SLPO (Simulation Launcher & Pareto Optimization) algorithm development for fluid dynamics multisimulations (OpenFOAM) and optimal case identification

Mateu Magem Ribas

Directed by Enric Bonet and Francesc Perez.

Degree of industrial electronics and automatic engineering
Departament d'Enginyeria Minera, Industrial i TIC
Universitat Politècnica de Catalunya (UPC)

Abstract

This project presents the development of the SLPO (Simulation Launcher & Pareto Optimization) algorithm, designed to automate fluid dynamics simulations and optimize design parameters using open-source computational fluid dynamics tools. The system studied is a simplified air injector, modeled as a curved pipe with a small inlet. Using Monte Carlo sampling, random parameter sets are generated, and simulations are conducted in OpenFOAM with the icoFoam solver for incompressible, isothermal fluid flow. Pareto optimization is applied to perform multiobjective optimization, identifying designs that balance the two competing objectives of minimizing inlet pressure and material cost. The algorithm explores the design space automatically and identifies Pareto optimal solutions. The workflow, built on Python, Gmsh, and PyFoam, offers a scalable approach to multiobjective optimization in CFD. While the case model is considered too simplisitic, the methodology demonstrates significant potential for more complex and practical engineering applications.

Contents

Chapter 1

Introduction

1.1 Objectives

In many cases, real-world systems are composed of components with intricated geometries and are subject to several operational requirements as well as various environmental factors. The search space of all design solutions rapidly becomes a vast higher dimensional space as designs have many degrees of freedom and tunable parameters with their corresponding hard to predict behaviours. Encountering optimal design solutions for such systems is generally a non trivial open problem with practical applications and it is industrially and economically sought-after. This is why non-deterministic algorithms involving Monte Carlo sampling, such as the one presented in the current work, are used. These algorithms do not guarante to find the best existing solution but allow to navigate the solution space automatically to discover localy optimal designs. New discovered designs can be continuously improved until no more meaningful improvements are achieved. These techniques require time and computing power but automate the design process and navigate its uncertainty providing refined results.

The objective of the present study is the development of a program structure for running an arbitrary amount of fluid dynamics simulations with randomized parameters, extracting simulation results and identifying the optimal simulation case given a set of constraints such as input pressure and material cost. A simplified case study is used to exemplify and make concrete the proposed structure. While physically too simplistic, the example chosen contains all relevant features in a general fluid dynamics simulation. Hence, the proposed structre can be scaled up to tackle more realistic cases. The proposed method can be easily adapted to solve more complex design problems requiring fluid dynamics simulations, material and cost considerations, and multiobjective optimization.

Gathering the data of many automated simulation cases, the ones which optimize the target characteristics can be found. These optimal cases are known as Pareto optimal solutions. It is looked for the reduction of the material cost of the component and, consequently, its manufacturing cost, and the reduction of the inlet's input pressure, which affects the required power of the input pump and, thus, the operational cost

during the life span of the system.

1.1.1 Planning

The essential tasks to complete this project's objectives are:

1. Define the model and its design requirements. Establish the limitations and conditions that the model must adhere to. Make a list of the parameters which describe the model both physical, geometrical, and simulation wise.
2. Develop methods to randomly generate and modify these input parameters. Succeed in creating or updating the necessary simulation files and develop a template case as a reference for other simulations.
3. Using the geometrical parameters, create or adjust the meshes used in the simulations.
4. Run the simulation's solver for each case to obtain results.
5. Implement control mechanisms and logic for handling multiple simulations.
6. Extract and collect the results from all the simulations.
7. Find the optimal cases using the Pareto optimization algorithm to identify the optimal case amongst the results.
8. Present the results.

Chapter 2

Fundamental equations of fluid mechanics

This chapter introduces the reader to the background physical fundations and theoretical fluid mechanics used in this study. The content of this chapter is based on the chapter "Basic Conservation Laws" from the book "Fundamental Mechanics of Fluids" [1]. Such book has been chosen because it provides a general introduction to the fundamental concepts of fluid mechanics and the derivation of its basic equations from the ground up. Other references have been used when necessary.

2.1 Statistical and continuum approaches

There are two main methods conventionally used to model and develop the equations ruling the motion and the physical behaviour of fluids.

The first one is the statistical approach which considers fluids from the molecular point of view. It applies fundamental mechanic principles and probability theory to predict the emergent macroscopic properties of the fluids. The second method uses the concept of continuum attempting to describe the fluid behaviour in a macroscopic non-local manner.

It is notable that both methods present their own limitations: the statistical approach works with light gases but is incomplete for heavier gases and liquids and the continuum approach requires the fluid to contain a sufficiently large number of particles.

2.2 Fluid properties

Constitutive equations are the equations relating the various thermodynamical properties that characterize the fluid. Some of the most important fluid properties are density and viscosity. Density is the amount of mass contained in a given volume of fluid. The viscosity of a fluid indicates the fluid's resistance to movement and deformation and it is related to the internal forces within the fluid. Both properties are associated to

temperature as viscosity depends on density and density is connected to temperature by the ideal gas law. This same law is the one used to compute the density value in a specified case.

Considering that in the current scenario the density is constant, the fluid must be incompressible as well. This is because the quantity of fluid particles in a given volume must remain constant at all times preventing the fluid from suffering any compression or decompression process.

2.2.1 Incompressible flow

Incompressibility in fluid mechanics refers to a condition where the density of a fluid element remains constant even when the pressure within the fluid changes. When a flow is incompressible, the volume of the fluid parcel does not change with pressure variations, leading to a constant density.

The assumption of incompressibility allows the use of simplified forms of the Navier-Stokes equations, specifically designed for incompressible flows, which are less complex than those for compressible flows as can be seen in 2.22. This simplification is commonly applied in scenarios where fluid density changes are negligible, such as in water flow or low-speed airflows. The concept is essential in fluid dynamics because it simplifies the analysis and solutions of fluid flow problems, making it a practical assumption to simplify mathematical models and computations [2].

Since the fluid of the study case is considered to be air at low speeds, it is reasonably assumed to be incompressible which permits approaching the problem with a simpler solver.

2.2.2 Ideal gas law

The fluid's density behaves as an ideal gas since the studied fluid is air at non-extreme conditions and all discrepancies with ideal gas behaviour are neglected. Realizing that approximation, the ideal gas law becomes one of the state equations of the fluid.

$$pV = nRT \tag{2.1}$$

where
p: pressure $[Pa]$
V: volume $[m^3]$
n: quantity of substance $[moles]$
R: ideal gas constant $[molesK/Pa/m^3]$
T: temperature $[K]$

The quantity of substance can be substituted by the mass of gas divided by the molar mass of the gas.

$$p = \rho R_{specific} T \tag{2.2}$$

where
m: mass $[kg]$

M: molar mass $[moles/kg]$
ρ: density $[kg/m^3]$
$R_{specific} = \frac{R}{M}$: specific gas constant $[kgK/Pa/m^3]$

2.2.3 Kinematic viscosity

The kinematic viscosity of the fluid is constant as the fluid is isothermic and the temperature range considered excessively high. Kinematic viscosity is computed as follows:

$$\nu = \frac{\mu}{\rho} \tag{2.3}$$

where
ν: kinematic viscosity $[m^2/s]$
μ: dynamical viscosity $[Pas = Ns/m^2 = kg/m/s]$
ρ: density $[kg/m^3]$

If the ideal gas law is applied:

$$\nu = \frac{\mu \cdot R_{specific} \cdot T}{p} \tag{2.4}$$

2.3 Turbulence and Reynolds number

Turbulence is the mixing of the layers of a fluid as it flows producing eddies. It's main causes are obstructions and sharp edges and high speed induced drag. The Reynolds number is an dimensionless indicator of fluid turbulence [3]. For flow in a pipe, it can be computed through the following equation:

$$Re = \frac{2\rho v r}{\mu} = \frac{v r}{\nu} \tag{2.5}$$

where
Re is the Reynolds number,
ρ is the fluid's density,
v is the fluid's speed,
r is the pipe radius,
μ is the dynamical viscosity, and
ν is the kinematic viscosity [4].

Laminar and tubulent flows can be characterized using Reynolds numbers. Only laminar flow is considered in this study as the solver employed solely accounts for this type of flow.

2.3.1 Laminar flow

Laminar flow is a type of fluid flow characterized by smooth, orderly, and parallel layers of fluid that move in a consistent direction with minimal mixing. In this flow regime, the fluid particles follow streamlined paths. This type of flow occurs at Reynolds numbers below 2000, where viscous forces are dominant. Laminar flow is commonly observed in applications involving low-speed fluid movement, such as blood flow in capillaries or oil flow in pipelines [3].

2.3.2 Turbulent flow

The turbulent flow regime contrasts with laminar flow as it is characterized by chaotic eddies and mixing. Flow is considered to be turbulent when its Reynolds number is above 3000 [3].

2.4 Conservation of mass

The mass conservation principle states that, if a specific mass of fluid of arbitrary volume V is chosen, although its volume or shape may change as it flows, the amount of mass will remain constant.

$$\frac{D}{Dt}\int_V \rho \, dV = 0 \tag{2.6}$$

where ρ is the fluid's density.
The Reynolds transport theorem expresses the arbitrary control volume in terms of the coordinates of the system containing it [5].

$$\frac{D}{Dt}\int_V \alpha(t) \, dV = \int_V \left[\frac{\partial \alpha}{\partial t} + \vec{\nabla}(\alpha \vec{u})\right] dV \tag{2.7}$$

where α corresponds to any property of the fluid.
Using the Reynolds transport theorem with ρ as the fluid property, the equation can be converted to a volume integral with Eulerian derivatives as integrands.

$$\int_V \left[\frac{\partial \rho}{\partial t} + \nabla(\rho \vec{u})\right] dV = 0 \tag{2.8}$$

Since the volume is arbitrary, the integrand must be zero to satisfy the equation for all possible volumes.

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \vec{u}) = 0 \tag{2.9}$$

Equation 2.9 is known as *continuity equation* as it implies that both mass is conserved and velocity is continuous. This derivation corresponds to a single-phase fluid.

2.5 Conservation of mass in incompressible fluids

Incompressible fluids pose a special case as their density does not vary. Because of that a given amount of mass will always occupy the same amount of space. Setting the density ρ constant in the continuity equation (2.9), and keeping in mind that density must always be positive for there to be a fluid, the expression below is obtained.

$$\nabla \vec{u} = 0 \tag{2.10}$$

This equation development results in the continuity equation for incompressible fluids.

2.6 Conservation of momentum

The concept of momentum emerges from Newton's second law of motion concluding that net force acting over a mass m is equivalent to a change in momentum [6].

$$\sum \vec{F}_{net} = m\vec{a} = m\frac{d}{dt}\vec{v} = \frac{d}{dt}m\vec{v} = \frac{d}{dt}\vec{p} \tag{2.11}$$

where $\vec{a}$ is the acceleration, $\vec{v}$ is the velocity, and $\vec{p}$ is the momentum. This is applicable in the case of a fluid since the mass is arbitrarily chosen and considered to be constant.

The external forces acting on a fluid can be either body forces, acting on the entire mass of the fluid, or surface forces, acting only on its surface. If $\vec{f}$ is the vector of the resultant of the body forces per unit of mass, the net external body force acting on a mass of volume V is $\int_V \rho\vec{f}\,dV$. Moreover, if the $\vec{P}$ is the vector of the resultant surface force per unit of area, the net external surface force over the surface S of the volume V is $\int_S \vec{P}\,dS$. Given these definitions, the equation of the variation in momentum 2.11 can be expressed as:

$$\frac{D}{Dt}\int_V \rho\vec{u}\,dV = \int_S \vec{P}\,dS + \int_V \rho\vec{f}\,dV \tag{2.12}$$

Stress is a physical magnitude used to quantify deformation forces as force divided by area units. The following figure represents the stress tensor, also known as Cauchy stress tensor [7] , which is composed of nine directional components σ_{ij} and normal unit vectors n_i. Although, the stress tensor is displayed using a cube, such cube is in reality equivalent to a infinitesimally small fluid element since the side of the cube tends to zero.

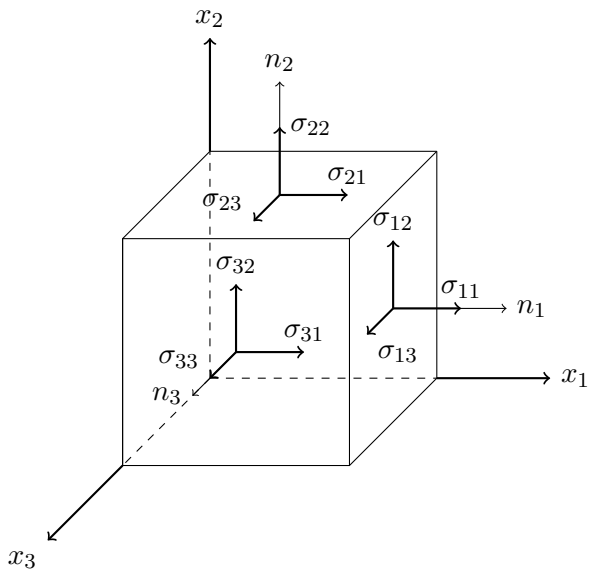


Figure 2.1: Components of stress σ_{ij} and normal unit vectors n_i

Equation 2.12 can be rewritten in tensorial form and the resultant surface force P expressed in terms of stress by the relation $P_i = \sigma_{ij} n_i$.

$$\frac{D}{Dt} \int_V \rho u_j \, dV = \int_S \sigma_{ij} n_i \, dS + \int_V \rho f_j \, dV \tag{2.13}$$

Reynolds transport theorem from equation 2.7 is applied to the change in momentum in tensorial form and the surface integral is converted to a volume integral thanks to the Gauss theorem.

$$\int_V \left[\frac{\partial}{\partial t}(\rho u_j) + \frac{\partial}{\partial x_k}(\rho u_j u_k) \right] dV = \int_V \frac{\partial \sigma_{ij}}{\partial x_i} \, dV + \int_V \rho f_j \, dV \tag{2.14}$$

Once all terms of the expression are integrals of the same volume V, all integrals can be merged.

$$\int_V \left[\frac{\partial}{\partial t}(\rho u_j) + \frac{\partial}{\partial x_k}(\rho u_j u_k) - \frac{\partial \sigma_{ij}}{\partial x_i} - \rho f_j \right] dV = 0 \tag{2.15}$$

As the volume V is arbitrary, the integrand must be zero to fulfill the equation for all possible volumes.

$$\frac{\partial}{\partial t}(\rho u_j) + \frac{\partial}{\partial x_k}(\rho u_j u_k) - \frac{\partial \sigma_{ij}}{\partial x_i} - \rho f_j = 0 \tag{2.16}$$

The term $\frac{\partial}{\partial x_k}(\rho u_j u_k)$ can be expanded considering the product of ρu_k and u_j.

$$\rho \frac{\partial u_j}{\partial t} + u_j \frac{\partial \rho}{\partial t} + u_j \frac{\partial}{\partial x_k}(\rho u_k) + (\rho u_k) \frac{\partial u_j}{\partial x_k} = \frac{\partial \sigma_{ij}}{\partial x_i} + \rho f_j \tag{2.17}$$

Additionally, the terms $u_j \frac{\partial \rho}{\partial t}$ and $u_j \frac{\partial}{\partial x_k}(\rho u_k)$ correspond to the continuity equation 2.9 and they are equal to zero.

$$\rho \frac{\partial u_j}{\partial t} + (\rho u_k) \frac{\partial u_j}{\partial x_k} = \frac{\partial \sigma_{ij}}{\partial x_i} \tag{2.18}$$

This equation is still equivalent the result extracted from Newton's law of acceleration 2.11 in which the variation of momentum of a mass is equivalent to the net sum of forces acting upon that mass.

2.7 Navier-Stokes equations

The Navier-Stokes equations, formulated by Claude-Louis Navier in 1822 and later refined by George Stokes, are fundamental to the modeling of fluid dynamics. They describe the motion of viscous fluid substances and incorporate the effects of viscosity, making them applicable to real-world fluid behaviour beyond idealized inviscid models. These equations have significant implications across various fields, such as aerodynamics, weather forecasting, and engineering. They are also a key element in the field of computational fluid dynamics (CFD), enabling the simulation of complex fluid interactions in environments ranging from industrial processes to natural phenomena [8].

Mathematically, the equations are a set of second order partial differential equations. They are obtained when combining the equation of conservation of momentum (2.18) and the next constitutive equation of stress for Newtonian fluids (2.19). Such equation of stress, also known as stress tensor, can be expressed as:

$$\sigma_{ij} = -p\delta_{ij} + \lambda \delta_{ij} \frac{\partial u_k}{\partial x_k} + \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \tag{2.19}$$

where the coefficients λ and μ correspond to viscosity coefficients and must be determined experimentally. The coefficient λ is known as second viscosity coefficient and the coefficient μ as dynamic viscosity.

This expression can now be replaced in the equation of momentum conservation to finally put together the Navier-Stokes equations. As the only nonzero terms are the ones where $i = j$, i is substituted by j for simplification.

$$\rho \frac{\partial u_j}{\partial t} + (\rho u_k) \frac{\partial u_j}{\partial x_k} = -\frac{\partial p}{\partial x_j} + \frac{\partial}{\partial x_j} \left(\lambda \frac{\partial u_k}{\partial u_k} \right) + \frac{\partial}{\partial x_i} \left[\mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \right] \tag{2.20}$$

This equation is in reality a set of three scalar equations for the three possible values of j.

2.8 Navier-Stokes equations for incompressible fluids

Given an incompressible fluid whose dynamic viscosity is constant, the expression can be further simplified as the term containing the secondary viscosity coefficient is null

and the viscous-shear terms is converted to:

$$\frac{\partial}{\partial x_i}\left[\mu\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)\right]=\mu\left[\frac{\partial}{\partial x_j}\left(\frac{\partial u_i}{\partial x_i}\right)+\frac{\partial^2 u_j}{\partial x_i \partial x_i}\right]=\mu\frac{\partial^2 u_j}{\partial x_i \partial x_i} \tag{2.21}$$

Substituting the term, the formulation of the Navier-Stokes equations for incompressible fluids is obtained.

$$\rho\frac{\partial u_j}{\partial t}+(\rho u_k)\frac{\partial u_j}{\partial x_k}=-\frac{\partial p}{\partial x_j}+\mu\frac{\partial^2 u_j}{\partial x_i \partial x_i} \tag{2.22}$$

2.9 Governing equations of incompressible Newtonian fluids in isothermal conditions

The set of governing equations for incompressible Newtonian fluids in isothermal conditions consists of four equations: one continuity equation (2.23), and three Navier-Stokes equations (2.24). These equations are used to solve for four unknowns: pressure and three velocity components. These equations represent fundamental principles: the continuity equation ensures mass conservation, and the Navier-Stokes equations account for momentum and viscous forces. Their combined analysis allows for the comprehensive modeling of fluid behaviour in various conditions. According to the requirements of the current case, energy related equations are not rellevant as the studied fluid is isothermal. Moreover, since the fluid in question is air, it is assumed to behave as an ideal gas. This implies that the state equation corresponds to the ideal gas law. Equations of state relate pressure, density, and temperature are not taken into account in the simulation environment since the considerd fluid is incompressible and in isothermal conditions.

$$\frac{\partial u_k}{\partial x_k}=0 \tag{2.23}$$

$$\rho\frac{\partial u_j}{\partial t}+(\rho u_k)\frac{\partial u_j}{\partial x_k}=-\frac{\partial p}{\partial x_j}+\frac{\partial}{\partial x_j}\left(\lambda\frac{\partial u_k}{\partial u_k}\right)+\frac{\partial}{\partial x_i}\left[\mu\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)\right]+\rho f_j \tag{2.24}$$

The density in the momentum conservation equation (2.24) is assumed constant, as the pressure variations are assumed to be low. Its value is taken using the ideal gas law (2.2) using the pressure and temperature at a reference point. To improve clarity, the continuity equation and the Navier-Stokes equations can be rewriten in divergence notation:

$$\nabla\cdot\vec{u}=0 \tag{2.25}$$

$$\frac{\partial}{\partial t}(\vec{u})+\nabla\cdot(\vec{u}\otimes\vec{u})-\nabla\cdot(\nu\nabla\vec{u})=-\nabla p \tag{2.26}$$

where
u is the velocity $[m/s]$,
p is the kinematic pressure $[m^2/s^2]$, and
ν is the kinematic viscosity $[m^2/s]$.

Such equation corresponds to the vector form of the Navier-Stokes equation for incompressible fluids including the viscous term $\nabla \cdot (\nu \nabla u) = \nu \nabla^2 \vec{u} = \frac{\mu}{\rho} \frac{\partial^2 u_j}{\partial x_i \partial x_i}$ (equation 2.22 in component form).

2.10 Initial conditions

Initial conditions are the necessary constraints for solving the differential equations that describe the behaviour of a system at the start of a process. Initial conditions define the state of the system at an initial time, such as the velocity, pressure, or temperature distribution throughout the fluid at the start of the simulation. They are essential in configuring a simulation case as they influence its entire causal trajectory.

2.11 Boundary conditions

Boundary conditions refer to the constraints necessary to solve differential equations that describe physical phenomena. They define the behaviour of a system at the boundaries of its domain. For instance, in fluid dynamics, boundary conditions can specify the velocity and pressure of a fluid at the walls of a container.

Boundary conditions can be classified into different types such as Dirichlet conditions, which specify the value of a function on a boundary, and Neumann conditions, which specify the value of the derivative of a function on a boundary. Properly defining boundary conditions is crucial in simulations and mathematical modeling as they influence the accuracy and stability of numerical solutions [9]. This case environment defines Dirichlet conditions as pressure and velocity are specified.

Chapter 3

Case study

3.1 Case introduction

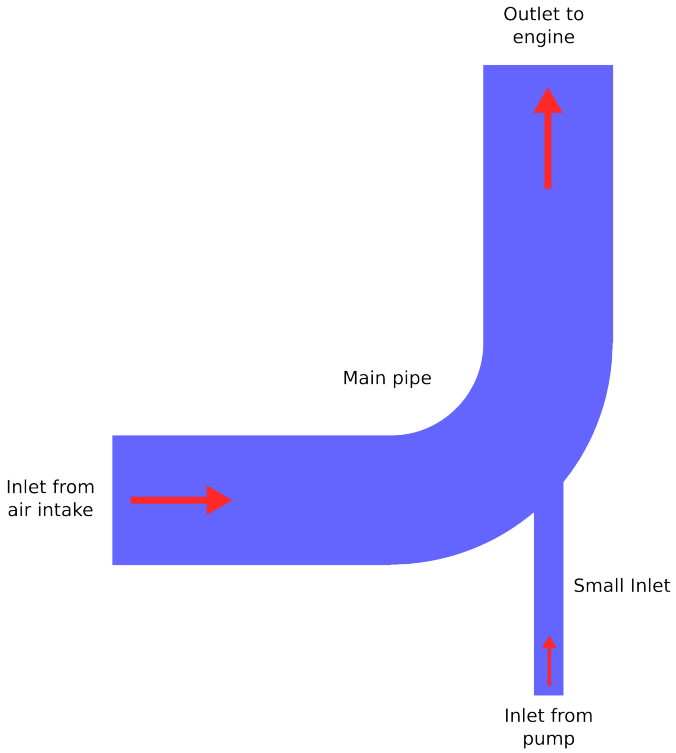


Figure 3.1: Case geometry and airflow illustration.

The present case study is a simplification of an air injector. An air injector's function is to mix the atmosferic air from the air intake with fuel to further on produce combustion inside the engine. Fuel is injected by a pump. This case's injector model only contains

incompressible air as a fluid at a uniform temperature. The presence of fuel or any thermal effects are ignored for the sake of keeping a simpler simulation environment. The model is composed of a main curved pipe forming a 90° turn and a small straight inlet at the center of the turn, aligned with the outlet. The main pipe would be the airflow conduit connecting the intake with the engine. The inlet of a smaller diameter than the main pipe would be the fuel input channel which in this case is accounted as air. Such shape is illustrated in figure 3.1.

The case contributes limited value in its current form. It has been chosen to demonstrate a solution finding methodology as it is simple enough to kick-start the project and, simultaneously, substantial enough to build a fundamental method to face problems of greater complexity.

3.2 Input parameters

These are the parameters given to each an every simulation ran. There are four types of input parameters:

a) Simulation parameters
These are used to control high level simulation behaviour such as simulation time, time interval and write interval.

b) Geometrical parameters
Geometrical parameters modify the mesh's geometry to be build.

c) Boundaries
Boundaries is the list of boundaries set in the mesh and used to specify boundary conditions in the simulation's initial conditions setup.

d) Physical parameters
Physical parameters are the parameters defined in the simulation's initial boundary conditions.

3.2.1 Simulation parameters

The simulation parameters define the temporal bounds and output control for a simulation. The parameters accounted for the implemented program are:

- `starttime` is the starting time of the simulation, set to 0.
- `endtime` is the ending time of the simulation.
- `deltat` is the time step of the simulation. This is the increment by which the simulation time advances in each step. A smaller deltat can lead to more accurate results but requires more computations.

- `writecontrol` determines how often simulation data is written to output. It can be set to "runTime", indicating that the data will be written based on simulation time, or set to "timeStep", would base data writing on the number of steps.
- `writeinterval` Specifies the interval at which data is written.

These parameters are rellevant to ensure simulation stability and the convergence between time steps. As it has been realized during tests that smaller `deltat` values increase the simulation's accuracy but can lead to long compute times. Using small `writeinterval` alongside increases memory usage.

Parameter	Value
starttime	0
endtime	0.01
deltat	0.0001
writeControl	timeStep
writeinterval	5

Table 3.1: Default values for the simulation parameters

3.2.2 Geometrical parameters

Geometrical parameters are the variables used to parametrize the shape of the figure described. These are used afterwards to build meshes with programatically modifiable geometry. All units used correspond to units from the international system of units, that is, meters for length and radians for angle.

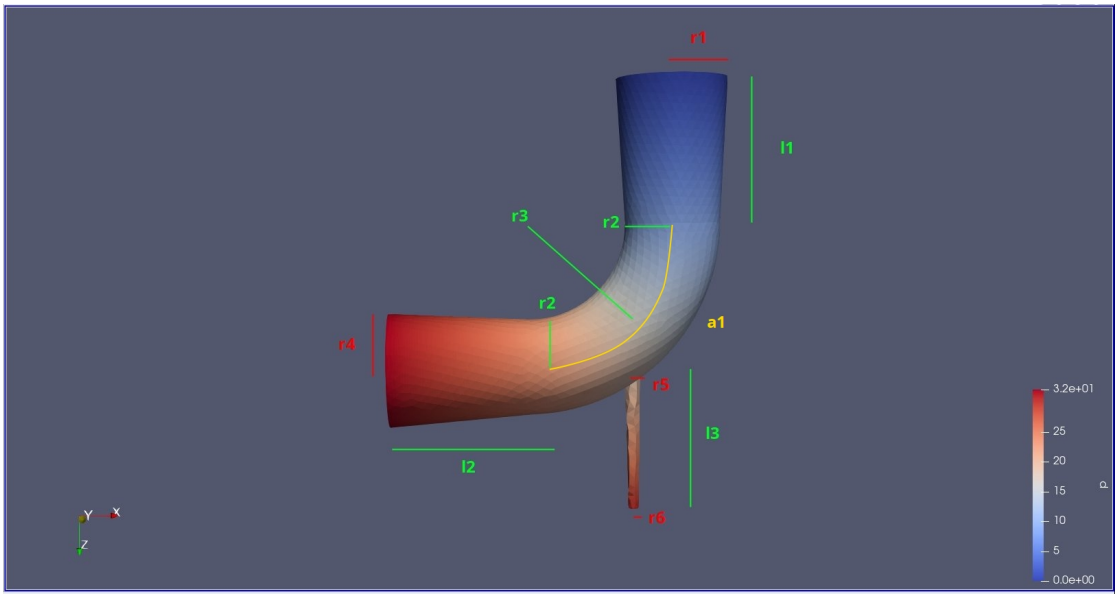


Figure 3.2: Paraview capture with geometrical parameters.

The geometrical parameters necessary to describe the elbow's geometry can be seen in figure 3.2 where:

- (ox, oy, oz) is the origin point,
- $r1$ is the outlet pipe radius,
- $a0$ is the arc length of the outlet pipe (since it is a circle, it is $2\pi = 360°$),
- $l1$ is the length from the outlet to the elbow turn,
- $a1$ is the arc length of the elbow,
- $r2$ is the radius of the elbow pipe,
- $r3$ is the turning radius of the elbow,
- $l4$ is the length of the inlet pipe,
- $r4$ is the radius of the inlet pipe,
- $r5$ is the radius of the intersection between the small inlet and the elbow pipe,
- $l3$ is the length of the small inlet and
- $r6$ is the radius of the small inlet.

The following constraints have been imposed to fully define the geometry and reduce its degrees of freedom:

a) The origin point is defined as $(ox, oy, oz) = (0, 0, 0)$.

b) The lenght of all inlet and outlet pipes is $l1 = l2 = l3 = 0.125\ m = 125\ mm$. The lengths of the pipes is fixed as they do not fluid flow at the element.

c) The arc length of the elbow turn is $a1 = \frac{\pi}{2} = 90°$.

d) The outlet radius $r1$ is a random value between $0.02\ m = 20\ mm$ and $0.045\ mm = 45\ mm$.

e) Both elbow pipe and inlet radiuses are equal ($r2 = r4$) and hold a random value between $0.02\ m = 20\ mm$ and $0.045\ m = 45\ mm$.

f) The elbow turn radius $r3$ has half of the length of the pipes $\frac{l1}{2} = 0.0625\ m = 62.5\ mm$.

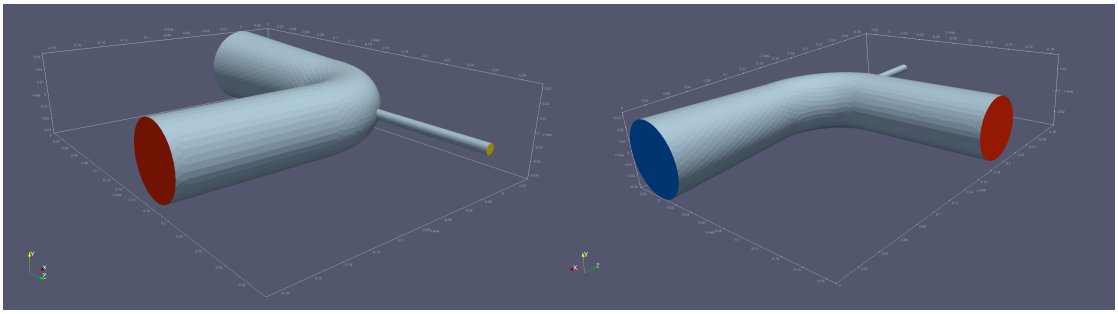
g) Both small inlet radiuses $r5$ and $r6$ are equal and have a random value between $0.0025\ m = 2.5\ mm$ and $0.005\ mm = 5\ mm$.

These constraints are case dependant and can be modified. It must be noted that all given values are reasonable for the set out case, although this work's intention does not pursue high precision. The number of free variables increases exponentially the calculation time for the optimization procedure, and it is for this reason that constraints are applied. The free variables left are: $r1$, $r2 = r4$, and $r5 = r6$.

3.2.3 Boundaries

Boundary are regions in the mesh geometry where boundary conditions are specified to model the behaviour of fluid flow or other physical phenomena near surfaces. The ones significant for the purposes of this simulation are:

1) Big inlet (surface, colored in images 3.3a and 3.3b)
2) Small inlet (surface, colored in image 3.3a)
3) Big outlet (surface, colored in image 3.3b)
4) Walls (surface, colored in images 3.3a and 3.3b)



(a) (b)

Figure 3.3: 3D view of the elbow with colored boundaries.

3.2.4 Boundary Conditions

Boundary conditions define the behaviour of the fluid on the previously described boundaries.

1) Pressure at the big inlet is atmosferic.
2) Pressure at the small inlet is unkown. It is determined by the resolution of the simulation and it corresponds to the fuel pressurizer pump.
3) Pressure is sub-atmosferic at the outlet as it would be caused by the movement of the hypothetical piston at the end of the injector.
4) Walls have the boundary condition `noSlip` which means that the fluid layer adjacent to the solid surface cannot move. This means that velocity is zero over solid surfaces.

Inlet velocities are respectively imposed using the `fixedValue` condition to assigne a constant and uniform velocity value.

3.2.5 Physical parameters

Physical parameters are the parameters which define some physical magnitude such as temperature, pressure or speed. All units used correspond to units from the international system of units, that is, kelvins for temperature, pascals for pressure, meters for length and seconds for time. The physical constraints defined by the case are:

a) The temperature of the system has a random value ranging from $-5°C = 268.15\ K$ to $40°C = 313.15\ K$.

b) The pressure at the outlet varies from atmosferic pressure ($\approx 101325\ Pa$) to 300 units below.

c) The flow at the big inlet ranges between 0.2 and 0.3 m^3/s. As the solver contemplates speed and not flow, the flow at the big inlet is divided by the area of the inlet surface to obtain the fluid's velocity.

d) The flow at the small inlet is the flow at the big inlet multiplied by a random factor between 0.065 and 0.07. Forcing this relationship establishes a reasonable proportion between big and small inlet flows and simplifies the amount of independent parameters to generate.

3.3 Parameter storage

In order to favor data accessibility and transparency, the parameter dictionary for every simulation case run is stored as a JSON file. The file contains all input parameters explained in this section for the specific run. A randomly generated example is provided below for the reader's comprehension.

```
{
  "starttime": 0,
  "endtime": 0.01,
  "deltat": 0.0001,
  "writeControl": "timeStep",
  "writeinterval": 5,
  "boundary_conditions": [
    "big-inlet",
    "big-outlet",
    "small-inlet",
    "walls",
    "fluid"
  ],
  "ox": 0,
  "oy": 0,
  "oz": 0,
  "a0": 6.28318531,
  "a1": 1.57079633,
  "l1": 0.125,
  "l2": 0.125,
  "l3": 0.125,
  "r1": 0.02317299,
  "r2": 0.0217572,
  "r3": 0.0625,
```

```
  "r4": 0.0217572,
  "r5": 0.00438081,
  "r6": 0.00438081,
  "area": 0.054689747231994656,
  "mesh_type": 0,
  "points_per_curve": 100,
  "small_inlet_minimum_mesh_size": 0.001,
  "small_inlet_maximum_mesh_size": 0.005,
  "mesh_maximum_length": 0,
  "small_inlet_minimum_distance": 0.0,
  "small_inlet_maximum_distance": 0.005,
  "temperature": 298.22054023,
  "big_outlet_presure": 101226.24585534,
  "big_inlet_speed": 1.85918711,
  "small_inlet_speed": -0.12157433,
  "kinematic_viscosity": 1.51146962e-05
}
```

Chapter 4

Methodology

4.1 Method overview

The chapter methodology contains all the tools and techniques employed to encounter solutions to the design problem of the studied case. The proposed workflow for every simulation run begins with the Monte Carlo generation of random parameters. Such parameters which include geometrical values and physical magnitudes are described in the case study chapter. The geometrical parameters are then used to construct the mesh with Gmsh. The physical parameters are introduced in the configuration files by the python libray PyFoam for the simulation to be runned with OpenFOAM's solver IcoFoam.

Once the batch of simulations is complete, the results of all simulations are compiled and processed. From amongst all the results, an optimization technique named Pareto optimization is used to find the results which satisfy the multi-objective requirements of the case. Finally the results of the simulation set are plotted hightlightning the Pareto frontier with the optimal results. The workflow is summarized in diagram 4.1.

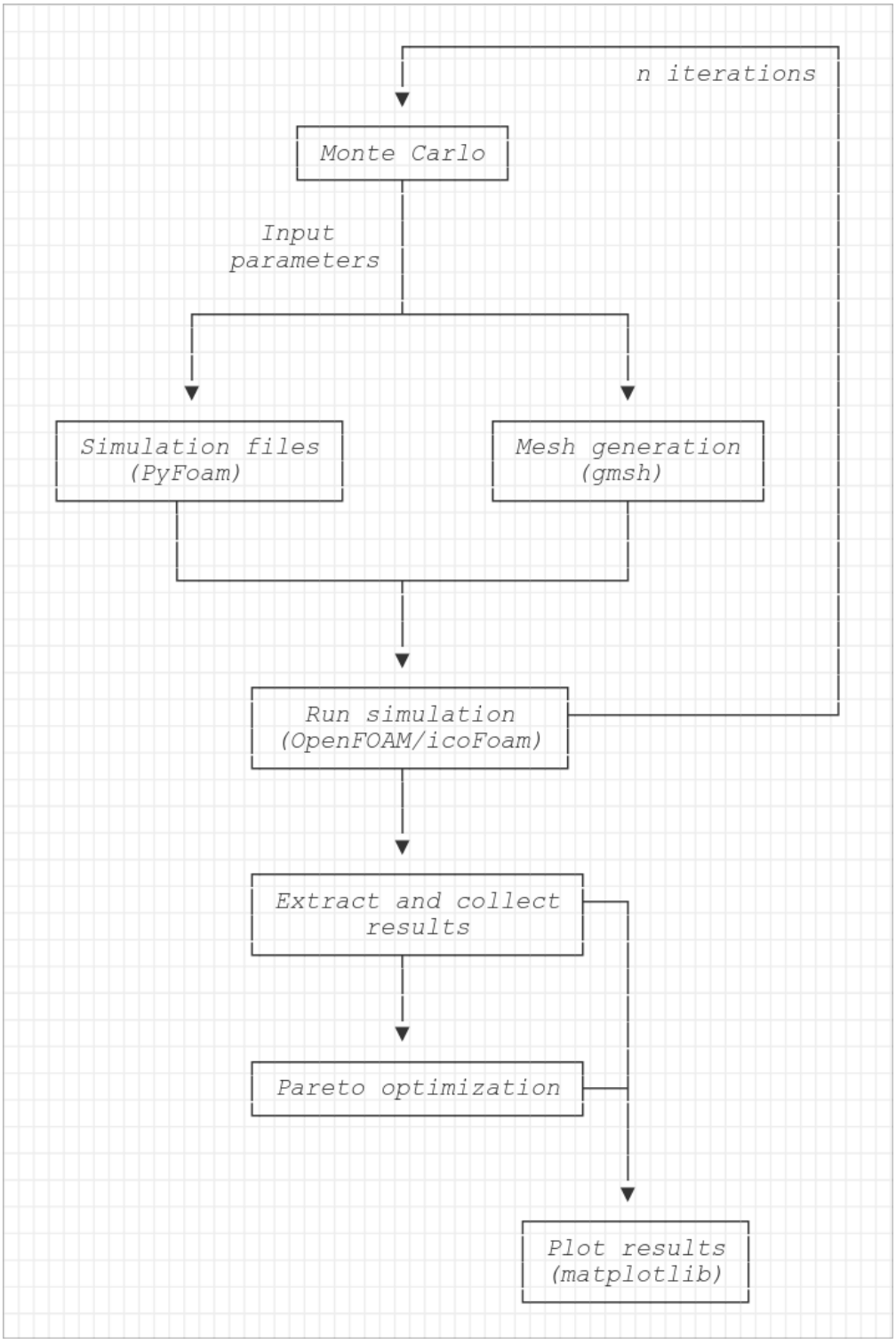


Figure 4.1: Worflow overview

4.2 Monte Carlo method

Monte Carlo methods are a class computational algorithms of generating random values to obtain results and analyse the behaviour of complex phenomena. These techniques were developed alongside computers as they require performing mathematical operations at a large-scale to be effective. This class of methods are specially useful for calculating integrals, differential equations, and various types of models involving some form of randomness.

Monte Carlo methods encompass a variety of techniques that use random sampling to solve complex problems, each tailored to different situations. Simple random sampling generates straightforward random samples for general problems. Importance sampling focuses on critical areas to improve efficiency and accuracy where it matters most. Other methods include Markov Chain Monte Carlo, used for detailed statistical analysis, or sequential Monte Carlo, useful for filtering and time series.

As means to verify if a Monte Carlo method is correctly set up, statistical validation and convergence analysis can be used [10]. On one hand, statistical validation checks the statistical properties of the results, such as their mean and variance, and comparing these with known theoretical values. On the other hand, convergence analysis is crucial for confirming the reliability of a Monte Carlo simulation; it involves observing how the output stabilizes or converges to a specific value as the number of simulations increases.

The present project uses Monte Carlo to generate the inputs corresponding to the geometrical and physical parameters of the simulations. Since the value generation is random, it is independent of the rest of the process. These inputs build the particular case instance to be simulated from the general model of the case, that is, to randomly characterize the shape of the pipes and the fluids temperature, pressures and flows with specific values. The technique is used together with optimization by generating the dataset among which optimal samples are found.

Variable	**Range**
r1	$(0.02, 0.045)$ $[m]$
r2 = r4	$(0.02, 0.045)$ $[m]$
r5 = r6	$(0.0025, 0.005)$ $[m]$
area	surface_area()
temperature	$(268.15, 313.15)$ $[K]$
big_outlet_pressure	$(101325, 101025)$ $[Pa]$
big_inlet_speed	$(0.2, 0.3)/(2\pi \cdot \text{r4})$ $[m/s]$
small_inlet_speed	$(0.065, 0.07) \cdot$ big_inlet_speed $[m/s]$
kinematic_viscosity	$0.00001789 \cdot 287.057 \cdot$ temperature $[m^2/s]$

Table 4.1: Monte Carlo generated parameters and their ranges

In table 4.1, one can see the variables generated by the Monte Carlo procedure with their corresponding ranges. All randomly generated values follow a uniform distribution and are the rounded to arbitrary precision. The computation of some variables is too

vast, complex, or depends on other variables, to be simply expressed as a range of values. In these cases, a formula or function name is provided. The implementation of the stochastic method can be seen in the code block below.

```
p_atm = 101325 # Atmosferic pressure in pascals
def monte_carlo():
  """
  Refresh the randomized inputs (Monte Carlo method).
  """
  global inputs
  inputs["r1"] = randfp(0.02, 0.045, precision)    # outlet pipe radius 20-45mm
  inputs["r2"] = randfp(0.02, 0.045, precision)    # turn pipe radius (torus)
  inputs["r4"] = inputs["r2"]                      # inlet radius
  inputs["r5"] = randfp(0.0025, 0.005, precision) # small inlet radius 2.5-5mm inside geometry
  inputs["r6"] = inputs["r5"]                      # small inlet radius 2.5-5mm outside geometry
  inputs["area"] = surface_area() # Geometry external area
  inputs["temperature"] = randfp(268.15, 313.15, precision) # T [K] (-5 žC, 40žC)
  inputs["big_outlet_presure"] = randfp(p_atm, p_atm-300, precision)
  inputs["big_inlet_speed"] = round(randfp(0.2, 0.3, precision) / (2 * math.pi * inputs["r4"]), precision)
  inputs["small_inlet_speed"] = round(randfp(0.065, 0.07, precision) * inputs["big_inlet_speed"], precision) * -1.0 # Negative z direction
  inputs["kinematic_viscosity"] = round(0.00001789 * 287.057 * inputs["temperature"] / p_atm, 14)
```

4.3 Mesh

A mesh is a grid or network of interconnected nodes used to represent geometrical objects. A mesh is composed of the following elements establishing a structural hierarchy:

a) Nodes are the fundamental building block of a mesh.

b) Edges connect the nodes.

c) Faces are 2D elements composed by an area with closed outline not containing any nodes nor edges inside.

d) Surfaces are a set of faces.

e) Volumes are 3D elements composed by a space fully enclosed by a set of faces or surfaces.

4.3.1 Types of mesh

Meshes can be classified according to different criteria such as dimensionality, cell shape and regularity.

Dimensionality

Mesh dimensionality refers to the number of spatial dimensions in which the mesh elements are contained. It indicates whether the mesh is composed of 1D, 2D, or 3D elements.

For instance, a one-dimensional meshes are the meshes which can be drawn in an infinitesimally thin and infinitely long line, that is, mesh exclusively made of line segments. Two-dimensional meshes are all the meshes with elements which drawn in coplanarity between each other, in other words, all the meshes which can be contain in a plane. These consist of planar elements like triangles or quadrilaterals. Three-dimensional meshes are the meshes which contain volumetric elements.

Cell shape

The cell shape of a mesh is the shape of the minimum structural unit of, at least, two dimensions, since nodes (0D) and edges (1D) are not taken into account. The most common bidimensional cell shapes are triangles, forming triangular meshes, and quadrilaterals, forming quadrilateral meshes.

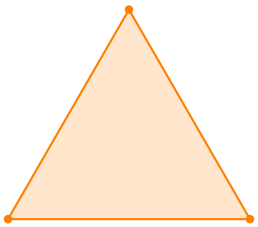


Figure 4.2: Triangle

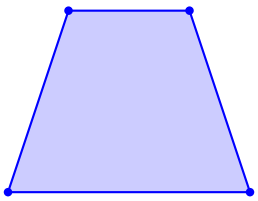


Figure 4.3: Quadrilateral

Additionally, the most common tridimensional cell shapes are tetrahedra, forming tetrahedral meshes, and hexahedra, forming hexahedral meshes.

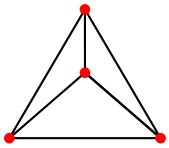


Figure 4.4: Tetrahedron

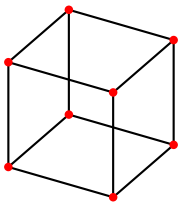


Figure 4.5: Hexahedron

Some meshes can be build using different cell shapes leading to mixed-element meshes. In general terms, the increase in cell shape complexity means an increase in the computational cost of generating the mesh.

The elements used for the mesh of interest in this study are triangles and hexahedrons. Although the program contemplates the generation of meshes with quadrilaterals

and tetrahedra, this has been discarded. This is due to the tendency of meshing errors to occur in combination with the random geometry from Monte Carlo methods in use, apart from the added computational and time cost of the tetrahedra meshing process. Since robustness and adaptability is sought-after for the program implementation to be able to solve other problems, triangles are the mesh shape chosen.

Grid regularity

Grid regularity refers to the arrangement of the elements composing the mesh. On one hand, there are structured meshes which have elements arranged in a regular, grid-like pattern, making them easier to generate and handle computationally. On the other hand, unstructured meshes contain elements arranged irregularly allowing for the representation of complex geometries. The combination of both creates hybrid meshes which merge different element types and distributions.

4.3.2 Gmsh library

Gmsh is a open-source finite element mesh generator with a built-in CAD engine and post-processor. It lets the user create, manipulate, and visualize 2D and 3D meshes. These meshes can be used afterwards for numerical simulations. It is widely used in scientific computing, engineering simulations, and other fields requiring mesh generation.

Although Gmsh provides its own scripting language, it can be used through its many application programming interfaces (APIs) for languages such as C, C++, Fortran, Python or Julia. The program's functionality can also be accessed through it's graphical user interface (GUI), which permits the visualization of meshes as it can be seen in figure 4.6. [11][12]

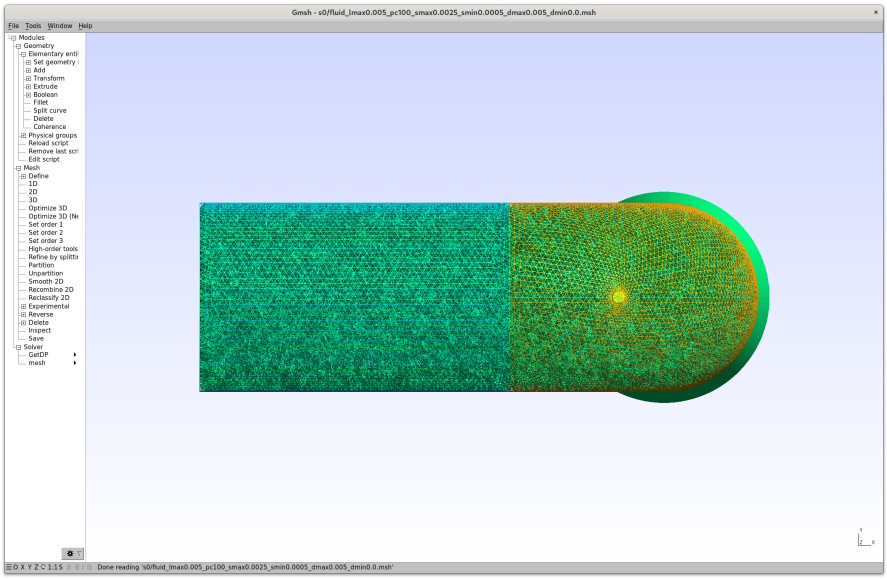


Figure 4.6: Gmsh GUI

4.3.3 Mesh generation

A meshing algorithm is tasked with transforming a determined geometry into a set of nodes and edges, that is dividing the geometry into discrete elements. Gmsh is compatible with the following algorithms being Front-Delaunay the default option [12]:

1. MeshAdapt
2. Automatic
3. Initial mesh only
4. Delaunay
5. Frontal-Delaunay
6. Bidimensional Anisotropic Mesh Generator (BAMG)
7. Frontal-Delaunay for Quads
8. Packing of Parallelograms
9. Quasi-structured Quad

Front-Delaunay algorithm

Front-Delaunay is an efficient unstructured mesh generation technique that combines Delaunay triangulation with the Bowyer-Watson algorithm. This method works by

incrementally inserting points into an initially coarse mesh, ensuring that the resulting triangles conform to the Delaunay condition. The Delaunay condition states that for a given triangulation of a set of points, the circumcircle of any triangle in the triangulation should not contain any other points from the set in its interior. Triangulation is the process of dividing a geometric object, such as a polygon or a set of points, into triangles such that the triangles together cover the object without overlapping [13]. The front, or boundary, of the mesh is maintained and updated as new points are added, which helps in managing the local topology and refining the mesh to meet specific quality criteria [14].

Given that the meshing procedure itself does not belong the core of the project, and that Front-Delaunay is a robust algorithm, easily translatable to other geometries, this algorithm has been selected. Its robustness facilitates the generalization of mesh generation to other models.

Boundary layer meshing

Boundary layer meshing is a technique specially used in computational fluid dynamics to generate meshes that accurately simulate the boundary layer of fluid flow near surfaces. The technique creates a series of layers of mesh cells close to the boundary. These layers are typically denser near the surface and become progressively scattered as they move away from the surface. The resultant type of mesh can describe more accurately the velocity gradients and other properties occurring near the boundary [15].

Simulating these layers with precision is rellevant to model the fluid properly as boundary fluid layers can affect overall flow dynamics, especially in viscous flows. Nevertheless, boundary layer meshing has not been used in the current mesh generation setup as the Gmsh library does not implement such feature.

Mesh generation implementation

The consequent code block is corresponds to the Python implementation using Gmsh API of the mesh generation function building the geometry of this particular case.

```
1  def generate_mesh():
2    """
3    Generate a mesh from input parameters and save to file.
4    """
5    gmsh.initialize()
6    gmsh.model.add("fluid") # Create model
7    # Create geometry
8    if inputs["r1"] == inputs["r2"]:
9      volume1 = gmsh.model.occ.addCylinder(inputs["ox"], inputs["oy"], inputs["oz
       "], 0, 0, inputs["l1"], inputs["r1"])
10   else:
11     volume1 = gmsh.model.occ.addCone(inputs["ox"], inputs["oy"], inputs["oz"],
       0, 0, inputs["l1"], inputs["r1"], inputs["r2"])
12   volume2 = gmsh.model.occ.addTorus(inputs["ox"] - inputs["r3"], inputs["oy"],
       inputs["oz"] + inputs["l1"], inputs["r3"], inputs["r2"], -1, inputs["a1"],
       zAxis=[0, 1, 0])
13   if inputs["r2"] == inputs["r4"]:
```

```
14      volume3 = gmsh.model.occ.addCylinder(inputs["ox"] + inputs["r3"] * (math.
        cos(inputs["a1"]) - 1), inputs["oy"], inputs["oz"] + inputs["l1"] + inputs[
        "r3"] * math.sin(inputs["a1"]), -inputs["l2"] * math.sin(inputs["a1"]), 0,
        inputs["l2"] * math.cos(inputs["a1"]), inputs["r4"])
15    else:
16      volume3 = gmsh.model.occ.addCone(inputs["ox"] + inputs["r3"] * (math.cos(
        inputs["a1"]) - 1), inputs["oy"], inputs["oz"] + inputs["l1"] + inputs["r3"
        ] * math.sin(inputs["a1"]), -inputs["l2"] * math.sin(inputs["a1"]), 0,
        inputs["l2"] * math.cos(inputs["a1"]), inputs["r2"], inputs["r4"])
17    if inputs["r5"] == inputs["r6"]:
18      volume4 = gmsh.model.occ.addCylinder(inputs["ox"] + inputs["r3"] * (math.
        cos(inputs["a1"] / 2) - 1), inputs["oy"], inputs["oz"] + inputs["l1"] +
        inputs["r3"] * math.sin(inputs["a1"] / 2), inputs["l3"] * math.cos(inputs["
        a1"]), 0, inputs["l3"] * math.sin(inputs["a1"]), inputs["r5"])
19    else:
20      volume4 = gmsh.model.occ.addCone(inputs["ox"] + inputs["r3"] * (math.cos(
        inputs["a1"] / 2) - 1), inputs["oy"], inputs["oz"] + inputs["l1"] + inputs[
        "r3"] * math.sin(inputs["a1"] / 2), inputs["l3"] * math.cos(inputs["a1"]),
        0, inputs["l3"] * math.sin(inputs["a1"]), inputs["r5"], inputs["r6"])
21    gmsh.model.occ.synchronize() # Sync changes
22    #gmsh.model.getEntities(dim=-1) # Print all entities
23    # Merge all volumes into a single one
24    volumes = [volume1, volume2, volume3, volume4]
25    volume_tuples = [(3, v) for v in volumes]
26    fused_volume, _ = gmsh.model.occ.fuse(volume_tuples, volume_tuples)
27    gmsh.model.occ.synchronize() # Sync changes
28    # Physical boundaries
29    gmsh.model.addPhysicalGroup(2, [7], 14)
30    gmsh.model.setPhysicalName(2, 14, inputs["boundary_conditions"][0])
31    gmsh.model.addPhysicalGroup(2, [3], 15)
32    gmsh.model.setPhysicalName(2, 15, inputs["boundary_conditions"][1])
33    gmsh.model.addPhysicalGroup(2, [6], 16)
34    gmsh.model.setPhysicalName(2, 16, inputs["boundary_conditions"][2])
35    gmsh.model.addPhysicalGroup(2, [2, 5, 1, 4], 17)
36    gmsh.model.setPhysicalName(2, 17, inputs["boundary_conditions"][3])
37    gmsh.model.addPhysicalGroup(3, [fused_volume[0][1]], 18)
38    gmsh.model.setPhysicalName(3, 18, inputs["boundary_conditions"][4]) # fluid
        volume
39    gmsh.model.occ.synchronize() # Sync changes
40    # Mesh size fields for small inlet
41    gmsh.model.mesh.field.add("Distance", 1)
42    gmsh.model.mesh.field.setNumbers(1, "FacesList", [4, 6])
43    #gmsh.model.mesh.field.setNumber(1, "NumPointsPerCurve", inputs["
        points_per_curve"])
44    gmsh.model.mesh.field.add("Threshold", 2)
45    gmsh.model.mesh.field.setNumber(2, "InField", 1)
46    gmsh.model.mesh.field.setNumber(2, "SizeMin", inputs["
        small_inlet_minimum_mesh_size"]) # Minimum element size in the small inlet
47    gmsh.model.mesh.field.setNumber(2, "SizeMax", inputs["
        small_inlet_maximum_mesh_size"]) # Maximum element size elsewhere
48    gmsh.model.mesh.field.setNumber(2, "DistMin", inputs["
        small_inlet_minimum_distance"])
49    gmsh.model.mesh.field.setNumber(2, "DistMax", inputs["
        small_inlet_maximum_distance"])
50    gmsh.model.mesh.field.setAsBackgroundMesh(2)
51    # Set options
52    gmsh.option.setNumber("Mesh.MshFileVersion", 2.2)
53    gmsh.option.setNumber("General.Terminal", 1) # Enable terminal output
54    gmsh.option.setNumber("Mesh.Algorithm", 6) # Front-Delaunay default algorithm
55    gmsh.option.setNumber("General.NumThreads", os.cpu_count()) # Parallel
        processing with all threads
56    if inputs["mesh_type"]:  # quad mesh specific options
```

```
57      gmsh.option.setNumber("Mesh.RecombineAll", 1) # Triangular recombination
        into quadrilaterals
58      surfaces = gmsh.model.getEntities(dim=2)
59      for surface in surfaces:
60        gmsh.model.mesh.setRecombine(2, surface[1])
61        gmsh.model.mesh.setSmoothing(2, surface[1], 10) # Smoothing to improve
        element quality
62    gmsh.model.mesh.generate(3) # Generate 3D mesh
63    #gmsh.write("fluid.msh") # Write mesh to file
64    mesh_file = f"fluid_smax{inputs['small_inlet_maximum_mesh_size']}_smin{inputs
        ['small_inlet_minimum_mesh_size']}_pc{inputs['points_per_curve']}_dmax{
        inputs['small_inlet_maximum_distance']}_dmin{inputs['
        small_inlet_minimum_distance']}.msh"
65    gmsh.write(mesh_file) # Write mesh to file
66    # gmsh.fltk.run() # Visualize
67    gmsh.finalize()
68    return mesh_file
```

4.3.4 Computational costs of mesh generation

A simple test has been performed to measure the approximate time it takes to fully generate a mesh base on its element size parameters. The mesh used is parametrized by the same Monte Carlo procedure employed in the main simulation program.

Monte Carlo methods are use to model or test phenomena with significant uncertainty in inputs. They consist in generating samples from random inputs to observe how a system behaves. In this case, both geometrical and mesh characteristic parameters are randomly defined for every meshing instance obtaining the time such process endured.

Test conditions

As the metric of this test is machine dependant in terms of hardware capabilities as well as software configuration, a humble attempt at profiling the machine performing the test is done.

The execution environment for the test is a Docker container running Ubuntu Jammy. The container has 2GB of RAM memory allocated and unlimited usage of a 19GB swap file to improve the stability of the test in the test hardware (as excessive memory usage causes the test to be killed by the system).

Component	Brand and model	Additional Specifications
Computer	Lenovo ideapad 330-15ICH	
Processor	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz	Threads 12; Max. freq. 4.1GHz
Mainboard	Lenovo LNVNB161216	
Memory	Samsung SODIMM DDR4 Synchronous 2400 MHz	7814MiB
OS	Arch Linux x86_64	Kernel 6.9.3-arch1-1

Table 4.2: Host machine description

Different mesh sizes have been applied to the mesh since the small inlet requires a smaller mesh size to have its geometry accurately represented, while the rest of the elbow

can be assigned a larger mesh size. These geometrical regions with varying mesh sizes are displayed in figure 4.7 where the different mesh densities can be appreciated.

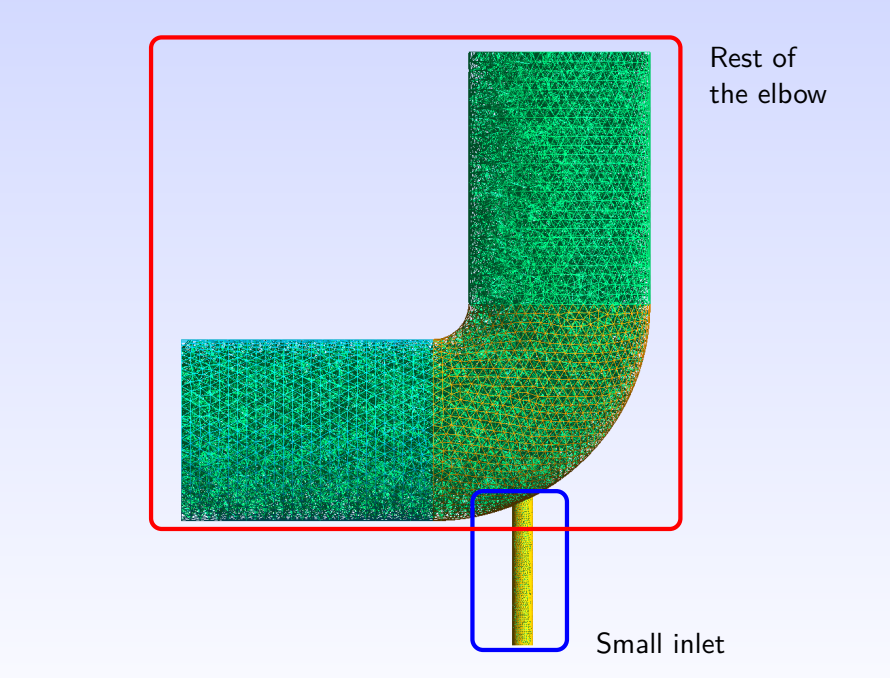


Figure 4.7: Mesh areas with different mesh sizes

Triangular mesh results

This meshing time test has generated 300 meshes with triangular mesh elements. A 3D plot (4.8) has been build with the test data plotting the independent variables on the bottom plane and the meshing time on the vertical axis. Supplementarily, some statistical metrics on the mesh sample set can be seen in table 4.3 and the distribution of meshing times and the convergence of mean meshing time are shown in charts 4.9.

Number of samples	300
Mean meshing time	35.72 s
Standard deviation of meshing time	158.91 s
Minimum meshing time	0.81 s
Maximum meshing time	2200.59 s

Table 4.3: Test information and statistical metrics

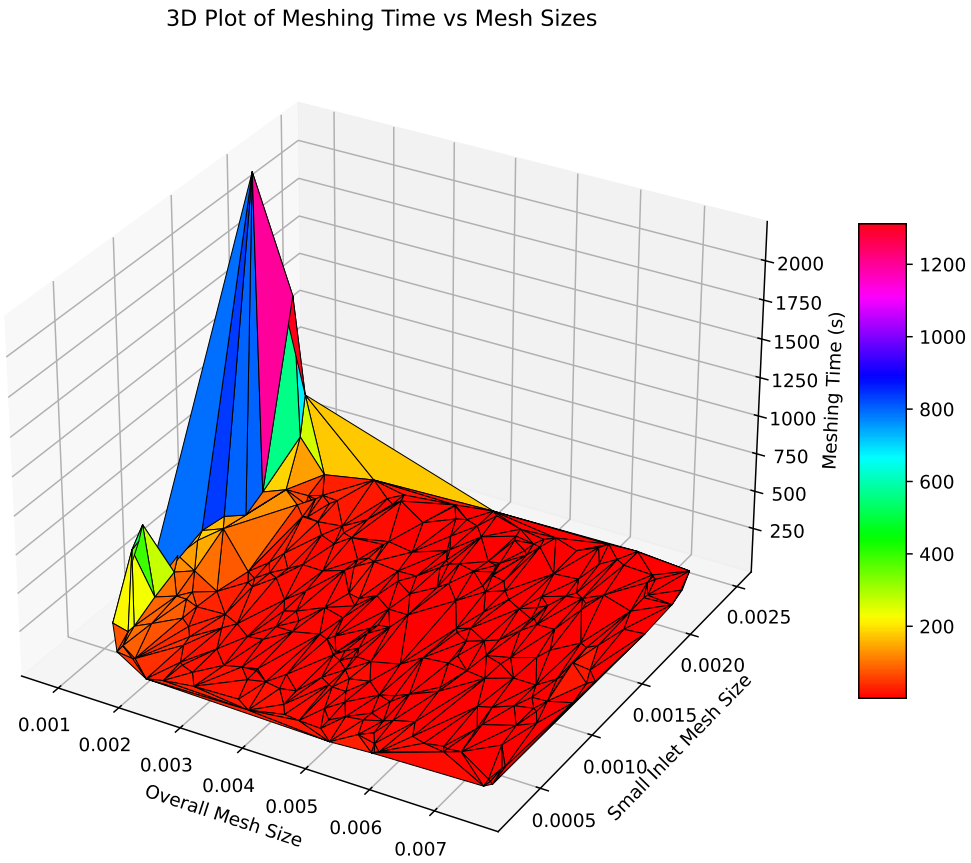


Figure 4.8: Mesh size (overall and small inlet) vs. meshing time

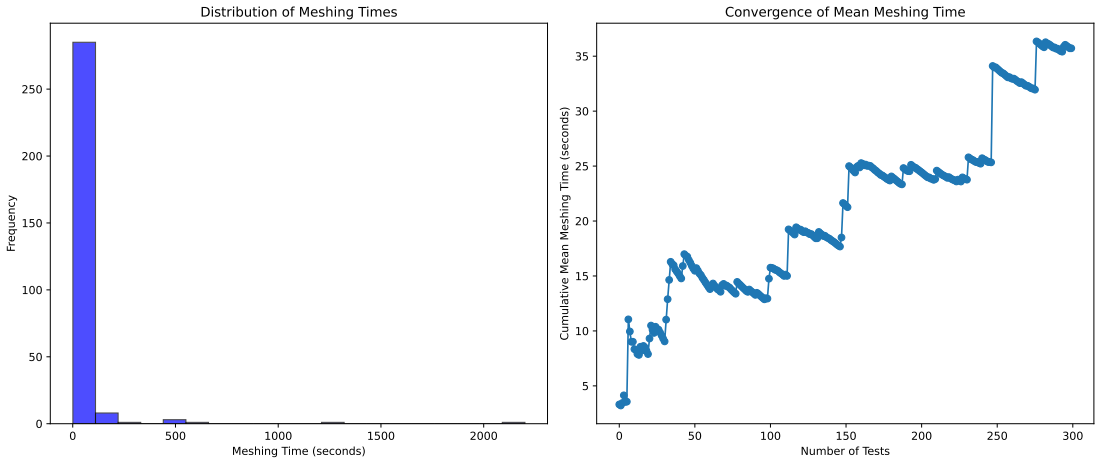


Figure 4.9: Distribution of meshing times and convergence of mean meshing time

Quadrilateral mesh results

This meshing time test has generated 52 meshes with quadrilateral mesh elements. A 3D plot (4.10) has been build with the test data plotting the independent variables on the bottom plane and the meshing time on the vertical axis. Supplementarily, some statistical metrics on the mesh sample set can be seen in table 4.4 and the distribution of meshing times and the convergence of mean meshing time are shown in charts 4.11.

Number of samples	52
Mean meshing time	177.06 s
Standard deviation of meshing time	377.07 s
Minimum meshing time	4.81 s
Maximum meshing time	1794.94 s

Table 4.4: Test information and statistical metrics

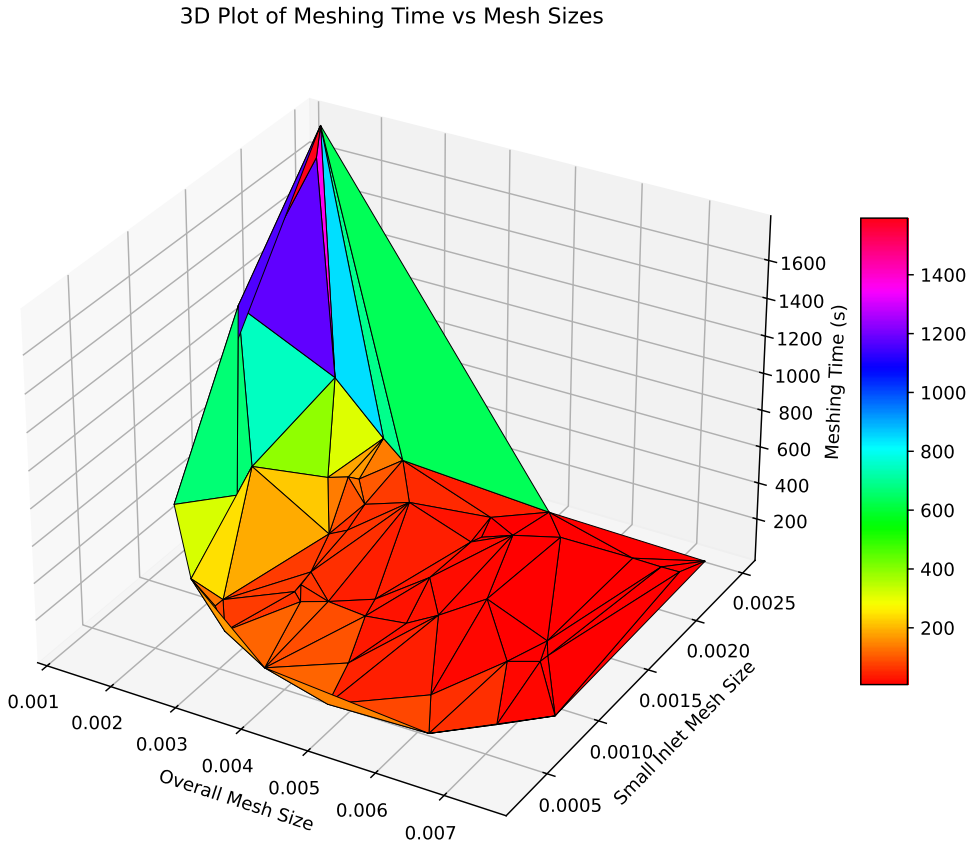


Figure 4.10: Mesh size (overall and small inlet) vs. meshing time

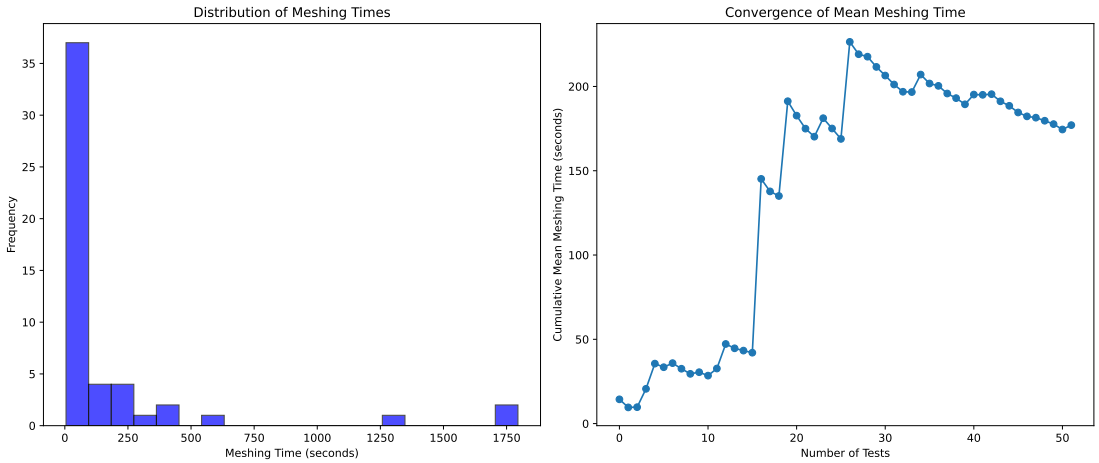


Figure 4.11: Distribution of meshing times and convergence of mean meshing time

Meshing time test conclusions

Both mesh size for the small inlet and the rest of the mesh have been given random and independet mesh size values. This is why no noticeable correlation exists between the two. Nonetheless, in the overall mesh size plot for both triangular and quadrilateral meshes, it can be seen how the reduction in mesh size seems to increase exponentially the compute time of the mesh. It can be observed to that the small inlet mesh size does not appear to affect meaningfully the meshing time.

It has been seen during the realization of test that quadrilateral meshes have way larger compute times for smaller mesh sizes than triangular meshes. The quadrilateral mesh sample size is reduced due to their extensive calculation period and error prone meshing process. The data set may be biased as the test machine was not able to complete really long quadrilateral meshing processes and store their results.

These mesh test have aided narrow down the mesh values to employ in simulations. In chosing the mesh sizes used in the model, a tradeoff is made between compute time and accuracy as smaller mesh size take longer to compute but produce more precise results and tend to be more stable. The mesh sizes used in final program are 0.001 for the small inlet and 0.005 for the overall mesh. Such values have been observed to provide robust results, fear model representation, and reasonable compute times.

4.4 Computational fluid dynamics software

4.4.1 OpenFOAM

OpenFOAM [16] (Open Field Operation and Manipulation) is an open-source computational fluid dynamics (CFD) software toolkit. It originated from the work done at the Imperial College London in the late 1980s and early 1990s. The initial development was

led by Henry Weller, Gavin Tabor, and Hrvoje Jasak, who aimed to create a flexible CFD code for academic and industrial use.

The open-source nature of OpenFOAM allows users to customize solvers and utilities for their specific needs, providing a powerful and versatile tool for simulations involving fluid flow, heat transfer, chemical reactions, and solid dynamics, among others. Its applications span across various industries, including automotive, aerospace, environmental engineering, and energy sectors [17][18].

4.4.2 icoFoam solver

IcoFOAM [19] is one of the many solvers available in the OpenFOAM suite. It is specifically designed for solving incompressible and laminar flow problems using the Navier-Stokes equations. IcoFOAM is suitable for applications where the flow velocity is relatively low, and the effects of turbulence can be neglected.

The solver employs the finite volume method to discretize the equations and typically uses the PISO (Pressure Implicit with Splitting of Operators) algorithm for pressure-velocity coupling. IcoFOAM is widely used in academic research and industrial applications where accurate modeling of incompressible laminar flow is required [20].

The solver computes the solution to the equations (2.25, 2.26) presented in section 2.9 on fluid governing equations.

4.5 Additional software

4.5.1 PyFoam library

PyFoam [21] is a Python library for working with OpenFOAM. It is designed to facilitate the use of OpenFOAM by providing tools for running and manipulating simulations, post-processing results, and automating workflows. The library allows users to script complex simulations, manage multiple runs, and perform parameter studies with ease. It integrates well with other Python libraries, enabling advanced data analysis and visualization. One of its main advantages is that it combines Python's flexibility and its vast collection of libraries with OpenFOAM's simulation capabilities [22][23].

The implementated simulator leverages mainly two PyFoam types of functions:

- **Runners**
  Runners give access to OpenFOAM binaries and they are used to run utilities and solvers. They provide logging capabilities an functions to check if their execution was successful.
  `BasicRunner()`
  It is used to convert the mesh from Gmsh to foam format and to run the solver.

```
1 msh_runner = BasicRunner(argv=["gmshToFoam", "-case",
    ".", mesh_file], silent=False)
2 msh_runner.start()
```

```python
from PyFoam.Execution.BasicRunner import BasicRunner
runner = BasicRunner(argv=["icoFoam", "-case", "."],
    silent=False)
runner.start()
runner.runOK()
```

`UtilityRunner()`
It is used to run the utility which converts foam simulation results to VTK format.

```python
from PyFoam.Execution.UtilityRunner import
    UtilityRunner
pv_runner = UtilityRunner(argv=["foamToVTK", "-case",
    "."])
pv_runner.start()
```

- **Case modifiers**
  Case modifiers are used to modify case files and the parameters they contain. Such parameters include the initial conditions of the system like pressure, speed or viscosity, boundary definitions, and simulation options like time and time steps, amongst others.
  `ParsedParameterFile()`
  It allows to modify case parameters.

```python
p_file = ParsedParameterFile(f'{path}/0/p')
p_file["boundaryField"][inputs["boundary_conditions"
    ][1]]["value"] = "uniform 0"
p_file.writeFile()
```

- **Other funcions**
  `SolutionDirectory()`
  It creates a foam file for the case.

```python
from PyFoam.RunDictionary.SolutionDirectory import
    SolutionDirectory
case = SolutionDirectory(path) # Create foam case file
```

4.5.2 ParaView

ParaView [24] is an open-source data analysis and visualization application. It was developed collaboratively by Kitware Inc., Los Alamos National Laboratory, and other institutions, with its origins tracing back to the early 2000s. The program is designed to handle large datasets generated by simulations and experiments, providing interactive and batch processing capabilities. Its applications include visualizing CFD results, structural analysis, climate data, and more. ParaView supports a wide range of data formats and provides tools for creating visual representations [25][26].

The current work uses ParaView to visualize meshes and simulation results to verify that the generated simulations are indeed correctly implemented and behave as expected.

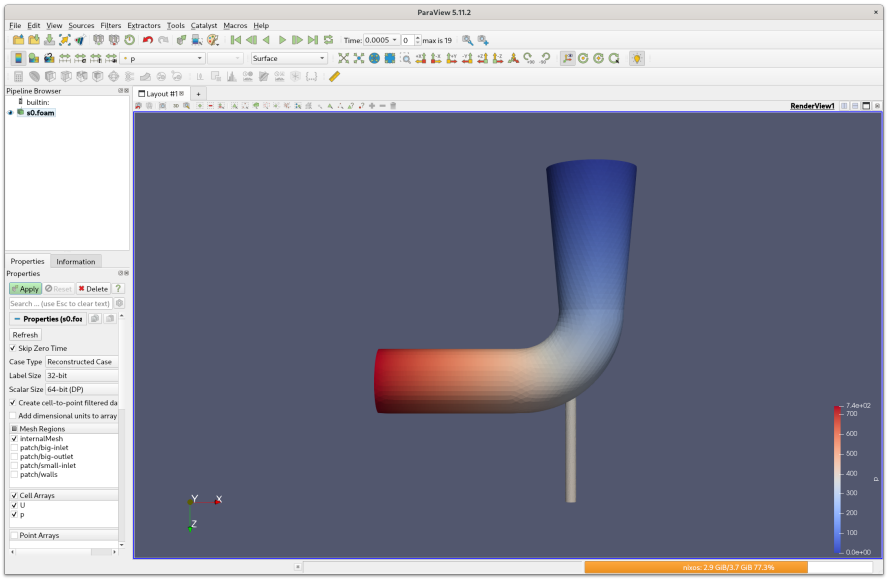


Figure 4.12: ParaView GUI

4.6 Optimization

Optimization is a field of applied mathematics concerned with encountering minimums and maximums of functions, that is, finding the optimal value given a set of constraints. The various methods of optimization are used across disciplines such engineering, economy, medicine, manufacturing, logistics, or statistics. [27]

Generally, optimization problem is composed of three main elements: decision variables, constraints and an objective function. Decision variables are the parameters that can be adjusted to achieve the desired outcome, representing the choices available within the problem. Constraints are the conditions or limitations imposed on the decision variables. The objective function is a mathematical expression that needs to be optimized (maximized or minimized), representing the goal of the problem. It is formulated in terms of the decision variables to evaluate the quality or performance of different possible solutions. [28]

4.6.1 Optimization method classification

There is no single classification for optimization methods and several criteria can be used to order them. Optimization methods can be classified as constrained or unconstrained, according to the presence of constraints, as single-objective or multiobjective based on the number of objectives, and as linear, nonlinear, geometric or non-separable, depending

on the nature of the objective functions and constraints. They are also categorized by the time dependency of constraints as online, if constrains are time dependant (for example in real time applications), or offline optimization, if time is not a limiting factor. Methods include deterministic algorithms that depend on clear relationships between system attributes and heuristic algorithms that create solutions using random sampling and specific problem knowledge. Metaheuristic techniques, which mix heuristics and objective functions, are popular for their effectiveness in tackling complex problems. These include evolutionary computation like genetic algorithms, swarm intelligence like particle swarm optimization, and methods inspired by physical processes like simulated annealing. [28]

The current optimization problem has two objectives to satisfy:

- Minimize the quantity of material required to manufacture the component. This is the independent variable of the optimization problem. The quantity of material is proportional to the surface area of the model which depends on the geometrical parameters generated by the Monte Carlo method.
- Reduce the input pressure at the small inlet provided by an injector pump. The input pressure at the small inlet is the dependent variables as it is computed by the simulation's solver.

The constraints of the problem are the inputs presented in section 3.2 and the constraints embeded in the model design itself. The objective function to be optimized is the function solved by the solver from its configuration files, containing boundary conditions and physical parameters, and the generated mesh from geometrical parameters.

As the simulation depends on time, it is considered an online optimization procedure, and, given that Monte Carlo is used, it is also a stochastic local optimization procedure. Accounting for the multiple objectives of the problem, facing such optimization problem requires multiobjective optimization methods.

4.6.2 Cost curves

Cost and material

The cost is expressed in terms of value per thickness unit, in this case, as Euros per milimeter of thickness $[e/mm]$. This is done to abstract away the thickness parameter as it is independent from all other aspects of the case. To formulate the cost, the next calculation must be conducted:

$$\text{cost} = \text{area} \cdot \text{material cost} \cdot \text{material density} \tag{4.1}$$

The area can be directly computed from the geometrical parameters but the volumetric cost of material is still to be determined. The chosen material is austenitic stainless steel 304. The price of bright drawn bar 304 in Europe, as of frebruary 2024, is 2.999 e/kg [29]. Furthermore, the density of stainless steel 304 is 7930 kg/m^3 [30]. This data allows

to calculate the cost per material thickness as follows:

$$\text{cost } [e/mm] = \text{area } [mm^2] \frac{10^6 \ mm^2}{1m^2} \frac{2.999 \ e}{1 \ kg} \frac{7930 \ kg}{1 \ m^3} \frac{1 \ m^3}{10^9 \ mm^3} \tag{4.2}$$

This expression provides the cost per material thinkness of every simulation case given the external surface are of the geometry.

Average pressure versus material cost

Once having done enough simulations, a scatter plot can be generated displaying the cost per material thickness over the average pressure of the small inlet at the last time step of every simulation. Every single point of the plot corresponds to a simulation case. The plot represents the space where an optimal solution must be found.

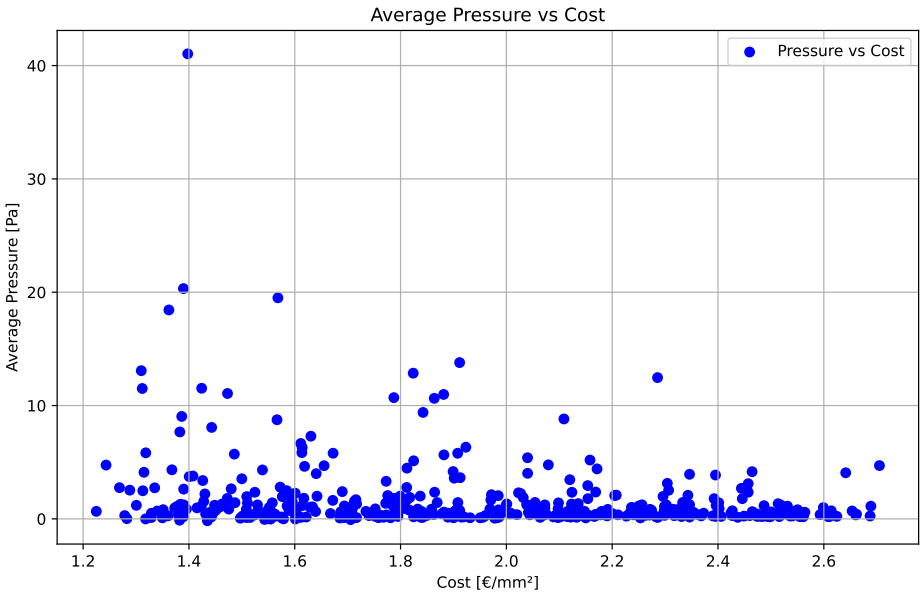


Figure 4.13: Cost per thickness unit vs. average pressure

4.6.3 Pareto optimization

Given an optimization problem, Pareto optimization is the process consisting of finding the set of solutions that cannot be improved on any objective without degrading another. These solutions are called Pareto optimal solutions. The process involves balancing multiple conflicting objectives without giving excessive priority to any, as achieving the best possible outcome in one aspect often leads to compromises in others.

Pareto efficiency or Pareto optimality is achieved when it is impossible to reallocate resources or make changes in a way that would benefit one aspect without causing detriment to another. This concept is crucial in economics, decision science, and engineering, providing a foundation for decisions that aim to achieve the best balance among various conflicting objectives. The set of all Pareto optimal solutions in the objective space compose what is known as a Pareto frontier, or Pareto front. It represents the boundary beyond which no improvements can be made to one objective without worsening another. In the chart 4.14, the concepts of Pareto optimal results and Pareto frontier are illustrated. [31] [32]

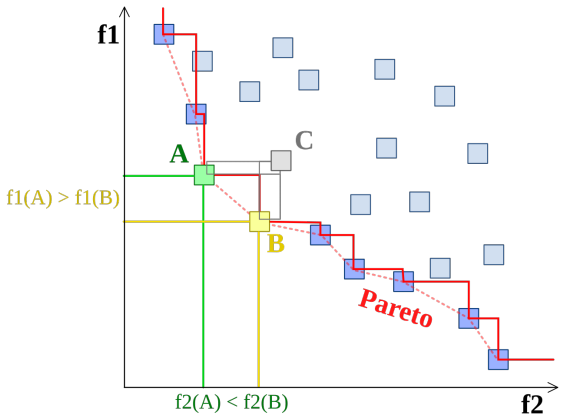


Figure 4.14: Pareto optimal results composing a Pareto frontier

Several multiobjective optimization algorithms exists to encounter Pareto optimal results. For the sake of simplity, this study has opted for the naive Pareto algorithm. Such algorithm is simple but inefficient with a computational complexity of $O(n^2)$ as the computational cost increases exponentially with input size. The naive Pareto algorithm is a brute-force method for identifying Pareto optimal solutions. This algorithm involves checking each solution against all other solutions to determine if it is dominated on any of the objectives as it can be seen in the code block below. A solution is considered Pareto optimal if there is no other solution that is better in all objectives simultaneously. In practice, this means for each candidate solution, the algorithm checks whether there exists another solution that is equal or better in all the evaluated criteria and strictly better in at least one. The naive procedure works great with fewer objectives and small datasets, but can be too computationally expensive for complex problems.

```
1 def not_dominated(point, others):
2   """
3   Check if a data point is dominated by others or a Pareto optimal
      point.
4   """
5   # A point is dominated if there is any other point with lower 'avg_p'
      and 'cost'
6   for other in others:
```

```
7       if other['avg_p'] <= point['avg_p'] and other['cost'] <= point['
        cost'] and (other['avg_p'] < point['avg_p'] or other['cost'] <
        point['cost']):
8           return False
9     return True
10
11  def find_pareto_points(input_csv, output_csv):
12    """
13    Given an input CSV, write Pareto optimal points to output CSV.
14    """
15    # Load CSV data
16    data = pd.read_csv(input_csv)
17    pareto_points = []
18    # Convert DataFrame to list of dicts for easier manipulation
19    points = data.to_dict('records')
20    # Check for Pareto optimality
21    for point in points:
22      if not_dominated(point, points):
23        pareto_points.append(point)
24    # Convert Pareto points back to DataFrame for easier viewing/
        manipulation
25    pareto_df = pd.DataFrame(pareto_points)
26    # Save the DataFrame of Pareto points to a new CSV file
27    pareto_df.to_csv(output_csv, index=False)
28    return pareto_df, data
```

There are more sophisticated approaches beyond the naive Pareto method, which are able to handle more efficiently larger datasets and more complex multiobjective problems. These include evolutionary algorithms like NSGA-II (Non-dominated Sorting Genetic Algorithm II) and SPEA2 (Strength Pareto Evolutionary Algorithm 2). These algorithms use mechanisms such as fitness sorting and crowding distance to maintain diversity in the solution set and to guide the search towards the Pareto front more effectively. Other techniques might involve decomposition approaches, breaking down the multiobjective problem into multiple single objective problems that are easier to solve. Each of these methods offers trade-offs between computational overhead, ease of implementation, and the quality of the solutions generated. [33]

When the naive Pareto algorithm is runned over the data build from the previous sections (the dataset is displayed in figure 4.13), the optimal cases are found. The optimal cases can be plotted hightlightning the Pareto frontier together with all the other cases as shown in figure 4.15 and focused as in figure 4.16. Such optimal cases are exhibited at the later results section in detail.

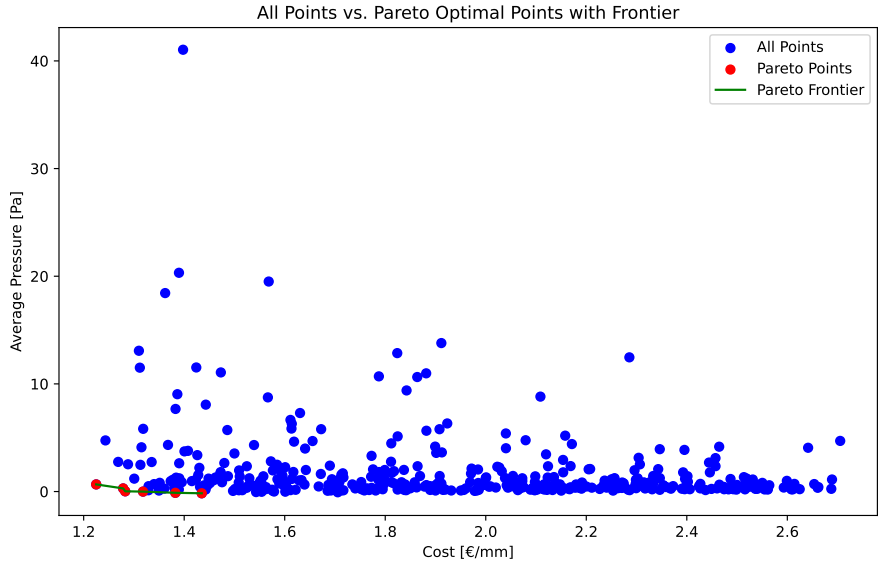


Figure 4.15: Cost per thickness unit vs. average pressure with Pareto optimal points and frontier

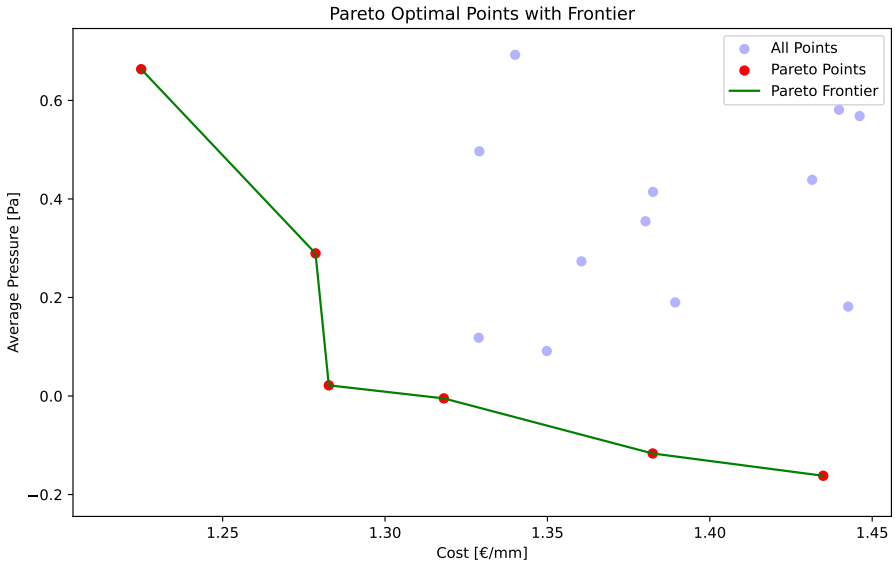


Figure 4.16: Cost per thickness unit vs. average pressure focused on Pareto optimal points and frontier

4.7 Result extraction and presentation

The result extraction pipeline is the process of converting and manipulating simulation results to extract and present rellevant information. The diagram 4.17 exposes the result processing workflow.

In the program, results are collected once from all simulation cases into a CSV file containing the data from all simulations performed. This data is the used to generate another CSV file containing only the material cost per case and the average small inlet pressure. Additional plots are created displaying the cost and the average pressure over the number of simulations and cost versus pressure. Finally, the Pareto optimization function creates another CSV file from the previous one containing only the optimal cases. The optimal case data is used to generate more plots to show the optimal data points, the optimal points among all the simulations done, and the Pareto frontier.

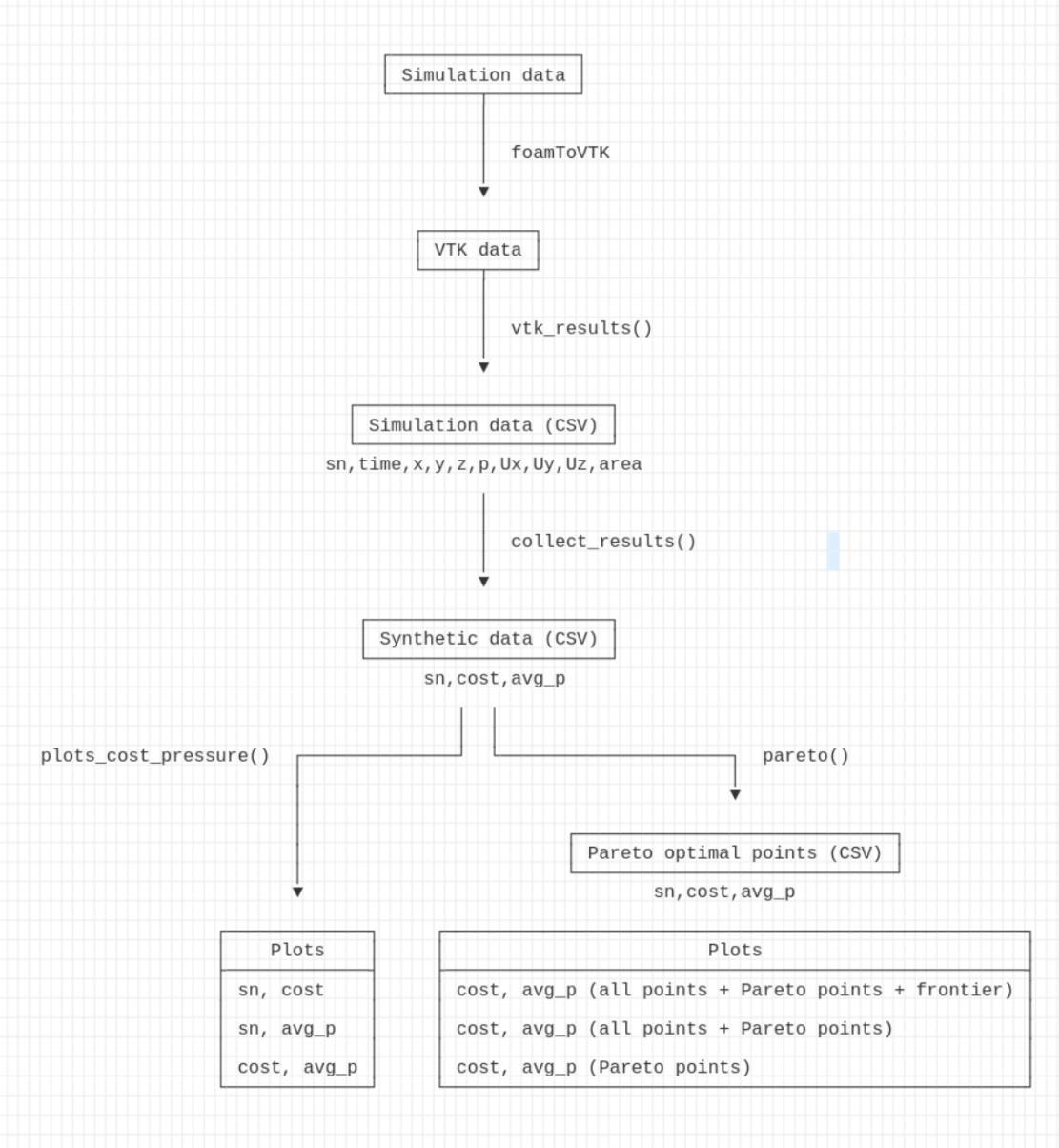


Figure 4.17: Result processing pipeline diagram

4.8 Program architecture

The program is composed of two main files, *run.py* and *func.py*, containing respectively the high level execution control and a collection of functions necessary for the program to work properly. Another auxiliar program, *test.py*, for testing purposes has been build to test program features during its development. A Docker container running OpenFOAM and all the required Python modules has been develop to ensure a stable and immutable

execution environment always exists.

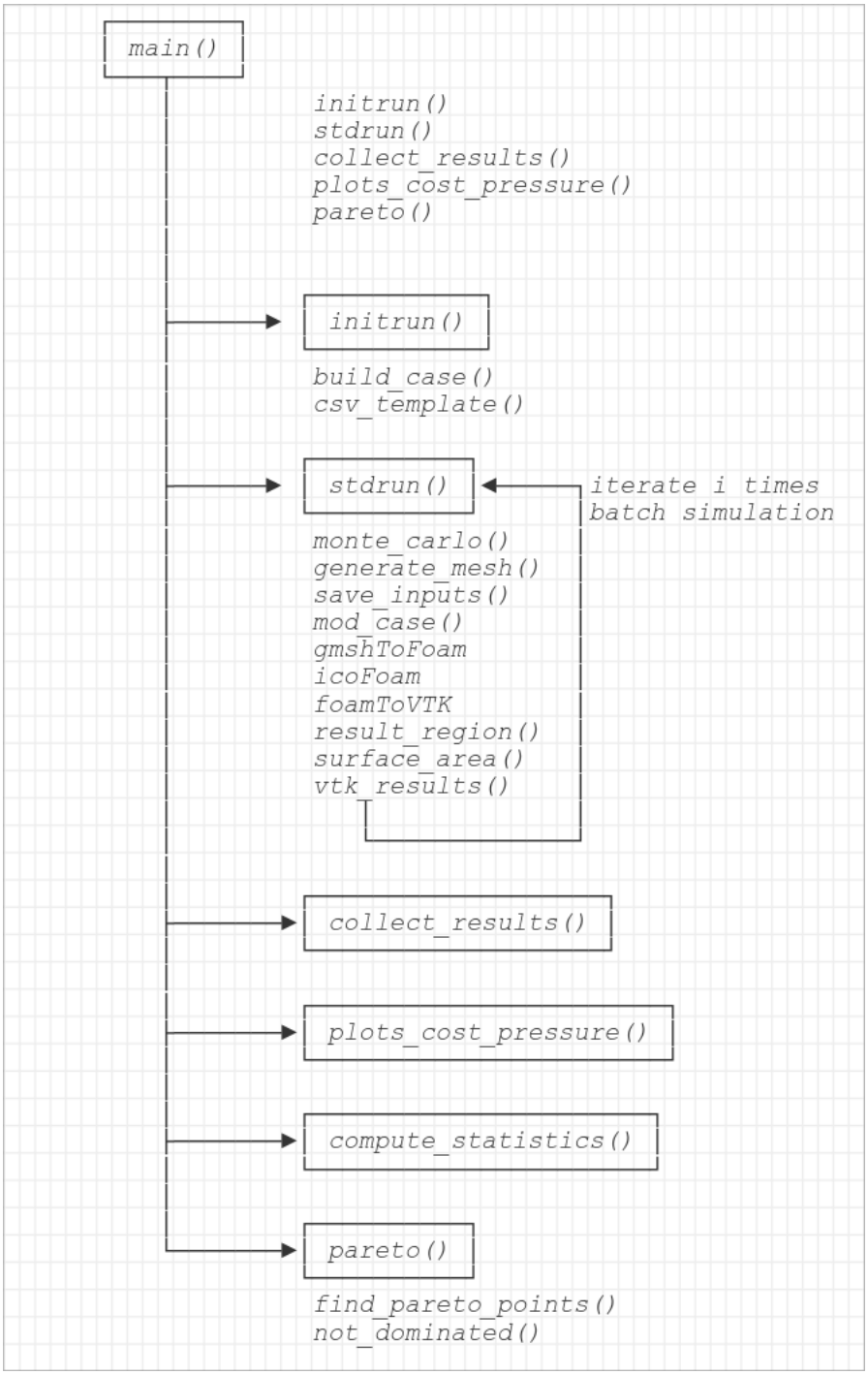


Figure 4.18: Functional architecture of the program

When *run.py* is executed, `main()` receives as an argument the number of simulations `i` to be performed. `initrun()` creates a template simulation case with the basic configuration files and creates CSV templates for storing the results.

The function `stdrun()` is executed `i` times, running each time a simulation instance. Every simulation instance creates a new directory and copies the template files there (`cpdir()`). Then generates a new mesh with random parameters (`generate_mesh()`), updates the rest of the inputs to new random values (`monte_carlo()`), and save the new parameters to a JSON file (`save_inputs()`). These new input parameters are writen to the base template files by the `mod_case()` function. Afterwards, PyFoam utilities convert the Gmsh formated mesh to a Foam mesh, run the solver icoFoam, and convert the produced results to VTK format. Eventually, `vtk_results` extracts geometrically restricted results from VTK files to save them in a CSV file containing simulation number, time, point coordinates, pressure, velocity components, and area.

The data in this CSV is mined by `collect_results()` using a Pandas dataframe to create another CSV containing only the simulation number, the cost of the material per thickness unit, and the average pressure at the small inlet in the last time step. The CSV table is used to plot simulation number, average pressure and material cost against each other using matplotlib (`plots_cost_pressure()`). In addition to that, `compute_statistics()` calculates several statistical metrics for average pressure and material cost variables such as maximum, minimum, average, median, standard deviation, and variance. In the end, `pareto()` calculates the pareto optimal points and plots the same optimal points, isolated and with the rest of the data, and the pareto fronier.

Chapter 5

Results

5.1 Explored solutions

Running the implemented program for n iterations creates and solves n simulation cases. From these n cases, different statistical parameters can be calculated. The Monte Carlo generation of geometric characteristics makes the surface area of the simulated fluid body an independent variable. The dependent variable of the problem becomes the average pressure at the small inlet. This average inlet pressure would be the pressure a pump would need to provide. The statistics below describe the dataset previously presented in figure 4.13.

	Number of samples	400
	Cost $[e/mm]$	**Average inlet pressure**$[Pa]$
Maximum	2.7052	41.0378
Minimum	1.2249	-0.1619
Mean	1.9376	1.8112
Median	1.9122	0.7217
Standard deviation	0.3843	3.4118
Variance	0.1477	11.6404

Table 5.1: Stadistical parameters of the bulk of simulations

5.2 Optimal solutions

Pareto optimization selects from the dataset the best cases in at least one of the objectives. As explained previously, no single Pareto optimal solution exists, but many solutions which compose the Pareto frontier. Table 5.2 presents the values of the objective variables for the optimal cases.

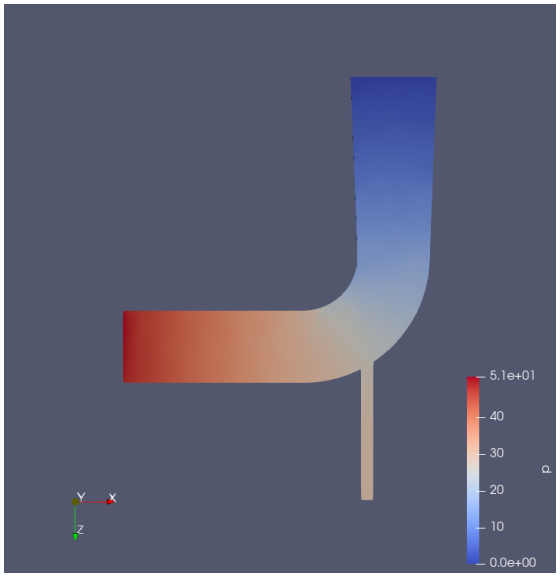
Optimal cases	**Cost** $[e/mm]$	**Average inlet pressure** $[Pa]$
Case s4	1.2249	0.6634
Case s55	1.4349	-0.1619
Case s169	1.3824	-0.1166
Case s203	1.2826	0.0216
Case s255	1.3181	-0.0048
Case s385	1.2785	0.2894

Table 5.2: Cost and average inlet pressure of the Pareto optimal cases.

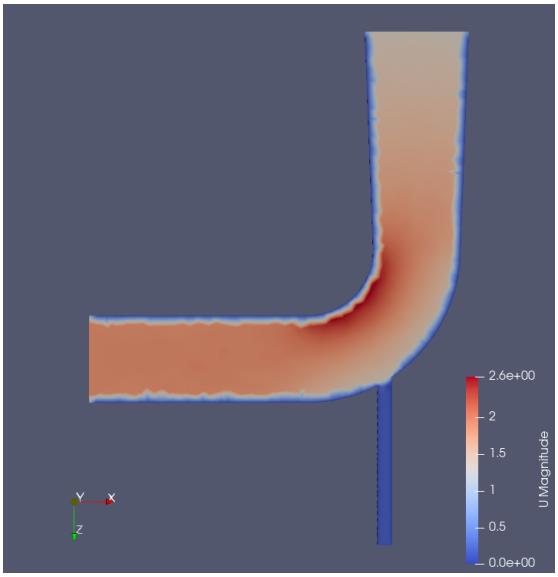
Some of the optimal cases have a negative inlet pressure. Negative pressures araise from the random parameter generation as the pressure in the small inlet is a dependent variable. These negative pressures occur for certain diameter settings because air is sucked inwards from the inside the geometry. This is a phenomena which carburetors take advantage of to mix air and fuel.

Parameter	**Case s4**	**Case s58**	**Case s169**	**Case s203**	**Case s255**	**Case s385**
ox $[m]$	0	0	0	0	0	0
oy $[m]$	0	0	0	0	0	0
oz $[m]$	0	0	0	0	0	0
a0 $[rad]$	6.283185	6.283185	6.283185	6.283185	6.283185	6.283185
a1 $[rad]$	1.570796	1.570796	1.570796	1.570796	1.570796	1.570796
l1 $[m]$	0.125	0.125	0.125	0.125	0.125	0.125
l2 $[m]$	0.125	0.125	0.125	0.125	0.125	0.125
l3 $[m]$	0.125	0.125	0.125	0.125	0.125	0.125
r1 $[m]$	0.024361	0.038984	0.023717	0.023565	0.028882	0.021438
r2 $[m]$	0.020449	0.021457	0.02338	0.021862	0.021510	0.021606
r3 $[m]$	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625
r4 $[m]$	0.020449	0.021457	0.02338	0.021862	0.021510	0.021606
r5 $[m]$	0.003226	0.004173	0.004209	0.003028	0.003144	0.004462
r6 $[m]$	0.003226	0.004173	0.004209	0.003028	0.003144	0.004462
area $[m^2]$	0.051506	0.060336	0.058128	0.053935	0.055425	0.053763
temperature $[K]$	281.44	312.40	270.06	269.75	272.40	302.22
big_outlet_presure $[Pa]$	101172.3	101091.7	101168.3	101249.8	101097.0	101192.4
big_inlet_speed $[m/s]$	1.8933	1.9349	1.9760	1.5240	1.4852	1.4906
small_inlet_speed $[m/s]$	-0.1244	-0.1269	-0.1367	-0.1038	-0.0998	-0.1018
kinematic_viscosity $[m^2/s]$	1.426E-05	1.583E-05	1.369E-05	1.367E-05	1.381E-05	1.532E-05

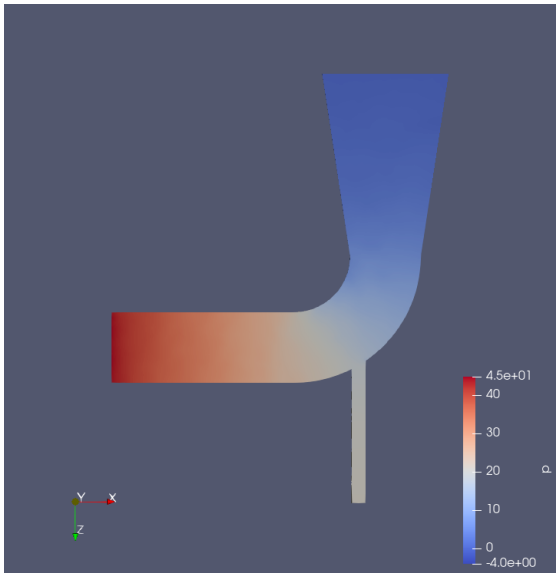
Table 5.3: Parameters used in the optimal cases.



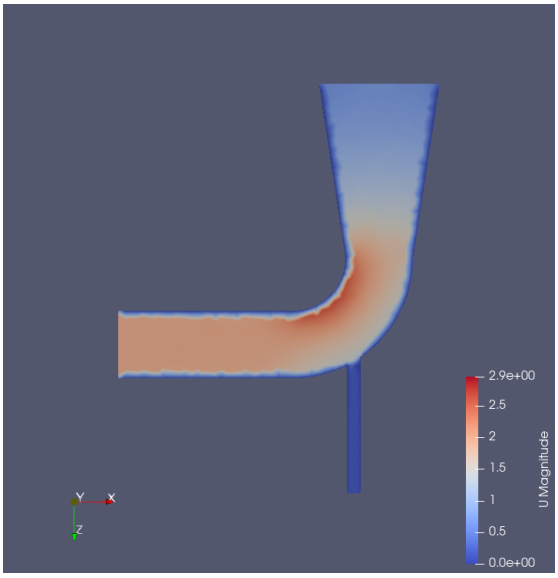
(a) Case *s4* initial pressure distribution



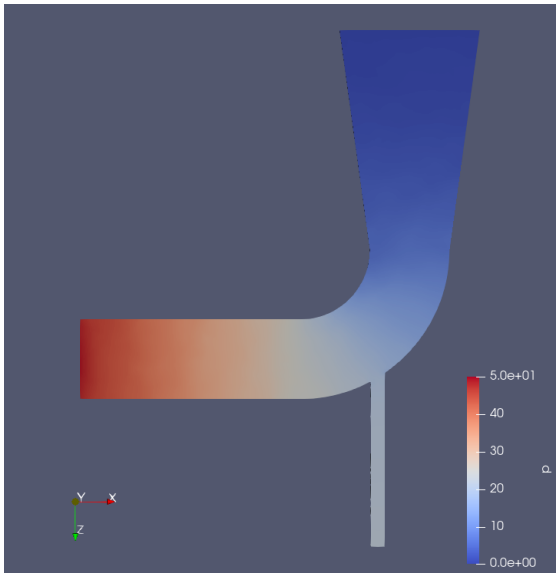
(b) Case *s4* final velocity distribution



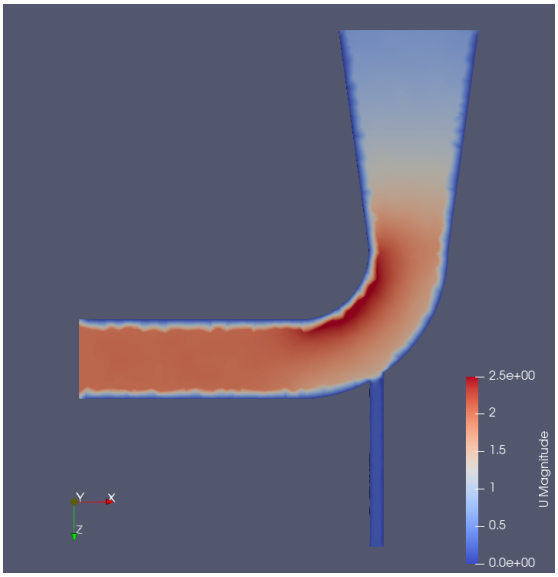
(a) Case *s58* initial pressure distribution



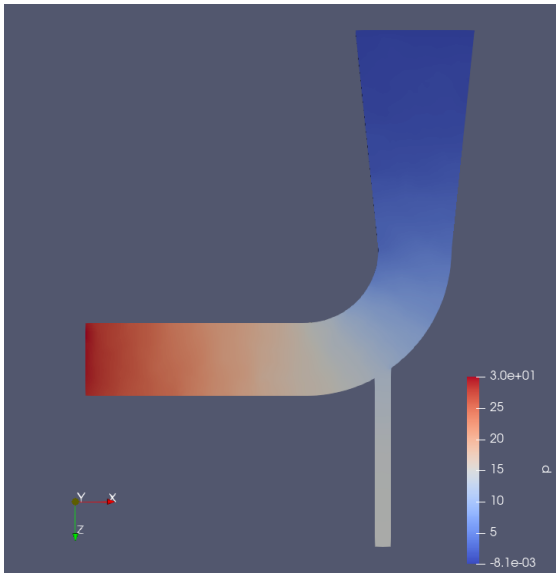
(b) Case *s58* final velocity distribution



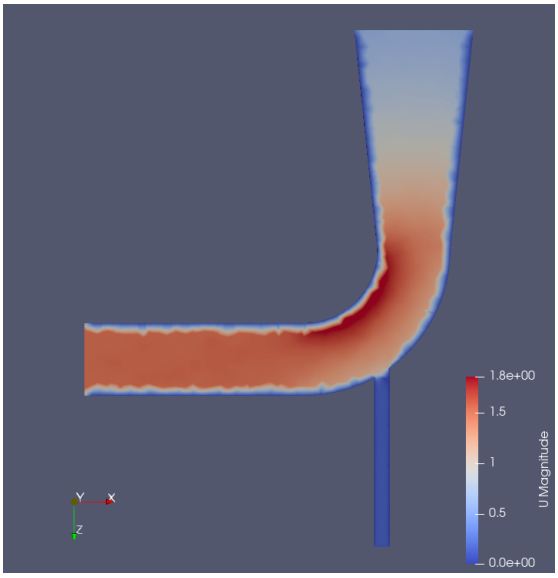
(a) Case *s169* initial pressure distribution



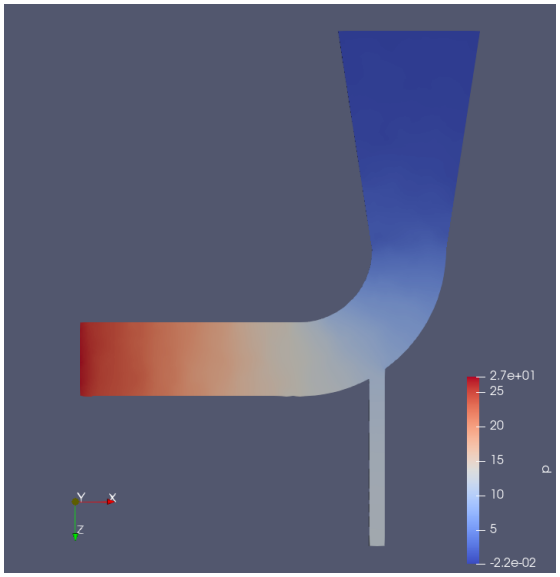
(b) Case *s169* final velocity distribution



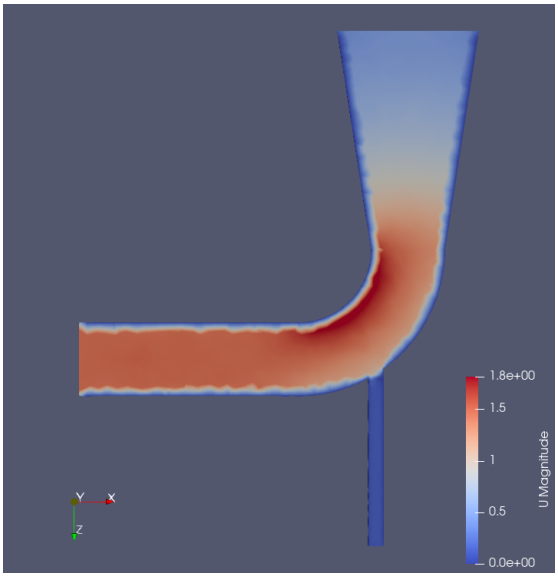
(a) Case *s203* initial pressure distribution



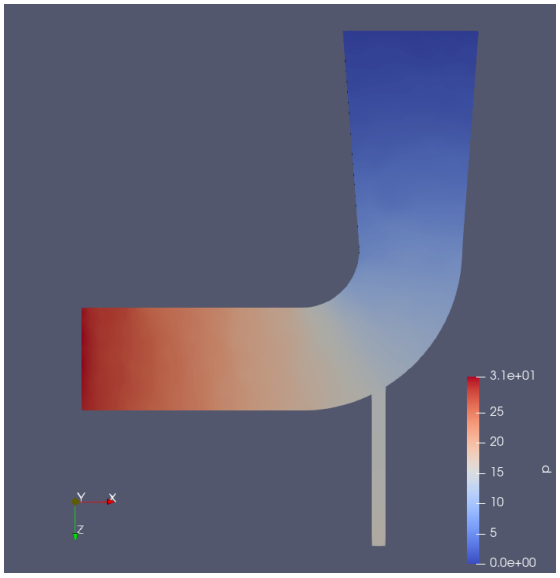
(b) Case *s203* final velocity distribution



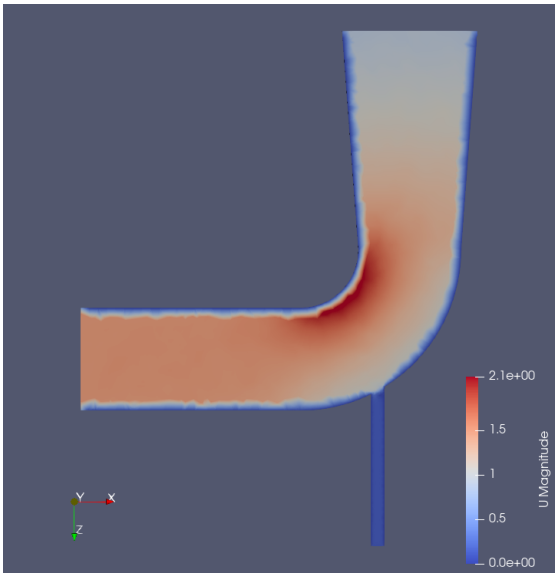
(a) Case *s255* initial pressure distribution



(b) Case *s255* final velocity distribution



(a) Case *s385* initial pressure distribution



(b) Case *s385* final velocity distribution

5.3 Comparison

5.3.1 Deviation

As a comparison metric the deviation from the mean cost and average inlet pressure of the dataset can be calculated using the formula:

$$\text{Deviation } = X - \bar{X} \tag{5.1}$$

where X is the actual value of the variable and $\bar{X}$ is the mean of such value in the dataset. If it is done for all the valid optimal cases, it results in:

	Mean cost $[e/mm]$	**Mean average inlet pressure** $[Pa]$
	1.9376	1.8112
Optimal cases	**Mean cost deviation** $[e/mm]$	**Mean average inlet pressure deviation** $[Pa]$
Case s4	-0.7126	-1.1478
Case s58	-0.5026	-1.9731
Case s169	-0.5551	-1.9279
Case s203	-0.6549	-1.7895
Case s255	-0.6194	-1.8160
Case s385	-0.6590	-1.5217

Table 5.4: Cases deviation from the mean

All values are negative since, for both parameters, these solutions are smaller than the mean values amongst all data, and, therefore being better values than most of the solutions obtained.

5.3.2 Optimal amongst optimals

Adding the heuristic of selecting the optimal case which has the best performance for both objectives, an optimal amongst optimal cases can be selected. Such case can be seen clearly as the central on in the Pareto frontier from figure 4.16. This is case s203 with a cost of 1.2826 e/mm and an average inlet pressure of 0.0216 Pa.

Chapter 6

Conclusion

The development of the SLPO (Simulation Launcher & Pareto Optimization) algorithm successfully met the primary objectives of automating fluid dynamics simulations and identifying optimal designs through Monte Carlo sampling and Pareto optimization. By leveraging open-source tools such as OpenFOAM for CFD, Gmsh for mesh generation, and Python for scripting, the project established a flexible, reproducible workflow for running multiple simulations. This workflow effectively navigated the design space, identifying Pareto-optimal solutions that balanced the objectives of minimizing inlet pressure and material cost.

Monte Carlo methods played a central role in generating randomized geometrical and physical parameters, allowing for broad exploration of the design space. The model's parametrization, including mesh geometry and boundary conditions, was essential in creating diverse simulation cases that represented various design possibilities. The incorporation of both geometrical and physical variables emphasized the importance of well-defined model parameters for accurate fluid dynamics simulations.

The computational efficiency of the workflow was enhanced by a detailed mesh time and computational cost analysis, which helped balance mesh resolution and simulation time. This trade-off analysis was critical in selecting appropriate mesh sizes for the final simulations, ensuring robust performance without excessive computational demands.

Pareto optimization provided an effective approach to multi-objective optimization, particularly in reducing material costs and minimizing inlet pressure. The visualization of the Pareto frontier allowed for insightful comparisons of the trade-offs between these competing objectives, offering both qualitative and quantitative guidance in identifying optimal design solutions.

While the proposed workflow was effective, it has limitations in its current form. The simplified physical model excluded important factors like compressibility and temperature variations, which are crucial in many real-world fluid dynamics scenarios. Incorporating these factors in future work would enhance simulation accuracy and realism. Additionally, while Monte Carlo sampling and basic Pareto optimization were sufficient for this study, more advanced optimization techniques, such as evolutionary algorithms like NSGA-II or SPEA2, could provide more efficient searches for globally optimal solu-

tions, particularly in more complex design spaces.

In conclusion, the SLPO algorithm offers a robust framework for automating and optimizing fluid dynamics simulations in a multi-objective context. Although the study focused on a simplified system, the methodology demonstrates significant potential for more complex and realistic engineering applications. Expanding the approach to address additional objectives, such as energy efficiency or environmental impact, and applying it to more intricate fluid systems would further enhance its industrial relevance. This work highlights the effectiveness of combining open-source CFD tools and optimization techniques, paving the way for future advancements in automated design optimization.

Bibliography

[1] I. G. Currie, *Fundamental mechanics of fluids*, eng, 4th ed. Boca Raton: CRC Press, 2013, ISBN: 9781439874608.

[2] M. O. Deville, *An Introduction to the Mechanics of Incompressible Fluids*, eng, 1st ed. 2022. Cham: Springer Nature, 2022, ISBN: 3-031-04683-8.

[3] OpenStax, *Viscosity and Laminar Flow*. 2023, ch. 14.9, Accessed on 2024-06-23. [Online]. Available: https://phys.libretexts.org/Bookshelves/University_Physics/University_Physics_(OpenStax)/Book%3A_University_Physics_I_-_Mechanics_Sound_Oscillations_and_Waves_(OpenStax)/14%3A_Fluid_Mechanics/14.09%3A_Viscosity_and_Turbulence.

[4] E. L. K. C. J. T. J. H. Williams; and F. Paillet, *Reynolds Number as an Indicator of Flow Regime*. Groundwater Project, 2022, ch. 4.2, Accessed on 2024-06-28, ISBN: 978-1-77470-040-2. DOI: https://doi.org/10.21083/978-1-77470-040-2. [Online]. Available: https://books.gw-project.org/introduction-to-karst-aquifers/chapter/reynolds-number-as-an-indicator-of-flow-regime/.

[5] S. Kumar, *Fluid Mechanics (Vol. 1): Basic Concepts and Principles*, eng, Fourth edition. Cham: Springer International Publishing AG, 2022, vol. 1, ISBN: 9783030997618.

[6] R. D. Martin, E. Neary, J. Rinaldo, and O. Woodman. "10.1: Momentum." Accessed on June 19, 2024, Physics LibreTexts. (2023), [Online]. Available: https://phys.libretexts.org/Bookshelves/University_Physics/Book%3A_Introductory_Physics_-_Building_Models_to_Describe_Our_World_%28Martin_Neary_Rinaldo_and_Woodman%29/10%3A_Linear_Momentum_and_the_Center_of_Mass/10.01%3A_Momentum.

[7] X. Oliver Olivella and C. Agelet de Saracibar Bosch, *Continuum Mechanics for Engineers. Theory and Problems*, eng. 2017.

[8] S. R. Bistafa, "200 years of the navier-stokes equation," *arXiv preprint arXiv:2401.13669*, 2022, Accessed on June 23, 2024. [Online]. Available: https://arxiv.org/abs/2401.13669.

[9] P. Knabner and L. Angerman, *Numerical Methods for Elliptic and Parabolic Partial Differential Equations* (Texts in Applied Mathematics, 44), eng, 1st ed. 2003. New York, NY: Springer New York, 2003, ISBN: 1-280-18853-7.

[10] R. W. Shonkwiler and F. Mendivil, *Explorations in monte carlo methods* (Undergraduate texts in mathematics), eng. Dordrecht, Netherlands ; Springer, 2009, ISBN: 1-280-38497-2.

[11] C. Geuzaine and J.-F. Remacle, "Gmsh: A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities," *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[12] G. D. Team, *Gmsh 4.13.1*, Gmsh, 2021. [Online]. Available: `https://gmsh.info/doc/texinfo/gmsh.html`.

[13] M. d. Berg, *Computational geometry: algorithms and applications*, eng, 3rd ed. Berlin: Springer, 2008, ISBN: 9783642096815.

[14] S. Rebay, "Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm," *Journal of Computational Physics*, vol. 106, no. 1, pp. 125–138, 1993. DOI: `10.1006/jcph.1993.1123`.

[15] C. Wollblad. "Your guide to meshing techniques for efficient cfd modeling." Accessed on 2023-07-01, COMSOL. (2018), [Online]. Available: `https://www.comsol.com/blogs/your-guide-to-meshing-techniques-for-efficient-cfd-modeling`.

[16] O. Foundation, *Openfoam v11*, Version 11, 2023. [Online]. Available: `https://github.com/OpenFOAM/OpenFOAM-11`.

[17] O. Foundation, *Openfoam: The open source cfd toolbox*, 2023. [Online]. Available: `https://www.openfoam.com/`.

[18] O. Foundation, *Openfoam documentation: Overview*, Accessed on 2024-06-23, 2023. [Online]. Available: `https://www.openfoam.com/documentation/overview`.

[19] O. Foundation, *Icofoam v11*, Version 11, 2023. [Online]. Available: `https://github.com/OpenFOAM/OpenFOAM-11/tree/master/applications/solvers/incompressible/icoFoam`.

[20] OpenFOAM Foundation, *Icofoam*, version v2106, Accessed on 2021-06-30. [Online]. Available: `https://www.openfoam.com/documentation/guides/latest/doc/guide-applications-solvers-incompressible-icoFoam.html`.

[21] J.-C. Pusch, *Pyfoam v2023*, Version 2023, 2023. [Online]. Available: `https://github.com/OpenFOAM/PyFoam`.

[22] B. F. W. Gschaider, *Pyfoam*, Accessed on 2024-06-23, 2023. [Online]. Available: `https://pypi.org/project/PyFoam/`.

[23] O. C. Jens-Christian Pusch, *Pyfoam: Python library for openfoam*, Accessed on 2024-06-23, 2023. [Online]. Available: `https://openfoamwiki.net/index.php/Contrib_PyFoam`.

[24] K. Inc., *Paraview v5.11*, Version 5.11, 2023. [Online]. Available: `https://github.com/Kitware/ParaView`.

[25] K. Inc., *Paraview: Open-source scientific visualization*, Accessed on 2024-06-23, 2023. [Online]. Available: `https://www.paraview.org/`.

[26] K. Inc., *Paraview documentation*, Accessed on 2024-06-23, 2023. [Online]. Available: `https://docs.paraview.org/en/latest/`.

[27] Y. Ye, *Mathematical optimization models and applications*, `https://web.stanford.edu/class/msande311/lecture01.pdf`, Accessed on 2023-07-03, Stanford, CA 94305, U.S.A., 2022.

[28] S. Datta and J. P. Davim, *Optimization in Industry Present Practices and Future Scopes* (Management and Industrial Engineering), eng, 1st ed. 2019. Cham: Springer International Publishing, 2019, ISBN: 3-030-01641-2.

[29] *Europe stainless hot rolled coil 304 price forecast*, `https://mepsinternational.com/gb/en/products/europe-stainless-steel-prices`, Accessed on 2024-07-04, Jul. 2024.

[30] *Weight & density of stainless steel 304, 316, 316l & 303*, `https://www.theworldmaterial.com/weight-density-of-stainless-steel/`, Accessed on 2024-07-04.

[31] A. Chinchuluun, *Pareto optimality, game theory and equilibria* (Springer optimization and its applications ; v. 17), eng, 1st ed. 2008. New York: Springer, 2008, ISBN: 1-281-49098-9.

[32] J. Branke, *Multiobjective Optimization Interactive and Evolutionary Approaches* (Theoretical Computer Science and General Issues, 5252), eng, 1st ed. 2008. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ISBN: 3-540-88908-6.

[33] M. T. M. Emmerich and A. H. Deutz, "A tutorial on multiobjective optimization: Fundamentals and evolutionary methods," eng, *Natural computing*, vol. 17, no. 3, pp. 585–609, 2018, ISSN: 1567-7818.