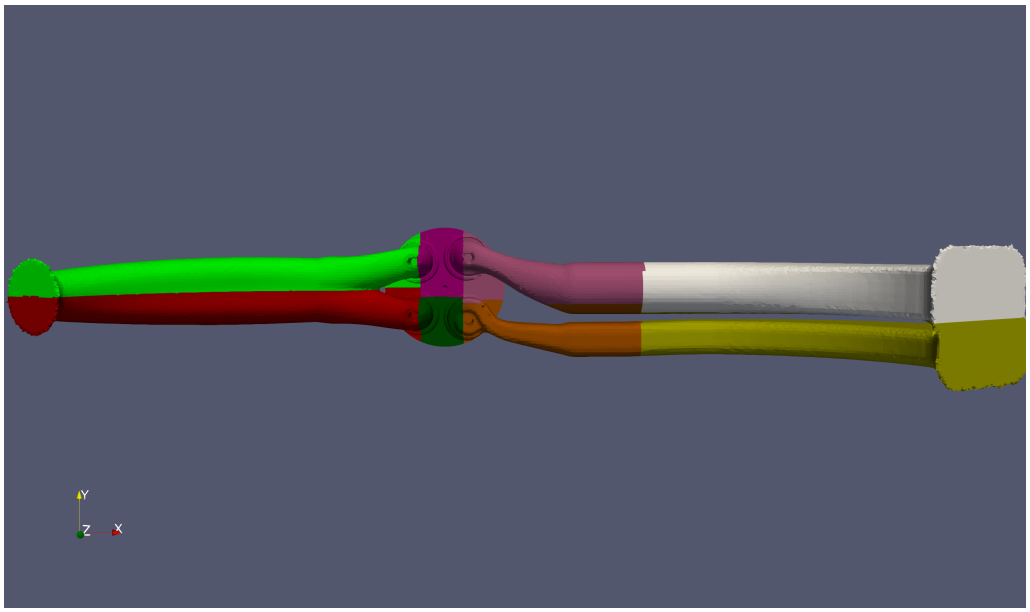
- **simple**: Simple geometric decomposition in which the domain is split into pieces by direction, e.g. 2 pieces in the x direction, 1 in y etc.
  When the number of cells is not evenly distributed, using this method is not recommended because you create subdomains with more cells compared to others.

- **hierarchical**: Hierarchical geometric decomposition which is the same as simple except the user specifies the order in which the directional split is done, e.g. first in the y-direction, then the x-direction etc.

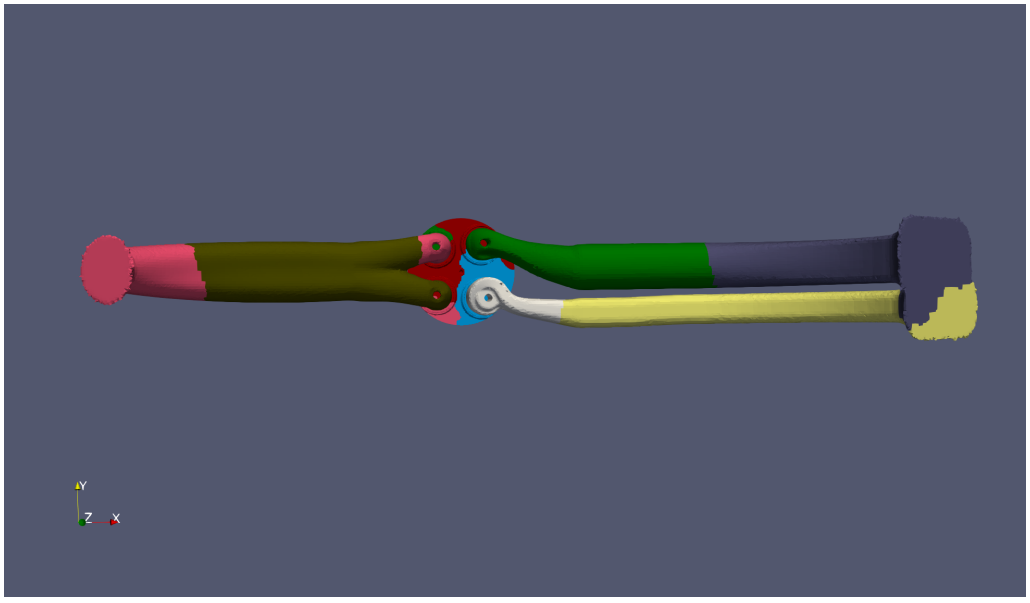
- **scotch**: Scotch decomposition which requires no geometric input from the user and attempts to minimise the number of processor boundaries. The user can specify a weighting for the decomposition between processors, through an optional processorWeights keyword which can be useful on machines with differing performance between processors.

- **manual**: Manual decomposition, where the user directly specifies the allocation of each cell to a particular processor.

The most suitable methods for this study are hierarchical and scotch method. An example is shown in Figure 5.4 where the engine has been divided into 8 subdomains, 4 in x direction and 2 in y direction.



(a) Hierarchical method



(b) Scotch method

Figure 5.4: Different decomposition method

Although the scotch method is also recommended by other users in the forums for its ability to minimize the number of processor boundaries, in the tests done in this case in most cases it reported single floating point error. Given these considerations, the method chosen for this simulation is the hierarchical one.
Another important parameter is the number of subdomains to use. Logically you could think that to speed up our simulation we must increase the number of processors to use. In reality the time taken is not proportional to the number of processors used. This depends very much on the number of cells present in the Mesh which in this case varies from CA to CA. Moreover OpenFOAM requires the assignment of a minimum number of cells for each processor. Increasing too much the number of processors you can also obtain a slowdown of the simulation.

5.2 Starting the simulation

At this point the folder containing the first work cycle has been created through the *Initialiesirungscase.sh* script. It's time to start our simulation through the *RechnungStarten* situated in the skripte folder in the work cycle folder. The script runs the following applications in order, beginning from the starting folder (i.e. 258 as shown in the Figure 5.3):

1. **checkMesh**:it controls the initial mesh and is required for the following application.

2. **moveEngineTopoMesh**: this application moves the mesh as set in engineGeometry Dict. The time step is set in the controlDict to 1 CA by default.

3. **topoSet**: this command creates the cell zones as indicated in the STL files (IntakeZone , ExhaustZone, CylinderZone), fundamental step to create the log files pertinent to the cylinder area.

4. **decomposePar**: the mesh is divided into several processors as indicated in the appropriate Dict so as to allow the simulation in parallel.

5. **ivkEDCsprayEngineFoam**: this application uses the solver developed at the University of Stuttgart for the simulation of engine duty cycles on the basis of existing solvers in LibICE.
   When this application is run in parallel you must make sure that the number of processors coincides with the number indicated in decomposeParDict.
   The results of this solver are written in real time in a file in each processor folder: logCylinderSummary*.

6. **reconstructPar**: rebuilds and merges processor folders.

7. **cp**: copies the logCylinderSummary* to the folder Log that collects all the results of the various CA.

8. **mapMyFields**: This application transfers the data from each cell obtained in this case folder to the next one.

Once these steps are completed, the script can jump in the next case folder as specified in the initialization folder.
The progress of the simulation can be monitored through the *top* command on the terminal to see which application the processors are running. We can also track in real time the logCylinderSummary that contains: time step, volume, pressure, temperature and turbulence.
This can be easily done using the *tail -f* terminal command as an example:

tail -f CA_0258/processor2/logCylinderSummary.258.dat

Usually the initial cycle does not produce results close to reality, because the velocity field is created from scratch. So after the so-called cycle 0, we should perform at least 1 or more cycles.
To transfer cell data from one cycle to another we still have to use the mapMyFields application, this time manually, as shown below:

mapMyFields arbeitsspiel_0/CA_0714 -sourceTime 720 -case arbeitsspiel_1/CA_00

5.2.1 Executing in background

Since the simulation of a single cycle requires several days of calculation, it is recommended to use a cluster that does not cause overheating problems and that can remain on for as long as necessary. In this case I had at disposal a cluster with 32 processors provided by IVK.
When you execute a job in the background (i.e. using &), and logout from the session, your process will get killed. You can avoid this executing the job with *nohup* as shown below:

nohup .RechnungStarten &

As support for the discussion above and the importance of running the simulation in the background in clusters, the timing of the tasks are shown in the Table 5.1.

Order	Task	Time
#1	Geometry creation	up to 8 hours
#2	Mesh creation	up to 2 days
#3	Running the simulation (cold flow)	up to 2 weeks
#4	Log Data transfer	few minutes
#5	Data transfer for visual analys	several hours

Table 5.1: Time needed for each Task

Chapter 6

Post-processing

This section is dedicated to data collection and analysis. After having carefully monitored in real time the development of the simulation, avoiding interruptions and unexpected CA skipping, the results of the entire cycle will be obtained.

6.1 Results analysis

As previously mentioned, pressure, temperature and volume data are saved in the respective log files (logCylinderSummary*). This data can be quickly analyzed using the Linux gnuplot application to view the trend of the above mentioned parameters. I preferred to analyze the data using Matlab for the convenience with which I could compare the data obtained from the simulation to those obtained experimentally.
Once we have discussed the parameters present in the log files, we can show the trend of some of these parameters inside the cylinder using Paraview.
In the figures below are shown the values of Volume (6.1), Pressure (6.2) and Temperature (6.3) obtained with OpenFOAM.

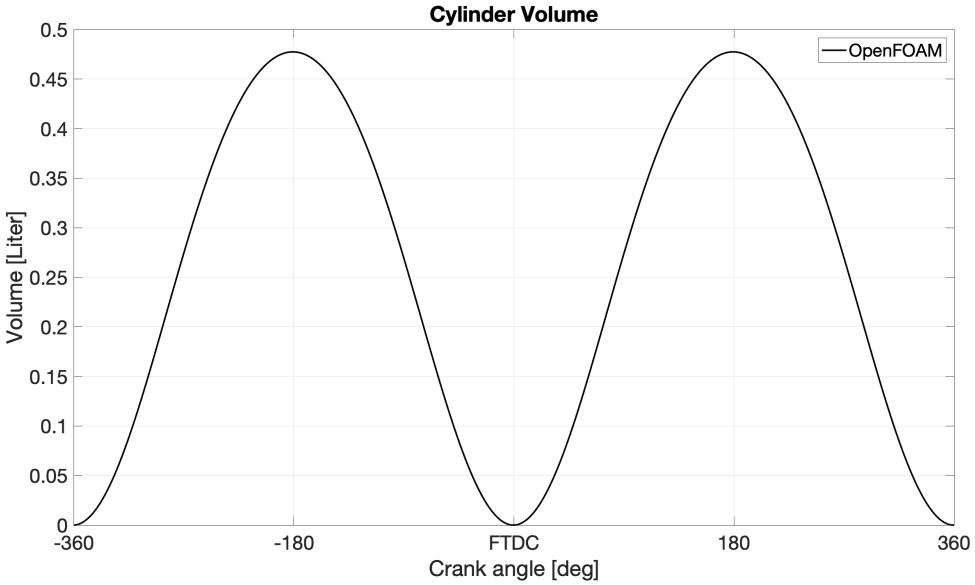


Figure 6.1: Cylinder volume data from OpenFOAM

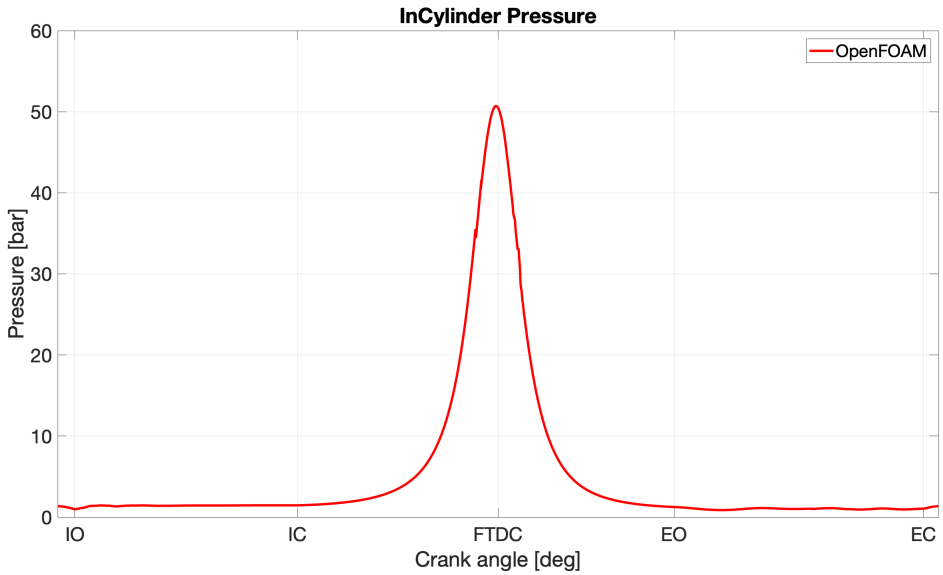


Figure 6.2: In Cylinder pressure data from OpenFOAM

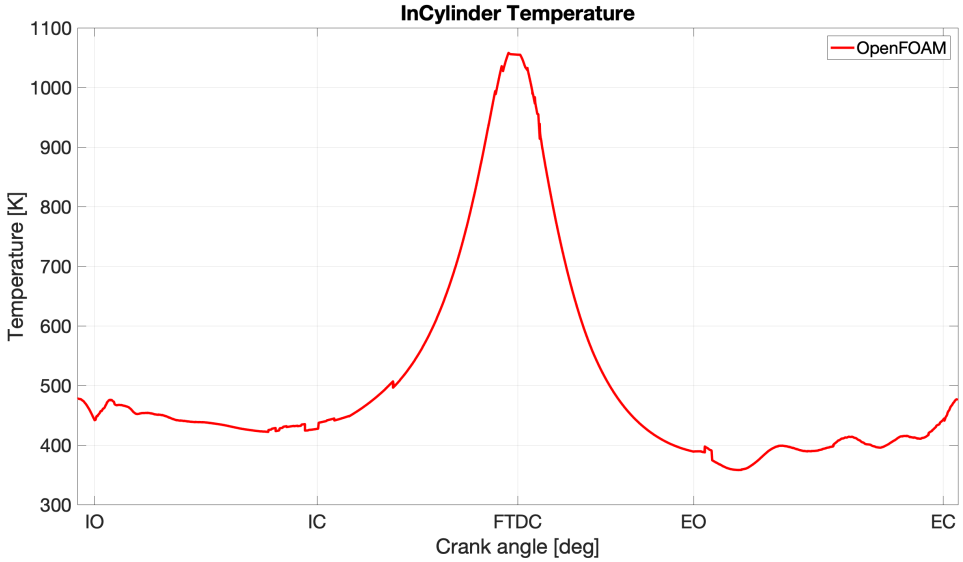


Figure 6.3: In Cylinder temperature data from OpenFOAM

6.2 Comparison with experimental data

The comparison of the simulation data with the experimental data is decisive to determine whether the calculation was performed well and whether the model is reliable.
The volume inside the cylinder measured during the simulation by OpenFoam is extremely similar to the experimental one, showing that the movement occurred as indicated which indicates that the Mesh is geometrically flawless (Figure 6.4).

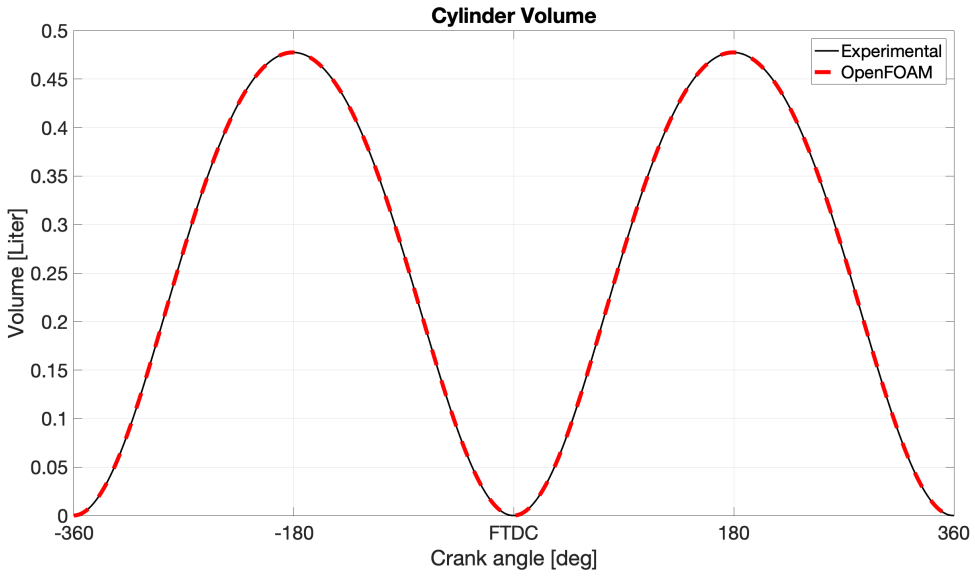


Figure 6.4: Volume data comparison

A data that can help us understand whether the model is reliable or not is the pressure. In the Figure (6.5) we can see that in a large part of the cycle the pressure trend in the chamber is equal to the experimental one. A closer look at the pressure curve in Figure (6.6) shows that the calculation produces constant results only after several cycles. This is due to the fact that at the beginning of the simulation the flow field of our model is stationary. This is why it is necessary to repeat more than one cycle, mapping the flow field of the previous cycle into the new one in order to develop the flux properties and achieve a model as close as possible to reality.

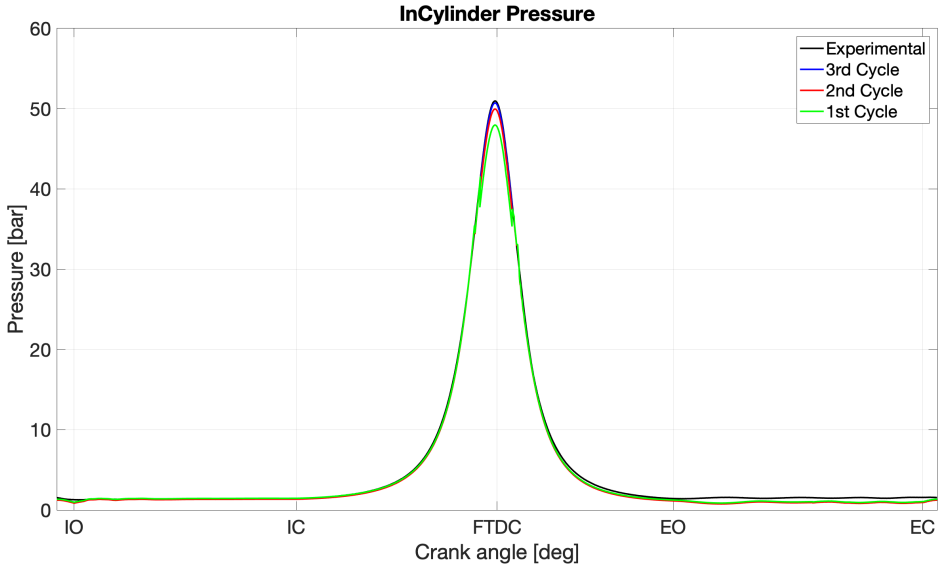


Figure 6.5: Pressure data comparison

The operation must be repeated until the results converge. In this case 3 cycles were necessary to obtain robust pressure traces. The analysis results show that the difference in peak pressure at TDC in the first cycle was 3 bar compared to the experimental results. This difference is reduced to 0.3 bar in the third cycle.
These results show that the model is quite reliable and that by increasing the computing power and the number of cells in the mesh it is possible to get closer and closer to real data.

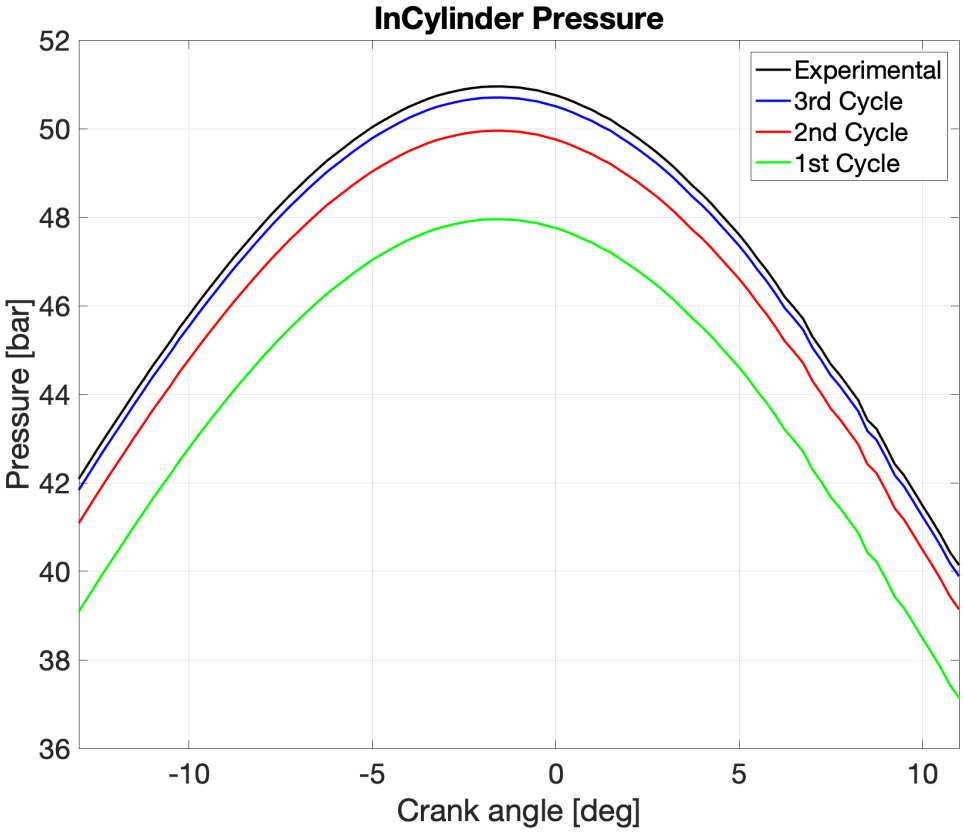


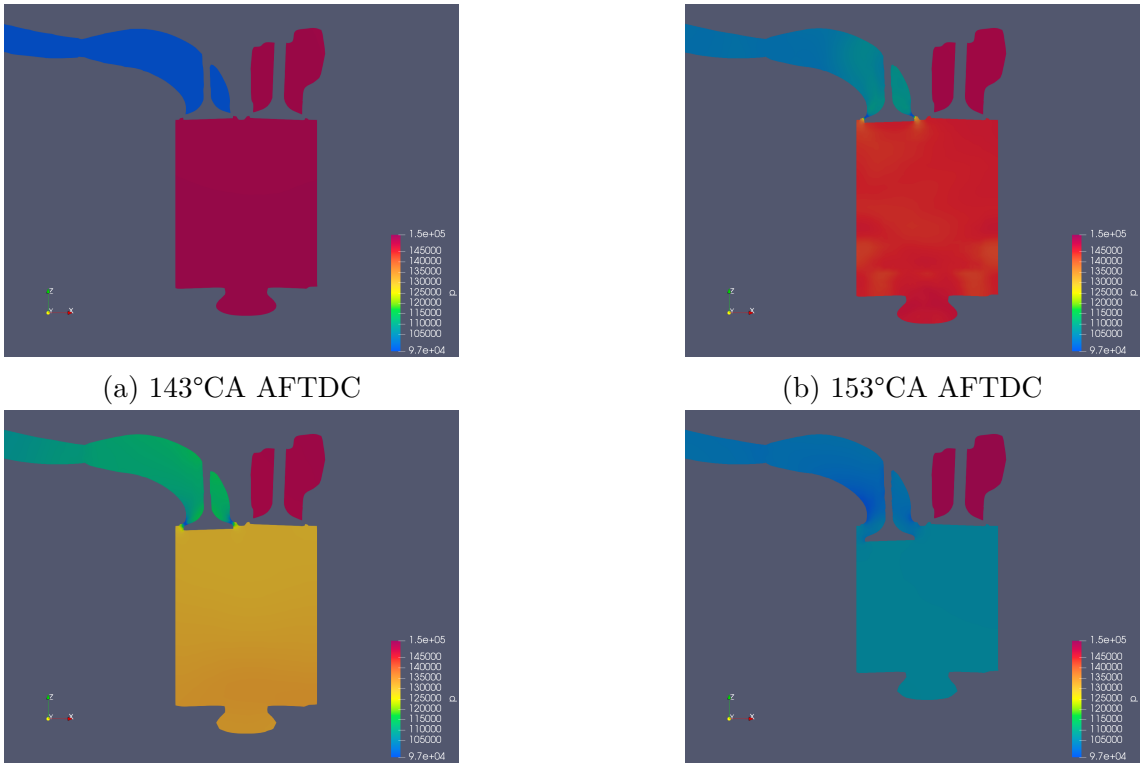
Figure 6.6: Close up of pressure about FTDC of different cycles

6.3 3D Data visualization

While the log files only show averaged values for the entire combustion chamber, a closer look at the flow conditions in the cylinder can be obtained with the help of Paraview. With the help of 3D simulation, local phenomena can be investigated.

6.3.1 Exhaust phase

The exhaust phase (EO) starts at 144° crank angle after TDC, in this instant the valve moves up by 0.7 mm, which in the calculation set in our calculation is the minimum lift to consider the valve open. As shown in Figure (6.7) below, the pressure in the cylinder before the valves are opened is higher than that of the exhaust pipe (6.7a). As soon as the exhaust phase starts, the air in the cylinder starts to flow into the exhaust pipe (6.7b). We can also observe the behaviour (direction and magnitude) of the speed in the Figure (6.8) . Initially directed towards the piston (6.8a) following the movement of the piston changes direction abruptly as soon as the valves open (6.8b). The maximum velocity magnitude, above 100 m/s can be observed during the first degrees of exhaust in which the passage area of the valves is very narrow. The change of flow direction in the cylinder causes a low intensity vortex that dissipates (6.8d), and the phase finally ends without any other particular phenomena.



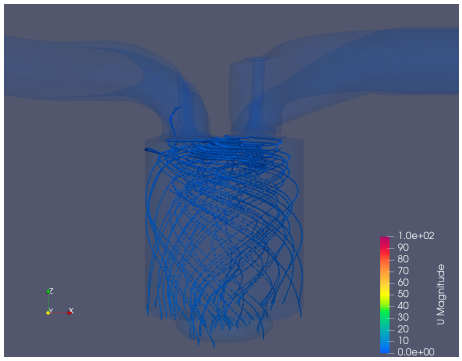
(a) 143°CA AFTDC

(b) 153°CA AFTDC

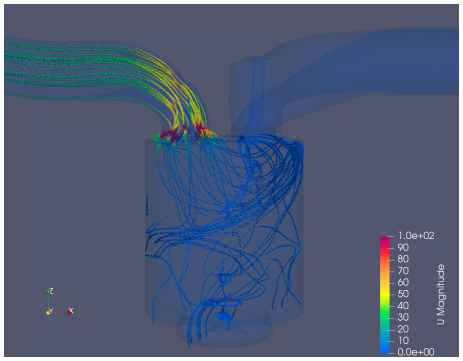
(c) 161°CA AFTDC

(d) 237°CA AFTDC

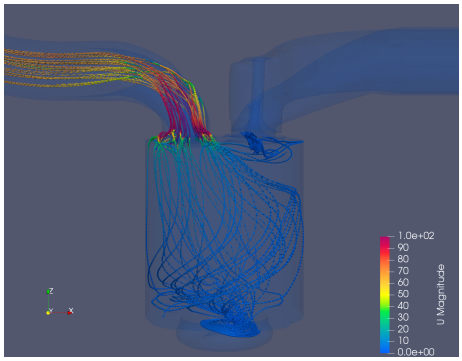
Figure 6.7: Pressure during exhaust phase



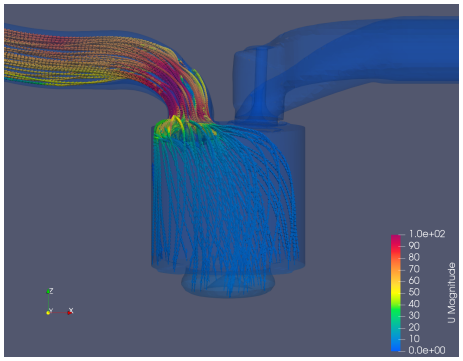
(a) 143°CA AFTDC



(b) 153°CA AFTDC



(c) 161°CA AFTDC

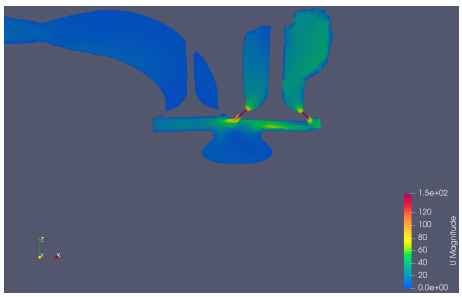


(d) 237°CA AFTDC

Figure 6.8: Velocity during exhaust phase

6.3.2 Intake phase

The intake phase (IO) starts at -346° CA AFTDC, in this instant the valve rises 0.7 mm in the same way as the exhaust phase. The descending piston causes a lower pressure in the cylinder than the intake manifold. This promotes flow entry into the cylinder.
The inlet flow velocity, shown in Figure (6.9), reaches its maximum value of around 150 m/s, immediately after the valves are opened (6.9a). From this moment on, the velocity slowly decreases to zero around the bottom dead center (6.9d).
The study of the motion field that is formed inside the cylinder in this phase of intake is very useful, because some interesting phenomena happen, which will be fundamental for the performance of the engine when fuel is injected into the cylinder.



(a) -340°CA AFTDC

(b) -318°CA AFTDC

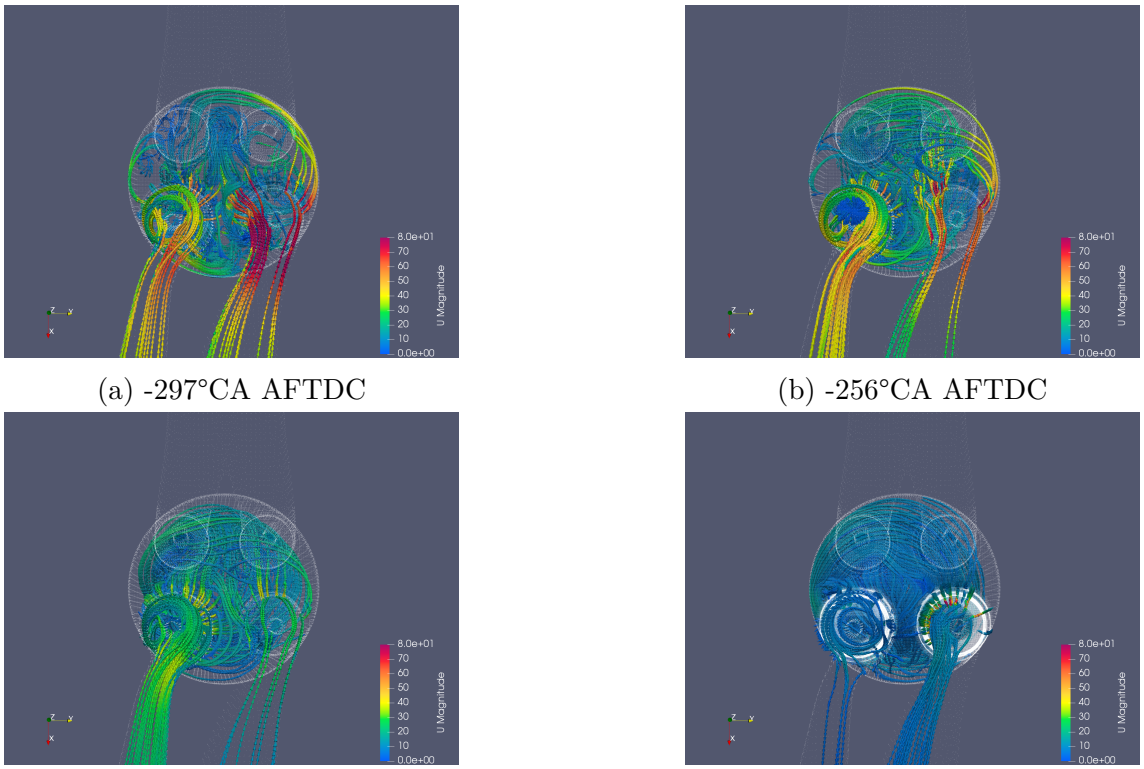
(c) -240°CA AFTDC

(d) -180°CA AFTDC

Figure 6.9: Velocity during intake phase

6.3.3 Charge motion: Swirl

The ability to accurately predict charge behavior is crucial in the initial design phase of an engine. Therefore it is necessary to verify the reliability of this simulation by comparing the results to the literature, previously reported in Chapter 1.
In the Figure (6.9c) we can see the jet-like character of the intake flow, interacting with the cylinder walls and moving piston, creates large-scale rotating flow patterns within the cylinder. These flows usually persist until about the end of the intake stroke, as in (6.9d). The analyzed engine is equipped with a directed duct and an helical duct combined in order to promote the creation of the swirl.
The Figure (6.10) show exactly the expected results: the helical port produce the swirl in the port upstream of the valve and the directed port produces a tangential flow into the cylinder by increasing the flow resistance through that part of the valve open area where flow is not desired.
From the side views in Figure (6.11) we can better observe the evolution of the vortex around the cylinder axis.



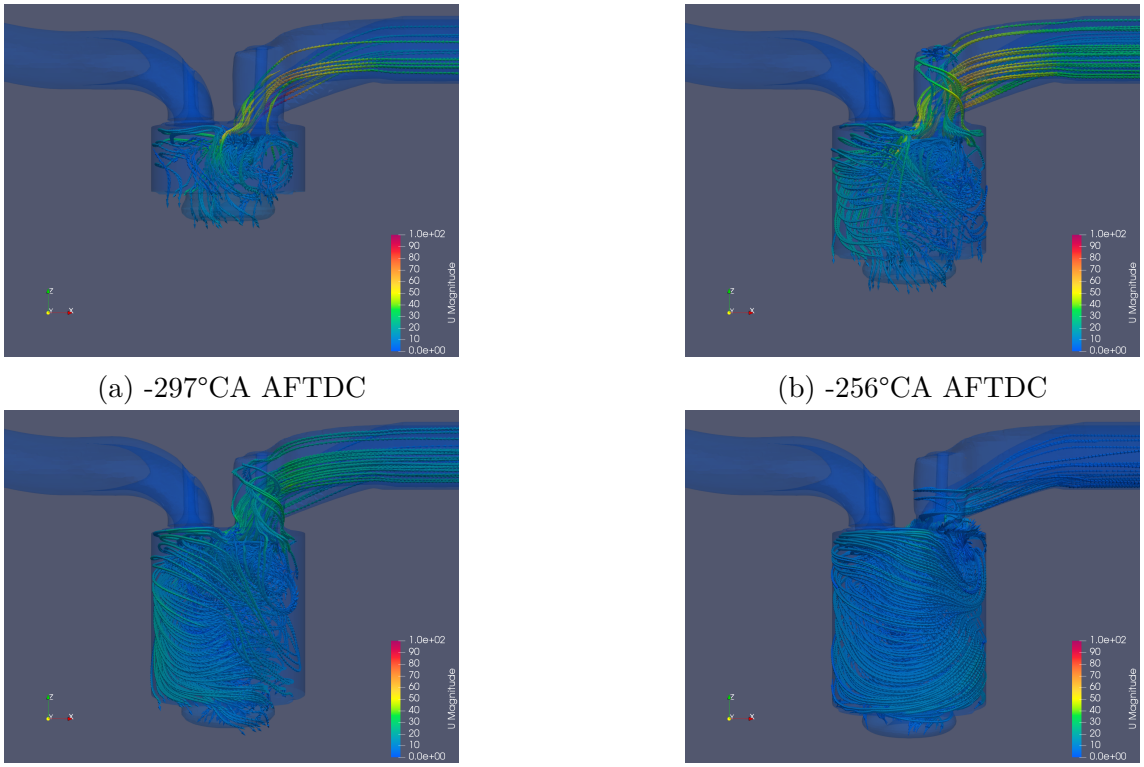
(a) -297°CA AFTDC

(b) -256°CA AFTDC

(c) -220°CA AFTDC

(d) -171°CA AFTDC

Figure 6.10: Top view: Evolution of the swirl during intake phase



(a) -297°CA AFTDC

(b) -256°CA AFTDC

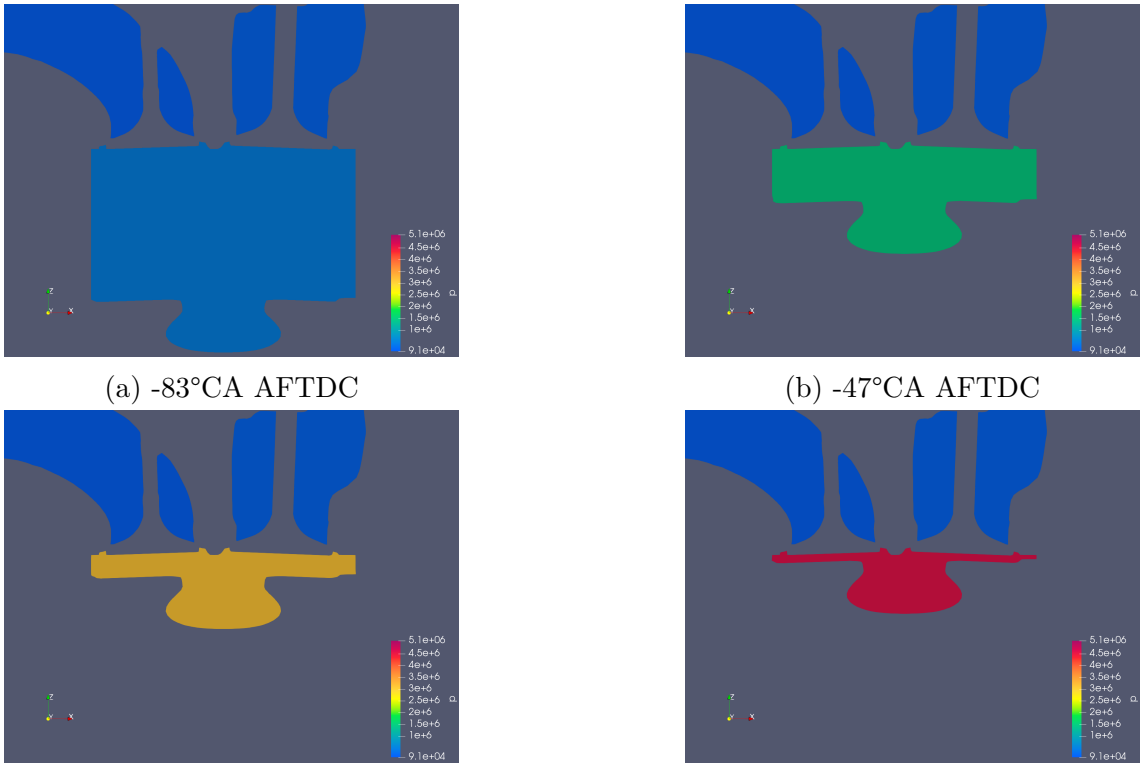
(c) -220°CA AFTDC

(d) -171°CA AFTDC

Figure 6.11: Side view: Evolution of the swirl during intake phase

6.3.4 Compression and expansion phase

The compression phase begins at -163° AFTDC. As the intake valves close, the pressure begins its gradual rise to its maximum value, which is reached when the piston, after having completed the entire compression stroke, reaches the firing top dead center. At this point the pressure begins to decrease, until it reaches, once the piston has made a good part of the expansion stroke, the value it had at the beginning of the cycle.
The pressure (Figure 6.12) starts, at the beginning of the compression phase with a value of 1.46 bar and reaches a maximum value, at the firing top dead center, of 50.7 bar and returns to about the initial value of 1.25 bar, just before the opening of the exhaust valve.
The temperature has not been described up to this point because it is of no interest, since its values are very close to the ambient temperature, since the conduits communicate with the tanks and the latter communicate with the external environment. In this phase instead the temperature value rises due to the increase in pressure and the decrease in cylinder volume. This starts from a value of 437 K at the beginning of compression and reaches a maximum value of 1055 K at the upper dead centre (Figure 6.13).
The velocity, both during the compression phase and during the expansion phase, maintains very low values starting from a value of little more than 20 m/s, up to values of about 8 m/s at the end of the expansion.
As far as the range of motion is concerned, at the beginning of the compression phase we have the presence of the large swirl vortex that remains from the suction phase. This vortex is crushed into the bowl by the rising of the piston. You can observe the evolution of the flow in the following images (Figure 6.14).



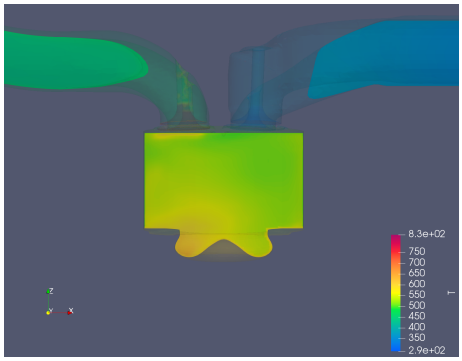
(a) -83°CA AFTDC

(b) -47°CA AFTDC

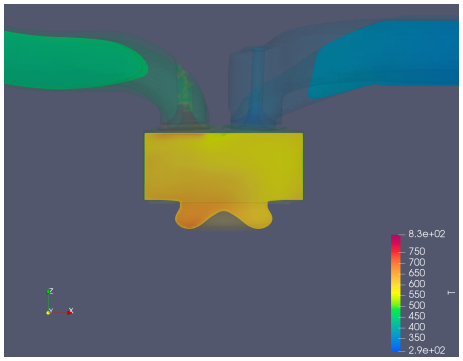
(c) -30°CA AFTDC

(d) FTDC

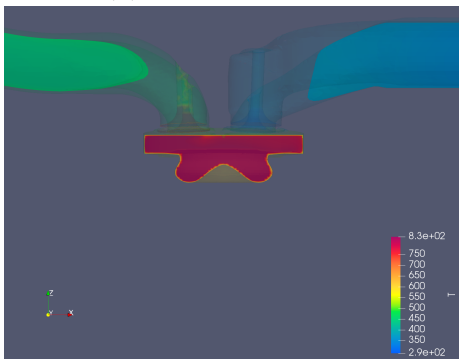
Figure 6.12: Pressure behaviour during compression



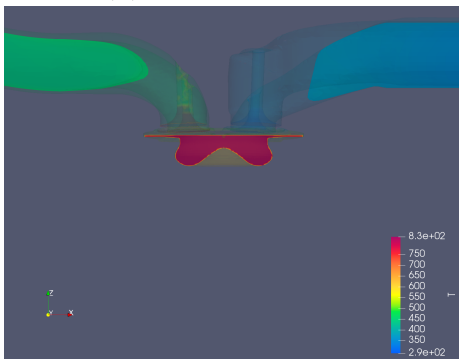
(a) -94°CA AFTDC



(b) -76°CA AFTDC

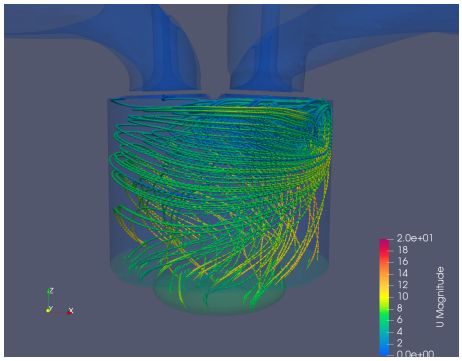


(c) -40°CA AFTDC

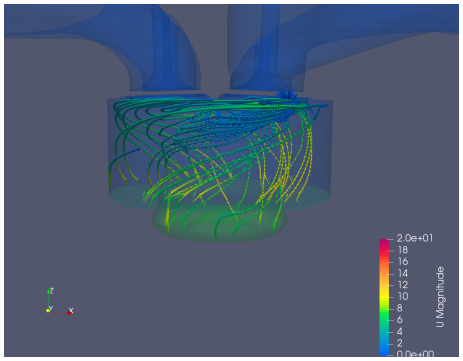


(d) FTDC

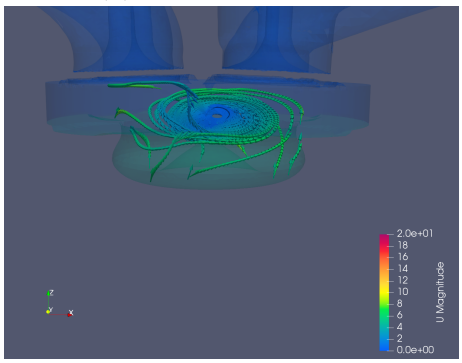
Figure 6.13: Temperature behaviour during compression



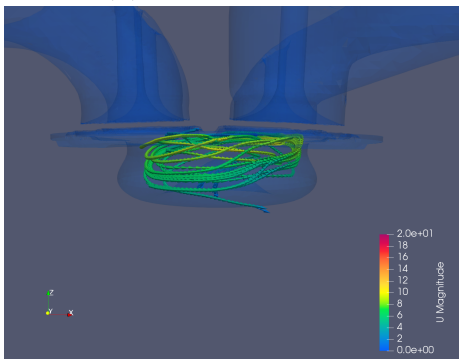
(a) -109°CA AFTDC



(b) -76°CA AFTDC



(c) -40°CA AFTDC



(d) FTDC

Figure 6.14: Side view: Evolution of the swirl during compression phase

6.4 Conclusions

The validation of calculation models has assumed an increasingly important role in recent years. In order to achieve this, increasingly detailed calculation models are needed, which are able to solve even the smallest of flow peculiarities.
In order to support the calculation models, there must be excellent quality meshes that ensure even better results.
The calculation mesh realized in this thesis work required a lot, both in terms of learning the program functionalities and in terms of study, in order to generate a mesh as suitable as possible to the engine morphology.
For the first time, this engine mesh generation work was performed, aimed at simulating an entire motored engine cycle, with the above-described grid generation program. This required a considerable effort, as the geometry of this engine was not simplified in any way, but the results in terms of mesh quality were pretty good, as it was possible to create a grid that allowed a wide adjustability to the movement of the valves and piston, while maintaining a good quality.
The mesh built using the solver snappyHexMesh, described in detail in Chapter 4, has made it possible to carry out successfully: the simulation of the stationary fluxing and the simulation of the entire motored cycle, of the ECN experimental engine. The results obtained from the engine cycle simulation were in line with the experimental values available, both in terms of pressure values.

6.4.1 Limits

To begin with, although all the simulations for the entire cylinder strokes are automated through the correction of the existing bash scripts and additional bash scripts, simulating more than one hundred mesh is still time-consuming process and it leads to ridiculously tiring post-processing tasks since all the results are generated in separate case folders and therefore, have to be viewed separately in ParaView.
The effort to achieve these results remains very high, despite the methods developed to simplify pretreatment. There are still many steps to set the calculation cases and there are many levers to optimize and adjust the simulation.
The snappyHexMesh solver, although accurate, requires meticulous attention and knowledge of each line of the setup code.

6.4.2 Further studies

Simulations could be implemented over the entire cycle, changing the turbulence model and evaluating the effects on the results. It is necessary to specify that, to carry out the simulation of a whole engine cycle, using the calculation in parallel with a last generation 12-processor computer, about 10 days have been used, using the right time step and this mesh with a maximum number of cells equal to 1400000.
By further refining the mesh and changing some details, improvements could be achieved. As regards the results, however, we can say that the calculation model is validated.
The calculation approach presented in this thesis work can be maintained in future applications, as far as this engine is concerned. In this thesis work we have limited ourselves to carry out simulations with a RANS approach, in the future we can think of dealing with LES type simulations, using, as a starting point for the calculation, the results obtained and compare them with the experimental data that will be made available by the ECN in the near future, in order to carry out a further validation of the calculation models used.
In addition, with the IVKsprayEngineFoam solver it is possible to introduce the fuel spray and also the combustion, without making any particular changes to the model created in this thesis. The mesh should only be thickened near the injector and the bowl which are the areas involved in the spray and combustion.

Bibliography

[1] The OpenFOAM Foundation , [Online] *https://openfoam.org/*, 2019.

[2] Wikipedia , [Online] *https://en.wikipedia.org/wiki/OpenFOAM*, 2019.

[3] OpenFOAM: User Guide , [Online] *https://www.openfoam.com/documentation/guides/latest/doc/*, 2019.

[4] CFD Direct, The Architects of OpenFOAM , [Online] *https://cfd.direct/openfoam/user-guide/v6-snappyhexmesh/*, 2019.

[5] ECN, Engine Combustion Network , [Online] *https://ecn.sandia.gov/*, 2019.

[6] CFD-Online , [Online] *https://www.cfd-online.com/Forums/openfoam-meshing/*, 2019.

[7] Dietmar Schmidt, *Engine Combustion and Emissions*, IVK University of Stuttgart, 2017.

[8] H K Versteeg and W Malalasekera, An Introduction to Computational Fluid Dynamics The Finite Volume Method (Second Edition), Pearson Prentice Hall, 2007.