IMPERIAL

Development of Mesh Deformation Methods for External Aerodynamics Simulations

Author

Alexandre Youssef

CID: 2520716

Supervisor

Dr Heikki Kahila

Internal reader

Prof. Joaquim Peiro

A Thesis submitted in fulfillment of requirements for the degree of

Master of Science in Advanced Aeronautical Engineering

Department of Aeronautics

Imperial College London

September 2024

Acknowledgements

This work concludes my studies as an Aeronautical Engineering master's student at Imperial College London. It follows my previous master's degree in Applied Mathematics at Université Catholique de Louvain, and my Bachelor's degree in Applied Mathematics and Mechanical Engineering. One of my goals in doing this work was to show how closely these two fields are related and how strongly they will continue to be so in the future.

To begin with, I would like to thank my supervisor from the McLaren Racing CFD team, Heikki Kahila. It was a pleasure to be mentored by you, to collaborate on methodologies, and to develop tools that I hope will be useful in the future. During my internship, I had a unique experience discovering industrial and cutting-edge applications of CFD, as well as gaining insight into the fast-paced Formula 1 environment from an internal perspective. I would also like to extend my thanks to the CFD and Aerodynamics teams for their interest in my work and for creating a supportive atmosphere. In particular, I would like to thank Julien Hoessler for giving me the opportunity to join his group and for his help during my internship. I am confident that McLaren will continue to perform well, considering the team's commitment and the positive results we are seeing this year.

On the Imperial College London side, I would like to thank Oliver Buxton for his support in helping me secure this internship. Lastly, I would like to extend my thanks to Professor Joaquim Peiro for reviewing my work and providing his valuable experience and feedback on meshing tools. It is always a pleasure for me to learn from and receive advice from experts at the top of their discipline.

Finally, I wish to acknowledge my parents for their moral support and interest in my work, as well as for their support during this year in London.

Contents

Abstract

This study focuses on the development of mesh deformation utilities within the OpenFOAM software [1] and their application in Formula 1 and motorsport contexts. Two key elements drive the investigation of mesh deformation: 1) enable complex aerodynamic simulations with moving geometries and 2) accelerate the mesh generation process during the preprocessing stage. The first utility developed deforms a mesh during runtime, using a prescribed body motion. The second one addresses mesh deformation between two different states of a geometry during the preprocessing stage and considers advanced geometry-to-geometry mapping methods. The method employed is known as Point-Set Registration.

Firstly, a rigid motion is applied to an Ahmed body benchmark to evaluate the quality and robustness of mesh deformation during runtime. Next, the efficiency and accuracy of Point-Set Registration methods are assessed through analyses of convergence and computational cost. Mesh deformation is then used as part of preprocessing, where the mesh is deformed between two different configurations of a same body. A wing profile and an F1-related suspension wheel geometry are tested. Both rigid and non-rigid mesh deformations are successfully implemented. A time gain of up to 45% is achieved compared to a traditional remeshing method. In industries with limited resources, this approach could thus allow to allocate more time to aerodynamic simulations. While the wing case yields accurate results, the second geometry highlights the need for further improvements at the junction between some of its subparts. The study concludes with recommendations for improving the code, including the integration of motion equations, machine learning algorithms, and enhancements in algorithmic efficiency.

Introduction

Understanding and optimizing the aerodynamics of motorsport cars is central for enhancing their performance. For instance, it allows reducing drag, tuning the downforce and improving stability, which leads to on-track lap-time improvements. In the process, aerodynamic flows around a motorsport car are mainly studied through computational simulations, wind tunnel testing, and on-track testing. Computational Fluid Dynamics (CFD) allows for airflow pattern analysis and design optimization. Wind tunnel testing allows for the evaluation of scaled models for aerodynamic performance and for the validation of CFD results. On-track testing guarantees the performance of the car under real racing conditions, which ensures theoretical and experimental changes translate effectively onto the race day. Integrating these methods mentioned assures continuous innovation and performance enhancement in motorsport. This work focuses on the CFD side of aerodynamic analysis.

When tuning the computational setup of a CFD simulation, like adjusting the car's position (wheels in a turn for example) or the ride height, a spatial discretization (or meshing/gridding) of the computational domain is carried out as a part of an automated industrial framework. It is a time-consuming process as it can require a manual user interaction; geometries can be complex and the number of discretization points can reach hundreds of millions. This is particularly a challenge in Formula 1 (F1) because the car has open wheels, open suspension geometry, and external wings on the front and back of the car. Instead of remeshing when dealing with small perturbations in geometry, one could deform the mesh node points explicitly and save considerable amount of time.

Furthermore, if this is possible, future simulations could feature run-time deformation of car parts, like time-dependent suspension kinematics movements. For example, combined with increasing computational resources, one could envision a transient CFD solution for the porpoising problem [2]. Porposing is a phenomena that happens when a unstable flow is formed under the car at high speeds, as it oscillates up and down. This phenomenon is primarily influenced by ground effect aerodynamics, where the shape of the car's floor and its proximity to the track surface are critical

factors. With enough computational resources, it is possible to simulate this issue, but it would require mesh deformation adequate mesh capabilities. Another motivation of using mesh motion techniques is the potential use of active aerodynamics in motorsports and F1 in the future. By deforming the mesh around a moving wing for example, it could be possible to model the flow dynamics and improve the car's design from it. Another use case could be incorporation of adjoint shape optimization, where non-rigid geometry deformation is guided by mathematical sensitivities of the flow gradients.

The purpose of this work is to explore mesh motion techniques applied to motorsport, specifically Formula 1. This project is based on the OpenFOAM C++ library [1], used extensively in the academy and the industry. It is an open-source finite volume solver for the Navier-Stokes equations. The tools are in this project are implemented in the OpenFOAM-dev version[1].

The OpenFOAM Software

To understand the spatial discretization and the subsequent mesh deformation, a brief summary on the numerical approach of OpenFOAM [1] is considered. The idea is to discretizes the governing equations over a finite volume, represented by control volumes. The method accounts for the balance of fluxes from inflows and outflows at the boundaries, along with any internal sources within the volume. This ensures that conservation equations are applied accurately across the entire system. The main equation in OpenFOAM's finite volume method is the conservation equation [3]

$$\frac{\partial(\rho\phi)}{\partial t} + \underbrace{\nabla\cdot(\rho\phi\mathbf{u})}_{\text{Convection term}} = \underbrace{\nabla\cdot(\Gamma\nabla\phi)}_{\text{Diffusion term}} + \underbrace{S_\phi}_{\text{Source term}} \tag{1}$$

where ϕ represents the variable of interest (e.g., velocity or pressure), ρ is the density, $\mathbf{u}$ is the velocity field, Γ is the diffusivity coefficient, and S_ϕ denotes source terms. This conservation equation is discretized using a finite volume approach and integrals over the control volumes are approximated. To deal with with the discretization, OpenFOAM includes various time discretization schemes (Euler, Crank Nicholson etc.) and space discretization schemes (Gauss linear, Gauss cubic etc.). For more details, the reader is referred to [1], as this work focuses exclusively on the meshing process, that comes prior to solving the Navier-Stokes equations.

The most direct way to solve the Navier-Stokes equations is by discretizing the volume down to the smallest eddies relevant for the problem (i.e. the Kolmogorov scale). However, this approach is computationally expensive, especially in cases like Formula 1, where the Reynolds number is

[1] `https://github.com/OpenFOAM/OpenFOAM-dev/tree/master`

of the order of $\mathcal{O}(10^7)$. To address these limitations, turbulence modelling techniques exist, and are implemented in OpenFOAM. Smaller scales are modelled to reduce the computational cost, while larger scales are solved [1]. Those techniques include Reynolds-Averaged Navier-Stokes (RANS) and Large Eddy Simulation (LES). RANS simplifies the turbulence by modelling 100% of the scales. It thus computes the mean flow field while modeling the effects of turbulence, which significantly reduces computational costs. RANS is widely used for steady state simulations. On the other side, LES resolves larger turbulent structures while modeling the smaller scales. It provides a more accurate representation of transient turbulent flows compared to RANS but requires more computational resources [1]. These methods balance accuracy and computational efficiency and are a viable for industrial applications. This work focuses on meshing tools that can be combined with those flow solvers to provide advanced aerodynamics simulations.

Mesh Deformation Utilities

OpenFOAM offers a wide range of solvers, capable of modelling various physics. Those range from the Navier-Stokes equations to magnetohydrodynamics. By using OpenFOAM's generalized framework, each solver can be extended to handle moving geometries. Different approaches exist to solve a flow past a moving geometry; those are referred in the literature as mesh motion techniques [4].

The choice of a mesh motion technique is case-dependent, as each one has its pros and cons. For instance, when the object motion is large, so-called overset grid methods are of interest. An example is a rotorcraft simulation, where the mesh has to move with the rotating parts without distorting the entire grid [5]. On the other hand, in fluid-structure interaction problems where the structure undergoes smooth deformations, a mesh deformation framework is preferable [6]. By mesh deformation, we refer to a state change of the spatial discretization which moves node points but retains the mesh topology structure, i.e. node-to-node connectivity.

This work focuses on mesh deformation tools applied to motorsport. After investigating existing deformation techniques, two frameworks are investigated. The first one is to robustly deform a mesh during runtime. The second one acts as a preprocessing step to deform a mesh between two states of an object, without given information on the equations of motion between those. Both frameworks have applications in the industry, and their implementation are undertaken in this work.

Aim and Objectives

The main aim of this work is to provide an approach on mesh deformation techniques in a motorsport context. Two key reasons motivate the investigation of deforming meshes. The first is that mesh deformation enables the modeling of body-motion-dependent flow characteristics. The second is that it reduces computational costs by eliminating the need for remeshing, while reducing manual user interactions during the meshing process.

The main objectives of this work are:

1. Develop and implement two variations of a motorsport relevant and currently existing mesh deformation utility of OpenFOAM. The first one tackles a runtime prescribed motion of a body. The second one is a preprocessing step to deform a mesh between two states of a body, without needing information on the motion between those.

2. Provide a methodology to non-rigidly deform a mesh in the preprocessing stage and to map a geometry from an initial position to a final position. Point Set Registration (PSR) techniques are investigated for this purpose.

3. Test and validate the concepts given above on simple geometries such as an Ahmed body and a 3D wing, and on more complex geometries mimicking F1 car features. For the runtime mesh deformation, an Ahmed body is considered. For the preprocessing mesh deformation, the first geometry is a non-rigid extruded airfoil deformation. The second one is a car-like model with a large steering angle and suspension changes. The objective of the latter is to test the utility on a complex case so that limitations and directives for further improvements can be outlined.

Chapter 1

Background and Literature Review

This chapter focuses on tools discussed in the literature that are applicable in the context of mesh deformation. The review is divided into two parts. Firstly, the different types of mesh deformation methods are presented. Secondly, techniques for finding the mapping between two point clouds are covered. Those are required to perform subsequential mesh deformation and to avoid remeshing of the whole domain. This theoretical background is used to develop an algorithm for mapping an arbitrary body from one configuration to another without prior knowledge of its kinematics.

As a starting point, let us consider an initial mesh around an arbitrary object. Let us further assume that this object is desired to be moving in space. According to Bos et al. [4], there are three mesh-based (mesh motion) approaches to model a flow past a moving object: Immersed Boundary Methods (IBM), Overset Grid Methods, and Mesh Deformation Methods.

Immersed Boundary Methods were introduced by Peskin [7] in the 70's for fluid-structure interaction purposes. This method does not deform the mesh properly speaking. Here, the interaction between the fluid and the structure is handled by calculating forces based on the difference between the structure's positions (Lagrangian markers) and the fluid's grid (Eulerian grid). These forces are then distributed onto the fluid's grid. It allows the structure to move through the fluid by updating its position according to the fluid's local velocity. The advantage of IBM is a simpler grid generation because the grid does not need to conform to the shape of the structure. However, it suffers from accuracy issues near boundaries, as it relies on interpolating forces between the Lagrangian markers and Eulerian grid [8].

Overset Grid Methods [9] involve two grids overlapping; a background one and another one attached to the body. Both grids exchange information to deform the mesh. They are not deformed but rather solidly moved on top of each other. An advantage of this method is that components

of a complex body can be added or removed without remeshing the entire configuration [9]. On the other side, a challenge is that conservation is lost through interpolation during the information exchange between the two grids [4]. This could lead to inaccuracies in the forces computation, especially at high velocity flows [10].

1.1 Mesh Deformation Methods

In Mesh Deformation Methods (also known as Arbitrary Lagrangian-Eulerian (ALE) Methods [4]), the equations are discretized on a deforming/moving mesh. The main challenge is thus to determine of how each mesh point moves based on the body's movement [6]. The mesh points can be split into two parts; the boundary points around the moving object and the remaining mesh points, called internal points. The boundary points have a given motion based on the prescribed motion of the body. The methods for determining the internal points motions are outlined below.

1.1.1 Partial Differential Equation (PDE)-based Methods

Those methods make the parallel between the deforming computational domain and a solid body undergoing internal stresses [4, 11]. Since the underlying equation is non-linear and expensive to solve, alternatives are developed in the literature. Those alternatives use simpler Partial Differential Equations (PDE's), making the underlying linear system to solve sparse and hence much faster. Two main alternatives exist, Laplacian equation-based and stress equation-based. A drawback is that this type of method involves solving a PDE over the entire computational domain, which can add considerable overhead [12]. Control of the motion constraints can be challenging.

Laplace Equation-based Method

Let us consider an arbitrary body being translated in the computational domain. One could consider that the movement of the mesh points is maximal close to the body. It then diffuses until it reaches zero far from the body. This leads to a Laplacian equation to solve over the whole domain to find the displacement/node-velocity field $\mathbf{x}$. Letting γ as called the motion diffusivity and r as the radius from the object boundary, the following equation is solved [4, 13]

$$\nabla \cdot (\gamma \nabla \mathbf{x}) = 0 \qquad \gamma(r) = \frac{1}{r^m} \tag{1.1}$$

Many choices exist for $\gamma(r)$, either independently point by point or through a function. A popular choice is a quadratic function ($m = 2$), but more exotic choices exist like exponentially decreasing functions [4]. OpenFOAM has this deformation method implemented ("displacementLaplacian-

FvMotionSolver"), with the freedom to choose the motion diffusivity and the prescribed motion of the body. This method is mostly suited for small amplitude translations with limited rotations. For larger motions, this method can introduce a high mesh skewness near moving walls [14].

Solid Body Rotation (SBR) Stress-based Method

This method is based on the linear elasticity equation [15]. The mesh is viewed as a body that deforms under a prescribed boundary displacement. Letting $\mathbf{x}$ as the position of an internal mesh point, $\mathbf{I}$ as the identity matrix and λ and μ as the Lamé constants, the solid body rotation stress can be derived. Letting $\lambda = -\mu$, the PDE to solve is of the form [4]

$$\nabla \cdot (\gamma \nabla \mathbf{x}) + \nabla \cdot (\gamma(\nabla \mathbf{x} - \nabla \mathbf{x}^T) - \lambda \, \mathrm{Tr}(\nabla \mathbf{x})\mathbf{I}) = 0 \tag{1.2}$$

where "Tr" is the trace operator. Similarly to the Laplacian approach, Equation (1.2) is equivalent to a linear system to solve, with γ to be specified as above. Its computational cost is similar to solving a Laplace equation [4]. In OpenFOAM, it is known as the "displacementSBRStressFvMotionSolver". It is more suited than the Laplacian method in some situations, especially when there is body rotation [14]. Indeed, in a case of a rotation, most cells are rotating to cope with the body rotation, avoiding a skewing of mesh cells. Nevertheless, PDE motion solvers in general suffer from a tendency of cells to shear, so that mesh quality can often degrade under fairly modest rotations [12]. The control of γ near geometry details can also be a challenge.

1.1.2 Radial Basis Function (RBF) Method

In this method, a radial basis function interpolation $k(\mathbf{x})$ is created by matching the displacement of the boundary mesh points (denoted $\mathbf{d}_{b_j}$) to the prescribed motion of the body. This interpolation is then used to find the deformation of the mesh internal points. According to [16], this method is more accurate and robust than the PDE based methods. It is also mesh independent and allows a node wise precision. The interpolation function is of the form [17]

$$k(\mathbf{x}) = \sum_{j=1}^{N_b} \alpha_j \phi(||\mathbf{x} - \mathbf{x}_{b_j}||) + q(\mathbf{x}) \tag{1.3}$$

where $\mathbf{x}_{b_j}$ are the boundary mesh points, N_b is the number of boundary mesh points, $q(\mathbf{x})$ is a polynomial, ϕ a given basis function and α_j unknown coefficients. $q(\mathbf{x})$ and α_j are found because the displacement of boundary points $\mathbf{d}_{b_j}$ is known, and using an additional condition. Those are

$$k(\mathbf{x}_{b_j}) = \mathbf{d}_{b_j} \qquad \text{and} \qquad \sum_{j=1}^{N_b} \alpha_j p(\mathbf{x}_{b_j}) = 0 \tag{1.4}$$

which has to hold for all polynomials $p(\mathbf{x})$ with a degree less or equal to the one of $q(\mathbf{x})$. Equations (1.4) form a linear system to solve. In general, it leads to a dense matrix system. It thus requires more computational resources than PDE ones which are sparse systems. Indeed, it needs to be solved directly by doing a LU decomposition, and has high memory/CPU requirements [14].

Once the interpolation function $k(\mathbf{x})$ from Equation (1.3) is determined, the displacement of the internal points $\mathbf{x}_{\text{in}_j}$ can be simply evaluated as $\delta\mathbf{x}_{\text{in}_j} = k(\mathbf{x}_{\text{in}_j})$. In the literature, two types of radial basis functions ϕ are commonly distinguished: those with compact support and those with global support [4]. Compact support basis functions allow tuning the sparsity of the system in Equation (1.4). The only public implementation of this method in OpenFOAM is in a different version[1] than used in this project, so the RBF method is not further investigated.

1.1.3 Spherical Linear Interpolation (SLERP) Method

For simplicity, let us first consider a translation over time t, denoted $\mathbf{t}(t)$. Denoting $\mathbf{x}$ as the mesh point coordinates, a scalar field $s(\mathbf{x})$, called the "scaling field", is determined over the whole domain, and each mesh point is moved depending on $s(\mathbf{x})$ and $\mathbf{t}(t)$. The method requires to specify an inner and an outer distance from the surface of the body, denoted r_{in} and r_{out}. Within the inner radius region r_{in}, the scaling factor is set to $s(\mathbf{x}) = 1$, meaning that the mesh node points are moved like the object motion itself. Outside of the outer radius region r_{out}, the points are not translated at all, meaning that the scaling factor is $s(\mathbf{x}) = 0$. In between, different choices exist to decrease the scaling factor from $s(\mathbf{x}) = 1$ to $s(\mathbf{x}) = 0$. It can for instance be linearly decreasing (denoted $s_{\text{lin}}(\mathbf{x})$), or under the form of a sinusoidal smoothing (denoted $s_{\text{sin}}(\mathbf{x})$).

A schematic of the concept is provided in Figure 1.1. In OpenFOAM, the smoothing correction is obtained by applying a correction to the linear scaling $s_{\text{lin}}(\mathbf{x})$, in the form $s_{\text{sin}}(\mathbf{x}) = 0.5 - 0.5\cos(s_{\text{lin}}(\mathbf{x}))\pi)$. An illustration of the smoothed scaling for $r_{in} = 2$ and $r_{out} = 4$ is provided in Figure 1.2. Once $s(\mathbf{x})$ is determined, each point is moved according to $\mathbf{x} = \mathbf{x} + s(\mathbf{x}) \cdot \mathbf{t}(t)$. It turns out that this approach is insufficient for complex body motions. The reason is that rotational motions around an axis cannot be accurately represented by a linear interpolation of translational displacements (mesh orthogonality is not preserved through a pure translation of its points [18]).

To fix that issue, Samareh [19] applied unit quaternions to mesh deformation. A description on quaternions is provided in Appendix A. Quaternions are widely used in computer graphics and attitude control of spacecrafts [18]. To preserve orthogonality properties, a quaternion at each boundary point is found by calculating the change of its orientation between the initial and de-

[1] `https://github.com/visiblehawk/foam-extend-4.1`

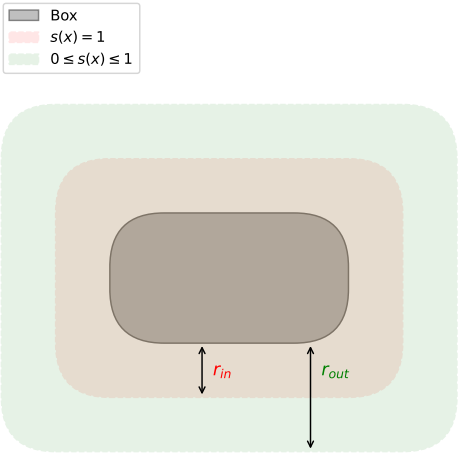

Figure 1.1: Concept of the "interpolating-SolidBodyMotion" solver on a 2D curved box.

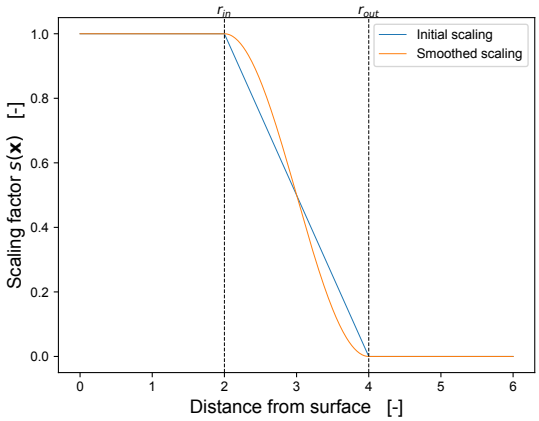

Figure 1.2: Illustration of the cosine smoothing ont the scaling factor $s(\mathbf{x})$.

formed meshes. The total displacement of any boundary cell is decomposed into two transformation processes: a rotation handled by a quaternion to preserve the orthogonality information, and a translation vector [18]. These two transformations are then propagated to all internal nodes in the volume to define their displacement. This propagation over the internal points is done via a Spherical Linear Interpolation (SLERP). The idea of the SLERP is to follow the shortest path on the surface of a sphere. The SLERP interpolation was originally introduced by Shoemake [20]. It enforces a smoothness of the mesh cells under rotations [12].

Let Q_0 be the quaternion describing the initial orientation of the body (time $t = 0$), $\mathbf{t}$ be the body displacement (translation vector) at time t and $s_i \in [0,1]$ be the scaling factor of point i. The new position $\mathbf{r}_i$ of a mesh point $\mathbf{p}_i$ can be computed as [18, 21]

$$\begin{cases} R_i & = s_i T + Q_{s_i} P_i Q_{s_i}^* \\ Q_{s_i} & = (Q Q_0^{-1})^{s_i} Q_0 \end{cases} \tag{1.5}$$

where Equation (1.5) is the SLERP formula between two unit quaternions (see Appendix A for a detailed explanation), $T = [0, \mathbf{t}]$ (quaternion notation of a pure translation vector), $P_i = [0, \mathbf{p}_i]$, $R_i = [0, \mathbf{r}_i]$, * is the conjugate operator and Q is the quaternion describing the state of the body at time t. For more detailed information, the reader is referred to [18]. The SLERP approach is faster than the other mesh deformation methods because it moves points explicitly without solving a linear system. Indeed, this method simply iterates over each mesh point to adjust its position. In OpenFOAM-dev, this deformation method is implemented under the "interpolatingSolidBody-Motion" utility.

1.2 Geometry Mapping

This part addresses tools that can be applied to find a mapping between two topologically similar geometries. It could then be used to provide information to OpenFOAM to deform a mesh during the preprocessing stage. A key question in this context is: Given an initial and final configuration of a body, how can the movement of each mesh point on the initial body's boundary be described?

1.2.1 Analytical Mapping Approach - Equations of Motion

A first way to tackle the problem is to determine analytically the equations of motion between the two configurations of the body. This task is out of scope of this work. In this project, extra information concerning the motion are neglected on purpose, as the objective is to have a generic method working for arbitrary types of geometries. The reason is that in some cases, such as such as a tyre deformation in a high-speed corner or aeroelasticity on an aircraft under high loads, determining the equations of motion is not always possible.

1.2.2 Machine Learning Approach - Autoencoders

If the mapping is not known analytically and if the two geometries are assumed to be topologically similar (a car stays a car), a way to tackle the problem is through unsupervised learning, specifically with autoencoders [22]. Let $\mathbf{X} \in \mathbb{R}^{N\times d}$ denote an initial point cloud and $\mathbf{Y} \in \mathbb{R}^{M\times d}$ a final one, $N \neq M$ being the size of the sets and d the dimension. Two sets of parameters are defined: θ_E for the encoder and θ_D for the decoder [22]. The goal is to learn a compact representation $\mathbf{Z}$ of the input surface $\mathbf{X}$ (depending on θ_E), such that $\mathbf{Z} = \text{Encoder}(\mathbf{X}, \theta_E)$ [23]. The model then reconstructs the target surface from this compact space, yielding $\hat{\mathbf{Y}} = \text{Decoder}(\mathbf{Z}, \theta_D)$. To determine θ_E and θ_D, a reconstruction loss function $\mathscr{L}_{\text{rec}}$ is minimized, through optimization algorithms such as the gradient descent. A popular choice for this loss function is of the form of a Mean Squared Error [22]

$$\mathscr{L}_{\text{rec}} = \frac{1}{M}\sum_{i=1}^{M} \|\mathbf{Y}_i - \hat{\mathbf{Y}}_i\|^2 \tag{1.6}$$

Litany et al. [23] proposes an approach to map non-rigid shapes by using an autoencoders. It is shown to effectively capture deformations. Similarly, Kingma and Welling [22] introduce "Variational Autoencoders" (VAEs), which are applied in surface mapping. VAEs is a probabilistic framework where the encoder maps the input surface to a distribution in the latent space rather than a fixed point. In a similar manner, Groueix et al. [24] use a convolutional autoencoder to learn a dense correspondence between 3D shapes. It encodes the surface into a compact space and decodes it to a canonical shape. It is validated on complex shapes such as human shapes [24].

1.2.3 Point Set Registration (PSR) Approach

Another approach to this challenge is to address it as a point-set registration (PSR) problem, commonly used in computer vision tasks like motion tracking and guided surgery [25]. Also known as point-cloud registration or scan matching, PSR involves computing a transformation that aligns two sets of points (potentially of different sizes) optimally, with the presence of noise. This problem assumes a meaningful correspondence between the points in both sets, implying topological similarity. Following [26, 27], the problem can be mathematically expressed as follows:

Let $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^T \in \mathbb{R}^{N \times d}$ and $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_M)^T \in \mathbb{R}^{M \times d}$ be two sets of d-dimensional points clouds of size N and M respectively. The set $\mathbf{X}$ is called the source point cloud, and the set $\mathbf{Y}$ the target point cloud. Denoting $\mathscr{T}$ as a chosen transformation set (rigid, affine, non-rigid etc.), the goal of the PSR problem is to find the optimal transformation model $T^*(\mathbf{X}) \in \mathscr{T}$ such that

$$T^* = \arg\min_{T \in \mathscr{T}} ||T(\mathbf{X}) - \mathbf{Y}|| \tag{1.7}$$

where the $||.||$ norm is to be chosen and where $\mathscr{T}$ is the set of all possible transformation within a specified domain. The norm is usually the Euclidian distance between each point in $\mathbf{x} \in \mathbf{X}$ and a corresponding point in $\mathbf{y} \in \mathbf{Y}$, so that the Mean-Squared Error (MSE) is minimized. In other words,

$$||T(\mathbf{X}) - \mathbf{Y}|| = \sum_{\mathbf{x} \in \mathbf{X}} ||\mathbf{x} - \mathbf{d}_\mathbf{x}||_2^2 \qquad \mathbf{d}_\mathbf{x} = \arg\min_{\mathbf{d} \in \mathbf{Y}} ||\mathbf{d} - \mathbf{x}||_2^2 \tag{1.8}$$

where $\mathbf{d}_\mathbf{x} \in \mathbf{Y}$ is the closest point to $\mathbf{x} \in \mathbf{X}$ after transformation. Various algorithms exist to solve this problem, depending on the transformation model; rigid, non-rigid, or affine. Appendix B provides a review of these methods. In this work, we focus on two algorithms: the Iterative Closest Point (ICP) for rigid deformations, and the Coherent Point Drift (CPD) for both rigid and non-rigid deformations. ICP is a straightforward approach, while CPD offers flexibility, scalability, and smoothness [28]. It is of interest in a motorsport, context where computational time is limited and deformations are smooth.

In the literature, applying PSR to meshes has already been considered, but according to the author's knowledge only as a conceptual idea. In [29], PSR tools are presented as application on meshes. For a non-rigid deformation, a rigid alignment followed by a non-rigid correction is presented. The reason is that non-rigid registration is very difficult as the solution space $\mathscr{T}$ is much larger than for a rigid one. Similarly, in [30], a rigid iterative closest point (ICP) followed by a Thin Plate Spline (TPS) deformation is described. However, those tools are presented as a potential application in a meshing context and are not tested in a deforming object framework.

Iterative Closest Point (ICP) Method

This method is introduced by Besl et al. [31] and is initially suited for rigid deformation. Variants are also developed for non-rigid deformations [32]. Those are out of scope of this work and could be the object of further studies. The rigid version is an iterative algorithm that consists in three different steps, repeated until convergence of the MSE. The idea is to find a succession of transformation matrices, applied on the source point cloud $\mathbf{X}$ one after another until an optimum of Equation (1.7) is reached. For a more technical description, the reader is referred to [31, 33].

1. For each point $\mathbf{x} \in \mathbf{X}$, find its closest point $\mathbf{y} \in \mathbf{Y}$. It can be done in an efficient manner using a binary tree. This outputs list of point pairs $(\mathbf{x}_i, \mathbf{y}_j)$. Once done, simplifying a step is to compute the centroids of each sets, and then centralize everything around the origin.

2. Compute the motion that minimizes Equation (1.8). To do so, a covariance matrix can be constructed from the two centralized sets and a Singular Value Decomposition (SVD) is performed on it to obtain the optimal rotation matrix. For the translation matrix, the translation between the two centroids is computed, since the motion is assumed to be rigid.

3. Apply the motion (rotation matrix and translation matrix) to the $\mathbf{X}$ set, and compute the MSE. The iterative process continues, with the transformation matrix being refined at each iteration, until the change in the alignment reaches a minima (i.e the MSE converges).

A limitation is its outlier-free nature, as point pairs are computed using a nearest-neighbor approach [31], though this is not an issue for mesh vertices point clouds. Another drawback is scalability with increasing point cloud size, making it less suitable for large-scale industrial applications in motorsport [34]. In this work, the ICP approach is used as a benchmark to validate the Coherent Point Drift method.

Coherent Point Drift (CPD) Method

The Coherent Point Drift (CPD) algorithm is an iterative method framed as a Maximum Likelihood (ML) estimation problem. It maximizes a log-likelihood function to make the observed data most probable under the assumed statistical model [35]. It is a common approach in statistics and machine learning. Initially introduced by Myronenko et al. [35, 36], CPD can be adapted for different transformation sets $\mathscr{T}$ (rigid, affine, or non-rigid). The choice of $\mathscr{T}$ in this work is described in Chapter 2, alongside with further explanation on the corresponding assumptions.

It is first important to understand the notion of Gaussian Mixture Model (GMM). A Gaussian Mixture Model (GMM) assumes the observed data is produced from a combination of a finite

set of Gaussian distributions, each characterized by unknown parameters [27]. The CPD model considers $\mathbf{Y}$ as the GMM centroids and the points in $\mathbf{X}$ as the data points generated by the GMM. The goal is to move the GMM centroids as a group to preserve the topological structure of the point sets [36]. The GMM probability density function $P(\mathbf{x})$ of every reference point $\mathbf{x} \in \mathbf{X}$ is the sum of M Gaussian distributions $\mathscr{N}$ centered at the $\mathbf{y}_m$'s [36]

$$P(\mathbf{x}) = \sum_{m=1}^{M+1} P(m) \underbrace{P(\mathbf{x}|m)}_{\mathscr{N}(\mathbf{y}_m, \sigma^2\mathbf{I})} \tag{1.9}$$

where $P(m) = \frac{1}{M}$ (uniform distribution of the M centroids) and all σ^2's (variances) are the same by assumption [36]. Some noise weighting can be added to that equation, but this is not relevant in this work as a mesh does not have outliers. The centroid locations is set by a parameter θ that is updated over iterations. The algorithm stops when the Mean Squared Error (MSE) converges. From Equation (1.9), a negative log-likelihood function $E(.)$ is minimized, by summing the GMM of all the reference points $\mathbf{x}$. Under an i.i.d[2] assumption, a simple sum can be used [36]

$$E(\theta, \sigma^2) = -\sum_{n=1}^{N} \log \sum_{m=1}^{M+1} P(m)P(\mathbf{x}|m) \tag{1.10}$$

To find the optimal θ and σ^2, the Expectation Maximization (EM) algorithm is used [37]. This approach consists into two steps iteratively repeared; the "E-step" and the "M-step". For the E-step, the goal is to guess the current values of the parameters (“old” parameter values) and then use Bayes theorem to compute a posteriori probability distributions, denoted $P^{old}(m|\mathbf{x}_n)$ [35]. Then the “new” (next iteration) parameter values are found by minimizing the expectation of the negative log-likelihood function (M-step). The E-step and M-step are then iterated until convergence. Following [36], the M-step is equivalent to minimizing the following objective function

$$Q(\theta, \sigma^2) = \frac{1}{2\sigma^2} \sum_{n=1}^{N} \sum_{m=1}^{M} P^{old}(m|\mathbf{x}_n)||\mathbf{x}_n - T(\mathbf{y}_m, \theta)||^2 + \frac{Nd}{2} \log \sigma^2 \tag{1.11}$$

As mentionned, the choice of the set $\mathscr{T}$ can be chosen (rigid, affine or non-rigid deformation). P^{old} is the posterior probabilities of GMM components calculated using the previous parameter values. From Bayes theorem and from the formula of a GMM model, it is computed as [36] (E-step)

$$P^{old}(m|\mathbf{x}_n) \triangleq \frac{P(m)P^{old}(\mathbf{x}_n|m)}{P(\mathbf{x}_n)} = \frac{\exp^{-\frac{1}{2}||\frac{\mathbf{x}_n - \mathscr{T}(\mathbf{y}_m, \theta^{old})}{\sigma^{old}}||^2}}{\sum_{k=1}^{M} \exp^{-\frac{1}{2}||\frac{\mathbf{x}_n - \mathscr{T}(\mathbf{y}_k, \theta^{old})}{\sigma^{old}}||^2}} \tag{1.12}$$

The use of this method in a mesh deformation context is presented in Chapter 2.

[2] Independent and identically distributed

Chapter 2

Methodology

In this chapter, the tools from Chapter 1 are presented as tools to enable mesh deformation 1) during runtime and 2) as a preprocessing stage. As a first step, the limitations of the currently existing SLERP mesh deformation technique of OpenFOAM are outlined. An improved utility is then presented and provides a constraint on the scaling factor $s(\mathbf{x})$ close to solid boundaries.

From there, two utilities are built. The first one is a more robust solver for mesh deformation during runtime. The second one is to preprocess a mesh and deform it around a given deformed object. It is explained how Point Set Registration (PSR) methods are used to translate boundary mesh points adequately to a target object position. From there, it is explained how this information is used within a tuned OpenFOAM utility to deform the entire mesh.

2.1 Interpolating Solid Body Motion Solver

In the OpenFOAM-dev version, the SLERP mesh deformation utility (see Section 1.1.3) is implemented under the "Interpolating Solid Body Motion" solver. In its current version, the patch (or geometry) around which the mesh is deformed is an input, as well as the internal and external radii from which the scaling function $s(\mathbf{x})$ is computed (see Figure 1.1). This implementation suffers from two main limitations. The first limitation is that it is limited to rigid body motions, as the solver only allows for a single motion applied to every boundary mesh points. It means that it does not support arbitrary deforming surfaces.

The second limitation is that the code is not taking into account fixed boundaries, such as computational domain boundaries. In the case of a moving car simulation, we would like the car to be moving very closely to the ground. This is not possible with the current OpenFOAM SLERP

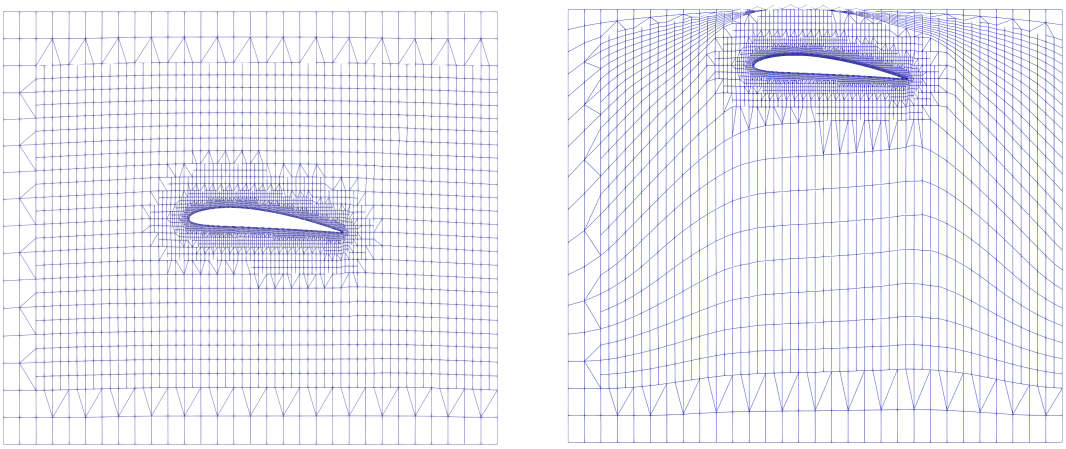

Figure 2.1: Limitations of the Interpolating Solid Body Motion utility on an oscillation airfoil. Left: initial mesh configuration. Right: close to boundary configuration.

utilty implementation. An illustration of this issue is provided on Figure 2.1 for the case of an oscillating airfoil close to a solid boundary. It is seen that that the inner radius deforms the solid boundary when intersecting it. This causes robustness issues near solid boundaries. What is desired in practice is to make the implementation aware of solid boundaries, so that the scaling factor $s(\mathbf{x})$ is computed accordingly.

2.2 Constrained Motion Solver

To fix the limitation shown on Figure 2.1, two elements are considered. Those improvements are implemented in an utility that is a variant of the "multiValveEngine" solver[1] from the same authors. They are implemented as part of the computation of the scaling function $s(\mathbf{x})$. The first improvement is about considering the presence of walls when computing the scaling $s(\mathbf{x})$, so that their deformation is avoided when the inner radius intersects them. To do so, a "frozen" region with non-zero thickness t_f is included, such that the scaling factor can be forced to $s(\mathbf{x}) = 0$ inside of it. If the inner radius and a frozen region are overlapping, the scaling $s(\mathbf{x})$ is set to zero. In that sense, this fix constrains the mesh deformation to stay within the computational domain. The frozen region is illustrated on Figure 2.2, on the wing test case presented on Figure 2.1. It is seen that the region with $s(\mathbf{x}) \neq 0$ (i.e. the deformed region) is bigger below the object than close to the boundary, because of the constraint on the scaling field $s(\mathbf{x})$.

The second improvement concerns the implementation of a sliding region within the domain, along which the mesh can slide. For instance, it is useful for the first test case of this work, which is an Ahmed body [38] oscillating close to the ground. Considering the struts as a sliding patch allows

[1] `https://github.com/OpenFOAM/OpenFOAM-dev/tree/e45bad9beb38b1aaceb36e94a12656ecd62cd062/src/fvMeshMovers/multiValveEngine`

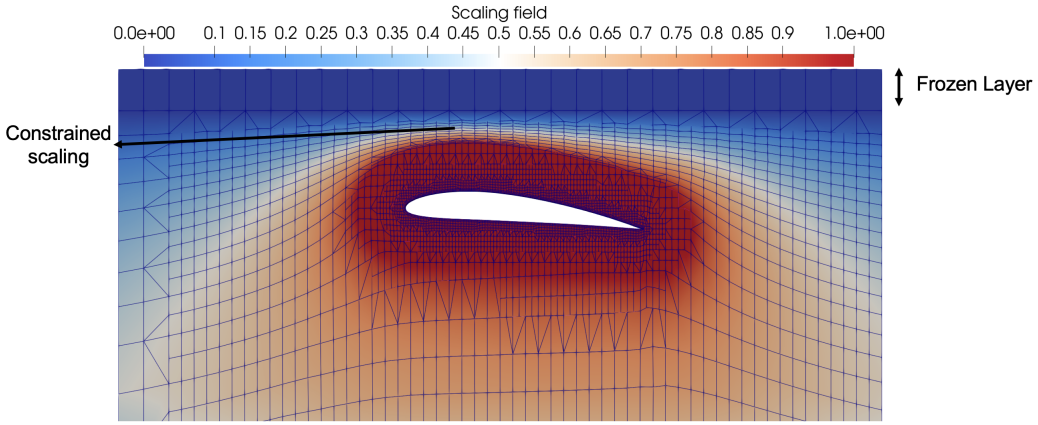

Figure 2.2: Constrained scaling field $s(\mathbf{x})$ on an oscillating airfoil close to a solid boundary (top).

for more accurate results, simply because the mesh points do not remain attached to them (it thus avoids a skewing of the mesh cells around the struts). In practice, one should imagine that the car is fixed to the ground and that there is an ellipsoidal whole in it to account for the contact patch of the car. Considering the contact patch and the lower boundary of the domain as sliding patches allows the car to slide on the ground as it moves. The improvements presented are a starting point to develop two mesh deformation utilities; one designed to work during runtime (Section 2.3) and the other to preprocess a mesh before runtime (Section 2.4).

2.3 Runtime Mesh Deformation

This section addresses runtime mesh deformation improvements with respect to the utility described in Section 2.2. As mentioned, a frozen region prevents ground deformation when deforming the mesh. However, a challenge still arises during runtime mesh deformation, when the inner radius r_{in} reaches the frozen region of thickness t_f. At the time step before the inner radius intersects the frozen region, the points are moved within the frozen region. Once they intersect, the points are not moved anymore ($(s(\mathbf{x}) = 0)$) because of the constraint from Section 2.2. This causes the mesh points to overlap in the frozen region and breaks the structure of the mesh in that region. If the object has to oscillate back and forth close to the boundary, it leads to robustness issues: for a same body position before and after touching the frozen region, the mesh structure is not the same because of the mesh point overlap in the frozen layer.

In the case of an Ahmed body [38] (more generally a car), let us define h_{ride} as the distance between the car's floor and the ground (or track). The configuration is provided on Figure 2.3. An issue arises as soon as $r_{in} = h_{ride} - t_f$, or equivalently where the inner radius and frozen regions intersect. To address this, the code can be modified to include a time-dependent adaptive scaling

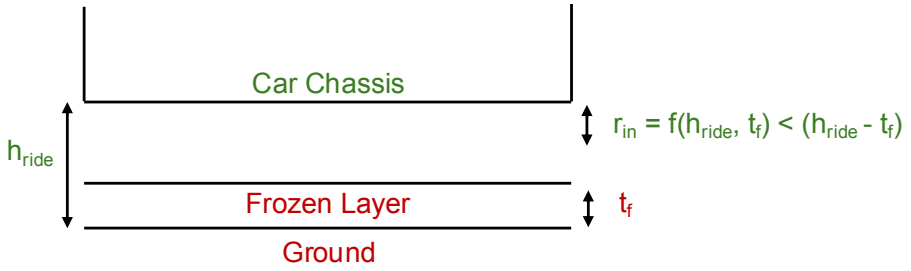

Figure 2.3: Configuration and nomenclature for a body approaching the ground. The red denotes fixed values over time. The green denotes updated values over time.

factor $s(\mathbf{x},t)$, which adjusts the inner radius r_{in} based on the car's ride height h_{ride} and frozen layer thickness t_f. The implementation involves recalculating $s(\mathbf{x},t)$ at each iteration. Specifically, h_{ride}, the smallest boundary mesh point coordinate relative to the ground, is used to update r_{in} according to $r_{in} = f(h_{ride}, t_f)$. Although there's no fixed formula, a sensitivity analysis suggests that $r_{in} = \frac{h_{ride} - t_f}{4}$ provides robust mesh deformation in the Ahmed body test case (presented in Section 3.1). This simplification is acceptable as a proof of concept. This approach maintains the boundary layer structure around the object, ensuring mesh topology is preserved by keeping the boundary layer thickness smaller than the inner radius. In the test cases, the outer radius, r_{out} will be fixed to a large value, so that $s(\mathbf{x})$ is only bound by the domain boundaries.

This method works well for purely vertical oscillations but has a limitation. Considering a car (or Ahmed body), if its chassis rotates, it might deform the wheels even though they are supposed to remain fixed to the ground. To counter this, $s(\mathbf{x},t)$ can be set to 1 around the wheels, ensuring they remain aligned with the car. This is a specific fix used as a proof of concept in this work.

2.4 Preprocessing Mesh Deformation

This section addresses preprocessing mesh deformation. The challenge is to deform a mesh before runtime between two slightly different configurations of a body, which avoids the need to remesh the entire domain and reduces manual user interaction. The goal is to find a mapping between an initial (source) and final (target) geometry, possibly involving non-rigid deformations, and then to move each mesh point accordingly. As mentioned in Section 1.2.1, the approach assumes no motion information between the two geometries.

The Point Set Registration (PSR) approach is chosen in this work. The reason is that PSR tools have public implementations and are mentioned in the literature to have potential [29] in a mesh context. The two Point Set Registration algorithms used are detailed in Chapter 1. PSR represents the source and target geometries as point clouds, potentially with difference sizes (see Equation

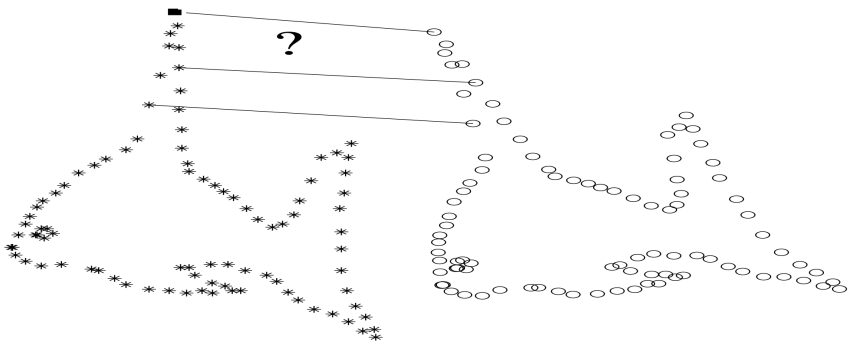

Figure 2.4: Point Set Registration (PSR) objective, from Myorenko et al. [36].

1.8). Figure 2.4 represents the fundamental concept behind PSR. It outlines the construction of a mapping between source and target point clouds. The way it is used in this framework is described hereafter.

2.4.1 Boundary Mesh Points Displacements

The goal is to deform a mesh around a body to a final position, by individually moving each mesh point. In this framework, the desired approach is to prescribe the translation direction for each boundary point by using an optimal mapping T^*. This can be achieved by solving a PSR problem.

In this proof of concept, the PSR step is implemented in Python and is decoupled from the OpenFOAM deformation utility. This approach allows full control over the process. The process begins by generating a mesh around the body's initial configuration. The boundary points are then extracted and supplied to Python, where the PSR finds the optimal mapping between the source and target geometries. Finally, this information is sent back to OpenFOAM and all mesh points, both boundary and internal (see Section 1.1), are adjusted to fit the target geometry.

Following [29], the use of PSR is subdivided into two steps. The first is a rigid deformation and the second is a non-rigid correction, which is a minor final deformation to fit the target point cloud. To do this, the Coherent Point Drift (CPD) approach is of interest, as the transformation set $\mathscr{T}$ (see Section 1.7 for details) can be tuned easily . The objective of the rigid deformation step is to deform the initial point cloud sufficiently close to the final one, so that the non-rigid deformation can correct the solution to fit the target geometry. The rigid deformation does not need extra parameters, whereas the non-rigid one depends on two hyper-parameters, described hereafter. This work uses the PyCPD Python package [39], which is a Numpy-based implementation of the CPD algorithm.

Point Clouds Selection

Defining suitable source and target point clouds is essential for the effective training of the PSR algorithm. The selection and size of these point clouds are particularly critical for non-rigid deformations, which involve significantly more variables than rigid deformations (as discussed later). In this study, CAD (Computer-Aided Design) vertices from both source and target geometries are used to train the PSR algorithm and to determine the optimal transformation T^*. This approach assumes that the CAD models are sufficiently detailed to produce an accurate transformation T^*, which is typically the case in industrial applications. The reason behind this choice is that, in the case of a limited number of boundary mesh points, training the PSR algorithm on CAD vertices can still yield a precise solution. After training, the optimal transformation T^* is applied to the boundary mesh points to determine their expected positions on the target geometry.

Deformation Step 1: Rigid/Affine Registration

For the case of a rigid deformation, the transformation of the centroid locations can be written as $T(\mathbf{y}_m) = s\mathbf{R}\mathbf{y}_m + \mathbf{T}$, s being a scaling factor, $\mathbf{R} \in \mathbb{R}^{d\times d}$ a rotation matrix and $\mathbf{T} \in \mathbb{R}^d$ a translation vector [36]. By using Equation (1.11) and by imposing properties to the matrix $\mathbf{R}$ so that it yields a rotation matrix, a constrained optimization problem is minimized to find s, $\mathbf{R}$, $\mathbf{T}$ and σ [36]

$$Q(s,\mathbf{R},\mathbf{T},\sigma^2) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}\sum_{m=1}^{M} P^{old}(m|\mathbf{x}_n)||\mathbf{x}_n - (s\mathbf{R}\mathbf{y}_m + \mathbf{T})||^2 + \frac{Nd}{2}\log\sigma^2 \tag{2.1}$$

$$\text{s.t} \qquad \mathbf{R}^T\mathbf{R} = \mathbf{I} \qquad \text{and} \qquad \det(\mathbf{R}) = 1 \tag{2.2}$$

where "det" is the determinant operator and $\mathbf{I}$ is the identity matrix. The solution can be found by applying standard methods of optimisation [36]. Once the four parameters are found, Equations (1.11) and (1.12) are recomputed, so that the "Expectation-Maximization" computation (see Section 1.2.3 and [37] for details) is iterated until convergence. As an output, the scaling, rotation and translation are found, as well as the update of the variance σ^2. On top of that, the probability of correspondence matrix $\mathbf{P}$ provides valuable information. For each point in the set $\mathbf{X}$, it provides with a probability that it belongs to every point in $\mathbf{Y}$.

Similarly to the rigid registration, an affine registration could also be of interest in some contexts. An example is a when a geometry is dilated uniformly because of thermal effects. Another example is when trying to map two manufactured components that are slightly different because of tolerancing. In this case, the transformation is written as $T(\mathbf{y}_m) = \mathbf{B}\mathbf{y}_m + \mathbf{T}$, where $\mathbf{B} \in \mathbb{R}^{d\times d}$ is an affine (linear) transformation matrix [36] and $\mathbf{T}$ is as above. This case is easier to solve since the

optimization problem is unconstrained. The reader is referred to [36] for additional details.

Deformation Step 2: Non-rigid Registration Correction

This section follows the work of Myorenko et al. [35], and summarizes the non-rigid transformation method and its principles. In this case, the transformation is defined as the initial position plus a displacement function v (also called velocity field [35]), so that the objective is to find $\theta \triangleq v$ and σ^2. The transformation set T between the source and the target point cloud is given by [36]

$$T(\mathbf{Y}, v) = \mathbf{Y} + v(\mathbf{Y}) \tag{2.3}$$

Now, a Tikhonov regularization framework is used [36, 40] to smooth that function v. It consists in adding a regularization term, so that the log-likelihood function $f(v, \sigma^2)$ is of the form [36]

$$f(v, \sigma^2) = E(v, \sigma^2) + \frac{\lambda}{2}\phi(v) \tag{2.4}$$

where λ is the first hyperparameter of the non-rigid CPD method. It allows tuning the smoothness of the regularization. Plugging that back in Equation (1.11), this turns out to be equivalent to minimize the following M-step (equivalent to minimize the log-likelihood function of Equation (1.10))

$$Q(v, \sigma^2) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}\sum_{m=1}^{M} P^{old}(m|\mathbf{x}_n)||\mathbf{x}_n - (\mathbf{y}_m + v(\mathbf{y}_m)))||^2 + \frac{Nd}{2}\log\sigma^2 + \frac{\lambda}{2}\phi(v) \tag{2.5}$$

whose solution for v is of the form (refer to Lemma 2 in [36] for the intermediate steps)

$$v(\mathbf{z}) = \sum_{m=1}^{M} \mathbf{w}_m G(\mathbf{z}, \mathbf{y}_m) \quad \text{with} \quad \mathbf{w}_m = \frac{1}{\lambda\sigma^2}\sum_{n=1}^{N} P^{old}(m|\mathbf{x}_n)(\mathbf{x}_n - (\mathbf{y}_m + v(\mathbf{y_m}))) \tag{2.6}$$

and has to be valid for all vectors $\mathbf{z}$. G is a Green's function (or kernel), which is also Gaussian distributed [36]. As a reminder, a Green function is the response of the system at point $\mathbf{x}$ due to a unit impulse (or delta function) applied at point. The reader is referred to [25] for more details. As a last step, $\mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_M)^T \in \mathbb{R}^{M\times d}$ can be found by solving the following linear system, obtained by evaluating Equation (2.6) at points $\mathbf{y}_m$ [36]

$$(\mathbf{G} + \lambda\sigma^2 d(\mathbf{P}1)^{-1})\mathbf{W} = d(\mathbf{P}1)^{-1}\mathbf{P}\mathbf{X} - \mathbf{Y} \tag{2.7}$$

Where $G(\mathbf{y}_i, \mathbf{y}_j) = e^{-\frac{1}{2}||\frac{\mathbf{y}_i - \mathbf{y}_j}{\beta}||^2}$, $d(.)$ is the inverse diagonal matrix and 1 is a column vector of ones. β is the second hyperparameter of the non-rigid CPD method, acting on the width of the Gaussian

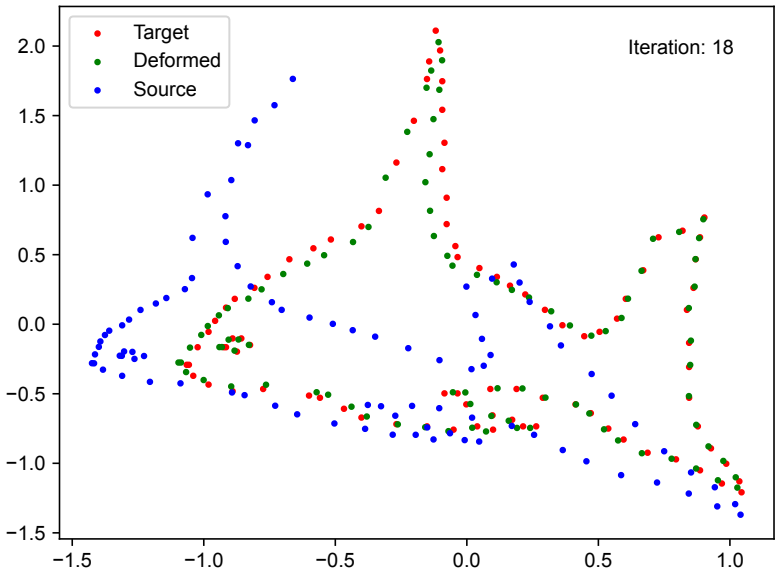

Figure 2.5: Non-rigid CPD example on a 2D fish dataset, after 18 iterations of the Expectation-Maximization algorithm, with $\lambda = \beta = 2$ [39].

kernel. Once $\mathbf{G}$ and $\mathbf{W}$ are found, the transformation is found according to Equation (2.3)

$$\mathbf{X} = T(\mathbf{Y}, v) = \mathbf{Y} + \underbrace{\mathbf{G}\mathbf{W}}_{=v(\mathbf{Y})} \tag{2.8}$$

As a last information, the update of the variance σ^2 is updated by equating the derivative of Equation (2.5) to zero [41]. For more details, the reader is referred to [35, 36]. A 2D example of the non-rigid deformation on a deforming fish is provided on Figure 2.5. On this Figure, a deformation from a given source point cloud (blue) to a target point cloud (red) is found by using the non-rigid CPD method. The example is publicly available in the PyCPD Python package [39].

The non-rigid CPD registration method depends on 2 main hyperparameters, λ and β. Concerning λ, it is a trade-off between making points align and the regularization of the deformation [39]. A higher value makes the deformation more rigid and a lower value makes the deformation more flexible [39]. Concerning the value of β, it is the width of the Gaussian kernel used to regularize the deformation [39]. It identifies how far apart points should be to move them together.

The biggest drawback of this method is that there is no direct way to guess the value of those two parameters [26]. Further information on the selection of those those hyperparameters is shown in [42]. An example of the effect of each parameter is provided in Appendix B. In this work, the value of those hyperparameters are directly given when the non-rigid deformation is used and are the object of iterated testings.

Deformation Step 3: Projection on Target Surface

As a last step, it is worth mentioning that the Point Set Registration methods could be insufficient to map perfectly a mesh on a target geometry. As shown on Figure 2.5, the deformation might get close to the target point cloud but not on top of it. However, when considering the method for 3D surface, not point clouds, we need to ensure that we respect the surface definition, i.e. points should lie on the surface.

To address this issue, the Point Set Registration is first performed to obtained a deformed point cloud for the boundary mesh points (rigid and non-rigid correction). It is then projected as a final deformation step onto the target surface. In this work, this surface is assumed to be under the form of a CAD meshed geometry. The assumption for this approach to work is that the point set registration brings the point cloud close enough to the final position. The projection algorithm involves a simple vector projection onto the triangulated surface of the target. The algorithm is fast, as a spatial data structure helps narrow down candidate triangles of the target geometry. The computational cost of the final projection step is assessed in Chapter 3.

2.4.2 Internal Mesh Points Displacements

In terms of practical implementation, once the mapping T^* is found and the final projection on the target surface is performed, the translation vector of each boundary mesh point is exported in a convenient format from Python to be read by OpenFOAM. In this work, it is chosen to export the data in a FOAM format.

After the boundary point displacements are read by OpenFOAM, the motion of the internal points must be determined. Calculating the displacement of each mesh point requires two components: a scaling factor $s(\mathbf{x})$ and a translation vector. The scaling function is predefined (see Section 1.1.3). To obtain the translation vector of an internal point $\mathbf{x}$, its nearest point on the boundary mesh, denoted as $\mathbf{x}_{\text{nearest}}$, is identified. The $\mathbf{x}_{\text{nearest}}$ point is found by iterating over the boundary mesh and by selecting the closest point to $\mathbf{x}$. The corresponding translation vector $\mathbf{p}_{\text{nearest}}$ of this boundary mesh point $\mathbf{x}_{\text{nearest}}$ is then used as the translation vector for $\mathbf{x}$.

It is worth mentioning that the implemented OpenFOAM preprocessing mesh utility can also handle geometries with multiple subparts. The user has the freedom to run the deformation algorithm on each of the subpart separately, which can provide more accurate results in the case the geometry to deform is too complex (like a car with tyres and suspensions moving differently for instance). To do so, a mapping methodology is implemented, enabling the deformation of multiple subparts in a continuous but efficient manner.

2.4.3 Final Algorithm

The methodology is summarized in Algorithm 1. The procedure is split into three steps; (1) the source mesh generation in OpenFOAM, (2) the Point Set Registration algorithm in Python and (3) the mesh deformation of the domain in OpenFOAM. Concerning the first OpenFOAM part (Alg. 1, line 4), it concerns the initial meshing around the source geometry, that will be deformed to match the target one. Concerning the Python part (Alg. 1, lines $7-15$), the first step is to extract the boundary points from the entire mesh (that contains boundary points and internal points). Finding the translation vectors is then done via a rigid registration, a non-rigid correction and a final projection on the target surface. Those translation vectors are supplied back in OpenFOAM and combined with $s(\mathbf{x})$ to deform locally the computational domain (Alg. 1, lines $18-29$).

Algorithm 1 Preprocessing Mesh Deformation Pseudo-Code

1: **Input:** A source geometry G_{source}, and a target geometry G_{target} around which the mesh has to be deformed during the preprocessing stage.
2:
3: **[OpenFOAM]**
4: Generate a meshing M around G_{source} if not given. Extract its information in a file.
5:
6: **[Python]**
7: Extract the boundary mesh points from M, denoted X.
8: **for** Every patch group **do**
9: Choose point clouds for G_{source} and G_{target} to train the Point Set Registration algorithm.
10: Rigid Point Set Registration.
11: Non-rigid Point Set Registration correction.
12: Get $X_{\text{deformed}} = T^*(X)$.
13: Project X_{deformed} on G_{target}. It gives X_{final}.
14: Output the translation vectors $P \triangleq X_{final} - X$.
15: **end for**
16:
17: **[OpenFOAM]**
18: Read the projection file containing the translation vectors P.
19: Determine the scaling field $s(\mathbf{x})$ over the domain, based on an inner radius r_{in} and an outer radius r_{out}.
20: **for** Every mesh point $\mathbf{x}_{\text{old}}$ of the computational domain Ω **do**
21: Find its corresponding scaling factor $s \in [0,1]$.
22: **if** $s = 0$ **then**
23: $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}}$
24: **else**
25: Find the nearest point of $\mathbf{x}_{\text{old}}$ on the boundary mesh X, denoted $\mathbf{x}_{\text{nearest}}$.
26: Extract its corresponding projection vector $\mathbf{p}_{\text{nearest}} \in P$ and use it as SLERP input.
27: Compute $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + s * \mathbf{p}_{\text{nearest}}$ and apply the SLERP algorithm.
28: **end if**
29: **end for**
30:
31: **Output:** The mesh around G_{target}.

Chapter 3

Results and Discussion

In this chapter, the methodologies from Chapter 2 are tested and analyzed. The results are split into two sections, each focusing on a different utility (runtime or preprocessing mesh deformation). In the first part, an Ahmed body [38] undergoes a solid motion and the mesh is deformed with it during runtime. A vertical translation is first examined, followed by a combination of translation and rotation (6 degrees of freedom motion), where the front of the Ahmed body points towards the ground. The mesh quality conservation is assessed for both test cases.

In the second part, preprocessing mesh deformation is evaluated between two states of a deforming object, as explained in Section 2.4. As mentioned in Algorithm 1, registration algorithms predict the movement of boundary mesh points (using the PyCPD Python package [39]), followed by the deformation of the entire computational mesh domain using OpenFOAM. The efficiency and accuracy of these methods are tested on a deforming quadcopter. The preprocessing deformation is first applied to a non-rigid extruded airfoil case. Then it is applied to a car-relevant model under cornering conditions, with multiple subparts moving differently.

3.1 Runtime Mesh Deformation

To begin with, a rigid body motion is applied to evaluate and validate the runtime improvements discussed in Section 2.3. The simplest test case for assessing an implementation in an automotive framework is the Ahmed Body benchmark [38]. In this work, the GitHub repository of Rooy et al. [43] is used, specifically the Ahmed body geometry with a decklid angle of $\phi = 25^\circ$, as shown on Figure 3.1. The upper body is set to be moving while the struts (i.e. the four components connecting the upper body to the ground) are forced to be fixed to the ground.

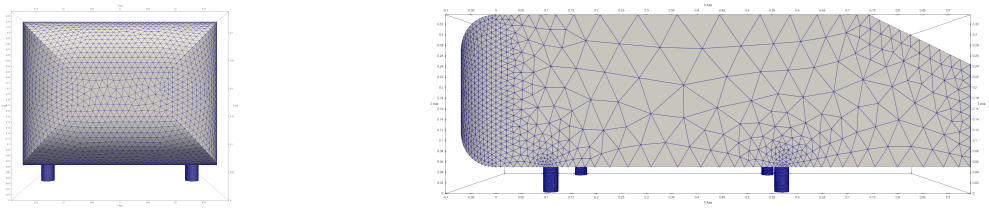

Figure 3.1: Ahmed body configuration [43], with a decklid angle $\phi = 25°$. The meshing of the original CAD geometry is shown on top. Left: front view. Right: side view.

3.1.1 Vertical Oscillating Motion

The first motion applied to the Ahmed body is a sinusoidal vertical motion. It moves the body up and down close to the ground (i.e $z = 0$). The struts patches (i.e struts subregions) are set as a sliding patch, as explained in Section 2.2. The amplitude of the oscillation is set to a $A = 4\%$ of the z-domain height, with a high translational frequency ($\omega = 4$ rad/s). With $[x_0, y_0, z_0]$ as the initial position of the body's center of gravity, the Ahmed body's rigid motion is then defined as

$$x(t) = x_0 \qquad y(t) = y_0 \qquad z(t) = z_0 + A\sin(\omega t) \tag{3.1}$$

As described in Section 2.3, the scaling field $s(\mathbf{x},t)$ is adapted at each iteration. The inner radius for mesh motion is adaptively adjusted based on the ride height and the frozen layer thickness ($r_{in} = \frac{h_{ride} - t_f}{4}$), while the outer radius is fixed to an arbitrarily large value. The frozen layer thickness (see Section 2.2) is set to approximately 1% of the z-domain size. The frozen region is then $z_f \in [0, t_f]$. For the solver to work effectively, t_f must be greater than 0, ensuring the frozen region has a finite thickness and ensuring the boundary layer remains fixed for accurate simulation results. The result is shown on Figure 3.2 on a slice capturing the struts. At $t = 0.5$s, the body is far from the ground and at $t = 1.2$s, it is close to it. The scaling field $s(\mathbf{x},t)$ is shown on top. It is chosen to be a

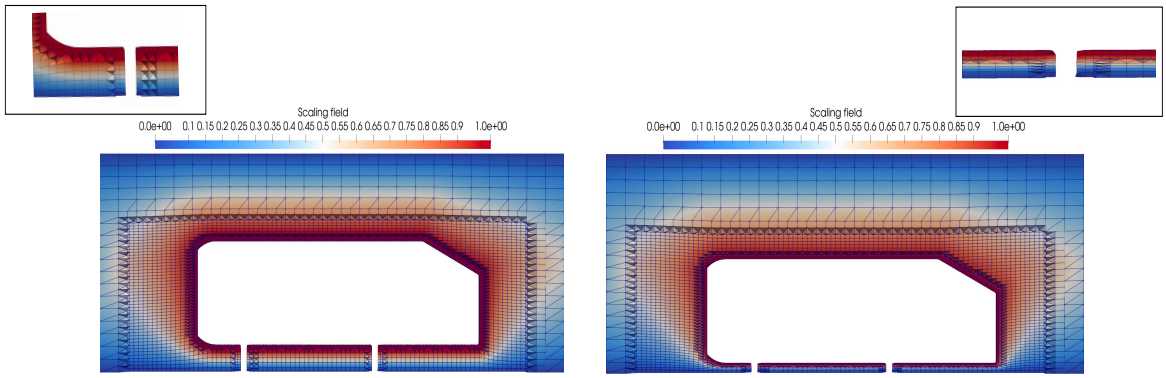

Figure 3.2: Side slice of the oscillating Ahmed body along the z-direction. The color map represents $s(\mathbf{x},t) \in [0,1]$. Left: $t = 0.5$s, far from the ground configuration. Right: $t = 1.2$s, close to the ground configuration.

cosine smoothing field (illustrated on Figure 1.2). As expected, $s(\mathbf{x},t) = 1$ close to the body and $s(\mathbf{x},t) = 0$ far from it. It can be seen that $s(\mathbf{x},t)$ is different for both cases, outlining the adaptive deforming region. When the body is far from the ground, the deformed region is much bigger than when the body is close to the ground. For pure translations, it validates the implementation described in Section 2.3.

Regarding the mesh, the structure remains stable under translational oscillations, and the ground stays undeformed as expected. As the body approaches the ground, the deformed region shrinks locally, causing mesh faces to compress without overlapping, demonstrating the method's robustness. The mesh quality is also inspected over time, for both the Constrained Solver (Section 2.2) and the improved solver with an adaptive scaling factor $s(\mathbf{x},t)$ (Section 2.3). While the Constrained Solver leads quickly to zero-volume cells (there is an overlapping of the mesh points, see Section 2.3), the adaptive solver maintains consistent mesh values, preserving the mesh skewness ($\simeq 1.6$) and keeping the average non-orthogonality stable ($\simeq 9.5^\circ \pm 0.3$). The maximal aspect ratio is also preserved ($\simeq 5$). To confirm the compatibility of this approach with OpenFOAM native flow solvers, this test case is run on the incompressible Navier-Stokes Equations, using a RANS simulation. The results are presented in Appendix C. The lift and drag coefficients are observed to oscillate with a fixed period, highlighting the inherent influence of an oscillating blunt body in cross flow.

3.1.2 Six Degrees of Freedom Motion

The next test case focuses on a six-degree of freedom type motion to the body. To do so, Equation (3.1) is used for the translation, alongside with a constant rotational frequency of $r_y = 3$ rad/s in the y-direction (pointing out the plane of Figure 3.2). The objective is to point down the nose of the Ahmed body. The motion is prescribed by tabulated time dependent translation and rotation values. The result is depicted on Figure 3.3 for two different times; $t = 2$s and $t = 5$s.

It is seen that the correction on the struts of the body (see Section 2.3) is effective, so that the struts slide along the lower boundary of the domain, while staying perpendicular to the upper body of the car. It is also seen that the mesh is rotated properly around the struts as the body rotates. Secondly, the deformed region is smaller under the body when the nose points further down, which helps preserving the structure of the deformed mesh cells. Furthermore, it is observed that mesh quality is retained throughout the simulation. The maximum skewness is 1.6 at $t = 0$, with a slight increase over time. The reason is the scaling function $s(\mathbf{x},t)$ is set to 1 around the struts but drops sharply to zero, causing some skewed mesh cells in these areas. This case-specific adjustment effectively

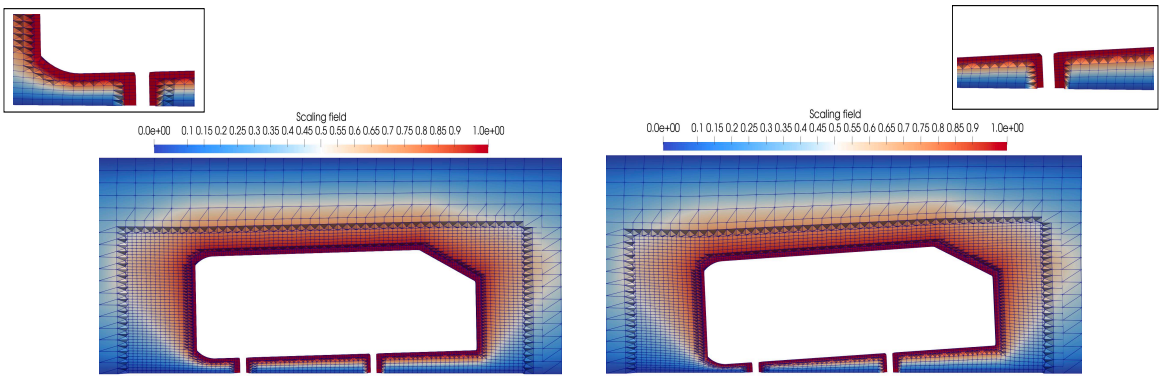

Figure 3.3: Side slice of the 6 degrees of freedom Ahmed body motion. The color map represents $s(\mathbf{x},t) \in [0,1]$. Left: $t = 2$s, small rotation configuration. Right: $t = 5$s, nose close to the ground configuration.

captures the rotational behavior of the upper body, which remains unaffected by skewness. The mesh non-orthogonality is also non-degenerating throughout the motion.

Finally, from a computational time perspective, it is important tout outline that remeshing the entire domain at each time step is not viable. For example, in this specific test case, the mesh consists of approximately 3×10^5 points, and the meshing process alone requires approximately 40 minutes on a single processor. For a simulation lasting 5 seconds with a time step of $\Delta t = 0.1$ seconds, remeshing at each iteration would require approximately 30 hours. In contrast, the mesh deformation is much faster, as it simply iterates over the domain and moves the mesh points explicitly. In this case, each time step with mesh deformation takes approximately 10 seconds on one core, i.e 8 minutes in total. This comparison underscores the critical importance of mesh deformation in simulations involving moving objects.

3.2 Preprocessing Mesh Deformation

To deform an volumetric mesh around an initial geometry to fit a target geometry, the translation vector of each boundary mesh point of the source geometry must be found. This step is done through Point Set Registration (Section 2.4) and does not require knowledge on the equations of motions between the two geometries. This approach is particularly useful when dealing with complex deformations, such as minor geometric changes in a new design or the arbitrary deformation of an F1 tire in high-speed corners, where determining motion equations is challenging.

In Section 3.2.1, the point set registration algorithm is validated, both in terms of accuracy and in terms of computational cost. Then, in Section 3.2.2, the utility is assessed on two test cases. The first one is a non-rigid wing deformation. The second one is a car-relevant geometry with a rotation of the tyres and a movement of the suspensions.

3.2.1 Registration Tools Analysis

This section aims 1) to validate the Python implementation of the two Point Set Registration algorithms (the ICP and the CPD, see Section 1.2.3) and 2) to assess the impact of point cloud size on its execution time. The computational cost is a critical factor, particularly in the context of complex industrial geometries with limited ressources.

Validation and Accuracy

To validate the implementation of Point Set Registration methods, the most straightforward approach is to ensure that the projected points converge to their expected positions on the target point cloud. The deformation tool is validated, on both the Iterative Closest Point (ICP) approach and the Coherent Point Drift (CPD) approach. The test involves a combination of rotation, translation, and scaling transformations applied to the object. Let us consider an arbitrary rotation of angle θ around the z-axis, combined with a translation vector $\mathbf{T} = [t_x, t_y, t_z]$ and a scaling vector $\mathbf{S} = [s_x, s_y, s_z]$. Letting $\mathbf{X}$ as the source point set, the relative error E_{rel} between the expected deformed point set $\mathbf{Y}_{True}$ and the obtained deformed point set $\mathbf{Y}$ can be evaluated as

$$\mathbf{Y}_{True} = \begin{bmatrix} s_x\cos\theta & -s_y\sin\theta & 0 & t_x \\ s_x\sin\theta & s_y\cos\theta & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \quad \Longrightarrow \quad E_{rel}\,[\%] = \max_{\mathbf{y}_i \in \mathbf{Y},\, \mathbf{y}_{i_{True}} \in \mathbf{Y}_{True}} \frac{|\mathbf{y}_i - \mathbf{y}_{i_{True}}|}{\mathbf{y}_{i_{True}}} \times 100 \tag{3.2}$$

The relative error, E_{rel}, is calculated as the maximum error across all points, focusing on the worst-case scenario. This study tests on a quadcopter geometry[1], with a stopping criterion for both algorithms based on the relative change in Mean Squared Error (MSE) between iterations. As done in [39], the stopping threshold is dimension-dependant and set to 10^{-3}. This value is observed to provide converged enough results for this test case. The accuracy of the two registration methods discussed (ICP and CPD) is illustrated in Figure 3.5. Both methods converge with a linear/$\mathcal{O}(n)$ order, where n represents the number of mesh points in the source/target point clouds used to train the PSR algorithm. This result aligns with Pottmann et al. [44] for the Iterative Closest Point (ICP) method. It is demonstrated that registration errors decrease linearly, particularly when the underlying geometries are smooth. Additionally, Cicconi et al. [45] confirmed that both ICP and CPD algorithms achieve a linear convergence rate in terms of accuracy when the point clouds are sufficiently dense and well-distributed. It underscores the importance of using an adequate number

[1] `https://cults3d.com/en/3d-model/game/quadcopter`

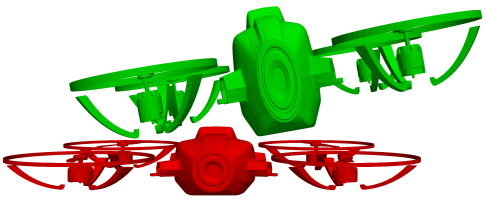

Figure 3.4: Tested drone geometry. Red: source geometry. Green: target geometry.

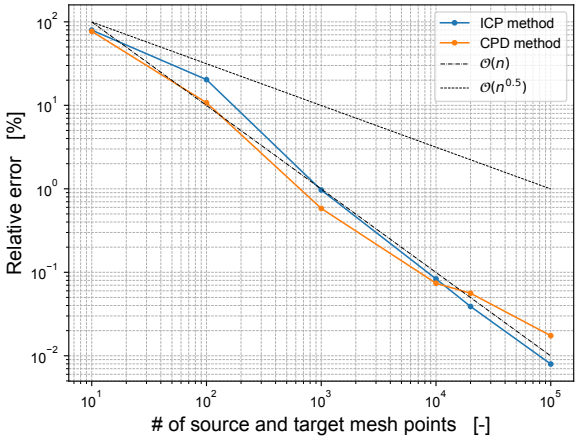

Figure 3.5: Relative error E_{rel} in % as a function of the source and target point cloud sizes.

of points in the source and target clouds to effectively train the algorithm.

Computational Cost

The next step is to analyze the computational cost of Point Set Registration. From now on, only the Coherent Point Drift (CPD) approach is investigated. The first reason is that the ICP and the CPD methods are similar in terms of accuracy convergence rate. The second one is that the CPD is shown to be easily scalable for large point cloud sizes, whereas the ICP is limited [28]. The third reason is that the CPD is working for non-rigid deformations. Those three reasons make the CPD more suitable in an industrial application.

In the methodology presented in this work (described in Algorithm 1), two elements influence the computational time; the cost of the registration algorithms (steps 1 and 2) and the cost of the final projection step (step 3) on the target geometry. Firstly, the computational cost is analyzed on the CPD algorithm for all three transformation choice sets $\mathcal{T}$ presented in Section 2.4.1. The source and target point cloud size influence on the computational time are investigated.

The results are provided on Figure 3.6. The left plot considers a fixed target point cloud size and a varying source one. The right one considers the opposite. Firstly, it is seen that the point cloud affecting the most the computational time is the target point cloud. This aligns with Myorenko [35], stating that the complexity is only depending on the target point size. The reason is that CPD treats the points in the target point cloud as centroids of a Gaussian Mixture Model (GMM). The algorithm involves computing the likelihood of each source point being generated by each Gaussian in the mixture, which scales with the number of points in the target cloud. It can also be observed that the computational complexity of the non-rigid deformation is the highest. The reason

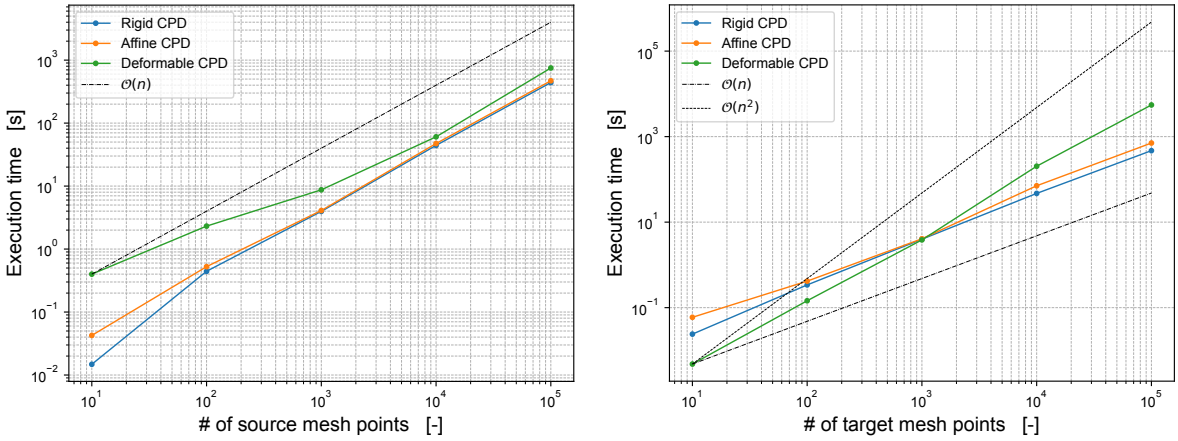

Figure 3.6: Execution times of both methods, on a single core execution. The left corresponds to a varying source point cloud size, while the right corresponding to a varying target one.

is that the non-rigid deformation implies to solve an additional linear system (Equation (2.7)), which is an $\mathcal{O}(M^3)$ operation (M being the number of unknowns). As mentioned by Myorenko [35], it is possible to reduce the computational complexity to be linear, up to a multiplicative constant. This is what is observed on Figure 3.6, as this is what is implemented in the PyCPD Python package [39]. To do so, a low rank matrix approximation is used for Equation (2.7). The reader is referred to [36] for additional information on this technique.

The second part of the analysis focuses on the efficiency of the final projection step discussed in Section 2.4.1. As explained in Section 2.4.1, this step involves projecting the deformed point cloud onto the target CAD surface. Figure 3.7 shows the execution time for the projection of one point on a varying target CAD geometry size. It exhibits linear behavior relative to the CAD geometry size. Given N boundary mesh points (source point cloud size) and M_{CAD} vertices in the target CAD geometry, the computational complexity is thus $\mathcal{O}(N \cdot M_{\text{CAD}})$ on a single processor.

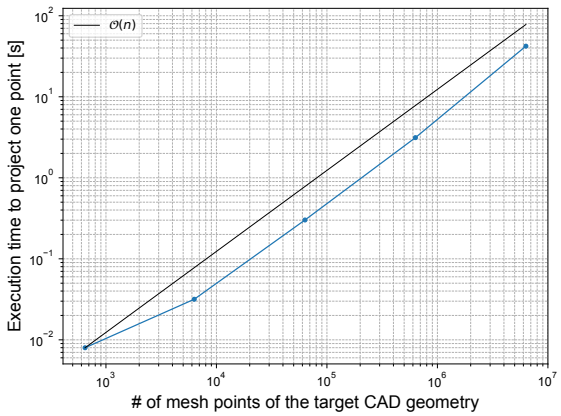

Figure 3.7: Efficiency analysis of the projection of one point on a varying target surface size.

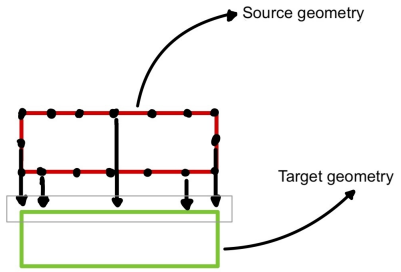

Figure 3.8: Limitations of the projection step: all the mesh points are projected on the upper surface.

As a last remark, performing the projection without prior registration is not viable. For example, in the case of a translated beam (Figure 3.8), a single projection would incorrectly flatten faces. It highlights that the projection step has to be used as a final correction step only.

3.2.2 Mesh Deformation Results for Arbitrary Non-rigid Geometry Changes

Two cases are tested in this work, the first one being wing, while the second one being on a car-like geometry. The idea of the second test case is to consider a situation where a designer changes a geometry only slightly and the objective is to avoid remeshing of the whole domain.

Extruded Airfoil Deformation

To validate the methodology in Section 2.4, a non-rigid deformation analysis is conducted on a wing. A wing motion tutorial from OpenFOAM-dev [2] is used as a starting point. The target geometry, representing a typical industry design variation, is created by thickening the profile shape. Figure 3.9 shows the source and target geometry, highlighting the differences in terms of profile shape.

A non-rigid deformation alone is observed to be insufficient for accurately deforming the airfoil. As discussed in [29] and Chapter 2, the approach involves first applying a rigid deformation, followed by a non-rigid correction, and finally projecting the points onto the target surface. The parameters $\beta = 10$ and $\lambda = 2$ are selected heuristically, based on iterative testing of the non-rigid deformation. The inner radius choice is rather arbitrary and set to $r_{in} = 15\%$ of the chord length. The outer radius r_{out} is chosen arbitrarily large, so that $s(\mathbf{x})$ is bound by domain boundary only. Figure 3.10 shows a deformed mesh slice and the local scaling factor $s(\mathbf{x})$, demonstrating that the deformation accurately fits the target airfoil geometry. Figure 3.11 further highlights the preservation of the boundary layer structure at the leading and trailing edges, outlining the method's potential for industrial applications in an accuracy point of view.

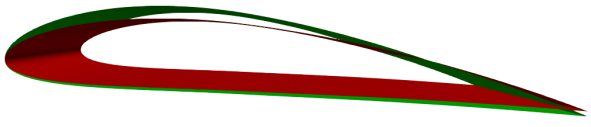

Figure 3.9: Geometries used for the non-rigid wing mesh deformation test case. Red: source geometry. Green: target geometry.

[2] `https://github.com/OpenFOAM/OpenFOAM-dev/tree/17280e978cf3fc8fd0f0bfd1e0eb4a0ed8fd1763/tutorials/incompressibleFluid/wingMotion`

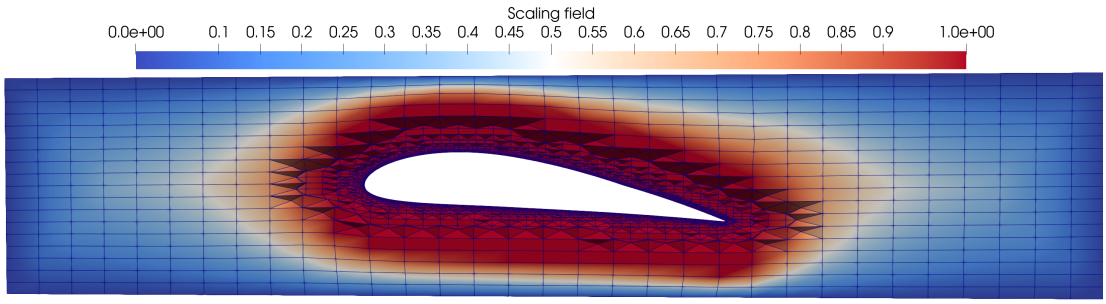

Figure 3.10: Deformed mesh around the deformed target file. The field represents the scaling factor $s(\mathbf{x})$ of the SLERP mesh deformation method.

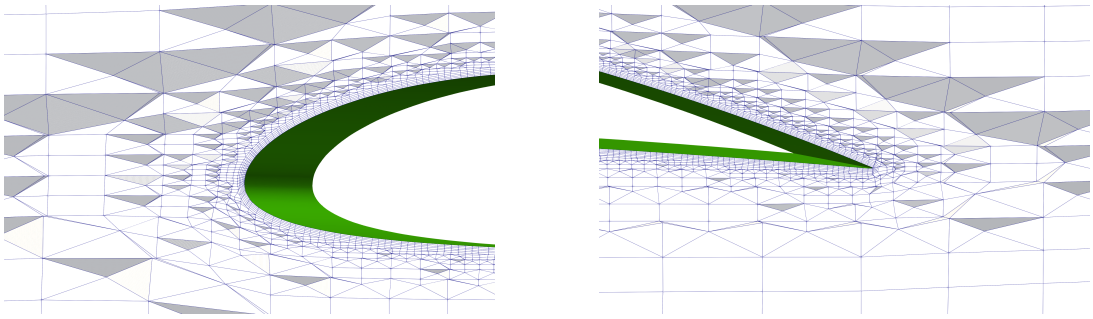

Figure 3.11: Deformed mesh around the target file (green). Left: zoom on the leading edge. Right: zoom on the trailing edge.

Lastly, concerning the computational time, the meshing contains $\simeq 10^5$ mesh points to deform. It is observed that the most expensive steps are the initial meshing and the the Python code that predicts the boundary mesh points displacement. The OpenFOAM step that deforms the mesh scales with with the number of mesh points and very fast. The meshing of this geometry takes $\simeq$ 15min (serial). The training of the mapping between the two geometries on the other side (rigid PSR, non-rigid PSR correction and final projection) takes $\simeq$ 8 minutes (serial). The mesh preprocessing Algorithm is thus $\simeq$ 46% faster (meshing vs mesh deformation), which outlines the potential of this method in industries where the computational ressources are limited.

Car Cornering with a Given Steering Angle

This section investigates the geometry of an F1-related car, focusing on a tire in a turn and suspension movements with a fixed main body. The deformation algorithm is applied individually to each patch, with the resulting translation vectors for each boundary mesh point inputted into the OpenFOAM preprocessing utility. As highlighted in [29], splitting the deformation algorithm enhances the accuracy of the result and reduces computational costs by minimizing the size of the source and target point clouds. Figure 3.12 illustrates the initial (red) and target (green) geometries.

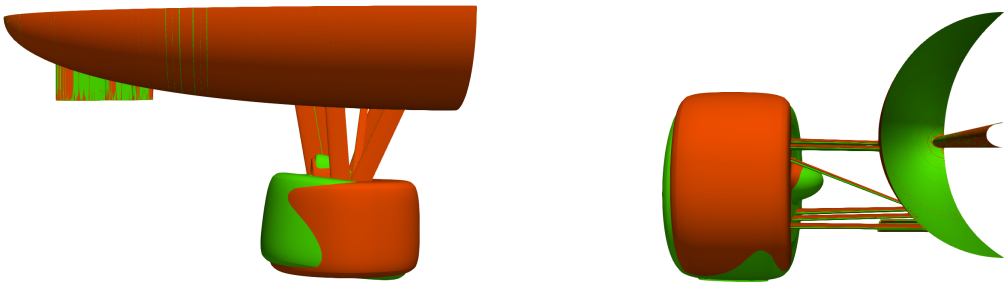

Figure 3.12: Computational files for of the car-like body in a turn. Red: Initial position. Green: target position.

This domain is composed of multiple patches (14 in total; suspensions, tyre, contact patch, main body etc.) and the deformation step is done separately on each of the patches. The initial meshing around the source geometry is generated following standard practices in CFD mesh generation.

The first result concerns the Python processing step explained in Algorithm 1, where the boundary mesh point displacements are predicted on the target geometry. The registration algorithm used involves the registration, followed by a projection onto the target. The result of the step is illustrated on Figure 3.13. On the left, the initial boundary mesh around the source geometry is outlined. On the right, the predicted boundary mesh (found by using Point Set Registration and a projection, see Algorithm 1) around the target geometry is shown. It is observed that methodology developed provides the same mesh topology and that it fits the target geometry.

However, it can be observed that an inaccuracy still persists at certain junctions between patches, as highlighted on Figure 3.14. Ideally, connected components should move cohesively, but the independent application of the registration step to each geometry followed by the final projection onto the target surface leads to motion discrepancies between patches. This issue is particularly pronounced when faces span multiple patches, causing artificial distortions at these junctions, especially between the suspension and the tire. It is an inherent sign of an implementation issue in

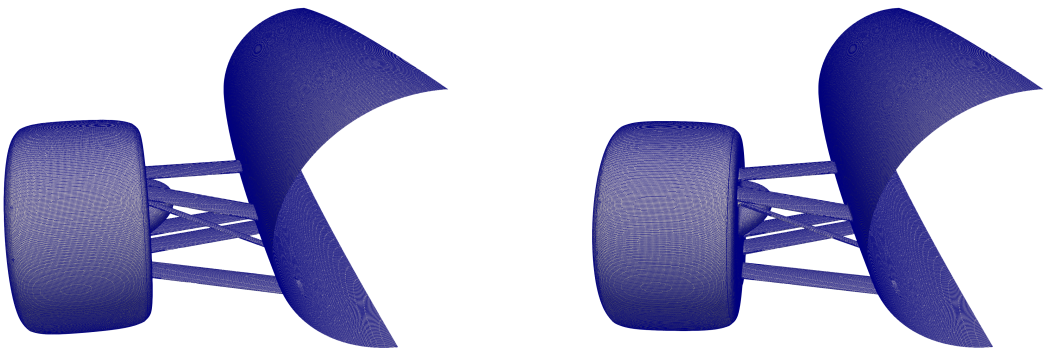

Figure 3.13: Left: Initial boundary mesh around the source geometry. Right: Predicted mesh around the target geometry, using the methodology presented in Section 2.4.1.

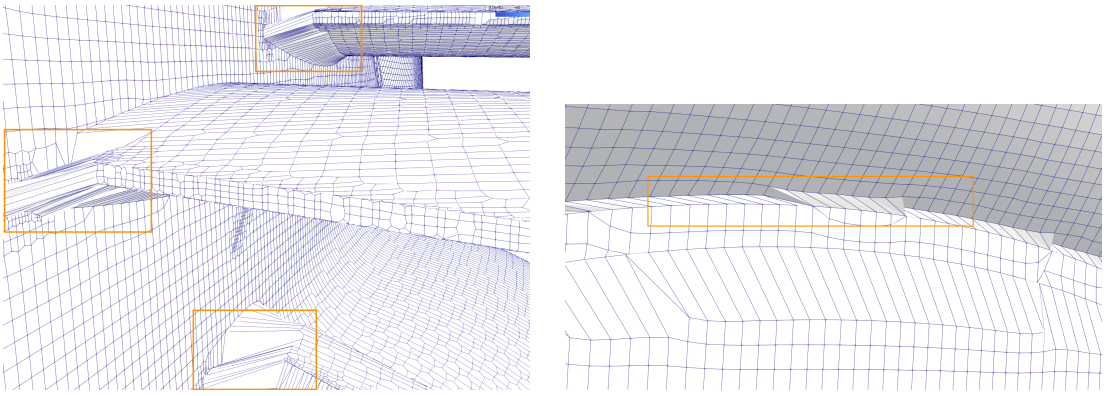

Figure 3.14: Remaining issues when dealing with a multiple patch deformation. Left: problem at the junction between patches. Right: skewing of cells shared by two different patches.

case of multiple patches as there should be no clear reason not to have a smooth transition therein. Executing the methodology with a single patch would avoid this problem, but it would also be less accurate as one global mapping would be found instead of one for each separate subpart.

The second result involves the deformed computational domain, based on predicted boundary mesh displacements (see Algorithm 1). Figure 3.15 shows the deformed mesh with a slice parallel to the ground on the wheel. The deformed mesh accurately fits the geometry, is not skewed, and shows displacement tapering to zero away from the tire. This validates the methodology in Section 2.4, with the boundary layer structure remaining intact and the mesh quality being preserved. Lastly, a comparison between full-domain remeshing and using the preprocessing mesh utility is essential. The mesh has a total of 1.3million mesh points. Remeshing the entire domain yields a better grid on the deform geometry, but the key question is whether the preprocessing mesh utility is faster. The initial meshing takes about 45 minutes (serial), whereas using the mesh deformation methodology from Algorithm 1 reduces it to around 25 minutes (serial). For this test case, a mesh deformation is thus $\simeq 45\%$ faster than an entire remeshing.

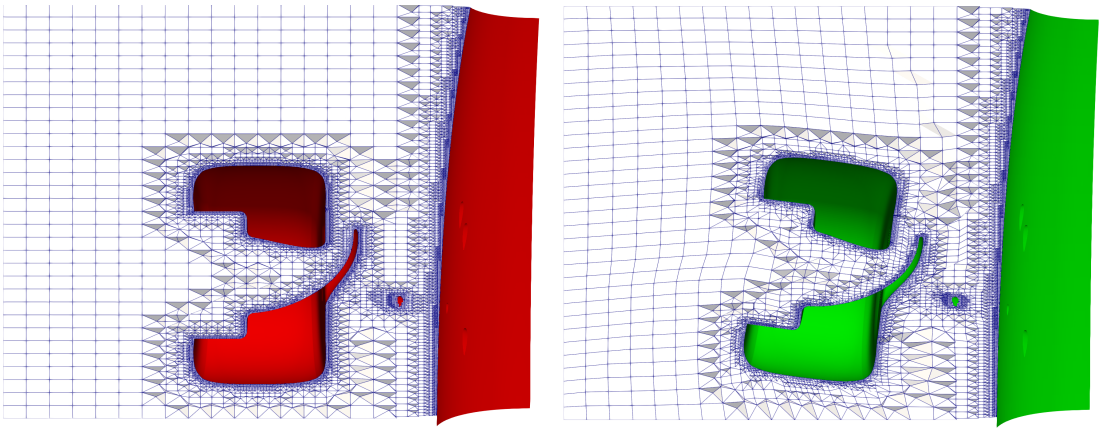

Figure 3.15: Slice across the tyre, on the initial mesh (left), and on the deformed mesh (right).

Conclusion

This research investigates mesh deformation techniques with a particular focus on their application in motorsport. The study is driven by two primary objectives: 1) to deform a mesh around a moving object robustly and 2) to reduce the computational costs associated with the meshing process. The first part of this work explores runtime mesh deformation, while the second part examines mesh deformation within as part of the preprocessing stage. The results satisfy the aim and objectives of this project, alongside with additional results and explanation to reinforce them.

Concerning runtime mesh deformation, mesh quality is preserved over time for the test cases considered. The importance of mesh deformation is outlined in a moving object framework, as it is much faster than remeshing the domain at each time step. The technique used allows for a locally tunable deformed region, while preserving the structure of the mesh. Lastly, the method developed allows a body to move close to the ground without deforming it. It is shown to be working accurately for translational and rotational body motions.

Concerning mesh deformation as part of the preprocessing stage, Point Set Registration (PSR) methods are shown to be a candidate to accelerate the meshing stage, as it preserve the mesh structure when mapping a boundary mesh from one configuration to another. PSR allows to deform a body rigidly or non-rigidly. This utility is demonstrated on a simple geometry and on a motorsport-relevant geometry. Firstly, the methodology developed is shown to work. Secondly, it is shown to be accurate. Lastly, this method is of interest in a motorsport context, as it is faster than remeshing utilities and do not require knowledge of the equations of motion between two configurations of a body. Concerning the computational efficiency, it is observed to provide up to a 45% gain of time. Through a computational cost analysis, the prediction of the boundary mesh points displacement is observed to be the most expensive step of the methodology.

Areas of Improvement

As an extension of this work, some improvements remain to be considered as a further step toward more efficient and accurate mesh deformation tools.

1. Code Parallelization. Adapting the two developed OpenFOAM utilities to be running on multiple processes is relevant as it would accelerate the mesh deformation process.

2. Flow Simulations. Running the Navier-Stokes equations on a moving geometry using the runtime mesh deformation utility developed is a challenge to be tackled.

3. Mesh Quality Improvements. The developed preprocessing mesh deformation utility faces challenges at the junctions of multiple patches. Addressing these issues is important for improving the accuracy and reliability of the mesh deformation process on complex geometries.

4. Further Computational Time Analyses. The computational time of the preprocessing mesh deformation utility needs to be further compared to the meshing time, especially for larger mesh sizes similar to those used in motorsport applications.

5. Non-Rigid Point Set Registration. The non-rigid Coherent Point Drift method is complex and requires the determination of two hyperparameters that are hard to identify, especially for complex geometries. Exploring alternative PSR algorithms could provide valuable insights for handling deformations.

6. Alternatives to Point Set Registration. If known, using the equations of motion between the source and target geometries has the potential to further decrease the computational time of the preprocessing mesh deformation utility. Alternatively, machine learning techniques, such as autoencoders, could be a viable alternative to Point Set Registration.

Bibliography

[1] C.J. GreenShields and H.J. Weller. *Notes on Computational Fluid Dynamics: General Principles*. Reading, UK: CFD Direct Ltd, 2022.

[2] M. Gadola et al. "Analyzing Porpoising on High Downforce Race Cars: Causes and Possible Setup Adjustments to Avoid It". In: *Energies* 15.18 (2022), p. 6677.

[3] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*. Vol. 113. Fluid Mechanics and Its Applications. Springer, 2016.

[4] F.M. Bos, B.W. van Oudheusden, and H. Bijl. "Radial basis function based mesh deformation applied to simulation of flow around flapping wings". In: *Computers and Fluids* 79 (2013), pp. 167–177.

[5] J.I. Leffell, S.M. Murman, and T.H. Pulliam. "Time-Spectral Rotorcraft Simulations on Overset Grids". In: *American Institute of Aeronautics and Astronautics (AIAA)* (2014).

[6] F. Duarte, R. Gormaz, and S. Natesan. "Arbitrary Lagrangian–Eulerian method for Navier–Stokes equations with moving boundaries". In: *Computer Methods in Applied Mechanics and Engineering* 193.45 (2004), pp. 4819–4836.

[7] C.S. Peskin. "The immersed boundary method". In: *Acta Numerica* 11 (2002), pp. 479–517.

[8] S. Brahmachary et al. "A Sharp-Interface Immersed Boundary Method for High-Speed Compressible Flows". In: *Immersed Boundary Method : Development and Applications*. Springer Singapore, 2020, pp. 251–275.

[9] M. Sun and Y. Xin. "Flows around two airfoils performing fling and subsequent translation and translation and subsequent clap". In: *Acta Mechanica Sinica* 19 (2003), pp. 103–117.

[10] S. Völkner, J. Brunswig, and T. Rung. "Analysis of non-conservative interpolation techniques in overset grid finite-volume methods". In: *Computers & Fluids* 148 (2017), pp. 39–55.

[11] R. Fedkiw. *Lecture 7: Mathematical Methods for Fluids, Solids and Interfaces (Stanford University, CS205b/CME306)*. 2024.

[12] *Mesh Motion and Topological Changes*. OpenFOAM. 2014. URL: `https://openfoam.org/release/2-3-0/mesh-motion/`.

[13] H. Jasak and Z. Tukovic. "Automatic mesh motion for the unstructured Finite Volume Method". In: *Transactions of FAMENA* 30 (2006), pp. 1–20.

[14] F. Bos, B.W. Van Oudheusden, and H. Bijl. "Moving and deforming meshes for flapping flight at low Reynolds numbers". In: *Delft University of Technology* 2 (2008), p. 19.

[15] R.P. Dwight. “Robust Mesh Deformation using the Linear Elasticity Equations”. In: *Computational Fluid Dynamics*. Ed. by H. Deconinck and E. Dick. Springer, 2006, pp. 401–406.

[16] M. Biancolini et al. “Radial Basis Functions Mesh Morphing: A Comparison Between the Biharmonic Spline and the Wendland C2 Radial Function”. In: *Computational Science* (2020), pp. 294–308.

[17] A. de Boer, M.S. van der Schoot, and H. Bijl. “Mesh deformation based on radial basis function interpolation”. In: *Computers & Structures* 85.11 (2007), pp. 784–795.

[18] D. Maruyama, D. Bailly, and G. Carrier. “High-Quality Mesh Deformation Using Quaternions for Orthogonality Preservation”. In: *AIAA Journal* 52.12 (2014), pp. 2712–2729.

[19] A. Samareh. *Application of Quaternions for Mesh Deformation*. Tech. rep. TM-2002-211646. NASA Langley Research Center, 2002.

[20] K. Shoemake. “Animating rotation with quaternion curves”. In: *12th annual conference on Computer graphics and interactive techniques*. Vol. 19. 3. San Francisco, California, USA, 1985.

[21] J. Palm and C. Eskilsson. “Facilitating Large-Amplitude Motions of Wave Energy Converters in OpenFOAM by a Modified Mesh Morphing Approach”. In: *International Marine Energy Journal* 5.3 (2022), pp. 257–264.

[22] D.P. Kingma and M. Welling. “Auto-Encoding Variational Bayes”. In: *2nd International Conference on Learning Representations*. Banff, AB, Canada, 2014.

[23] O. Litany et al. “Deformable shape completion with graph convolutional autoencoders”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. Salt Lake City, 2018, pp. 1886–1895.

[24] T. Groueix et al. “AtlasNet: A Papier-Mâché Approach to Learning 3D Surface Generation”. In: *EEE Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT, USA, 2018, pp. 216–224.

[25] B. Beuthien, A. Kamen, and B. Fischer. “Recursive Green’s Function Registration”. In: *Medical Image Computing and Computer-Assisted Intervention*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 546–553.

[26] B. Maiseli, Y. Gu, and H. Gao. “Recent developments and trends in point set registration methods”. In: *Journal of Visual Communication and Image Representation* 46 (2017), pp. 95–106.

[27] B. Jian and B.C. Vemuri. “Robust Point Set Registration Using Gaussian Mixture Models”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.8 (2011), pp. 1633–1645.

[28] H. Zhu et al. “A Review of Point Set Registration: From Pairwise Registration to Groupwise Registration”. In: *Sensors* 19.5 (2019), p. 1191.

[29] G. Tam et al. “Registration of 3D Point Clouds and Meshes: A Survey from Rigid to Nonrigid”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.7 (2013), pp. 1199–1217.

[30] B.J. Brown and S. Rusinkiewicz. “Global Non-Rigid Alignment of 3-D Scans”. In: *ACM Transactions on Graphics* 26 (2007), p. 21.

[31] P.J. Besl and N.D. McKay. “A method for registration of 3-D shapes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992), pp. 239–256.

[32] S. Cheng et al. “Statistical non-rigid ICP algorithm and its application to 3D face alignment”. In: *Image and Vision Computing* 58 (2017), pp. 3–12.

[33] F. Wang and Z. Zhao. “A survey of iterative closest point algorithm”. In: *Chinese Automation Congress (CAC)*. Jinan, China, 2017, pp. 4395–4399.

[34] J. Zhang, Y. Yao, and B. Deng. “Fast and Robust Iterative Closest Point”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.7 (2022), pp. 3450–3466.

[35] A. Myronenko and X. Song. *Non-rigid point set registration: Coherent Point Drift*. 2006.

[36] A. Myronenko and X. Song. “Point Set Registration: Coherent Point Drift”. In: *IEEE transactions on pattern analysis and machine intelligence* 32.12 (2010), pp. 2262–75.

[37] A. Dempster, N. Laird, and D. Rubin. “Maximum likelihood from incomplete data via the EM algorithm”. In: *Journal of the Royal Statistical Society. Series B* 39.1 (1977), pp. 1–38.

[38] SR. Ahmed, G. Ramm, and G. Faltin. *Some salient features of the time-averaged ground vehicle wake*. Tech. rep. 840300. SAE Technical Paper, 1984.

[39] A.A. Gatti and S. Khallaghi. “PyCPD: Pure NumPy Implementation of the Coherent Point Drift Algorithm”. In: *Journal of Open Source Software* 7.80 (2022), p. 4681.

[40] A.N. Tikhonov and V.Y Arsenin. *Solutions of ill-posed problems*. V. H. Winston & Sons, 1977.

[41] O. Hirose. “A Bayesian Formulation of Coherent Point Drift”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.7 (2021), pp. 2269–2286.

[42] A.L. Yuille and N.M. Grzywacz. “A mathematical analysis of the motion coherence theory”. In: *International Journal of Computer Vision* 3.2 (1989), pp. 155–175.

[43] N. Rooy. *ahmed-bluff-body-cfd*. GitHub repository. 2024. URL: `https://github.com/nathanrooy/ahmed-bluff-body-cfd`.

[44] H. Pottmann et al. “Geometry and convergence analysis of algorithms for registration of 3D shapes”. In: *International Journal of Computer Vision* 67.3 (2006), pp. 277–296.

[45] M. Cicconi et al. “Statistical Shape Model: Comparison between ICP and CPD algorithms on medical applications”. In: *International Journal on Interactive Design and Manufacturing* 14.4 (2020), pp. 1341–1350.

[46] J.C Hart, G.K Francis, and L.H Kauffman. “Visualizing quaternion rotation”. In: *ACM Trans. Graph.* 13.3 (1994), pp. 256–276.

[47] L. Li et al. “Rigid Point Set Registration Based on Cubature Kalman Filter and Its Application in Intelligent Vehicles”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.6 (2018), pp. 1754–1765.

[48] A. Rasoulian, R. Rohling, and P. Abolmaesumi. “Group-Wise Registration of Point Sets for Statistical Shape Models”. In: *IEEE Transactions on Medical Imaging* 31.11 (2012), pp. 2025–2034.

[49] F. Wang, B.C. Vemuri, and A. Rangarajan. “Groupwise point pattern registration using a novel CDF-based Jensen-Shannon Divergence”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. New York, NY, USA, 2006, pp. 1283–1288.

[50] G.D. Evangelidis et al. “A Generative Model for the Joint Registration of Multiple Point Sets”. In: *European Conference on Computer Vision*. Springer. Zurich, Switzerland, 2014, pp. 109–122.

Appendix A

Quaternions

This Appendix focuses on quaternion notations, important to understand the theory underlying the SLERP mesh motion method. This reminder closely follows the work of Foley et al. [46] and Maruyama [18]. The reader is referred to those documents for additional information. Quaternions are widely used in computer graphics, robotics and aerospace for instance. In the aerospace framework, one could for example mention flight dynamics or attitude control systems for satellites. The quaternion is a three-dimensional extension of a complex number, whose set is often denoted as $\mathbb{H}$. The particularity of this set is that it does not satisfy commutative operations for the multiplication. A quaternion Q is a vector defined as [46]

$$Q \triangleq [q_0, q_1, q_2, q_3]^T = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} = [s, \mathbf{v}] \tag{A.1}$$

where **i**, **j** and **k** are the fundamental quaternion units (or equivalently basis vectors). It can alternatively be represented by a scalar part s (real number) and a vector part **v** (imaginary number). The basis vectors follow the following rules

$$\begin{aligned} \mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1 \qquad & \mathbf{ij} = \mathbf{k} \qquad & \mathbf{ji} = -\mathbf{k} \\ \mathbf{jk} = \mathbf{i}, \quad \mathbf{kj} = -\mathbf{i} \qquad & \mathbf{ki} = \mathbf{j} \qquad & \mathbf{ik} = -\mathbf{j} \end{aligned} \tag{A.2}$$

Similarly to a complex number, the conjugate operation Q^* can be defined, alongside with the magnitude and the inverse operations

$$Q^* = q_0 - q_1\mathbf{i} - q_2\mathbf{j} - q_3\mathbf{k} \qquad |Q| = \sqrt{QQ^*} = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} \qquad Q^{-1} = \frac{Q^*}{QQ^*} \tag{A.3}$$

Rotation Quaternions

Among the quaternion set $\mathbb{H}$, a subset of interest is the set of quaternions of magnitude $|Q| = 1$. It is called the unit quaternion set, or rotation quaternion set. It forms a hypersphere and is denoted as $\mathbb{S}^3 \subset \mathbb{H}$. It is initially introduced by Shoemake [20]. To represent a rotation, the unit quaternion can be used. A rotation θ about the axis $\mathbf{u}$ can be represented by the unit quaternion [18]

$$Q = \cos\frac{1}{2}\theta + \mathbf{u}\sin\frac{1}{2}\theta \qquad (|Q| = 1) \tag{A.4}$$

Quaternions are often used to represent rotations because they avoid some of the problems associated with other rotation representations, such as gimbal lock with Euler angles. It is a loss of one degree of rotational freedom in a 3D space due to the alignment of two of the three gimbals, causing the system to become unable to rotate around one axis [18]. Quaternions are also useful for smooth interpolation between rotations (SLERP). Lastly, Rotating a vector using quaternions is computationally efficient. Those three reasons make the use of quaternions optimal in a mesh deformation context. Let us now consider a point $\vec{p} = (x, y, z)$. It can be written as a quaternion $P = [0, \mathbf{p}]$. To rotate that point, the following equation can be used [18]

$$P_{\text{rot}} = QPQ^* \tag{A.5}$$

where Q is a rotation quaternion defined as in Equation (A.4). In a mesh deformation context, a translation vector is added to that equation, which is the one found using a point set registration algorithm. This gives Equation (1.5). In a mesh deformation context, this vector Q is found as following. Normal vectors ($\mathbf{n}_u$, $\mathbf{n}_d$) are defined using cells composed of neighboring dual-mesh nodes. The rotation angle and axis from Equation (A.4) are defined as [18]

$$\theta = \cos^{-1}(\mathbf{n}_u \cdot \mathbf{n}_d) \qquad \mathbf{u} = \mathbf{n}_u \times \mathbf{n}_d \tag{A.6}$$

It gives Q_1, which represents the modification of the cell orientation during the mesh deformation. In 3D, the internal shape modification is also taken into account. To do so, a quaternion Q_2 representing of the rotation due to cell shape modification during the mesh deformation is take into account [18]. The rotation to apply to every point is then a composition of those two quaternions, such that $Q = Q_2 Q_1$. For more detailed information, the reader is referred to [18]. As a last remark, OpenFOAM already provides an efficient implementation of the quaternions and the respective SLERP framework.

Appendix B

Point Set Registration (PSR) Overview

This section summarize briefly the different point set registration algorithms. Two PSR algorithms are investigated in this work. Providing an overview of the alternatives will gives an idea on how the current methodology could be improved. A detailed comparison of the differently existing methods is provided in [28]. There is overall two types of registration algorithms; pairwise and groupwise point set registration. A summary of the main existing methods is provided on Figure B.1.

Concerning pairwise point set registration, its goal is to find the suitable transformation and to establish the correct correspondences between two point clouds, **X** and **Y**. Within this category, the first subdivision is distance-based methods, which is a two steps method. The first step is to compute a distance between two point sets and to find the correspondences. Then, the distance between two point sets with the determined correspondences is minimized in the second step. This is the idea behind the ICP method from Besl et al. [31].

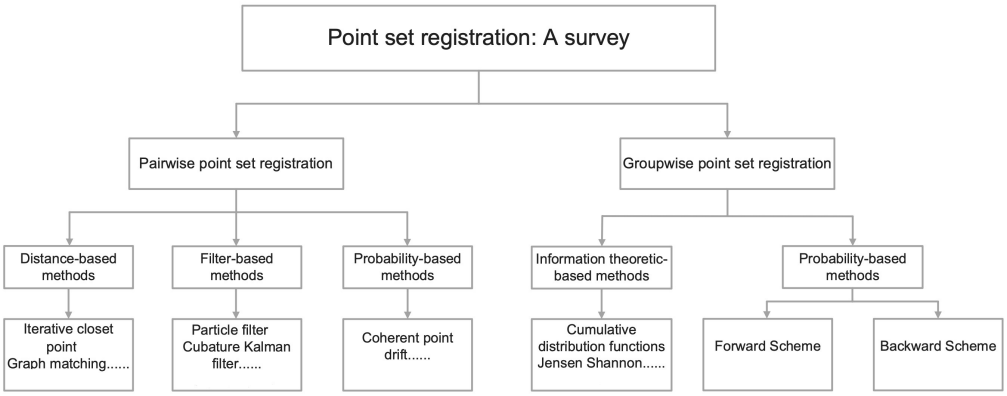

Figure B.1: Taxonomy of point set registration methods, from Zhu et al. [28].

The second subcategory concerns filtering-based methods. It is for example investigated in [47]. Those use a dynamical system modelling approach. The state $\mathbf{x}$ represents the transformation parameters (translation, rotation, and possibly scaling) that align the point sets. The state $\mathbf{x}_k$ at step k of the algorithm is described as

$$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} + \mathbf{v}_k \tag{B.1}$$

where $\mathbf{A}$ is the state transition matrix, and $\mathbf{v}_k$ is the process noise, usually assumed to be zero-mean Gaussian noise. Filtering-based methods such as the Cubature Kalman Filter (CKF) [47] are used to estimate the state vector iteratively. They are two-stepped. A prediction step and an update step. Those methods are useful for handling noise and dynamic changes in the data, making them robust for real-time applications in robotics and computer vision.

Groupwise point set registration refers to the process of aligning multiple sets of points simultaneously, rather than aligning just two sets as is done in pairwise registration. While pairwise registration involves aligning two sets of points at a time, groupwise registration aims to find a common coordinate system that minimizes the alignment error across all sets. As described in [48], this method can more effectively handle deformations, noise, occlusions, and outliers by leveraging information from multiple point sets.

The Cumulative Distribution Function (CDF) is used to capture and compare the overall distribution of points across the sets [49]. By aligning the point sets to minimize differences in their CDFs, the method effectively addresses variations such as noise and outliers, focusing on the global structure rather than individual point correspondences. This ensures that the point sets are aligned in a way that their distributions become as similar as possible, leading to more robust and accurate registration outcomes.

The forward scheme involves iteratively aligning each point set to a mean shape, which is continuously refined throughout the process. It is for instance investigated in [48]. Initially, an estimated mean shape is computed, and each point set is aligned to this shape using transformations that minimize the alignment error. After aligning all point sets, the mean shape is updated based on the newly aligned sets. In contrast, the backward scheme reverses the process by focusing on transforming the mean shape to align with each individual point set [50]. Starting with an initial mean shape, the algorithm calculates the transformations needed to map the mean shape onto each point set. This approach allows for direct adjustments to the mean shape according to the specific variations in each point set.

Coherent Point Drift (CPD) Hyperparameters

This section presents sensitivity testing on the hyperparameters of the Coherent Point Drift (CPD) algorithm, using a non-rigid transformation to deform the fish example from Figure 2.5. The first hyperparameter examined is λ, which influences the regularization term in Equation (2.4). According to [39], λ controls the trade-off between the goodness of fit and regularization. A higher λ value results in more rigid deformations, while a lower λ allows for more flexibility in the deformation. This parameter thus balances data fidelity (matching points) with transformation smoothness (rigidity). The effects of varying λ are illustrated in Figures B.2, B.3, and B.4, with β held constant at 2. The fish test case is taken from [39].

The second hyperparameter is β, which represents the width of the Gaussian kernel and determines the scale of interactions between points. This parameter adjusts the spatial extent over which points influence each other, thus controlling whether the deformation is more local or global. As mentioned in [39], when point clouds are normalized to a unit sphere, the scale of the points is consistent across different datasets, making the choice of β more dependent on the relative spacing between points rather than the original scale of the point cloud. The effects of varying β are shown in Figures B.5, B.6, and B.7.

In conclusion, high β and high λ lead to a very smooth and rigid transformation, similar to a global affine transformation. In contrast, low β and low λ allow for highly detailed, localized, and flexible deformations, though this may result in overfitting or non-smooth transformations.

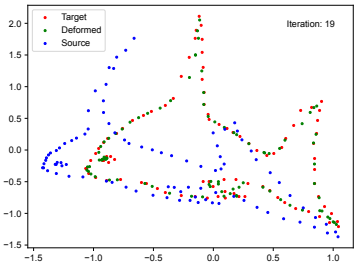

Figure B.2: $\lambda = 1$, $\beta = 2$

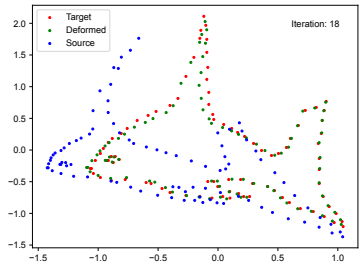

Figure B.3: $\lambda = 2$, $\beta = 2$

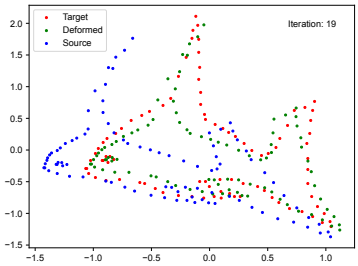

Figure B.4: $\lambda = 100$, $\beta = 2$

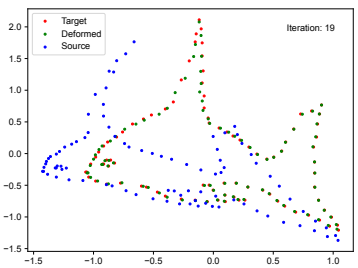

Figure B.5: $\lambda = 2$, $\beta = 1$

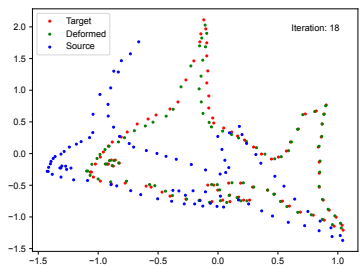

Figure B.6: $\lambda = 2$, $\beta = 2$

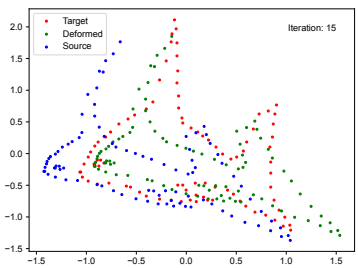

Figure B.7: $\lambda = 2$, $\beta = 100$

Appendix C

Ahmed Body Flow Simulation

The objective of this Appendix is to show that the implemented utility is compatible with native OpenFOAM flow solvers. These results are explanatory only, as finding a proper mesh resolution and numerical setup is considered to be outside the scope of this thesis. A Reynolds-Averaged Navier-Stokes (RANS) simulation in OpenFOAM is conducted to evaluate the impact of mesh deformation on the lift and drag coefficients for a vertically oscillating Ahmed body. The simulation setup is similar to the transient flow wing test case from OpenFOAM[1]. The results show that both lift and drag coefficients oscillate over time with a similar period. As mentioned in Section 3.1.1, the mesh quality remains high throughout the deformation, allowing a stable simulation over multiple cycles. On top of that, it outlines that the utility developed functions as intended with OpenFOAM solvers. For more information on the computation of the lift and drag coefficients and on the background behind RANS simulations, the reader is referred to [1].

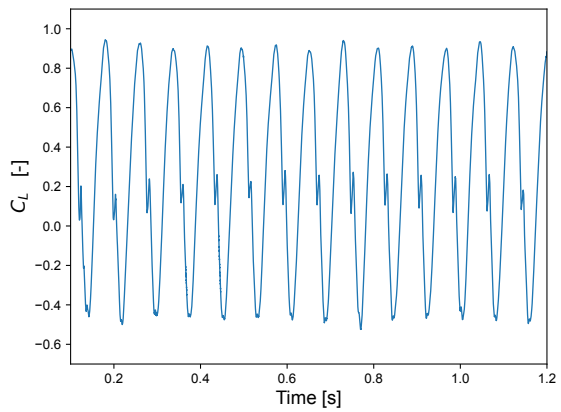

Figure C.1: Evolution of the lift coefficient C_L over time.

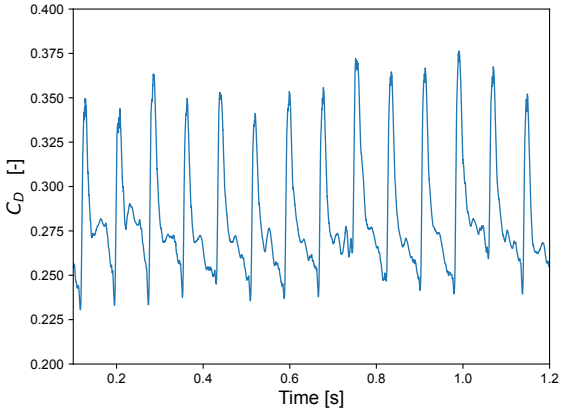

Figure C.2: Evolution of the drag coefficient C_D over time.

[1] `https://github.com/OpenFOAM/OpenFOAM-dev/tree/17280e978cf3fc8fd0f0bfd1e0eb4a0ed8fd1763/tutorials/incompressibleFluid/wingMotion/wingMotion2D_transient`