Design of an Aero Kit and CFD Workflow Development

Bachelor's Project

Association EPFL Racing Team
EPFL SKIL (RT), Bâtiment SKIL, Station 18, 1015 Lausanne

epfl-racing-team@epfl.ch
https://epflracingteam.ch

Prepared by: Maxence Colnot (373843), Nour Kanaan (357573), Daren Pilet (380012), Joseph Rizk (381086)

Supervisors: M. Favre & M. Boujo

Date: Spring Semester 2026



Acknowledgements

We would like to thank M. Favre for his valuable help in the completion of this project, as well as Gaël Vignau and all the members of the Aerodynamics Section for their continued support and guidance. We would also like to thank the DLL for access to the remote computers, without which running CFD simulations would not have been possible.

Contents

Acknowledgements **1**

Introduction **7**

1 CFD Automation **9**
1.1 Introduction . . . 9
1.1.1 CFD in the Racing Team . . . 9
1.1.1.1 General process . . . 9
1.1.1.2 Methodology . . . 10
1.1.2 Problem Statement . . . 11
1.1.2.1 Automation as a solution . . . 11
1.2 Post Processing automation in Paraview . . . 13
1.2.1 Status of Paraview Automation before the project . . . 13
1.2.2 Post process metodology . . . 13
1.2.2.1 Pressure coefficient . . . 14
1.2.2.2 Total pressure coefficient . . . 14
1.2.3 Implementation of the code . . . 15
1.2.3.1 Rewriting the Parascript . . . 15
1.2.3.2 Modifications to the parafunction . . . 18
1.2.4 Improving the code . . . 19
1.2.4.1 Point to cell . . . 20
1.2.4.2 Color mapping . . . 20
1.2.4.3 Cleanup code . . . 23
1.2.5 Conclusion on post process automation . . . 24
1.3 Fluent Automation : technical specifications . . . 24
1.3.1 Overview . . . 24
1.3.1.1 Summary . . . 24
1.3.1.2 Glossary . . . 24
1.3.2 Background and Problem Statement . . . 25
1.3.2.1 Context . . . 25
1.3.2.2 Current level of Automation . . . 26
1.3.2.3 Why the Problem is Worth Solving . . . 27
1.3.3 Literature Review . . . 27
1.3.4 Goals and Scope . . . 27
1.3.4.1 Goals . . . 27
1.3.4.2 Scope . . . 28
1.3.5 Assumptions and Dependencies . . . 28
1.3.5.1 Assumptions . . . 28
1.3.5.2 Dependencies . . . 29
1.3.6 Proposed Solution . . . 29
1.3.6.1 Design Principle . . . 29
1.3.6.2 High Level Architecture . . . 30

1.3.7 Module Specifications and Implementation . . . 32
1.3.7.1 CAD import . . . 32
1.3.7.2 Mesh generation . . . 34
1.3.7.3 Physics Setup . . . 37
1.3.7.4 Reports Definitions . . . 41
1.3.7.5 Initialization and Run . . . 42
1.3.7.6 Post process . . . 44
1.3.7.7 Result exports . . . 45
1.4 Implementation of the Fluent Automation . . . 47
1.4.1 Main.jou . . . 48
1.4.1.1 Specification deviation : suppression of the error log file . 48
1.4.2 globals.scm . . . 49
1.4.3 Watertight geometry workflow . . . 52
1.4.4 CAD import module . . . 53
1.4.5 Meshing . . . 53

2 2D **55**
2.1 Introduction . . . 55
2.1.1 Context . . . 55
2.1.2 Objectives . . . 55
2.2 Multi-Element Wing Theory . . . 56
2.2.1 Airfoil terminology . . . 56
2.2.2 Aerodynamic Behaviour of a Single Element airfoil . . . 57
2.2.3 Limitations of a Single Wing . . . 59
2.2.4 Theory of a Multi-Element Wing . . . 62
2.2.5 Parameters of a Multi-Element Wing . . . 62
2.2.5.1 Slot Gap and Overlap . . . 62
2.2.5.2 Element Loading Distribution . . . 63
2.2.5.3 Global Camber and Alignment . . . 63
2.2.6 Performance benefits . . . 64
2.3 Description of the Workflow and Software Used . . . 66
2.3.1 Simulation approach . . . 66
2.3.2 Workflow Overview . . . 68
2.3.3 Geometry generation . . . 68
2.3.3.1 Matlab Code overview . . . 69
2.3.3.2 Geometry and Enclosure creation . . . 71
2.3.3.3 Rotation Convention adaptation . . . 72
2.3.4 Mesh . . . 73
2.3.4.1 Mesh Parameters . . . 73
2.3.4.2 Adaptive Mesh Refinement . . . 74
2.3.5 CFD Configuration and Solution . . . 75
2.3.5.1 Parameters . . . 75
2.3.5.2 Post-Processing . . . 76
2.4 Mesh Convergence Study and Method Choice . . . 78
2.4.1 Solver Method Comparison . . . 78
2.4.2 Mesh Convergence Study . . . 78
2.4.2.1 Procedure . . . 78
2.4.2.2 Results . . . 79

2.5 Case Study — 2D vs 3D Configuration . . . 81
2.5.1 Physical Differences Between 2D and 3D Flow . . . 81
2.5.2 Model Environment Differences . . . 82
2.5.3 Correction Method — Rotation of the Inflow Direction . . . 84
2.5.3.1 Spanwise section selection . . . 84
2.5.3.2 Inflow rotation . . . 85
2.5.4 Results and Selection of the Baseline . . . 85
2.5.4.1 Variable of Interest . . . 85
2.5.4.2 Graphical Results and interpretation . . . 86
2.6 Optimisation of the Four-Element Rear Wing . . . 88
2.6.1 Initial 3D Geometry . . . 89
2.6.2 2D Geometry Presentation . . . 90
2.6.2.1 2D baseline parameters . . . 90
2.6.3 Comparison: 2D Baseline vs 3D Baseline . . . 91
2.6.4 Description of the Analysis . . . 91
2.6.4.1 Starting Point . . . 91
2.6.4.2 Chord Variation . . . 91
2.6.4.3 Slot Gap and Overlap Variation . . . 91
2.6.4.4 Angle of Attack Variation . . . 92
2.6.5 Summary of Iterations . . . 92
2.6.6 Conclusion of the 2D Analysis and Transfer to 3D . . . 92
2.7 Results and Discussion . . . 93
2.7.1 3D CFD Validation of the Updated Design . . . 93
2.7.2 Overall Assessment . . . 93
2.7.3 Limitations of This Study . . . 94
2.7.3.1 Residual 2D/3D Gap . . . 94
2.7.3.2 Turbulent Inflow . . . 94
2.7.3.3 Steady-State Assumption . . . 94
2.7.4 Benefits in the Formula Student Context . . . 94
2.7.5 Possible Improvements and Future Work . . . 94
2.7.6 Conclusion . . . 95

3 Rear Wing 96
3.1 Introduction . . . 96
3.1.1 Context . . . 96
3.1.2 Problem Statement . . . 96
3.1.3 Objectives and Approach . . . 97
3.2 Constraints and Design Space . . . 97
3.2.1 Current FSG 2026 Rules . . . 97
3.2.2 Draft 2027 Ruleset . . . 98
3.2.3 Comparison Between 2026 and Draft 2027 . . . 99
3.2.4 Resulting Design Envelope . . . 99
3.3 Multi-Element Wing Theory . . . 99
3.3.1 Airfoil terminology . . . 99
3.3.2 Force Generation and Pressure Distribution . . . 100
3.3.2.1 Force Generation . . . 100
3.3.2.2 Pressure distribution . . . 100
3.3.3 Limits of single element Wing . . . 101

3.3.4 Principle of multi-element . . . 101
3.3.5 Slot Gap, Overlap . . . 102
3.4 Design methodology . . . 103
3.5 Design Iteration . . . 105
3.5.1 Three-element Configuration . . . 105
3.5.2 Four-element configuration . . . 109
3.6 Results and Discussion . . . 121
3.7 Conclusion . . . 122

4 Front Wing **123**
4.1 Introduction . . . 123
4.1.1 Context . . . 123
4.1.2 Problem statement . . . 123
4.1.3 Objectives and approach . . . 124
4.2 Constraints and design space . . . 124
4.2.1 Current FSG rules . . . 125
4.2.2 Draft 2027 ruleset . . . 126
4.2.3 Resulting design envelope . . . 127
4.2.4 Manufacturability . . . 129
4.3 Theoretical background . . . 129
4.3.1 Ground effect . . . 130
4.3.2 Aerodynamic balance . . . 131
4.4 Wing architecture . . . 132
4.5 Design methodology . . . 133
4.5.1 The C_{zt} versus Front Balance plot . . . 134
4.5.2 Worked example: the 5F family . . . 134
4.5.3 CFD post-processing . . . 135
4.5.4 Closing the loop . . . 136
4.6 Design iterations . . . 136
4.6.1 Establishing the architecture . . . 137
4.6.2 The four-element wing . . . 141
4.6.3 Convergence and strake studies . . . 151
4.7 Results and discussion . . . 154
4.8 Conclusion . . . 156
4.8.1 Two strands, one open comparison . . . 156
4.8.2 Reaching the balance target . . . 156
4.8.3 Future work . . . 156

Conclusion **158**

5 Annex **159**
5.1 Paraview automation complete code . . . 159
5.1.1 ParaScript_baseline . . . 159
5.1.2 ParaFunctions . . . 163
5.2 Fluent TUI journals . . . 190
5.2.1 main.jou . . . 190
5.3 Scheme codes . . . 191
5.3.1 globals.scm . . . 191
5.4 Matlab multi element set up code . . . 202

5.5 Solver method simuations for 2D workflow 214
5.5.0.1 Options Tested 215
5.5.0.2 Mathematical Differences 215
5.5.0.3 Results and Justification 215

Introduction

Formula Student is an international engineering competition in which student teams design, build, and race a small formula-style car. A team is scored not only on the speed of its car in the dynamic events (acceleration, skidpad, autocross, and endurance) but also on the engineering reasoning behind its design choices, defended in static events in front of judges who are practicing engineers. Performance and the justification of that performance carry roughly equal weight, and both reward the same thing: design decisions that are deliberate, quantified, and traceable rather than guessed.

At the level the EPFL Racing Team competes, aerodynamics has become one of the main levers separating cars. The dynamic events are dominated by cornering, braking, and short acceleration phases rather than by sustained high-speed straights, so the time a car loses to drag is small compared with the time it can gain by carrying more speed through corners. Downforce, which presses the tires into the road and lets them generate more grip, is therefore the primary aerodynamic objective for the team, with drag treated as a secondary cost to be controlled rather than minimized at the expense of load. The aerodynamic package that produces this downforce is built from several interacting surfaces: the front wing, the floor and diffuser, the side structures, and the rear wing, each of which sets the flow conditions for the others. Because of that coupling, the package cannot be developed one piece at a time in isolation, and it cannot be developed at all without a reliable way to measure what each design choice does to the car.

This report presents the aerodynamic development work carried out for the next EPFL Racing Team car. The car follows a two-year build cycle and the aerodynamic package is a clean-sheet redesign, timed to coincide with the major change in the Formula Student Germany (FSG) technical rules expected for the 2027 season. At the time of writing the 2027 ruleset exists only in draft form (FSG, 25.10.2025, *Draft Only, Subject to Changes*) [1], with the final version expected in September or October 2026. The package has therefore been developed against the draft rules and designed to need only minimal modification once they are finalized, ahead of manufacture.

The work is the product of four authors working in parallel within the aerodynamics division, and the report is organized to follow it. Chapter 1 addresses the foundation on which all of the design work rests: the team's CFD process. Every design decision in the later chapters is justified by a CFD simulation, and running those simulations by hand is slow, repetitive, and a source of inconsistency. The chapter describes the automation of the CFD pipeline, from post-processing in ParaView through to a full meshing-to-results workflow in ANSYS Fluent, with the goal of cutting the user time per iteration and standardizing the numbers that the rest of the report relies on.

Chapter 2 develops a complementary tool aimed at the same bottleneck from a different angle. Full 3D CFD of the whole car is accurate but expensive, which limits how many design variants can be explored. The chapter presents a 2D CFD workflow for the cross-section of a multi-element wing, validated against the 3D model through a mesh convergence study and an inflow-rotation correction, that reduces the turnaround time of a geometric iteration from days to hours and lets the most promising candidates be identified cheaply before they are confirmed in 3D.

Chapters 3 and 4 apply this foundation to the two wings. Chapter 3 covers the rear

wing, the largest single aerodynamic device on the car and the main contributor to total downforce, developing it from a three-element concept into a four-element configuration. Chapter 4 covers the front wing, the first surface the flow meets and the primary lever for tuning the longitudinal distribution of downforce between the two axles. The two wings are designed against a shared whole-car baseline that is updated every two to three weeks, so that each wing is always evaluated in the context of the current state of the rest of the car.

Taken together, the four chapters describe a single effort: building the tools to measure aerodynamic performance efficiently and consistently, and then using those tools to design the front and rear wings of the 2027 car. The conclusion draws the four strands back together and sets out the work that remains before the package is ready to manufacture.

Chapter 1

CFD Automation

1.1 Introduction

1.1.1 CFD in the Racing Team

Note

This section will describe the current manual CFD pipeline used within the Racing Team. The methodology was established by the team members in charge of the CFD. Therefore this section only aims to give the reader context on what the automation will replicate and build on top of. In particular how the methodology was implemented won't be discussed.

Through the Make initiative the Racing Team has access to remote workstation including *Ansys Fluent* licenses. These computer are used for the simulations on design iterations. Theses computer allow for meshes of up to around 13 millions elements and are shared with all the members of the different Make associations. The team also has access to a dedicated simulation computer which is roughly 4 times more powerful than the remote workstations allowing for meshes of up to 60 millions cells. This computer usage is reserved for methodology development, and cornering simulations. The automation will be implemented on the design simulation and intended to be used on the remote workstations.

The aerodynamic team is split between member dedicated on design and member dedicated on CFD. Within the CFD pole three subgroups are defined : CFD methodology, CFD automation and CFD validation. The general goal of the CFD group is to provide the necessary tools for the other members to run CFD on their own.

1.1.1.1 General process

The general process of the simulation from geometry to results can be summarized as the following :

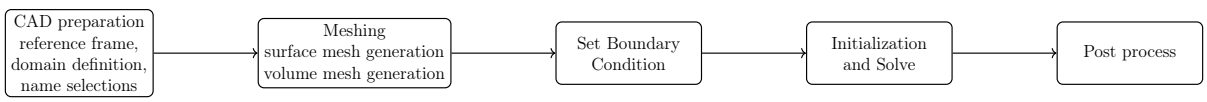


Figure 1.1: General structure of a CFD run

CAD preparation is handled in SpaceClaim creating an scdoc file that contains the prepared domain including partitioning the geometry in different name-selections that

will be used to define the boundary conditions.

Then this file is imported into Fluent Meshing Watertight Workflow where the surface mesh is created. The volume mesh is then created and refined using different Body Of Influence (BOI). Once the mesh is generated and improved, it is exported as a mesh .msh file.

Then the mesh file is imported into Fluent Solution Mode where Boundary Conditions are set, as well as turbulence model parameters. To monitor the convergence of the simulations force reports are defined on the whole car. Following this is the solution is initialized (standard initialization) and the calculations are run.

After the solution is converged scalar values are extracted from Fluent, they include the down force , lift coefficient (Cl), drag coefficient (Cd) for all the name selections, summing to the total over the whole car. The center of pressure in the length (COPx) allowing us to extract the balance of the car and the down-force on the front and rear axles (Czf and Czr respectively).

The rest of the post process focuses on qualitative analysis of the flow. The exact post process was not standardized but commonly used surface plots both on the car surface itself, where static pressure and skin friction coefficient are the commonly used variables. And surface plots of planar cuts over the length of the car at different offset values, for these velocity magnitude is plotted.

1.1.1.2 Methodology

The simulation used for design iteration is setup as the car moving in a straight line at 20 $[m/s]$. Therefore the simulation uses a half car domain with a symmetry plane in the middle. The car characteristic length are defined as follows :

	Length [mm]	Width [mm]	Heights [mm]
Whole car	2947	1534	1100
Front wing	650	1534	182
Rear wing	986	1418	410
One side	1150	491	366

Table 1.1: Characteristic dimensions of the car

The domain size is defined with respect to the car length which is approximated as $c.l = 3000\ [mm]$. The domain has the following dimensions :

Behind [c.l]	in front [c.l]	Above [c.l]	Side [c.l]
10	2	1.5	1.5

Table 1.2: Characteristic dimensions of the fluid domain

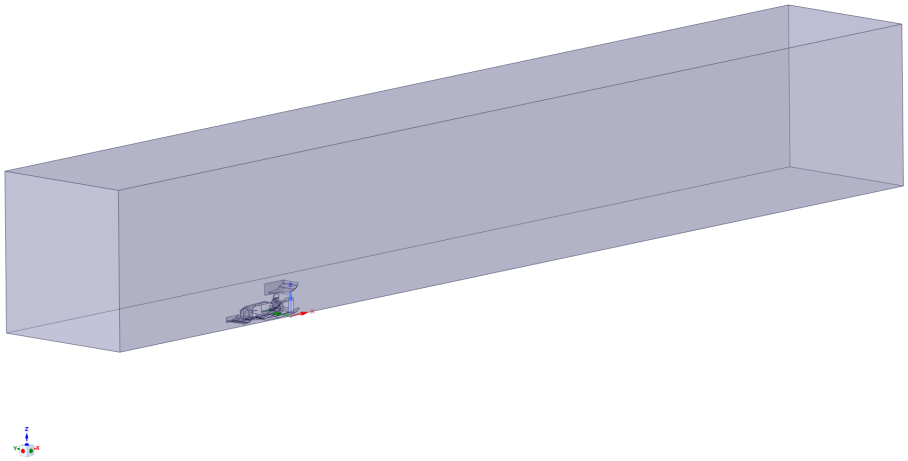


Figure 1.2: Resulting fluid domain used for simulations.

This fluid domain will be the input for the automation. The subsequent steps will only be described briefly here as the details will be listed in relevant automation sections. Meshing uses poly-hexcore elements, 3 refinement boxes are generated around the car in addition to boxes defined relative to some aerodynamic elements of the car, namely the front and rear wings as well as the diffuser. Inflation layers are defined around the car and on the moving ground, the simulation opts for a high $y+$ approach.
The solver uses a RANS $k-\omega$ shear stress transport turbulence model with single precision.

1.1.2 Problem Statement

In order to produce the most performant aerodynamic package for a formula student race car a large number of design iteration is necessary meaning an equally large number of CFD simulations to quantify the designs performance. For a student team the most critical resource is time as most of the work is uncredited and directly taken either from member's free time or study time. As the simulations are directly ran by the design team each minute spent on CFD could be better spend in optimizing the design. Additionally repetitive tasks are of little interest from the learning aspect of an interdisciplinary project. Two approaches can be followed in order to reduce time spent on CFD. Firstly increasing the amount of information that can be extracted from a single CFD run steering the design toward an optimal solution in less iteration. Secondly lowering the user time spent on a simulation. If successfully implemented equal performance could be reached faster and with equal time spend a better package could be produced whilst providing the members with more interesting work.

1.1.2.1 Automation as a solution

Automation was selected as the solution to this problem. The post processing could be improved since it was perviously limited by the time taken to extract all the relevant images. Work on this side had already been started before this project as it was deemed more critical and it would be possible to implement it with a manual workflow easily. The post processing automation will consist of a python script automating Paraview. Once completed the work transitioned into automating the rest of the pipeline.

However as mentioned previously time is the most critical resource within the team and it is legitimate to consider wether or not it is necessary to allocate a team member full time to automation. To answer this question let's exanimate the user time spent on a simulation run for a design iteration :

- Envelope preparation in Space claim : 10-15 minutes
- Fluent meshing : 40 minutes
- Fluent solution setup : 10-15 minutes
- Fluent post processing (scalar values, file export) : 5 minutes
- Post processing automation code launch : 2 minutes
- Value reporting in excel spread sheet : 5-10 min
- Images and video reporting in excel spread sheet : 20 minutes

Note

The time reported previously has been reported by team members after the completion of the post processing automation and the general process has been optimized by other CFD team members trough the automation of the envelop generation and distribution of reference fluent case files.

Over the length of the academic year over 150 simulation were ran, meaning the equivalent of 300 man hour of work was spent on repetitive CFD work. In term of design iteration a conservative estimate would mean that if the CFD time was cut by half at least a hundred more iteration could be made. The gain would also be retained for all the team's future members. Therefor the adventages far outweighed the cost of investing a team member full time on this project.

Another point worth mentioning for post processing automation is consistency. Multiple times during the year design decision were misguide by scaling errors, this would be completely alleviated by the use of an automation tool. This tool would also standardize the post process allowing the team to always have a good comparison point.

Of course automation cannot cut down computing resource usage and the time taken to complete a simulation. The real time gain is user time. However a full automation from meshing up to the post process could help the team to optimize the resource usage. Currently the simulation are usually launched in the evening and left to run for a higher number of iteration than needed for convergence (up to 10 times the necessary) using the automation and a batch script Fluent could be launched once and run multiple simulations convergence in the same time frame.

The following section of this chapter will describe the implementation of the Paraview script for post processing, the establishing of technical specifications for Fluent automation and the actual implementation under said specification.

1.2 Post Processing automation in Paraview

1.2.1 Status of Paraview Automation before the project

As mention in the introduction work on automating the post processing had started before the start of this project. The choice of using Paraview was actually guided by an external team member who had experience using the software. However as the software was not installed on remote computer and said team member did not have access to sufficient computing power the code was developed using a small simulation available from the Ansys tutorial for formula student [?].
A critical choice should be mentioned here : the Paraview code uses Ansys Fluent HDF5 files, namely cas.h5 and dat.h5 files. This has very limited compatibility, when opened in Paraview these files will only give access to the mesh and the nodal values stored as Fluent macros, for example the U velocity component is `SV_U` all the fluent macros that store the data use the prefix `SV_`. These limitation could be removed had we used a more compatible format like Ensight Gold but the team had struggled to have sufficient storage for the heavy simulation file and choose to keep the more compact HDF5 files.

Before the start of the semester the Paraview automation worked under the following structure : `paraScritp.py` would be called in the terminal and contained all the function calls to the second file `paraFunctions.py` which contained all the function and utilities to generate the images. Theses two files used the python API to control Paraview `pvpython`. The work done in this project included a complete rewriting of the `paraScritp.py` as `paraScritp_baseline.py` and modifications to `paraFunctions.py` to improve the images or add different variables to be plotted. The implementation in itself was done mostly focusing on fast implementation rather than clean implementation, for example some function have almost identical bodies such that no additional parameters and if condition needed to be written.

The final python scripts can be found in the Annex under 5.1. We will now describe the different work that has been completed, code snippet that will be mentioned here only contain code that was written during this project and not work done by other team members.

1.2.2 Post process metodology

The first step to complete the post process automation was defining which values we would extract. The main capabilities of the code are to extract : surface distribution on the mesh boundary, images of slices along all the direction of the domain and video of multiple slices in a direction. We thus need to choose which field variables would be plotted. The value selected are the pressure coefficient (C_p) and the total pressure coefficient (C_pT) though the velocity magnitude would be added later at the request of the design team who where used to mainly use it in the previous post processing.

1.2.2.1 Pressure coefficient

The pressure coefficient is defined as :

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho V_\infty^2} \tag{1.1}$$

This coefficient normalizes local static pressure against free stream dynamic pressure. Meaning that a value of $C_p \approx 1$ corresponds to stagnation and negative value means local succussion whilst undisturbed flow would have a local $C_p = 0$. Moreover the coefficient of pressure can easily be related to the force generated on an aerodynamic element. For example the lift coefficient of a 2D lifting wing can be obtained as an integral over the chord position $\frac{x}{c}$ of the pressure coefficient difference:

$$C_l = \int_0^1 (C_{p,\,lower} - C_{p,\,higher})\, d\left(\frac{x}{c}\right) \tag{1.2}$$

Therefor plotting the C_p on the surface of the allows to observe the repartition of the aerodynamic load on each elements. Consequently under loaded elements, premature pressure recovery or out of position stagnation can be easily identified. It also provides a good point to understand how different design can generate similar amount of down force.

When plotted on slices over the length or width of the car the pressure coefficient provides information on the pressure field around the car. Length wise slices help to better identify the suction peaks and ground effect phenomenons as well as additional visualization of pressure recovery and stagnation points. Meanwhile in the spanwise direction lateral non-uniformity and tip effects can be diagnosed.

1.2.2.2 Total pressure coefficient

The total pressure coefficient can be defined as :

$$C_{pT} = \frac{p_0 - p_\infty}{\frac{1}{2}\rho V_\infty^2} \tag{1.3}$$

The total pressure coefficient normalizes the local total (stagnation) pressure to the free stream dynamic pressure. Whilst looking similar to the pressure coefficient the information that can be derived from the total pressure coefficient are different. The total pressure coefficient is used as a measure of energy lost in the fluid. We can easily see that for the unperturbed free stream $C_{pT} = 1$, values over 1 are impossible as they would entail energy generation (note that energy generation could be observed as a result of numerical errors) and values $C_{pT} < 1$ indicate energy loss.

Plotting the total pressure coefficient on slices length wise and spanwise allow to examine flow structure, determine where losses are generated and where low energy flow or "dirty air" propagates to. Low energy zone can be caused by viscous forces from boundary layer, separated zones from a stalled wing on the tire wake and vortices. Careful visual inspection is necessary in order to identify which type of loss is present at different areas of the domain.

1.2.3 Implementation of the code

We will now discuss how the previously mentioned field variables extraction and visualization has been achieved. The process has generally taken two steps : first the existing code had to be adapted to work on the real simulation files and then the visualization was improved. The first step mainly consisted of writing the `paraScritp_baseline.py` file and then the second step modified the `paraFunctions.py` code.

1.2.3.1 Rewriting the Parascript

Whilst a new code with the majority of it's content rewritten some part of the previous `paraScritp_baseline.py` could be reused with minimal adaptation. One of such example is the code argument, to call the script tow argument are necessary `--dir` and `--file` and respectively contain the path to the folder housing the cas.h5 and dat.h5 files and the name of the files. The only adjustment made was to change the previous string operation used to more robust path operation using `os.path.join()`.

Once having dealt with file loading issues the work pivoted into figuring out how to extract the images and thus how to control the camera. The simplest already available function was `Visulaize3DFieldSurface`, this function allows to display the domain boundaries while plotting the different variables available directly from the Fluent files. Using the already available function controlling the camera required 4 variables : `cameraPosition`, `cameraOrientation`, `Zoom` and `cameraViewUP`. They defined respectively : the point at which the camera is placed, the point toward which the camera is looking, the zoom and which direction is up in the exported image. Note here that the origin of the coordinate system is placed on the ground at the center of the rear wheel axle. After fine tuning the variables the following view was extracted :

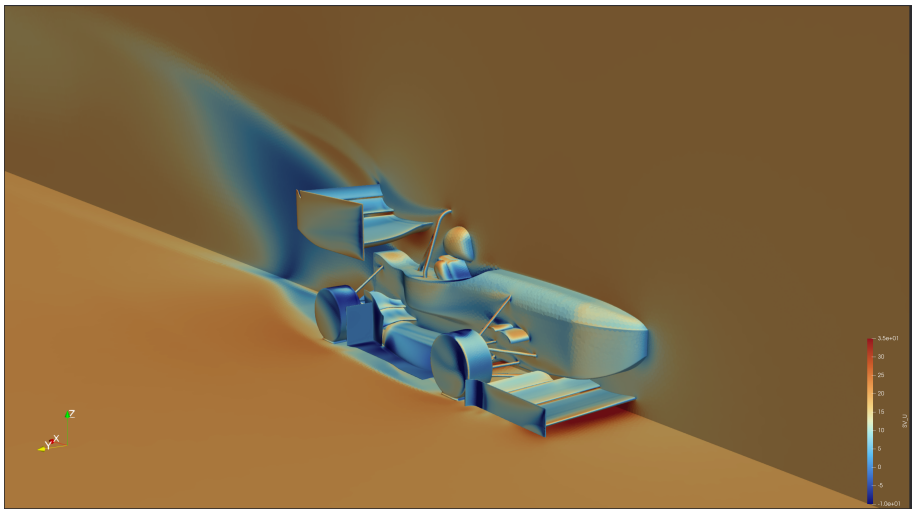


Figure 1.3: First that could be successfully extracted from Paraview — later used a context image to verify the CAD iteration.

The variable plotted here is the U velocity. Although this image is of limited interest for interpretation it was kept as a way to check what design iteration was present on a simulation should confusion arise.

Then implementing the surface plot was easy as it just required finding suitable position of the camera and calling the function. More modification were needed in the function called but from a `paraScript` point of view there where no challenges to the implementation.

The next step was generating the slices over the length of the car, as they where set to be static images rather than a video as we will see later for spanwise slices. The initial adaptation was made using the existing `VisualizePlaneCuts` function, when it was implemented it aimed to emulate how slices where used in fluent where multiple ones would be created and a single image exported showing the different slices from an off angle and setting them to be slightly transparent. This method created views that were difficult to analyze at best and misleading at worst. Thus it was chosen to simply export one image by slice, to gain time the function was not rewritten but simply got a single slice as input. The whole slice generation snippet would be :

Note

The following code corresponds to the final version of the `paraScript_baseline.py` and includes new functions `VisualizePlaneCutsCp` and `VisualizePlaneCutsSV_Mag`. In term of the implementation here there are no difference with `VisualizePlaneCuts`.

```
SliceDir = os.path.join(args.dir, "slices_Cp")
os.makedirs(sliceDir, exist_ok=True)
sliceDir = os.path.join(args.dir, "slices_SV_Mag")
os.makedirs(sliceDir, exist_ok=True)
# Define slicing inputs here:
# definition of equidistant slices in the direction given by the sliceNormal
↪  vector
slicePositions = [] # keep this array empty. It will get filled later on
sliceNormal = [0, 1, 0] # vector normal to the plane slice
#list the y values you want slices of in the following array :
y_values = [0.01, 0.2, 0.275, 0.3, 0.4, 0.525, 0.65]


#Slices of normal y plotting Cp
for sliceCoordinate in y_values:
    slicePositions.append([0, sliceCoordinate, 0])
    cameraPosition = [-1, sliceCoordinate-4.8, 1.3]
    cameraOrientation = [-1, sliceCoordinate, 1.3]
    imageName = f"{args.file}_y_{sliceCoordinate:.3f}"
    VisualizePlaneCutsCp(cash5, "Cp", Range_Cp, slicePositions, sliceNormal,
    ↪  cameraPosition, cameraOrientation , False, args.saveImages,
    ↪  str(args.dir), imageName, U_infty)
    VisualizePlaneCutsSV_Mag(
        cash5, "SV_Mag", Range_SV_Mag, slicePositions, sliceNormal,
        ↪  cameraPosition, cameraOrientation,False, args.saveImages,
        ↪  str(args.dir), imageName)
```

```
22    slicePositions=[] #this ensures that you get a single slice by image,
      ↪  initally the code is meant to make multiple slices on a single image,
      ↪  the array slicePositions is still here to allow easy implementation of
      ↪  this if needed
```

Since quite a few images would be saved a folder is created using `os.makedirs` and then a loop calls the function to generate images of the slices with different offset in the y direction (the normal of said slices), to keep the same point of view the camera position and orientation are offset by the same value.

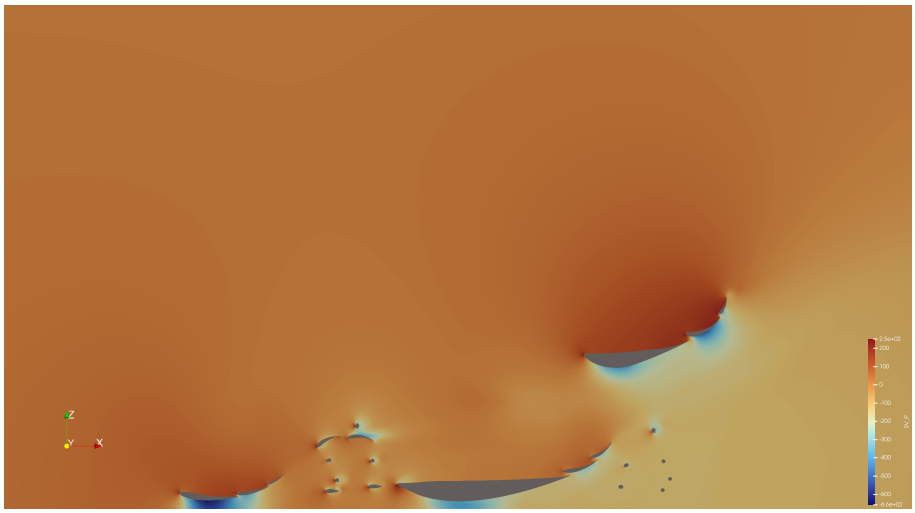


Figure 1.4: Result of the initial slices — here plotting the static pressure.

The last part of the `paraScript_baseline` that needed adjustment where the videos generated from slices in the spanwise direction. In the same way the previous part was implemented we started with a plain function that could only plot the flow variables stored as macros in the fluent files to figure out how to control it before adapting the function to plot the actual values we would be interested in. The video function worked in the following way : a static camera is defined and then the domain is sliced and successive images. A draw back from this implementation is the slight offset from image to image due to the static camera and the advance of the slices in the domain. Once again due to the focus on implementation speed it was judged as an acceptable compromise.

Additionally to the usual camera setup the slicing code required 3 additional parameters : `x_start`, `x_end` and the `NumberOfFrame`. The last parameter is set directly in the function the other two are passed as argument. Whilst sounding easy to use the only evident variable was the number of frame, the limits to the start and end of the slices did not correspond to actual coordinates. The underlying patter was hard to understand, varying one value and not the other would still impact the position of the unchanged "coordinate". The only solution was trial and error but this had proven extremely time consuming as Paraview did not seem very optimized at least in the context of this automation. A test would take about 10 minutes to run.
As videos with more slices took longer to run the initial idea was to find appropriate values with a limited number of slices and then increasing the resolution again. After fine tuning the limit it was discovered that changing the number of slices also impacted the

limits and the trial and error process had to be restarted.
After a huge time loss the final values would be 400 frames with `x_start = -13.5` and `x_end = 0`.



Figure 1.5: Screen shot from the initial video implementation — here plotting the static pressure.

1.2.3.2 Modifications to the parafunction

Once the correct function calls had been completed and every parameter was set as to be compatible with the racing team own CFD setup. Work could begin on working with the functions to get the actual variables we wanted, C_p and C_{pT}. In the same logic as previously mentioned of minimizing the time to implementation instead of a clean rewrite of the function to support C_p and C_{pT} the easier option of copy pasting the previous function bodies and changing the variable in the specific function body. The main advantage of this method is bypassing the need for new function argument meaning that the only adjustment needed in the function call was changing the name of the function called itself. The only exception would be the function plotting surface distribution which was the first to be implemented.

Let us use it to show how the two coefficient can be implemented in comparison to code simply plotting one of the available fluent "SV" variable. For this we will compare the contents of the `VisualizePlaneCuts` to the `VisualizePlaneCutsCP` function. First when using available variable :

```
1     source = get_data_source(readCase)
2    renderView = CreateView('RenderView')
3    slices = []
```

```
4
5       Display = Show(source, renderView, 'UnstructuredGridRepresentation')
6       Display.Representation = 'Surface'
7
8       # reset view to fit data
9       renderView.ResetCamera(False, 0.9)
10      renderView.Update()
11      ColorBy(Display, (data_type(), variable))
```

Secondly for C_p we need to define a Calculator that will compute the coefficient and then use it to color the mesh :

```
1       source = get_data_source(readCase)
2       renderView = CreateView('RenderView')
3       slices = []
4
5       # ---- Cp calculation ----
6       CpCalculator = Calculator(registrationName='Cp', Input=readCase)
7       U_sqr = str(U_infty * U_infty)
8       CpCalculator.Function = 'SV_P / (0.5 * SV_DENSITY * ' + U_sqr + ')'
9       CpCalculator.ResultArrayName = 'cp'
10
11      CpSource = get_data_source(CpCalculator)
12
13      Display = Show(CpSource, renderView, 'UnstructuredGridRepresentation')
14      Display.Representation = 'Surface'
15
16      renderView.ResetCamera(False, 0.9)
17      renderView.Update()
18
19      ColorBy(Display, (data_type(), 'cp'))
```

Straight forward implementation, meaning taking the base function body copy pasting it and adding the relevant calculator, was followed for the function that generated the pressure coefficient slice image over the length of the car and videos of the slices in the spanwise direction. Namely the following function where created under this pattern : `VisualizePlaneCutsCP`, `sliceMovieCp` and `sliceMovieCpT`. The last two function that where implemented differently are the `PlotSurfaceDistributionVariable` which contains an if condition when the variable Cp is passed. Meanwhile a function that plotted C_pT already existed `PlotCpT_xy_yz` but made slices both lengthwise and spanwise at the same time so the only adjustment made here was adding an if condition that removed the spanwise cut if no coordinate are specified.
The incoherent implementation was made following what at the time felt as the fastest solution. Whenever that is actually the case is up to debate.

1.2.4 Improving the code

The previous sub section described the minimum changes that where necessary to obtain a working code. The close collaboration with team member within the project in charge

of design of the front and rear wing allowed to collect feedback that was used to improve the quality of the post process.

Note

Whilst this report presented the steps of adapting the `paraScript` and the `paraFunction` as chronological steps before improving the code in reality the different step overlapped. We choose to present like this to make the progress easier to follow for the reader.

The goal of these improvement is to increase the readability of the images by emphasizing the part of the flow where something is happening and de cluttering the rest of the images.

1.2.4.1 Point to cell

The initial implementation of the code did not include any interpolation and the viewer would see the different mesh elements. To have a clearer view of the fluid structure using the interpolated values is desirable. As such the following utility functions were defined :

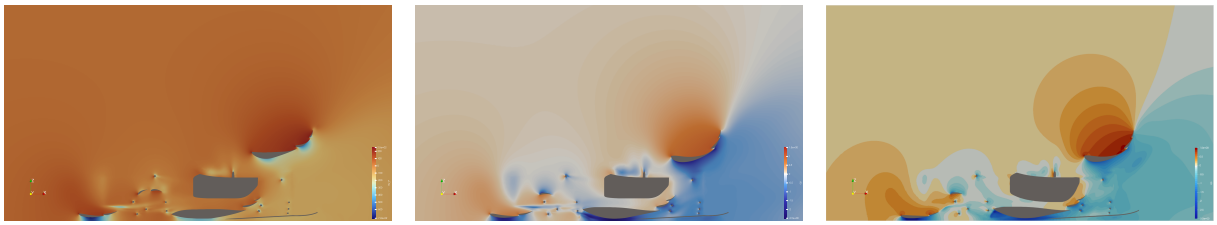
```
    def get_data_source(source):
        if USE_POINT_DATA:
            return CellDatatoPointData(Input=source)
        else:
            return source
    def data_type():
        if USE_POINT_DATA:
            return 'POINTS'
        else:
            return 'CELLS'
```

When called the data source would be converted from cell data to point data applying the interpolation as wanted. The source being the Fluent files as loaded by the paraview function `FluentCFFCaseReader`. This was implemented early in the development of the code so the only image present in this report without featuring it is 1.3.
Only the surface distribution function would not allow for point data to be implemented as it's logic worked by plotting element that are under a certain offset from the domain edge.

1.2.4.2 Color mapping

Most of the improvement made to the post processing entailed to color map optimization for both C_p and C_{pT}. The goal was to reach a point where the user eye would be attracted by the part of the flow impacted by the aero dynamic element whilst facilitating the interpretation of said fluid disturbance.

The color map evolution for the pressure coefficient :



(a) Initial color map (b) First iteration (c) Final iteration

Figure 1.6: Process of improving the color map for the pressure coefficient.

Note

Keen eyed reader will have seen that the first image represent the static pressure rather than the static pressure coefficient. This was an error caused by miscommunication and was rapidly corrected.

As can be seen the first improvement was to recenter white at $C_p = 0$ to better differentiate areas of higher pressure against areas of lower pressure. In a second time the color were discretized, this allows the user to better see the variations and almost look at iso surfaces of the pressure coefficient. Critically the stagnation points on the wing leading edges are made clearer.
The implementation in python consisted in the following utility function :

```
    def ApplyCpColormap(cpLUT, lims):
        # Set white to be Cp=0
        cp_min = lims[0]
        cp_max = lims[1]

        # Negative side
        neg_cyan1 = 0.06 * cp_min  # very close to 0
        neg_cyan2 = 0.18 * cp_min
        neg_lightblue = 0.35 * cp_min
        neg_mid = 0.50 * cp_min  # middle of negative range
        neg_blue2 = 0.72 * cp_min
        neg_blue3 = 0.88 * cp_min

        # Positive side
        pos1 = 0.08 * cp_max
        pos2 = 0.35 * cp_max
        pos3 = 0.70 * cp_max

        cpLUT.ColorSpace = 'Lab'
        cpLUT.RGBPoints = [
            # Negative side: cp_min -> 0
            cp_min, 0.070, 0.100, 0.650,  # deep blue
            neg_blue3, 0.080, 0.180, 0.760,  # strong deep blue
            neg_blue2, 0.100, 0.300, 0.840,  # blue
            neg_mid, 0.130, 0.450, 0.900,  # clear blue at cp_min/2
            neg_lightblue, 0.220, 0.650, 0.930,  # light blue
            neg_cyan2, 0.420, 0.820, 0.900,  # soft cyan-blue
```

```
            neg_cyan1, 0.680, 0.900, 0.930,  # less saturated cyan just below
            ↪  0
            0.00, 0.960, 0.960, 0.960,  # white at Cp = 0

            # Positive side
            pos1, 0.980, 0.920, 0.700,  # pale yellow
            pos2, 0.950, 0.650, 0.200,  # orange
            pos3, 0.850, 0.350, 0.100,  # orange-red
            cp_max, 0.650, 0.000, 0.000  # dark red
        ]
        cpLUT.Discretize = 1
        cpLUT.NumberOfTableValues = 36
```

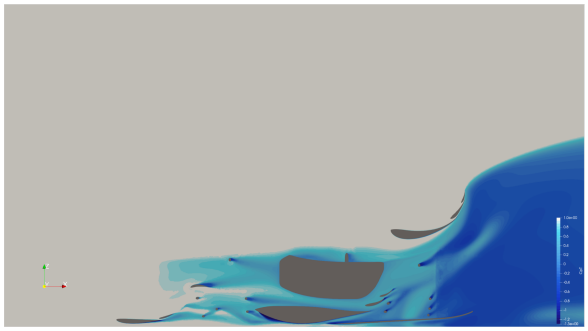
The total pressure coefficient color map also received gradual improvement to better visualize flow structures :



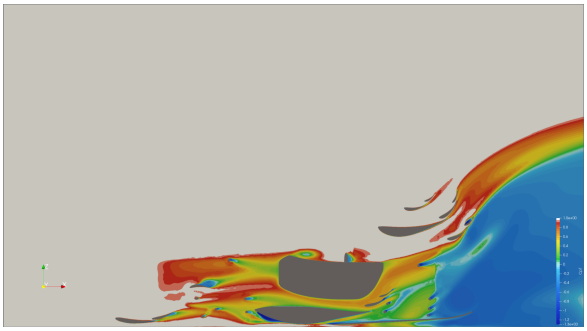
(a) Initial colormap



(b) Second iteration



(c) Third iteration



(d) Final iteration

Figure 1.7: Evolution of the color mapping for C_{pT}.

The first step in improving the color map was to put white at the free steam value of $C_{pT} = 1$ removing distraction from areas of the fluid that contained no useful information. Note that this color scale included red at values higher than 1, area near the floor can be see reaching such values. Whilst interesting for CFD validation, as it showed non physical numerical result, it is of little help for design optimization and can be, rightfully so, be considered confusing.
The next iteration capped the value at 1 and tried to help better differentiate the types of losses and better show flow structures. It was however deemed to have not enough contrast to achieve this goal. Discrete colors where also added at this stage.

This lead to the final version of the color map which included red, yellow and green before the blue scale, this greatly helped to differentiate between types of losses. Red comes right under the free steam and is mostly linked with small losses due to viscous forces, next comes yellow which is still small losses that are normally seen in the wake of any object. Green is set to represent values of $C_{pT} \approx 0$ these are more serious losses and could indicate a wing near it's stall point or delimits a vortex / recirculation region if blue can be seen in the middle. This leads us nicely to mention all shades of blue which correspond to negative total pressure coefficient values. This means significant losses either caused by a stalled object recirculation area or a vortex. To make a conclusion visual analysis of the flow structure is necessary. The implementation in python is equivalent to the C_p color scale and defines the function `ApplyCpTColormap`. The details can be seen in 5.1.

1.2.4.3 Cleanup code

After deploying the code one important issue was found : the lack of linkage between the image / video exported and the scalar results from the same simulation. This slowed down comparison between design iteration and complicated result archiving. Combined with another common issue in the number of files generated by the automation code a solution had to be devised.
To address both issue another python code was written, it would create a single pdf file with all the images from a simulation run and add title boxes containing all the scalar data from the simulation.

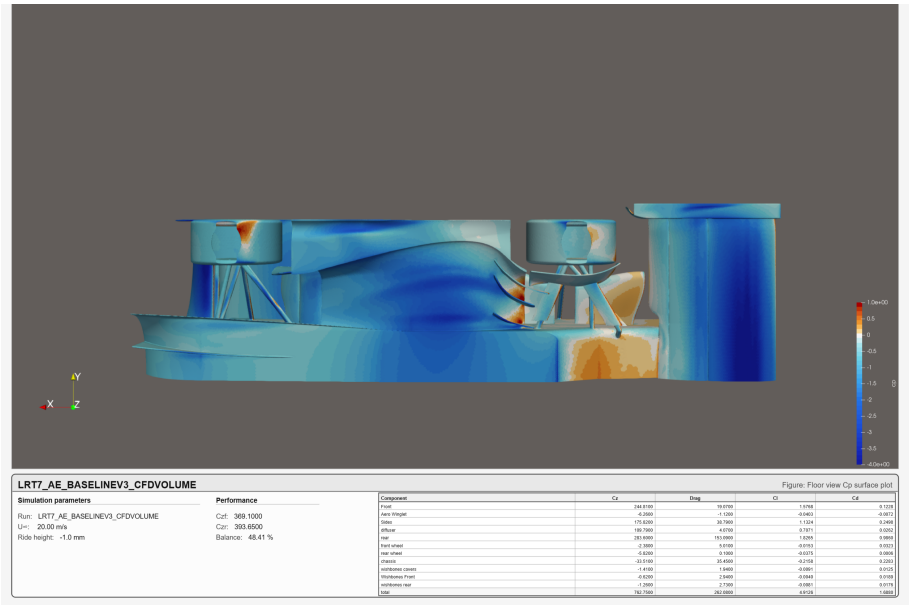


Figure 1.8: Resulting title box — is present on all the images.

Due to the minimal learning that could be gained from writing such code, which is just pdf formatting, the entire code was written by Artificial Intelligence. Due to the same limited interest in this code for the reader the implementation will not be mentioned here nor the full code will be added to the already long annex section.

1.2.5 Conclusion on post process automation

Thanks to the already present skeleton the automation of the post processing was successfully implemented and deployed during the span of this project. The new capabilities for design team member to extract high quality images rapidly with a single line of command and in a standardized fashion has proven instrumental in the optimization of all the aerodynamic elements of the car. It's application to Rear and Front wing design will be described in Chapter 3 and 4.

1.3 Fluent Automation : technical specifications

After successfully completed the automation of post processing completing the first approach to the problem at hand : minimizing the time spent on CFD within the racing team. It is now time to look into the second solution : the automation of the simulation itself.
As it is an ambitious project careful considerations had to be made in advance to build a coherent and successful automation. Thus the first part of the work done during the semester was establishing precise technical specification for the automation of Fluent.

1.3.1 Overview

1.3.1.1 Summary

This document defines the technical specification for the automation of the Fluent stage of the EPFL Racing Team CFD pipeline. This project aims to replace the current manual GUI based pipeline in order to reduce the human time spent by the design team on simulations. It will start from the SpaceClaim-prepared geometry and output the reports from the CFD run, the different metrics used by the design team and everything needed by the Paraview post process automation code.

This would normally have been implemented using a python pyfluent script, however due to license limitation it isn't possible. Thus the project will be structured around a combination of Ansys TUI commands in journal files and Scheme scripts.

1.3.1.2 Glossary

- **SpaceClaim**: CAD preparation environment used to orient the car, define the domain, and partition the geometry into Named slections.
- **Fluent Meshing**: Meshing environment used to generate surface and volume meshes.
- **Workflow .wft file** : A file type used to predefine the meshing steps, each steps then needs to be updated manually.
- **Fluent Solution Mode**: Solver environment used to define boundary conditions, initialize, and solve the CFD case.

- **TUI**: Text User Interface used to execute Fluent commands through journal files .jou or scripting.
- **Scheme**: Scripting language supported by Fluent and used to orchestrate commands, logic, and file generation. Allows more advanced scripting than plain TUI.
- **Named selection**: Named geometric ensemble from the base geometry. Used for Mesh generation and post process.
- **Cleanup script**: Python script generating the cleanup report from simulation outputs, includes all the images and scalar data from the CFD run.
- **Log file** text file storing the statuses of each step in the automation process, allowing to control program behavior on previous events.
- **RP variable** Read-Permit variables are Scheme variables stored as data outside of the Fluent Scheme interpreter and will persist after the switch to solution mode.

1.3.2 Background and Problem Statement

1.3.2.1 Context

The current CFD workflow within EPFL Racing Team is a mix of some automated steps and manual intervention. The pipeline can be schematized as seen below :



Figure 1.9: Specific structure of a CFD run

CAD preparation is handled in SpaceClaim creating an scdoc file that contains the prepared domain including the name selection that will be used to define the boundary conditions.

Then this file is imported into Fluent Meshing Watertight Workflow where the surface mesh is created. The volume mesh is then created and refined using different Body Of Influence (BOI). Once the mesh is generated and improved, it is exported as a .msh file.

Then the mesh file is imported into Fluent Solution Mode where Boundary Conditions are set, as well as turbulence model parameters. To monitor the convergence of the simulations force reports are defined on the whole car. Following this is the solution is initialized (standard initialization) and the calculations are run.

After the solution is converged scalar values are extracted form Fluent, they include the downforce Cz, lift coefficient (Cl), drag coefficient (Cd) for all the name selections, summing to the total over the whole car. The center of pressure in the length (COPx) allowing us to extract the balance of the car and the downforce on the front and rear axles (Czf and Czr respectively).

For qualitative analysis of the flow different views are extracted, firstly in Fluent skin friction is plotted on the surface of the car and cuts in the longitudinal direction plotting the velocity magnitude.

Then a Paraview automation script extracts the following views : static pressure coefficient (Cp) on the surface of the car and slices both along the length of the car of the

Cp (exported as images) and the width (exported as a video). The same slices images/ video are extracted for the total pressure coefficient (CpT). It is important to note that these steps are addition and not automation of previous manual tasks. When requested Cp plots on the wing can be extracted but this is not widely implemented nor is it an optimised process, in particular only a single team member knows how to do it. Once the post process is finished the results are manually reported in a shared excel sheet.

1.3.2.2 Current level of Automation

Note

The time needed for the simulations has been significantly reduced by the CFD team members. The main improvement has been the addition of a script handling the CAD preparation steps in Space Claim. All the steps form meshing to postprocessing in Fluent are done manually but they have been facilitated by loading a workflow .wft file for meshing and a previous case .cas.h5 file for the boundary conditions, model parameters. It also imports the different views, report definitions for the post processing.

The current process exhibits the following limitations :

- The manual inputs inside of the Fluent GUI demands a certain familiarity with the interface, leading to a significant time loss for the design team. Same steps are repeated multiple times by each team member.
- Fluent Meshing is dependent on up to date workflow file, which needs constant upkeep from the CFD team. When major design changes occurs (e.g. the addition of a new aerodynamic element on the car) the .wtf has to be updated.
- Whilst facilitated by the workflow file meshing requires the user to manually update each step, this requires the user to wait until completion of the surface and volume mesh before manually switching to Fluent.
- The boundary conditions and model parameters are dependent on the previous case file loaded, as the case are updated by each design team member for their personal post process needs and reused in the following cases there are no measure to prevent errors in BCs and model parameters form propagating to subsequent simulations neither any standard way to verify it.
- Changes in methodology require each user to manually change the parameters in the simulations or to redefine their post process in a case file shared by the design team.
- File management/ imports can cause different issues, in particular some CFD runs have been compromised due to incorrect mesh import in existing case file.
- Downstream scripts currently rely on manually entered values or hard-coded simulation parameters.
- Manual report of the scalar values isn't efficient, can induce some mistake and isn't compatible with the Cleanup script.

1.3.2.3 Why the Problem is Worth Solving

Automating the Fluent stage is worth solving because it:

- reduces setup time for repeated design evaluations,
- improves consistency across runs,
- reduces the risk of manual configuration errors,
- makes the CFD work easier to transfer between student generations,
- allows for standardized, controlled, and fast deployment of methodology improvements.

1.3.3 Literature Review

To gain sufficient information to write the detailed specification a thorough review of all the available documentation for this project was completed. Some useful information on meshing was found the the fluent user's guide [2] and most of the relevant text user interface information was found in the Ansys text command list [3] and additional information on data structure was found in the Ansys customization manual [4].

However finding proper documentation on Scheme for Ansys was significantly more challenging. No official document exist and only an external one was found [5]. The preamble mentioned that, circa 2003, Ansys had officially chosen python as the official scripting language replacing Scheme. Thus additional information on the Scheme integration in fluent was practically inexistent.

1.3.4 Goals and Scope

1.3.4.1 Goals

Note

This section defines the general goals of the fluent automation, what should be achieved to have a successful implementation.

The goals of the project are:

- automate the Fluent stage from Space Claim prepared CAD .scdoc file,
- keep manual input to a minimum, guarantee that it is simple and fast,
- remove dependence on predefined Fluent Meshing workflow files,
- tolerate varying named selections for car component excluding the wheel and contact patches,
- reproduce the Meshing, boundary condition, model parameters and initialization of the current CFD pipeline,
- parametrize the previously listed elements to allow for easy Methodology adjustments,

- export the views and scalar data in a format compatible with the cleanup code,
- produce an additional report on the simulation allowing to check its sanity, and trace the parameter used.

1.3.4.2 Scope

Note

The project mainly limits itself to the different tasks that are done by every team member in Fluent starting from the CAD import up to the post process.

This project aims to automate standard simulations run by the design team. To achieve the current goals the automation code should match the semi automated pipeline. The improvement that will be implemented are meant to improve either reliability and/or traceability. They include removing the dependency on workflow files and the addition of a simulation report.

The Fluent automation therefore finds itself in the middle of the pipeline. Upstream the Space Claim will provide the CAD domain and downstream the paraview and cleanup code complete the post process. This project will only guarantee the compatibility of the Fluent automation code and these steps. In particular the following elements are excluded from the scope of the project :

- correctness and maintenance of the SpaceClaim automation script,
- sanity check of the outputted .scdoc file from the SpaceClaim,
- automation of cornering simulation,
- simulation sanity checks outside of fatal errors and reports,
- higher level integration of a full design to result process (including the calls to SpaceClaim, Paraview and the cleanup),
- closed-loop design optimization or design-space exploration,
- higher level of automation like job scheduling,
- elimination of the need for review of unusual or failed case.

1.3.5 Assumptions and Dependencies

1.3.5.1 Assumptions

The proposed automation assumes:

- the geometry has already been prepared to a CFD-usable state in SpaceClaim,
- adequate file/ directory management including access to all the automation files and appropriate storage and access to the .scdoc file.
- Fluent version is 2024 R2 (FLuent v.242) or is compatible with TUI command for Fluent 2024 R2,

- the user can provide a run configuration file with physical and numerical parameters,
- ParaView and cleanup scripts can be adapted to read exported structured files.

1.3.5.2 Dependencies

The automation depends on:

- prepared geometry from the SpaceClaim stage,
- Fluent Meshing and Fluent Solution Mode availability,
- file-system access for run directories and output files,
- downstream Python scripts for visualization and report generation,
- documentation of current team CFD standards and baseline conditions.

1.3.6 Proposed Solution

1.3.6.1 Design Principle

Note

The automation process will be partitioned into different modules which will handle separate tasks in the simulation pipeline. Due to the post process being the most time consuming task currently it will be the first module to be completed. Following this the other modules will be added sequentially in the order of execution.

The general principle stems from the structure of the tools available namely the Fluent TUI and Scheme integration.

Fluent can be controlled by the TUI either as direct commands in the consol or by loading a journal file .jou . TUI is essentially a scripting language where the expression mimic GUI action (eg. "/define/boundary-conditions/set/wall surface () shear-bc yes shear-bc-spec-shear q", this sets a 0 shear boundary condition) it is organized as a menu system where the function is found under a "path" and then parameters are passed. The list of parameters are not listed in the documentation. However when in the consol direct command will lead the consol to prompt the user on how to fill each parameter. The parameters loosely correspond to the different fields a GUI user would fill inside of a dialog box. TUI only allows to execute fluent commands.

Loading a TUI journal file can be done either inside of the GUI or in a command prompt in the terminal. Note that a journal file can load another journal file.

Scheme on the other hand allows for all more "advanced" programming tasks, this includes variable definitions, function definition, and loading / parsing of external files. Fluent has different ways to support Scheme : for simple commands it can be incorporated in a TUI journal with all Scheme line contained in parenthesis, for more advanced application Fluent also handles scheme .scm files. TUI command can be included inside of a Scheme code from string using (ti-menu-load-string "TUI command").

Note that when using the journal feature in the GUI the .jou file that is not written in plain TUI but in Scheme and the TUI calls wrapped inside it are written using the abbreviation system for commands (e.g. "solve/initialize/hyb-initialization" will be written as

"solve init hyb"). The abbreviations are interpreted instead of being hard coded meaning that there are multiple possible abbreviate for the same command (e.g. s-i-h would also call the hybrid initialization). For improved readability this project will be written in full commands.

Note

Note that User Defined Functions (UDFs) in C are also available however since they aren't used in the current simulation process they won't be necessary in the scope of this project. They remain a useful tool that should be explored after completion for possible improvement to the simulations.

Given this framework each module of the automation will consist of both a .jou file with the main TUI commands as well as some simple Scheme command that will call upon a separate .scm file. This file will contain the custom function and variables used for this module, it will be loaded inside of the TUI journal using "File/Read/Scheme".

1.3.6.2 High Level Architecture

Note

Each module will be called by the flow control journal file main.jou whose role will strictly be limited to module call and flow control based on log status. The individual modules are defined by each separate process in the CFD pipline (CAD import, mesh, BCs,..). Global variables and helper functions are stored in a separate file as the switch from Fluent meshing to Solution mode may wipe the variables.

The high level architecture of the project including the different modules and the general flow structure as the contents of the main journal file can be seen on the file below:

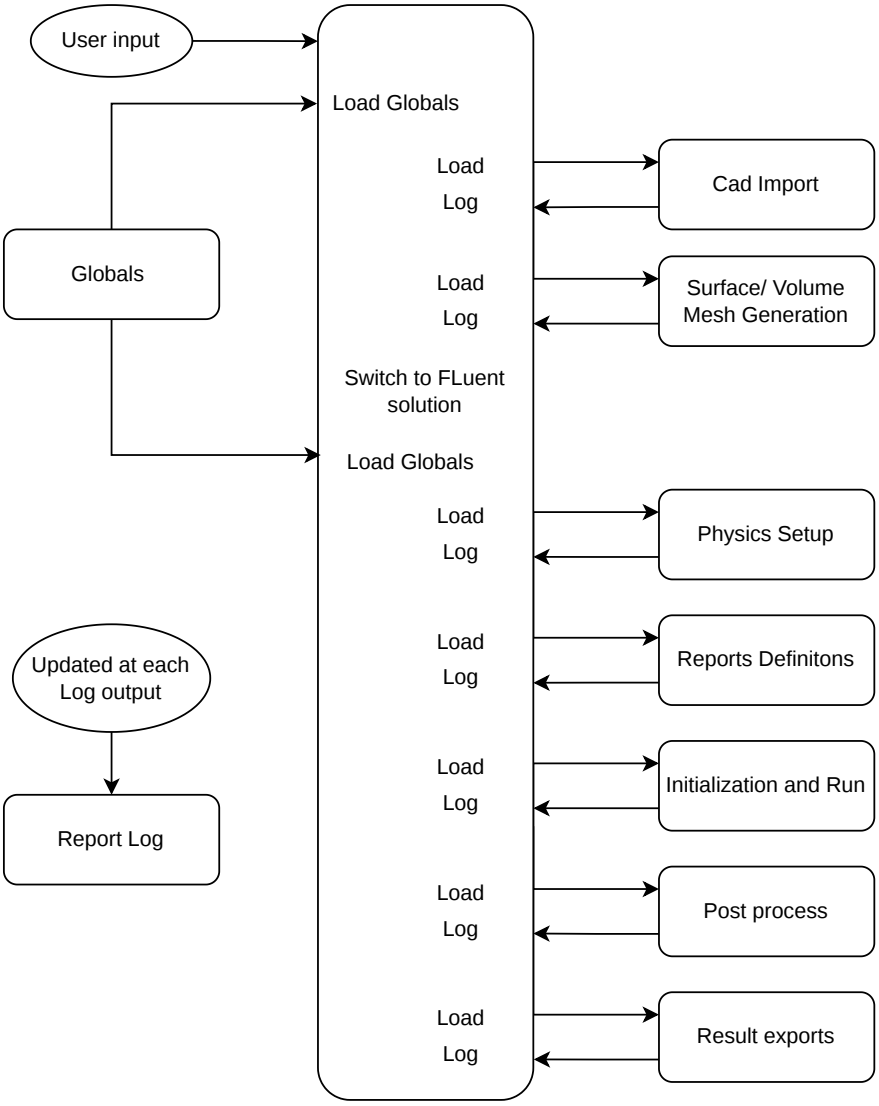


Figure 1.10: Automation General Structure

As already mentioned, each module will consist of a TUI journal file and a Scheme script for local variables / function. The journal will be loaded from the main, the Scheme will be called inside of the module journal.

At the end of each module a sanity check will be performed and the result will be recorded into the Log file. The log file will have 2 main roles : allow for error handling to control script exit conditions and to be a readable text file for the user.

The Global scheme file will contain global variables and helper functions. The global variables will include the simulation parameters (e.g. inlet velocity, turbulence model parameters, mesh refinement) which will also allow the user to change them in a safe and accessible way also allowing for easy parametric study on the simulations parameters (mesh/model).

1.3.7 Module Specifications and Implementation

Note

This section will detail the implementation of each module. It will detail each step and parameters of the current CFD process that should be replicated in the automation. The general structure of each module will be detailed and relevant TUI function will be listed.

1.3.7.1 CAD import

As the first step of the simulation the CAD import will import the relevant file containing the envelope. Then it should extract and parse the name selection contained in the .scdoc file, a sanity check that the name selection contain the envelope, wheels and contact patch will be performed. This is critical as specific BCs need to be defined on theses name selections. All the other name selections will be considered as part of the car and simple wall BC will be applied.
This setup allows for flexible name selection as the design of the car evolves. Then theses name selection need to be stored in accessible Scheme variables. However simple assignment will have a too limited scope. If defined in the CAD import journal or Scheme code the variable are under the risk of being wiped as the Scheme interpreter resets when switching to solution mode (otherwise variables defined in any Scheme script are defined outside a function are global and will be accessible anywhere).
The solution is to use Read-Permit variables (RP variables), these variables are stored as data outside of the Scheme interpreter and will persist after the switch to solution mode. They will also be accessible in UDFs should we want to use them. Note that once imported in fluent the name selection are stored as face zone labels (name + numerical id).

Main Steps of the Module

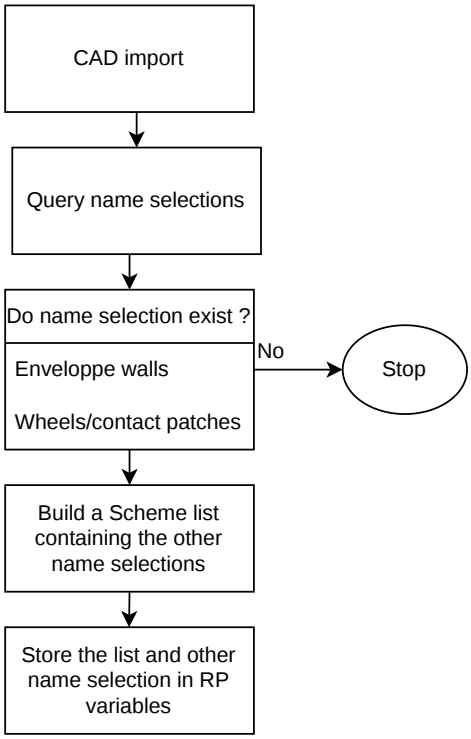


Figure 1.11: CAD import module structure

List of Relevant TUI/ Scheme Functions

Command	**Description**
file/import/cad	TUI command to import a CAD file
(inquire-zone-names)	Scheme command that inquires all the zone names
(get-zone)	Scheme command to get a specific zone as a Fluent object
(rp-var-define name type # f value))	Defines an RP variable of any type
(rpsetvar)	tEdits the value of an RP variable
(rpgetvar)	returns the value of an RP variable

1.3.7.2 Mesh generation

Note

Meshing is one of the task where the TUI is best suited for improvement on the current process. These specifications will be limited to the current process but after completion of the project exploring improvement to the meshing logic could lead to significantly higher mesh quality.

The element type used is poly-hexcore, the general structure of the mesh refinement is to create static Body Of Influence (BOIs) where mesh size is then specified. These BOIs are both defined as simple coordinate specified in the referential and with respect to the name selections. The meshing steps as parametrized in the workflow file are :

- Define refinement zones (BOIs) :
  Currently there are 3 BOIs that are defined from coordinates, they are simple rectangular boxes : near-field, middle-field, far-field. They are defined as, values in [mm] :

Box	X_{min}	X_{max}	Y_{min}	Y_{max}	Z_{min}	Z_{max}	**Sizing**
Near	-3000	3000	0	1500	0	2000	32
Middle	-4000	8500	0	2000	0	3000	64
Far	-5000	1500	0	2500	0	4000	128

  Then additional refinement boxes are defined from CAD name selection, note that here the coordinate correspond to an offset from the min / max coordinate of the zone in each direction.

Box	X_{min}	X_{max}	Y_{min}	Y_{max}	Z_{min}	Z_{max}	**Sizing**
Front wheel	0.1	1	0.1	0.1	0.1	0.1	16
Rear wheel	0.1	1	0.1	0.1	0.1	0.1	16
Rear wing	0.1	1.5	0	0.5	0.1	1	16
Diffuser	0.3	0.5	0.8	0.8	0.3	0.1	16

- Add local sizing :
  To refine the surface mesh local sizing is added on each name selection of the CAD. They are defined as follows, aero contains all the surfaces of the aero kit, values are given in [mm]:

Surface	**Growth Rate**	**Min Size**	**Max Size**	**Curvature normal angle [deg]**	**Scope to**
Chassis	1.2	1	8	12	Faces and Edges

Surface	**Growth Rate**	**Min Size**	**Max Size**	**Curvature normal angle [deg]**	**Scope to**
Aero	1.2	0.5	8	18	Faces and Edges
Wheels + patches	1.2	0.5	32	18	Faces

- Generate surface mesh :
  Generate the surface mesh and then improve it with a face quality limit of 0.6

- Describe geometry :
  The geometry consist of a single fluid zone. Then the boundaries are updated, all of type wall. Finally the region is update as Fluid.

- Add boundary layer :
  Inflation layer are defined on all the car elements as well as on the ground. The offset method use is *Last Ratio*. The layer parameters for both groups of zones are :

Zones (labels)	**Number of layers**	**Transition Ratio**	**First Height**	**Add in**
Car	4	0.272	1	Fluid region
Ground	6	0.272	1	Fluid region

- Generate Volume mesh :
  The volume Mesh is generated with the following settings :
  - Solver : Fluent
  - Fill with : poly-hexcore
  - Peel Layers : 1
  - Min Cell Length : 1 [mm]
  - Max Cell Length : 256 [mm]
  - Quality Method : Orthogonal
  - Quality Improve Limit : 0.05
  - Use Field Size ? : Yes
  - Polyhedral Mesh Feature Angle [deg] : 30
  - Avoid 1/8 octree transition ? : No
  - Check Self Proximity : No
  - Write Prism Control File : No
  - Merge Boundary Layer Cells within Regions? : Yes
  - Gap Factor : 0.25

- Max Aspect Ratio : 25
- Min Aspect Ratio : Keep First Boundary Layer Height? : No
- Adjacent Attach Angle [deg] : 80
- Use default stair-step handling? : Yes
- Normal Smooth Relaxation Factor : 100

Once the volume mesh is generated it is improved to a cell quality (orthogonal quality of 0.2.

Main Steps of the Module

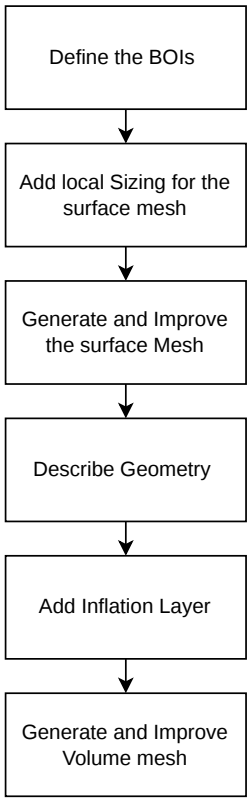


Figure 1.12: General structure of the meshing module

List of Relevant TUI/ Scheme Functions

Command	Description
scoped-sizing/create	TUI command to create a scoped sizing which is used to create both the surface mesh local sizing and the BOIs for volume Mesh
diagnostics/quality/general-improve	TUI command to improve the surface mesh
objects/improve-object-quality	Other TUI command to improve the surface Mesh
mesh/prism/create	TUI command to create an inflation layer on a specific Boundary face zone
objects/volumetric-regions/compute	TUI command to compute the volumetric region as needed in describe geometry
objects/volumetric-regions/change-type	TUI command allowing to set a volumetric region type as Fluid
mesh/hexcore/controls	TUI command to set the parameters of the poly-hexcore mesh
mesh/auto-mesh	TUI command to generate the volume mesh
mesh/modify/auto-node-move	TUI command to improve the Volume mesh
mesh/prepare-for-solve	TUI command to prepare the mesh for solving by deleting all the non necessary objects (e.g. deletes unused faces)

1.3.7.3 Physics Setup

The Physics setup will start by defining the solver parameters then will specify the turbulence model after which the physical description of the problem will be completed : first the material and operating conditions and then all the Boundary conditions on each name selection of the CAD.

The solver parameters used in our simulations are :

Type	Time	Velocity Formulation
Pressure Based	Steady	Absolute

The Operating Conditions are : Atmospheric pressure, no gravity.
The turbulence model is $k-\omega$ SST all the input parameters are left to their default values(see: Ansys documentation).
Then the material *air* is created, leaving all the default material property. It is then applied to the envelope.
Boundary condition will be applied to the different CAD name selection, the envelope and wheel name selection will get the relevant Boundary Conditions whilst all the rest of

the name selections, stored in the car Scheme list, will be assigned simple no-slip walls. The current set of Boundary conditions can be found in the table below :

Name selection	**Boundary type**	**Configuration**
Inlet	Velocity Inlet	Freestream speed, default 20 m/s
Outlet	Pressure Outlet	Gauge pressure 0
Wall-side / Wall-side	Slip Wall	Zero shear
Symmetry	Symmetry	Default symmetry condition
wall-ground	Moving Wall	Translational wall, speed aligned with flow direction
wheel-front	Rotating Wall	Angular speed and front wheel axis
wheel-rear	Rotating Wall	Angular speed and rear wheel axis
patch-front / patch-rear	Slip Wall	Zero shear
aero-front / aero-rear / aero-winglet / aero-diffuser / aero-sides / wishbones-cover / chassi	Wall	Stationary no-slip wall unless specifically overridden

The implementations of these different task is handled by the TUI functions easily but with one exception : the rotating walls BCs require the axis position which cannot be reliably computed in Fluent Solution nor in Fluent Meshing. These have to be extracted from the space claim. Since these values will not change except in the case of a rule variation the values will be stored as simulation parameters inside of Global and the user will have to insure that they are correct. Currently the values are measured at :

Wheel axis	**X [mm]**	**Y [mm]**	**Z [mm]**
Front	-1530	710.4	203.2
Rear	0	710.4	203.2

Main Steps of the Module

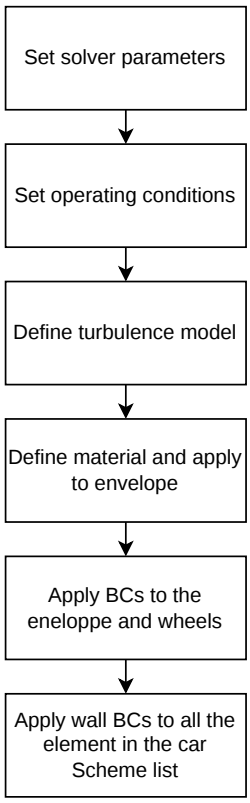


Figure 1.13: General structure of the Physics setup module

List of Relevant TUI/ Scheme Functions

Command	**Description**
define/models/solver/pressure-based	TUI command to set the solver as pressure-based
define/models/steady	TUI command setting the solver to steady time
define/models/solver/velocity-formulation absolute	TUI command setting the velocity formulation as absolute
define/models/viscous/kw-sst	TUI command to set the Turbulance model as $k-\omega \quad SST$
define/materials/change-create air	TUI command to import the predefined air material
define/operating-conditions operating-pressure	TUI command setting the operation pressure
define/boundary-conditions/set `<type> <zones>`	TUI command adding a Boundary Condition of a specific type for multiple zones
define/boundary-conditions/wall	Tu command setting wall BC (then specify static, translating, rotating)
define/boundary-conditions/velocity-inlet	TUI command to set velocity inlet BC
define/boundary-conditions/symmetry	TUI command to set symmetry BC
define/boundary-conditions/pressure-outlet	TUI command to set pressure outlet BC

1.3.7.4 Reports Definitions

The report will allow the user to verify the quality of the simulation. The current reports plot the following values :

- Default Fluent reports : Continuity, velocity components (u,v,w), k and omega
- Force reports on Cl, Cd over the whole car

Before plotting the different coefficient reference values have to be set for correct calculation : the free stream velocity and the frontal area of the car.
Additionally report file will be written for all the scalar quantities that will be extracted later on, this allow for averaging of the values for the last N iterations.

Main Steps of the Module

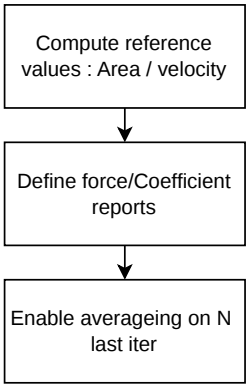


Figure 1.14: General structure of the Report definitions module

List of Relevant TUI/ Scheme Functions

Command	Description
report/reference-values/area	TUI command setting the reference area for computations
report/reference-values/velocity	TUI command setting the reference velocity for computations
solve/report-definitions/add	TUI command to add a report definition
solve/report-plots/add	TUI command creating a report plot
solve/report-files/add	TUI command creating a report file

1.3.7.5 Initialization and Run

Currently the team uses standard initialization for the CFD run and simply sets a high number of iterations and visually inspect the residual / force reports to check for convergence and manually stop the simulation.

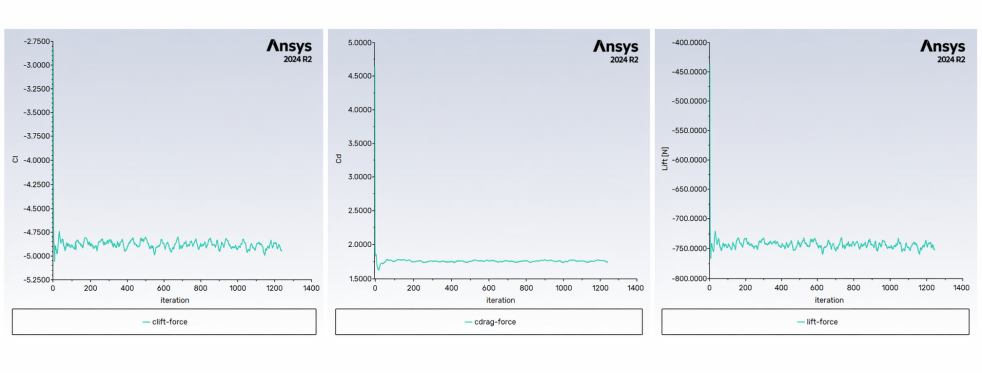


Figure 1.15: Example of force reports on a design CFD run

The automation should ideally use a custom convergence criterion based on the variation of the forces. However it is difficult to set a fixed criteria as theses oscillation are significant and can vary between design (for example a large stall will cause more oscillation). Additionally implementing this is non-trivial as it requires the condition to be checked at the end of each iteration, which isn't feasible with Scheme as when calling *solve/iterate* no Scheme command will be read until either the number of iteration is hit or the native Fluent convergence criterion are hit. To have a custom criterion that will be checked at each iteration a Scheme UDF is necessary. Due to this being the only time where a UDF is necessary this implementation will not be pursued at first. The other solution is to make multiple calls to *solve/iterate* with a low number of iteration for example in a Scheme while loop and extract the values of Cl / Cd at the end of each call and evaluate the criterion in Scheme. The issue will be an increase in solve time due to repeated calls.

An hybrid solution will therefore be applied : a number of iteration will be run as a single call to *solve/iterate* for a window where we know the simulation won't be converged. Then a while loop will iterate on small chunks, then the results from these chunks will be averaged in a sample. Then the convergence criterion will check that the variation between different sample averages are small. This method allows the criterion to ignore the oscillations of the solution, and to not slow the simulation too much. However it depends on a correct choice of chunk and sample sizes. A hard limit on the number of iteration will be put in place to handle non-convergent simulations.

Main Steps of the Module

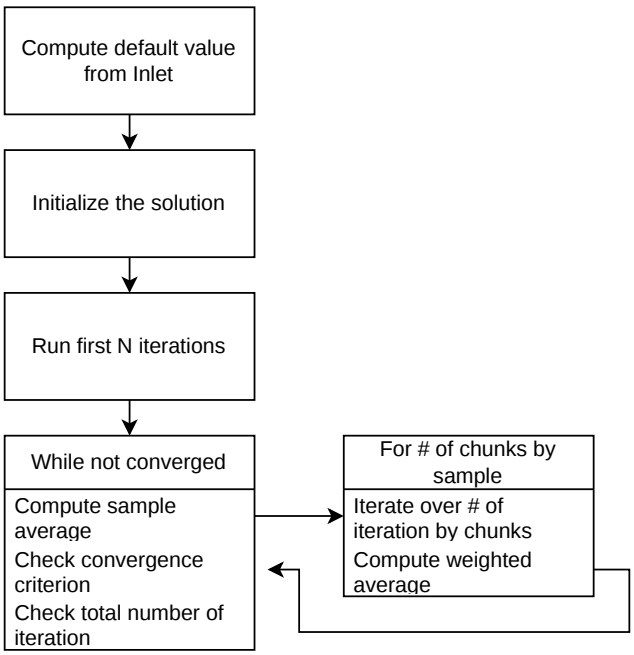


Figure 1.16: General structure of the initialize and run module

List of Relevant TUI/ Scheme Functions

Command	Description
solve/initialize/compute-defaults/zone	TUI command to compute default value for initialization from a specific zone (velocity-inlet in our case)
solve/initialize/initialize-flow	TUI command to initialize the flow at a current default value
solve/initialize/hyb-initialization	TUI command for hybrid initialization
solve/iterate	TUI command executing a fixed number of solve iterations

1.3.7.6 Post process

The post process module is responsible to compute the different scalar metrics from the converged simulation. However the fluent TUI / Scheme integration isn't tailored for this application and does not mimic all the GUI capabilities. A critical limitation for this project is that no TUI function exist to compute lift or drag coefficient.

To circumvent this limitation the only solution is to have report definition (or Monitors) for all the name selection and compute the values from these reports after the simulation has converged. This solution provides another advantage in the capability to compute values as averages over a certain number of iterations mitigating the effect of the oscillation that we observe currently.

The following values have to be computed for all the aero elements of the car and totaled onto the whole car :

- Downforce Cz [N]
- Drag [N]
- Lift coefficient Cl
- Drag coefficient Cd

Additionally for balance computation the Center Of Pressure over the length of the car (COPx) has to be extracted for balance computation. Once again a command exists in the GUI to get the COP but not in the TUI. Therefore the COPx will have to be computed as a moment report over a force report.

Note

There exists another possible solution to the issue of non TUI accessible GUI function is to use the cx Scheme macros. These macros are used to interact directly with the GUI from a scheme script (e.g. (cx-create-panel title apply-cb update-cb) will create a custom dialog box). However only the creation of new user interface element is documented, not how to interact with existing elements. The sole solution is to use the record Journal option then perform the GUI task and then edit the code that has been produced.

All the extracted values from this module will be stored as a Scheme variable that will be used for the export module.

Main Steps of the Module

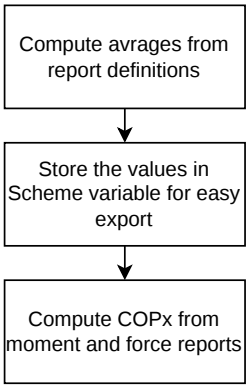


Figure 1.17: General structure of the Post process module

List of Relevant TUI/ Scheme Functions

Command	Description
solve/report-definitions/edit/ `<report-name>` average-over	TUI command to edit a report definition such that it averages over the last N iteration
report/definitions/compute	TUI command to update the report definition (necessary after the iterations
(rpgetvar 'report-definitions/ `<report-name>`)	Scheme command to get the value of a report definition (which is stored as an RP-variable)

1.3.7.7 Result exports

This section will be the interface between the scalar values previously extracted by the post process module and the cleanup code that will record them with the images extracted in Paraview. Therefor this data needs to be recorded in a way that is compatible with the python code. The best solution is to export the results in a CSV file which will be read and parsed by the python code. This will also allow the design team to easily use the data in a spread sheet. The export must impose a clear structure in the data to allow parsing by the pyhton code, hard coding the name selections is not possible as they will vary design by design

Main Steps of the Module

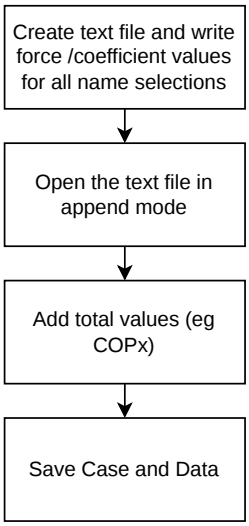


Figure 1.18: General structure of the Result Export module

List of Relevant TUI/ Scheme Functions

Command	Description
file/write-case-data	TUI command saving the case and data files of the simulation
(with-output-to-file "file.txt" (lambda () ...))	Scheme command to output the return of a lambda expressions (Scheme equivalent to anonymous function in Matlab) in a file
(let ((port (open-output-file "file.txt" "a"))) (display "New line" port) (newline port) (close-output-port port))	Scheme function definition that opens a file in append mode

1.4 Implementation of the Fluent Automation

Having established a complete plan for the Fluent automation it is now time to actually implement the code. The following section will describe what was achieved during the semester and mention the different challenges that were faced. Special attention will be paid to adjustment made that deviate from the technical specifications.
As the code was implemented mostly following it's execution order this section will base itself on this for it's structure. Additionally the file structure of the project can be a good reference point into the structure that will be described afterward :

```
fluent_automation/
├── main.jou          top-level flow-control journal
├── <model>.scdoc     CAD geometry
├── scheme/
│   ├── globals.scm   global RP variables & helpers
│   ├── cad_import.scm
│   ├── mesh.scm
│   ├── physics.scm
│   ├── reports.scm
│   ├── init_run.scm
│   ├── post.scm
│   └── export.scm
├── modules/
│   ├── 01_cad_import.jou
│   ├── 02_mesh.jou
│   ├── 03_physics.jou
│   ├── 04_reports.jou
│   ├── 05_init_run.jou
│   ├── 06_post.jou
│   └── 07_export.jou
└── output/           case/data files, post-processing artifacts
```

Figure 1.19: Project file structure for the Fluent automation pipeline.

Note that the following report will contain a high quantity of Scheme code. As explaining how Scheme works would be outside of the scope of this report we encourage the reader to take a look at the Scheme introduction found here [6]. When particularities of Scheme have heavily influenced the design choices relevant clarification will be made. To differentiate between native Scheme function and Fluent scheme function with those that are defined in the automation itself the prefix `ert-` will be used to flag our functions.
All of the complete code can be found in the annex 5.2 for the journal files and under

1.4.1 Main.jou

The main journal is the entry point of the pipeline automation. It can be launched through a loaded Fluent meshing session using the file read journal menu option, or it can be launched more directly via the fluent launcher by enabling the read journal option before starting fluent and finally it can be launched trough command in the terminal. For any of the aforementioned method the main file will regulate the execution of the pipeline.

In itself the content of the main is pretty straight forward it first loads the global Scheme utility file via `(load "scheme/globals.scm")` and then calls all the modules and handle flow control.
To handle flow control a function run-module is defined :

```
1  (define (run-module path label)
2  (format #t "~%====================  ~a  ====================~%" label)
3  (ti-menu-load-string (string-append "/file/read-journal " path))
4  (if (not (ert-status-ok?))
5      (begin
6        (format #t "~%main : aborting - module '~a' failed~%" label)
7        (exit))))
```

This function uses one of the utility defined in the global.scm `ert-status-ok?`, we will see it's implementation later but to shortly resume it simply check that no error flag where raised during execution. If an error has been raised the module fail text is printed in the consol and the run is stopped by calling `exit`.
Using this function each module can be ran like `(run-module "path" "name")`.

1.4.1.1 Specification deviation : suppression of the error log file

Even if we have only seen a single function it represents an important choice that had been made deviating from the specification. When trying to build the log file utility functions some bugs when appending content to a text file proved to be difficult to solve. Combined with the fact that after a journal file had been run Fluent will write a text file (saved as a .trn) containing all the contents from the command window another solution was selected. Instead of making a custom log file the automation code would use the .trn file, to better see the progress and the sanity check performed by the code itself the execution was made more verbose using the `format` function (Scheme equivalent to printf in C). This is especially important when loading a scheme file as it is normally silent in the consol.
Thus using Fluent own .trn file allowed to circumvent log file writing errors and keeps fluent native error messages.

1.4.2 globals.scm

Now let's move trough the actual automation process the first file called in main : global.scm . This file contains all the shared utilities and all the parameters used in the subsequent modules.

As all the simulation parameter are stored as RP variable to mitigate any scoping problem a convenient wrapper is made for the `rp-var-define` procedure.

```
(define (ert-rpvar-define name default type)
  (let ((full-name (string->symbol
                     (string-append "user/"
                                    (symbol->string name)))))
    (if (not (rp-var-object full-name))
        (rp-var-define full-name default type #f))
    (rpgetvar full-name)))
```

This adds 3 useful utilities : first the prefix "user/" is added to the variable name allowing to easily differentiate between fluent defined variables and manually defined variables. This is done by first converting the`name` symbol given to the function to a string then concatenate it with "user/" and re transform it into a symbol. Secondly the if condition checks that the name isn't already defined elsewhere. Finally the call to `rpgetvar` returns the current value (default) set to the variable which is used as a sanity check.

Note

The order of the string to symbol called can be confusing. This is a particularity of Scheme : it uses reverse polish notation. For example 3 + 4 - 5 would be written in Scheme as `(- (+ 3 4) 5)`.

The next part of the global file contains all the relevant variables that need to be defined and when necessary their default values. For reference all the values are listed in the table below :

Table 1.10: RP variables defined in `globals.scm` (all in the `user/` namespace).

Name (user/...)	Default	Type	Purpose
1–2. Paths			
`project-root`	`"."`	`string`	Absolute project root; resolved at load time.
`input-dir`	`"."`	`string`	Directory searched for the CAD file.
`output-dir`	`"output"`	`string`	Case/data and post-processing output directory.

continued on next page

Table 1.10 continued from previous page

Name (user/...)	Default	Type	Purpose
cad-file	""	string	Resolved CAD path (auto-discovered or user override).
3. Required name selections			
enclosure-zones	("inlet""outlet""symmetry""wall-side""wall-top""wall-ground")	list	Enclosure-wall zones that must exist in the CAD.
wheel-zones	("wheel-front" "wheel-rear")	list	Rotating-wheel zones.
patch-zones	("patch-front" "patch-rear")	list	Wheel contact-patch zones.
car-zones	()	list	Everything else (no-slip car walls); filled at import.
cad-object-name	""	string	Imported CAD object name; filled by cad_import.
4. CAD import options			
cad-length-unit	"mm"	string	Length unit assumed on import.
cad-tessellation	"cad-faceting"	string	Tessellation/faceting method.
cad-feature-angle	40.0	real	Feature-capture angle [deg].
5. Physics / operating conditions			
inlet-velocity	20.0	real	Free-stream inlet velocity [m/s].
op-pressure	101325.0	real	Operating pressure [Pa].
wheel-front-axis	(-1.530 0.7104 0.2032)	list	Front-wheel rotation axis point [m].
wheel-rear-axis	(0.000 0.7104 0.2032)	list	Rear-wheel rotation axis point [m].
6.1–6.2. Bodies of influence (BOIs)			
mesh-static-bois	5 absolute boxes (near/middle/far field, rwing, diffuser)	list	Static refinement boxes: name, bounds, sizing-mm.

continued on next page

Table 1.10 continued from previous page

Name (user/...)	Default	Type	Purpose
`mesh-wheel-bois`	`2 label-relative boxes (fw, rw)`	`list`	Wheel BOIs relative to label bbox; ratios + sizing.
6.4–6.5. Surface sizing & improve			
`mesh-local-sizing`	`3 groups (chassis, aero, wheel)`	`list`	Curvature sizing: growth, min/max, normal-angle, edges.
`mesh-surface-quality-limit`	`0.6`	`real`	Surface-mesh improve threshold.
6.6. Inflation layers (prisms)			
`mesh-prism-controls`	`2 controls (last-ratio_car, smooth-transition_ground)`	`list`	Scoped prism layers: scope, offset method, layers, ratio, first height.
6.7. Poly-hexcore volume mesh			
`mesh-hex-peel-layers`	`1`	`integer`	Hexcore peel layers.
`mesh-hex-min-cell-length-mm`	`1.0`	`real`	Minimum hex cell length [mm].
`mesh-hex-max-cell-length-mm`	`256.0`	`real`	Maximum hex cell length [mm].
`mesh-hex-buffer-layers`	`1`	`integer`	Hexcore buffer layers.
`mesh-poly-feature-angle-deg`	`30.0`	`real`	Poly feature angle [deg].
`mesh-hex-avoid-1to8-jump?`	`#f`	`boolean`	Avoid 1-to-8 transition jumps.
`mesh-hex-merge-bl-cells?`	`#t`	`boolean`	Merge boundary-layer cells.
`mesh-prox-gap-factor`	`0.25`	`real`	Proximity gap factor.
`mesh-prox-max-aspect-ratio`	`25.0`	`real`	Proximity max aspect ratio.
`mesh-adjacent-attach-angle`	`80.0`	`real`	Adjacent-face attach angle [deg].
`mesh-normal-smooth-relax`	`100.0`	`real`	Normal smoothing relaxation.
6.8. Volume mesh improve			
`mesh-volume-quality-limit`	`0.05`	`real`	Improve threshold (orthogonal quality).
`mesh-volume-quality-target`	`0.2`	`real`	Target quality (spec: orth ≥ 0.2).

continued on next page

Table 1.10 continued from previous page

Name (user/...)	Default	Type	Purpose
Status (flow control)			
`last-status`	`"OK"`	`string`	Last module outcome ("OK" / "FAILED"); drives `main.jou` aborts.
`last-module`	`"init"`	`string`	Name of the last module that ran.

To help the in case the user wants to overwrite any parameter an optional override file is then loaded after the default values are defined.

```
(if (file-exists? "config/run_config.scm")
(begin
  (load "config/run_config.scm")
  (format #t "globals : loaded user overrides from
  ↪  config/run_config.scm~%"))
(format #t "globals : no run_config.scm found, using defaults~%"))
```

This allows to simple use `(rpsetvar 'user/inlet-velocity 30.0)` in this file should we want to increase the inlet velocity. Therefore the user doesn't need to go look into globals to modify a value.

Finally a last group of function are defined in global, their use is to auto discover the .scdoc file to allow variation in the name of said file. As file management is not easy in Scheme this part of the code was generated using generative AI. As it has been working well so far the code was kept as generated.

1.4.3 Watertight geometry workflow

Before diving into the cad import module another change to the initial specifications needs to be discussed : switching from the legacy TUI command mesher to the more modern water tight geometry workflow.
Initially the CAD import was driven by the simple TUI command `/file/import/cad` and the mesh would also be written using TUI commands. This approach drives Fluent legacy mesher and not the currently used watertight geometry workflow. This was a non issue for the CAD import but was proving tricky when meshing as there is no one to one equivalent TUI command to the manual mesh steps. In particular while trying to define BOIs and failing a discovery was made : when using the write journal feature on a water thigh workflow instead of returning unusable CX commands the resulting journal had python command wrapped in scheme.
An example of the content would be (`\%py-exec "workflow.TaskObject['Import Geometry'].Execute()"`
This meant that the journal gave a template as to what was needed for meshing using the watertight geometry.

Here the concept of "metaprogramming" common in Scheme becomes incredibly useful. In a very simplified way it corresponds to Scheme capability of having code that

writes codes. In practice this was used by building the correct python calls with the parameters defined in globals as strings and then execute those custom commands in the Fluent scheme interpreter. This is a byproduct of the fact that scheme doesn't differentiate between strings and code when passed in the interpreter.

1.4.4 CAD import module

As per the specification the journal first calls the corresponding scheme file to get the relevant functions. However since the python wrapping is fully scheme command and no TUI is used the separation is a bit different for this module. The journal file contains all the commands to import the file and then the scheme function `ert-cad-process` activates all the function in `cad_import.scm` which process the name selection parsing them and verifying the existence of the required ones.
During the cad import process the workflow is initialized :

```
1
2 (ti-menu-load-string
3   "(%py-exec \"workflow.InitializeWorkflow(WorkflowType='Watertight
    ↪  Geometry')\")")
```

The rest of the implementation is relatively straightforward although hard to read due to Scheme syntax, or more correctly lack of syntax. The full codes are available in the annex.

1.4.5 Meshing

Meshing was the most critical and difficult module to implement. As the final implementation went the journal file only had two active line to load and call the scheme implementation of meshing.
Since a more complex python usage would be needed a few helper are made to bridge the gap between scheme and python

Formatter	Example output	What it does
`ert-mesh-py-string`	`r'wall-ground'`	Wraps a Scheme string as a Python *raw* string literal (`r'...'`), so backslashes survive the Scheme → Python boundary.
`ert-mesh-py-num`	`4  0.272`	Emits a number as a Python numeric literal: integers stay integers, everything else is coerced to a float via `exact->inexact`.
`ert-mesh-py-bool`	`True / False`	Converts a Scheme boolean (`#t`/`#f`) to the Python keywords `True`/`False`.
`ert-mesh-py-list-of-strings`	`['a', 'b']`	Builds a Python list literal of raw-string elements by mapping each item through `ert-mesh-py-string` and joining with commas.

Table 1.11: Python formatters defined in the meshing module scheme file.

These are mostly necessary as python has a syntax and scheme does not (in a broad way of speaking). Then a specific wrapper is made to contain the

Chapter 2

2D

2.1 Introduction

2.1.1 Context

Formula Student is an international engineering competition in which student teams design, build, and race a small formula-style car. Aerodynamic performance plays an increasingly important role at the highest levels of the competition, where the difference between teams is often determined by the efficiency of the aerodynamic package. The rear wing is the largest and most influential aerodynamic device on the car. It operates at the downstream end of the vehicle, in a complex flow environment shaped by every upstream component. Maximising its downforce while controlling drag is therefore a central objective of the aerodynamics division. At EPFL Racing Team, the aerodynamic development process relies primarily on full 3D CFD simulations of the car. These simulations are accurate and capture the complete interaction between all aerodynamic surfaces, but they are expensive in terms of setup time and computational cost. This limits the number of design iterations that can realistically be completed within the timeframe of a student project. This report presents the development and application of a 2D CFD workflow specifically designed to address this bottleneck. The workflow focuses on the cross-sectional aerodynamics of the four-element rear wing and serves as the primary tool for geometric optimisation.

2.1.2 Objectives

The 2D CFD workflow serves two complementary purposes within the EPFL Racing Team aerodynamics division. It was originally developed as a personal tool to build a deeper understanding of the flow behaviour around a multi-element wing configuration, and to support the 3D design work presented later in Chapter 3.

Before being able to optimise anything, it was necessary to develop a solid intuition of how different airfoil parameters influence the overall aerodynamic behaviour of the assembly. The 2D workflow provided exactly that foundation. Beyond this initial personal use, it has naturally grown into a resource that can benefit future members of the association, giving them the same structured entry point into multi-element aerodynamics.

From an academic standpoint, it allows direct visualisation of the flow physics around a multi-element wing through velocity contours and pressure coefficient distributions. This kind of hands-on experience builds physical intuition that is very difficult to acquire from theory alone. It also provides a structured introduction to ANSYS Fluent, the professional CFD environment already used by the team for 3D simulations, making this workflow an effective training and familiarisation tool.

From an engineering standpoint, reducing the turnaround time of aerodynamic iterations from several days to only a few hours considerably enhances the development capabilities achievable within the framework of a student Formula Student project. The workflow does not reduce the number of iterations that aerodynamic optimisation inherently requires, it reduces the time cost of each one. The geometry identified as optimal through the 2D study is then transferred to the 3D model for final validation, ensuring that the time saved in exploration does not come at the cost of result quality.

Research Question

2D 2D

2.2 Multi-Element Wing Theory

Before going deeper into the model presented above, it is essential to recall some fundamental aerodynamic principles. Aerodynamics is the study of fluid dynamics around solid bodies. In our case, we focus on wings, which are the primary aerodynamic devices used in motorsport to generate additional vertical load on the tires when cornering through their interaction with the airflow, a phenomenon known as downforce. We will therefore go through key principles of wing theory, based on both lift and downforce, in order to better understand the design decisions.

2.2.1 Airfoil terminology

An airfoil is the two-dimensional cross-sectional shape of a wing, designed to generate an aerodynamic force when placed in a moving fluid. The main geometric parameters that define an airfoil profile are illustrated in Figure 2.1. The chord line is the straight line drawn from the leading edge to the trailing edge, c. The mean camber line runs midway between the upper and lower surfaces and describes the overall curvature, or camber, of the profile. The maximum camber is the greatest perpendicular distance between the mean camber line and the chord line, and the maximum thickness is the largest distance between the upper and lower surfaces; both are conventionally expressed as a percentage of the chord length and located at a specific chordwise position. Together, these parameters fully characterise the shape of a profil [7].

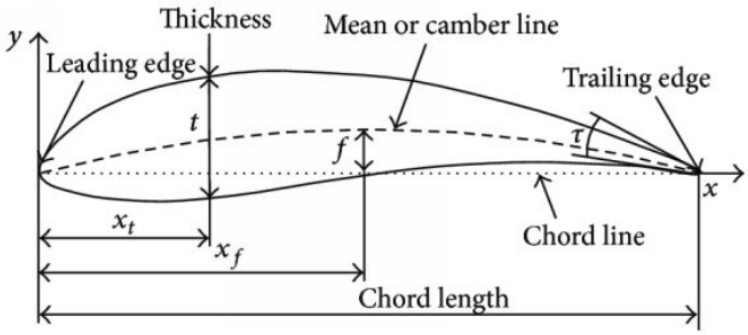


Figure 2.1: Airfoil geometry parameters, including chord line, mean camber line, maximum camber, and maximum thickness. Taken from Liang et al. [8].

The angle of attack (AoA), α, is defined as the angle between the chord line and the incoming freestream velocity vector V_∞, as illustrated in Figure 2.2. It is the primary parameter through which aerodynamic loading is controlled: increasing α generally raises the lift produced. For our case, we will take an inverted airfoil, and it will do the opposite: increase downforce.

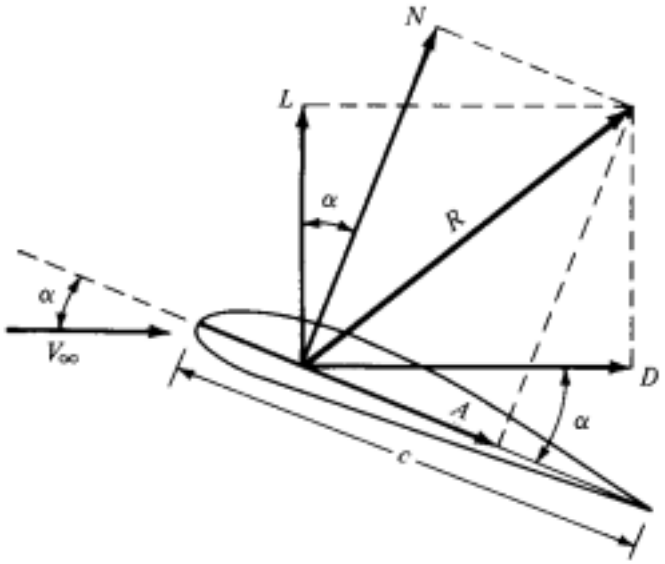


Figure 2.2: Definition of the angle of attack, α, between the chord line and the freestream velocity vector, V_∞. Taken from Anderson [7].

2.2.2 Aerodynamic Behaviour of a Single Element airfoil

When air flows around an object such as a wing, the fluid particles are forced to adapt to the geometry of the surface, modifying the velocity and pressure fields around it. Due to viscosity, the air in immediate contact with the wing adheres to the surface, satisfying the no-slip condition and forming a thin region called the boundary layer. Within this layer, strong velocity gradients develop between the stationary fluid at the wall and the outer flow (see Fig. 2.3). As the flow progresses along the wing, this boundary layer grows and its behavior depends on the balance between stabilizing viscous effects and destabilizing inertial effects. For sufficiently strong viscous influence, the flow remains smooth and ordered, known as laminar flow. As inertial effects become dominant, disturbances within the boundary layer amplify, eventually leading to a chaotic and highly mixed turbulent flow. This transition directly impacts airfoil performance by increasing drag and influencing flow separation over the surface.

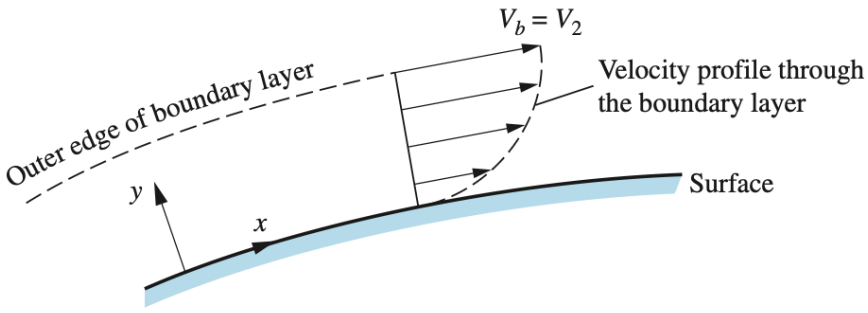


Figure 2.3: Boundary layer structure taken from []

Away from the surface, where viscous effects are negligible, the flow is governed by Bernoulli's equation, fig2.4. This relation, which applies under the assumptions of steady, incompressible, and inviscid flow along a streamline, states that an increase in flow velocity corresponds to a decrease in static pressure. It provides a useful description of the pressure distribution around the wing.

$$p + \frac{1}{2}\rho V^2 + \rho g z = \text{constant}$$

Figure 2.4: Bernoulli equation for steady, incompressible and inviscid flow

The origin of the aerodynamic force can be understood through two complementary perspectives.

From a momentum standpoint, the wing deflects the incoming flow downward; by Newton's third law, the reaction force acting on the airfoil is directed upward as shown in fig2.5. In our case, we are aiming to generate upwash ,instead, by using reversed airfoils.

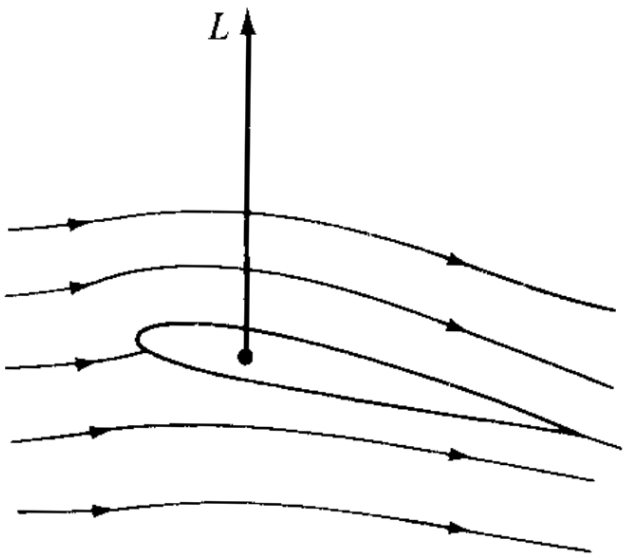


Figure 2.5: Downward deflexion of the airflow called downwash. Taken from [9]

From a pressure standpoint, the continuity equation requires the flow to accelerate where streamlines are compressed as occurs over the curved upper surface. By Bernoulli's principle, this acceleration corresponds to a lower static pressure, in green on the fig2.6, on the upper surface relative to the lower surface, in red, and the resulting pressure difference produces a net upward force called lift.

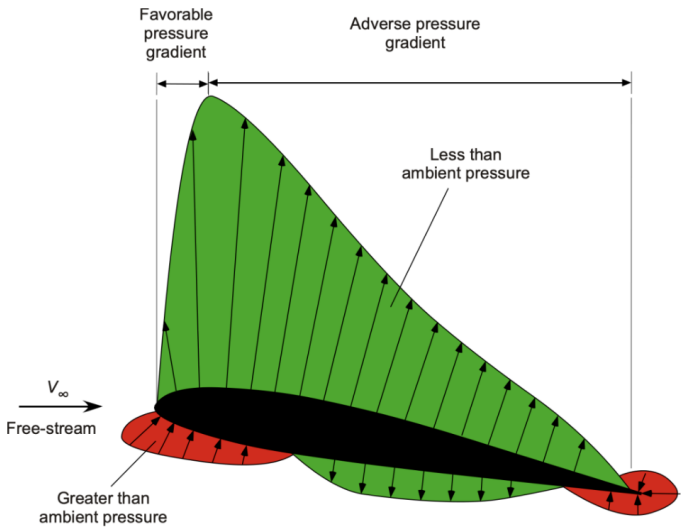


Figure 2.6: Pressure gradient around an airfoil. Taken from [10]

For the flow to generate lift efficiently, it must remain attached to the surface along the entire chord. As shown in fig2.7, the boundary layer is subjected to an adverse pressure gradient, where the pressure increases in the direction of the flow. At a certain location, referred to as the separation point, this pressure gradient becomes sufficiently strong that the boundary layer loses momentum. As a result, the near-wall flow can no longer overcome the adverse pressure gradient, leading to flow recirculation and separation from the surface.Hence, an airfoil that crosses this limit is no longer efficient and may even begin to degrade the airflow.

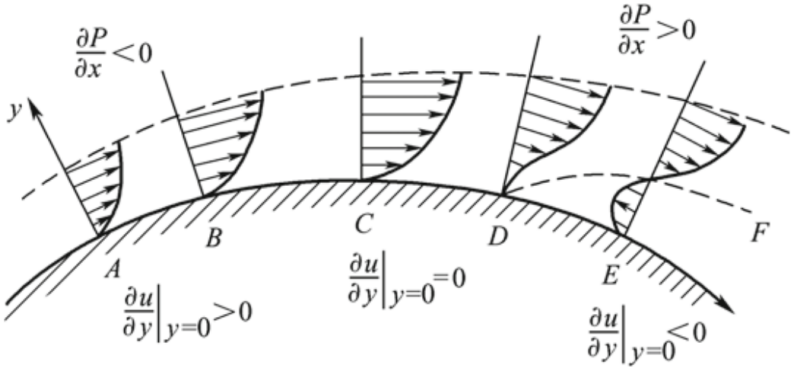
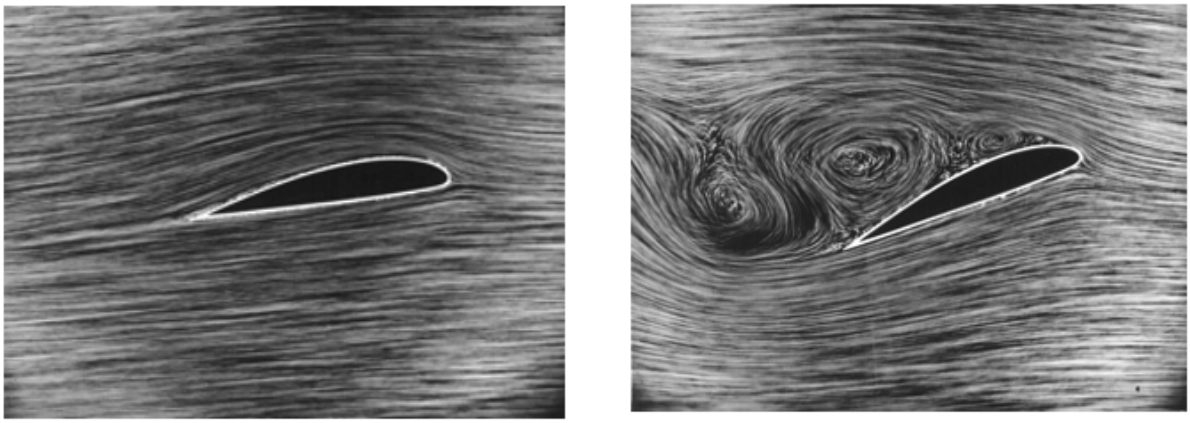


Figure 2.7: Adverse pressure gradient across the boundary layer taken from [11]

2.2.3 Limitations of a Single Wing

The lift force generated by a wing is not solely determined by its geometry, it also depends on its orientation relative to the incoming flow controlled by the AoA. Increasing this angle

causes the flow to be deflected more strongly downward, amplifying both the reaction force and the pressure asymmetry between the upper and lower surfaces. As a result, the lift coefficient increases approximately linearly with AoA, up to a critical value beyond which the boundary layer separates and the wing stalls as shown in figure X.



(a) Attached flow over the airfoil.

(b) Separated flow at high angle of attack.

Figure 2.8: Limitation of a single-element airfoil: attached flow at moderate angle of attack, flow separation at high angle of attack. Taken from [12]

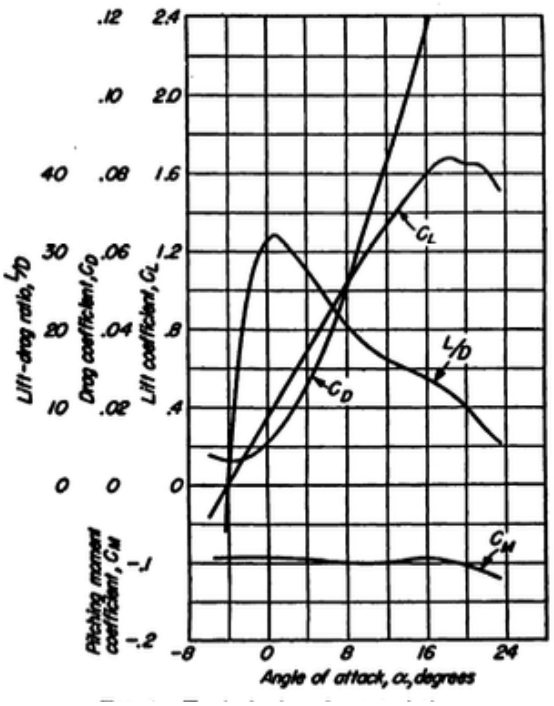


Figure 2.9: Lift coefficient variation with angle of attack taken from Abbott [13]

In figure 2.9 we can see the difference between an attached and stalled flow, and the fig **??**, shows the limitation of a single element configuration. We can see that the useful operation window of the AoA of the airfoil in question is from$-7°$ to $23°$ while the angle that maximize C_L is approximately $17°$. (fig **??** is not related to fig 2.8a and 2.8b)

The other geometric parameter that influences lift generation is airfoil's camber previously defined in section 2.2.1). The cambered airfoil's inherent asymmetry produces a pressure difference between the two upper and lower surfaces and therefore generates lift without requiring any inclination of the wing. As shown in 2.10, the lift coefficient curve of a cambered airfoil is shifted upward compared to that of a symmetric profile, reflecting this offset in lift at zero angle of attack. Increasing the camber amplifies this effect, raising the lift coefficient for a given angle of attack.

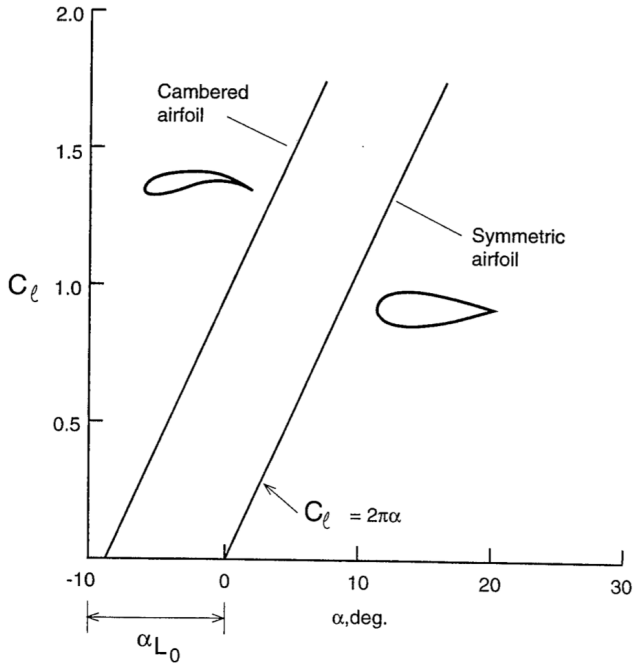


Figure 2.10: Graph that shows a comparison for Cl over AoA between a symmetric profle and a cambered one. Taken from [14]

As shown in the graphs above, increasing the angle of attack or the camber of a single-element wing raises the lift coefficient, but only up to a critical point. Beyond this threshold, the adverse pressure gradient near the trailing edge becomes too strong for the boundary layer to remain attached. The flow separates from the surface and the wing stalls.

This limitation becomes even more pronounced with high camber or a long chord. A more cambered profile forces the flow to accelerate over a longer and more curved suction surface, intensifying the adverse pressure gradient. Similarly, a longer chord gives the boundary layer more distance over which to accumulate momentum losses, making separation more likely before the trailing edge is reached. In both cases, the single-element wing hits a fundamental aerodynamic ceiling beyond which further geometric changes no longer improve performance.

2.2.4 Theory of a Multi-Element Wing

A multi-element wing addresses the fundamental limitations of a single-element configuration by dividing the total chord into several smaller airfoils placed in sequence.

Each element operates with a reduced effective camber and a shorter chord, which individually limits the strength of the adverse pressure gradient and reduces the distance over which the boundary layer accumulates momentum losses. As a result, each element can generate significant lift while maintaining attached flow along its surface.

The interaction between consecutive elements is governed by the the gap formed between the trailing edge of one element and the leading edge of the next as shown in fig2.11. This geometry is characterized by two parameters, the slot gap and the overlap, whose influence on aerodynamic performance will be discussed in detail later. From a fluid mechanics perspective, the slot acts as a converging nozzle and a localised source of momentum. High-energy fluid from the pressure side of the upstream element accelerates through the gap and is injected into the boundary layer developing on the suction side of the downstream element. This re-energizes the boundary layer, which has already lost momentum travelling over the upstream surface, and allows it to withstand a stronger adverse pressure gradient without separating.

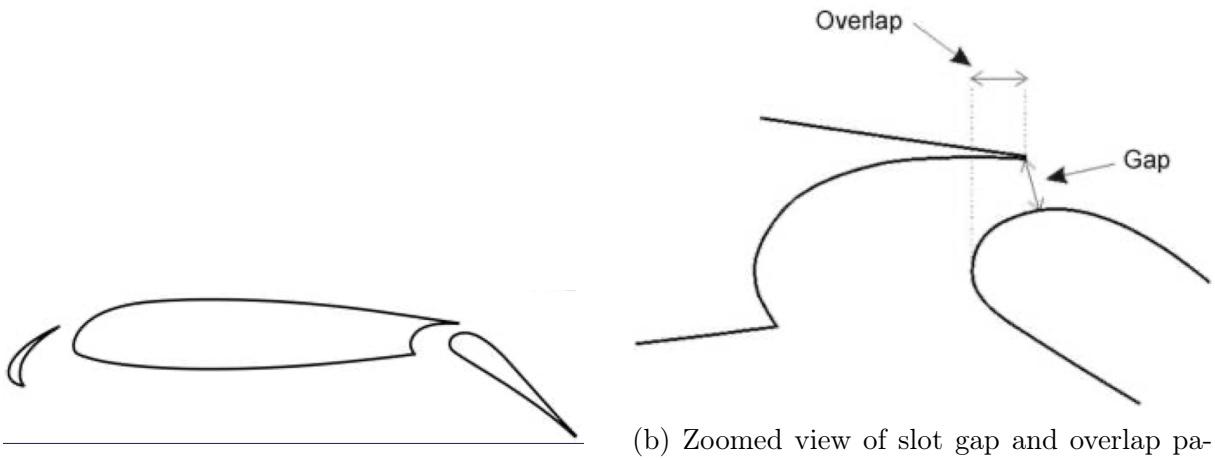
The combined effect of these two mechanisms allows a multi-element wing to achieve lift coefficients far beyond what a single-element wing of equivalent total chord could produce since it is composed of higher global camber while keeping flow attached.

2.2.5 Parameters of a Multi-Element Wing

The performance of a multi-element wing is highly sensitive to its geometry. The following sections present the key parameters that must be carefully considered when designing such a configuration, and whose understanding forms the very foundation of the study conducted in this report.

2.2.5.1 Slot Gap and Overlap

The geometry of the slot is defined by two parameters, illustrated in fug2.11. The slot gap is the perpendicular distance between the trailing edge of the upstream element and the leading edge of the downstream one. The overlap is the chordwise distance by which the leading edge of the downstream element is recessed behind the trailing edge of the upstream element. Together, these two parameters directly control the characteristics of the slot jet and are among the most influential variables in multi-element wing design.



(a) Global view of a multi-element wing

(b) Zoomed view of slot gap and overlap parameters

Figure 2.11: Overview of a three-stage multi-element wing configuration taken from [15]

The slot gap controls the trade-off between jet velocity and mass flow between two successive elements. A narrow gap generates a high-velocity jet but limits the mass flow, whereas a wider gap increases the mass flow at lower velocity.

The overlap determines the direction of the slot jet onto the downstream element and therefore influences the ability of the flow to re-energize the boundary layer and delay separation. The combination of slot gap and overlap must consequently provide sufficient jet momentum while maintaining an appropriate jet direction.

2.2.5.2 Element Loading Distribution

In a multi-element configuration, the aerodynamic loading is distributed across several elements rather than concentrated on a single surface. The angle of attack generally increases from the main element to the rearmost flap, allowing a progressive pressure recovery while limiting the stall tendency of each element.

The chord length is typically reduced toward the downstream elements in order to limit boundary layer development under increasingly adverse pressure gradients.

2.2.5.3 Global Camber and Alignment

Increasing the global camber increases therefore downforce generation but also strengthens the adverse pressure gradient on the rearmost element, bringing it closer to stall. The global geometry must therefore balance aerodynamic loading and flow attachment margin.

Finally, the elements must remain properly aligned so that the slot jet is smoothly guided onto the downstream surface without excessive flow mismatch or local incidence as shown in fig2.12.

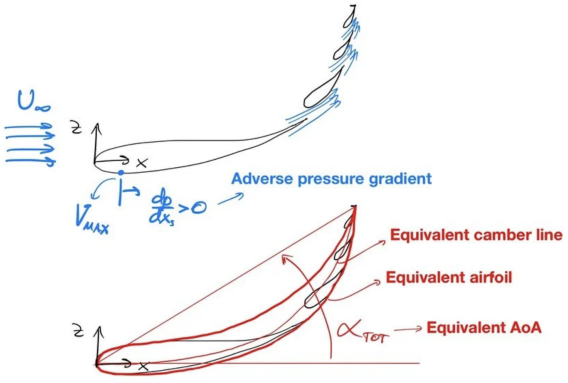


Figure 2.12: Multi element wings comparison between 1 stage and 4 stages with fixed chord and fixed LE and TE.Taken from [16]

2.2.6 Performance benefits

The payoff is visible directly in the surface loading. Figure 2.13 compares the chordwise pressure on a single element with that on a double-element wing. The single element reaches a moderate suction peak that recovers steadily toward the trailing edge. The double-element configurations reach a markedly stronger suction peak on the main element, and the jump near $x/c \approx 0.57$ marks the start of the flap, which carries its own loaded surface behind the slot. The wing therefore gains both deeper suction and a second loaded element, and the total load, the area enclosed by the curve, is correspondingly larger.

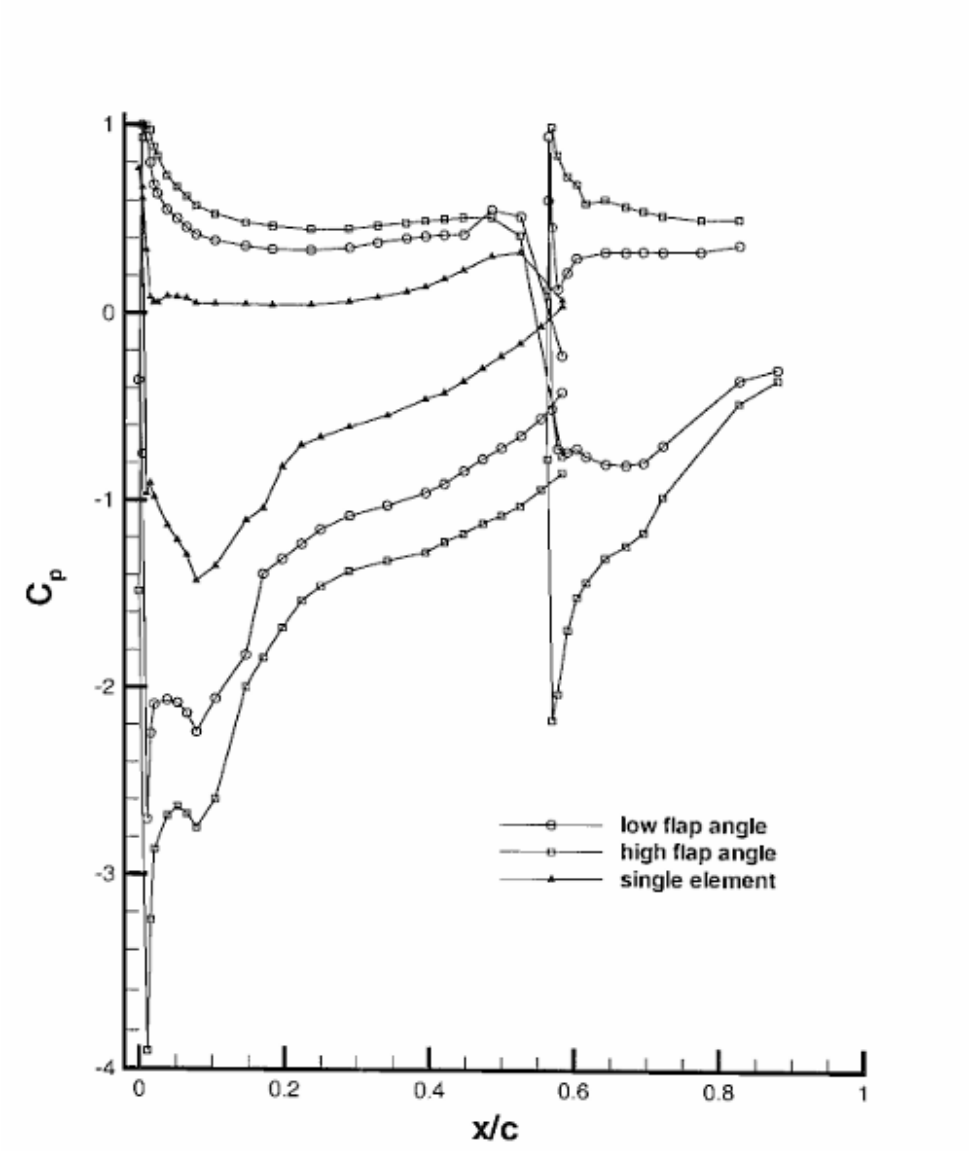


Figure 2.13: Chordwise surface pressures at the wing centre for low-flap, high-flap (both double-element) and single-element wings in freestream. Reproduced from [17], Figure 38(b), p. 119.

These benefits are not automatic. They depend on the parameters described above for each element, and these parameters will be optimized later in the study to achieve the best possible configuration for our case.

2.3 Description of the Workflow and Software Used

With the aerodynamic foundations established in the previous chapter, this section describes the numerical workflow developed to carry out the optimisation study. It focuses on the main choices related to the numerical approach, the simulation workflow, and the software tools employed to analyse the aerodynamic behaviour of the multi-element configurations.

2.3.1 Simulation approach

The first methodological choice concerns the simulation approach. Two main options were considered: a panel code solver and a Reynolds-Averaged Navier–Stokes (RANS) model. Given the complexity of the geometry and the expected flow conditions, this choice directly affects both the reliability of the results and the flexibility of the workflow.

Panel code solvers such as XFOIL or Javafoil provide a fast and computationally inexpensive framework, making them well suited for preliminary airfoil design. However, their formulation is based on potential flow theory, which assumes inviscid and fully attached flow. As a result, they cannot accurately capture boundary layer separation, a key phenomenon for highly cambered multi-element wings operating near their aerodynamic limits. In addition, these solvers are primarily intended for single-element configurations, and extending them to multiple elements with varying airfoil shapes and slot geometries considerably reduces their robustness and flexibility for a parametric optimisation study.

A Reynold Average Navier Stockes (RANS) based approach overcomes these limitations by solving the time-averaged Navier–Stokes equations together with a turbulence model. This allows the simulation to account for viscous effects, boundary layer development and flow separation governing multi-element wing performance. Although more computationally expensive than panel methods, 2D RANS simulations remain sufficiently efficient for iterative use within the optimisation workflow developed for this study. This approach also ensures consistency with the simulation methods already employed within the aerodynamics division of the EPFL Racing Team for higher-fidelity 3D validation.

For these reasons, a 2D RANS model was selected as the simulation framework for this study. It provides the physical fidelity required to meaningfully evaluate the influence of geometric parameters on the flow behaviour while remaining computationally accessible for iterative design work throughout the time constraint imposed by Formula Student. A more detailed description of the model will be introduced later.

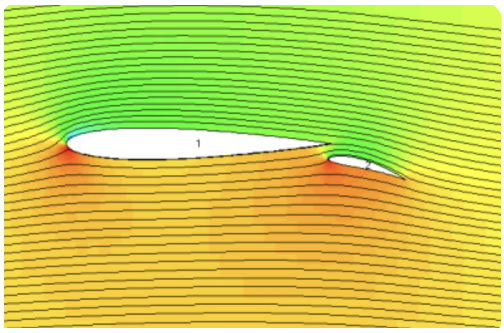


Figure 2.14: Static pressure contour via a panel code solver. Taken from [18]

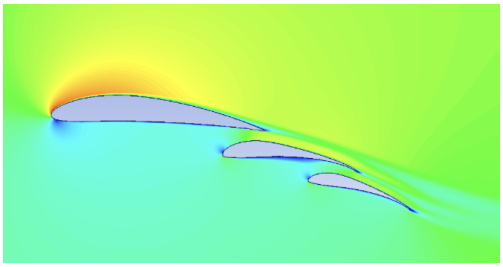


Figure 2.15: Static pressure contour extracted from the RANS solver in Ansys fluent.

2.3.2 Workflow Overview

Taking into consideration the simulation framework selected, a structured workflow was developed to generate the desired geometries and extract the results of interest for each configuration. This section briefly presents the overall architecture of the workflow, while the details of each individual step are covered in the following sections. The core idea behind this workflow is to operate entirely with coordinates in two dimensional space. A standardised set of coordinates was therefore generated for each airfoil studied using Airfoil Tools, providing a uniform coordinate format that is subsequently rescaled to the correct dimensions downstream in the process.

The general architecture of the workflow is illustrated by the block diagram below, where all the connections between each step are shown.

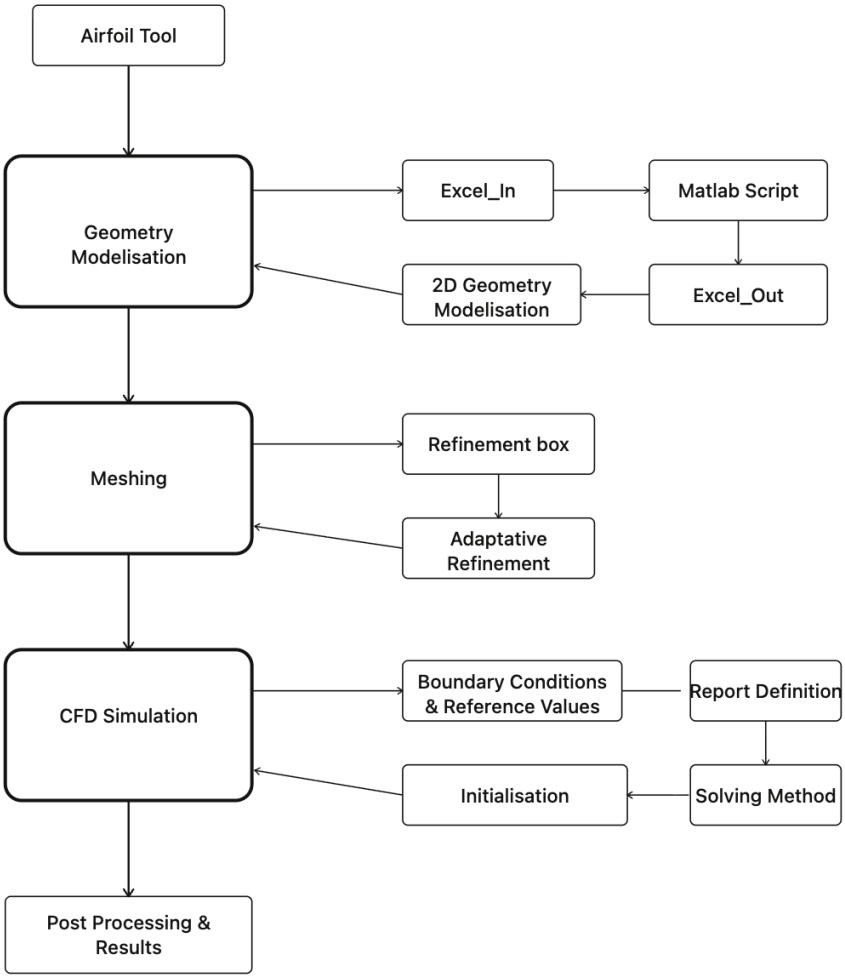


Figure 2.16: Structure diagram of the global workflow

2.3.3 Geometry generation

The first step of the workflow consists in generating the geometry of the multi-element wing. Two approaches were considered: a CAD-based method and a coordinate-based

method. Since the study relies on an iterative process requiring a large number of configurations to be tested, the choice was made in favour of the coordinate-based approach, which offers greater flexibility and can be fully automated.

The implemented solution takes the geometric parameters of interest — such as chord lengths, angles of attack, slot gap, and overlap — as input values, and outputs a fully assembled, translated, and rotated wing geometry, correctly positioned in the computational domain. This automation is the key enabler of the iterative workflow.

MATLAB was selected to handle this task, as it offers straightforward data import and export capabilities, as well as convenient visualisation tools that allow the generated geometry to be checked before being passed to the next step of the workflow.

2.3.3.1 Matlab Code overview

The MATLAB script reads normalised airfoil profiles from a CSV file. Each profile is defined by coordinate sets in Selig format, normalised by chord length. The script then scales each geometry to its target chord, specified as an input parameter, by applying a uniform scaling to the coordinates.

The trailing edge of each element is subsequently adjusted to a target thickness of 1 mm, consistent with the 3D model used by the aerodynamics division of the EPFL Racing Team. This step is necessary because database airfoil profiles typically exhibit very sharp or near-zero trailing edges, which are difficult to mesh reliably and may adversely affect wake resolution and the prediction of separation phenomena. In addition, the script provides a visual comparison between the imported and modified airfoils, allowing a clear verification of the geometric changes introduced by the processing step prior to their placement in the spatial configuration. These results are further illustrated in the accompanying figures.

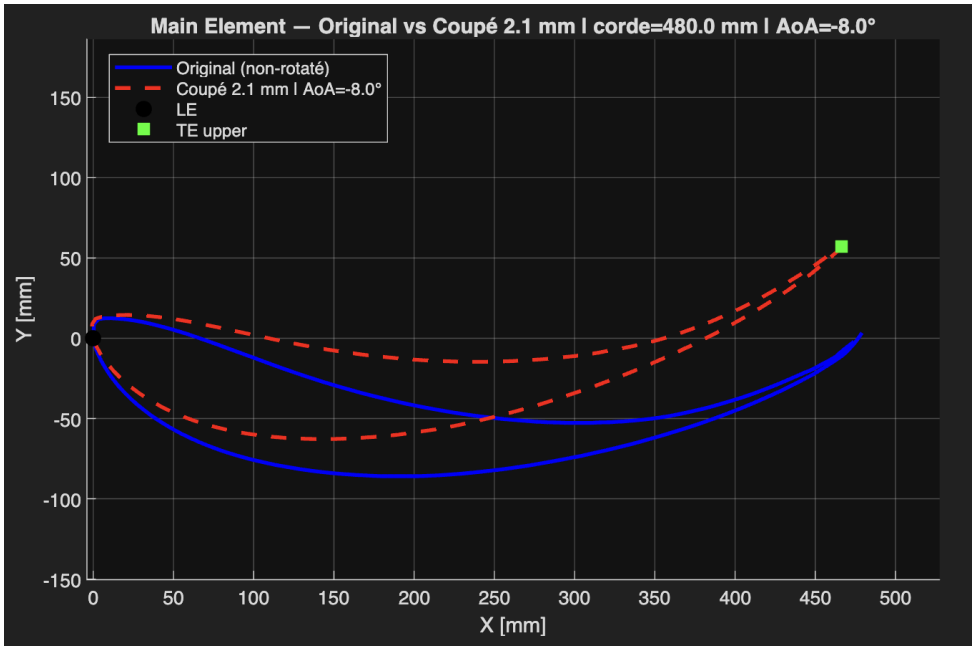


Figure 2.17: Imported VS Modified airfoil in matlab

Once the geometry is defined, each airfoil is rotated to its prescribed angle of attack using a dedicated transformation function, taking into account the relative positioning between elements defined in the configuration inputs. As illustrated in Figure X, beyond the fixed assembly parameters of the multi-element configuration, several construction parameters remain adjustable to ensure full control over the final geometry. To maintain consistent comparisons between iterations, the rotation pivot can be defined anywhere along the chord line, from leading to trailing edge, in accordance with design constraints. Full implementation details are provided in Appendix X.

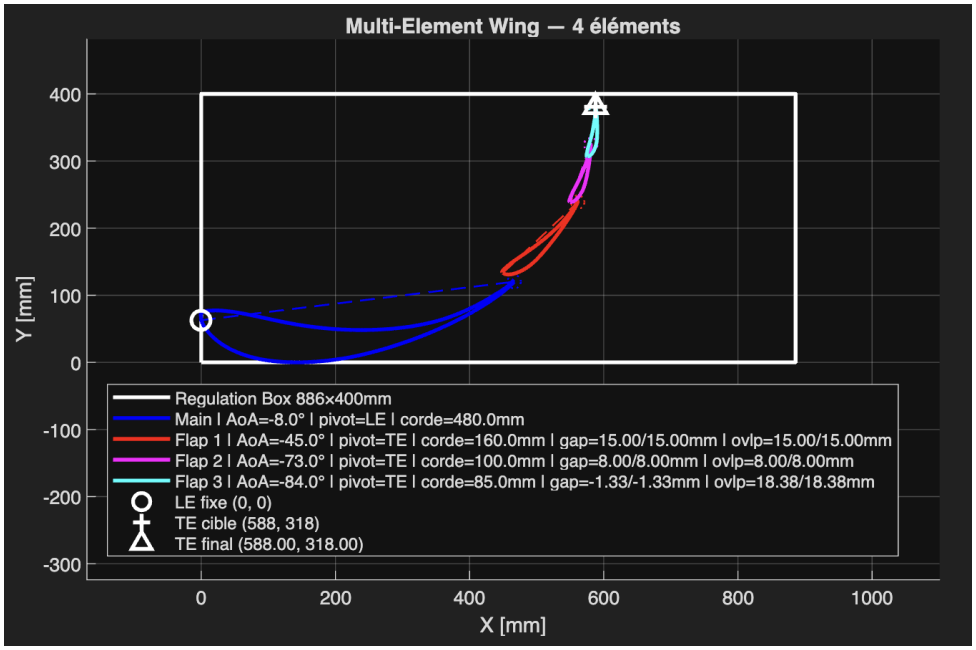


Figure 2.18: Final figure produced by the Matlab script before coordinates export

The script output can be inspected in both tabular and graphical form. The final coordinates are exported in the format required by the downstream geometry software via an Excel file. In parallel, a visual representation of the resulting configuration is generated, enabling comparison across iterations together with a summary of the input parameters. This visual output also includes the regulatory envelope defining the design space constraints. This aspect is further detailed in the section dedicated to the 3D design workflow.

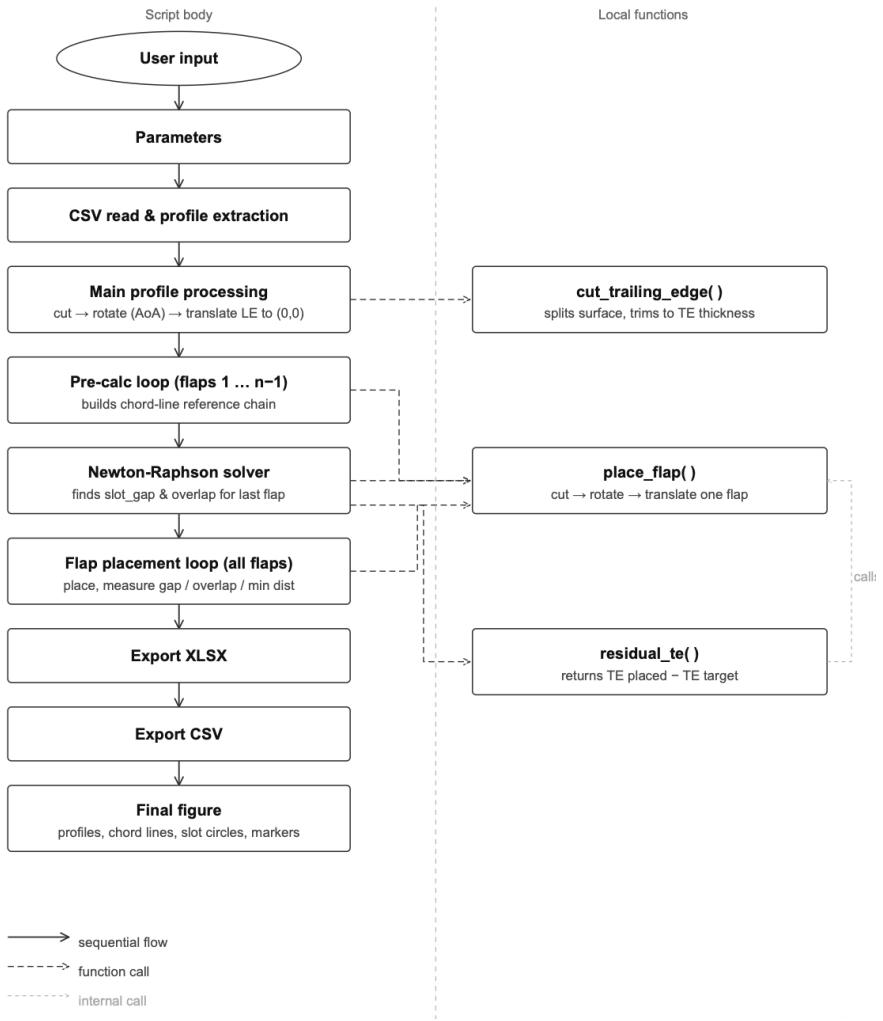


Figure 2.19: Matlab's structure diagram

After these steps, the data exported in the Excel file is transferred to the geometry creation workflow implemented in Ansys Workbench, enabling consistent control across all upstream processes up to the final results. This stage serves as the interface between the airfoil preprocessing and the subsequent geometry construction phase, where the curves and design enclosure are defined and assembled into the complete multi-element configuration.

2.3.3.2 Geometry and Enclosure creation

The transition from discrete coordinate points to continuous geometric curves is performed using DesignModeler within the Ansys Workbench environment. This step also enables the construction of the computational domain, which represents the fluid region of the simulation. More generally, it is used to define the various bounding volumes that structure the study domain, including the outer enclosure and the local refinement regions required for mesh control in subsequent steps.

The computational domain is defined as a rectangular region surrounding the wing assembly. It must extend sufficiently in all directions to ensure that the boundary conditions do not influence the solution in the vicinity of the wing. The dimensions, shown in the

table2.1, are taken from the aerodynamics team and satisfy the general requirements for an adequate domain size, ensuring the accuracy of the results. Therefore they will be used for the rest of the analysis as our air domain.
The bodies of influence, used later as mesh refinement regions in the meshing study, are also generated using this tool. As with the computational enclosure, their dimensions were not the primary focus of this study and are taken from established data provided by the aerodynamics team. They are illustrated in fig2.20. Nevertheless, it is relevant to mention them here, as a mesh convergence study will be carried out later in the workflow to ensure the accuracy and independence of the numerical results.

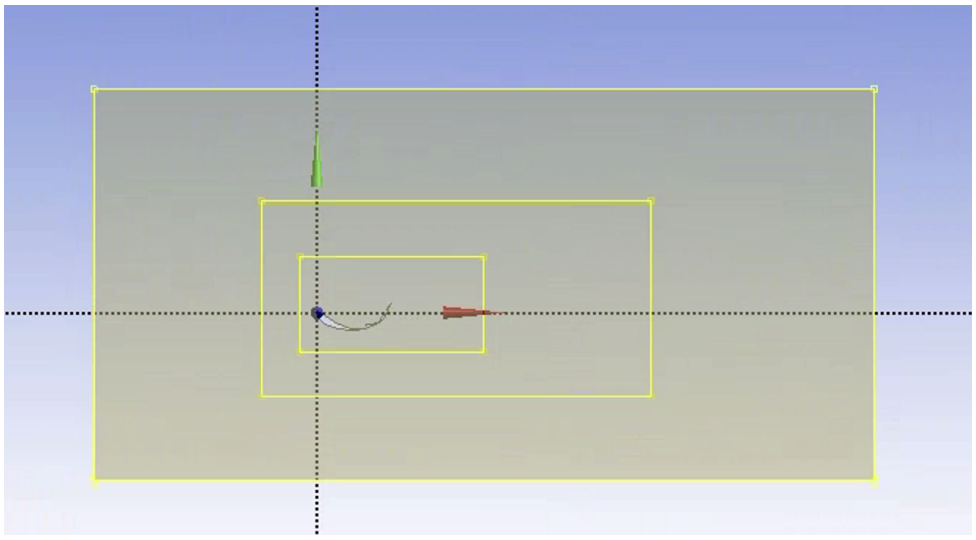


Figure 2.20: Box of refinement

Box	**Front**	**Top / Bottom**	**Behind**
Outer envelope	1c	1.5c	10c
Medium refinement	1c	2c	5c
Inner refinement	1c	1c	2c

Table 2.1: Recaptilative table of refinement box dimensions (in chord lengths)

2.3.3.3 Rotation Convention adaptation

Since this study is conducted in a 2D framework, it is important to acknowledge that it cannot fully reproduce the physics of a three-dimensional configuration. In particular, a correction related to the incoming flow direction is required to account for the differences between finite-span (3D) and infinite-span (2D) wing assumptions. The detailed implications of this discrepancy will be discussed in the later section comparing 2D and 3D results. The corresponding correction is implemented at this stage by rotating the entire multi-element wing assembly relative to the incoming flow direction as shown in fig2.21.

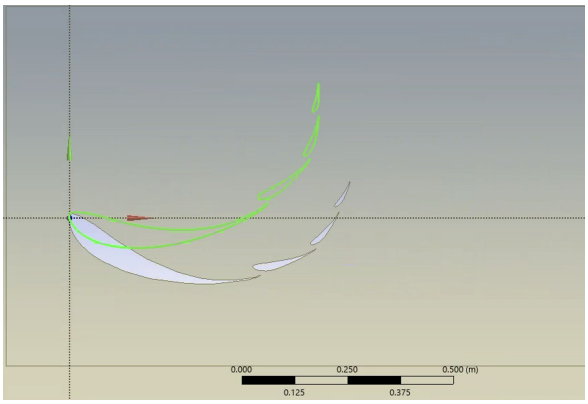


Figure 2.21: Initial Geometry (green) VS Rotated geometry (white)

2.3.4 Mesh

Solving the RANS equations requires the computational domain to be discretised into a finite number of small cells, forming what is called a mesh. The governing equations are solved numerically at each cell, meaning that the accuracy of the solution is directly tied to the quality and resolution of the mesh. Regions of high flow gradients, such as the boundary layer, the wake, and the areas around the leading and trailing edges require a finer mesh to capture the physical phenomena accurately, while regions far from the wing where the flow is nearly uniform can be meshed more coarsely. Striking the right balance between mesh refinement and computational cost is therefore a central challenge in any CFD workflow.

2.3.4.1 Mesh Parameters

The computational domain is divided into several refinement regions, each associated with a different cell size. As shown in fig**??**, these regions are defined by nested boxes of increasing resolution as the wing surface is approached. The cell sizes assigned to each region were not the subject of this study and were defined in accordance with the mesh settings established by the aerodynamics division of the EPFL Racing Team, ensuring consistency with the existing 3D simulation framework. The focus of the present work therefore lies in the geometric parameters of the wing rather than in the mesh refinement strategy itself.

Refinement region	**Cell size [mm]**
Outer domain	40
Middle Field	10
Near Field	2

Table 2.2: Mesh refinement regions and associated parameters

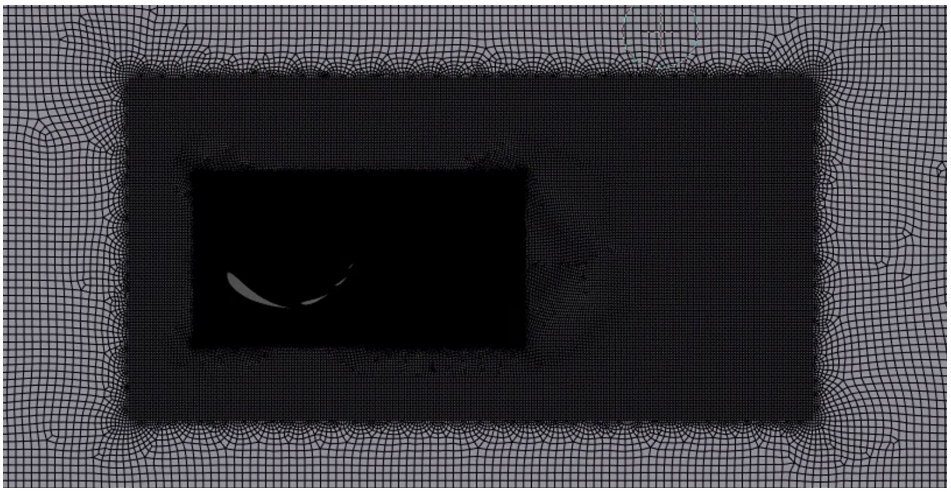


Figure 2.22: Global mesh with box of refinement

2.3.4.2 Adaptive Mesh Refinement

After defining the refinement boxes, an adaptive mesh refinement tool was used to further improve the mesh resolution in the regions of interest. This approach requires an initial converged solution as a starting point, which is precisely why a first mesh was generated using the rectangular refinement boxes described above.

In this study, the velocity gradient was chosen as the driving criterion, as it is particularly relevant for identifying regions where the flow has interacted with a solid surface and exhibits strong gradients such as the boundary layer and the slot regions, which are critical for the purpose of this study. The refinement threshold was set such that cells where the local velocity gradient exceeds the global average are refined. Once this criterion is applied, the simulation is re-run on the updated mesh. This ensures that regions of high flow complexity are resolved more finely, while areas of uniform flow remain coarsely meshed, keeping the computational cost under control.

Since this adaptive refinement process forms the foundation of the mesh strategy adopted in this study, its influence on the solution must be carefully assessed through a mesh convergence analysis, which is presented in the following section.

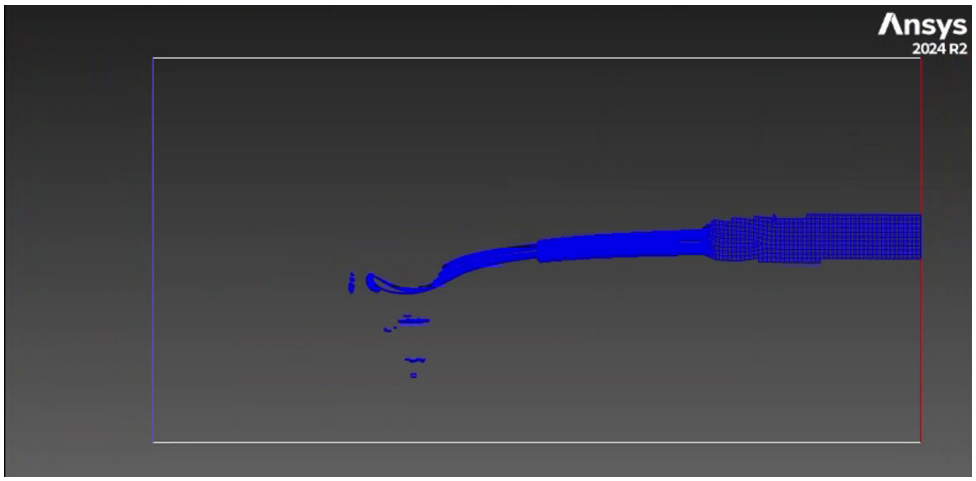


Figure 2.23: Adaptive mesh refinement according to the velocity gradient refinement criterion

2.3.5 CFD Configuration and Solution

Once the mesh has been generated and refined, the computational domain is ready to be solved. This section describes the solver settings and boundary conditions that form the core of the CFD framework, all performed within Ansys Fluent in the Workbench environment, consistently with the methodology employed by the aerodynamics division of the EPFL Racing Team.

2.3.5.1 Parameters

The following numerical parameters define the CFD setup used throughout the study. To maintain consistency with the validated 3D numerical framework developed by the aerodynamics division, these settings are directly adopted without further optimisation.

- **Turbulence model:** The k-ω SST turbulence model is selected for this study. This model is widely used in external aerodynamics as it combines the robustness of the k-ω formulation near the wall with the free-stream independence of the k-ε model in the outer flow. It is particularly well suited for flows involving adverse pressure gradients and boundary layer separation, which are central phenomena in multi-element wing aerodynamics.
- **Boundary conditions:** A velocity inlet is applied at the upstream boundary, with a freestream velocity consistent with the operating conditions of the vehicle. A pressure outlet is applied at the downstream boundary. The wing surfaces are treated as no-slip walls. The upper and lower boundaries of the domain are set as symmetry conditions to minimise their influence on the flow around the wing.

Boundary	Type	Value
Inlet	Velocity inlet	$U_\infty = 20$ m/s
Outlet	Pressure outlet	$p = 0$ Pa (gauge)
Wing surfaces	No-slip wall	$U = 0$
External walls	Specified Shear	$\tau = 0$ Pa

Table 2.3: Summary of boundary conditions applied in the CFD simulation

- **Solver settings:** A pressure-based coupled solver is used with second-order spatial discretisation schemes for all transport equations, ensuring sufficient accuracy for the flow gradients present in the domain. Convergence is monitored through the residuals of all solved quantities as well as the lift and drag coefficients, which are tracked throughout the simulation to confirm that a steady solution has been reached.

This numerical setup ensures both computational efficiency and sufficient physical fidelity for iterative aerodynamic analysis, allowing the 2D model to remain coherent with the higher-fidelity simulations used for final validation.

2.3.5.2 Post-Processing

The following quantities are extracted from the CFD simulations to evaluate and compare the aerodynamic performance of each configuration. They are computed in predefined regions of interest around each airfoil element and in the wake, ensuring a consistent basis for comparison across all iterations. These metrics form the foundation for the analysis presented in the results section.

- **Lift and drag coefficients:** The lift and drag coefficients are used as the primary performance indicators of the different configurations. They provide a global evaluation of the aerodynamic efficiency of the multi-element wing and allow direct comparison between design iterations. In this study, they are used as the main ranking criteria to assess the influence of geometric modifications on overall performance.
- **Pressure coefficient distribution:** The pressure coefficient distribution is extracted along each airfoil surface to analyse the local aerodynamic loading. It provides insight into the pressure difference between the upper and lower surfaces of each element and allows identification of the regions primarily responsible for lift generation. This metric is particularly relevant for evaluating aerodynamic interaction effects within the multi-element configuration.

Figure 2.24: Pressure coeffcient distihution across a multi element wing

- **Velocity field analysis:** Velocity contours are used to visualise the flow behaviour around the wing, with particular focus on the slot regions and the wake. These regions are analysed to assess flow acceleration through the gaps between elements, as

well as the development and recovery of the wake downstream. This provides a qualitative understanding of the energy exchange between elements and the effectiveness of the multi-element configuration.

The extracted quantities provide both global and local insight into the aerodynamic behaviour of each configuration. They form a consistent basis for comparison between iterations and will be used in the following sections to analyse and interpret the influence of geometric modifications on the overall performance of the multi-element wing.

2.4 Mesh Convergence Study and Method Choice

This section presents the convergence study conducted on both the mesh and the numerical schemes used to solve the RANS equations. These two aspects represent critical choices in the workflow, as they directly condition the accuracy and reliability of the results. The numerical scheme selection is addressed first, as its outcome directly influences the mesh convergence analysis that follows

2.4.1 Solver Method Comparison

While the global CFD workflow follows the one established by the aerodynamics division, a dedicated comparison of numerical schemes was conducted after convergence issues were observed with the baseline 2D configuration during initial runs. This subsection presents the different methods considered and the arguments that led to the selection of the most appropriate one. As shown in the table below, here are the potential choices:

Case	Coupling	Discretisation
1	COUPLED	2nd order upwind
2	COUPLED	1st order upwind
3	SIMPLE	1st order upwind
4	SIMPLE	2nd order upwind

Table 2.4: Solver configurations compared in the method study.

Among these possibilities, the one meet the most of the goals of this study is the second option : COUPLED and Simple order. The details of the results of this short study has been published in appendix.

2.4.2 Mesh Convergence Study

Although mesh optimisation is not the primary objective of this study, it is essential to verify that the simulation results are independent of the spatial discretisation of the domain. A mesh that is too coarse introduces numerical errors that may mask or distort the physical phenomena of interest, while an excessively fine mesh leads to prohibitive computational costs without meaningful gain in accuracy. The goal of the convergence study is therefore to identify the level of refinement beyond which the results no longer vary significantly, ensuring that the extracted quantities reflect the fluid dynamics of the configuration rather than development of the discretisation. The convergence study was conducted using the adaptive mesh refinement approach described in sec**??**, which ensures that the refinement is concentrated in the regions most relevant to this study.

2.4.2.1 Procedure

The mesh convergence study is performed on the baseline configuration, which will be fully defined later. This configuration is representative of the initial geometry used in the 3D design workflow and ensures consistency between the 2D and 3D comparisons. At this stage, the focus is placed exclusively on the quality of the mesh and the assessment of its convergence, independently of any geometric optimisation.

Six mesh levels are evaluated, ranging from the initial refined configuration described in Sto a significantly finer discretisation. At each level, the adaptive refinement procedure, using the criteria defined below, is applied following an initial converged solution, and the process is repeated iteratively until the target refinement level is reached. The quantities monitored throughout this study are the lift coefficient C_L and the drag coefficient C_D, tracked as a function of the total number of elements in the mesh.

Figure 2.25: Initial Mesh

Figure 2.26: Final Mesh

2.4.2.2 Results

The results of the mesh convergence study are presented below. Figure X shows the evolution of the lift and drag coefficients as a function of the number of mesh elements, allowing the convergence behaviour of the solution to be assessed visually. The numerical values associated with each mesh level are summarised in Table X. A mesh is considered converged when the variation in C_L and C_D between two consecutive levels falls below a threshold of X%, indicating that further refinement no longer has a significant influence on the solution. The velocity contour plots presented alongside provide a qualitative assessment of the flow field at each refinement level, allowing any remaining mesh-induced artefacts to be identified.

Mesh level	**Number of cells**	C_L Level 1 (coarse)
430 795	511.21	
Level 2	581 531	530.95
Level 3	767 165	544.2
Level 4	938 834	545.31
Level 5	1 018 213	545.67

Table 2.5: Mesh convergence study results

Figure 2.27: Mesh convergence graph

(a) Cl plot

(b) Zoomed Cl plot

(c) Cl results for mesh convergence

(d) Cl plot

(e) Zoomed Cl plot

(f) Cl results for mesh convergence

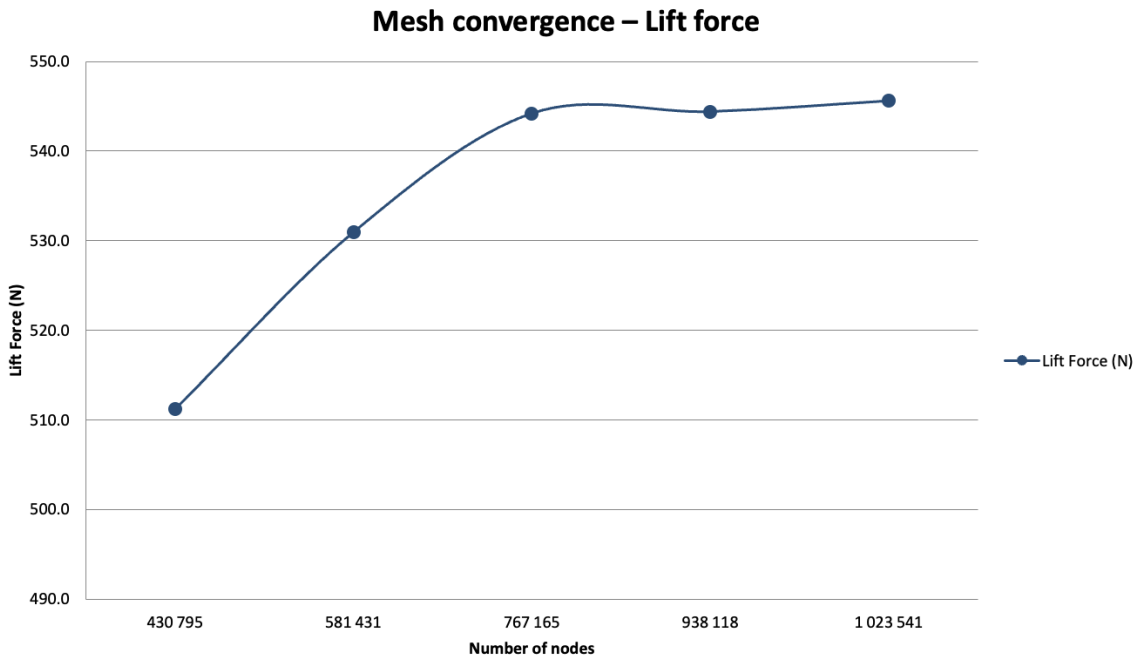


Figure 2.29: Mesh convergence plot with Lift force as a value of reference.

Table 2.6: Mesh convergence study — Lift force as a function of node count

headerblue

1	430 795	511.21	—	—
rowalt 2	581 431	530.95	19.74	3.861
3	767 165	544.20	13.25	2.496
rowalt 4	938 118	544.40	0.20	0.037
5	1 023 541	545.60	1.20	0.220

To summarise, the workflow developed in this study combines a coordinate-based geometry generation script in MATLAB, a structured meshing strategy with adaptive refinement concentrated in the critical flow regions, and a RANS solver using a coupled scheme with first-order discretisation. The mesh convergence study demonstrated that a sufficient level of independence from the spatial discretisation is achieved at mesh level X, which is therefore adopted for all subsequent simulations. This workflow provides a reliable and computationally efficient framework for the iterative geometric optimisation study presented in the following sections.

2.5 Case Study — 2D vs 3D Configuration

With a fully defined and validated numerical model in hand, this section presents the case study conducted to establish the correlation between the 2D and 3D simulation environments. This comparison forms the foundation of the study, as the reliability of the 2D workflow depends on its ability to reproduce the aerodynamic trends observed in the 3D model. The following sections detail the assumptions made to bridge the two environments, define the respective computational domains, and describe the methodology used to carry out the matching process.

2.5.1 Physical Differences Between 2D and 3D Flow

Understanding the theoretical differences between a 2D and a 3D aerodynamic domain is essential, as they directly condition the interpretation of the results. The fundamental distinction lies in the assumption of an infinite span in the 2D model, which simulates a single chordwise section of the wing, versus the finite span of the real three-dimensional geometry.

In three dimensions, spanwise pressure gradients cause the flow to move around the wing tip, forming a tip vortex. This vortex creates a downwash on the wing, which reduces the effective angle of attack seen by each section. As shown in fig2.30, additional vortex formations also appear in the slot regions, often referred to as omega vortices. These are generated by the interaction between the slot flow and the spanwise motion. Secondary vortices can also appear due to the interaction with the endplates. Although the endplates help reduce the strength of the tip vortex, the omega vortices remain an important three-dimensional effect that cannot be captured in the 2D model.

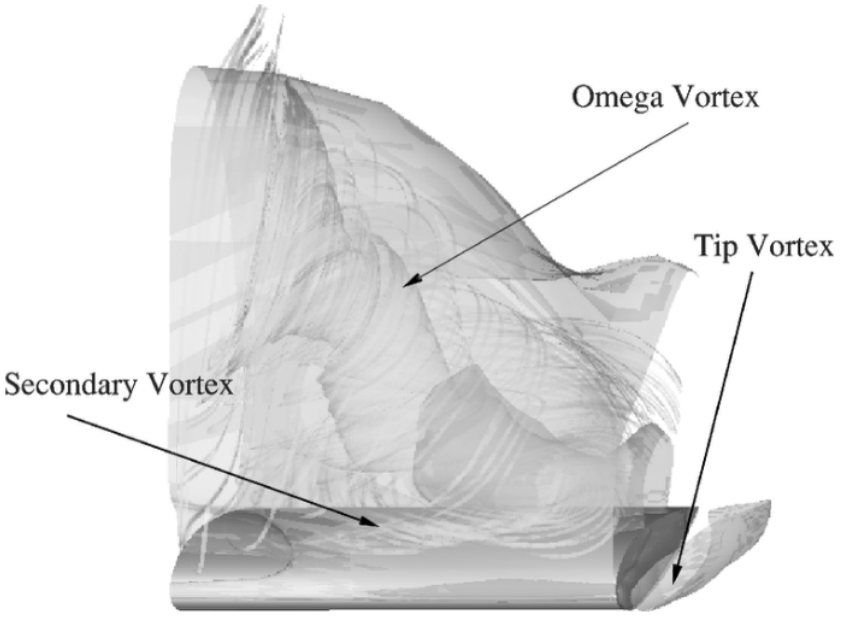


Figure 2.30: Types of vortices. Taken from [19]

As a direct consequence, the 2D simulation systematically overpredicts the downforce compared to the three-dimensional configuration, as illustrated in fig2.31, which compares the lift coefficient between the two environments across a range of angles of attack. It can be observed that the stall point occurs at a lower angle of attack in the 2D case than in the 3D one, consistent with the more aggressive flow conditions imposed by the infinite span assumption. Furthermore, if the same angle of attack is imposed in both models, a configuration operating close to stall in 3D will likely exhibit fully separated flow in the 2D simulation, as the absence of spanwise relief makes the flow significantly more aggressive near the leading edge. This behaviour, while a limitation of the 2D model, also acts as a conservative indicator: maintaining attached flow in 2D provides a stronger guarantee that the 3D configuration remains within a safe aerodynamic margin.

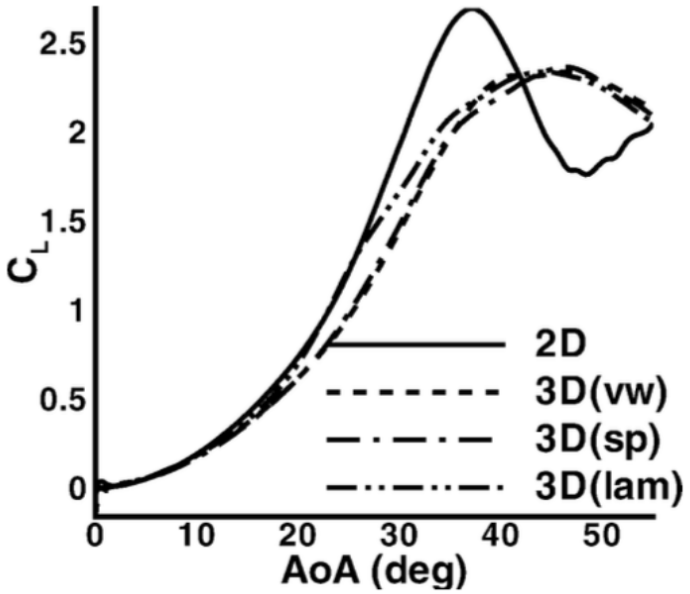


Figure 2.31: Comparison graph between a 2D airfoil and the 3D version of it in terms of Cl over AoA. Taken from [19]

This systematic offset must be understood and quantified before any meaningful comparison between the two environments can be established, which motivates the correlation methodology presented in the following sections.

2.5.2 Model Environment Differences

Beyond the physical differences discussed above, the two simulation environments also differ significantly in terms of geometric complexity. The following paragraphs detail the respective domains and highlight the key differences that must be taken into account when establishing the correlation between the two models.

3D simulation environment:

The three-dimensional domain integrates the full car geometry, introducing upstream flow interactions that are entirely absent from the 2D environment. The components that most directly influence the flow reaching the rear wing are identified in Table 2.7 and annotated in Figure 2.32.

N°	Component
1	Driver
2	LiDAR sensor and camera pod
3	Roll hoop
4	Endplate
5	Chassis

Table 2.7: Components in the 3D domain that are missing from the 2D model

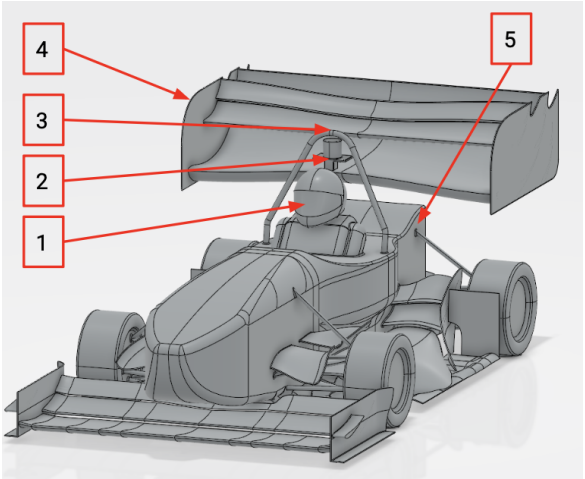


Figure 2.32: 3D domain with annotated components

2D simulation environment:

The 2D simulation uses a clean and uniform freestream, with no upstream geometry included. The inflow is therefore homogeneous and unaffected by any upstream aerodynamic devices, as illustrated in Figure 2.20.

Comparison:

The fundamental differences in inflow conditions and geometric complexity between the two domains imply that a direct comparison of absolute C_L values is not physically meaningful without proper interpretation. As shown in fig 2.33 and fig**??**, the two computational setups highlight the contrast between a clean freestream 2D environment and the complex 3D flow field generated by the full vehicle geometry.

It should also be noted that a small geometric discrepancy may exist between the 2D and 3D models due to the fundamental difference in geometry generation methods coordinate-based for the 2D workflow and CAD-based for the 3D design. This introduces a tolerance on parameters such as the slot gap and overlap that cannot be fully eliminated, and may contribute to the residual differences observed in the correlation results. This uncertainty is acknowledged and accepted as an inherent limitation of the methodology. The final comparison between the 3D reference and the 2D baseline will be studied in the Optimisation section.

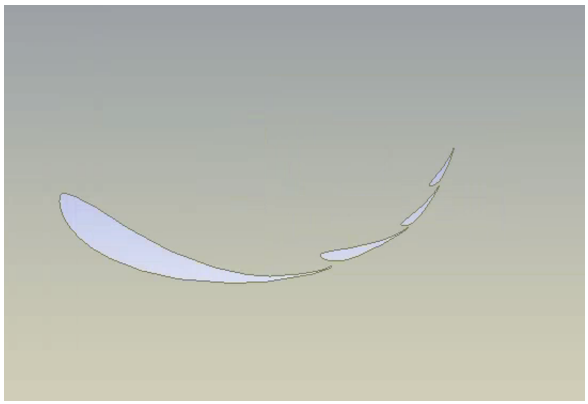


Figure 2.33: Side view of the 2D geometry

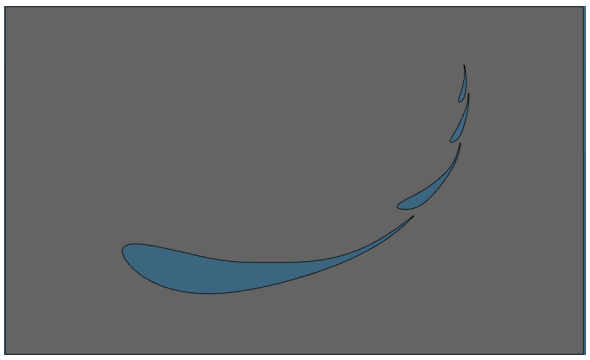


Figure 2.34: Side view od the 3D geometry

To ensure a consistent comparison, the 2D model is interpreted as representative of an isolated wing section, and the analysis focuses on aerodynamic trends rather than absolute values. An inherent offset between the two environments is therefore expected and accepted, as it stems directly from the physical and geometric differences described above. The objective of this correlation is not to obtain identical results, but to verify that the 2D model captures the same aerodynamic tendencies as the 3D configuration, enabling its use as a reliable and efficient tool for design iterations.

2.5.3 Correction Method — Rotation of the Inflow Direction

Taking these differences into account, the 2D model must be adapted to minimise their impact and establish a consistent basis for comparison. The correction method adopted in this study follows two principles: the selection of a representative spanwise section and an equivalent rotation of the incoming airflow direction.

2.5.3.1 Spanwise section selection

The rear wing geometry presents variable adjustment along its span, which would ideally require the study to be conducted at multiple spanwise positions to fully characterise the three-dimensional behaviour. However, in order to ensure a direct and meaningful comparison, the section at $y = 0.525$ m is selected, as it corresponds to the reference section used in the 3D design workflow. This choice guarantees that the 2D and 3D models share the same initial geometry, avoiding any discrepancy that would arise from comparing two geometrically different configurations.

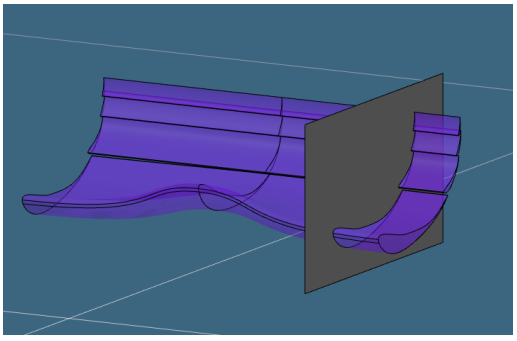


Figure 2.35: 3d view of the 0.525 m outboard section compared with the whole Rear Wing

2.5.3.2 Inflow rotation

The second correction concerns the direction of the incoming flow. As discussed previously, the three-dimensional flow environment introduces complex phenomena, including spanwise pressure gradients, vortical structures, and the influence of surrounding car components, that collectively modify the effective flow conditions seen by the rear wing. Rather than attempting to model each of these effects individually, a global correction is applied in the form of a rotation of the inflow direction in the 2D model, so that the effective angle of attack seen by the wing section is consistent between the two environments. This rotation is parametrised directly in Ansys Workbench as a named input parameter, allowing multiple inflow angles to be evaluated automatically without requiring manual intervention at each iteration. The effective rotation angle that best reconciles the 2D and 3D aerodynamic coefficients is identified through a dedicated parametric study presented in the following subsections.

2.5.4 Results and Selection of the Baseline

Having established the two corrections required to bridge the 2D and 3D environments which are the spanwise section selection and the inflow rotation, the following results present the outcome of the correlation study conducted at $y = 0.525$ m. At this location, a parametric study is carried out to determine the inflow rotation angle that best aligns the 2D results with the 3D reference. Before presenting the results, the following subsection introduces the comparison metric selected and justifies its relevance for this analysis.

2.5.4.1 Variable of Interest

Selecting the appropriate quantity of interest is crucial in the context of this study, as it serves as the foundation of the entire correlation and all subsequent results will be interpreted relative to it.

Two main candidates can be considered: the integrated aerodynamic coefficients C_L and C_D, which provide a global measure of wing performance, or the pressure coefficient C_p distribution along each element, which gives a local description of the flow field. The aerodynamic coefficients aggregate the contribution of all elements into a single value, masking local discrepancies between the two environments. The C_p distribution, on the other hand, allows the loading of each individual element to be examined separately. For a study focused on the geometric optimisation of a multi-element wing, this local insight is far more valuable than a global metric alone.

The pressure coefficient C_p is in Equation 2.1, where p is the local static pressure, p_∞ the freestream static pressure, ρ_∞ the freestream density, and V_∞ the freestream velocity.

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty V_\infty^2} \tag{2.1}$$

Being non-dimensional, C_p allows a meaningful comparison between the 2D and 3D environments even when freestream conditions differ slightly. A negative C_p indicates accelerated flow and low pressure, while a positive value corresponds to decelerated flow and high pressure. As illustrated in Figure 2.36, the distribution typically exhibits a suction peak near the leading edge on the suction side, followed by a pressure recovery toward the trailing edge.

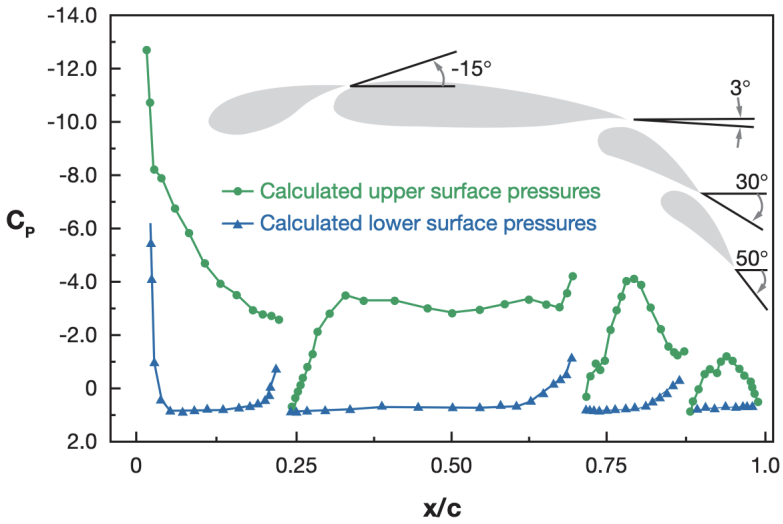


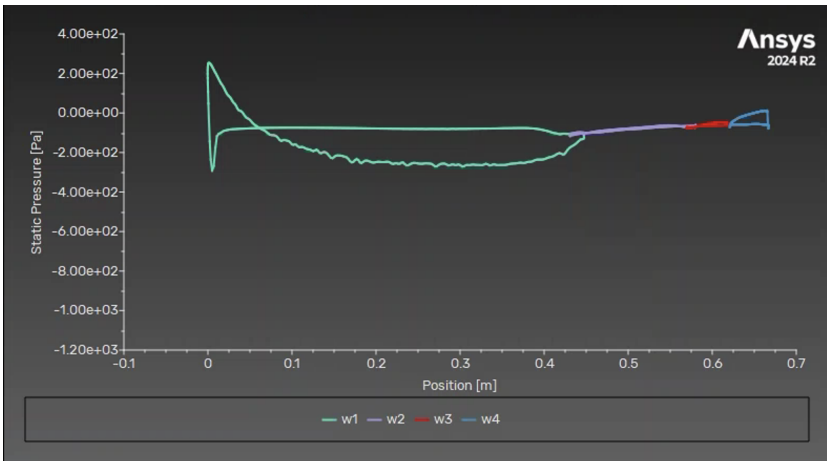
Figure 2.36: Typical C_p distribution along a lifting airfoil. Taken from [14]

It should be noted that even after applying the rotation correction, a perfect agreement between the 2D and 3D C_p distributions is not expected. The residual differences are accepted as an inherent limitation of the modelling assumptions, and the focus is placed on achieving the best possible trend agreement along each element rather than an exact match.

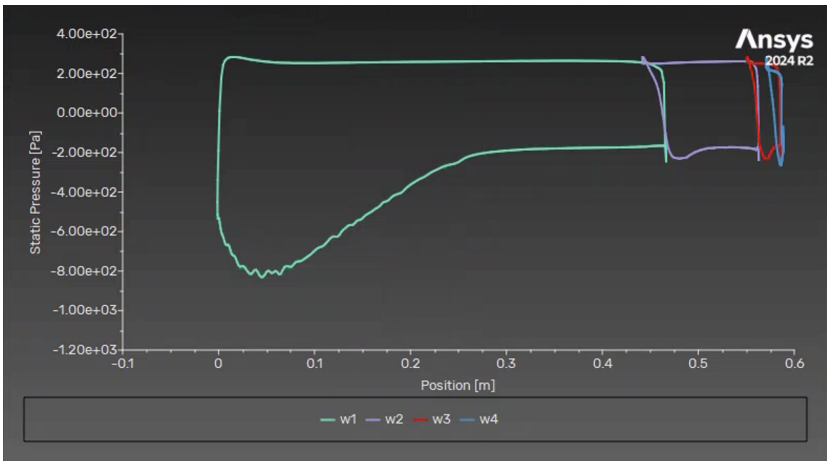
2.5.4.2 Graphical Results and interpretation

The following results present the outcome of the rotation angle parametric study. For each configuration tested, the C_p distribution obtained from the 2D simulation is compared against the reference 3D result on the baseline geometry. The rotation angle that yields the closest agreement across all elements is selected as the inflow correction for the 2D model, which then serves as the validated baseline for the geometric optimisation study conducted in the following sections.

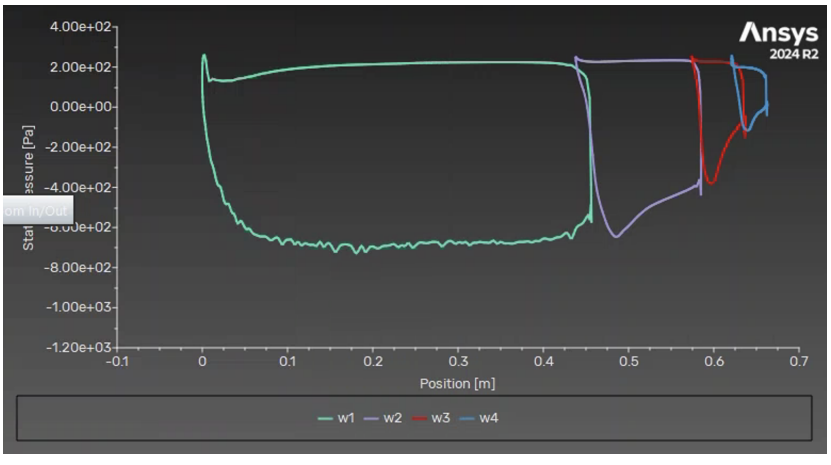
Among all the configurations evaluated, the rotation angle yielding the closest C_p agreement is retained and presented below, from which a constructive comparison is made based on the overall consistency across all four elements between the principle options.



(a) Extreme rotation angle (25°)



(a) Extreme rotation angle (25°)



(a) Extreme rotation angle (25°)

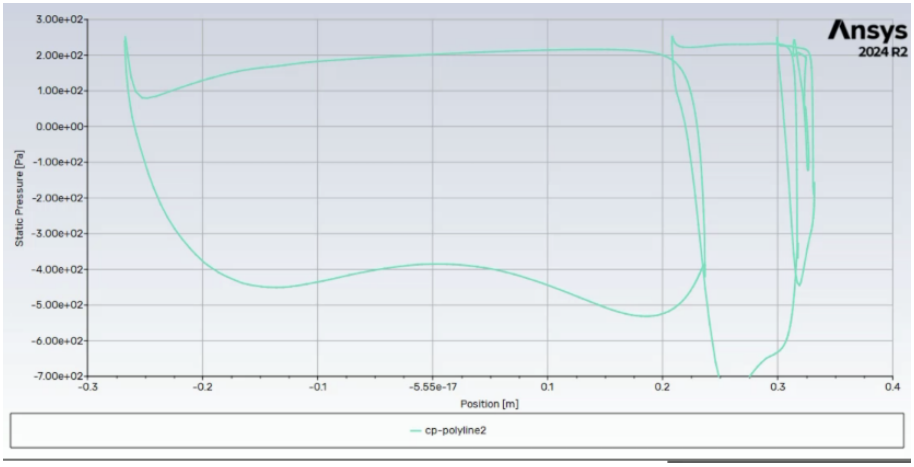


Figure 2.40: 3D reference C_p distribution

The C_p distributions presented in Figure **??** illustrate the effect of the inflow rotation on the agreement between the 2D and 3D results. At 0, the suction peak on the leading element is significantly overpredicted by the 2D model, and the pressure recovery along the suction side diverges from the 3D reference toward the trailing edge. The worst rotation angle further deteriorates this agreement, shifting the stagnation point and distorting the loading distribution across all elements. At the optimal rotation angle of $X°$, the suction peak amplitude and position are well matched with the 3D reference, the pressure recovery follows a consistent trend along the suction side, and the stagnation point at the leading edge is correctly captured. The agreement is particularly good on the upstream elements, while a small residual discrepancy remains on the rearmost element, consistent with the three-dimensional effects discussed previously that the 2D model cannot fully reproduce.

The rotation angle of $X°$ is selected as the inflow correction for the 2D model, as it yields the best overall agreement with the 3D reference C_p distributions across all elements. This corrected configuration is adopted as the validated 2D baseline for all geometric iterations conducted in the optimisation study presented in Section 2.6.

2.6 Optimisation of the Four-Element Rear Wing

This section presents the core of the study: the geometric optimisation of the multi-element rear wing. Building on the validated numerical workflow and the established 2D baseline, a series of iterations is conducted by systematically varying the key parameters identified in the theoretical framework while monitoring the aerodynamic response of the configuration. Each significant change is accompanied by an interpretation of the results, with the aim of building a comprehensive understanding of the influence of each parameter on the aerodynamic performance of the multi-element assembly. To this end, the 3D reference geometry is first described, as it defines the geometric parameters from which the 2D baseline is derived, before the optimisation iterations are presented.

2.6.1 Initial 3D Geometry

The 3D reference geometry used in this study corresponds to the initial rear wing design produced by the aerodynamics division of the EPFL Racing Team, designated 15R2 according to their internal nomenclature. This first design serves as the starting point of the optimisation process, its geometric parameters are extracted to define the 2D baseline, from which the iterative improvement study is conducted. These parameters, summarised in the table and figures below, will be the primary variables modified throughout the study and are subject to significant variation across the different configurations explored. The global camber of the assembly is also reported for each iteration as an indicative measure of the overall geometric aggressiveness, but is not used as a direct optimisation parameter.

Parameter	**Element 1**	**Element 2**	**Element 3**	**Element 4**
Chord length [mm]				
Angle of attack [°]				
Slot gap [mm]				
Overlap [mm]				
Airfoil profile				
Global camber [%]*				

Table 2.8: Geometric parameters of the 3D reference geometry

* Indicative measure only — not used as an optimisation parameter.

Figure 2.41: 3D view of the reference rear wing

Figure 2.42: Cross-section at $y = 0.525$ m

Figure 2.43: Annotated 2D cross-section with geometric parameters

2.6.2 2D Geometry Presentation

The 2D baseline configuration and all subsequent iterations are derived from the 3D reference geometry presented previously. The optimisation process is conducted within the 2D design space, where geometric modifications and design constraints are applied directly to the cross-sectional profile.

Each configuration is characterised by its corresponding geometric parameters and illustrated through two assembly views: one in the reference orientation and another rotated to account for the inflow correction implemented in the CFD simulations.

To ensure consistency and enable meaningful comparisons throughout the optimisation study, the following design constraints are enforced in accordance with the methodology adopted in the 3D design workflow:

- The geometry must remain within the regulation box defined by the Formula Student rules.
- The leading edge of the main element is fixed.
- The trailing edge of the fourth and final flap is fixed.

These constraints ensure that the explored design space remains physically relevant while preserving direct comparability between all generated configurations and the baseline geometry.

Prior to exporting the coordinates, the workflow generates a visual representation of the geometry shown in figure below, including construction references that facilitate the assessment of the implemented modifications. These references include the chord line of each element, which allows a quick verification of the individual angles of attack, as well as a parametrisable circle highlighting the slot region between consecutive elements according to the specified target diameter, providing an immediate indication of whether the gap and overlap remain within a reasonable range.
This visualisation is presented below, alongside the rotated configuration used directly as input for the CFD simulation.

(a) Reference orientation (b) Rotated for inflow correction

Figure 2.44: Standard template used for geometry visualisation in the optimisation study.

2.6.2.1 2D baseline parameters

The 2D baseline is derived directly from the 3D reference geometry, with the geometric parameters extracted at the spanwise section $y = 0.525$ m and adapted to the coordinate-based workflow. The key parameters of the baseline configuration are summarised in Table 2.9 and the geometry is illustrated in Figure **??**.

Parameter	Element 1	Element 2	Element 3	Element 4
Chord length [mm]				
Angle of attack [°]				
Slot gap [mm]				
Overlap [mm]				
Airfoil profile				
Global camber [%]*				

Table 2.9: Geometric parameters of the 2D baseline configuration

* Indicative measure only — not used as an optimisation parameter.

Figure 2.45: 2D baseline — reference orientation

Figure 2.46: 2D baseline — rotated configuration

2.6.3 Comparison: 2D Baseline vs 3D Baseline

Add a side-by-side overlay figure: 2D geometry in one colour, 3D mid-span cross-section in another.
Show at least two viewing angles and highlight geometric differences.

The two configurations are geometrically very close at the mid-span section. Small differences exist at the trailing edges due to manufacturing constraints in the 3D design.

2.6.4 Description of the Analysis

2.6.4.1 Starting Point

All iterations start from the corrected 2D baseline, ensuring that every case is evaluated under the same reference conditions.

2.6.4.2 Chord Variation

A longer chord increases the loading on that element but also the adverse pressure gradient and the risk of separation. A shorter chord reduces loading but may shift too much of the burden to adjacent elements.

Add velocity contour and C_p plots for the most representative chord iterations with a brief interpretive paragraph after each figure.

2.6.4.3 Slot Gap and Overlap Variation

The slot gap and overlap are the most sensitive parameters in this study. A larger slot gap reduces jet velocity but increases mass flow. A smaller gap accelerates the jet but risks choking the slot. The overlap controls the angle at which the jet impinges on the downstream element.

Add velocity contour, C_p, and velocity vector plots for the most representative iterations with a brief interpretive paragraph after each.

2.6.4.4 Angle of Attack Variation

Increasing the AoA of a flap increases its individual loading but brings it closer to stall. The modified circulation also changes the pressure field seen by adjacent elements, since the elements are aerodynamically coupled.

Add velocity contour and C_p plots for the most representative AoA iterations. Comment on any observed separation.

2.6.5 Summary of Iterations

Table 2.10: Summary of geometric families and aerodynamic results.

headerblue

Baseline	Corrected 2D initial geometry
rowalt Small chord	Reduced main plane chord
Mid chord	Intermediate main plane chord
rowalt Large chord	Extended main plane chord
Best	**Optimal configuration**

2.6.6 Conclusion of the 2D Analysis and Transfer to 3D

The 2D parametric study identifies the parameters with the greatest influence on the performance of this four-element configuration. The optimised 2D geometry maximises C_L/C_D while maintaining attached flow on all four elements.

The optimised cross-section is applied to the 3D assembly. The chord, AoA, slot gap, and overlap values are transferred to the 3D CAD model. Non-aerodynamic components are preserved from the initial design.

Add multi-angle renders of the updated 3D assembly: isometric, front, rear, top views.

Table 2.11: Geometric comparison between initial and updated 3D rear wing designs.

Parameter	**Initial**	**Updated**	**Change**
Chord main [mm]			
Chord flap 1 [mm]			
Chord flap 2 [mm]			
Chord flap 3 [mm]			
AoA flap 1 [deg]			
AoA flap 2 [deg]			
AoA flap 3 [deg]			
Slot gap 1–2 [mm]			
Slot gap 2–3 [mm]			
Slot gap 3–4 [mm]			

2.7 Results and Discussion

2.7.1 3D CFD Validation of the Updated Design

Both the initial and updated 3D designs are simulated under identical conditions.

Add velocity contour comparisons and C_p plots for both configurations, per element if possible.

Table 2.12: Aerodynamic performance comparison between the initial and updated 3D designs.

Configuration	C_L	C_D	C_L/C_D
Initial 3D design			
Updated 3D design			
Change [%]			

Fill in the table with your CFD results.

2.7.2 Overall Assessment

The 2D workflow successfully identifies a geometry that improves C_L/C_D by approximately [X]% compared to the initial design. The improvement is primarily driven by [main geometric change].

Replace [X] and describe the main geometric change responsible.

The 3D CFD results confirm the direction of the improvement predicted by the 2D study. The relative ranking of the two configurations is consistent between the 2D parametric study and the 3D validation.

2.7.3 Limitations of This Study

2.7.3.1 Residual 2D/3D Gap

The rotation correction reduces but does not eliminate the difference between 2D and 3D C_p distributions. A more complete correction would require a dedicated study of the non-uniform inflow profile at the rear wing inlet.

2.7.3.2 Turbulent Inflow

The 3D rear wing receives a turbulent, non-uniform inflow from the upstream car body. The 2D model uses a clean freestream. This difference affects boundary layer transition and separation characteristics and is not captured by the rotation correction alone.

2.7.3.3 Steady-State Assumption

The RANS solver produces a steady-state solution. The flow in the slot regions and in the wake of the rearmost flap may in reality be unsteady. A URANS or DDES simulation would capture these effects at a higher computational cost.

2.7.4 Benefits in the Formula Student Context

The 2D workflow provides a fast, accessible tool for aerodynamic optimisation in a student environment. It is particularly valuable for training new members, who can explore the parameter space and develop physical intuition without waiting for long 3D simulations.

The ability to evaluate ten configurations in one afternoon instead of one per week changes the pace of the design season.

The workflow could also be extended to other aerodynamic surfaces on the car. Standardising on a small number of aerofoil profiles reduces tooling costs, which is relevant in the budget-constrained context of a Formula Student team.

2.7.5 Possible Improvements and Future Work

- **Custom panel code.** A panel code adapted for multi-element aerofoils would reduce computation time by an order of magnitude for exploratory iterations.
- **Adjoint-based optimisation.** The adjoint solver in Fluent computes sensitivity of C_L/C_D to geometric changes, enabling automated optimisation rather than manual iteration.
- **Unsteady simulation.** A URANS or DDES study would quantify unsteady effects in the slot regions and validate the steady-state assumption.
- **Non-uniform inflow boundary condition.** Applying velocity and turbulence profiles extracted from the 3D simulation as 2D inlet conditions would reduce the residual 2D/3D gap.
- **Endplate and 3D effect study.** A systematic study of endplate geometry would complement the 2D aerofoil study and provide a more complete picture of the rear wing aerodynamics.

2.7.6 Conclusion

This report presents the development and application of a 2D CFD workflow for the optimisation of a four-element rear wing for the EPFL Racing Team.

The theoretical basis of multi-element aerodynamics is established and the key design parameters are identified. A workflow connecting MATLAB, Ansys Design Modeler, Ansys Meshing, and Ansys Fluent is developed into a repeatable pipeline. The mesh convergence and solver method are validated before the optimisation study is launched.

A correction method based on inflow rotation accounts for the differences in flow environment between the 2D model and the 3D car simulation. The corrected 2D model is used to run a parametric study and an optimised configuration is identified and transferred to the 3D design.

The updated 3D design shows a measurable improvement in aerodynamic performance confirmed by 3D CFD simulation. The workflow is now a repeatable tool for the aerodynamics division, applicable to future rear wing designs and adaptable for other aerodynamic surfaces on the car.

Chapter 3

Rear Wing

3.1 Introduction

3.1.1 Context

The aerodynamic package on the 2027 EPFL Racing Team car is built around one goal: downforce. Within that package, the rear wing does more of the heavy lifting than any other single component. It is the last surface the flow encounters before leaving the car, operating in the wake of everything upstream: the diffuser, side structures, chassis winglets, and wishbone covers. That position also makes it the primary contributor to the rear aerodynamic load $C_{z,r}$, meaning any change to its geometry has a big impact on total downforce $C_{z,t}$.

The wing is designed around the draft 2027 Formula Student Germany (FSG) ruleset [1], which introduces updated height and width restrictions for aerodynamic devices. The detailed envelope is examined in Section 3.2. It is worth noting that at the time of writing, this ruleset is still in draft form, with the final version expected in September or October 2026. The wing has therefore been developed to need only minimal modifications once the rules are finalized, ahead of manufacturing.

3.1.2 Problem Statement

In the Formula Student dynamic events, the car is rarely driving at high speed for a long period of time. Most of the lap is made up of tight corners, quick direction changes, braking zones, and short acceleration phases. Because of this, the time lost due to drag on long straights is relatively small compared to the time that can be gained by increasing cornering speed.

For this reason, drag is not treated as the main design priority for the rear wing. It is still considered, but it is accepted as a necessary trade-off when generating more downforce. The focus of the rear wing design is therefore to produce as much effective rear downforce as possible, since this gives the tires more grip, improves vehicle stability, and allows the car to carry higher speed through corners and transitions.

This part of the project starts from a three-element rear wing concept and investigates how an additional element can be integrated to create a four-element configuration. The goal is to improve rear downforce and overall aerodynamic performance while respecting the geometric, structural, and manufacturing constraints of the draft 2027 ruleset.

Research Question

How can a Formula Student rear wing be developed from a three-element configuration into a four-element configuration to increase rear downforce within the geometric, structural, and manufacturing constraints of the draft 2027 ruleset?

3.1.3 Objectives and Approach

This work has two main objectives. The first is to build a clear design methodology based on CFD post-processing. Instead of only comparing final numbers, each simulated configuration is analysed to understand where the design performs well, where it loses performance, and what can be improved in the next iteration. The main comparison criterion is the rear downforce coefficient.

The second objective is to use this methodology to develop a rear-wing geometry that improves the performance of a three-stage starting configuration. All designs are evaluated on the shared whole-car baseline V3, which is used as the common reference throughout the study. This makes it possible to compare each iteration under the same vehicle-level conditions.

All CFD simulations are carried out in ANSYS Fluent at a freestream velocity of 20,m/s. This speed is representative of the operating range of the car during the dynamic events and is therefore used as the reference condition for the aerodynamic development. It also defines the Reynolds number of the wing elements and guides the choice of airfoils, element positions, and angles of attack.

The design process starts with a three-stage rear wing configuration. This configuration is first analysed in detail to identify its strengths and limitations, including the behaviour of the flow around the elements, the effectiveness of the slots, and the regions where separation or reduced loading may appear. Based on this analysis, the study then investigates how an additional element can be introduced to create a four-stage configuration.

The four-stage development focuses on whether the added element can increase rear downforce by improving the pressure distribution, increasing the effective camber of the wing system, and making better use of the available design volume. Several iterations are compared, with changes made to the position, angle of attack, and interaction between the elements. Each step is reported as part of the design reasoning: what was changed, why it was changed, what improved, and which directions were not kept.

3.2 Constraints and Design Space

This section defines the geometric envelope within which the rear wing must be designed. Two rulesets are relevant: the current FSG 2026 rules [20], under which the car competes this season, and the draft 2027 ruleset [1], against which the present wing is designed. Only the articles that apply specifically to the rear wing are covered here.

3.2.1 Current FSG 2026 Rules

The 2026 ruleset defines the geometric constraints that are directly relevant to the rear wing. For the design phase, the focus is placed on the rear-wing aerodynamic-device

envelope. Additional requirements related to structural loading, materials, fasteners, surface finishing and manufacturing will be taken into account during the manufacturing phases.

T2.1.3 – Open-wheel keep-out zones. The vehicle must remain open-wheeled when viewed from the side. No part of the vehicle may enter the keep-out zones around the rear tires, defined by vertical planes located 75 mm in front of and 75 mm behind the tire outer diameter, with the steering system straight ahead. For the rear wing, this rule is mainly relevant for endplates, supports or any rear aerodynamic element located close to the rear tires.

T8.2 – Rear aerodynamic-device envelope. The aerodynamic-device envelope defines the allowable position of the rear wing on the vehicle:

- **T8.2.1, height.** Rear aerodynamic devices located behind the driver head-restraint reference plane must be lower than 1100 mm from the ground.
- **T8.2.2, width.** Rear aerodynamic devices lower than 500 mm from the ground must remain within the width defined by the relevant wheel/tire planes. Rear aerodynamic devices higher than 500 mm from the ground must remain inboard of the most inboard point of the rear wheel/tire assembly.
- **T8.2.3, length.** Rear aerodynamic devices must not extend further rearward than 250 mm from the rearmost part of the rear tires.

Rule T8.2.4 states that all restrictions must be fulfilled with the wheels pointing straight ahead and with any suspension setup, with or without the driver seated in the vehicle.

3.2.2 Draft 2027 Ruleset

The draft 2027 aerodynamic rules shown in the FSG academy material are marked *Draft Only, Subject to Changes*. They are therefore not treated as final requirements, but they provide useful guidance for future-proofing the rear-wing concept.

T8.2.1, height. The draft 2027 rules introduce an updated height window for rear aerodynamic devices. Rear aerodynamic devices located behind the driver head-restraint reference plane must be lower than 1100 mm and higher than 700 mm from the ground. This directly affects the rear wing, as it defines a vertical operating window for the main rear aerodynamic elements.

T8.2.2, width. The draft 2027 rules modify the width reference for high rear aerodynamic devices. Rear aerodynamic devices higher than the top edge of the tires must not extend outboard of the most outboard point of the rear wheel/tire assembly. This changes the lateral design space available for the upper rear wing and must be checked again once the final 2027 rules are released.

T8.2.3, length. The longitudinal limit remains unchanged in the draft material considered here. Rear aerodynamic devices must not extend further rearward than 250 mm from the rearmost part of the rear tires.

T8.2.4, setup condition. As in the 2026 ruleset, all restrictions must be satisfied with the wheels pointing straight ahead and with any suspension setup, with or without the driver seated in the vehicle.

3.2.3 Comparison Between 2026 and Draft 2027

The main differences between the current 2026 ruleset and the draft 2027 aerodynamic rules for the rear wing are summarized in Table 3.1.

Table 3.1: Comparison of the 2026 and draft 2027 rear-wing aerodynamic envelopes.

Item	FSG 2026	Draft FSG 2027
Rear-wing height	Rear aerodynamic devices must be lower than 1100 mm from the ground.	Rear aerodynamic devices must be higher than 700 mm and lower than 1100 mm from the ground.
Rear-wing width	Rear aerodynamic devices higher than 500 mm must remain inboard of the most inboard point of the rear wheel/tire assembly.	Rear aerodynamic devices higher than the top edge of the tires must not extend outboard of the most outboard point of the rear wheel/tire assembly.
Rearward length	Maximum 250 mm rearward of the rearmost part of the rear tires.	Unchanged: maximum 250 mm rearward of the rearmost part of the rear tires.
Structural and manufacturing requirements	To be considered during detailed design and manufacturing.	To be reviewed again once the final 2027 rules are released.

3.2.4 Resulting Design Envelope

To reduce the risk of redesign for the following season, the rear wing is developed using the draft 2027 rear-wing envelope while still remaining compatible with the official 2026 rules where applicable. The resulting design envelope is therefore:

- **Height:** between 700 mm and 1100 mm from the ground.
- **Width:** bounded laterally by the rear wheel/tire assembly. Based on the draft 2027 rules, the rear wing will be designed up to the outboard plane of the rear wheel/tire assembly. This reference will be reviewed once the final 2027 rules are released.

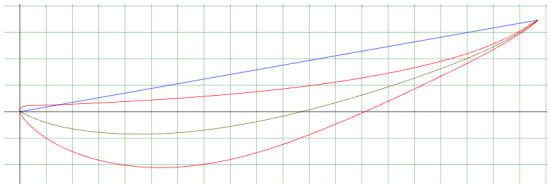
3.3 Multi-Element Wing Theory

3.3.1 Airfoil terminology

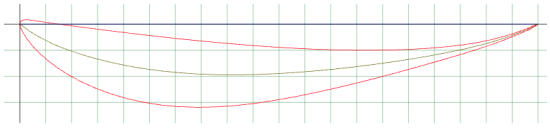
The following discussion makes use of standard airfoil terminology, which is covered in detail in Section 2.2.1. For clarity, the key parameters referred to throughout are the

chord line, the straight line connecting the leading edge to the trailing edge (Fig. 2.1), the mean camber line, which runs midway between the upper and lower surfaces and describes the overall curvature of the profile (Fig. 2.1), the trailing edge, which is the rearmost point of the airfoil, and the angle of attack, which is the angle between the chord line and the incoming freestream velocity vector V_∞ (Fig. 2.2).

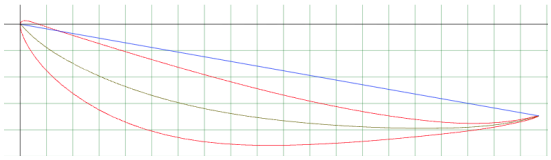
The convention for α will be taken like shown figure 3.1



(a) Positive angle of attack(+10).



(b) Zero angle of attack.



(c) Negative angle of attack−10).

Figure 3.1: Comparison between positive, zero and negative angle of attack definitions.

3.3.2 Force Generation and Pressure Distribution

3.3.2.1 Force Generation

Aerodynamic force generation is mainly the result of the pressure difference between the two surfaces of the airfoil, a mechanism described in detail in Section 2.2.2. For the suction and pressure side, if you look at figure 3.1b the suction will be the one below the camber line (line in green) and the pressure side will be the one on top of it.

3.3.2.2 Pressure distribution

The pressure distribution around an airfoil describes how the static pressure varies along its surface. This distribution is directly related to the local flow velocity: regions where the flow accelerates experience lower static pressure, while regions where the flow slows down experience higher static pressure. Therefore, the pressure distribution provides a useful way to understand how lift or downforce is generated.

The pressure distribution is commonly expressed using the pressure coefficient, C_p, defined as:

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho V_\infty^2} \tag{3.1}$$

where p is the local static pressure, p_∞ is the freestream static pressure, ρ is the air density, and V_∞ is the freestream velocity. A negative value of C_p indicates that the local pressure is lower than the freestream pressure, while a positive value indicates that the local pressure is higher.

The suction side is expected to show lower pressure values, or more negative C_p, compared to the pressure surface. The highest suction is usually found near the leading

edge, where the flow accelerates strongly around the profile. Towards the trailing edge, the pressure gradually recovers. Fig. 3.2 is an example of a C_p plot for an inverted airfoil (generating downforce), where the horizontal axis represents the position along the chord, x/c. Here, x is the distance from the leading edge and c is the total chord length. Near the leading edge, the pressure coefficient reaches its highest value, approximately $C_p = 1$, which corresponds to the stagnation region where the flow velocity is very low. On the suction side, C_p becomes strongly negative just after the leading edge, indicating a low-pressure region caused by the strong acceleration of the flow. Moving towards the trailing edge, the suction-side pressure gradually recovers, meaning that C_p increases towards less negative values.

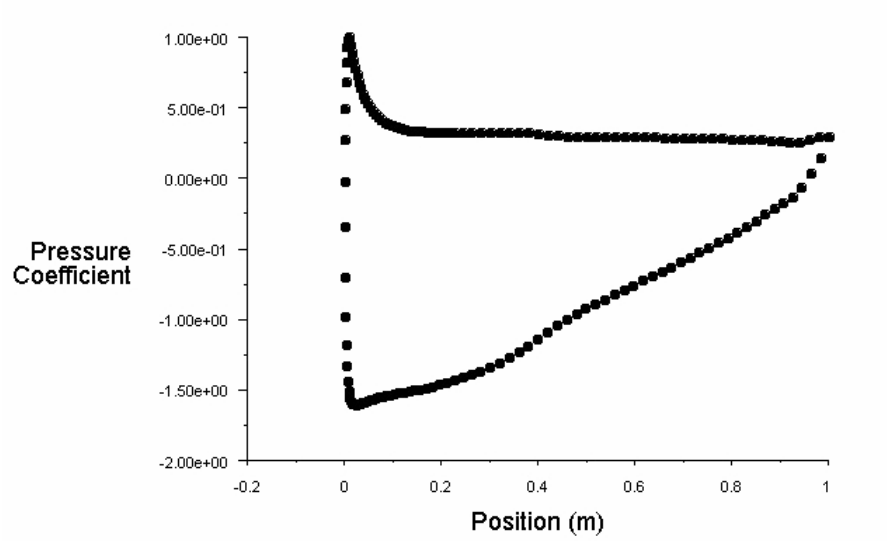


Figure 3.2: Cp plot Example inverted airfoil

3.3.3 Limits of single element Wing

As described in Section 2.2.3, a single-element wing stalls beyond a critical angle of attack, limiting the maximum downforce it can produce. This limitation is the main motivation for moving to a multi-element configuration, where additional elements allow higher downforce to be achieved while keeping the flow attached.

3.3.4 Principle of multi-element

A multi-element wing consists of a main element combined with one or more additional elements, called flaps. Compared to a single-element wing, this configuration allows the wing to generate higher downforce without immediately reaching the stall limit, like said earlier. Instead of relying on one airfoil to produce all the required downforce, the total pressure difference is distributed between the several elements.

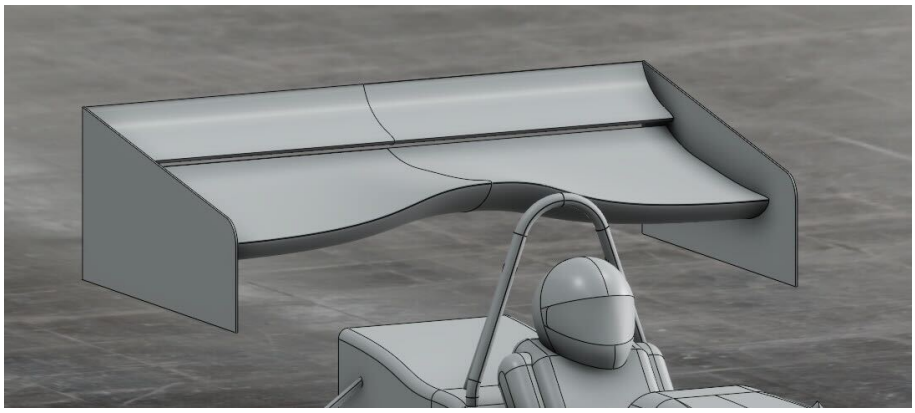
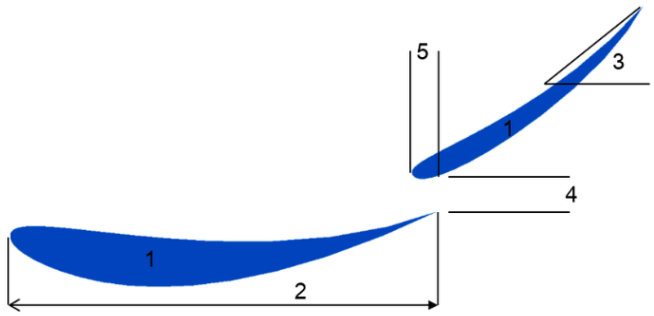


Figure 3.3: Example of two element configuration

The main principle of a multi-element wing is to increase the effective camber of the overall wing system. By placing a flap behind the main element and deflecting it at an angle, the combined profile behaves like a highly cambered airfoil. This increases the turning of the flow and strengthens the pressure difference between the suction side and the pressure side. As a result, the rear wing can generate more downforce than a single-element wing of similar size.

Another important effect is that the elements interact with each other. The slot between the main element and the flap allows the flow to pass through the gap and influence the pressure distribution on the following element. This helps the flap operate at a higher loading while keeping the flow attached over a larger part of its surface. Therefore, the multi-element configuration can delay flow separation and stall compared to a single-element wing.



1. Airfoil profile
2. Chord
3. Angle of attack
4. Slot gap
5. Overlap

Figure 3.4: Multi-element wing configuration, taken from [21]

- The total chord is the line between the leading edge of the first wing and the trailing edge of the last wing.
- The effective camber is the mean camber line of the equivalent single airfoil. Imagine the two elements combined into one single airfoil.

3.3.5 Slot Gap, Overlap

The definitions of gap and overlap for a multi-element airfoil configuration are shown in Figure 3.4.

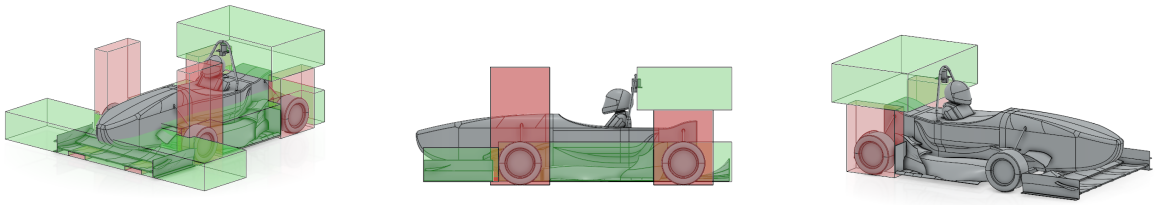
The gap and overlap allows part of the flow from the pressure side of the upstream element to pass towards the suction side of the downstream element. This flow modifies the pressure distribution around the flap and helps it operate at a higher AoA, bigger than if the airfoil was isolated. This is one of the main reasons why multi-element wings can generate higher downforce than single-element wings.

The slot gap size must be carefully selected. If the gap is too small, the flow may be restricted, which can reduce the effectiveness of the flap. If the gap is too large, the interaction between the elements becomes weaker, and the wing may behave more like separate airfoils rather than one combined high-lift system. Therefore, an optimum gap exists where the slot flow improves the pressure recovery and helps maintain attached flow on the flap.

The overlap also plays an important role in the performance of a multi-element wing. Changing the overlap affects how the wake of the main element interacts with the flap and how strongly the flow accelerates between the elements. Gap and overlap are usually optimized together, since both parameters influence the pressure distribution, separation behaviour, downforce, and drag.

3.4 Design methodology

The first thing I did before starting the design phase was drawing the box where I can design according to the 2027 rules fig 3.5.



(a) box total (b) side view (c) Box rear wing

Figure 3.5: Comparison of the three configurations

After the design area was defined I began with a simple 2-element configuration. The design was used as a preliminary study and therefore won't be discussed in the results, configuration is shown in fig 3.3.

After that, to start the design of a three-element configuration, I used the parameters of the rear wing designed last year and adjusted it to fit in the box.

The parameters were:

- For Wing 1: profile FX74, chord = 550 mm
- For Wing 2: profile MSHD, chord = 200 mm
- For Wing 3: profile MSHD, chord = 110 mm

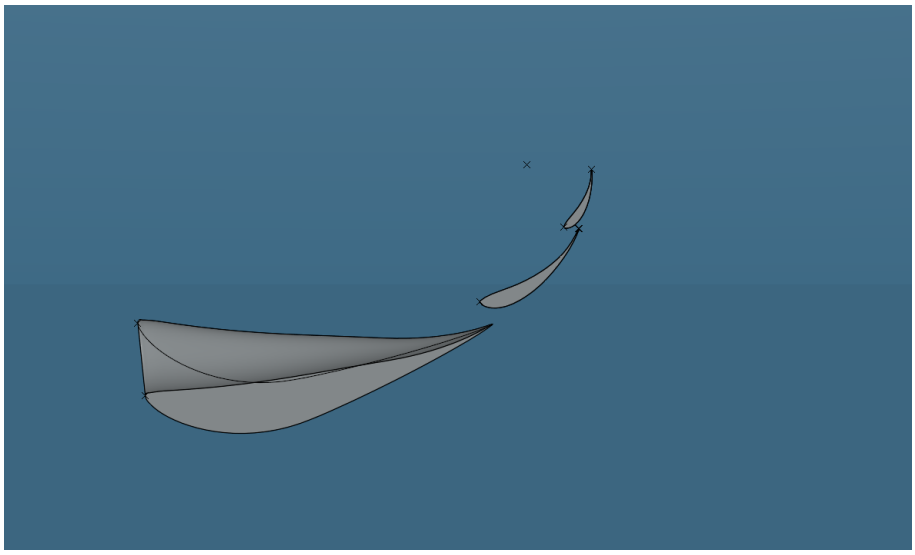


Figure 3.6: Side view of the three-element configuration

We can see in fig 3.6 that Wing 1 changes AoA between inboard (symmetry plane) and outboard (near the endplates). Furthermore, as explained in section 3.1.3 all iterations will be designed using baseline V3 and same endplates 3.7, the only thing that will change will be the rear wing.

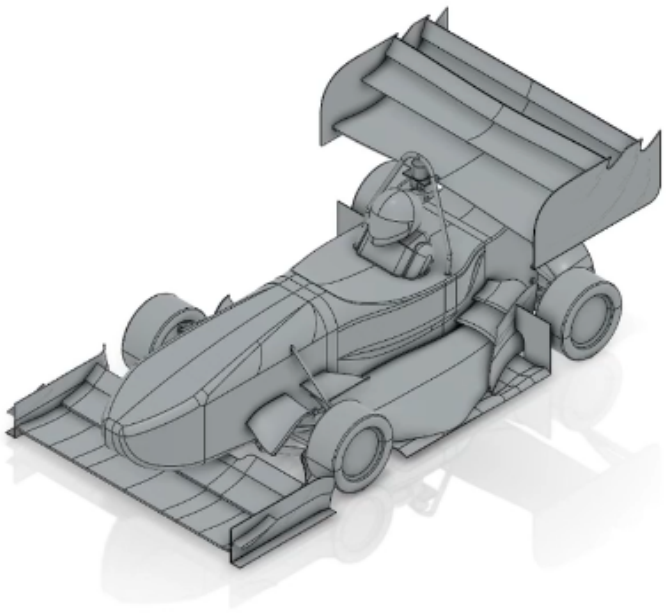


Figure 3.7: Baseline V3

I then iterated with this three element configuration before adding the fourth element. To get some ideas for designing the four-element configuration I used the tables provided in *Theory of Wing Sections* by Abbott and Von Doenhoff [13].

Slot combination	$C_{L_{max}}$	$C_{D_{min}}$	$\frac{C_{L_{max}}}{C_{D_{min}}}$	$\alpha_{C_{L_{max}}}$ degrees
	1.291	0.0152	85.0	15
	1.772	0.0240	73.8	24
	1.596	0.0199	80.3	21
	1.548	0.0188	82.3	19
	1.440	0.0164	87.8	17
	1.902	0.0278	68.3	24
	1.881	0.0270	69.7	24
	1.813	0.0243	74.6	23
	1.930	0.0340	56.8	25
	1.885	0.0319	59.2	24
	1.885	0.0363	51.9	25
	1.850	0.0298	62.1	24
	1.692	0.0228	74.2	22
	1.672	0.0214	78.2	22
	1.510	0.0208	72.6	19
	1.662	0.0258	64.4	22

(a) Table 1

Slot combination	$C_{L_{max}}$	$C_{D_{min}}$*	$\frac{C_{L_{max}}}{C_{D_{min}}}$	$\alpha_{C_{L_{max}}}$ degrees
	1.950	0.0152	128.2	12
	2.182	0.0240	91.0	19
	2.235	0.0278	80.3	20
	2.200	0.0340	64.7	21
	2.210	0.0270	81.8	20
	1.980	0.0164	120.5	12
	1.770	0.0164	108.0	14
	2.442	0.0208	117.5	16
	2.500	0.0258	96.8	18
	2.185	0.0214	102.0	18
	2.261	0.0243	93.2	19
	2.320	0.0319	72.7	20
	2.535	0.0363	69.8	20
	2.600	0.0298	87.3	20
	2.035	0.0298	68.3	21

(b) Table 2

Figure 3.8: Tables for designing multi-element wing, taken from Abbott [13] p228-229

Unfortunately, this book was discovered late into the design of the four-element configuration so most of the designs are iterative, testing changes, keeping what worked and discarding what did not.

3.5 Design Iteration

As mentioned earlier in section 3.4, the first designs are made with a three-element wing. Here I will detail how the design is made and show the results of the post process, and try to analyze them. For reference, the nomenclature of the designs will be `XXRYY`, `XX` meaning the number of the design and `YY` changes made to it and `R` for rear wing. For example, `3R5` refers to the fifth iteration of the third rear wing design. It's not robust but we're working on it.

3.5.1 Three-element Configuration

Since the baseline V3 was introduced later in the design phase, the iterations of the three-element were on the baseline V2. However, the best performing three-element configuration was also tested on the updated baseline V3. This comparaison between them will provide an order of magnitude for the expected downforce. Therefore, the comparison between the V2 and V3 results should not be interpreted as a direct optimization study, but rather as an indication of how sensitive the rear-wing performance is to changes in the surrounding vehicle geometry.

It is worth noting that between baseline V2 and V3 new designs were introduced, but the most important of them was the introduction of the LiDar sensor, it's like camera

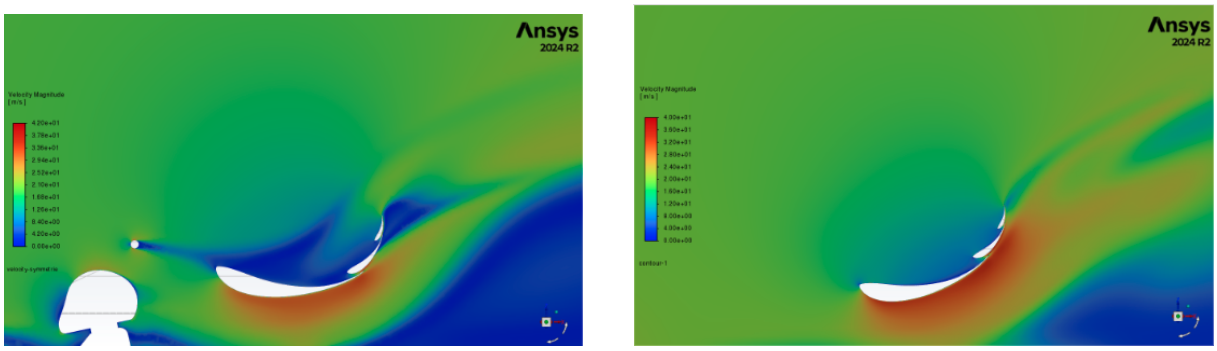
positioned between the main hoop and the driver's head, and this perturbed the flow as we will see in the post process. The Lidar can be observed by comparing figure 3.7 to figure 3.3.

At the beginning of the design process, the CFD post-processing was carried out by the team because the complete CFD workflow and result files were not directly accessible. Therefore, only limited information was available for the early design iterations, mainly velocity magnitude contours. These early comparisons were used primarily as qualitative guidance to identify flow acceleration and possible separation regions.

The first designs worth mentioning are `4R1` and `4R2`, the two of them were designed as shown in table 3.2. The MSHD airfoil was selected for the first wing due to its highly cambered profile and its greater potential for generating downforce.

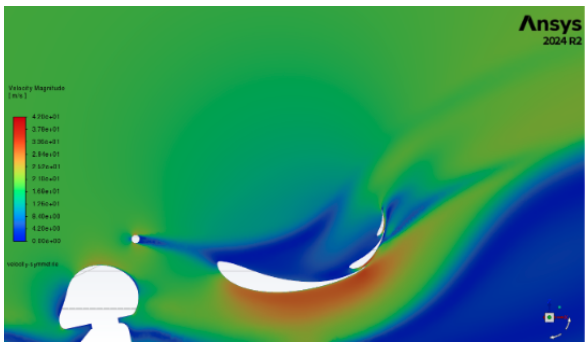
Table 3.2: Parameters of the 2 three-element wing iterations

Iteration	Wing	Airfoil	Chord [mm]	AoA [°]
4R1	W1 inboard	MSHD	550	0
	W1 outboard	MSHD	550	15
	W2	MSHD	200	50
	W3	MSHD	110	80
4R2	W1 inboard	MSHD	550	0
	W1 outboard	MSHD	550	15
	W2	MSHD	200	48
	W3	MSHD	110	80

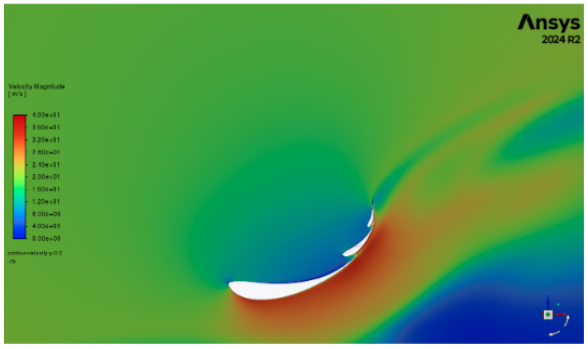


(a) Symmetry plane (b) $y = 525\,\text{mm}$

Figure 3.9: Velocity contours for the 4R1 configuration at the symmetry plane and at $y = 525\,\text{mm}$.



(a) Symmetry plane



(b) $y = 525\,\text{mm}$

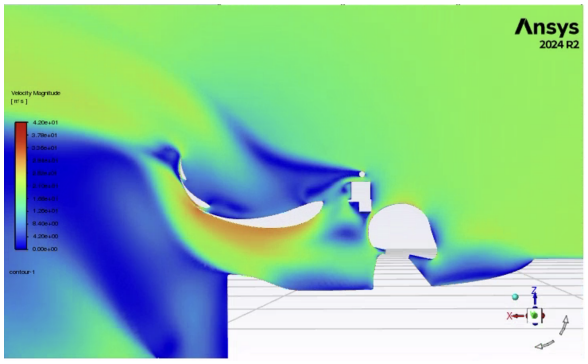
Figure 3.10: Velocity contours for the 4R2 configuration at the symmetry plane and at $y = 525\,\text{mm}$.

Table 3.3: Downforce obtained for each iteration

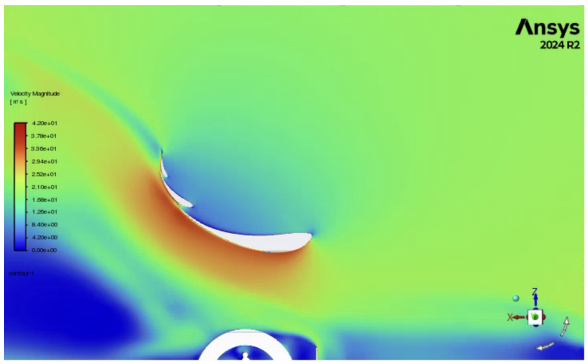
Iteration	Downforce [N]
4R1	255.18
4R2	259

For iteration `4R1`, Wing 2 clearly experiences stall. This explains the higher downforce obtained with `4R2`, where flow separation is still present but less noticeable.

The best performing iteration of the three-element configuration (`4R2`) was then selected and tested with the baseline V3 geometry. The results are shown below.

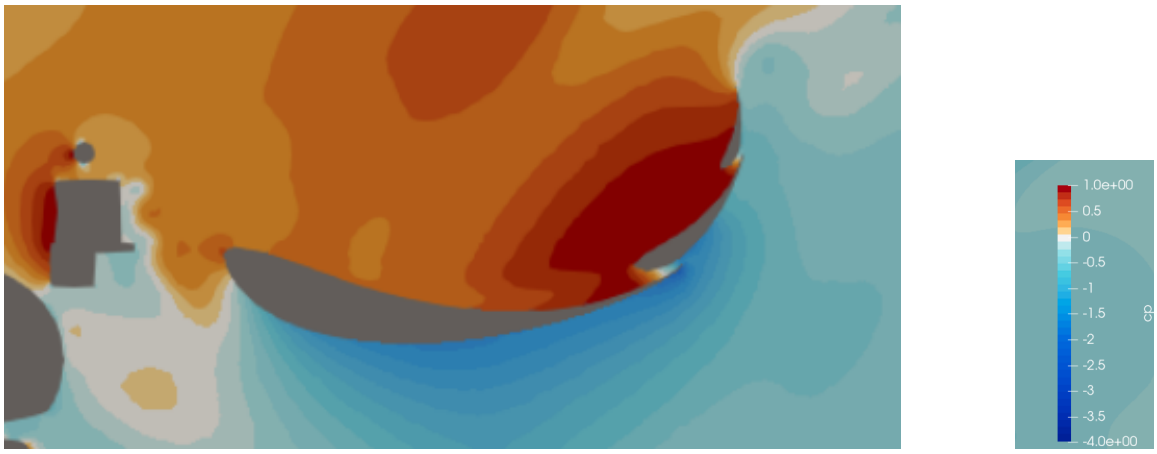


(a) Symmetry plane

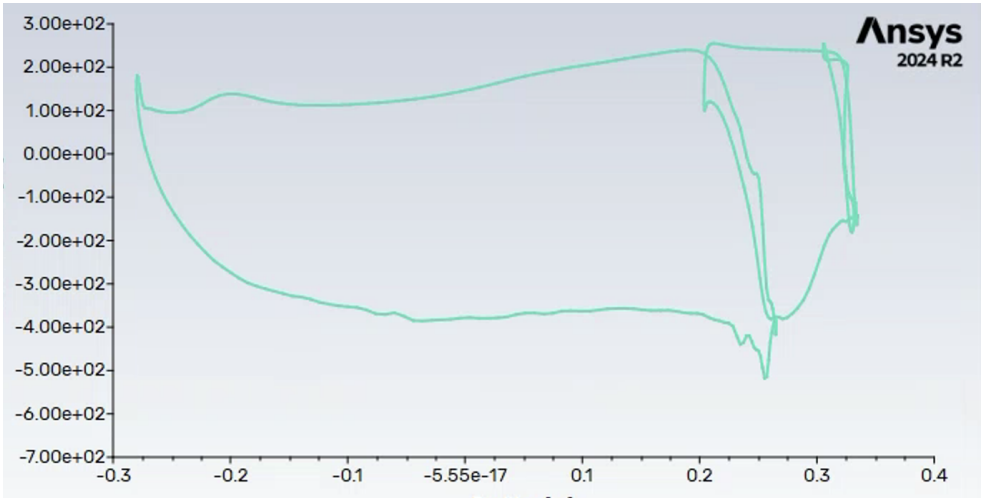


(b) $y = 525\,\text{mm}$

Figure 3.11: Velocity contours for the 4R2 configuration at the symmetry plane and at $y = 525\,\text{mm}$ with baseline V3.

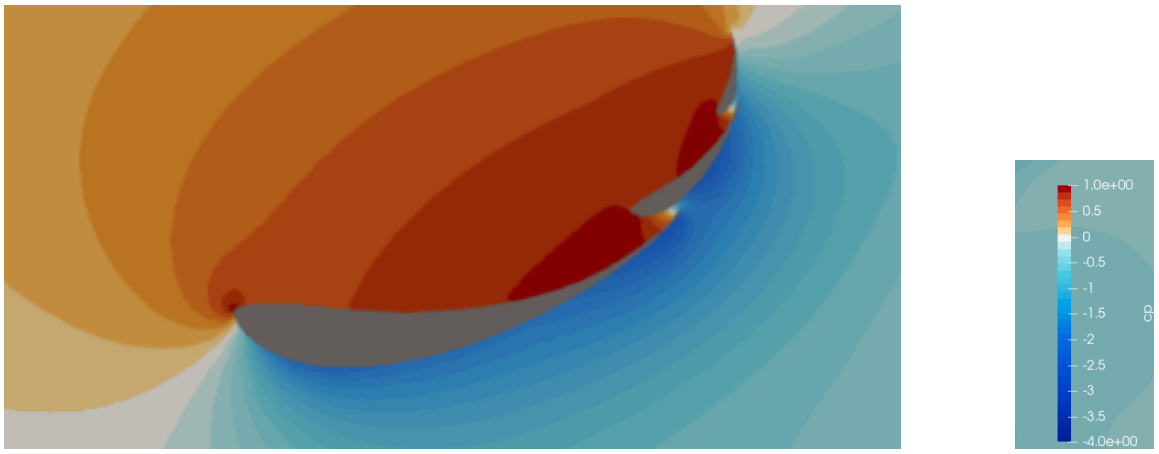


(a) Cp contour at the symmetry plane ZX

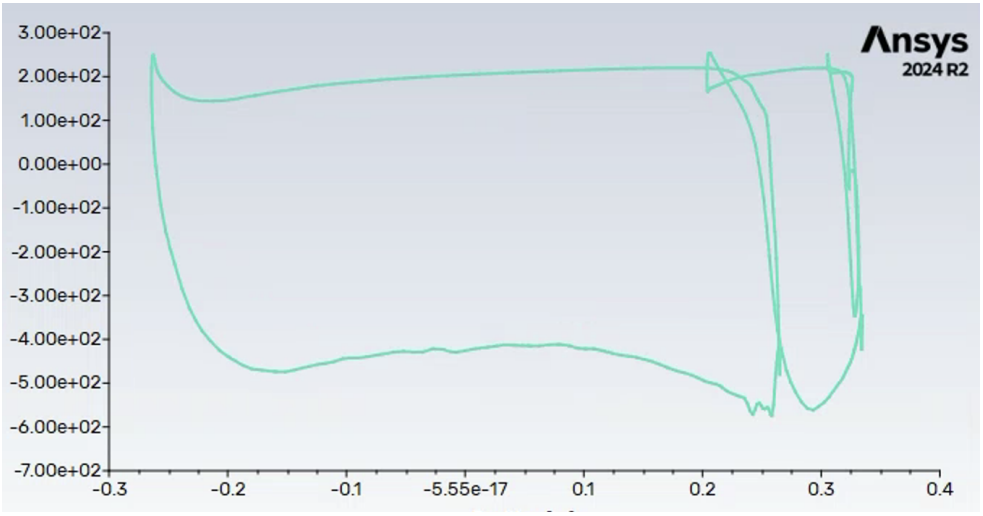


(b) Cp distribution at the symmetry plane ZX

Figure 3.12: Pressure coefficient results at the symmetry plane ZX.



(a) Cp contour at $y = 525\,\text{mm}$



(b) Cp distribution at $y = 525\,\text{mm}$

Figure 3.13: Pressure coefficient results at $y = 525\,\text{mm}$.

The downforce obtained is 256.62N slightly less than baseline V2, the main difference is the stalling of wing 2. From the figures 3.12 ad 3.13, we can see that outboard, near the endplates, the suction is stronger than inboard, it's designed like this because at the symmetry plane there's the LiDar that generates dirty air.
A larger reduction was initially expected, since the LiDAR is positioned near the symmetry plane and was anticipated to disturb the incoming flow to the rear wing more significantly.

3.5.2 Four-element configuration

To begin this phase of the design, the leading edge (LE) of Wing 1 and the trailing edge (TE) of Wing 3 were kept fixed. This ensured that the global chord of the equivalent airfoil formed by the three-element configuration was the same as that of the four-element configuration. Therefore, the main geometric change between the two configurations was the camber of the equivalent airfoil, rather than the overall chord length.

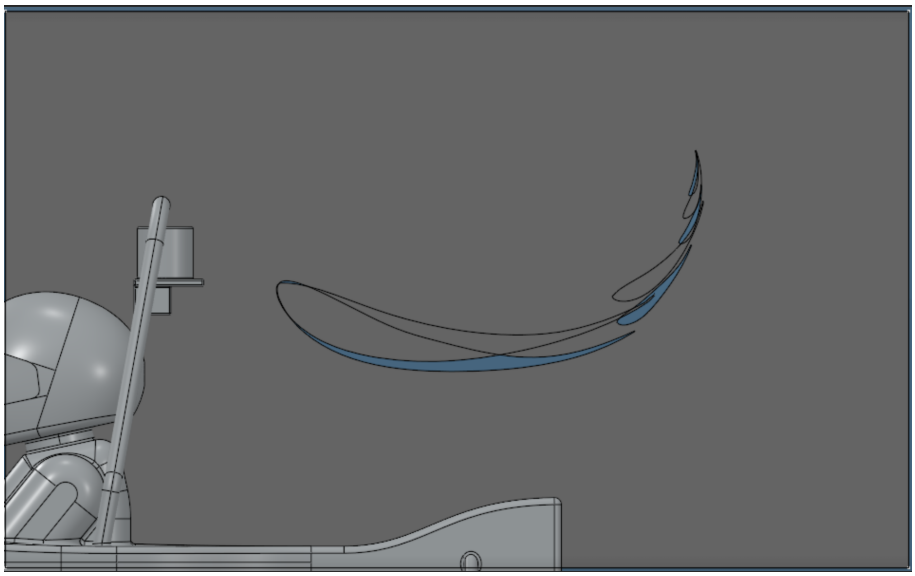


Figure 3.14: Comparison between 3 and 4 stage

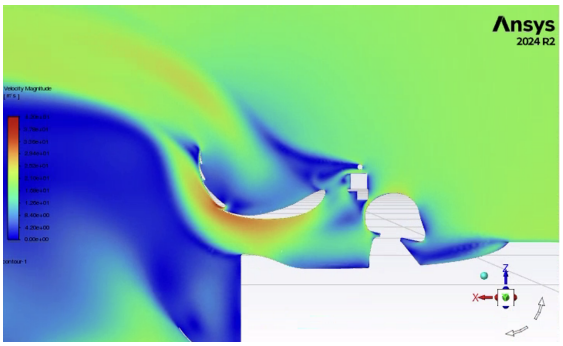
In figure 3.14, we can see in gray the section of the three stage and in blue the one of the four stage.

The first four-stage rear wing designs were created before this condition was applied. For this reason, they were not included in the final comparison, as the global chord was not kept constant and the comparison would not have been consistent. The `13RX` family was the first set of designs developed while satisfying this condition.

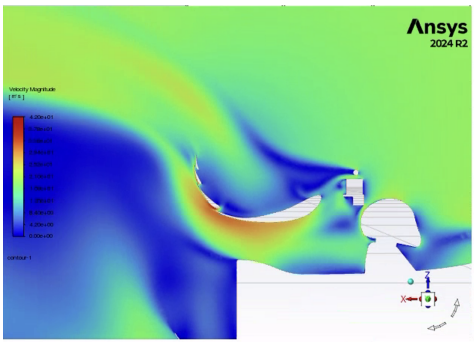
Since the MSHD profile was found to be too cambered to use on the flap elements, the S1223 airfoil was selected instead. To keep this section concise, only two iterations of the `13RX` family are discussed: one of the initial design `13R2` and the best performing iteration obtained from the design process `13R7` .

Table 3.4: Parameters of the two selected four-element wing iterations of `13RX`

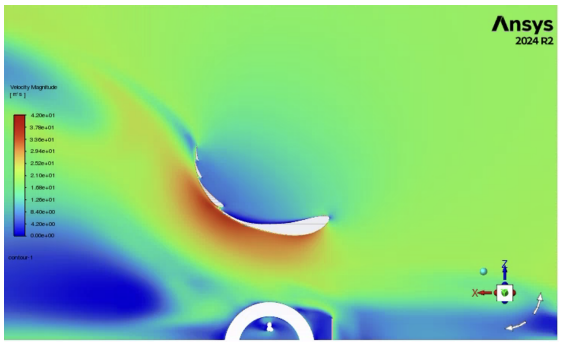
Iteration	Wing	Airfoil	Chord [mm]	AoA [°]
13R2	W1 inboard	MSHD	515	−2
	W1 outboard	MSHD	505	9
	W2	S1223	155	47
	W3	S1223	90	70
	W4	S1223	65	82
13R7	W1 inboard	MSHD	515	−5.5
	W1 outboard	MSHD	505	9
	W2	S1223	155	47
	W3	S1223	90	68
	W4	S1223	65	82



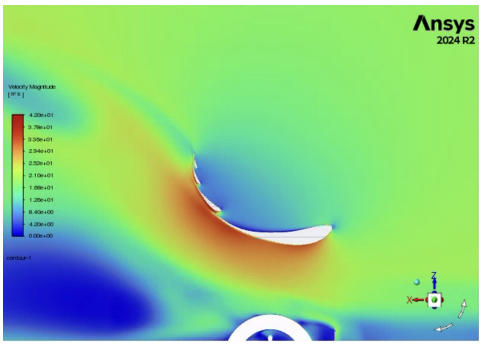
(a) `13R2`, velocity at $y = 0$



(b) `13R7`, velocity at $y = 0$

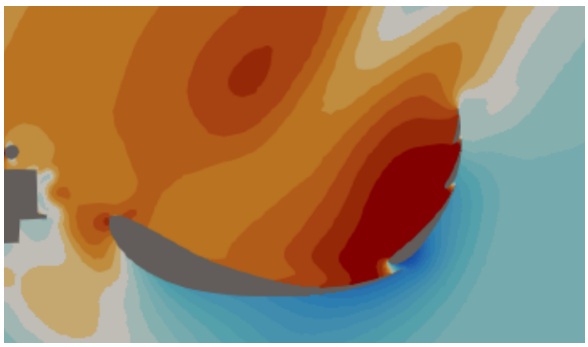


(c) `13R2`, velocity at $y = 0.525\,\mathrm{m}$

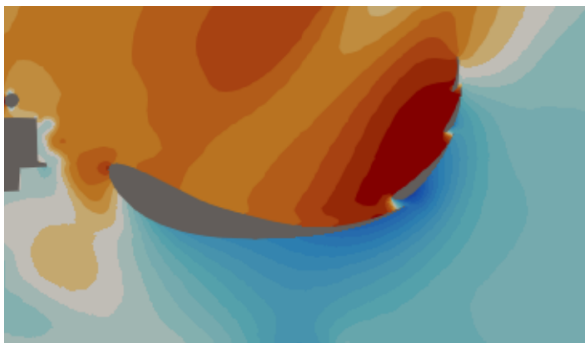


(d) `13R7`, velocity at $y = 0.525\,\mathrm{m}$

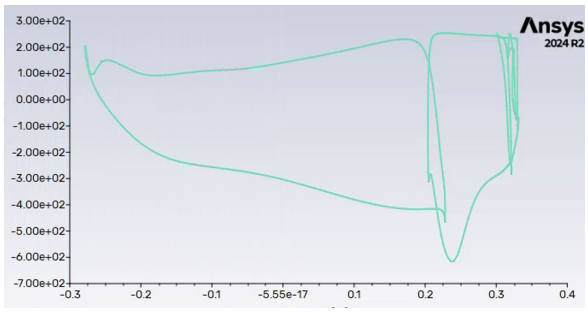
Figure 3.15: Velocity contours for `13R2` and `13R7` at the symmetry plane and near the endplates.



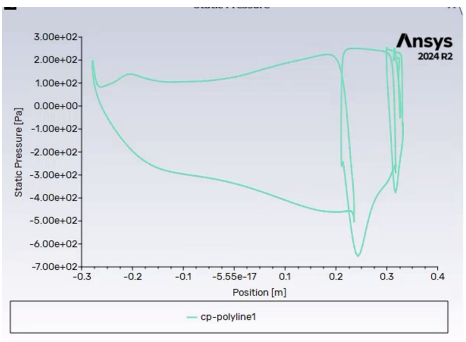
(a) `13R2`, Cp contour



(b) `13R7`, Cp contour

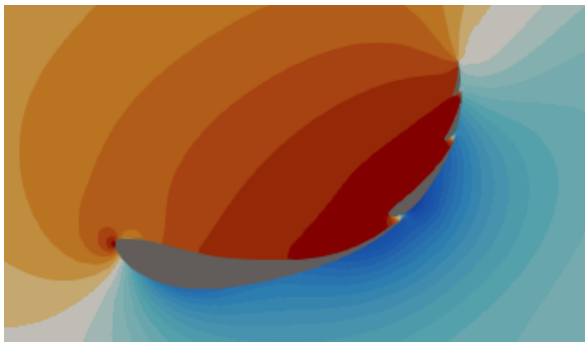


(c) `13R2`, Cp plot

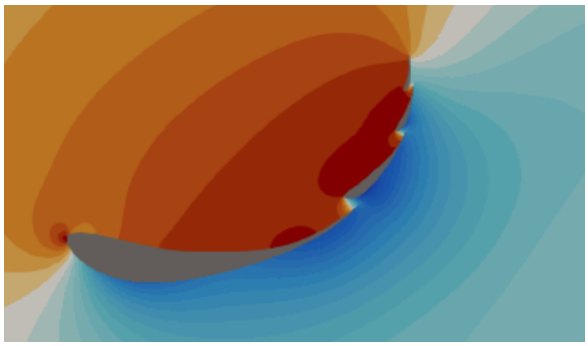


(d) `13R7`, Cp plot

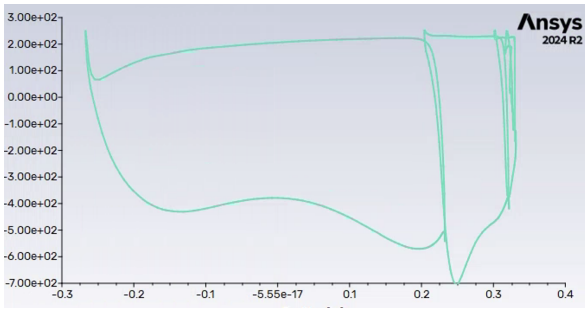
Figure 3.16: Cp comparison between `13R2` and `13R7` at the symmetry plane ($y = 0$).



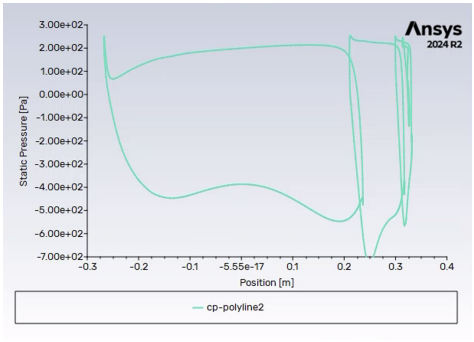
(a) `13R2`, Cp contour



(b) `13R7`, Cp contour

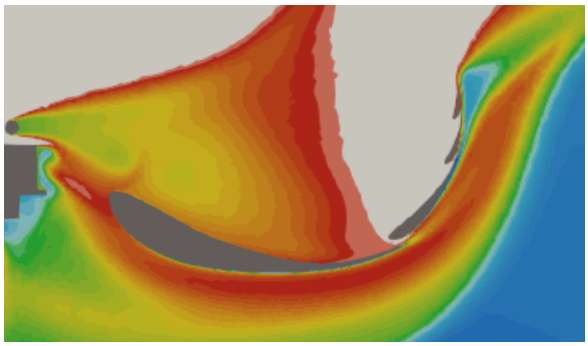


(c) `13R2`, Cp plot

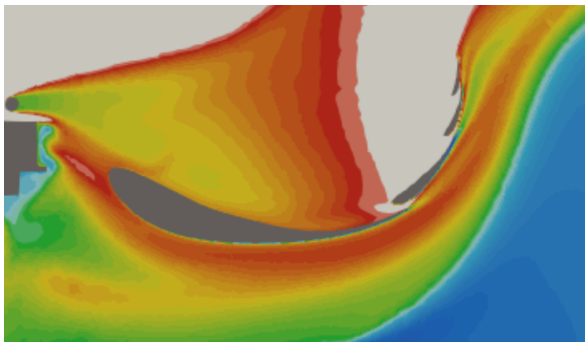


(d) `13R7`, Cp plot

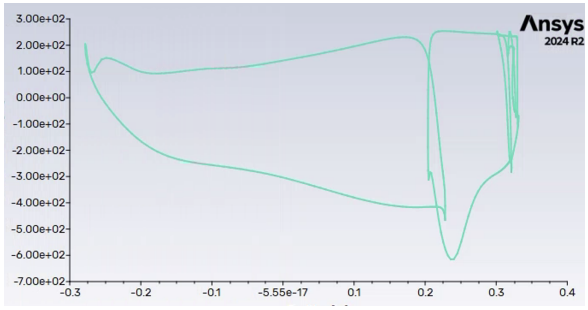
Figure 3.17: Cp between `13R2` and `13R7` near the endplates ($y = 0.525\,\mathrm{m}$).



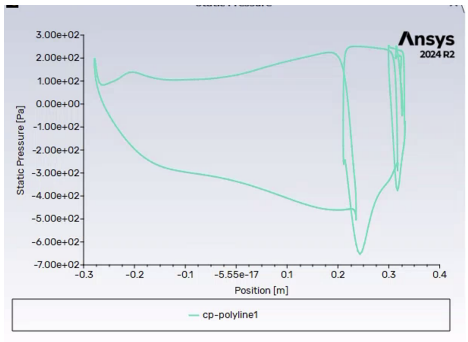
(a) `13R2`, CpT contour



(b) `13R7`, CpT contour



(c) `13R2`, Cp plot



(d) `13R7`, Cp plot

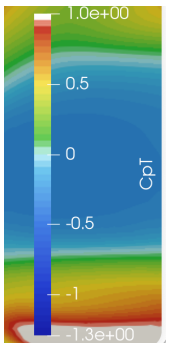
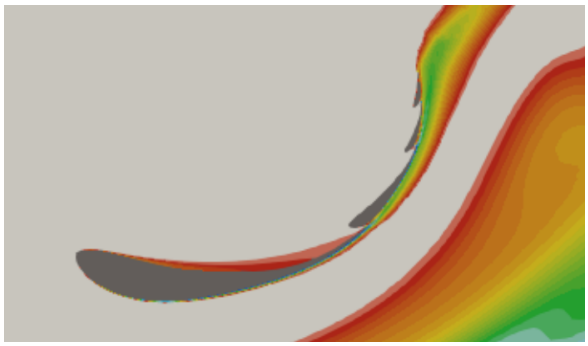
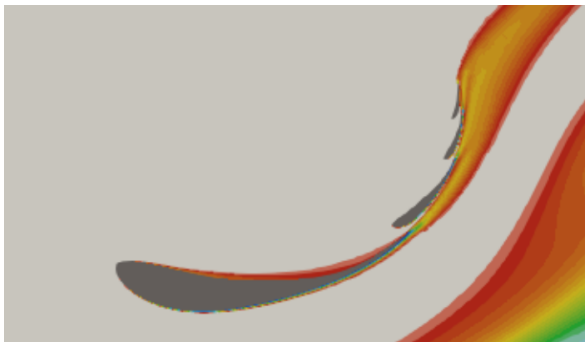


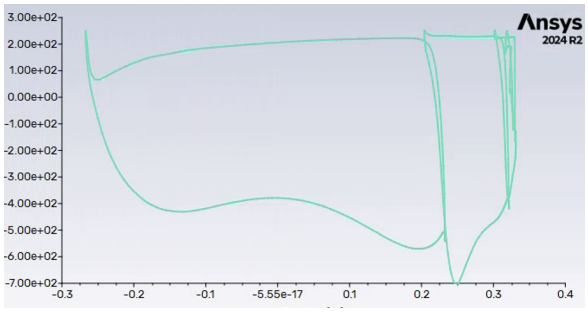
Figure 3.18: CpT comparison between `13R2` and `13R7` at the symmetry plane ($y = 0$).



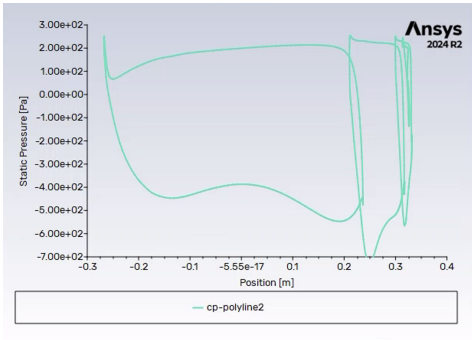
(a) `13R2`, CpT contour



(b) `13R7`, CpT contour



(c) `13R2`, Cp plot



(d) `13R7`, Cp plot

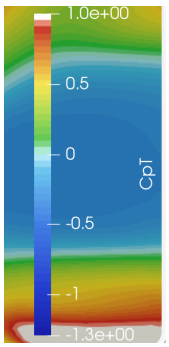


Figure 3.19: CpT comparison between `13R2` and `13R7` near the endplates ($y = 0.525\,\text{m}$).

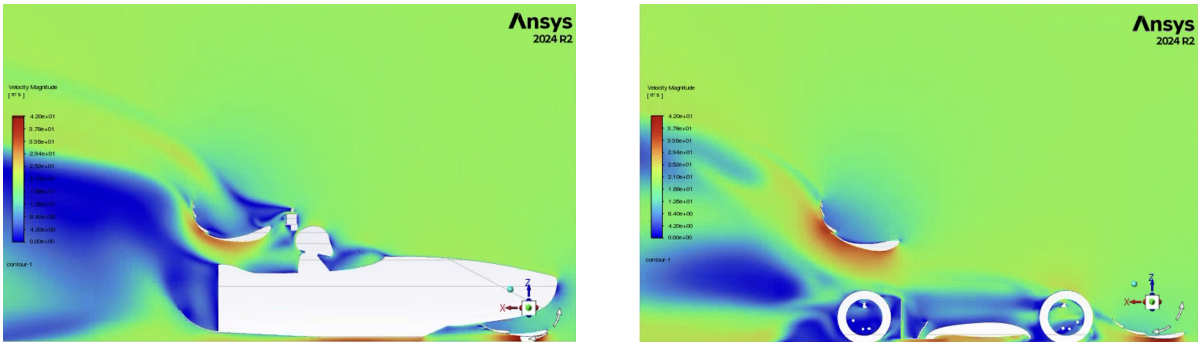
When analysing the post-processing of these two iterations, we can clearly see that one of the reasons for the downforce increase is the fact that W3 in `13R2` stalls, meaning that reducing its AoA was a good decision, as shown in Fig. 3.15. Furthermore, since the AoA of W1 was also made more negative, we can observe from Fig. 3.16 that the low-pressure region on the suction side of W1 is more pronounced in `13R7`, indicating that it may be generating more downforce compared to `13R2`.

The next family explored was `15RX`, where the only change made was replacing the S1223 airfoil on W2 with the MSHD profile.

The first iteration that we will analyse is `15R2`, which produced downforce result close to `13R7`.

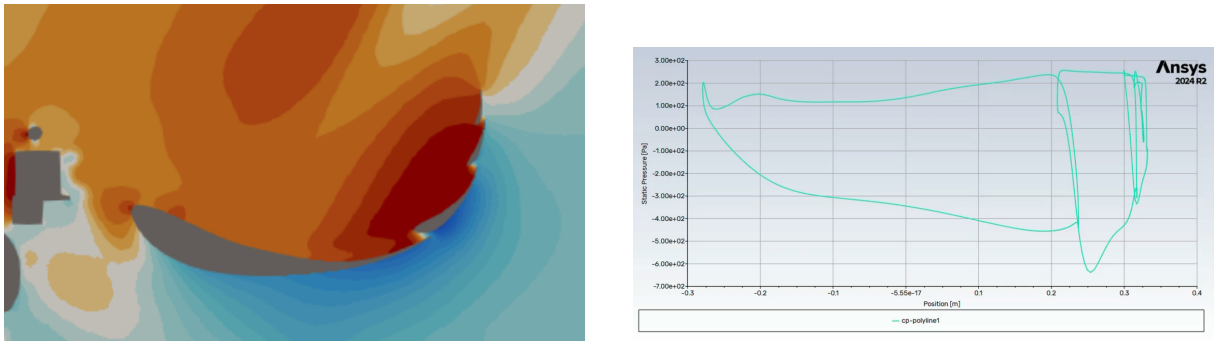
Table 3.5: Parameters of `15R2`

Iteration	Wing	Airfoil	Chord [mm]	AoA [°]
15R2	W1 inboard	MSHD	515	−3.5
	W1 outboard	MSHD	505	9
	W2	MSHD	155	47
	W3	S1223	90	70
	W4	S1223	65	82



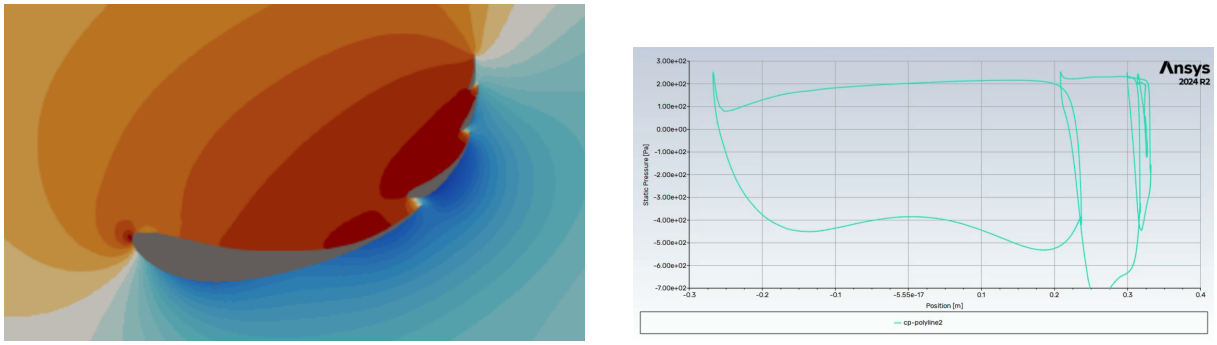
(a) `15R2`, velocity at $y = 0$ (b) `15R2`, velocity at $y = 0.525\,\mathrm{m}$

Figure 3.20: Velocity contours for `15R2` at the symmetry plane and near the endplates.



(a) `15R2`, Cp contour (b) `15R2`, Cp plot

Figure 3.21: Cp comparison for `15R2` at the symmetry plane ($y = 0$).



(a) `15R2`, Cp contour (b) `15R2`, Cp plot

Figure 3.22: Cp comparison for `15R2` near the endplates ($y = 0.525\,\mathrm{m}$).

Changing the second flap from S1223 to MSHD led to a more performant design, even if the downforce produced was still the same (257N). As seen in Fig. 3.22, the pressure on W2 drops lower than in `13R7` (Fig. 3.17d), indicating that the suction side of W2 is working more effectively with the MSHD profile. This suggests that `15R2` is a promising base for further optimisation, and the following iterations will focus on improving it. Looking at Table 3.8, one direction explored was splitting W1, effectively reducing its chord while redistributing it across the three flaps. This leads to the `15R7` iteration, whose parameters are shown in Table 3.6.

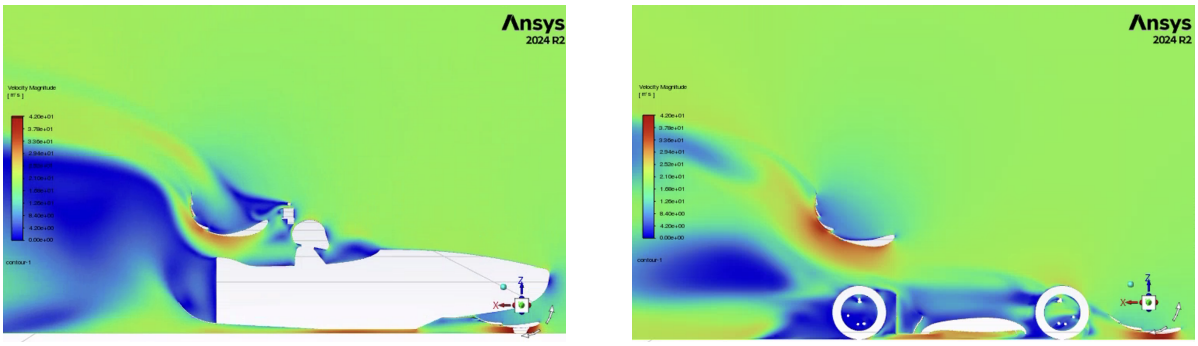
Table 3.6: Parameters of `15R7`

Iteration	Wing	Airfoil	Chord [mm]	AoA [°]
15R7	W1 inboard	MSHD	490	−7
	W1 outboard	MSHD	480	6
	W2	MSHD	155	39
	W3	S1223	100	68
	W4	S1223	85	84



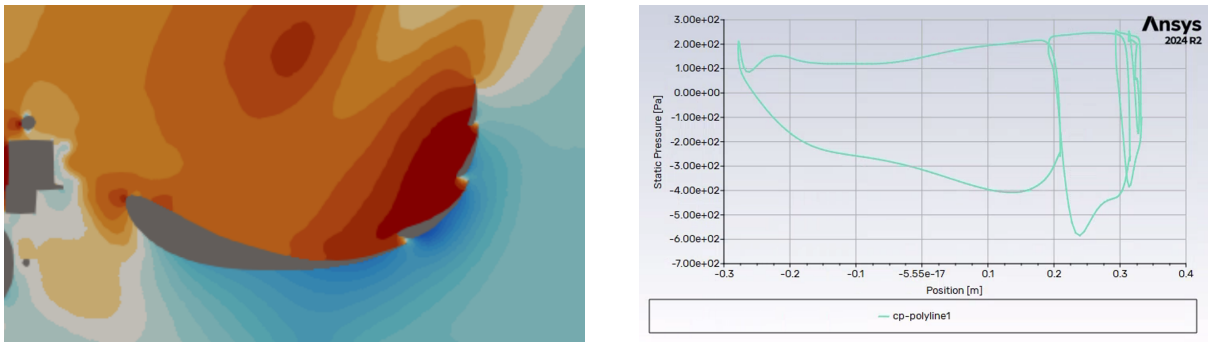
(a) Comparison at $y = 0$ (b) Comparison at $y = 0.525\,\text{m}$

Figure 3.23: comparison between `15R2` and `15R7` at the symmetry plane and near the endplates(blue 15R7 and grey 15R2).



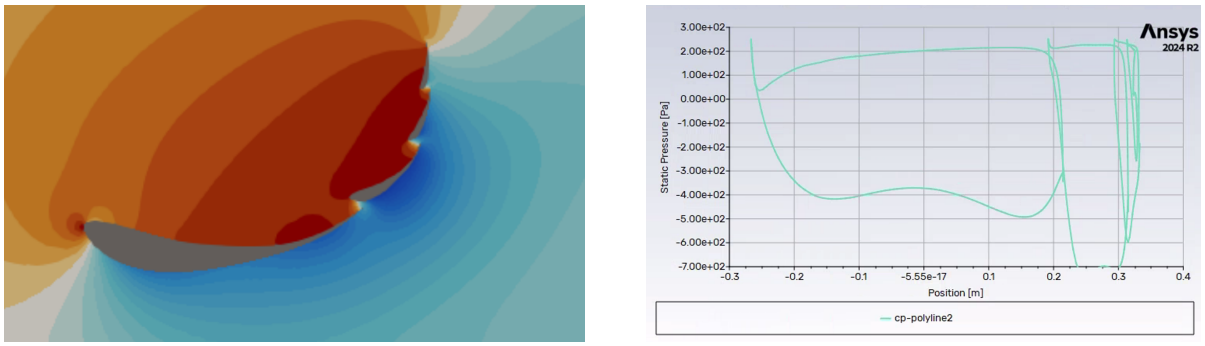
(a) `15R7`, velocity at $y = 0$ (b) `15R7`, velocity at $y = 0.525\,\text{m}$

Figure 3.24: Velocity contours for `15R7` at the symmetry plane and near the endplates.



(a) `15R7`, Cp contour (b) `15R7`, Cp plot

Figure 3.25: Cp comparison for `15R7` at the symmetry plane ($y = 0$).



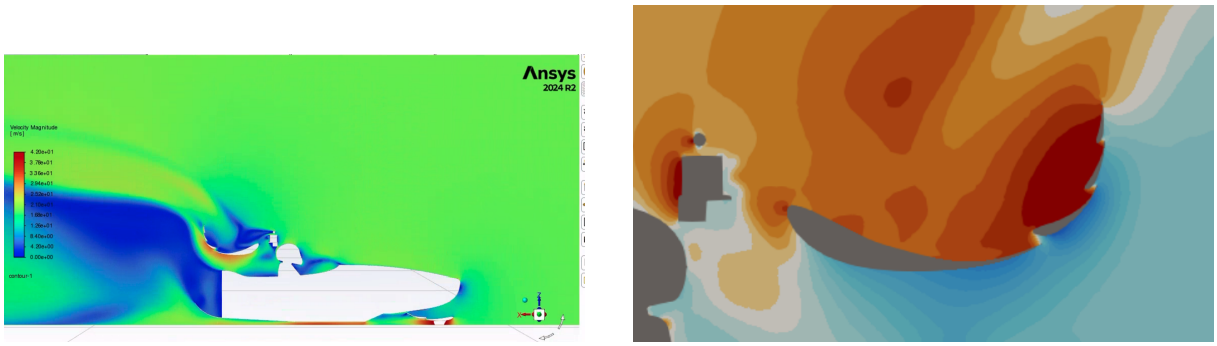
(a) `15R7`, Cp contour (b) `15R7`, Cp plot

Figure 3.26: Cp comparison for `15R7` near the endplates ($y = 0.525\,\text{m}$).

`15R7` only produced 247N of downforce, which is lower than the 257N achieved by `15R2`. We can see in figure 3.26 that Wing 2 is overloaded which is not the purpose of a flap. Looking at figure 3.24a, W2 stalls on the pressure side while W1 stalls near its trailing edge. `15R11` addresses these issues by revising the angles of attack of both elements.

Table 3.7: Parameters of `15R7`

Iteration	Wing	Airfoil	Chord [mm]	AoA [°]
	W1 inboard	MSHD	490	−6
	W1 outboard	MSHD	480	8
15R11	W2	MSHD	155	38
	W3	S1223	100	68
	W4	S1223	85	84



(a) `15R7`, velocity at $y = 0$ (b) `15R7`, Cp contour at $y = 0$

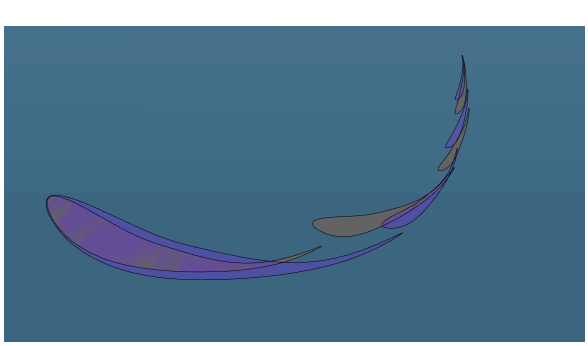
Figure 3.27: Velocity and Cp contours for `15R11` at the symmetry plane.

Figure 3.27 shows that the pressure on the pressure side of W2 is more uniform in `15R11` compared to `15R7`, indicating that the stall is partially recovered, which explains the increase in downforce from 248N to 253N.

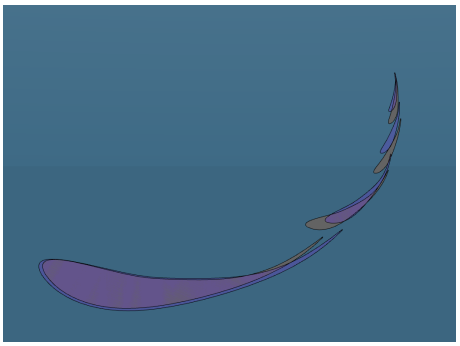
Since the issues were mainly concentrated inboard, a lot of iterations were focused on solving them inboard while keeping the outboard section constant. The best result from this process was `15R19`, which was a combination of reducing significally chord W1 and playing with AoA.

Table 3.8: Parameters of `15R19`

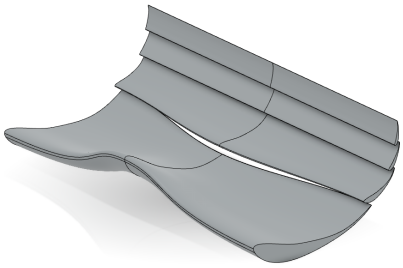
Iteration	Wing	Airfoil	Chord [mm]	AoA [°]
15R19	W1 inboard	MSHD	400	-8
	W1 outboard	MSHD	480	8
	W2 inboard	MSHD	220	23
	W2 outboard	MSHD	155	38
	W3	S1223	100	64
	W4	S1223	85	84



(a) Comparison at $y = 0$

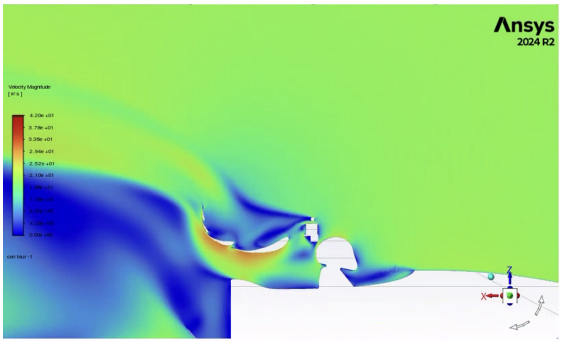


(b) Comparison at $y = 0.525\,\mathrm{m}$

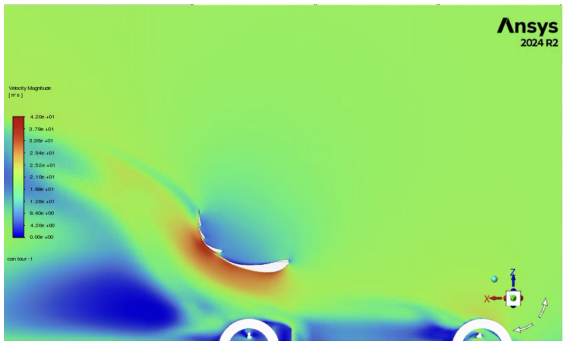


(c) `15R19` view

Figure 3.28: View of `15R19` and Cp comparison between `15R19` and `15R2` at the symmetry plane and near the endplates.

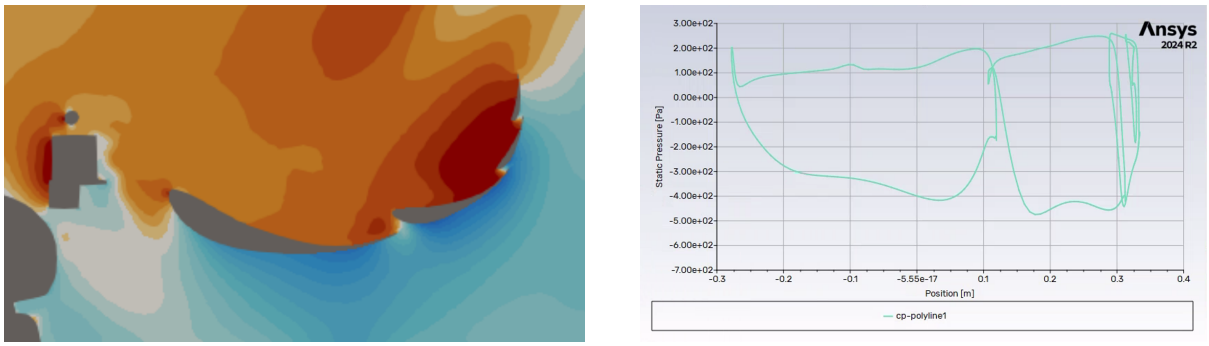


(a) `15R19`, velocity at $y = 0$



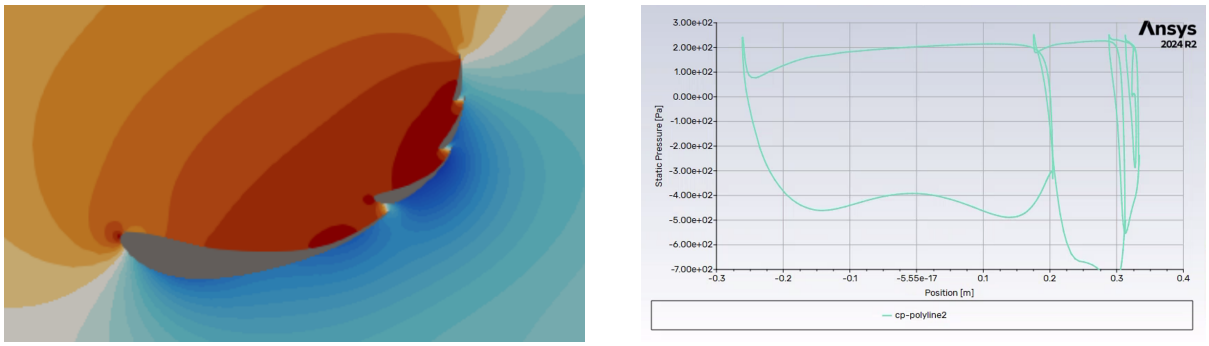
(b) `15R19`, velocity at $y = 0.525\,\mathrm{m}$

Figure 3.29: Velocity contours for `15R19` at the symmetry plane and near the endplates.



(a) `15R19`, Cp contour (b) `15R19`, Cp plot

Figure 3.30: Cp comparison for `15R19` at the symmetry plane ($y = 0$).



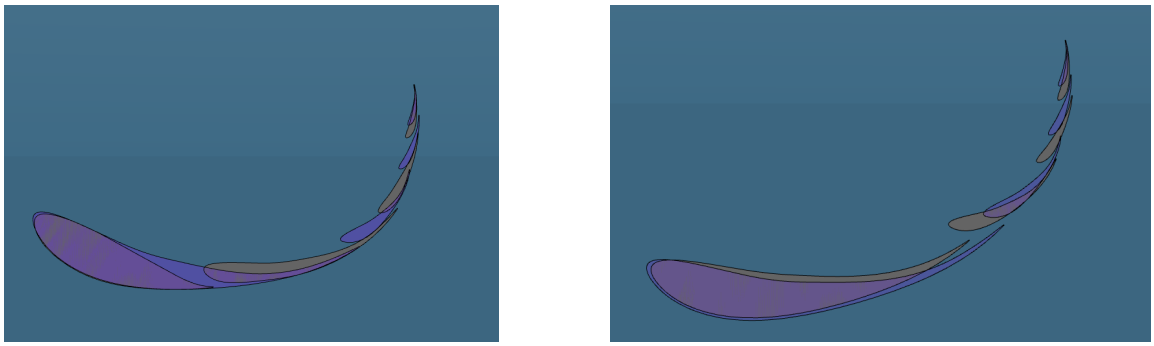
(a) `15R19`, Cp contour (b) `15R19`, Cp plot

Figure 3.31: Cp comparison for `15R19` near the endplates ($y = 0.525\,\text{m}$).

As seen in Fig. 3.29a, the issues on the pressure side are now partly resolved and W2 is less loaded inboard. However, the problem outboard remains and even got worse, as seen in Fig. 3.30 and 3.31. Despite this, the design managed to produce 259N of downforce.

Another family `16RX` was also explored where the trailing edge of W4 is no longer fixed and was moved back. For the sake of keeping the report concise, it will not be discussed further as the best iteration only produced 258,N of downforce, which is approximately the same as `15R19`.

The last iteration explored was `15R23`, which introduced two new airfoils for W1 inboard and W2 inboard. Unfortunately, due to a surface mesh issue, the simulation could not be completed and only the concept will be presented here.



(a) Comparison at $y = 0$ (b) Comparison at $y = 0.525\,\text{m}$

Figure 3.32: Comparison between `15R23` and `15R2` at the symmetry plane and near the endplates.

As seen in Fig. 3.32a, the airfoil profile of W1 was designed to cover the entire front area of W1 in `15R2`. Looking at the porcupine analysis of this airfoil in Fig. 3.33, we can already note several things.

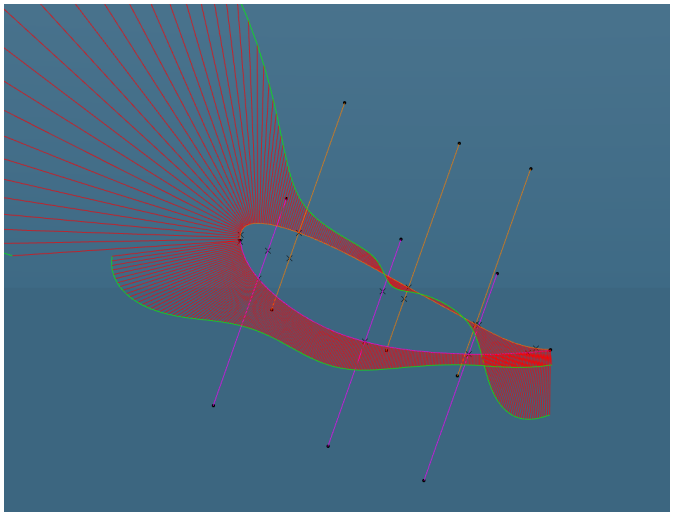


Figure 3.33: Porcupine Analysis Wing 1 Inboard

- The suction side looks promising but could of course be better designed. A large peak is visible right at the leading edge, followed by a constant recovery at the end. The middle part could use some changes.
- The pressure side is where work needs to be done, as it changes concavity three times, which is not what we are looking for.

3.6 Results and Discussion

Table 3.9 summarises the downforce produced by each iteration discussed in this chapter.

Table 3.9: Downforce summary across all iterations

Family	Iteration	Downforce [N]
Three-element	4R1	255.18
	4R2 (V2)	259
	4R2 (V3)	256.62
13RX	13R2	253
	13R7	257
15RX	15R2	257
	15R7	247
	15R11	253
	15R19	259
16RX	best iteration	258

Several observations can be drawn from these results. First, the transition from three to four elements did not produce a clear improvement in downforce, with the best four-element iteration `15R19` only marginally outperforming the three-element `4R2`. This suggests that simply adding an element is not sufficient on its own, and that the distribution of load across the elements plays a critical role.

The `13RX` family showed that stall management is key reducing the AoA of W3 in `13R7` immediately recovered performance. The switch to MSHD on W2 in the `15RX` family was a promising direction, as the suction side of W2 showed better behaviour, but the inboard loading issues on W2 proved difficult to resolve. `15R19` managed to partly address the inboard problems through a combination of chord reduction on W1 and AoA adjustments, achieving the best result of 259 N, but the outboard section remained problematic.

Overall, the results highlight that designing a four-element rear wing is a complex task between loading each element effectively without pushing any of them into stall, and that further work is needed to fully exploit the potential of the four-element configuration.

3.7 Conclusion

Designing a rear wing is an iterative process, there is no magic formula to make it work. It is through trial and error, analysing what works and pushing it further, that progress is made. This is only possible thanks to a post-processing workflow that allows each iteration to be understood and improved upon. Unfortunately, despite the numerous iterations explored, a four-element configuration that outperforms the three-element design in terms of downforce could not be achieved within the scope of this project. For future work, one idea would be to maximise the global chord by placing the leading edge of W1 at the lowest and furthest forward position allowed by the regulations, and the trailing edge of W4 at the highest and furthest rearward position. Another direction would be to make more use of porcupine analysis to better understand the pressure recovery and guide the choice of camber and thickness of each airfoil. Finally, adding extra wing elements in front of or below the main stack could also be explored to further increase downforce.

Chapter 4

Front Wing

4.1 Introduction

4.1.1 Context

The general introduction has established that the aerodynamic package of the car is developed with as much downforce possible as the primary objective and drag as a secondary cost. Within that package, the front wing plays a particular role. It is the first surface the flow encounters, it sets the boundary conditions for every device downstream (diffuser, side structures, chassis winglets, wishbone covers, rear wing), and it is the primary lever for tuning the longitudinal distribution of downforce between the front and rear axles. A change in front-wing geometry therefore shifts the load on both axles; directly on the front, and indirectly on the rear through the inflow it sets for every device downstream.

The wing is designed against the draft 2027 Formula Student Germany (FSG) ruleset [1], which differs from the current rules in three ways that matter specifically for the front wing: a taller envelope forward of the front tires, an unchanged forward length limit, and a new designer-placeable keep-out zone. The detailed envelope is examined in the next section. At the time of writing, this ruleset is published only in draft form (FSG, 25.10.2025, *Draft Only, Subject to Changes*). The final ruleset is expected in September or October 2026, the wing is designed to require only minimal modification once it is finalized, before manufacture.

4.1.2 Problem statement

The relevant performance quantities are the total downforce coefficient of the car, C_{zt}, and the share of that downforce carried by the front axle, C_{zf}/C_{zt}, defined as the aerodynamic *Front Balance*. C_z values are defined in the SAE vehicle-axis convention [22] (z positive downward, so $C_z > 0$ corresponds to downforce) and are normalized by the same fixed reference area, with $C_{zt} = C_{zf} + C_{zr}$. This notation is used throughout the combined report.

For the 2027 car, the Vehicle Dynamics team has set a target of 55 % Front Balance, derived from the weight distribution and other vehicle-level considerations developed in parallel with this project. The front wing must therefore be designed to deliver as much total downforce as possible while landing on this Front Balance figure, under the geometric constraints of the draft 2027 ruleset.

The project addresses the following question:

Research question

How can a Formula Student front wing be designed to reach 55 % Front Balance with maximum downforce, under the evolving 2027 rules?

4.1.3 Objectives and approach

This work has two deliverables. The first is a design methodology built on two complementary tools: CFD post-processing of each candidate, used to identify where each wing can be improved, and a C_{zt} versus Front Balance representation, used to compare candidates against each other and make the downforce/balance trade-off explicit. The second is the front-wing geometry that this methodology converges on, evaluated on the latest whole-car baseline available at the time of writing. The path from the first to the second is reported as an iteration history, told as a reasoned narrative: what was learned at each step, why each move was made, and which paths were abandoned, including the dead ends.

The design is conducted in two phases. The first phase, which constitutes the bulk of the work, maximizes the downforce produced at the front, C_{zf}: the effect on C_{zt} and on Front Balance is monitored but not enforced, and the T2.1.4 keep-out zones are not yet applied, with the goal being to establish how much load the front wing can carry before any trade is made. The second phase brings the wing into rule compliance and to the 55 % Front Balance target, using whatever geometric levers are needed (flap angle-of-attack changes, element repositioning, addition or removal of elements, application of the T2.1.4 keep-out zones), accepting controlled C_{zt} loss as the cost of hitting both. This ordering means the front wing is first pushed to its physical limit, then pulled back only as much as the rules and the balance target require.

The front wing is not designed in isolation. The team works from a shared whole-car baseline (geometry, current aero kit, vehicle setup) that is updated every two to three weeks. At each baseline release, the best front-wing iteration from the previous baseline is carried forward, re-run on the new car, and used as the anchor point from which the next round of iterations diverges. This produces a sequence of iteration families on the C_{zt} vs. Front Balance plot, each anchored on a different baseline version. The work reported here spans baselines V2 through V5, with V3 and V4 carrying the largest number of iterations and V5 being the most recent baseline against which the design is converged at the time of writing.

A practical note on the target: 55 % Front Balance is treated as the nominal design point of the wing, not as a fixed operating condition. The car is intended to carry an adjustable-flap mechanism that allows the Front Balance to be tuned by a few percent around the nominal value at the track, accommodating circuit-specific setup choices and driver preference. The mechanism is being developed in parallel and is not part of the present report; the wing geometry is, however, designed so that the 55 % point sits within the middle of the achievable adjustment range rather than at one of its extremes, leaving headroom in both directions.

4.2 Constraints and design space

This section defines the geometric envelope inside which the front wing must live. Two rulesets are relevant: the current FSG rules [20], under which all cars built before 2027

must comply, and the draft 2027 ruleset [1], against which the present wing is designed. Only the rules that apply specifically to the front wing are reproduced here; the broader regulatory context is given in the general introduction. The section closes with the resulting design envelope.

4.2.1 Current FSG rules

The current ruleset constrains the front wing through three families of articles: a generic open-wheel keep-out zone (T2.1.3), a global ground-clearance limit (T2.2.1), and the aerodynamic-device envelope (T8.2.1 to T8.2.4).

The generic open-wheel keep-out zone, T2.1.3, defines a region around each front and rear tire that no part of the vehicle may enter. Viewed from the side, the region extends 75 mm in front of and 75 mm behind the outer diameter of the tire, between the outside and inboard planes of the wheel/tire assembly (Figure 4.1). For the front wing this fixes the rearward extent of any element that lies laterally over the tire.

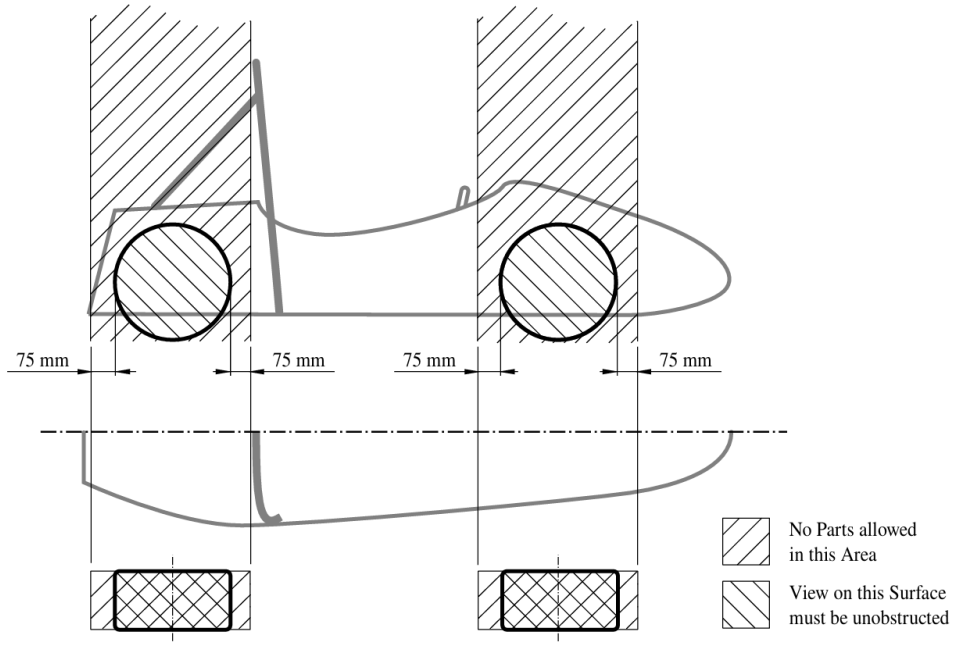


Figure 4.1: Generic open-wheel keep-out zones around the front and rear tires (T2.1.3). The 75 mm regions in front of and behind each tire must remain free of bodywork, and the view onto the tire surface must be unobstructed. Reproduced from [20].

The ground-clearance rule, T2.2.1, sets a minimum static clearance of 30 mm for any portion of the vehicle other than the tires. Because downforce on a ground-effect-sensitive element increases as the gap to the floor closes, the wing is designed at this minimum: the lowest point of the mainplane sits at 30 mm above the reference surface, and the geometry is checked under suspension travel to ensure no other point of the wing falls below the limit.

The aerodynamic-device envelope is set by T8.2. Three sub-rules apply to the front wing:

- **T8.2.1, height.** All aerodynamic devices in front of the front axle and extending further outboard than the most inboard point of the front tire must be lower than

250 mm from the ground. Devices forward of the head-restraint plane must remain below 500 mm. The outboard 250 mm limit is the binding constraint for the front wing.

- **T8.2.2, width.** Below 500 mm in height, devices may not extend outboard of the outermost plane of the front and rear wheels.
- **T8.2.3, length.** Devices may not extend further forward than 700 mm from the front of the front tires.

Rule T8.2.4 collects these limits into a single envelope diagram (Figure 4.2). The diagram is the standard reference for inspection and is reproduced here in the form published with the 2026 rules.

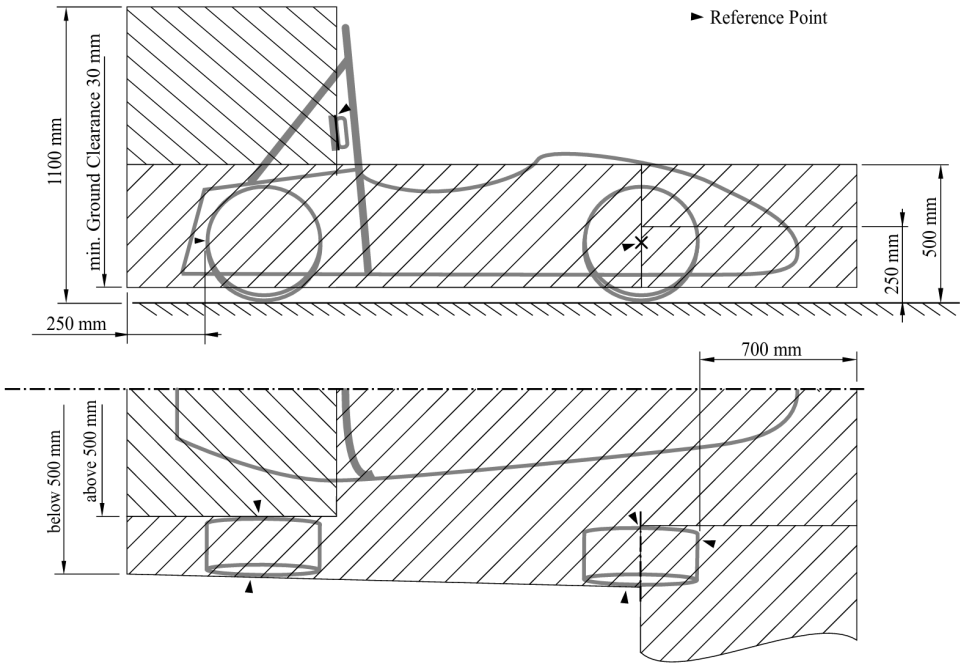


Figure 4.2: Current FSG envelope for aerodynamic devices, consolidated by rule T8.2.4. Side view (top) shows the 250 mm outboard and 500 mm inboard height limits, the 1100 mm overall height, and the 30 mm minimum ground clearance. Top view (bottom) shows the 700 mm forward limit and the laterally bounded region inboard of the front tire. Reproduced from [20].

The current-rules envelope for the front wing is therefore a region approximately 700 mm long forward of the front tires, 250 mm tall in the outboard region, 500 mm tall in the inboard region, and bounded laterally by the front-wheel outboard plane.

4.2.2 Draft 2027 ruleset

The draft 2027 ruleset, published on 25.10.2025 and marked *Draft Only, Subject to Changes* [1], affects the front-wing envelope through three articles: two introduce changes, a taller outboard height limit and a new variable keep-out zone, while the third retains the existing forward length limit.

T8.2.1, height. The outboard 250 mm limit is replaced by a 350 mm limit applied to all aerodynamic devices forward of a vertical plane through the leading edge of the front tires (Figure 4.3).

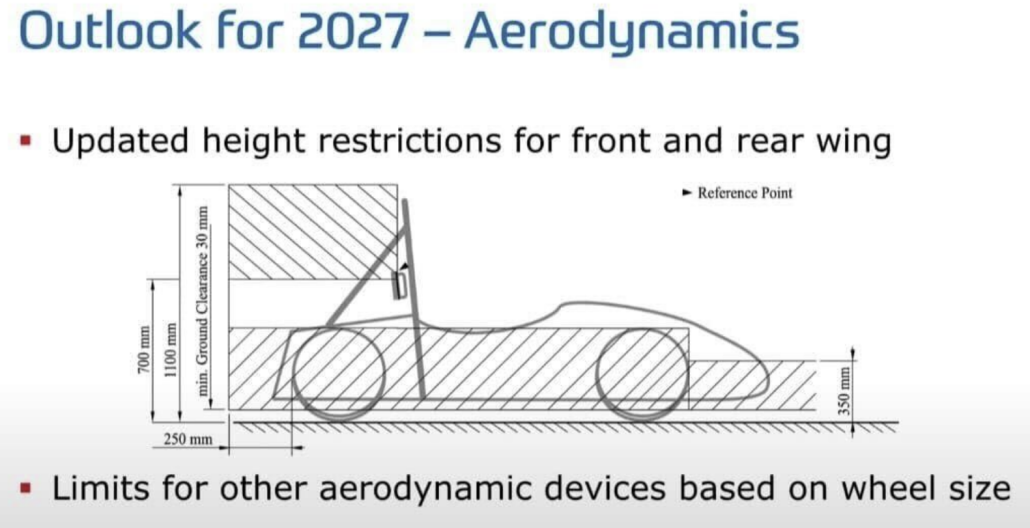


Figure 4.3: Draft 2027 height restriction for aerodynamic devices, replacing the outboard 250 mm limit of T8.2.1 with a single 350 mm limit applied forward of the front-tire leading edge. Reproduced from the FSG Academy outlook slides, 25 October 2025 [1].

T8.2.3, length. The 700 mm forward limit is retained. The forward extent of the wing is unchanged between rulesets.

T2.1.4, variable keep-out zone. A new rule introduces a variable keep-out zone in front of the front tires, separate from the generic T2.1.3 KOZ. It is defined by two rectangular cuboids, each with a 75 mm by 250 mm footprint on the ground and infinite extent upward, with the 250 mm dimension aligned with the longitudinal axis of the car. The two cuboids are placed by the designer and must not extend outboard of the most outboard point of the front wheel/tire. They do not have to intersect or sit symmetrically, leaving the team free to choose where in front of each tire the KOZ is placed (Figure 4.4).

4.2.3 Resulting design envelope

Combining the rule envelope and the architectural choices that hold across the iteration history, the front wing under development is a multi-element wing approximately 700 mm long in the streamwise direction (set by T8.2.3), 30 mm above the ground at its lowest point (set by T2.2.1), and with a maximum height of 350 mm above the ground in the outboard region under the draft 2027 rules (set by T8.2.1). Laterally it spans the full width allowed by T8.2.2 and respects the variable KOZ of T2.1.4 by placing the two cuboids in positions that do not interfere with the wing surface. The design space inside this envelope is large; the work of the following sections is to navigate it efficiently.

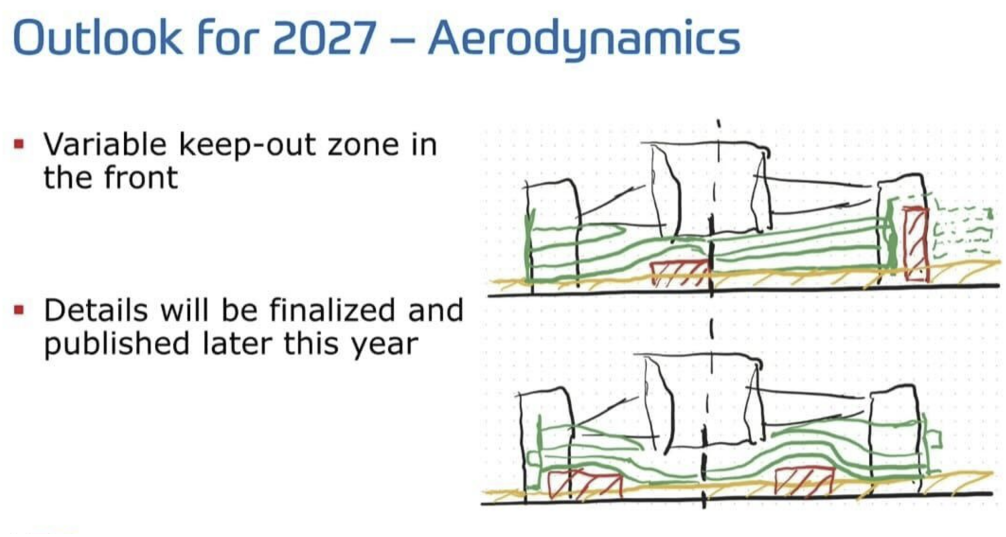


Figure 4.4: Draft 2027 variable keep-out zone (T2.1.4): two designer-placed cuboids forward of the front tires. Reproduced from the FSG Academy outlook slides, 25 October 2025 [1].

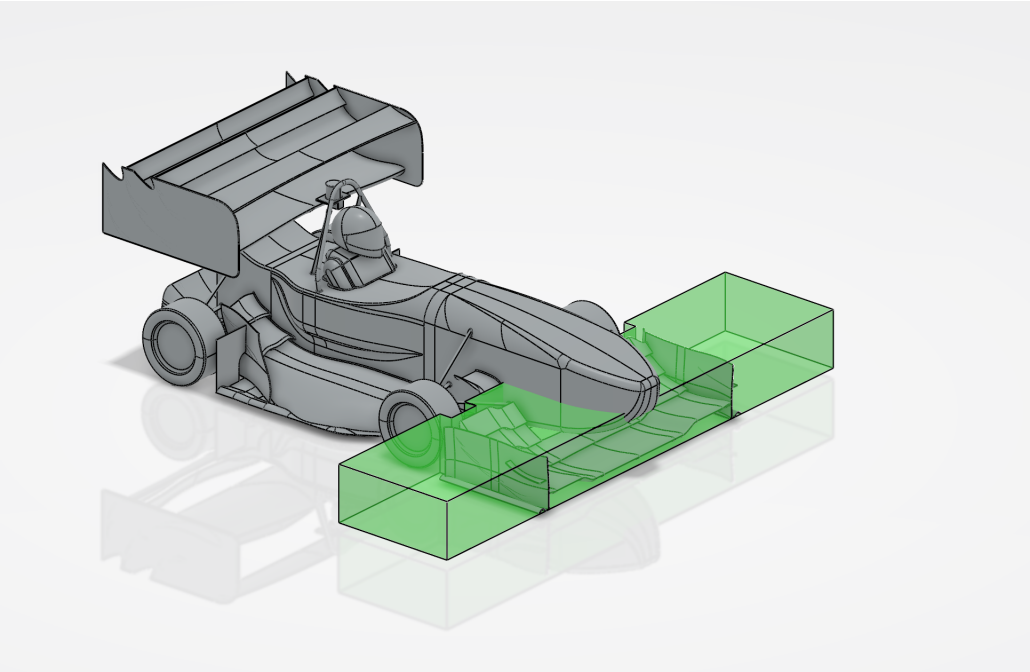


Figure 4.5: Resulting design envelope on the V4 car: front-wing geometry shown together with the two T2.1.4 keep-out cuboids and the rule envelope volume. The wing surface lies entirely inside the envelope and clear of the cuboids.

Constraint	Current rules [20]	Draft 2027 [1]
Outboard height (T8.2.1)	250 mm forward of front axle	350 mm forward of front-tire leading edge
Inboard height (T8.2.1)	500 mm	500 mm (unchanged)
Forward length (T8.2.3)	700 mm	700 mm (unchanged)
Lateral width (T8.2.2)	Front-wheel outboard plane	Front-wheel outboard plane (unchanged)
Ground clearance (T2.2.1)	30 mm minimum	30 mm minimum (unchanged)
Open-wheel KOZ (T2.1.3)	75 mm fore/aft of each tire	75 mm fore/aft of each tire (unchanged)
Variable KOZ (T2.1.4)	None	Two designer-placed cuboids, 75 mm × 250 mm, in front of front tires

Table 4.1: Front-wing envelope under the current FSG rules and the draft 2027 ruleset. The draft changes the outboard height limit and adds the variable keep-out zone; the remaining constraints are retained.

4.2.4 Manufacturability

The envelope above is the geometric limit the rules allow; what the team can actually build inside it is further constrained by manufacturing. The wing is produced in-house with finite tooling accuracy and a finite tolerance budget, which limits how complex a surface and how tight a feature the design can call for and still be made faithfully. The geometry is therefore kept to shapes that can be manufactured reliably, rather than pushed to the most aggressive form the rules would permit.

Two height choices follow from this. The mainplane is designed to sit at the 30 mm ground-clearance minimum of T2.2.1, both to exploit ground effect as fully as the rules permit (Section 4.3.1) and because the lowest legal position is a clean, repeatable assembly datum. At the top, the wing is kept about 30 mm below the draft 2027 height limit rather than built right up to it. That buffer absorbs the manufacturing and tolerance stack, and it leaves room to raise the whole wing slightly on track if testing shows the front is overloaded or the balance needs adjusting, instead of locking the wing to a single height with no margin to move. Designing at the legal floor with headroom above thus serves performance while keeping the wing buildable and giving an adjustment margin for both manufacturing variation and setup changes.

4.3 Theoretical background

The front wing operates in a flow regime that differs in two important respects from a wing in free air: it runs in close proximity to the ground, with its suction surface lowermost, and it is built from several closely spaced elements rather than one. Both features are what make the wing produce the downforce it does, and both set the limits on how much it can produce. This section establishes the physics behind both: how a wing works in ground effect, and why splitting it into multiple elements lets it carry more load. It then defines the aerodynamic balance the wing is asked to deliver.

4.3.1 Ground effect

A cambered wing generates downforce by carrying a region of low pressure on its suction surface. When that surface faces the ground and the wing is brought close to it, the gap between the suction surface and the ground acts as a constriction: the flow accelerates through it, the static pressure in the gap falls, and the suction on the lower surface increases beyond its free-air value. The downforce therefore rises as the ride height is reduced, an effect absent in free air and central to the performance of any near-ground wing [17].

Two complementary pictures explain the mechanism. In the image picture, the ground plane is replaced by a mirror-image wing beneath it; the induced flow between the real wing and its image raises the velocity, and hence the suction, on the surface facing the ground. In the diffuser picture, the gap between the suction surface and the ground narrows to a minimum and then widens, behaving as a diffuser that accelerates the flow into the throat and recovers pressure beyond it (Figure 4.6) [17]. Both lead to the same result: stronger suction, and a downforce that grows as the wing descends.



Figure 4.6: The two pictures of ground effect: (a) image model and (b) diffuser effect. The aerofoil is drawn lift-up; the present wing runs inverted, with the suction surface lowermost, and the mechanism is identical. Reproduced from [17], Figure 16, p. 55.

This growth does not continue indefinitely. As the ground clearance is reduced, the downforce first rises asymptotically, then passes through an intermediate region whose character depends on the wing loading: at low loading it tends to plateau, while at high loading it dips before recovering. It then climbs to a maximum and, below that height, collapses. The maximum marks the point where the gain in lower-surface suction is overtaken by losses on both surfaces as the gap closes further. At high loading the drop just past the maximum is sharp, driven by boundary-layer separation and the resulting loss of circulation on the main element [23]. The collapse is associated with the merging of the suction-surface boundary layer and the boundary layer growing along the ground, which throttles the flow in the gap and destroys the suction. This behaviour is the force reduction phenomenon, observed experimentally and reproduced computationally for both single and multi-element wings in ground effect [17,23]. The practical consequence is that a ground-effect wing has an optimum ride height: low enough to exploit the suction gain, but not so low that the loading is lost.

Figure 4.7 shows this behaviour measured on a double-element wing (a main element with a single flap above it), for two settings of that flap. The two settings differ only in how hard the wing is loaded, not in the number of elements: the higher flap angle

loads the wing harder, which sharpens the dip and the subsequent collapse. The curves are plotted against ride height h/c and are read from right to left as the wing descends: the downforce rises to a maximum and then falls away at the smallest heights. The wing studied is simpler than the four-element configuration of this work, but the trend it shows is a property of the lowest element running in ground effect and carries over directly.

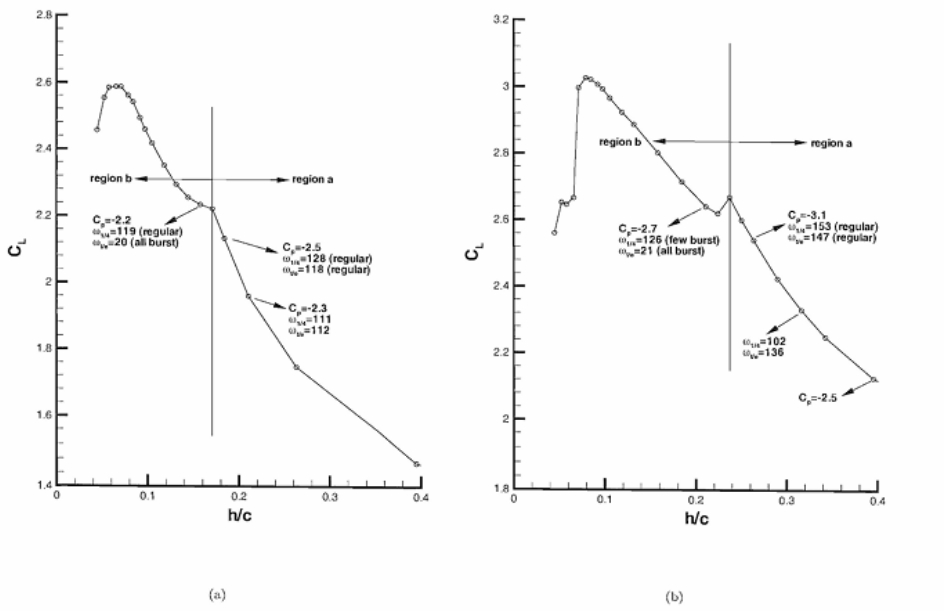


Figure 4.7: Downforce against ride height h/c for a double-element wing in ground effect, at a low (a) and high (b) flap setting. Zerihan's lift coefficient C_L is the downforce coefficient C_z used here. Reproduced from [17], Figure 37, p. 118.

Ground effect is the main reason the mainplane W1 sits as low as it does. Its lowest point is set at 30 mm, the minimum clearance T2.2.1 allows, so that the suction gain from running close to the ground is exploited as fully as the rules permit. At this height the wing operates near the peak of the downforce curve and well above the heights where the loading would collapse.

4.3.2 Aerodynamic balance

The total downforce of the car is shared between the two axles. Defining $C_{zt} = C_{zf} + C_{zr}$, the aerodynamic Front Balance is the fraction carried by the front axle, C_{zf}/C_{zt}. It is a distribution quantity, independent of the absolute level of downforce: two cars with very different C_{zt} can share the same Front Balance, while the same car's balance shifts as the front wing is loaded more or less.

The front wing is the dominant aerodynamic lever on the front axle. It is the first surface the flow meets, it carries a large share of the front-axle load directly, and it sets the inflow to every device downstream, so a change to the wing moves both C_{zf} directly and C_{zr} indirectly through the wake it sheds onto the floor, side-structures, diffuser, and rear wing [23]. For context, on a contemporary Formula 1 car the front wing produces of the order of a quarter to a third of the total car downforce [23]; the proportion differs for a Formula Student car, but the front wing's role as the primary front-axle aerodynamic device is the same.

Balance matters because it governs handling. Too little front relative to rear produces understeer, the car refusing to turn into a corner; too little rear produces oversteer. Beyond outright grip, a balance that holds steady as the car pitches, brakes, and changes ride height is what makes the car drivable and predictable, which is a safety consideration as much as a performance one [23]. The Vehicle Dynamics team's target of 55 % Front Balance (Section 4.1.3) is the front-axle share the rest of the car is tuned around; the front wing must deliver it.

4.4 Wing architecture

The front wing is a four-element configuration, a choice settled early in the iteration history and stable across all subsequent families. From that point onward, the number of elements, their order in the stack, and the spanwise extent of each element are stable architectural choices; the chord lengths, slot positions, and angles of attack are working values that the iteration history explores. The present section describes the stable architecture; the rationale for each choice, and the path by which the chord values settled, are part of the design history covered in Sections 4.5–4.7.

The four elements, ordered from the lowest to the highest, are:

- **W1, mainplane.** Chord of approximately 300 mm. Full span, 1430 mm from the inner face of one endplate to the inner face of the other. This is the lowest and largest element, designed at the minimum allowed ground clearance set by T2.2.1 to exploit ground effect to the legal limit.
- **W2.** Chord of approximately 180 mm. Full span, 1430 mm endplate to endplate. Non-adjustable. Carries the bolted assembly of W3 and W4 and serves as the structural backbone of the upper flap stack.
- **W3.** Chord of approximately 145 mm. Partial span of 470 mm, extending from one endplate inboard and terminating short of the chassis, with a finite gap between its inboard tip and the chassis sidewall. Adjustable in angle of attack. The span value is current and subject to change across the remaining iterations.
- **W4.** Chord of approximately 120 mm. Partial span of 470 mm, matching W3. Adjustable in angle of attack. The span value is current and subject to change across the remaining iterations.

The full-span elements W1 and W2 reach 1430 mm from inner endplate face to inner endplate face. The chord ratios above (*approximately* 300:180:145:120 from W1 to W4) place most of the planform area on the mainplane and the structural element, with the upper flaps progressively shorter as the stack rises. Element profiles are based on the NACA 6412 and A18 (smoothed) airfoils from the UIUC database; the rationale for the airfoil choice is covered in the 2D CFD chapter of the combined report.

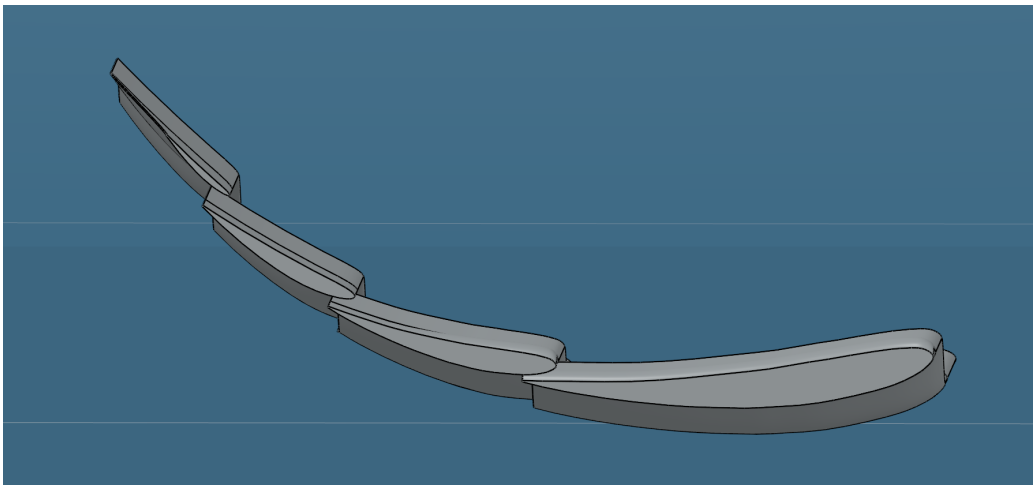


Figure 4.8: Side view of the four-element architecture. W1 is the mainplane, W2 the non-adjustable structural element, W3 and W4 the angle-of-attack-adjustable upper flaps.

The decision to terminate W3 and W4 short of the chassis, rather than carrying them all the way inboard, is made on flow-quality grounds; the inboard gap and its position with respect to the chassis are part of the iteration history. W1 and W2 remain full-span because they sit low enough to clear the chassis underside, and because their structural and downforce roles benefit from the additional planform area.

Endplates. The endplates are vertical surfaces at each spanwise extremity of the wing. They serve two functions: they reduce the tip vortex by obstructing the flow that would otherwise spill from the pressure side to the suction side around the wing tip, and they provide the structural attachment surface against which the four elements are bolted. In this work, the endplates are sized to perform these two roles and are not used as a primary downforce-tuning device; their geometry was not a focus of the iteration history.

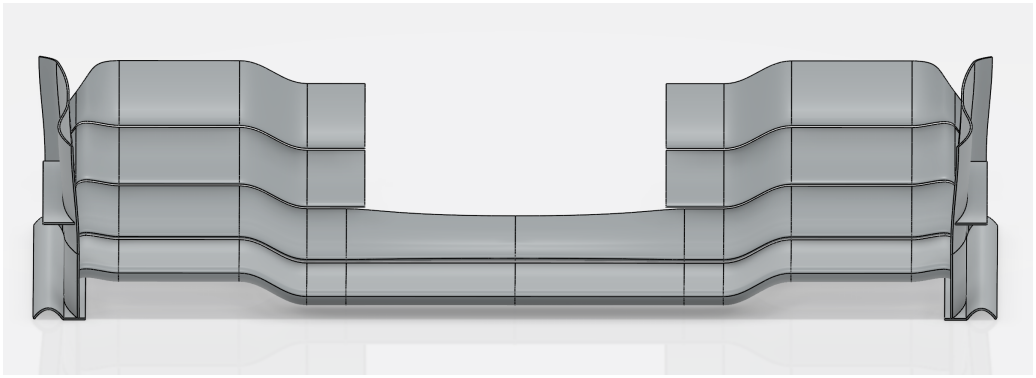


Figure 4.9: Wing architecture in three-quarter view: W1 and W2 spanning the full width between endplates, W3 and W4 terminating short of the chassis sidewall.

4.5 Design methodology

The design methodology rests on two complementary tools. The first is the C_{zt} versus Front Balance plot, which compares iterations on a single chart and makes the downforce/balance relationship explicit. The second is CFD post-processing of each iteration

(C_p slices, C_{p_t} slices, and velocity-field slices), which identifies *why* a given iteration produces the C_{zt} and Balance it does, and where the geometry can be changed to improve it. The plot ranks iterations against each other; the post-processing drives the next geometric change. This section describes how the two are used together, illustrated on the 5F iteration family. 5F was one of the early iteration families of the design work, run against the V3 baseline. The wings in this family were three-element configurations with variations explored across airfoil profiles, chord lengths, angles of attack, slot gaps, and overlaps. The family provides a clean case in which the methodology both ranked a set of changes and diagnosed why some of them did not work.

4.5.1 The C_{zt} versus Front Balance plot

Each whole-car CFD run produces a single (C_{zt}, Balance) pair and contributes one point on the plot. A given wing geometry swept through a range of flap angles of attack produces a sequence of points, all sharing the same underlying geometry and differing only in flap setting. Connected together, these points form a *line* on the plot: as the flaps rotate to add front load, both C_{zt} and Balance rise; as they rotate to shed front load, both fall. The slope of this line characterizes how much C_{zt} the geometry gains per unit of Balance change.

Lines from different geometries are then plotted together. Above-line means a geometry that produces more C_{zt} than the reference at a given Balance; below-line means less. A new iteration that lands below the line of an existing reference geometry has not advanced the design at that Balance.

The plot has two roles. *Within a baseline*, it ranks every iteration that has been run against the same whole-car configuration, against the same reference line. *Across baselines*, the rear-of-car contributions to C_{zt} change when the baseline updates, so C_{zt} values are not directly comparable from one baseline to the next; the carry-over rule (Section 4.1.3) re-runs the best previous wing on the new baseline to provide a fresh anchor.

4.5.2 Worked example: the 5F family

A *family* is a group of iterations that share a common starting geometry and explore one set of related changes from it, named with a common numeric prefix (the 5F family, the 8F family, and so on); within a family, individual iterations are numbered sequentially as the geometry or flap setting is varied.

The 5F family illustrates the methodology end-to-end. The starting point is 5F3, a wing with a 350 mm W1 chord and a 125 mm W2 chord. A flap-angle sweep on this geometry yields 5F4, with the W3 angle of attack reduced from 30° to 27° to give up some front load. The two points define a reference line for the 5F family, plotted in blue in Figure 4.10. The line has a positive slope, more Balance bringing more C_{zt}, and it is the reference that subsequent 5F variations are measured against.

Iterations 5F5 and 5F6 are not flap-only changes; they modify the wing geometry. The W1 chord is shortened from 350 mm to 300 mm, the W2 chord is lengthened from 125 mm to 150 mm, and the W2 inboard angle of attack is pushed from 8° to 12° (W2 carries a spanwise twist, with separate inboard and outboard sections; the inboard section is the one under the chassis). 5F5 and 5F6 differ from each other only in the W3 angle of attack, equivalent to a flap-only sweep on top of this modified geometry.

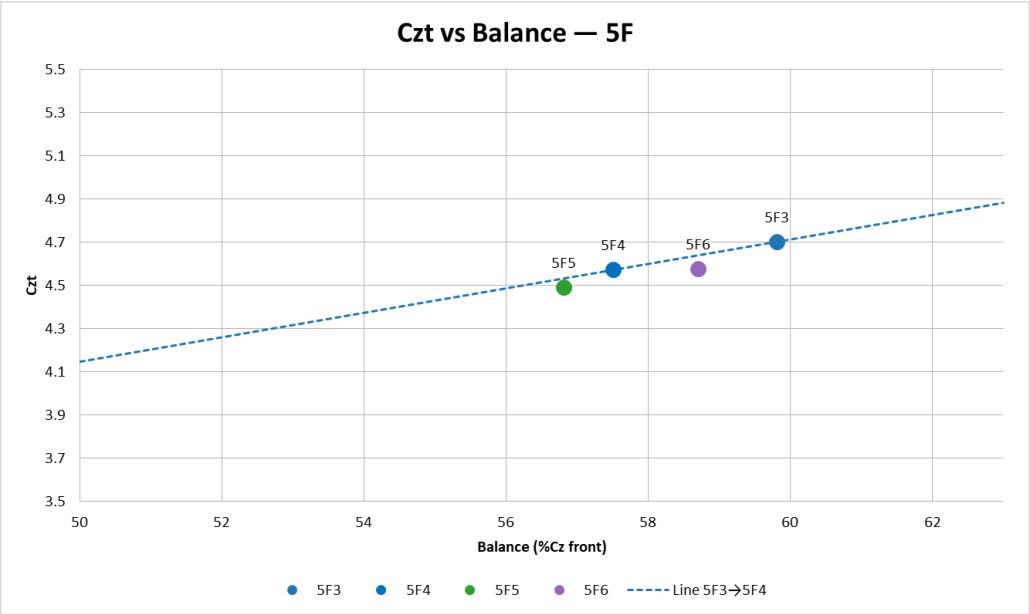


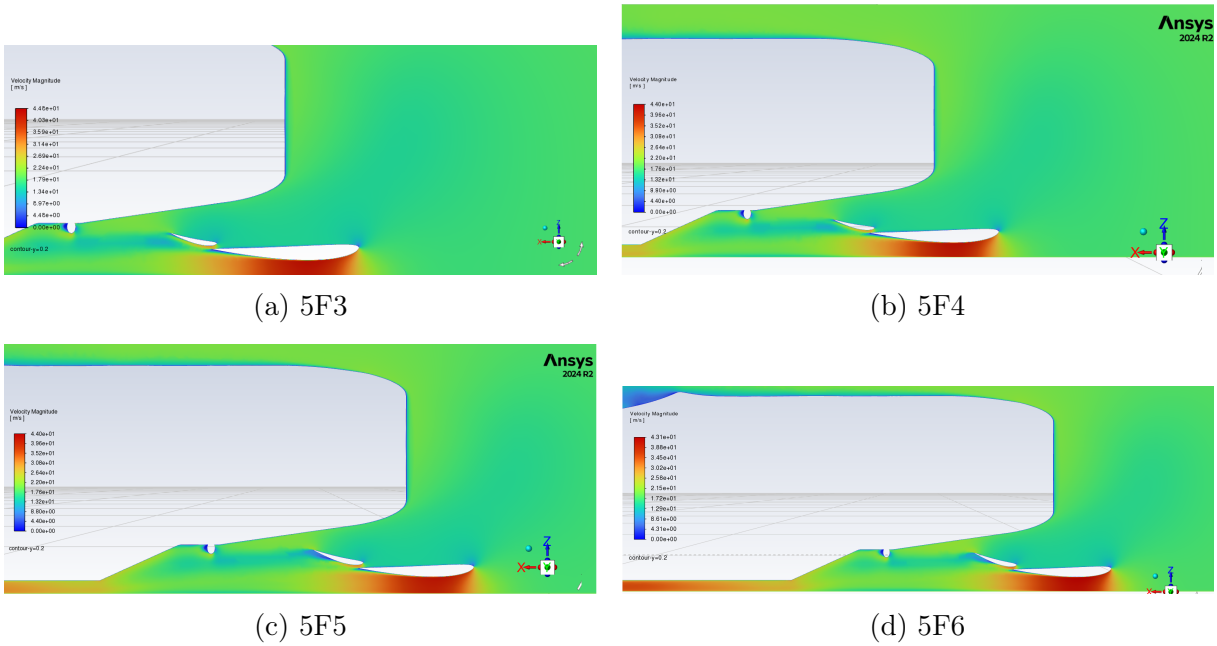
Figure 4.10: C_{zt} versus Front Balance plot for the 5F family. The blue line is the flap-sweep reference defined by 5F3 and 5F4. 5F5 and 5F6 introduce a geometric change (shorter W1, longer W2, higher W2 inboard angle of attack) and both land on or below the reference line.

On the plot, both 5F5 and 5F6 sit on or below the 5F3–5F4 reference line. 5F5 is the weaker of the two: at a Balance of 56.7 % it produces a C_{zt} of 4.49, against roughly 4.55 expected from the reference line at the same Balance. 5F6 is closer to the line but still does not improve on it. The plot alone is enough to conclude that this geometric change has not advanced the design. It is not enough, however, to explain *why*.

4.5.3 CFD post-processing

Each CFD run is post-processed on slices through the symmetry plane and on additional planes along the span. Three quantities are looked at routinely: the pressure coefficient C_p on the wing surfaces and on the slice planes, the total-pressure coefficient C_{p_t} on the same slices, and the velocity magnitude on the slices. C_p identifies where the wing is loaded and where the loading collapses, C_{p_t} identifies where the flow has lost total pressure (wakes, separated regions) or gained it (the wing's high-energy outflow), and the velocity field identifies where the boundary layer is healthy, stressed, or detached.

For the 5F example, the diagnostic comes from the velocity slices on the $y = 0.2\,\text{m}$ plane (Figure 4.11). The flow is clean in all four cases: the slot region is reasonable throughout, with no marked separation on any of the wings. The difference traces back to the W1 chord. The longer 350 mm mainplane of 5F3 and 5F4 carries more load than the shorter 300 mm mainplane of 5F5 and 5F6, which is why the longer-chord wings reach a higher C_{zt} at a given Balance. The chord reduction, made to shift load and Balance, simply gave up front downforce, and that is enough to explain why 5F5 and 5F6 fall below the 5F3–5F4 line.



(a) 5F3 (b) 5F4

(c) 5F5 (d) 5F6

Figure 4.11: Velocity magnitude on the $y = 0.2\,\text{m}$ plane for the 5F3, 5F4, 5F5 and 5F6 wings. The flow is clean in all four cases; the longer 350 mm W1 of 5F3 (a) and 5F4 (b) carries more load than the shorter 300 mm W1 of 5F5 (c) and 5F6 (d).

4.5.4 Closing the loop

The methodology, applied to 5F, produces three outputs. First, the ranking: 5F5 and 5F6 sit on or below the reference line and do not improve on the 5F3–5F4 wings. Second, the diagnosis: the flow is healthy on all four wings, and the shorter W1 chord of 5F5 and 5F6 is what costs front downforce; the 5F5-to-5F6 flap sweep confirms that the modified geometry stays below the reference at any flap setting. Third, the decision for the next iteration: keep the longer W1 chord, since shortening it gave up load without a compensating gain.

The same loop, generate, evaluate, diagnose, decide, is applied to every iteration in the history of Section 4.6. The plot ranks; the post-processing diagnoses; together they drive the next geometric change.

4.6 Design iterations

This section follows the front wing through its iteration history, in the order the work was carried out. The methodology of Section 4.5 is applied throughout: each iteration is scored on the C_{zt} versus Front Balance plot against the wing it diverged from, and the CFD post-processing explains the score. Families are reported as they happened, including the ones that did not advance the design, since the reasons a direction failed are as much a part of the result as the directions that succeeded.

The history falls into three phases. The first establishes the wing architecture itself, moving from an initial two-element wing through a three-element configuration to the four-element stack described in Section 4.4. The second refines that four-element wing, tuning the upper elements, the keep-out-zone region, and the flap settings. The third converges the design and studies the remaining geometric features. Each family begins

from the best wing carried over from the previous baseline (Section 4.1.3), so the narrative also tracks how the anchor improved from one baseline to the next.

4.6.1 Establishing the architecture

At the beginning of the design process the CFD post-processing was carried out by the team, because the complete CFD workflow and result files were not yet directly accessible. Only limited information was therefore available for the earliest iterations, mainly velocity-magnitude contours. These early slices were used as qualitative guidance, to spot flow acceleration and likely separation regions rather than to quantify them; the comparisons in this phase rest mainly on the force coefficients.

1F: two-element baseline. The first family was a two-element wing, two full-span elements (Figure 4.12), run before any further elements were added. Its purpose was to establish a working baseline and to confirm the expected response to the second element's incidence: loading the second element harder should raise the downforce the wing carries. Three settings were run, 1F1, 1F2 and 1F3, varying the angle of attack of the second element through 8°, 10° and 12° (Table 4.2). The whole-car downforce coefficient rose with angle of attack as expected, and the step from 10° to 12° is notably larger than from 8° to 10°, the wing gaining far more downforce over the last two degrees than the first two. Front Balance is not yet a meaningful design quantity at this stage, since the wing is still a simple two-element layout far from its final form; the 1F family serves only to fix the baseline load level that the subsequent additions are measured against.

Iteration	Second-element AoA	FW downforce [N]	C_{zf}	C_{zt}
1F1	8°	81.6	0.39	2.15
1F2	10°	96.9	0.47	2.28
1F3	12°	112.5	0.76	3.30

Table 4.2: 1F family: front-wing downforce, front-wing coefficient C_{zf} and whole-car coefficient C_{zt} at three second-element angles of attack.

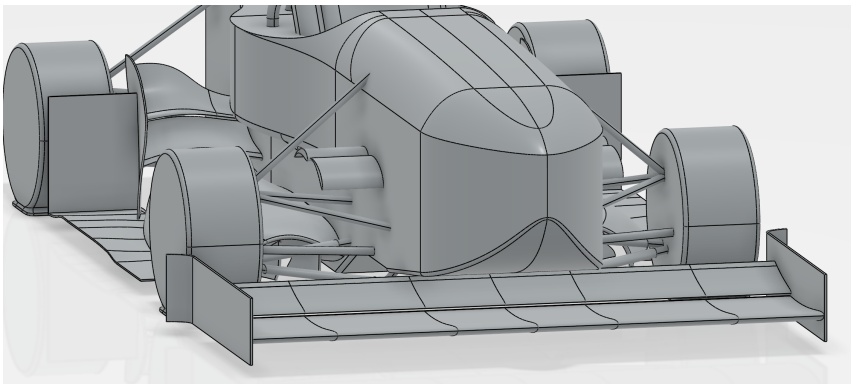


Figure 4.12: The 1F two-element baseline wing: two full-span elements.

2F: spanwise twist on the second element. The 2F family kept the two full-span elements of 1F and continued the exploration, this time introducing a spanwise twist on the second element (Figure 4.13). Rather than a single angle of attack across the span,

the second element was split into three sections, a centre section and two outer sections, each at its own angle. The centre was set lower than the sides, so the element is loaded more heavily toward the outer span than at the centre. The slot gap between the two elements was set using a 6 mm cylinder, fixing the minimum gap at that value. Two twist configurations were run (Table 4.3): 2F1 at 21°/15°/21° (sides/centre/sides) and 2F2 at the more aggressive 24°/18°/24°. The more aggressive twist carried more load: 2F2 reached a whole-car C_{zt} of 3.61 against 3.44 for 2F1, with the front wing's own downforce rising from 171.2 N ($C_{zf} = 1.20$) to 183.6 N ($C_{zf} = 1.29$), so 2F2 was taken as the better of the two.

Iteration	W2 twist (side/centre/side)	FW downforce [N]	C_{zf}	C_{zt}
2F1	21°/15°/21°	171.2	1.20	3.44
2F2	24°/18°/24°	183.6	1.29	3.61

Table 4.3: 2F family: front-wing downforce, front-wing coefficient C_{zf} and whole-car coefficient C_{zt} for the two W2 twist configurations, with the 6 mm slot gap.

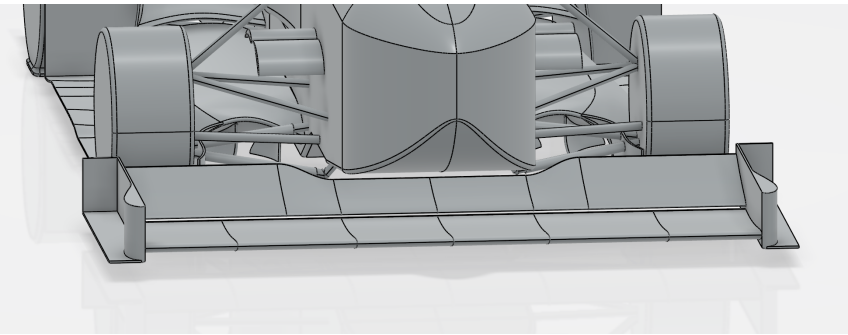


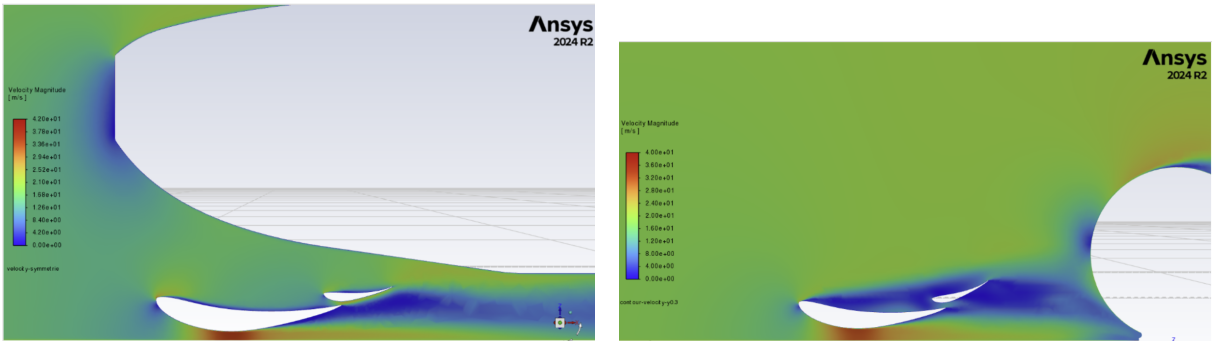
Figure 4.13: The 2F wing: two full-span elements with a spanwise twist on the second element (centre section at lower angle of attack than the outer sections).

3F: misaligned first element. The 3F family tested a different airfoil on the first element, thicker and more cambered than the profiles used elsewhere, with the intent of carrying more load through greater camber. The result was the worst of any iteration in the history: the front wing produced only about 67 N of downforce ($C_{zf} = 0.47$, whole-car $C_{zt} = 2.85$, Table 4.4), well below the 171 to 184 N of the 2F wings.

The velocity contours showed the flow failing to follow the first element from the leading edge, separating rather than accelerating cleanly over it (Figure 4.14). The element was not aligned with the oncoming flow, so it could not work regardless of its profile. The thick, highly cambered airfoil was dropped, and the design returned to the NACA 6412 and A18 sections used elsewhere (Section 4.4).

Iteration	FW downforce [N]	C_{zf}	C_{zt}
3F1	67.2	0.47	2.85

Table 4.4: 3F1: front-wing downforce, front-wing coefficient C_{zf} and whole-car coefficient C_{zt}.



(a) Symmetry plane (b) Offboard slice ($y = 0.3$)

Figure 4.14: Velocity-magnitude contours for 3F1. At both the symmetry plane (a) and an offboard slice (b).

5F: first three-element wing. The 5F family was the first three-element wing. The three elements used the N22 profile on W1, a mixed E423/N22 profile on W2 (E423 outboard, N22 inboard), and E423 on W3. Across 5F1 to 5F7 the chords and the angles of attack of all three elements were varied (Tables 4.5 and 4.6).

5F also introduced a curved transition at the outboard end of the wing, where the elements sweep downward as they approach the endplate (Figure 4.15). It was chosen to improve the flow in the outboard region of the wing, and it is kept as a feature in the families that follow. Its effect on the outboard flow is examined in more detail on the four-element wing later in this section.

The first iteration, 5F1, ran W1 at 0° and produced a low front load ($C_{zf} = 1.10$, Balance 51.2 %). Pitching W1 to −2° from 5F2 onward unlocked a large gain at the front: C_{zf} jumped to 1.52 and the whole-car C_{zt} rose from 4.05 to 4.67, with Balance moving to about 60 % (Figure 4.16). The family then settled around this level, with 5F2, 5F3 and 5F7 forming its high point at C_{zt} between 4.66 and 4.70 and Balance near 59 to 60 %. 5F3 in particular, at $C_{zt} = 4.70$, was the strongest wing of the family.

The remaining iterations explored the chord and loading trades: 5F4 backed off the W3 angle of attack, and 5F5 and 5F6 shortened the W1 chord from 350 to 300 mm while lengthening the W2 chord and loading the inboard W2 harder. This combination did not advance the design. 5F was the last three-element family; the design moved to a four-element stack immediately afterward, where the lasting anchor wings were found, so no 5F iteration was carried forward in its own right.

	W1	W2	W3
Iteration	chord [mm] / $\mathrm{AoA_{in}}$,$\mathrm{AoA_{out}}$ [°]	chord [mm] / $\mathrm{AoA_{in}}$,$\mathrm{AoA_{out}}$ [°]	chord [mm] / AoA [°]
5F1	350 / 0, 0	125 / 8, 19	125 / 30
5F2	350 / 0,−2	125 / 8, 19	125 / 30
5F3	350 / −2, 0	125 / 8, 19	125 / 30
5F4	350 / −2, 0	125 / 8, 16	125 / 27
5F5	300 / −2, 0	150 / 12, 16	125 / 27
5F6	300 / −2, 0	150 / 12, 19	125 / 30
5F7	350 / −2,−2	150 / 10, 19	125 / 30

Table 4.5: 5F family geometry. Chords in mm and angles of attack in degrees. Where two angles are given they are the inboard and outboard values of a spanwise twist, inboard first.

Iteration	FW downforce [N]	C_{zf}	C_{zt}	Balance
5F1	156.2	1.10	4.05	51.2 %
5F2	217.0	1.52	4.67	60.0 %
5F3	216.8	1.52	4.70	59.2 %
5F4	201.3	1.41	4.57	57.5 %
5F5	193.3	1.36	4.49	56.7 %
5F6	207.6	1.46	4.58	58.7 %
5F7	218.0	1.53	4.66	59.4 %

Table 4.6: 5F family results: front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance.

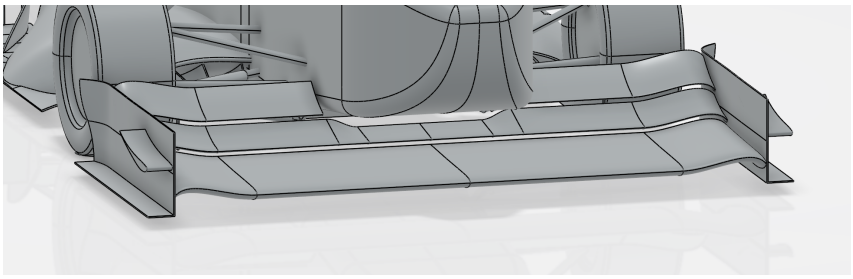


Figure 4.15: The 5F three-element wing, showing the curved outboard transition where the elements sweep down into the endplate.

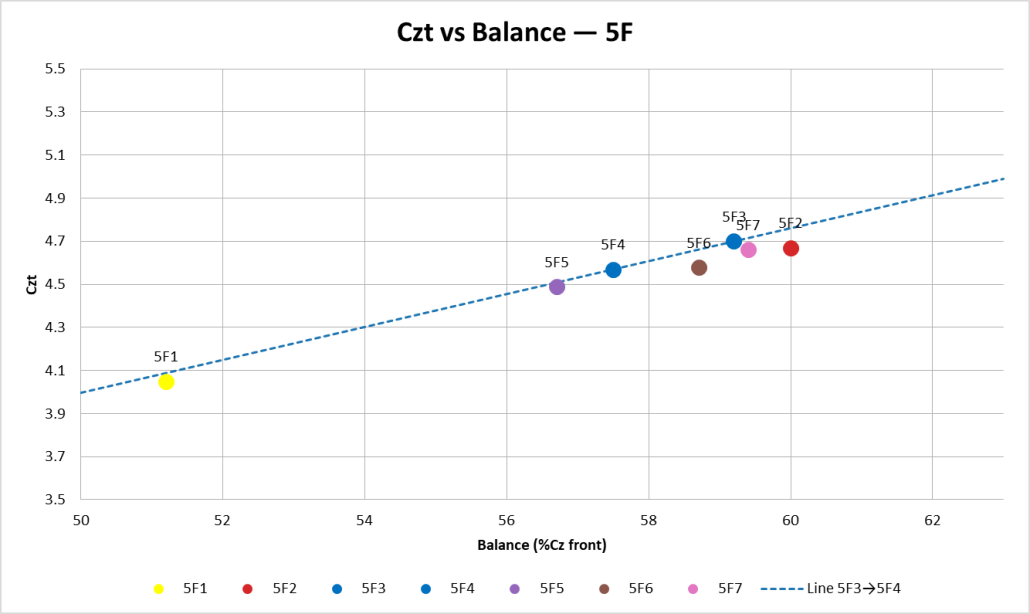


Figure 4.16: C_{zt} versus Front Balance for the 5F family.

7F: NACA 6412 three-element wing. The 7F family was a single three-element iteration, 7F1, kept close to the 5F layout but with every element switched to the NACA 6412 profile and the W1 chord lengthened to 380 mm. It tested the airfoil change on an otherwise familiar three-element wing. The result fell short of the strongest 5F wing: 7F1 reached $C_{zt} = 4.20$ at 53.8 % Balance, against the $C_{zt} = 4.70$ of 5F3 (Table 4.7). The

configuration was not pursued as a three-element wing, and the design moved to the four-element architecture in the next family. The NACA 6412 profile itself, however, returns successfully on the four-element wings that follow.

Iteration	FW downforce [N]	C_{zf}	C_{zt}	Balance
5F3	216.8	1.52	4.70	59.2 %
7F1	177.4	1.24	4.20	53.8 %

Table 4.7: 7F1 compared against 5F3, the strongest three-element wing: front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance.

4.6.2 The four-element wing

8F: four elements and the keep-out zones. The 8F family is the first four-element wing and the point from which the architecture of Section 4.4 is fixed. Two changes were made together. First, a fourth element was added to the stack, giving the W1 to W4 configuration carried by every subsequent family. Second, the two T2.1.4 keep-out zones (Section 4.2) were introduced into the model, and the family was used to study where to place them.

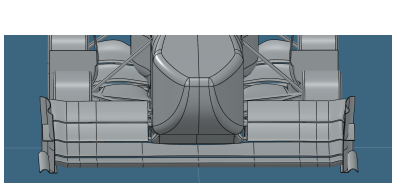
The placement study used 8F4 and 8F5 as keep-out-free references, two flap settings of the same four-element wing run without any keep-out zone, so that the cost of introducing the zones could be read against a wing of known load. 8F5 is 8F4 with 2° added on the flaps. Four placements were then tested, 8F6 through 8F9, each positioning the two keep-out rectangles at a different station in front of the front tires (Figure 4.17):

- **8F6:** the two keep-out rectangles low and flat, just inboard of the front tire.
- **8F7:** the rectangles underneath the chassis.
- **8F8:** the rectangles low and flat at the outboard station, near the tire and endplate.
- **8F9:** the rectangles standing vertically at the extreme outboard edge, against the endplate.

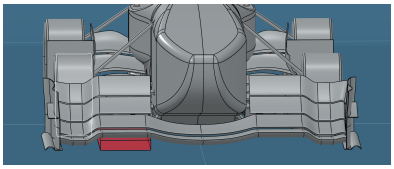
The goal at this stage was narrow: find the placement that lets the front wing carry the most downforce, with Front Balance recorded but not yet used to choose between placements. On that measure the result is clear. 8F6 and 8F8 stand well above the others, reaching C_{zt} of 5.34 and 5.33 respectively, against 5.00 for 8F7 and 4.46 for 8F9 (Table 4.8). The two strong placements also carry the most front load, 300.6 N for 8F6 and 320.0 N for 8F8, while the vertical outboard placement of 8F9 is the worst of the set on every measure. 8F6 and 8F8 were both taken forward as anchors for the next family.

Iteration	Keep-out placement	FW downforce [N]	C_{zf}	C_{zt}	Balance
8F4	none (reference)	274.7	1.77	4.96	55.23 %
8F5	none (reference, +2° flap)	283.6	1.83	4.94	56.59 %
8F6	inboard of tire	300.6	1.94	5.34	58.12 %
8F7	under chassis	271.1	1.75	5.00	54.80 %
8F8	outboard, near endplate	320.0	2.06	5.33	61.09 %
8F9	vertical, outboard edge	197.6	1.27	4.46	44.37 %

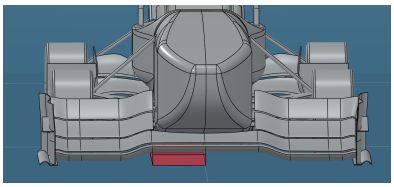
Table 4.8: 8F family: front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance for the two keep-out-free references (8F4, 8F5) and the four keep-out-zone placements (8F6 to 8F9).



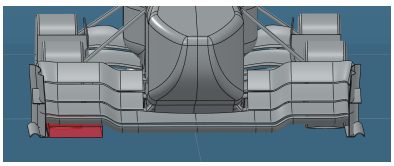
(a) 8F4 (no zone)



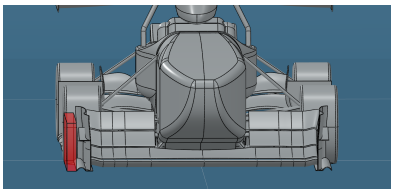
(b) 8F6



(c) 8F7



(d) 8F8

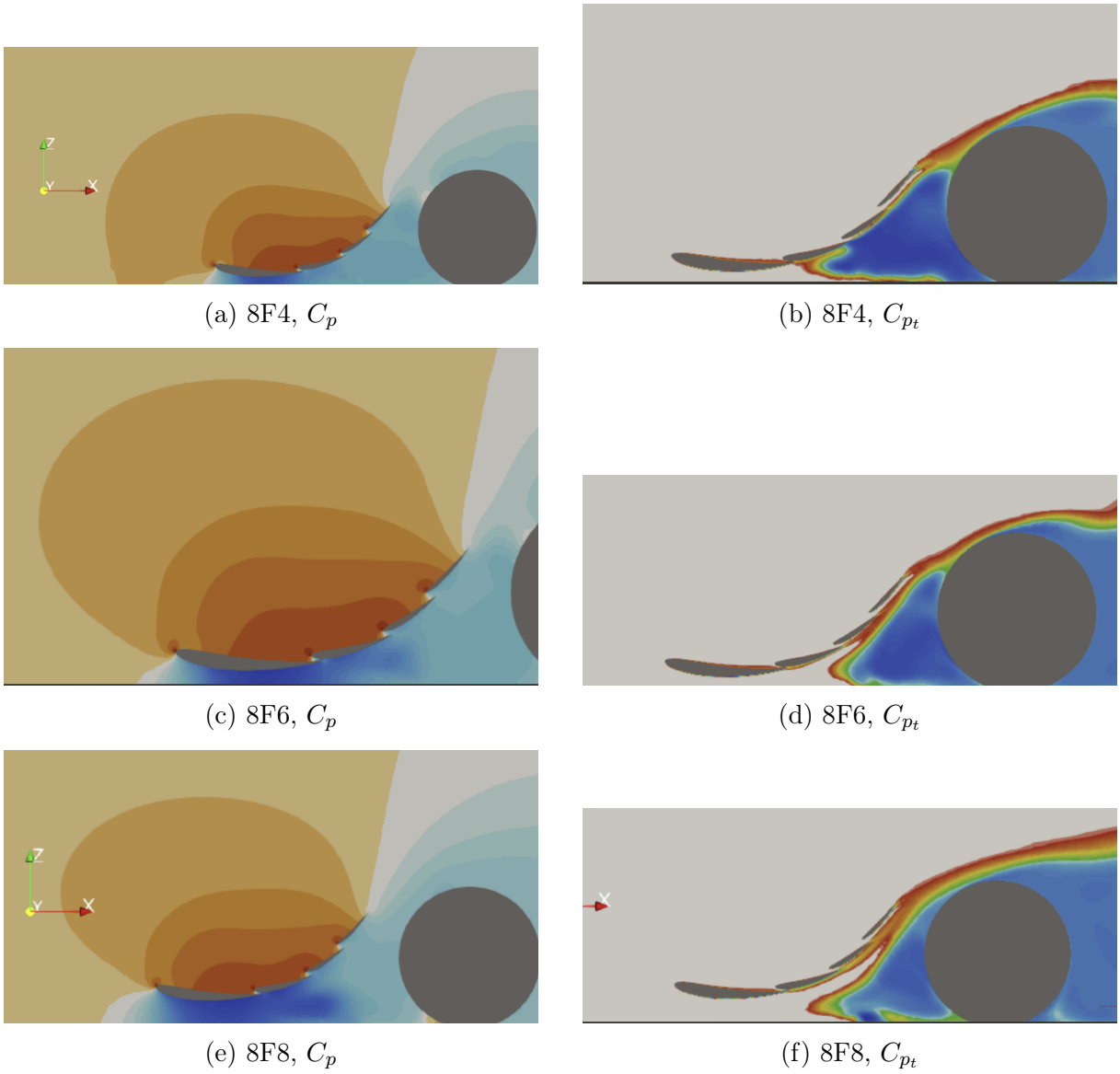


(e) 8F9

Figure 4.17: Keep-out-zone placements in the 8F family, front view, with the two T2.1.4 rectangles shown in red. 8F4 (a) carries no zone and is the reference; 8F6 (b) places the rectangles just inboard of the front tire, 8F7 (c) under the chassis, 8F8 (d) outboard near the endplate, and 8F9 (e) vertically at the outboard edge.

The post-processing was run as two comparisons against the 8F4 reference, 8F4 versus 8F6 and 8F4 versus 8F8, on C_p and C_{p_t} slices at the $y = 0.65$ plane (Figure 4.18). The flow is attached on all three wings, with no separation. The C_p panels show a larger and stronger suction region beneath the mainplane for 8F6 and 8F8 than for 8F4, the deeper loading that gives them their higher front-wing downforce (300.6 N and 320.0 N against 274.7 N). The C_{p_t} panels add a second difference: the wake behind the wing is less aggressive for 8F6 and 8F8 than for 8F4. Both show a lighter region of recovered total pressure directly behind the elements where 8F4 carries a stronger loss, indicating that the two strong placements not only load the wing harder but shed a cleaner wake behind it.

For 8F7 and 8F9 the slices were produced but are not presented, since the force coefficients already make clear which placements are worth examining. 8F9 places the zone vertically at the outboard edge, where it most obstructs the outboard flow, and its load collapses accordingly, while the under-chassis placement of 8F7 sits in already low-energy flow and costs less but still does not reach 8F6 or 8F8.



(a) 8F4, C_p

(b) 8F4, C_{p_t}

(c) 8F6, C_p

(d) 8F6, C_{p_t}

(e) 8F8, C_p

(f) 8F8, C_{p_t}

Figure 4.18: C_p (left) and C_{p_t} (right) on the $y = 0.65$ plane for the 8F4 reference (top) and the two strong placements 8F6 (middle) and 8F8 (bottom). The C_p panels show the deeper mainplane loading of 8F6 and 8F8; the C_{p_t} panels show their lighter wake behind the wing, a smaller total-pressure loss than the 8F4 reference.

Although 8F6 and 8F8 reach effectively the same whole-car C_{zt}, 8F8 carries the highest front-wing downforce of the study, 320.0 N against 300.6 N for 8F6. In keeping with the first-phase goal of maximizing front load before the Balance target is enforced (Section 4.1.3), 8F8 was taken as the wing to develop further, and every family from 9F onward derives from it. 8F6, despite sitting closer to the 55 % target, was not given its own derivative branch, a gap in the search rather than a conclusion that 8F8 was the better root.

9F: anchor lines and outboard bend on the fourth element. The 9F family served two purposes. The first was to establish reference lines on the C_{zt} versus Front Balance plot for the two strong 8F wings by sweeping their flap angles. 9F2 is 8F6 with the flap angle of attack reduced by 2°, and 9F1 is 8F8 with the same −2° change. Each parent

and its swept variant define a line, 8F6 with 9F2 and 8F8 with 9F1 (Figure 4.19). The 8F8 line sits at higher Front Balance, more front-biased than the 55 % target, while the 8F6 line is better centred on it. As set out in the 8F study, 8F8 was nonetheless the wing carried forward, since the first phase of the design prioritizes front load over Balance; the 8F6 line is kept on the plot as the alternative that was closer to target but not pursued.

The second purpose was 9F3, a bend introduced on the outboard portion of the fourth element. Its purpose was to reduce the upwash shed by the four-element wing, which should in turn raise C_{zt}. On the plot, 9F3 lands essentially on top of 8F8, at C_{zt} 5.34 and 61.1 % Balance against 5.33 and 61.09 % for 8F8 (Table 4.9). The bend produced no measurable gain: the expected C_{zt} increase did not materialize. 9F3 was nonetheless retained as the base geometry from which the 10F, 11F and 13F families are derived, so that those families build on the bent-W4 wing rather than on raw 8F8.

Iteration	Description	C_{zt}	Balance
8F6	placement anchor	5.34	58.12 %
9F2	8F6, −2° flap	5.25	57.40 %
8F8	placement anchor	5.33	61.09 %
9F1	8F8, −2° flap	5.29	60.50 %
9F3	8F8 with outboard W4 bend (carried forward)	5.34	61.10 %

Table 4.9: 9F family: whole-car coefficient C_{zt} and Front Balance for the two flap-sweep variants (9F1, 9F2) that define the 8F8 and 8F6 lines, and the outboard-bend variant 9F3 carried forward as the base geometry for the 10F, 11F and 13F families.

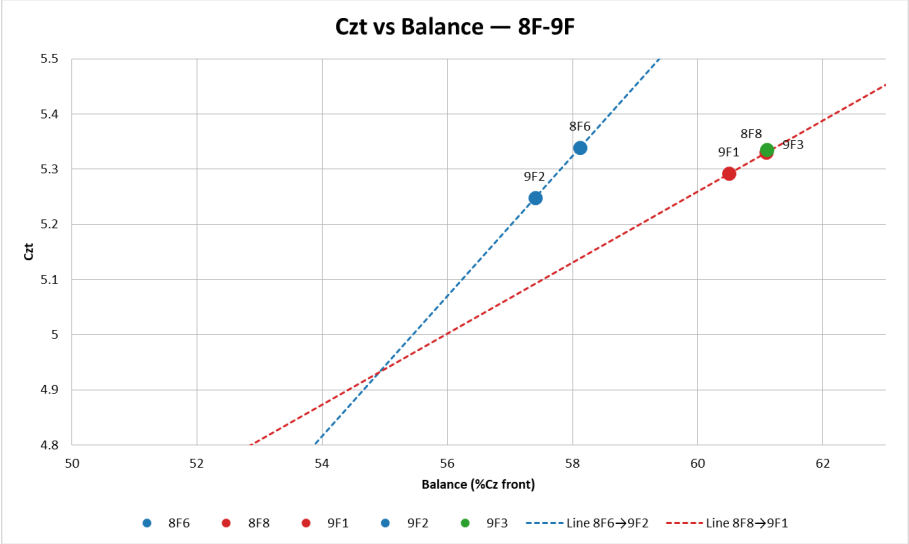


Figure 4.19: C_{zt} versus Front Balance for the 9F family. The two flap-sweep lines define the 8F6 (→9F2) and 8F8 (→9F1) trades; 9F3, the outboard-W4 bend on 8F8, sits on top of 8F8 and shows no C_{zt} gain.

10F: outboard fourth element. 9F3 introduced a bend on the outboard portion of the fourth element without raising C_{zt}. The 10F family followed it up by sweeping the outboard W4 feature across four iterations, 10F2 to 10F5, each built on the 9F3 geometry (Figure 4.20), to test whether the feature helped once it was tuned rather than fixed at the

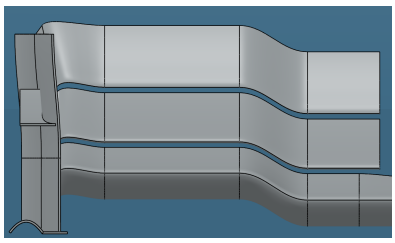
9F3 value. In 10F2 the outboard curve was flattened, running straight from the trailing edge of W2 through W3 and W4 rather than sweeping down into the endplate.

On the C_{zt} versus Front Balance plot (Figure 4.21), the four iterations cluster tightly around the 9F3 reference, all between 60.4 and 61.1 % Balance and within a narrow C_{zt} band. 10F2 is the best of the four, reaching C_{zt} 5.38 at 60.4 % Balance against 5.34 for 9F3, a small gain. 10F5 is marginally above the reference at 5.37, while 10F3 (5.32) and 10F4 (5.30) sit just below it (Table 4.10). The spread across the whole sweep is small: tuning the outboard W4 feature moved C_{zt} only at the margin and did not unlock the gain the bend was meant to provide.

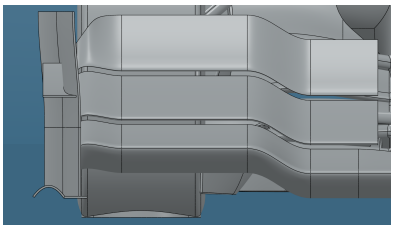
The C_{p_t} post-processing tells the same story from the flow side. Across the six wings (8F8, 9F3 and the four 10F iterations) the wake structures behind the wing are very similar (Figure 4.22). 10F2 shows a slightly smaller low-total-pressure burst behind the wing than the others, consistent with its being the marginal best on the plot, but the difference is minor and the remaining wings are hard to separate by eye. Neither source on its own would settle the ranking. Read together, the plot and the C_{p_t} frames agree that the outboard W4 sweep produced no meaningful gain, with 10F2 the best of a tightly clustered set.

Iteration	C_{zt}	Balance
9F3 (reference)	5.34	61.10 %
10F2	5.38	60.40 %
10F3	5.32	60.90 %
10F4	5.30	61.10 %
10F5	5.37	60.60 %

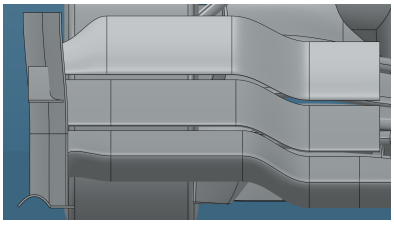
Table 4.10: 10F family: whole-car coefficient C_{zt} and Front Balance for the four outboard-W4 iterations against the 9F3 reference. 10F2 is the marginal best; the spread across the sweep is small.



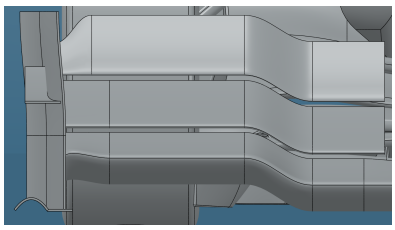
(a) 8F8



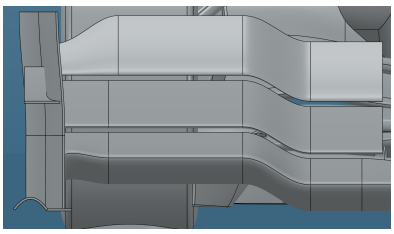
(b) 9F3



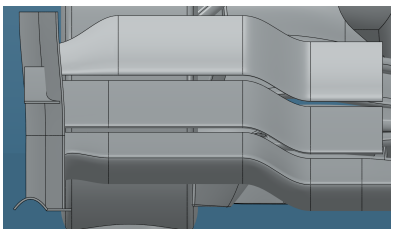
(c) 10F2



(d) 10F3



(e) 10F4



(f) 10F5

Figure 4.20: Outboard W4 geometry across the 10F family, in the same order as the C_{p_t} comparison. 8F8 and 9F3 are the references; 10F2 to 10F5 vary the outboard feature, with 10F2 flattening the outboard region from the W2 trailing edge through W3 and W4.

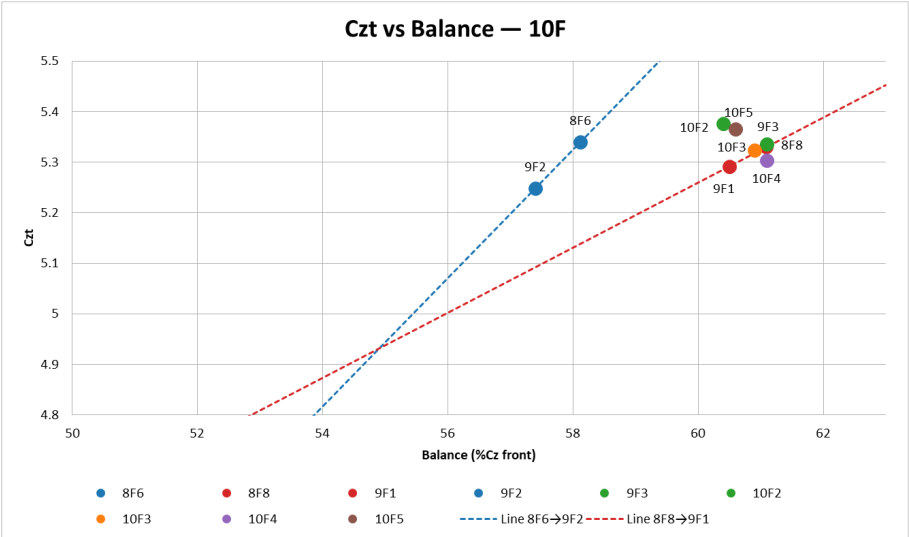
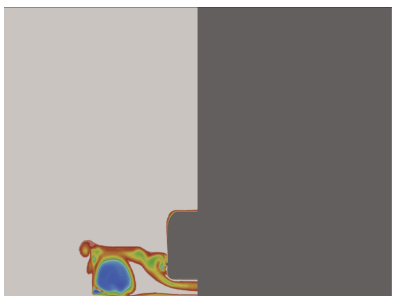


Figure 4.21: C_{zt} versus Front Balance for the 10F family. The four outboard-W4 iterations cluster tightly around the 9F3 reference near 60–61 % Balance; 10F2 is the marginal best.



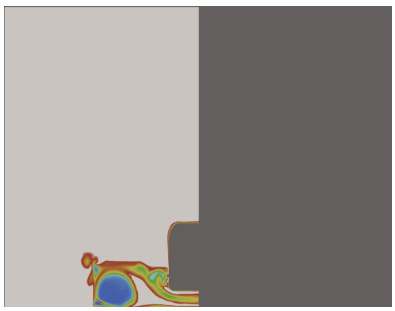
(a) 8F8



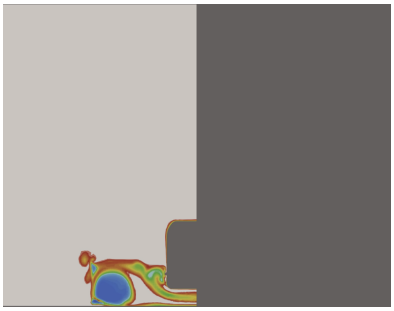
(b) 9F3



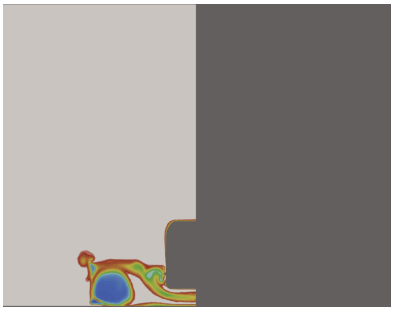
(c) 10F2



(d) 10F3



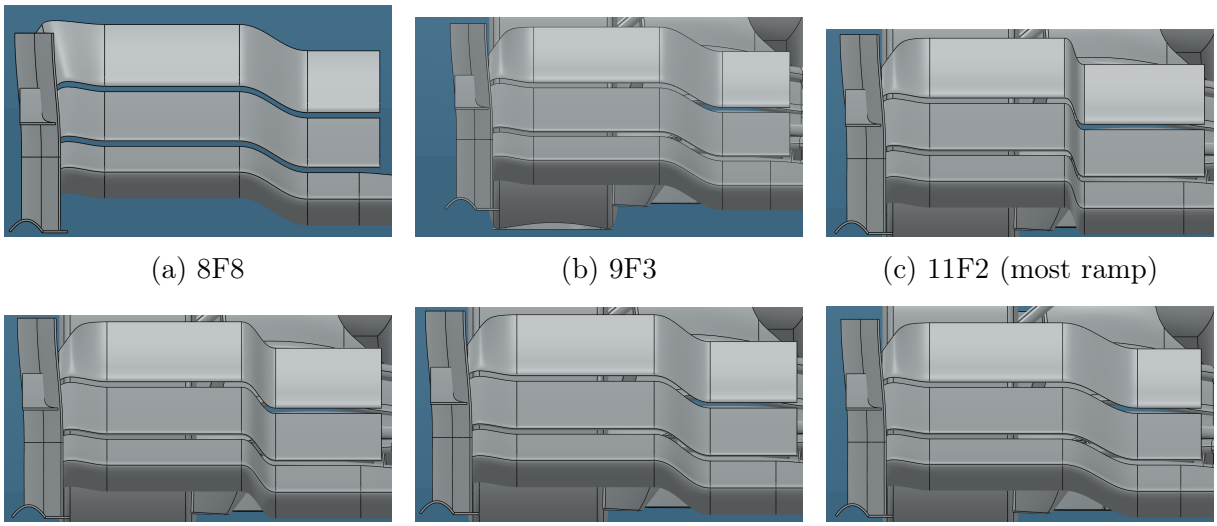
(e) 10F4



(f) 10F5

Figure 4.22: C_{p_t} behind the wing at $T = 22\,\mathrm{s}$ for 8F8, 9F3 and the four 10F iterations. The wake structures are very similar across the set; 10F2 (c) shows a slightly smaller low-total-pressure burst than the others, a minor difference.

11F: mainplane ramp ahead of the keep-out zones. The 11F family turned from the upper elements to the mainplane, varying the ramp on W1 ahead of the two T2.1.4 keep-out zones (Section 4.2), the part of the mainplane just upstream of the zones (Figure 4.23). The ramp was varied across five iterations, 11F1 to 11F5, built on the 9F3 geometry; the keep-out zones themselves are mandatory and remain in place throughout.



(a) 8F8 (b) 9F3 (c) 11F2 (most ramp)

(d) 11F3 (e) 11F4 (f) 11F5 (least ramp)

Figure 4.23: Mainplane ramp geometry across the 11F family, ahead of the keep-out zones. 8F8 and 9F3 are the references; 11F2 to 11F5 reduce the ramp progressively, 11F2 the most aggressive and 11F5 the least.

No CAD panel is shown for 11F1: its ramp is so much more aggressive than the rest of the family that it sits well outside the range the other iterations occupy, and it is documented by its force coefficients alone as the collapsed extreme.

The study returned a clear and single-direction result: more ramp is worse. The most aggressive ramp, 11F1, collapsed the wing entirely, falling to a C_{zt} of 4.90 at 51.8 % Balance, far below every other iteration in the family and a clear dead end. Pulling the ramp back recovers the load: across 11F2 to 11F5 the ramp is progressively reduced, and C_{zt} climbs steadily from 5.30 to 5.33 as it does (Table 4.11). 11F5, the least pronounced ramp, is the best of the family at $C_{zt} = 5.33$ and 60.5 % Balance, landing essentially back on the 9F3 reference (5.34) rather than above it. The ramp, in other words, does not add load; the wing is best when the ramp is kept to a minimum. 11F5 was nonetheless retained as the best ramp setting and is later combined with the 10F2 wing to form 16F1.

The C_{p_t} post-processing explains the trend (Figure 4.25). The most aggressive ramp sheds a small vortex at the ramp itself, visible in 11F2, which persists downstream and carries a total-pressure loss with it. As the ramp is reduced toward 11F5 this vortex weakens and the associated loss falls, which is why the shallower-ramp iterations recover toward the 9F3 level. The flow reason and the plot ranking agree: the ramp generates a loss that grows with its severity, so the minimal-ramp 11F5 is the best of the set.

Iteration	FW downforce [N]	C_{zf}	C_{zt}	Balance
9F3 (reference)	320.0	2.06	5.34	61.10 %
11F1	253.4	1.63	4.90	51.80 %
11F2	318.9	2.05	5.30	61.30 %
11F3	318.7	2.05	5.32	61.00 %
11F4	315.4	2.03	5.32	60.60 %
11F5	315.2	2.03	5.33	60.50 %

Table 4.11: 11F family: front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance for the ramp variants against the 9F3 reference. 11F1 has the most aggressive ramp and collapses the wing; reducing the ramp through to 11F5 recovers the load.

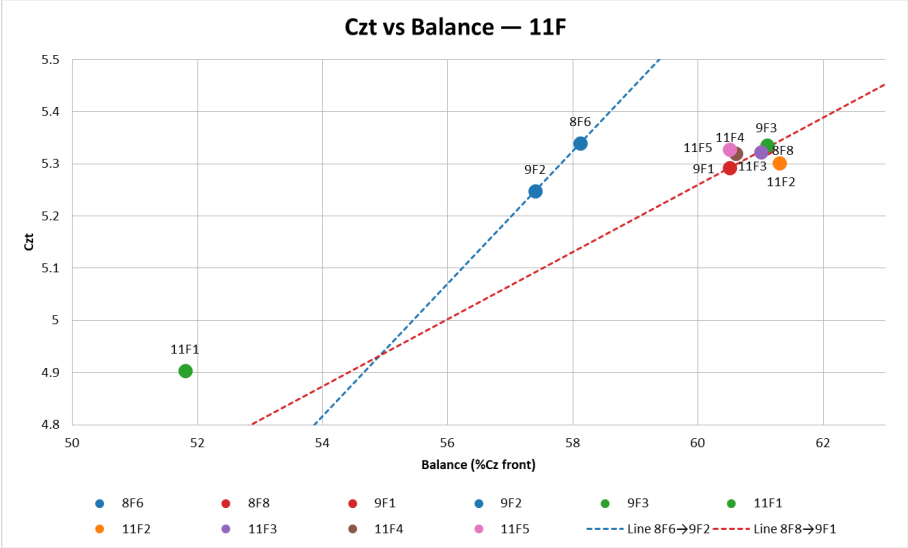
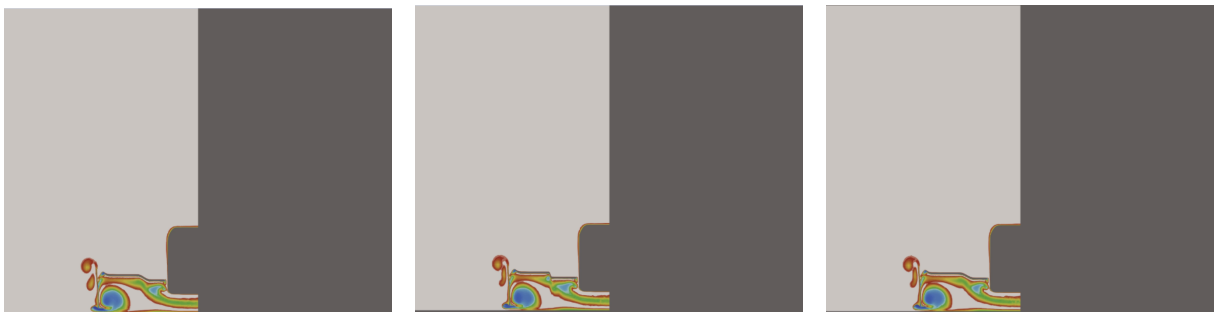


Figure 4.24: C_{zt} versus Front Balance for the 11F family.



(a) 9F3 (b) 11F2 (most ramp) (c) 11F5 (least ramp)

Figure 4.25: C_{pt} behind the wing at $T = 20\,\mathrm{s}$ for 9F3, 11F2 and 11F5.

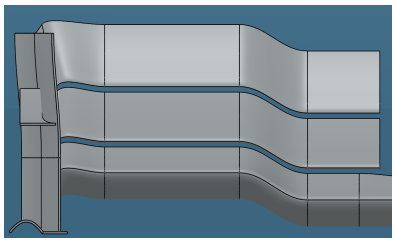
13F: inboard fourth element. Having worked the outboard end of the fourth element in 9F and 10F without advancing the design, the 13F family turned to the inboard end, modifying the inboard portion of W4 (Figure 4.26). 13F1 and 13F3 reduce the inboard W4, cutting the element back near the chassis by two different extents, while 13F4 does the opposite and increases it, extending the inboard W4 inward. The family was run on

the V4 baseline rather than the V3 baseline of the earlier families, and all V4 runs carried winglets on the car. Both differences lower the absolute C_{zt} relative to the V3 families and mean these values are not directly comparable to the 8F to 11F numbers. They do not, however, affect the comparison within the family: 10F2.2 and every 13F iteration carry the winglets equally, so the 10F2 wing was re-run on V4 (10F2.2) and used as the reference, and the 13F gains are read against it on equal terms. Carrying the winglets at this stage is best avoided: winglets are a downstream device whose shape depends on the flow the front wing delivers, so they are best designed only once the front wing is converged, rather than added to the baseline mid-development, where they lower the absolute C_{zt} and complicate comparison without yet serving their purpose.

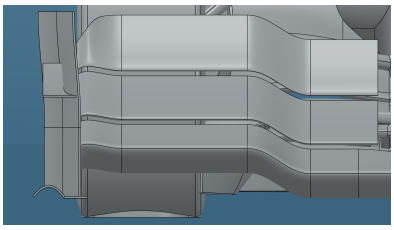
Because the baseline changed, the reference line on the C_{zt} versus Front Balance plot was rebuilt for V4. Rather than fit a new line, the existing reference-line slope (the 8F8→9F1 line of the earlier families) was translated to pass through the V4 point 10F2.2, keeping the same C_{zt}-per-Balance slope and re-anchoring it on the new baseline. The 13F points are then judged above or below this translated line (Figure 4.27).

13F was the first family to clear the reference. 13F1, the reduced inboard W4, sits above the translated line at $C_{zt} = 5.34$ and 56.6 % Balance, against 5.32 at 57.1 % for the 10F2.2 reference: more C_{zt} at a lower, better-placed Balance (Table 4.12). 13F4, the increased inboard W4, lands essentially on the reference at 5.32, adding nothing, and 13F3, the other reduction, falls below it at 5.27. Reducing the inboard W4 by the right amount, as in 13F1, therefore helped, while increasing it did not and a different reduction hurt.

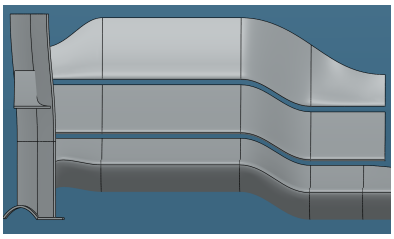
The C_{p_t} post-processing points to why (Figure 4.28). Behind the inboard region, 13F1 shows a noticeably larger white zone of recovered total pressure than 10F2.2, that is, cleaner flow with less loss where the inboard W4 was cut back. This cleaner inboard air feeds the side structures downstream, whose load rises from 170.2 N on 10F2.2 to 173.7 N on 13F1. The gain in C_{zt} is consistent with this: part of it comes from the front wing, and part from the side structures working better on the cleaner inflow that the reduced inboard W4 delivers.



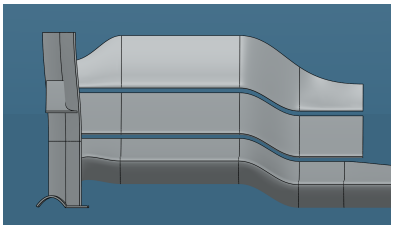
(a) 8F8



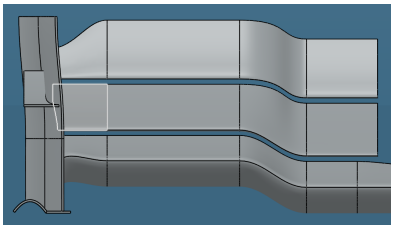
(b) 9F3



(c) 13F1 (inboard reduced)



(d) 13F3 (inboard reduced)



(e) 13F4 (inboard increased)

Figure 4.26: Inboard W4 geometry across the 13F family. 8F8 and 9F3 are the references; 13F1 and 13F3 reduce the inboard W4 by two different extents, while 13F4 increases it. The modified inboard region is marked on each.

Iteration	FW downforce [N]	C_{zf}	C_{zt}	Balance
10F2.2 (V4 reference)	306.7	1.98	5.32	57.08 %
13F1	304.1	1.96	5.34	56.60 %
13F3	300.9	1.94	5.27	56.80 %
13F4	306.8	1.98	5.32	57.34 %

Table 4.12: 13F family on the V4 baseline (all runs with winglets): front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance against the 10F2.2 reference (10F2 re-run on V4). 13F1, the reduced inboard W4, is the only iteration above the reference; the C_{zt} values are on V4 and are not comparable to the V3 families.

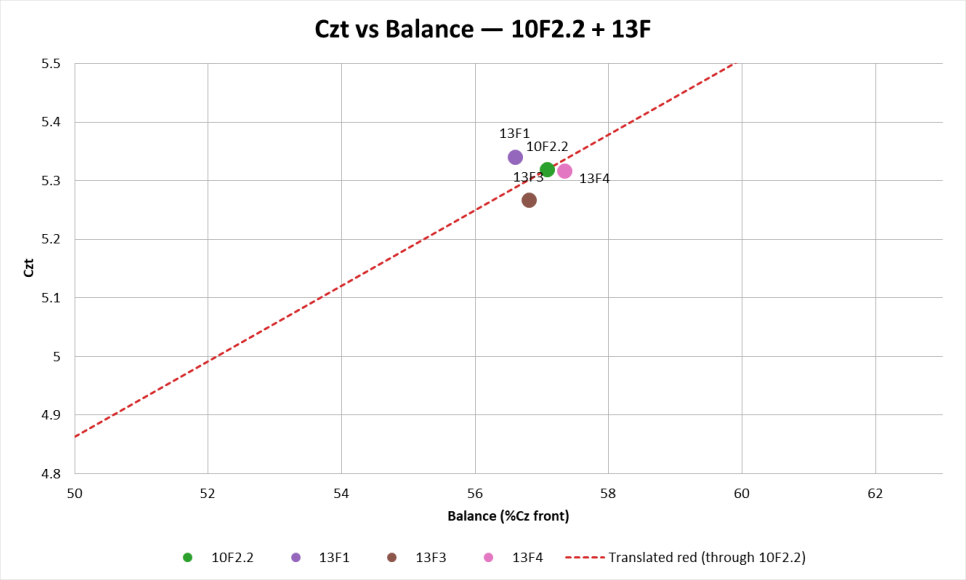
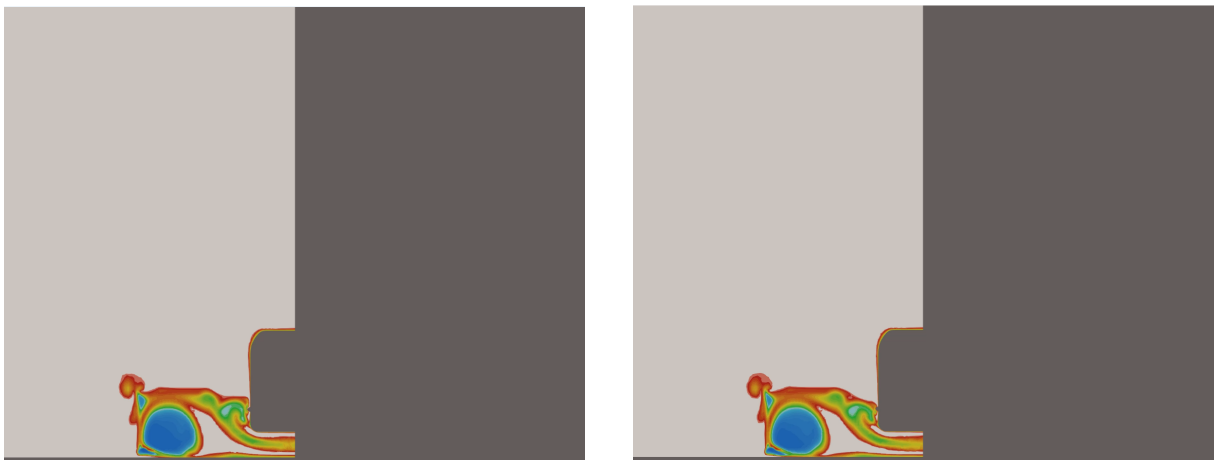


Figure 4.27: C_{zt} versus Front Balance for the 13F family on V4, against the 10F2.2 reference. The dashed line is the earlier reference-line slope translated to pass through 10F2.2. 13F1 sits above the line; 13F4 on it; 13F3 below.



(a) 10F2.2 (b) 13F1

Figure 4.28: C_{p_t} behind the wing at $T = 22\,\mathrm{s}$ for 10F2.2 and 13F1. The reduced inboard W4 of 13F1 leaves a larger white zone of recovered total pressure inboard, cleaner flow with less loss, which feeds the side structures downstream.

4.6.3 Convergence and strake studies

This phase closes the iteration history. It first converges the design on the V4 baseline with 16F, the wing that combines the two best directions found so far, and then examines the one remaining flow-management feature, the strakes. The two studies sit on different baselines: 16F is run on V4 against the 10F2.2 reference, while the strake study is run on V3 against 13F1.3 (the 13F1 wing re-run on V3), because the V4 CAD could not be meshed for the strake geometry and the study was moved back to V3 to obtain usable results.

16F: combining the best directions. The 16F family brought together the two directions that had worked best independently: the 10F2 wing and the 11F5 minimal-ramp mainplane (Section 4.6, 11F). 16F1 combines them on a single wing to test whether their gains add up. Like 13F, it was run on the V4 baseline against the 10F2.2 reference.

The combination lands at the top of the V4 cluster. 16F1 reaches $C_{zt} = 5.35$ at 57.5 % Balance, the highest C_{zt} of the V4 work, sitting just above the translated reference line and marginally ahead of 13F1 (Table 4.13, Figure 4.29). The margin is small: 16F1's C_{zt} edges 13F1's 5.34 by a few thousandths, and it does so at a slightly higher, more front-biased Balance (57.5 % against 56.6 %), so 13F1 remains marginally better placed relative to the 55 % target. On total downforce, however, 16F1 is the strongest wing the iteration history produced, and it is taken forward as the converged candidate for that reason: it offers the most C_{zt}, with the Balance to be trimmed to target in the second phase of the design (Section 4.1.3).

Iteration	FW downforce [N]	C_{zf}	C_{zt}	Balance
10F2.2 (V4 reference)	306.7	1.98	5.32	57.08 %
13F1	304.1	1.96	5.34	56.60 %
16F1	309.7	2.00	5.35	57.50 %

Table 4.13: 16F1 on the V4 baseline against the 10F2.2 reference and the previous best, 13F1. 16F1 combines the 10F2 wing with the 11F5 mainplane and reaches the highest C_{zt} of the V4 work. Values are on V4 and not comparable to the V3 families.

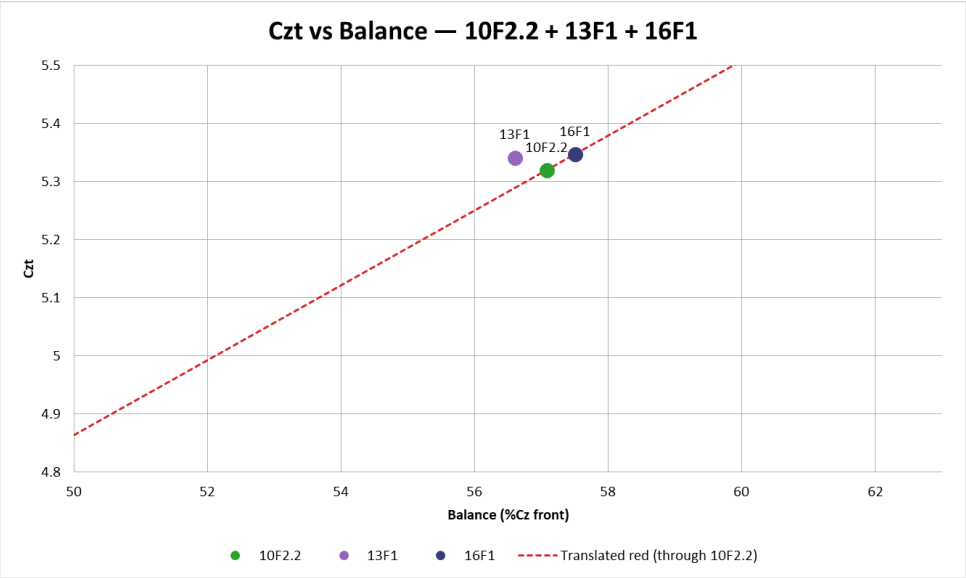


Figure 4.29: C_{zt} versus Front Balance for 16F1 on V4, against the 10F2.2 reference and 13F1. The dashed line is the reference-line slope translated through 10F2.2. 16F1 sits at the top of the cluster, just above the line.

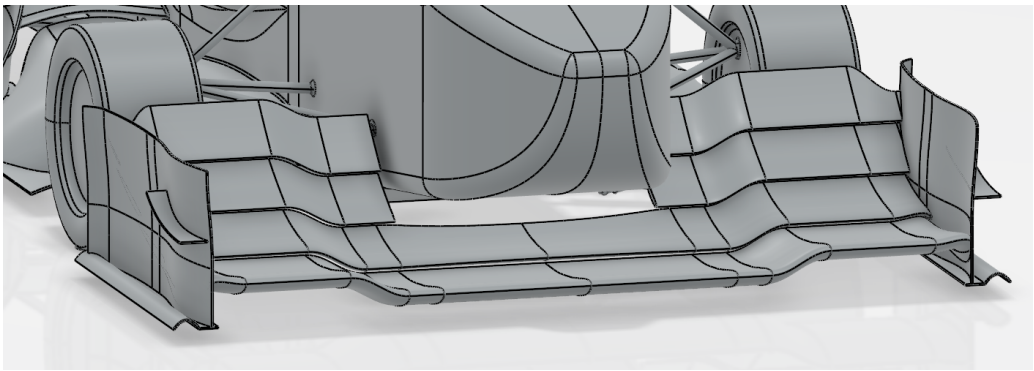


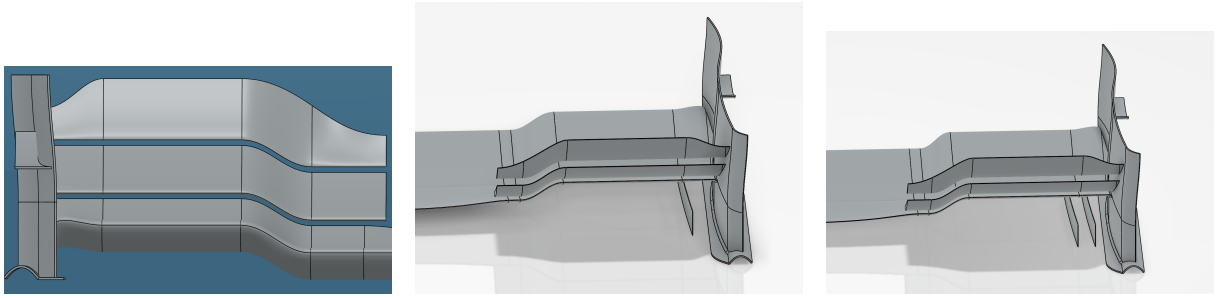
Figure 4.30: The 16F1 wing, combining the 10F2 geometry with the 11F5 minimal-ramp mainplane. This is the converged candidate carried forward to the results.

Strake study The final study looked at adding strakes to the wing, short vertical fences intended to organise the flow behind the elements. The idea was to use the strakes to nullify the burst behind the wing, shrinking it markedly: a smaller burst would raise the pressure on the wing and so its downforce, and, by sending fewer losses downstream, would also help the rest of the car. The study was run on the V3 baseline against 13F1.3, the 13F1 wing re-run on V3, because the V4 geometry could not be meshed with the strakes attached. The naming 13F13 to 13F16 denotes 13F1.3 with a strake added; these are strake variants of that wing, not a continuation of the inboard-W4 13F family.

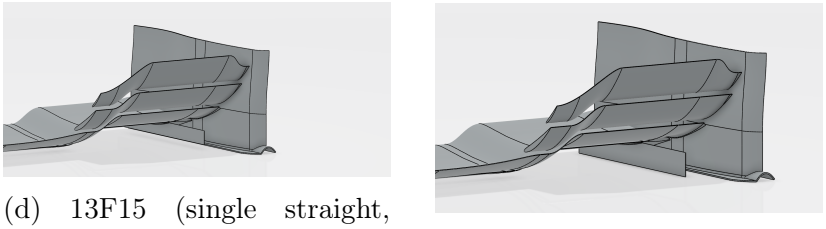
Four configurations were tried, varying the number of strakes, their curvature and their height (Figure 4.31). 13F13 is a single curved strake, 13F14 a double curved strake, 13F15 a single straight strake of shorter height, and 13F16 a single straight strake. The two best were the single curved 13F13 and the single straight 13F16. None of the four improved the whole-car C_{zt}. Every strake variant fell below the 13F1.3 reference on the C_{zt} versus Front Balance plot (Table 4.14, Figure 4.32): 13F13 at 5.30, 13F16 at 5.25, 13F15 at 5.21, and 13F14 worst at 5.07, against 5.35 for 13F1.3.

The post-processing, however, shows that the strakes did exactly what they were meant to do on the wing itself. The C_{p_t} slices for the curved 13F13 and the straight 13F16, each against 13F1.3 at $t = 23$ and $t = 31$ s (Figure 4.33), show a markedly smaller burst behind the wing with the strake fitted. The effect on the front load is real: 13F13 carries more front-wing downforce than the reference, 316.8 N against 311.2 N, with C_{zf} rising from 2.00 to 2.04. The smaller burst did raise the load on the wing.

The problem is downstream. Although the strakes added load at the front, the whole-car C_{zt} fell, because the strakes pushed more loss into the floor and side structures than they saved at the wing. The side-structure load drops from 179.2 N on 13F1.3 to 171.5 N on 13F13, the opposite of the intended effect: the strake moved load forward at the expense of the rest of the car. The one clear benefit the study delivered is the visual confirmation that a strake can nullify the burst; the net aerodynamic result was negative, and no strake configuration was adopted.



(a) 13F1.3 (no strake) (b) 13F13 (single curved) (c) 13F14 (double curved)



(d) 13F15 (single straight, shorter) (e) 13F16 (single straight)

Figure 4.31: Strake configurations added to the 13F1.3 wing. 13F13 is a single curved strake, 13F14 a double curved strake, 13F15 a single straight strake of shorter height, and 13F16 a single straight strake.

Iteration	FW downforce [N]	C_{zf}	C_{zt}	Balance
13F1.3 (reference, no strake)	311.2	2.00	5.35	59.50 %
13F13 (single curved)	316.8	2.04	5.30	60.00 %
13F14 (double curved)	304.6	1.96	5.07	60.20 %
13F15 (single straight, shorter)	305.3	1.97	5.21	58.70 %
13F16 (single straight)	307.5	1.98	5.25	58.30 %

Table 4.14: Strake study on the V3 baseline against the 13F1.3 reference: front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance.

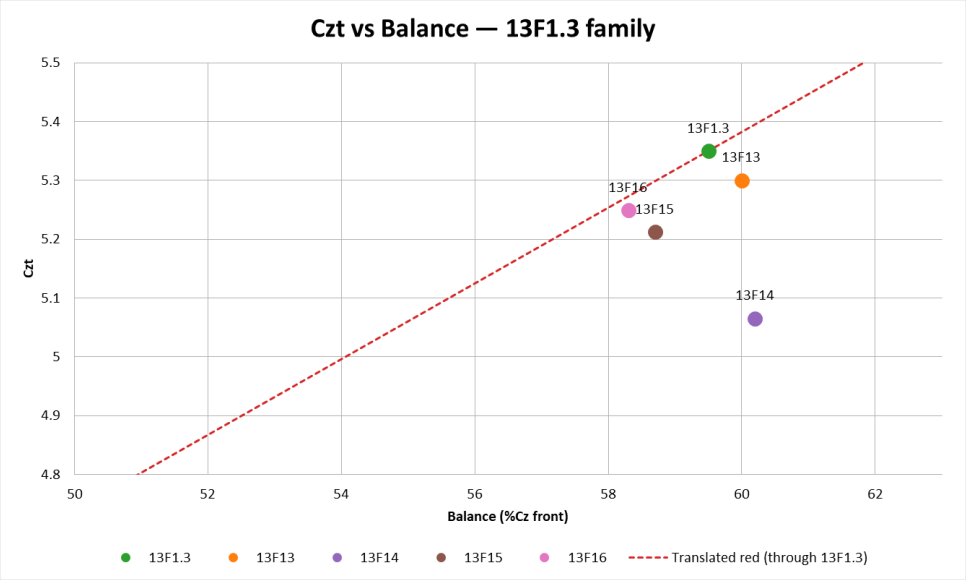
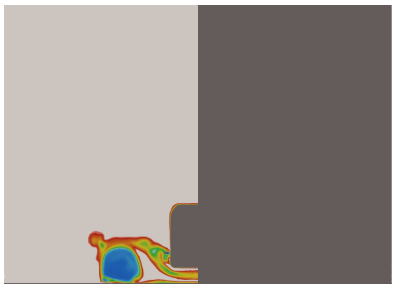
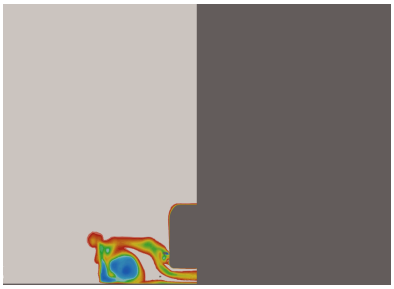


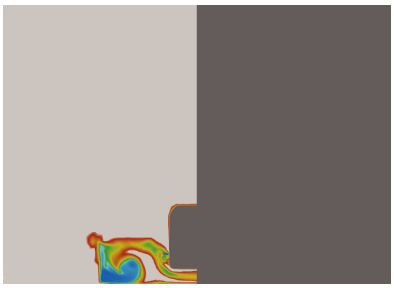
Figure 4.32: C_{zt} versus Front Balance for the strake study on V3, against the 13F1.3 reference. The dashed line is the reference-line slope translated through 13F1.3.



(a) 13F1.3, $t = 23\,\mathrm{s}$



(b) 13F13, $t = 23\,\mathrm{s}$



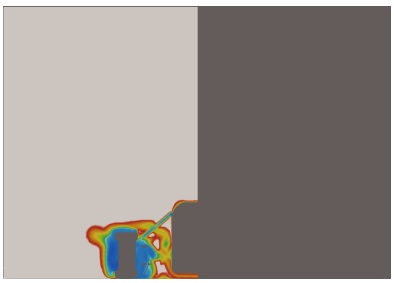
(c) 13F16, $t = 23\,\mathrm{s}$



(d) 13F1.3, $t = 31\,\mathrm{s}$



(e) 13F13, $t = 31\,\mathrm{s}$



(f) 13F16, $t = 31\,\mathrm{s}$

Figure 4.33: C_{p_t} behind the wing for the no-strake reference 13F1.3 and the two single-strake designs, curved 13F13 and straight 13F16, at $t = 23\,\mathrm{s}$ (top) and $t = 31\,\mathrm{s}$ (bottom).

4.7 Results and discussion

The iteration history converges on 16F1 as the front wing carried forward from this work. It is the strongest wing the study produced, combining the 10F2 geometry with the 11F5 minimal-ramp mainplane, and it sits at the top of the V4 cluster on the C_{zt} versus Front Balance plot. Its figures are collected in Table 4.15: a whole-car C_{zt} of 5.35 with a front-wing downforce of 309.7 N, at a Front Balance of 57.5 %.

Wing	FW downforce [N]	C_{zf}	C_{zt}	Balance
16F1 (V4)	309.7	2.00	5.35	57.50 %

Table 4.15: Converged front wing 16F1, evaluated on the V4 baseline: front-wing downforce, front-wing coefficient C_{zf}, whole-car coefficient C_{zt} and Front Balance.

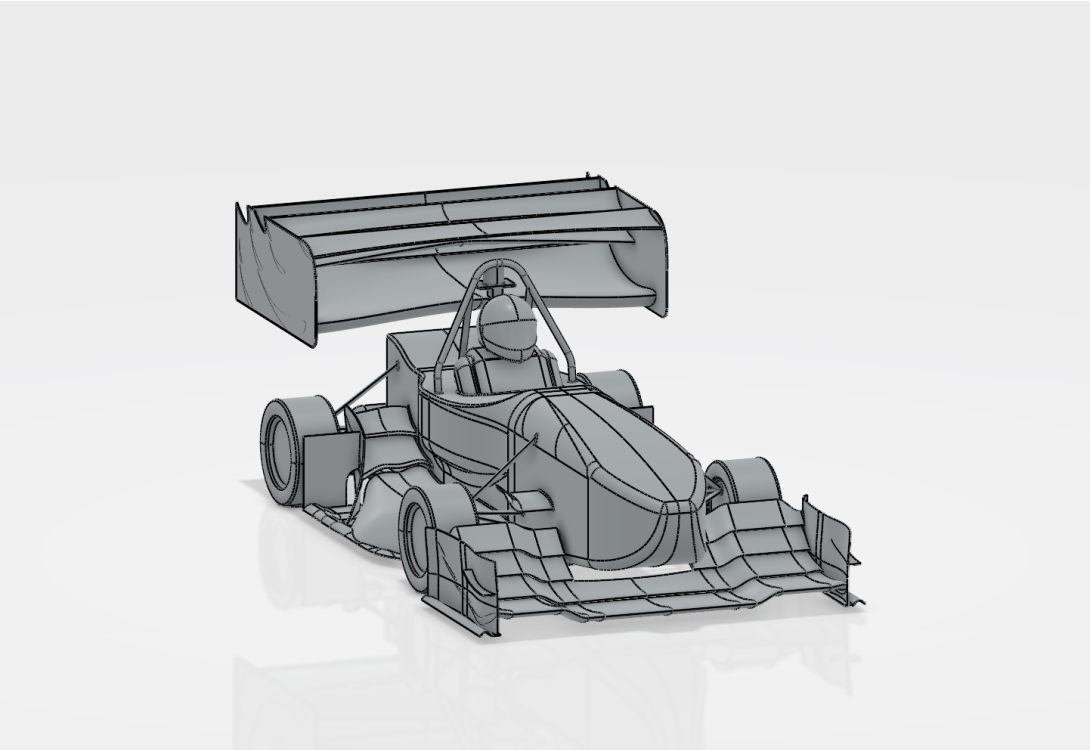


Figure 4.34: The converged front wing 16F1 on the complete V4 car, shown in context with the side structures, floor and rear of the vehicle that its wake feeds.

Two points qualify this result. The first is the baseline. 16F1 is reported on V4, the most recent baseline against which it was run; a V5 confirmation was not carried out as part of this project, so V4 is the baseline on which the converged wing stands. The second is the Front Balance. At 57.5 % the wing is more front-biased than the 55 % target. The intention is not to reach 55 % by shedding front load, which would give up the downforce this work was spent maximizing, but to raise rear downforce with the latest rear wing so that the front's share falls to 55 % while the front load is kept. That step depends on the rear wing, developed in parallel and outside the scope of this work, and on a whole-car simulation pairing 16F1 with it; it is not demonstrated here. The 55 % target is therefore approached, not closed, within this project.

These two qualifications aside, 16F1 meets the objective the work set itself: it is the converged maximum-downforce front wing produced under the draft 2027 envelope, the first deliverable of the two-phase methodology (Section 4.1.3), with the balance trim left as the second-phase step to be completed once the rear wing is final. Several directions that would improve the wing and its effect on the rest of the car, the endplates, a deeper strake study, the upper elements W3 and W4, and turning vanes, were outside the scope of this project and are set out as the necessary next steps in the conclusion.

4.8 Conclusion

This work set out to design a Formula Student front wing for the draft 2027 ruleset that reaches maximum downforce at 55 % Front Balance. It delivered the two things the objectives asked for (Section 4.1.3): a design methodology pairing the C_{zt} versus Front Balance plot with CFD post-processing, and a converged front-wing geometry produced by applying it across the iteration history.

4.8.1 Two strands, one open comparison

The iteration history ran across two whole-car baselines, and the leading wing on each is clear within its own baseline. On V3, the bulk of the work, the strongest directions were 10F2, the flattened outboard fourth element, and 11F5, the minimal-ramp mainplane. On V4, 16F1, which combines those two directions on a single wing, was the best, sitting above its 10F2.2 reference and ahead of 13F1.

These two strands cannot be ranked against each other directly. The V3 and V4 baselines differ in their rear-of-car configuration, and the V4 runs additionally carried winglets, so their C_{zt} values are not comparable: 16F1's V4 figure of 5.35 is lower in absolute terms than 10F2's V3 figure of 5.38, but that gap reflects the baseline and the winglets, not the wings. Nothing in the data shows 16F1 to be worse than 10F2. By design intent the opposite should hold, since 16F1 is 10F2 with the 11F5 mainplane added, so 16F1 should be at least as strong as 10F2; but the project cannot prove this, because the two were never run on a common baseline.

This is the one comparison the work leaves open, and closing it is the immediate next step. The leading candidates, 16F1 and the standalone 10F2, should be re-run on a single current baseline so they can be compared on equal terms, and the final model selected from that run rather than across baselines. Selecting the final wing is therefore a matter of refinement and confirmation, not of further exploration: the design has converged on 16F1, and a clean common-baseline run is what confirms it.

4.8.2 Reaching the balance target

16F1 sits at 57.5 % Front Balance, above the 55 % target. The intended route to 55 % is not to shed front load, which would surrender the downforce the work was spent maximizing, but to raise rear downforce with the latest rear wing so that the front's share falls to target while the front load is kept. That step requires a whole-car simulation pairing 16F1 with the final rear wing, and is the second phase of the methodology, left to be completed once the rear wing is final.

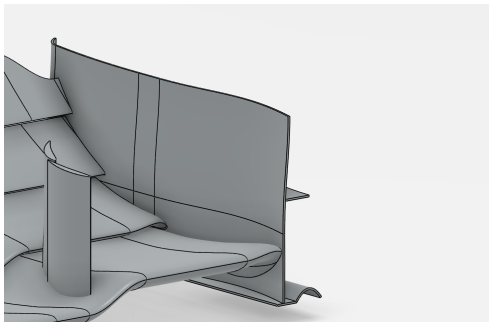
4.8.3 Future work

Several directions were outside the scope of this project but are the necessary next steps to improve the wing and, in particular, its effect on the rest of the car.

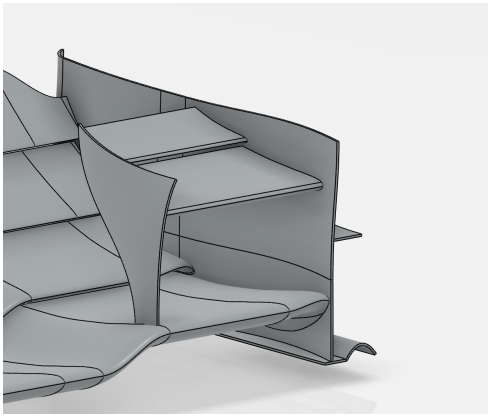
The endplates in the present design were sized only for their structural and tip-vortex roles (Section 4.4). A four-element wing loaded to this level pushes a strong, energetic wake into the region behind it, where it interacts with the front tire, the side structures, the floor, and the rear-wing inflow. Shaping the endplates, with footplates, outwash features, and tailored profiles, to steer that wake is a necessary step for improving the

performance of the car as a whole, and is the single largest piece of front-wing work left undone.

Three further devices were left unstudied. Turning vanes (Figure 4.35a) and small upper elements (Figure 4.35b), inboard flow-management surfaces that organise the air heading toward the rest of the car, were not analyzed in this project and are open areas of study. The strake study should also be taken further: it showed a strake can markedly shrink the burst behind the wing and even raise front load, but at the cost of sending more loss into the floor and side structures (Section 4.6). A deeper study could pursue the burst reduction while controlling that downstream penalty, which the present configurations did not.



(a) Turning vane in the inboard region.



(b) Small upper element above the inboard region (highlighted).

Figure 4.35: Two prototype flow-management devices, shown only to illustrate possible shapes. Neither was part of the converged wing or analyzed in this project, and both are open areas for future work.

Finally, the simulations to date omit the wing mounts. Adding the correct mounting hardware to the model would capture the losses those mounts introduce, which the current geometry does not, and is a necessary step toward a more representative whole-car result. This too remains to be studied.

Two lessons from the process are worth recording. First, the winglets should not have been carried on the baseline while the front wing was still being developed: as a downstream device, their shape depends on the flow the front wing delivers, and adding them early lowered the absolute C_{zt} and complicated comparison without serving their purpose. They are best designed once the front wing is locked. Second, when 8F6 and 8F8 emerged from the keep-out-zone study at effectively the same C_{zt}, only 8F8 was developed further; 8F6, which sat closer to the balance target, was never given its own derivative branch, and a parallel line of iterations from it was worth running. Both points are search and process choices rather than aerodynamic conclusions, and both would be handled differently in a second pass.

Conclusion

This report set out to develop the aerodynamic package of the 2027 EPFL Racing Team car under the draft 2027 FSG ruleset, and to do so in a way that is defensible rather than merely fast: every design choice tied to a measurement, and every measurement produced by a process that is consistent from one iteration to the next. The four chapters split that effort into the two tools that make the measurement reliable and the two wings that the measurement was used to design.

The two tooling chapters delivered a CFD process that is faster and more consistent than the manual one it replaces. The automation of the pipeline, from post-processing through to a full meshing-to-results workflow, cuts the user time spent on each iteration and removes the scaling and comparison errors that had previously misled design decisions. The 2D CFD workflow attacks the same bottleneck from the other side, letting wing cross-sections be explored in hours rather than days while staying anchored to the 3D model through validation. Neither tool reduces the number of iterations that aerodynamic optimization requires; both reduce the cost of each one, and both remain available to future members rather than ending with this project.

The two design chapters used that process to develop the front and rear wings, each converging on a four-element configuration under the draft 2027 envelope. The rear wing was carried from a three-element concept to a four-element one to increase rear downforce, and the front wing was pushed to maximize the load it carries before any trade was made. Both results are reported honestly, with the dead ends and the open comparisons left in rather than airbrushed out, so that what was achieved and what remains uncertain are both clear.

Taken together, the four chapters leave the team in a stronger position than the individual results alone suggest. The design work produced two converged wings, but the more durable outcome is the way they were produced: a CFD process that is faster and consistent, a 2D workflow that makes exploration cheap, and a design methodology in which each choice is tied to a measurement and each measurement is trusted. That foundation is what let a clean-sheet package be developed against a ruleset that is still in draft, and it is what will carry into the next car. The project does not claim a finished package; it claims a set of validated tools, two converged components, and an honest account of what each is and is not, which is the right basis from which the 2027 car can be brought to the grid.

Chapter 5

Annex

5.1 Paraview automation complete code

5.1.1 ParaScript_baseline

```
# IMPORTANT INFORMATION FOR EDITING THIS SCRIPT:
# follow the comments and only edit the things which they mention.
# in general, you can edit any CameraPosition array. This gives the cartesian coordinates of where the viewing will originate. CameraOrientation is the focal point of the camerea
# If you want to get a closer look at anything of move anything dynamically around in any function change Interact from False to True.
# Note that if you do want to interact the script will stop at this point and only continue running when you close the window which you are looking at


# imports needed for plotting

import argparse
import os
from paraview.simple import *
from paraview import servermanager
from ParaFunctions import *

# get user input variables:
parser = argparse.ArgumentParser()

parser.add_argument("-d", "--dir", required = True, type = str ,help = "Directory with files")
parser.add_argument("-f", "--file", required = True, type = str ,help = "name of .cas and .dat files")
parser.add_argument("-sI", "--saveImages", default = False, type = bool ,help = "set saving of images True")

args = parser.parse_args()


# path to files
file_path = str(args.dir)
caseFile = os.path.join(file_path, str(args.file) + '.cas.h5')
datFile = os.path.join(file_path, str(args.file) + '.dat.h5')
print(args)
```

```
print(caseFile)
print(datFile)

#### disable automatic camera reset on 'Show'
paraview.simple._DisableFirstRenderCameraReset()

# create a new 'Fluent CFF Case Reader'
# reading of the case file, does not need to explicitly read in the data file since it is linked in the case file
cash5 = FluentCFFCaseReader(registrationName = str(args.file) + '.cas.h5', FileName = [caseFile])



# compute and plot cp, requires the free stream velocity if you change the value it is required to change it here as well
# input free stream velocity in m/s
U_infty = 20

#Ranges for each variabl:
Range_SV_U = [-10, 35]
Range_SV_P = [-600, 150]
Range_SV_Mag = [0, 42]
Range_Cp = [-4, 1]
Range_CpT=[-1.3,1]

# visualize the solution (any field possible here)
# visualize the solution (any field possible here)
# Can edit the "variable" section into any of the default variables exported by fluent


#Visualization of the car in "isometric" view
isometric_view= 'isometric_view'
cameraPosition = [-6, 4, 2]
cameraOrientation = [-2, 1, 1]

Visulaize3DFieldSurface(cash5, "SV_U", Range_SV_U, cameraPosition, cameraOrientation,[0,0,1], False, args.saveImages, str(args.dir), isometric_view)



# Slices
sliceDir = os.path.join(args.dir, "slices_Cp")
os.makedirs(sliceDir, exist_ok=True)
sliceDir = os.path.join(args.dir, "slices_SV_Mag")
os.makedirs(sliceDir, exist_ok=True)
# Define slicing inputs here:
# definition of equidistant slices in the direction given by the sliceNormal vector
```

```
slicePositions = [] # keep this array empty. It will get filled later on
sliceNormal = [0, 1, 0] # vector normal to the plane slice
#list the y values you want slices of in the following array :
y_values = [0.01, 0.2, 0.275, 0.3, 0.4, 0.525, 0.65]


#Slices of normal y plotting Cp and CpT
for sliceCoordinate in y_values:
    slicePositions.append([0, sliceCoordinate, 0])
    cameraPosition = [-1, sliceCoordinate-4.8, 1.3]
    cameraOrientation = [-1, sliceCoordinate, 1.3]
    imageName = f"{args.file}_y_{sliceCoordinate:.3f}"
    VisualizePlaneCutsCp(cash5, "Cp", Range_Cp, slicePositions, sliceNormal,
    ↪  cameraPosition, cameraOrientation , False, args.saveImages,
    ↪  str(args.dir), imageName, U_infty)
    VisualizePlaneCutsSV_Mag(
        cash5, "SV_Mag", Range_SV_Mag, slicePositions, sliceNormal,
        ↪  cameraPosition, cameraOrientation,False, args.saveImages,
        ↪  str(args.dir), imageName)
    slicePositions=[] #this ensures that you get a single slice by image,
    ↪  initally the code is meant to make multiple slices on a single image,
    ↪  the array slicePositions is still here to allow easy implementation of
    ↪  this if needed



#
#CalcAndPlotCp(cash5, U_infty, cameraPosition, cameraOrientation, False,
↪  args.saveImages, str(args.dir))
#
# plot the surface distribution of some value (here Cp)
# This script is not able to interpolate field values onto the surface of the
↪  car
# Thus we simply display our fields very close to the car, by coloring every
↪  cell which is less than WallnormalCoord away from the surface of the car


#Plotting the surface distribution of Cp
WallnormalCoord = 0.0007
clipZ = 0.005

# Floor view :
cameraPosition = [-1, 0.7, -4]
cameraOrientation = [-1, 0.7, -1]
cameraUp = [0, 1, 0]
zoom = 1
imageName = "Floor_view"

PlotSurfaceDistributionVariable(cash5,'Cp',Range_Cp,U_infty,
↪  WallnormalCoord,cameraPosition,cameraOrientation,cameraUp,zoom,False,args.saveImages,str(
```

```
115 #Rear view :
116 cameraPosition = [2.5, 0.5, 0.57]
117 cameraOrientation = [0, 0.5, 0.57]
118 cameraUp = [0, 0, 1]
119 zoom = 1
120 imageName = "Rear_view"
121 PlotSurfaceDistributionVariable(cash5,'Cp',Range_Cp,U_infty,
    ↪ WallnormalCoord,cameraPosition,cameraOrientation,cameraUp,zoom,False,args.saveImages,str(
122
123
124 #Rear wing bottom view :
125 cameraPosition = [0, 0.3, -1.5]
126 cameraOrientation = [0, 0.3, 1]
127 cameraUp = [0, 1, 0]
128 zoom = 1
129 clipZ = 0.63
130 imageName = "Rear_wing_bottom_view"
131 PlotSurfaceDistributionVariable(cash5,'Cp',Range_Cp,U_infty,
    ↪ WallnormalCoord,cameraPosition,cameraOrientation,cameraUp,zoom,False,args.saveImages,str(
132 # number of streamlines desired
133 StreamlineDensity = 100
134
135
136
137 # plotting of slices of CpT:
138 # definition of slices of the xy plane and yz plane (exclusive function for CPT
    ↪ due to extra calculation needed)
139
140 sliceDir = os.path.join(args.dir, "slices_CpT")
141 os.makedirs(sliceDir, exist_ok=True)
142
143 # Define slicing inputs here:
144 #list the y values you want slices of in the following array :
145 y_values = [0.01, 0.2, 0.275, 0.3, 0.4, 0.525, 0.65]
146
147 slicePositions = [] # keep this array empty. It will get filled later on
148
149
150
151 for sliceCoordinate in y_values:
152
153     slicePositions.append([0, sliceCoordinate, 0])
154
155     cameraPosition = [-1, sliceCoordinate-4.8, 1.3]
156     cameraOrientation = [-1, sliceCoordinate, 1.3]
157     imageName = f"{args.file}_y_{sliceCoordinate:.3f}"
158     PlotCpT_xy_yz(cash5, U_infty,Range_CpT, [0,sliceCoordinate,0], [],
    ↪ cameraPosition, cameraOrientation, False, args.saveImages,
    ↪ str(args.dir), imageName)
159     slicePositions=[]
160
```

```

# extraction of a slicing movie
# subsequent slicing of the domain and extraction of a movie showing the
↪  evolution of 'variable' in the domain with limits to the color space
↪  'lims'
cameraPosition = [-6.5, 0, 1]
cameraOrientation = [0, 0, 1]


x_start = -13.5
x_end = 0
#pressure coefficient Cp movie
sliceMovieCp(cash5, U_infty, Range_Cp, x_start, x_end, cameraPosition,
↪  cameraOrientation, False, True, args.dir, 'Cp')

#CpT movie
sliceMovieCpT(cash5,U_infty, Range_CpT, x_start, x_end, cameraPosition,
↪  cameraOrientation, False, True, args.dir,'CpT')


cameraPosition = [-1.2, -4.8, 1.3]
cameraOrientation = [-1.2, 0.35, 1.3]

sliceMovieSV_Mag( cash5,Range_SV_Mag,0.01,0.75,
↪  cameraPosition,cameraOrientation,False,True,
↪  args.dir,'SV_Mag_y_movie',num_frames=70,image_resolution=[2160, 1440])


```

5.1.2 ParaFunctions

```
from cmath import isclose

from paraview.simple import *
import numpy as np
#from pyvista import CameraPosition


# readCase: the casefile that has been read into Paraview
# variable: string which defines the variable which is to be shown
# lims: limits for colour display
# cameraPosition: Array with Position of camera
# cameraOrientation: Array with focus point of Camera
# interact: bool for interaction or not: default = False
# save: bool whether to save data or not
# directory: string of directory where the final view is to be saved.
# ============================================================
# Global visualization option
```

```
# True  -> convert cell data to point data (smooth slices)
# False -> use original cell data
# ============================================================

USE_POINT_DATA = True


def get_data_source(source):
    if USE_POINT_DATA:
        return CellDatatoPointData(Input=source)
    else:
        return source


def data_type():
    if USE_POINT_DATA:
        return 'POINTS'
    else:
        return 'CELLS'

#To improve memory efficiency once an image is exported the corresponding
↪  paraview window and all pyhton objects associated with it will be deleted
↪  by the following function :
def cleanup_view(renderView, *objects):
    # delete pipeline objects first
    for obj in objects:
        try:
            if obj is not None:
                Delete(obj)
        except:
            pass

    # delete the render view itself
    try:
        Delete(renderView)
    except:
        pass

    # remove Python references
    del renderView

def ApplyCpTColormap(cptLUT, lims):
    cpt_min = lims[0]
    cpt_max = lims[1]
    cpt_center = 1.0

    if cpt_min < cpt_center < cpt_max:
        # Below 1: Blue
        low1 = cpt_min + 0.20 * (cpt_center - cpt_min)
        low2 = cpt_min + 0.55 * (cpt_center - cpt_min)
        low3 = cpt_min + 0.85 * (cpt_center - cpt_min)
```

```
67
68          # Above 1: Red (shows non-physicality in the simulation)
69          high1 = cpt_center + 0.40 * (cpt_max - cpt_center)
70
71          cptLUT.ColorSpace = 'Lab'
72          cptLUT.RGBPoints = [
73              cpt_min, 0.120, 0.050, 0.450,  # lowest values: blue-purple
74              low1, 0.180, 0.180, 0.700,  # purple-blue
75              low2, 0.250, 0.500, 0.900,  # strong blue
76              low3, 0.820, 0.900, 0.980,  # pale blue near white
77              1.00, 1.000, 1.000, 1.000,  # white at CpT = 1
78              high1, 0.995, 0.965, 0.945,  # keep above-1 values almost white
79              cpt_max, 0.750, 0.150, 0.120  # red only near the very top
80          ]
81      elif isclose(cpt_max,1):
82          # Negative side: cpt_min -> 0
83          low1 = cpt_min + 0.25 * (0.0 - cpt_min)
84          low2 = cpt_min + 0.60 * (0.0 - cpt_min)
85          low3 = cpt_min + 0.90 * (0.0 - cpt_min)
86
87          # Positive side: 0 -> 1
88          green_pt = 0.10
89          yellow_pt = 0.50
90          red_pt = 0.92
91
92          cptLUT.RGBPoints = [
93              # Negative values: blue -> cyan
94              cpt_min, 0.080, 0.120, 0.700,  # deep blue
95              low1, 0.100, 0.250, 0.850,  # blue
96              low2, 0.140, 0.500, 0.920,  # light blue
97              low3, 0.250, 0.780, 0.950,  # cyan-ish near 0
98              0.00, 0.700, 0.950, 0.980,  # light cyan at 0
99
100             # Positive values: cyan -> green -> yellow -> red -> white
101             green_pt, 0.000, 0.800, 0.250,  # green around 0.1
102             yellow_pt, 0.980, 0.900, 0.150,  # yellow at 0.5
103             red_pt, 0.850, 0.120, 0.120,  # red just below 1
104             1.00, 1.000, 1.000, 1.000  # white at 1
105         ]
106         cptLUT.Discretize = 1
107         cptLUT.NumberOfTableValues = 48
108     else:
109         print("Warning: CpT range does not include 1, skipping custom
        ↪  white-centered CpT colormap.")
110
111 def ApplyCpColormap(cpLUT, lims):
112     # Set white to be Cp=0
113     cp_min = lims[0]
114     cp_max = lims[1]
115
116     # Negative side
```

```
    neg_cyan1 = 0.06 * cp_min  # very close to 0
    neg_cyan2 = 0.18 * cp_min
    neg_lightblue = 0.35 * cp_min
    neg_mid = 0.50 * cp_min  # middle of negative range
    neg_blue2 = 0.72 * cp_min
    neg_blue3 = 0.88 * cp_min

    # Positive side
    pos1 = 0.08 * cp_max
    pos2 = 0.35 * cp_max
    pos3 = 0.70 * cp_max

    cpLUT.ColorSpace = 'Lab'
    cpLUT.RGBPoints = [
        # Negative side: cp_min -> 0
        cp_min, 0.070, 0.100, 0.650,  # deep blue
        neg_blue3, 0.080, 0.180, 0.760,  # strong deep blue
        neg_blue2, 0.100, 0.300, 0.840,  # blue
        neg_mid, 0.130, 0.450, 0.900,  # clear blue at cp_min/2
        neg_lightblue, 0.220, 0.650, 0.930,  # light blue
        neg_cyan2, 0.420, 0.820, 0.900,  # soft cyan-blue
        neg_cyan1, 0.680, 0.900, 0.930,  # less saturated cyan just below 0
        0.00, 0.960, 0.960, 0.960,  # white at Cp = 0

        # Positive side
        pos1, 0.980, 0.920, 0.700,  # pale yellow
        pos2, 0.950, 0.650, 0.200,  # orange
        pos3, 0.850, 0.350, 0.100,  # orange-red
        cp_max, 0.650, 0.000, 0.000  # dark red
    ]
    cpLUT.Discretize = 1
    cpLUT.NumberOfTableValues = 36


def Visulaize3DFieldSurface(readCase, variable, lims, cameraPosition,
↪   cameraOrientation, up, interact, save, directory, imageName):
    source = get_data_source(readCase)
    renderView = CreateView('RenderView')
    renderView.ViewSize = list((3000, 2000))
    Display = Show(source, renderView, 'UnstructuredGridRepresentation')
    Display.Representation = 'Surface'

    # reset view to fit data
    renderView.ResetCamera(False, 0.9)
    renderView.Update()
    ColorBy(Display, (data_type(), variable))
    varColorFunction = GetColorTransferFunction(variable)
    varOpacityFunction = GetOpacityTransferFunction(variable)
    var2DFunction = GetTransferFunction2D(variable)

    varColorFunction.RescaleTransferFunction(lims[0], lims[1])
```

```
    varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
    var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

    Display.SetScalarBarVisibility(renderView, True)

    # current camera placement for renderView1
    renderView.CameraPosition = cameraPosition
    renderView.CameraFocalPoint = cameraOrientation  # What it's looking at
    renderView.CameraViewUp = up  # Which direction is "up"
    renderView.CameraParallelScale = 1  # Zoom level

    if interact:
        Interact()

    if save:
        ExportView(directory + '\\' +imageName + '_' +variable + '.pdf',
        ↪  view=renderView)
    cleanup_view(renderView, Display, source)
# position: array of positions for slices
# normals: corresponding normal vector of slices to be produced

def VisualizePlaneCuts(readCase, variable, lims, slicePosition, sliceNormal,
↪  cameraPosition, cameraOrientation, interact, save, directory, imageName):
    source = get_data_source(readCase)
    renderView = CreateView('RenderView')
    slices = []

    Display = Show(source, renderView, 'UnstructuredGridRepresentation')
    Display.Representation = 'Surface'

    # reset view to fit data
    renderView.ResetCamera(False, 0.9)
    renderView.Update()
    ColorBy(Display, (data_type(), variable))
    varColorFunction = GetColorTransferFunction(variable)
    varOpacityFunction = GetOpacityTransferFunction(variable)
    var2DFunction = GetTransferFunction2D(variable)

    varColorFunction.RescaleTransferFunction(lims[0], lims[1])
    varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
    var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

    Display.SetScalarBarVisibility(renderView, True)

    print("Current slice positions:")

    for (idxSlice, position) in enumerate(slicePosition):
        print(idxSlice, position)
        slice = Slice(registrationName = variable + 'Slice', Input =source)
        slice.SliceType.Origin = position
        slice.SliceType.Normal = sliceNormal
```

```
216             # get active view
217             renderView = GetActiveViewOrCreate('RenderView')
218 
219             # show data in view
220             sliceDisplay = Show(slice, renderView, 'GeometryRepresentation')
221 
222             # trace defaults for the display properties.
223             sliceDisplay.Representation = 'Surface'
224 
225             Hide(source, renderView)
226 
227             # show color bar/color legend
228             sliceDisplay.SetScalarBarVisibility(renderView, True)
229 
230             # update the view to ensure updated data information
231             renderView.Update()
232 
233             slices.append(slice)
234 
235         # Set camera parameters
236         renderView.CameraPosition = cameraPosition  # Where the camera is
237         renderView.CameraFocalPoint = cameraOrientation  # What it's looking at
238         renderView.CameraViewUp = [0, 0, 1]  # Which direction is "up"
239         renderView.CameraParallelScale = 1.0  # Zoom level
240         renderView.ViewSize = list((3000, 2000))
241         if interact:
242             Interact()
243 
244         if save:
245             ExportView( directory + '\\' + 'slices'+ '\\' + variable +'_' +
                ↪  imageName + 'slices.pdf', view = renderView)
246         cleanup_view(renderView, Display, source, *slices)
247 
248 def VisualizePlaneCutsCp(readCase, variable, lims, slicePosition, sliceNormal,
    ↪  cameraPosition, cameraOrientation, interact, save, directory, imageName,
    ↪  U_infty):
249     source = get_data_source(readCase)
250     renderView = CreateView('RenderView')
251     slices = []
252 
253     # ---- Cp calculation ----
254     CpCalculator = Calculator(registrationName='Cp', Input=readCase)
255     U_sqr = str(U_infty * U_infty)
256     CpCalculator.Function = 'SV_P / (0.5 * SV_DENSITY * ' + U_sqr + ')'
257     CpCalculator.ResultArrayName = 'cp'
258 
259     CpSource = get_data_source(CpCalculator)
260 
261     Display = Show(CpSource, renderView, 'UnstructuredGridRepresentation')
262     Display.Representation = 'Surface'
263 
```

```
    renderView.ResetCamera(False, 0.9)
    renderView.Update()

    ColorBy(Display, (data_type(), 'cp'))

    varColorFunction = GetColorTransferFunction('cp')
    varOpacityFunction = GetOpacityTransferFunction('cp')
    var2DFunction = GetTransferFunction2D('cp')

    varColorFunction.RescaleTransferFunction(lims[0], lims[1])
    varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
    var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
    ApplyCpColormap(varColorFunction,lims)
    Display.SetScalarBarVisibility(renderView, True)

    print("Current slice positions:")

    for (idxSlice, position) in enumerate(slicePosition):
        print(idxSlice, position)

        slice = Slice(registrationName='CpSlice', Input=CpSource)
        slice.SliceType.Origin = position
        slice.SliceType.Normal = sliceNormal

        renderView = GetActiveViewOrCreate('RenderView')

        sliceDisplay = Show(slice, renderView, 'GeometryRepresentation')
        sliceDisplay.Representation = 'Surface'

        Hide(CpSource, renderView)

        sliceDisplay.SetScalarBarVisibility(renderView, True)

        renderView.Update()

        slices.append(slice)
    varColorFunction = GetColorTransferFunction('cp')
    varOpacityFunction = GetOpacityTransferFunction('cp')
    var2DFunction = GetTransferFunction2D('cp')

    varColorFunction.RescaleTransferFunction(lims[0], lims[1])
    varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
    var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
    ApplyCpColormap(varColorFunction, lims)
    Display.SetScalarBarVisibility(renderView, True)
    renderView.CameraPosition = cameraPosition
    renderView.CameraFocalPoint = cameraOrientation
    renderView.CameraViewUp = [0, 0, 1]
    renderView.CameraParallelScale = 1.0
    renderView.ViewSize = list((3000, 2000))

```

```
    if interact:
        Interact()

    if save:
        ExportView(directory + '\\' + 'slices_Cp' + '\\' + 'Cp_' + imageName +
        ↪  'slices.pdf', view=renderView)
    cleanup_view(renderView, Display, CpSource, CpCalculator, source, *slices)

def VisualizePlaneCutsSV_Mag(readCase, variable, lims, slicePosition,
↪  sliceNormal, cameraPosition, cameraOrientation, interact, save, directory,
↪  imageName):
    source = get_data_source(readCase)
    renderView = CreateView('RenderView')
    slices = []

    # ---- Velocity magnitude calculation ----
    VelMagCalculator = Calculator(registrationName='VelMag', Input=readCase)
    VelMagCalculator.Function = 'sqrt(SV_U^2 + SV_V^2 + SV_W^2)'
    VelMagCalculator.ResultArrayName = 'SV_Mag'

    VelMagSource = get_data_source(VelMagCalculator)

    Display = Show(VelMagSource, renderView, 'UnstructuredGridRepresentation')
    Display.Representation = 'Surface'

    renderView.ResetCamera(False, 0.9)
    renderView.Update()

    ColorBy(Display, (data_type(), 'SV_Mag'))

    varColorFunction = GetColorTransferFunction('SV_Mag')
    varOpacityFunction = GetOpacityTransferFunction('SV_Mag')
    var2DFunction = GetTransferFunction2D('SV_Mag')

    varColorFunction.RescaleTransferFunction(lims[0], lims[1])
    varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
    var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

    Display.SetScalarBarVisibility(renderView, True)

    print("Current slice positions:")

    for (idxSlice, position) in enumerate(slicePosition):
        print(idxSlice, position)

        slice = Slice(registrationName='VelMagSlice', Input=VelMagSource)
        slice.SliceType.Origin = position
        slice.SliceType.Normal = sliceNormal

        renderView = GetActiveViewOrCreate('RenderView')

```

```
363         sliceDisplay = Show(slice, renderView, 'GeometryRepresentation')
364         sliceDisplay.Representation = 'Surface'
365 
366         Hide(VelMagSource, renderView)
367 
368         sliceDisplay.SetScalarBarVisibility(renderView, True)
369 
370         renderView.Update()
371 
372         slices.append(slice)
373 
374     # enforce limits again (same pattern as Cp function)
375     varColorFunction = GetColorTransferFunction('SV_Mag')
376     varOpacityFunction = GetOpacityTransferFunction('SV_Mag')
377     var2DFunction = GetTransferFunction2D('SV_Mag')
378 
379     varColorFunction.RescaleTransferFunction(lims[0], lims[1])
380     varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
381     var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
382 
383     Display.SetScalarBarVisibility(renderView, True)
384 
385     renderView.CameraPosition = cameraPosition
386     renderView.CameraFocalPoint = cameraOrientation
387     renderView.CameraViewUp = [0, 0, 1]
388     renderView.CameraParallelScale = 1.0
389     renderView.ViewSize = list((3000, 2000))
390 
391     if interact:
392         Interact()
393 
394     if save:
395         ExportView(directory + '\\' + 'slices_SV_Mag' + '\\' + 'SV_Mag_' +
        ↪  imageName + 'slices.pdf', view=renderView)
396 
397     cleanup_view(renderView, Display, VelMagSource, VelMagCalculator, source,
    ↪  *slices)
398 
399 def CalcAndPlotCp(readCase, U_infty, cameraPosition, cameraOrientation,
↪   interact, save, directory):
400     source = get_data_source(readCase)
401     renderView = CreateView('RenderView')
402     Display = Show(source, renderView, 'UnstructuredGridRepresentation')
403     Display.Representation = 'Surface'
404     renderView.ResetCamera(False, 0.9)
405     renderView.Update()
406 
407     CpCalculator = Calculator(registrationName='Cp', Input=readCase)
408 
409     # Properties modified on calculator1
410     U_sqr = str((U_infty * U_infty))
```

```
    CpCalculator.Function = 'SV_P / (0.5 * SV_DENSITY * ' + U_sqr +')'
    CpCalculator.ResultArrayName = 'cp'

    # show data in view
    CpDisplay = Show(CpCalculator, renderView,
    ↪  'UnstructuredGridRepresentation')

    # trace defaults for the display properties.
    CpDisplay.Representation = 'Surface'

    #
    CpDisplay.SetScalarBarVisibility(renderView, True)
    renderView.CameraPosition = cameraPosition  # Where the camera is
    renderView.CameraFocalPoint = cameraOrientation  # What it's looking at
    renderView.CameraViewUp = [0, 1, 0]  # Which direction is "up"
    renderView.CameraParallelScale = 1.0  # Zoom level
    renderView.ViewSize = list((3000, 2000))
    #
    if interact:
        Interact()

    if save:
        ExportView(directory + '\\' +  'coefficient_of_pressure.pdf',
        ↪  view=renderView)

def PlotSurfaceDistributionVariable(
    readCase,
    variable,
    lims,
    U_infty,
    WallNormalCoordinate,
    cameraPosition,
    cameraOrientation,
    viewUp,
    zoom,
    interact,
    save,
    directory,
    imageName,
    clipZ=None,
    keepAboveClip=True
):
    source = get_data_source(readCase)
    renderView = CreateView('RenderView')
    Display = Show(source, renderView, 'UnstructuredGridRepresentation')
    Display.Representation = 'Surface'
    renderView.ResetCamera(False, 0.9)
    renderView.Update()

    threshold = Threshold(registrationName='WallThreshold', Input=readCase)
    threshold.Set(
```

```
        Scalars=['CELLS', 'SV_WALL_DIST'],
        UpperThreshold=WallNormalCoordinate,
    )

    currentSource = threshold

    # Optional clip to remove ground / lower part of domain
    if clipZ is not None:
        clip = Clip(registrationName='GroundClip', Input=threshold)
        clip.ClipType = 'Plane'
        clip.ClipType.Origin = [0.0, 0.0, clipZ]
        clip.ClipType.Normal = [0.0, 0.0, 1.0]


        try:
            clip.Invert = not keepAboveClip
        except:
            clip.InsideOut = keepAboveClip

        currentSource = clip

    currentDisplay = Show(currentSource, renderView,
    ↪  'UnstructuredGridRepresentation')
    currentDisplay.Representation = 'Surface'
    Hide(source, renderView)
    renderView.Update()

    if variable == 'Cp':
        CpCalculator = Calculator(registrationName='Cp', Input=currentSource)
        U_sqr = str((U_infty * U_infty))
        CpCalculator.Set(
            ResultArrayName='cp',
            Function='SV_P / (0.5 * SV_DENSITY * ' + U_sqr + ')',
        )

        CpDisplay = Show(CpCalculator, renderView,
        ↪  'UnstructuredGridRepresentation')
        CpDisplay.Representation = 'Surface'
        Hide(currentSource, renderView)
        CpDisplay.SetScalarBarVisibility(renderView, True)
        renderView.Update()

        cpLUT = GetColorTransferFunction('cp')
        cpLUT.RescaleTransferFunction(lims[0], lims[1])
        cpPWF = GetOpacityTransferFunction('cp')
        cpPWF.RescaleTransferFunction(lims[0], lims[1])
        cpTF2D = GetTransferFunction2D('cp')
        cpTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

        ApplyCpColormap(cpLUT,lims)
```

```
509
510         renderView.CameraPosition = cameraPosition
511         renderView.CameraFocalPoint = cameraOrientation
512         renderView.CameraViewUp = viewUp
513         renderView.CameraParallelScale = zoom
514         renderView.ViewSize = list((3000, 2000))
515
516     else:
517         ColorBy(currentDisplay, (data_type(), variable))
518         varLUT = GetColorTransferFunction(variable)
519         varPWF = GetOpacityTransferFunction(variable)
520         varTF2D = GetTransferFunction2D(variable)
521
522         varLUT.RescaleTransferFunction(lims[0], lims[1])
523         varPWF.RescaleTransferFunction(lims[0], lims[1])
524         varTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
525
526         currentDisplay.SetScalarBarVisibility(renderView, True)
527
528         renderView.CameraPosition = cameraPosition
529         renderView.CameraFocalPoint = cameraOrientation
530         renderView.CameraViewUp = viewUp
531         renderView.CameraParallelScale = zoom
532         renderView.ViewSize = list((3000, 2000))
533
534     if interact:
535         Interact()
536
537     if save:
538         ExportView(directory + '\\' + imageName + '_' + variable + '_' +
        ↪  'surface_plot.pdf', view=renderView)
539     objs_to_delete = [currentDisplay, Display, threshold, source]
540
541     if clipZ is not None:
542         objs_to_delete.append(clip)
543
544     if variable == 'Cp':
545         objs_to_delete.extend([CpDisplay, CpCalculator])
546
547     cleanup_view(renderView, *objs_to_delete)
548 def PlotStreamlines(readCase, startPoint, endpoint, number_of_lines,
    ↪  WallNormalCoordinate, cameraPosition, cameraOrientation, interact, save,
    ↪  directory):
549
550     renderView = CreateView('RenderView')
551     Display = Show(readCase, renderView, 'UnstructuredGridRepresentation')
552     Display.Representation = 'Surface'
553     renderView.ResetCamera(False, 0.9)
554     renderView.Update()
555
556     calculator3 = Calculator(registrationName='Calculator1', Input=readCase)
```

```
557
558     # Properties modified on calculator1
559     calculator3.Set(
560         ResultArrayName='Velocity_vec',
561         Function='iHat*SV_U+jHat*SV_V+kHat*SV_W',
562     )
563     # show data in view
564
565     calculator3Display = Show(calculator3, renderView,
        ↪  'UnstructuredGridRepresentation')
566
567     # trace defaults for the display properties.
568     calculator3Display.Representation = 'Surface'
569
570     # hide data in view
571     Hide(readCase, renderView)
572
573     # update the view to ensure updated data information
574     renderView.Update()
575
576     # create a new 'Stream Tracer'
577     streamTracer = StreamTracer(registrationName='StreamTracer',
        ↪  Input=calculator3, SeedType='Line')
578
579     streamTracer.SeedType.Point1 = startPoint
580     streamTracer.SeedType.Point2 = endpoint
581     streamTracer.SeedType.Resolution = number_of_lines
582
583     # show data in view
584     streamTracerDisplay = Show(streamTracer, renderView,
        ↪  'GeometryRepresentation')
585
586     # trace defaults for the display properties.
587     streamTracerDisplay.Representation = 'Surface'
588
589     # hide data in view
590     Hide(calculator3, renderView)
591
592     # update the view to ensure updated data information
593     renderView.Update()
594
595     # Properties modified on streamTracer1
596     streamTracer.IntegrationDirection = 'FORWARD'
597
598     # update the view to ensure updated data information
599     renderView.Update()
600
601     # set scalar coloring
602     ColorBy(streamTracerDisplay, ('POINTS', 'Velocity_vec', 'Magnitude'))
603
604     # rescale color and/or opacity maps used to include current data range
```

```
    streamTracerDisplay.RescaleTransferFunctionToDataRange(True, False)

    # show color bar/color legend
    streamTracerDisplay.SetScalarBarVisibility(renderView, True)

    # get color transfer function/color map for 'Velocity_vec'
    velocity_vecLUT = GetColorTransferFunction('Velocity_vec')

    # get opacity transfer function/opacity map for 'Velocity_vec'
    velocity_vecPWF = GetOpacityTransferFunction('Velocity_vec')

    # get 2D transfer function for 'Velocity_vec'
    velocity_vecTF2D = GetTransferFunction2D('Velocity_vec')

    # set active source
    SetActiveSource(readCase)

    # toggle interactive widget visibility (only when running from the GUI)
    HideInteractiveWidgets(proxy=streamTracer.SeedType)

    # get display properties
    first_simcash5Display = GetRepresentation(readCase, view=renderView)

    # create a new 'Threshold'
    threshold1 = Threshold(registrationName='Threshold1', Input=readCase)

    # Properties modified on threshold1
    threshold1.Set(
        Scalars=['CELLS', 'SV_WALL_DIST'],
        UpperThreshold=WallNormalCoordinate,
    )

    # show data in view
    thresholdDisplay = Show(threshold1, renderView,
    ↪  'UnstructuredGridRepresentation')

    # trace defaults for the display properties.
    thresholdDisplay.Representation = 'Surface'

    # hide data in view
    Hide(readCase, renderView)

    renderView.CameraPosition = cameraPosition  # Where the camera is
    renderView.CameraFocalPoint = cameraOrientation  # What it's looking at
    renderView.CameraViewUp = [0, 1, 0]  # Which direction is "up"
    renderView.CameraParallelScale = 1.  # Zoom level
    renderView.ViewSize = list((3000, 2000))
    #
    if interact:
        Interact()
```

```
655     if save:
656         ExportView(directory + '\\' + 'coefficient_of_pressure.pdf',
            ↪  view=renderView)
657 
658 def PlotCpT_xy_yz(readCase, U_infty, lims, xy_slice_position,
    ↪  yz_slice_position,cameraPosition, cameraOrientation, interact, save,
    ↪  directory, imageName):
659     source = get_data_source(readCase)
660     renderView = CreateView('RenderView')
661     Display = Show(source, renderView, 'UnstructuredGridRepresentation')
662     Display.Representation = 'Surface'
663     renderView.ResetCamera(False, 0.9)
664     renderView.Update()
665 
666     CpTCalculator = Calculator(registrationName='CpT', Input=readCase)
667 
668     # Properties modified on calculator1
669     UU = U_infty * U_infty
670     U_sqr = str((U_infty * U_infty))
671     CpTCalculator.Function = '(SV_P + 0.5 * SV_DENSITY * (SV_U^2 + SV_V^2 +
        ↪  SV_W^2)) / (0.5 * SV_DENSITY * ' + U_sqr + ')'
672     CpTCalculator.ResultArrayName = 'CpT'
673 
674     CpTSource = get_data_source(CpTCalculator)
675 
676     # show data in view
677     CpTDisplay = Show(CpTSource, renderView, 'UnstructuredGridRepresentation')
678 
679     # trace defaults for the display properties.
680     CpTDisplay.Representation = 'Surface'
681     Hide(source, renderView)
682     slice_xy = None
683     slice_yz = None
684     if len(xy_slice_position) != 0:
685         slice_xy = Slice(registrationName='CpT_' + 'Slice', Input=CpTSource)
686         slice_xy.SliceType.Origin = xy_slice_position
687         slice_xy.SliceType.Normal = [0, 1, 0]
688         # get active view
689         renderView = GetActiveViewOrCreate('RenderView')
690 
691         # show data in view
692         sliceDisplay = Show(slice_xy, renderView, 'GeometryRepresentation')
693 
694         # trace defaults for the display properties.
695         sliceDisplay.Representation = 'Surface'
696         Hide(CpTSource, renderView)
697 
698         ColorBy(sliceDisplay, (data_type(), 'CpT'))
699 
700         CpTLUT = GetColorTransferFunction('CpT')
701         CpTPWF = GetOpacityTransferFunction('CpT')
```

```
        CpTTF2D = GetTransferFunction2D('CpT')

        CpTLUT.RescaleTransferFunction(lims[0], lims[1])
        CpTPWF.RescaleTransferFunction(lims[0], lims[1])
        CpTTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
        ApplyCpTColormap(CpTLUT, lims)
        # show color bar/color legend
        sliceDisplay.SetScalarBarVisibility(renderView, True)

        # update the view to ensure updated data information
        renderView.Update()

    if len(yz_slice_position) != 0:
        slice_yz = Slice(registrationName='CpT_' + 'Slice', Input=CpTSource)
        slice_yz.SliceType.Origin = yz_slice_position
        slice_yz.SliceType.Normal = [1, 0, 0]
        # get active view
        renderView = GetActiveViewOrCreate('RenderView')

        # show data in view
        sliceDisplay = Show(slice_yz, renderView, 'GeometryRepresentation')

        # trace defaults for the display properties.
        sliceDisplay.Representation = 'Surface'
        Hide(CpTSource, renderView)

        ColorBy(sliceDisplay, (data_type(), 'CpT'))

        CpTLUT = GetColorTransferFunction('CpT')
        CpTPWF = GetOpacityTransferFunction('CpT')
        CpTTF2D = GetTransferFunction2D('CpT')

        CpTLUT.RescaleTransferFunction(lims[0], lims[1])
        CpTPWF.RescaleTransferFunction(lims[0], lims[1])
        CpTTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
        ApplyCpTColormap(CpTLUT, lims)
        # show color bar/color legend
        sliceDisplay.SetScalarBarVisibility(renderView, True)

        # update the view to ensure updated data information
        renderView.Update()

    #
    CpTDisplay.SetScalarBarVisibility(renderView, True)
    renderView.CameraPosition = cameraPosition  # Where the camera is
    renderView.CameraFocalPoint = cameraOrientation  # What it's looking at
    renderView.CameraViewUp = [0, 0, 1]  # Which direction is "up"
    renderView.CameraParallelScale = 1.0  # Zoom level
    renderView.ViewSize = list((3000, 2000))
    #
    if interact:
```

```
753         Interact()
754 
755     if save:
756         ExportView(directory + '\\' + 'slices_CpT'+ '\\'  + imageName +'_'
        ↪  +'slices.pdf', view=renderView)
757     objs_to_delete = [CpTDisplay, Display, CpTSource, CpTCalculator, source]
758 
759     if slice_xy is not None:
760         objs_to_delete.append(slice_xy)
761 
762     if slice_yz is not None:
763         objs_to_delete.append(slice_yz)
764 
765     cleanup_view(renderView, *objs_to_delete)
766 
767 def VisualizeIsoQ(readCase, lims, IsoSur, variable, cameraPosition,
    ↪  cameraOrientation, interact, save, directory):
768 
769     renderView = CreateView('RenderView')
770 
771     # show data in view
772     Display = Show(readCase, renderView, 'UnstructuredGridRepresentation')
773     Display.Representation = 'Surface'
774 
775     # reset view to fit data
776     renderView.ResetCamera(False, 0.9)
777     renderView.Update()
778 
779     # cell-centered data
780     ColorBy(Display, ('CELLS', 'SV_U'))
781     varColorFunction = GetColorTransferFunction(variable)
782     varOpacityFunction = GetOpacityTransferFunction(variable)
783     var2DFunction = GetTransferFunction2D(variable)
784 
785     varColorFunction.RescaleTransferFunction(lims[0], lims[1])
786     varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
787     var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
788 
789     Display.SetScalarBarVisibility(renderView, True)
790 
791     calculator1 = Calculator(registrationName='Calculator1', Input=readCase)
792 
793     # Properties modified on calculator1
794     print('initializing velocity vector')
795     calculator1.Set(
796         ResultArrayName='U_vec',
797         Function='SV_U * iHat + SV_V * jHat + SV_W * kHat',
798     )
799 
800     # get active view
801     renderView1 = GetActiveViewOrCreate('RenderView')
```

```
802
803     calculator1Display = Show(calculator1, renderView1,
        ↪  'UnstructuredGridRepresentation')
804     calculator1Display.Representation = 'Surface'
805     Hide(readCase, renderView1)
806     renderView1.Update()
807     ColorBy(calculator1Display, ('CELLS', 'U_vec', 'Magnitude'))
808     calculator1Display.RescaleTransferFunctionToDataRange(True, False)
809     calculator1Display.SetScalarBarVisibility(renderView1, True)
810
811     print('calculating Q')
812     gradient1 = Gradient(registrationName='Gradient1', Input=calculator1)
813
814     # Properties modified on gradient1
815     gradient1.Set(
816         ScalarArray=['CELLS', 'U_vec'],
817         ComputeVorticity=1,
818         ComputeQCriterion=1,
819     )
820
821     # show data in view
822     gradient1Display = Show(gradient1, renderView1,
        ↪  'UnstructuredGridRepresentation')
823
824     # trace defaults for the display properties.
825     gradient1Display.Representation = 'Surface'
826
827     # hide data in view
828     Hide(calculator1, renderView1)
829
830     # show color bar/color legend
831     gradient1Display.SetScalarBarVisibility(renderView1, True)
832
833     # update the view to ensure updated data information
834     renderView1.Update()
835
836     # create a new 'Contour'
837     print('setting up contour plot')
838     contour1 = Contour(registrationName='Contour1', Input=gradient1)
839
840     # Properties modified on contour1
841     contour1.Set(
842         ContourBy=['POINTS', 'Q Criterion'],
843         Isosurfaces=[1000.0],
844     )
845
846     # show data in view
847     contour1Display = Show(contour1, renderView1, 'GeometryRepresentation')
848     contour1Display.Representation = 'Surface'
849     Hide(gradient1, renderView1)
850     contour1Display.SetScalarBarVisibility(renderView1, True)
```

```
851
852     # update the view to ensure updated data information
853     renderView1.Update()
854
855     # set scalar coloring
856     ColorBy(contour1Display, ('CELLS', 'U_vec', 'Magnitude'))
857
858     # rescale color and/or opacity maps used to include current data range
859     contour1Display.RescaleTransferFunctionToDataRange(True, False)
860
861     # show color bar/color legend
862     contour1Display.SetScalarBarVisibility(renderView1, True)
863     qCriterionLUT = GetColorTransferFunction('QCriterion')
864     SV_ULUT = GetColorTransferFunction('SV_U')
865
866     HideScalarBarIfNotNeeded(qCriterionLUT, renderView1)
867     HideScalarBarIfNotNeeded(SV_ULUT, renderView1)
868
869     ULUT = GetColorTransferFunction('U_vec')
870     UPWF = GetOpacityTransferFunction('U_vec')
871     UTF2D = GetTransferFunction2D('U_vec')
872
873     ULUT.RescaleTransferFunction(lims[0], lims[1])
874     UPWF.RescaleTransferFunction(lims[0], lims[1])
875     UTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
876
877     renderView.CameraPosition = cameraPosition  # Where the camera is
878     renderView.CameraFocalPoint = cameraOrientation  # What it's looking at
879     renderView.CameraViewUp = [0, 0, 1]  # Which direction is "up"
880     renderView.CameraParallelScale = 1.0  # Zoom level
881     renderView.ViewSize = list((3000, 2000))
882
883     if interact:
884         Interact()
885
886     if save:
887         ExportView(directory + '\\' + 'IsoQ' + 'streamw_vel' + '.pdf',
        ↪  view=renderView)
888     cleanup_view(renderView, contour1Display, Display, contour1)
889
890 def sliceMovie(readCase, variable, lims, x_start, x_end, cameraPosition,
    ↪  cameraOrientation, interact, save, directory, filename):
891     source = get_data_source(readCase)
892     renderView = CreateView('RenderView')
893
894     # show data in view
895     Display = Show(source, renderView, 'UnstructuredGridRepresentation')
896     Display.Representation = 'Surface'
897
898     # reset view to fit data
899     renderView.ResetCamera(False, 0.9)
```

```
900     renderView.Update()
901
902     # cell-centered data
903     ColorBy(Display, (data_type(), variable))
904     varColorFunction = GetColorTransferFunction(variable)
905     varOpacityFunction = GetOpacityTransferFunction(variable)
906     var2DFunction = GetTransferFunction2D(variable)
907
908     varColorFunction.RescaleTransferFunction(lims[0], lims[1])
909     varOpacityFunction.RescaleTransferFunction(lims[0], lims[1])
910     var2DFunction.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
911
912     #Display.SetScalarBarVisibility(renderView, True)
913
914     slice1 = Slice(registrationName='Slice1', Input=source)
915
916     # get active view
917     renderView = GetActiveViewOrCreate('RenderView')
918
919     # show data in view
920     slice1Display = Show(slice1, renderView, 'GeometryRepresentation')
921
922     # trace defaults for the display properties.
923     slice1Display.Representation = 'Surface'
924
925     # hide data in view
926     Hide(source, renderView)
927
928     # show color bar/color legend
929     #slice1Display.SetScalarBarVisibility(renderView1, True)
930
931     # update the view to ensure updated data information
932     renderView.Update()
933
934     # get color transfer function/color map for 'SV_U'
935     sV_ULUT = GetColorTransferFunction(variable)
936
937     # get opacity transfer function/opacity map for 'SV_U'
938     sV_UPWF = GetOpacityTransferFunction(variable)
939
940     # get 2D transfer function for 'SV_U'
941     sV_UTF2D = GetTransferFunction2D(variable)
942
943     sV_ULUT.RescaleTransferFunction(lims[0], lims[1])
944
945     # Rescale transfer function
946     sV_UPWF.RescaleTransferFunction(lims[0], lims[1])
947
948     # Rescale 2D transfer function
949     sV_UTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
950     Display.SetScalarBarVisibility(renderView, False)
```

```
951
952     renderView.Set(
953         CameraPosition=cameraPosition,
954         CameraFocalPoint=cameraOrientation,
955         CameraViewUp=[0, 0, 1],
956         CameraParallelScale = 3.2008492319250466,
957     )
958     # get animation track
959     sliceTrack  = GetAnimationTrack('ContourValues', index=-1, proxy=slice1)
960
961     # create a new key frame
962     key1 = CompositeKeyFrame()
963     key1.KeyValues = [x_start]
964
965     key2 = CompositeKeyFrame()
966     key2.KeyTime = 1.0
967     key2.KeyValues = [x_end]
968
969     sliceTrack.KeyFrames = [key1, key2]
970     animationScene1 = GetAnimationScene()
971     animationScene1.NumberOfFrames = 400
972     animationScene1.Play()
973
974     if save:
975         # save animation
976         SaveAnimation(filename = directory + '\\' + filename + '.mp4',
        ↪  viewOrLayout=renderView, location=16,
977                       ImageResolution=[2160, 1440],
978                       FrameWindow=[0, 99],
979                       # MP4 Writer options
980                       FileName='')
981
982         print('Exported animation' + filename + '!')
983
984     if interact:
985         Interact()
986     cleanup_view(renderView, slice1Display, Display, slice1, source)
987
988 def sliceMovieCp(readCase, U_infty, lims, x_start, x_end, cameraPosition,
    ↪  cameraOrientation, interact, save, directory, filename):
989     source = get_data_source(readCase)
990     renderView = CreateView('RenderView')
991
992     Display = Show(source, renderView, 'UnstructuredGridRepresentation')
993     Display.Representation = 'Surface'
994
995     renderView.ResetCamera(False, 0.9)
996     renderView.Update()
997
998     # ---- Cp calculation ----
999     CpCalculator = Calculator(registrationName='Cp_movie', Input=readCase)
```

```
    U_sqr = str(U_infty * U_infty)
    CpCalculator.Function = 'SV_P / (0.5 * SV_DENSITY * ' + U_sqr + ')'
    CpCalculator.ResultArrayName = 'Cp'

    CpSource = get_data_source(CpCalculator)

    CpDisplay = Show(CpSource, renderView, 'UnstructuredGridRepresentation')
    CpDisplay.Representation = 'Surface'

    Hide(source, renderView)
    renderView.Update()

    ColorBy(CpDisplay, (data_type(), 'Cp'))

    CpLUT = GetColorTransferFunction('Cp')
    CpPWF = GetOpacityTransferFunction('Cp')
    CpTF2D = GetTransferFunction2D('Cp')

    CpLUT.RescaleTransferFunction(lims[0], lims[1])
    CpPWF.RescaleTransferFunction(lims[0], lims[1])
    CpTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

    ApplyCpColormap(CpLUT, lims)

    # ---- Slice ----
    slice1 = Slice(registrationName='Cp_SliceMovie', Input=CpSource)

    renderView = GetActiveViewOrCreate('RenderView')

    slice1Display = Show(slice1, renderView, 'GeometryRepresentation')
    slice1Display.Representation = 'Surface'

    Hide(CpSource, renderView)

    ColorBy(slice1Display, (data_type(), 'Cp'))
    slice1Display.SetScalarBarVisibility(renderView, False)

    renderView.Update()

    # enforce limits again
    CpLUT.RescaleTransferFunction(lims[0], lims[1])
    CpPWF.RescaleTransferFunction(lims[0], lims[1])
    CpTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
    ApplyCpColormap(CpLUT, lims)
    renderView.Set(
        CameraPosition=cameraPosition,
        CameraFocalPoint=cameraOrientation,
        CameraViewUp=[0, 0, 1],
        CameraParallelScale=3.2008492319250466,
    )
```

```
1051
1052        # ---- Animation ----
1053        sliceTrack = GetAnimationTrack('ContourValues', index=-1, proxy=slice1)
1054
1055        key1 = CompositeKeyFrame()
1056        key1.KeyValues = [x_start]
1057
1058        key2 = CompositeKeyFrame()
1059        key2.KeyTime = 1.0
1060        key2.KeyValues = [x_end]
1061
1062        sliceTrack.KeyFrames = [key1, key2]
1063
1064        animationScene1 = GetAnimationScene()
1065        animationScene1.NumberOfFrames = 400
1066        animationScene1.Play()
1067
1068        if save:
1069            SaveAnimation(
1070                filename=directory + '\\' + filename + '.mp4',
1071                viewOrLayout=renderView,
1072                location=16,
1073                ImageResolution=[2160, 1440],
1074                FrameWindow=[0, 99],
1075                FileName=''
1076            )
1077
1078            print('Exported animation ' + filename + '!')
1079
1080        if interact:
1081            Interact()
1082        cleanup_view(renderView, slice1Display, CpDisplay, Display, slice1,
            ↪ CpSource, CpCalculator, source)
1083
1084    def sliceMovieCpT(readCase, U_infty, lims, x_start, x_end, cameraPosition,
        ↪ cameraOrientation, interact, save, directory, filename):
1085        source = get_data_source(readCase)
1086        renderView = CreateView('RenderView')
1087
1088        # show data in view
1089        Display = Show(source, renderView, 'UnstructuredGridRepresentation')
1090        Display.Representation = 'Surface'
1091
1092        # reset view to fit data
1093        renderView.ResetCamera(False, 0.9)
1094        renderView.Update()
1095
1096        # create CpT field
1097        CpTCalculator = Calculator(registrationName='CpT_movie', Input=readCase)
1098
1099        U_sqr = str(U_infty * U_infty)
```

```
    CpTCalculator.Function = '(SV_P + 0.5 * SV_DENSITY * (SV_U^2 + SV_V^2 +
    ↪  SV_W^2)) / (0.5 * SV_DENSITY * ' + U_sqr + ')'
    CpTCalculator.ResultArrayName = 'CpT'

    CpTSource = get_data_source(CpTCalculator)

    # show CpT in view
    CpTDisplay = Show(CpTSource, renderView, 'UnstructuredGridRepresentation')
    CpTDisplay.Representation = 'Surface'

    # hide original case
    Hide(source, renderView)
    renderView.Update()

    # color by CpT
    ColorBy(CpTDisplay, (data_type(), 'CpT'))

    CpTLUT = GetColorTransferFunction('CpT')
    CpTPWF = GetOpacityTransferFunction('CpT')
    CpTTF2D = GetTransferFunction2D('CpT')

    CpTLUT.RescaleTransferFunction(lims[0], lims[1])
    CpTPWF.RescaleTransferFunction(lims[0], lims[1])
    CpTTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
    ApplyCpTColormap(CpTLUT, lims)
    # create slice
    slice1 = Slice(registrationName='CpT_SliceMovie', Input=CpTSource)

    renderView = GetActiveViewOrCreate('RenderView')

    # show slice in view
    slice1Display = Show(slice1, renderView, 'GeometryRepresentation')
    slice1Display.Representation = 'Surface'

    # hide full CpT field
    Hide(CpTSource, renderView)

    # color slice by CpT
    ColorBy(slice1Display, (data_type(), 'CpT'))
    slice1Display.SetScalarBarVisibility(renderView, False)

    renderView.Update()

    # enforce CpT range on slice as well
    CpTLUT = GetColorTransferFunction('CpT')
    CpTPWF = GetOpacityTransferFunction('CpT')
    CpTTF2D = GetTransferFunction2D('CpT')

    CpTLUT.RescaleTransferFunction(lims[0], lims[1])
    CpTPWF.RescaleTransferFunction(lims[0], lims[1])
    CpTTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)
```

```
    ApplyCpTColormap(CpTLUT, lims)
    renderView.Set(
        CameraPosition=cameraPosition,
        CameraFocalPoint=cameraOrientation,
        CameraViewUp=[0, 0, 1],
        CameraParallelScale=3.2008492319250466,
    )

    # animate slice position in x-direction
    sliceTrack = GetAnimationTrack('ContourValues', index=-1, proxy=slice1)

    key1 = CompositeKeyFrame()
    key1.KeyValues = [x_start]

    key2 = CompositeKeyFrame()
    key2.KeyTime = 1.0
    key2.KeyValues = [x_end]

    sliceTrack.KeyFrames = [key1, key2]

    animationScene1 = GetAnimationScene()
    animationScene1.NumberOfFrames = 400
    animationScene1.Play()

    if save:
        SaveAnimation(
            filename=directory + '\\' + filename + '.mp4',
            viewOrLayout=renderView,
            location=16,
            ImageResolution=[2160, 1440],
            FrameWindow=[0, 99],
            FileName=''
        )

        print('Exported animation ' + filename + '!')

    if interact:
        Interact()
    cleanup_view(renderView, slice1Display, CpTDisplay, Display, slice1,
    ↪  CpTSource, CpTCalculator, source)


def sliceMovieSV_Mag(readCase, lims, y_start, y_end, cameraPosition,
↪  cameraOrientation, interact, save, directory, filename,
↪  num_frames=50,image_resolution=[2160, 1440]):
    source = get_data_source(readCase)
    renderView = CreateView('RenderView')

    # show data in view
    Display = Show(source, renderView, 'UnstructuredGridRepresentation')
```

```
    Display.Representation = 'Surface'

    # reset view
    renderView.ResetCamera(False, 0.9)
    renderView.Update()

    # ---- SV_Mag calculation ----
    VelMagCalculator = Calculator(registrationName='SV_Mag_movie',
    ↪  Input=readCase)
    VelMagCalculator.Function = 'sqrt(SV_U^2 + SV_V^2 + SV_W^2)'
    VelMagCalculator.ResultArrayName = 'SV_Mag'

    VelMagSource = get_data_source(VelMagCalculator)

    VelMagDisplay = Show(VelMagSource, renderView,
    ↪  'UnstructuredGridRepresentation')
    VelMagDisplay.Representation = 'Surface'

    Hide(source, renderView)
    renderView.Update()

    ColorBy(VelMagDisplay, (data_type(), 'SV_Mag'))

    VelMagLUT = GetColorTransferFunction('SV_Mag')
    VelMagPWF = GetOpacityTransferFunction('SV_Mag')
    VelMagTF2D = GetTransferFunction2D('SV_Mag')

    VelMagLUT.RescaleTransferFunction(lims[0], lims[1])
    VelMagPWF.RescaleTransferFunction(lims[0], lims[1])
    VelMagTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

    # ---- Slice ----
    slice1 = Slice(registrationName='SV_Mag_SliceMovie', Input=VelMagSource)
    slice1.SliceType.Origin = [0, y_start, 0]
    slice1.SliceType.Normal = [0, 1, 0]

    renderView = GetActiveViewOrCreate('RenderView')

    slice1Display = Show(slice1, renderView, 'GeometryRepresentation')
    slice1Display.Representation = 'Surface'

    Hide(VelMagSource, renderView)

    ColorBy(slice1Display, (data_type(), 'SV_Mag'))
    slice1Display.SetScalarBarVisibility(renderView, False)

    renderView.Update()

    # enforce limits again on slice
    VelMagLUT = GetColorTransferFunction('SV_Mag')
    VelMagPWF = GetOpacityTransferFunction('SV_Mag')
```

```
    VelMagTF2D = GetTransferFunction2D('SV_Mag')

    VelMagLUT.RescaleTransferFunction(lims[0], lims[1])
    VelMagPWF.RescaleTransferFunction(lims[0], lims[1])
    VelMagTF2D.RescaleTransferFunction(lims[0], lims[1], 0.0, 1.0)

    renderView.Set(
        CameraPosition=cameraPosition,
        CameraFocalPoint=cameraOrientation,
        CameraViewUp=[0, 0, 1],
        CameraParallelScale=3.2008492319250466,
    )

    # ---- Animation in y-direction ----
    originYTrack = GetAnimationTrack('Origin', index=1, proxy=slice1.SliceType)

    key1 = CompositeKeyFrame()
    key1.KeyValues = [y_start]

    key2 = CompositeKeyFrame()
    key2.KeyTime = 1.0
    key2.KeyValues = [y_end]

    originYTrack.KeyFrames = [key1, key2]

    animationScene1 = GetAnimationScene()
    animationScene1.NumberOfFrames = num_frames
    animationScene1.Play()

    if save:
        SaveAnimation(
            filename=directory + '\\' + filename + '.mp4',
            viewOrLayout=renderView,
            location=16,
            ImageResolution=image_resolution,
            FrameWindow=[0, num_frames - 1],
            FileName=''
        )

        print('Exported animation ' + filename + '!')

    if interact:
        Interact()

    cleanup_view(renderView, slice1Display, VelMagDisplay, Display,
                 slice1, VelMagSource, VelMagCalculator, source)
```

5.2 Fluent TUI journals

5.2.1 main.jou

```
;;
↪  ================================================================================
;;  main.jou
;; --------------------------------------------------------------------------------
;;  Top-level flow-control journal for the EPFL Racing Team Fluent automation.
;;
;;  Launch:
;;     fluent 3ddp -meshing -i main.jou
;;  Or from inside Fluent Meshing:
;;     /file/read-journal main.jou
;;
;;  This file uses ONLY Scheme expressions (no bare TUI lines).  Driving
;;  every TUI call through (ti-menu-load-string ...) avoids two parser
;;  quirks: menu-listing spam on blank/comment lines, and silent prompt
;;  consumption when one TUI command has hidden sub-prompts.
;;
;;  PROJECT LAYOUT (CAD lives in the project root, not a subfolder):
;;     fluent_automation/
;;       main.jou
;;       <whatever>.scdoc        <- CAD here, exactly one .scdoc allowed
;;       scheme/
;;         globals.scm
;;         cad_import.scm
;;         mesh.scm
;;       modules/
;;         01_cad_import.jou
;;         02_mesh.jou
;;       output/
;;         (case/data files, post-processing artifacts)
;;
;;  Why this layout: the CAD import in 2024 R2 silently changes Cortex's
;;  cwd to the directory containing the CAD file.  Keeping the CAD in the
;;  project root means cwd stays at the project root throughout, so plain
;;  relative paths in (load ...) and /file/read-journal keep working.
;;
↪  ================================================================================


;; ===== 0. Bootstrap
↪  ============================================================
(load "scheme/globals.scm")

;; Make sure the output directory exists.  Used by later modules for case/data
;; saves and post-processing artifacts (not for our own logging - we rely on
;; Fluent's transcript .trn for that).
(system (string-append "mkdir \"" (%rpgetvar 'user/output-dir) "\""))

```

```
45
46  ;; ===== Helper : run a module journal and abort on failure
    ↪  ====================
47  (define (run-module path label)
48    (format #t "~%====================  ~a  ====================~%" label)
49    (ti-menu-load-string (string-append "/file/read-journal " path))
50    (if (not (ert-status-ok?))
51        (begin
52          (format #t "~%main : aborting - module '~a' failed~%" label)
53          (exit))))
54
55
56  ;; ===== 1. CAD IMPORT (Meshing mode)
    ↪  ========================================
57  (run-module "modules/01_cad_import.jou" "cad_import")
58
59
60  ;; ===== 2. MESH GENERATION (Meshing mode)
    ↪  ====================================
61  (run-module "modules/02_mesh.jou" "mesh")
62
63
64  ;; ===== 3. SWITCH TO SOLUTION MODE
    ↪  ==========================================
65  ;; (ti-menu-load-string "/switch-to-solution-mode yes")
66  ;; (load "scheme/globals.scm")    ;; Scheme interpreter resets, RP vars
    ↪  persist
67
68
69  ;; ===== 4. PHYSICS SETUP (Solution mode)
    ↪  ====================================
70  ;; (run-module "modules/03_physics.jou" "physics")
```

5.3 Scheme codes

5.3.1 globals.scm

```
1   ;;
    ↪  ============================================================================
2   ;;  globals.scm
3   ;; ----------------------------------------------------------------------------
4   ;;  Central parameter store + helper utilities for the EPFL Racing Team
5   ;;  Fluent automation pipeline.
6   ;;
7   ;;  Loaded TWICE during a run:
8   ;;     1. by main.jou at the very start (Meshing mode)
9   ;;     2. by main.jou again immediately after the switch to Solution mode
10  ;;
11  ;;  All user-tunable parameters are exposed as RP variables (rp-var-define)
12  ;;  so they:
```

```
;;     - persist across the Meshing -> Solution switch
;;     - are saved with the case file
;;     - are accessible from UDFs via RP_Get_*
;;
;;  An optional override file (config/run_config.scm) is loaded LAST so the
;;  user can change any parameter without editing globals.scm directly.
;;
↪  ============================================================================


;; ----------------------------------------------------------------------------
;; 1. RP-VARIABLE HELPER
;; ----------------------------------------------------------------------------
;; make-new-rpvar is shipped with Fluent, but we wrap it to:
;;   - guarantee the user/ namespace prefix
;;   - silently no-op if the variable was already restored from the case file
;;   - return the *current* value after definition

(define (ert-rpvar-define name default type)
  "Create an RP variable in the user/ namespace if it does not yet exist.
   NAME    : symbol  (e.g. 'inlet-velocity)
   DEFAULT : initial value
   TYPE    : 'real | 'integer | 'boolean | 'string | 'list"
  (let ((full-name (string->symbol
                     (string-append "user/"
                                    (symbol->string name)))))
    (if (not (rp-var-object full-name))
        (rp-var-define full-name default type #f))
    (%rpgetvar full-name)))


;; ----------------------------------------------------------------------------
;; 2. PATHS  (every module reads these instead of hard-coding paths)
;; ----------------------------------------------------------------------------
;; input-dir is "." (the project root) rather than a subfolder because Fluent
;; Meshing's CAD import silently changes Cortex's cwd to the directory
;; containing the .scdoc.  If the CAD lived in a subfolder, every relative
;; path resolved AFTER the import would break.  Keeping the CAD in the
;; project root sidesteps this entirely - cwd stays put throughout the run.
(ert-rpvar-define 'project-root  "."                        'string)
(ert-rpvar-define 'input-dir     "."                        'string)
(ert-rpvar-define 'output-dir    "output"                   'string)

;; CAD file: empty by default = auto-discover the single .scdoc in input-dir
;; (i.e. the project root).  Set explicitly in run_config.scm to bypass
;; auto-discovery if you have several .scdoc files lying around.
(ert-rpvar-define 'cad-file      ""                         'string)


;; ----------------------------------------------------------------------------
;; 3. REQUIRED NAME SELECTIONS  (sanity-checked by cad_import)
```

```
;; ------------------------------------------------------------------------
;; The names below MUST exist in the .scdoc, otherwise the run aborts.
;; They mirror the ones produced by SpaceClaim_preparation_BASELINE3.py.
;;
;; Anything that is NOT in this list and NOT an enclosure wall is treated
;; as "car" (no-slip wall) and stored in user/car-zones at import time.

(ert-rpvar-define 'enclosure-zones
                  '("inlet" "outlet" "symmetry"
                    "wall-side" "wall-top" "wall-ground")
                  'list)

(ert-rpvar-define 'wheel-zones
                  '("wheel-front" "wheel-rear")
                  'list)

(ert-rpvar-define 'patch-zones
                  '("patch-front" "patch-rear")
                  'list)

;; Filled by cad_import.scm at runtime - everything else
(ert-rpvar-define 'car-zones '() 'list)

;; Name of the CAD object after import (e.g. "Enveloppe").  Filled by
;; cad_import.scm and reused by the mesh module.
(ert-rpvar-define 'cad-object-name "" 'string)

;; Name of the FLUID REGION created by WTM's Update Regions task.  WTM
;; derives this from the SpaceClaim body name; the saved-workflow recording
;; uses 'enveloppe-enveloppe1' (CAD body 'Enveloppe').  Auto-detected at
;; runtime by cad_import.scm: when WTM Import Geometry exposes the body
;; as a pseudo-label, we capture that name here and use it for boundary-
;; layer scoping.  The default below is just a fallback.
(ert-rpvar-define 'cad-region-name "enveloppe-enveloppe1" 'string)


;; ------------------------------------------------------------------------
;; 4. CAD IMPORT OPTIONS
;; ------------------------------------------------------------------------
;; Length unit of the .scdoc (SpaceClaim script works in mm).
(ert-rpvar-define 'cad-length-unit "mm"               'string)
;; Tessellation mode: "cad-faceting" or "cfd-surface-mesh".
;; cad-faceting is the safer default for the F1-style geometry.
(ert-rpvar-define 'cad-tessellation "cad-faceting"    'string)
(ert-rpvar-define 'cad-feature-angle 40.0             'real)


;; ------------------------------------------------------------------------
;; 5. PHYSICS (used by later modules - declared here so globals owns them all)
;; ------------------------------------------------------------------------
(ert-rpvar-define 'inlet-velocity   20.0  'real)   ;; [m/s]
```

```
(ert-rpvar-define 'op-pressure   101325.0 'real)   ;; [Pa]

;; Wheel rotation axes - extracted from SpaceClaim, see Specification §6.3.
;; Stored in metres because Fluent operates in SI internally.
(ert-rpvar-define 'wheel-front-axis '(-1.530  0.7104  0.2032) 'list) ;; [m]
(ert-rpvar-define 'wheel-rear-axis  '( 0.000  0.7104  0.2032) 'list) ;; [m]


;; ============================================================================
;; 6. MESHING PARAMETERS
;; ============================================================================
;; All meshing parameters are exposed as RP variables so they:
;;   - persist across the Meshing -> Solution switch
;;   - are saved with the case file
;;   - can be overridden in config/run_config.scm without touching this file
;;
;; Tables are stored as Scheme association lists (lists of lists of values).
;; RP-var typing for nested lists is just 'list - Fluent stores the
;; s-expression verbatim.
;;
;; Coordinate convention (matches Specification §6.2):
;;   Static BOIs   : absolute coordinates [mm].
;;   Wheel BOIs    : OFFSET from the bbox of the named selection [m].
;;   Per-car BOIs  : OFFSET applied to every label in user/car-zones.
;;   Local sizing  : surface mesh size controls (curvature + face-size).


;; ---- 6.1  Static BOIs (absolute coordinates, mm) -------------------------
;; (name xmin xmax ymin ymax zmin zmax sizing-mm)
;; Names match the working GUI workflow's RefinementRegionsName values
;; (local-refinement-* prefix).  Far-field extends 15m downstream of the
;; car (Xmax=15000) to fully resolve the wake.
;;
;; rwing / diffuser : final absolute boxes (no longer expressed as
;; base-bbox * ratio).  Coordinates rounded from the manual baseline boxes
;; for clarity - to update them, measure the desired final box in the
;; Fluent GUI centroid tool and paste the values here directly.
(ert-rpvar-define 'mesh-static-bois
                  '(("local-refinement-near-field"    -3000.0  3000.0     0.0
                    ↪  1500.0     0.0 2000.0    32.0)
                    ("local-refinement-middle-field"  -4000.0  8500.0     0.0
                    ↪  2000.0     0.0 3000.0    64.0)
                    ("local-refinement-far-field"     -5000.0 15000.0     0.0
                    ↪  2500.0     0.0 4000.0   128.0)
                    ("local-refinement-rwing"          -520.0  1780.0     0.0
                    ↪  1080.0   660.0 1500.0    16.0)
                    ("local-refinement-diffuser"      -1450.0  1000.0  -240.0
                    ↪  540.0   -30.0  275.0    16.0))
                  'list)
```

```
158
159 ;; ---- 6.2  Label-bbox-relative BOIs (wheels) ------------------------------
160 ;; (name target-label rx-min rx-max ry-min ry-max rz-min rz-max sizing-mm)
161 ;; Ratios sent verbatim to Fluent; Fluent reads the label's tessellation bbox
162 ;; at meshing time and expands by ratio*label_size on each side, exactly
163 ;; matching the manual baseline workflow.  These follow the wheel geometry
164 ;; wherever it sits in the CAD.  Per the tech spec the wheel-front /
165 ;; wheel-rear name selections are guaranteed to exist, so this label-relative
166 ;; form is safe.  The manual baseline only sets XminRatio (0.1) and XmaxRatio
167 ;; (1.0) explicitly; the other ratios use Fluent's default (0.1).
168 ;; Names match the working GUI workflow (local-refinement-fw/-rw).
169 (ert-rpvar-define 'mesh-wheel-bois
170                   '(("local-refinement-fw" "wheel-front" 0.1 1.0  0.1 0.1  0.1
                    ↪  0.1  16.0)
171                     ("local-refinement-rw" "wheel-rear"  0.1 1.0  0.1 0.1  0.1
                      ↪  0.1  16.0))
172                   'list)
173
174 ;; ---- 6.4  Local surface-mesh sizing (curvature controls) -----------------
175 ;; (group-name growth-rate min-size max-size normal-angle scope-edges?)
176 ;; min/max in mm, normal-angle in deg.  Values match the working GUI workflow
177 ;; (screenshots verified):
178 ;;   "chassis"            -> chassis label only, faces+edges, normal-angle 12
179 ;;   "aero"               -> aero kit + wishbones, faces only, normal-angle 18
180 ;;   "wheel"              -> wheels + patches, faces only, normal-angle 18
181 ;; Note: the technical spec lists chassis max=8mm; the screenshot shows 32mm.
182 ;; Per user confirmation, the screenshot is authoritative (spec typo).
183 (ert-rpvar-define 'mesh-local-sizing
184                   '(("chassis" 1.2  1.0  32.0  12.0 #t)
185                     ("aero"    1.2  0.5   8.0  18.0 #f)
186                     ("wheel"   1.2  0.5  32.0  18.0 #f))
187                   'list)
188
189 ;; ---- 6.5  Surface mesh improve -------------------------------------------
190 (ert-rpvar-define 'mesh-surface-quality-limit  0.6  'real)
191
192 ;; ---- 6.6  Inflation layers (scoped-prisms) ------------------------------
193 ;; (control-name scope-key offset-method n-layers transition-ratio
    ↪  first-height-mm)
194 ;; scope-key is a SYMBOL; mesh.scm resolves
195 ;;   'car+wheels -> user/car-zones + user/wheel-zones (NOT patches - they're
196 ;;                  conceptually the ground surface and shouldn't get a BL)
197 ;;   'ground     -> ("wall-ground")
198 ;; offset-method is a STRING: "last-ratio" or "smooth-transition".
199 ;; For "smooth-transition" the FirstHeight is computed by WTM from
200 ;; transition-ratio + n-layers; the explicit first-height we pass is ignored.
201 ;;
202 ;; Per the working GUI workflow (JSON dump):
203 ;;   last-ratio_car        : 13 labels (chassis + aero + wishbones + wheels +
    ↪  patches),
204 ;;                           4 layers, FirstHeight=1mm, last-ratio offset
```

```
;;   smooth-transition_ground : wall-ground only, 4 layers, smooth-transition offset
;; We deliberately exclude patch-front and patch-rear from last-ratio_car
;; (12 labels instead of 13): patches are static contact prints on a moving
;; ground and shouldn't have an inflation layer growing off them.  This is
;; an intentional refinement over the recorded GUI workflow.
;; Note this is also DIFFERENT from the technical spec which says ground=6
;; layers with last-ratio.  Per user instruction we follow the GUI workflow.
(ert-rpvar-define 'mesh-prism-controls
                  '(("last-ratio_car"           car+wheels  "last-ratio"
                    4 0.272 1.0)
                    ("smooth-transition_ground" ground      "smooth-transition"
                     4 0.272 1.0))
                  'list)

;; ---- 6.7  Poly-hexcore volume mesh ----------------------------------------
(ert-rpvar-define 'mesh-hex-peel-layers           1     'integer)
(ert-rpvar-define 'mesh-hex-min-cell-length-mm    1.0   'real)
(ert-rpvar-define 'mesh-hex-max-cell-length-mm  256.0   'real)
(ert-rpvar-define 'mesh-hex-buffer-layers         1     'integer)
(ert-rpvar-define 'mesh-poly-feature-angle-deg   30.0   'real)
(ert-rpvar-define 'mesh-hex-avoid-1to8-jump?     #f     'boolean)
(ert-rpvar-define 'mesh-hex-merge-bl-cells?      #t     'boolean)
(ert-rpvar-define 'mesh-prox-gap-factor          0.25   'real)
(ert-rpvar-define 'mesh-prox-max-aspect-ratio   25.0    'real)
(ert-rpvar-define 'mesh-adjacent-attach-angle   80.0    'real)
(ert-rpvar-define 'mesh-normal-smooth-relax    100.0    'real)

;; ---- 6.8  Volume mesh improve ---------------------------------------------
(ert-rpvar-define 'mesh-volume-quality-limit    0.05  'real)  ;; improve threshold
(ert-rpvar-define 'mesh-volume-quality-target   0.2   'real)  ;; spec: orth >= 0.2


;; ============================================================================
;; 7. STATUS TRACKING
;; ============================================================================
;; A module is a SUCCESS if its sanity check passes; otherwise FAILED.
;; main.jou inspects user/last-status after each module call and aborts
;; if it is not "OK".  We do not write a separate log file - Fluent's
;; transcript (.trn) already captures everything in more detail than we
;; could reproduce, and the console echo from ert-log-line below makes
;; module status easy to grep for in the trn.

(ert-rpvar-define 'last-status "OK"     'string)   ;; "OK" | "FAILED"
(ert-rpvar-define 'last-module "init"   'string)


(define (ert-log-line module status message)
  "Print a status line to the console and update last-status / last-module.
```

```
   MODULE  : string, e.g. \"cad_import\"
   STATUS  : string, \"OK\" or \"FAILED\"
   MESSAGE : string, free-form description

   We do not write to a separate log file: Fluent already produces a
   detailed transcript (.trn) that is more useful than anything we could
   write ourselves, and writing files from Scheme has proven fragile across
   the meshing<->solution boundary.  The status RP vars (queried via
   ert-status-ok?) are sufficient for main.jou flow control."
  (rpsetvar 'user/last-status status)
  (rpsetvar 'user/last-module module)
  (format #t "[~a] ~a : ~a~%" status module message))


(define (ert-log-info module message)
  (ert-log-line module "OK" message))

(define (ert-log-fail module message)
  (ert-log-line module "FAILED" message))


(define (ert-status-ok?)
  (string=? (%rpgetvar 'user/last-status) "OK"))


;; ---------------------------------------------------------------------------
;; 8. SMALL UTILITIES SHARED ACROSS MODULES
;; ---------------------------------------------------------------------------

(define (ert-list-contains? lst item)
  "True if STRING-EQUAL? ITEM appears in the list LST of strings."
  (cond ((null? lst) #f)
        ((string=? (car lst) item) #t)
        (else (ert-list-contains? (cdr lst) item))))


(define (ert-strip-zone-id name)
  "Fluent face-zone labels are 'name:id' or 'name.id' in some versions.
   Returns just 'name'.  Safe on already-stripped names."
  (let* ((s (if (symbol? name) (symbol->string name) name))
         (n (string-length s)))
    (let loop ((i 0))
      (cond ((>= i n) s)
            ((or (char=? (string-ref s i) #\:)
                 (char=? (string-ref s i) #\.))
             (substring s 0 i))
            (else (loop (+ i 1)))))))


(define (ert-zone-names-as-strings)
  "Wrapper around (inquire-zone-names) that returns plain strings,
```

```
   stripped of any trailing ':id'."
  (map (lambda (z) (ert-strip-zone-id z))
       (inquire-zone-names)))


;; ---------------------------------------------------------------------------
;; 9. OPTIONAL OVERRIDE FILE
;; ---------------------------------------------------------------------------
;; Loaded last so user values win over defaults declared above.
;; The override file is OPTIONAL - if it does not exist we silently continue.

(if (file-exists? "config/run_config.scm")
    (begin
      (load "config/run_config.scm")
      (format #t "globals : loaded user overrides from
      ↪  config/run_config.scm~%"))
    (format #t "globals : no run_config.scm found, using defaults~%"))


;; ---------------------------------------------------------------------------
;; 10. PIN PATHS TO ABSOLUTE FORM
;; ---------------------------------------------------------------------------
;; Fluent Meshing's CAD import silently changes the working directory of one
;; or more processes (Cortex / mesher), so a relative log path that worked
;; at startup fails after the import with 'sopenoutputfile: unable to open'.
;; We resolve every file path to absolute ONCE here, then forevermore the
;; logger and other helpers use those absolute strings.
;;
;; We use a temp file to capture the OS cwd because Fluent's Scheme does not
;; expose a portable getcwd().  This runs ONCE at startup.

(define (ert-current-directory)
  "Return the absolute path of the current working directory as a string.
   Uses the OS shell to print cwd into a temp file, then reads it back.
   On Windows: 'cd' alone prints cwd.
   On Linux:   'pwd' prints cwd ('cd' alone changes to HOME, doesn't print)."
  (let* ((tmp "__ert_pwd.tmp")
         (cmd (if (unix?) "pwd" "cd")))
    (system (string-append cmd " > " tmp))
    (let ((p (open-input-file tmp)))
      (let loop ((acc '()))
        (let ((c (read-char p)))
          (cond
            ((eof-object? c)
             (close-input-port p)
             ;; remove-file is portable in Fluent Scheme
             (remove-file tmp)
             ;; Trim trailing whitespace / newlines
             (let* ((s (list->string (reverse acc)))
                    (n (string-length s)))
               (let trim ((i n))
```

```
                  (cond
                    ((<= i 0) "")
                    ((let ((c (string-ref s (- i 1))))
                       (or (char=? c #\newline)
                           (char=? c #\return)
                           (char=? c #\space)
                           (char=? c #\tab)))
                     (trim (- i 1)))
                    (else (substring s 0 i))))))
              (else (loop (cons c acc)))))))))


(define (ert-slashify s)
  "Convert any '\\' in S to '/'.  Fluent's TUI accepts forward slashes
   on both Linux and Windows, so we normalise everything to '/'."
  (let* ((n (string-length s))
         (buf (make-string n)))
    (let loop ((i 0))
      (cond ((>= i n) buf)
            (else
             (let ((c (string-ref s i)))
               (string-set! buf i (if (char=? c #\\) #\/ c))
               (loop (+ i 1)))))))))


(define (ert-make-absolute path cwd)
  "If PATH is already absolute (starts with / or X:\\), return it.
   Otherwise prepend CWD with the right separator."
  (let ((n (string-length path)))
    (cond
      ((= n 0) cwd)
      ;; Linux absolute (/...)
      ((char=? (string-ref path 0) #\/) path)
      ;; Windows absolute (X:\... or X:/...)
      ((and (>= n 2)
            (char=? (string-ref path 1) #\:))
       path)
      ;; Otherwise relative - prepend cwd
      (else
       (string-append cwd "/" path)))))


(define (ert-read-temp-lines tmp)
  "Read a temp file produced by a shell listing command, return a list
   of non-empty trimmed STRINGS (one per line).  Removes the temp file."
  (let ((p (open-input-file tmp)))
    (let loop ((acc '()) (cur '()))
      (let ((c (read-char p)))
        (cond
          ;; End of file - close, delete, package up the last line if any
          ((eof-object? c)
```

```
            (close-input-port p)
            (remove-file tmp)
            (let ((final (if (null? cur)
                             acc
                             (cons (list->string (reverse cur)) acc))))
              (reverse
               ;; Filter out empty strings (blank lines)
               (let filter ((src final) (out '()))
                 (cond
                   ((null? src) out)
                   ((= (string-length (car src)) 0)
                    (filter (cdr src) out))
                   (else
                    (filter (cdr src) (cons (car src) out)))))))))
          ;; End of line - flush accumulated chars as one string
          ((or (char=? c #\newline) (char=? c #\return))
           (if (null? cur)
               (loop acc cur)
               (loop (cons (list->string (reverse cur)) acc) '())))
          ;; Regular char - accumulate
          (else
           (loop acc (cons c cur)))))))


(define (ert-list-scdoc-files dir)
  "List '*.scdoc' filenames in DIR (just basenames, no path).
   Uses the OS shell because Fluent Scheme has no directory primitive.
   Returns a list of strings; empty list if nothing matches."
  (let* ((tmp (string-append dir "/.__ert_scdoc_list.tmp"))
         (cmd (if (unix?)
                  ;; ls returns non-zero with a message on no-match;
                  ;; redirect stderr to /dev/null and let ls print just names.
                  (string-append "ls \"" dir "\"/*.scdoc 2>/dev/null > \"" tmp
                  ↪  "\"")
                  ;; dir /B prints bare filenames, one per line; redirect both
                  ;; streams so the 'File Not Found' message doesn't leak.
                  (string-append "dir /B \"" dir "\\*.scdoc\" > \"" tmp "\"
                  ↪  2>nul"))))
    (system cmd)
    (let ((entries (ert-read-temp-lines tmp)))
      (if (unix?)
          ;; Linux 'ls' returns full paths - strip the directory prefix
          ;; so we get just the filenames.  basename is not portable in
          ;; Fluent's Scheme so we strip manually.
          (map (lambda (full)
                 (let loop ((i (- (string-length full) 1)))
                   (cond
                     ((< i 0) full)
                     ((char=? (string-ref full i) #\/)
                      (substring full (+ i 1) (string-length full)))
                     (else (loop (- i 1))))))
```

```
                entries)
          ;; Windows 'dir /B' already returns bare filenames
          entries))))


(define (ert-resolve-cad-file cad-rpvar input-dir-abs cwd)
  "If user/cad-file (CAD-RPVAR) is empty, auto-discover the unique
   .scdoc in INPUT-DIR-ABS.  If non-empty, treat as an explicit user
   override; relative paths are resolved against CWD.
   Errors via (error ...) if zero or multiple .scdoc files are present."
  (cond
    ((> (string-length cad-rpvar) 0)
     (let ((abs (ert-make-absolute cad-rpvar cwd)))
       (format #t "globals : using user-specified CAD file = ~a~%" abs)
       abs))
    (else
     (let ((found (ert-list-scdoc-files input-dir-abs)))
       (cond
         ((null? found)
          (error
            (string-append
              "globals : no .scdoc found in " input-dir-abs
              " - place exactly one CAD file there, or set user/cad-file "
              "explicitly in run_config.scm")))
         ((not (null? (cdr found)))
          (error
            (string-append
              "globals : multiple .scdoc files found in " input-dir-abs
              " ("
              (apply string-append
                     (map (lambda (s) (string-append s " ")) found))
              ") - keep only one, or set user/cad-file explicitly "
              "in run_config.scm")))
         (else
          (let ((picked (string-append input-dir-abs "/" (car found))))
            (format #t "globals : auto-discovered CAD file = ~a~%" picked)
            picked)))))))


(let* ((cwd        (ert-slashify (ert-current-directory)))
       (input-abs  (ert-slashify (ert-make-absolute
                                   (%rpgetvar 'user/input-dir)  cwd)))
       (output-abs (ert-slashify (ert-make-absolute
                                   (%rpgetvar 'user/output-dir) cwd))))
  (format #t "globals : working directory = ~a~%" cwd)
  (rpsetvar 'user/project-root cwd)          ;; absolute project root
  (rpsetvar 'user/input-dir  input-abs)
  (rpsetvar 'user/output-dir output-abs)
  ;; CAD file: respect user override if given, otherwise auto-discover.
  (rpsetvar 'user/cad-file
            (ert-slashify
```

```
              (ert-resolve-cad-file (%rpgetvar 'user/cad-file)
                                    input-abs
                                    cwd))))


(format #t "globals : initialisation complete~%")
```

5.4 Matlab multi element set up code

```
% ============================================================
%  PARAMÈTRES MULTI-ELEMENT (à modifier ici)
% ============================================================
%
%  LOGIQUE DE PLACEMENT :
%  - Flaps 1 à n_flaps-1 : placés avec slot_gap et overlap explicites
%  - Dernier flap        : slot_gap et overlap calculés automatiquement
%                          pour que son TE upper atterrisse exactement
%                          sur TE_final_target
%
%  NaN dans slot_gap / overlap → uniquement pour le dernier élément
%  (les deux valeurs NaN ensemble déclenchent le solveur)
% ============================================================

LE_main         = [0, 0];           % [mm] position fixe du LE du main
TE_final_target = [588, 318];       % [mm] position cible du TE du dernier
↪  flap

slot_gap        = [15, 8, NaN];     % [mm] NaN = calculé auto (dernier flap
↪  uniquement)
overlap         = [15, 8, NaN];     % [mm] NaN = calculé auto (dernier flap
↪  uniquement)

AoA_flap_deg    = [-45, -73, -84];  % [deg] AoA de chaque flap (nose-down
↪  positif)
pivot_mode      = {'TE', 'TE', 'TE'};
te_target       = [2.1, 1, 1, 1];  % [mm] épaisseur cible TE (main en premier)
te_nb_point     = 200;
AoA_main_deg    = -8                % [deg]
pivot_main      = 'LE';             % obligatoirement 'LE' pour fixer le LE à
↪  LE_main
chord           = [480, 160, 100, 85]; % [mm]
slot_gap_circle = [10, 10, 10, 10];

% Options d'affichage
show_chord_line   = true;
show_slot_circles = true;
show_reg_box      = true;

% ============================================================
%  LECTURE CSV
```

```
% ========================================================
cd(fileparts(mfilename('fullpath')));
data = readmatrix('Reference_Sheet_Matlab_Final_In.csv');

n_cols     = size(data, 2);
n_elements = n_cols / 2;
n_flaps    = length(slot_gap);

fprintf('=== Détection automatique ===\n');
fprintf('Colonnes CSV   : %d\n', n_cols);
fprintf('Éléments total : %d (1 main + %d flap(s))\n\n', n_elements, n_flaps);

assert(length(overlap)         == n_flaps,    'overlap doit avoir %d valeurs', n_flaps);
assert(length(AoA_flap_deg)    == n_flaps,    'AoA_flap_deg doit avoir %d valeurs',    n_flaps);
assert(length(pivot_mode)      == n_flaps,    'pivot_mode doit avoir %d valeurs',      n_flaps);
assert(length(te_target)       == n_elements, 'te_target doit avoir %d valeurs',       n_elements);
assert(length(chord)           == n_elements, 'chord doit avoir %d valeurs', n_elements);
assert(length(slot_gap_circle) == n_elements, 'slot_gap_circle doit avoir %d valeurs', n_elements);
assert(n_elements == n_flaps + 1, ...
    'CSV contient %d éléments mais les paramètres en définissent %d', n_elements, n_flaps+1);

% NaN uniquement sur le dernier flap, et en paire
for i = 1:n_flaps-1
    assert(~isnan(slot_gap(i)) && ~isnan(overlap(i)), ...
        'slot_gap/overlap NaN uniquement sur le dernier flap (index %d)', n_flaps);
end
assert(isnan(slot_gap(end)) == isnan(overlap(end)), ...
    'slot_gap et overlap du dernier flap doivent être tous les deux NaN ou tous les deux définis');

profiles_raw = cell(n_elements, 1);
for k = 1:n_elements
    col_x = data(:, 2*k-1);
    col_y = data(:, 2*k);
    valid = ~isnan(col_x) & ~isnan(col_y);
    profiles_raw{k} = [col_x(valid) * chord(k), col_y(valid) * chord(k)];
end

% ========================================================
%  TRAITEMENT DU PROFIL PRINCIPAL
% ========================================================
[x_main, y_main, x_te_main, y_te_upper_main, y_te_lower_main, ...
 x_te_upper_raw_main, y_te_upper_raw_main] = ...
```

```
    cut_trailing_edge(profiles_raw{1}(:,1), profiles_raw{1}(:,2), te_target(1), te_nb_point);

[~, idx_le_main] = min(x_main);
x_le_main_pt = x_main(idx_le_main);
y_le_main_pt = y_main(idx_le_main);

x_piv_main = x_le_main_pt;
y_piv_main = y_le_main_pt;

theta_main = deg2rad(-AoA_main_deg);
R_main = [cos(theta_main), -sin(theta_main);
          sin(theta_main),  cos(theta_main)];

coords_main_rot = R_main * [(x_main - x_piv_main)'; (y_main - y_piv_main)'];
x_main = coords_main_rot(1,:)' + x_piv_main;
y_main = coords_main_rot(2,:)' + y_piv_main;

te_upper_rot    = R_main * [x_te_upper_raw_main - x_piv_main; y_te_upper_raw_main - y_piv_main];
x_te_main       = te_upper_rot(1) + x_piv_main;
y_te_upper_main = te_upper_rot(2) + y_piv_main;

le_rot          = R_main * [x_le_main_pt - x_piv_main; y_le_main_pt - y_piv_main];
x_le_main_final = le_rot(1) + x_piv_main;
y_le_main_final = le_rot(2) + y_piv_main;

% Translation pour fixer LE à LE_main
dx_fix          = LE_main(1) - x_le_main_final;
dy_fix          = LE_main(2) - y_le_main_final;
x_main          = x_main          + dx_fix;
y_main          = y_main          + dy_fix;
x_te_main       = x_te_main       + dx_fix;
y_te_upper_main = y_te_upper_main + dy_fix;
x_le_main_final = x_le_main_final + dx_fix;
y_le_main_final = y_le_main_final + dy_fix;

fprintf('=== PROFIL PRINCIPAL (AoA=%.1f° | pivot=LE | corde=%.1fmm) ===\n', ...
    AoA_main_deg, chord(1));
[~, idx_le_m] = min(x_main);
fprintf('LE          → X = %.4f | Y = %.4f\n', x_main(idx_le_m), y_main(idx_le_m));
fprintf('TE Upper    → X = %.4f | Y = %.4f\n', x_te_main, y_te_upper_main);
fprintf('TE Lower    → X = %.4f | Y = %.4f\n', x_te_main, y_te_lower_main);
fprintf('Épaisseur TE = %.4f mm  (cible : %.4f)\n\n', ...
    y_te_upper_main - y_te_lower_main, te_target(1));

% ============================================================
%  PRÉ-CALCUL : placer flaps 1..n_flaps-1 pour obtenir la
%  chord line de référence du dernier flap
```

```
126 % ========================================================
127 te_upper_pre = cell(n_flaps, 1);
128 le_pre       = cell(n_flaps, 1);
129 x_ref_pre    = x_te_main;
130 y_ref_pre    = y_te_upper_main;
131
132 for i = 1:n_flaps-1
133     if i == 1
134         vec_chord_pre = [x_te_main - x_le_main_final; y_te_upper_main -
            ↪ y_le_main_final];
135     else
136         vec_chord_pre = [te_upper_pre{i-1}(1) - le_pre{i-1}(1); ...
137                          te_upper_pre{i-1}(2) - le_pre{i-1}(2)];
138     end
139     vec_chord_unit_pre = vec_chord_pre / norm(vec_chord_pre);
140
141     [~, ~, xte, yte, xle, yle] = place_flap( ...
142         profiles_raw{i+1}, te_target(i+1), te_nb_point, ...
143         AoA_flap_deg(i), pivot_mode{i}, ...
144         x_ref_pre, y_ref_pre, vec_chord_unit_pre, ...
145         slot_gap(i), overlap(i));
146
147     te_upper_pre{i} = [xte, yte];
148     le_pre{i}       = [xle, yle];
149     x_ref_pre       = xte;
150     y_ref_pre       = yte;
151 end
152
153 % ========================================================
154 %  SOLVEUR NEWTON-RAPHSON POUR LE DERNIER FLAP
155 %  Trouve slot_gap(end) et overlap(end) tels que
156 %  TE_placed == TE_final_target
157 % ========================================================
158 if isnan(slot_gap(end)) && isnan(overlap(end))
159
160     fprintf('=== Solveur TE cible → dernier flap ===\n');
161
162     % Chord line de l'avant-dernier élément
163     % NOTE : la boucle pré-calcul remplit les indices 1..n_flaps-1.
164     %        Quand n_flaps==1, aucun pré-calcul n'a eu lieu → on utilise le
        ↪ main.
165     %        Quand n_flaps>1,  on utilise l'entrée n_flaps-1 (pas {end} qui
        ↪ vaut n_flaps).
166     if n_flaps == 1
167         vec_chord_last = [x_te_main - x_le_main_final; y_te_upper_main -
            ↪ y_le_main_final];
168     else
169         vec_chord_last = [te_upper_pre{n_flaps-1}(1) - le_pre{n_flaps-1}(1);
            ↪ ...
170                           te_upper_pre{n_flaps-1}(2) - le_pre{n_flaps-1}(2)];
171     end
```

```
    vec_chord_last_unit = vec_chord_last / norm(vec_chord_last);

    x_ref_last = x_ref_pre;
    y_ref_last = y_ref_pre;

    h        = 1e-4;
    tol      = 1e-8;
    max_iter = 50;
    params   = [0; 0];   % point de départ : sg=0, ov=0

    for iter = 1:max_iter
        F = residual_te(params, profiles_raw, te_target, te_nb_point, ...
            AoA_flap_deg, pivot_mode, x_ref_last, y_ref_last, ...
            vec_chord_last_unit, TE_final_target, n_flaps);

        if norm(F) < tol, break; end

        F1 = residual_te(params + [h;0], profiles_raw, te_target, te_nb_point,
        ↪  ...
            AoA_flap_deg, pivot_mode, x_ref_last, y_ref_last, ...
            vec_chord_last_unit, TE_final_target, n_flaps);
        F2 = residual_te(params + [0;h], profiles_raw, te_target, te_nb_point,
        ↪  ...
            AoA_flap_deg, pivot_mode, x_ref_last, y_ref_last, ...
            vec_chord_last_unit, TE_final_target, n_flaps);
        J = [(F1 - F)/h, (F2 - F)/h];

        params = params - J \ F;
        fprintf('  iter %2d | sg=%.4f  ov=%.4f | résidu=%.2e\n', ...
            iter, params(1), params(2), norm( ...
            residual_te(params, profiles_raw, te_target, te_nb_point, ...
            AoA_flap_deg, pivot_mode, x_ref_last, y_ref_last, ...
            vec_chord_last_unit, TE_final_target, n_flaps)));
    end

    F_final = residual_te(params, profiles_raw, te_target, te_nb_point, ...
        AoA_flap_deg, pivot_mode, x_ref_last, y_ref_last, ...
        vec_chord_last_unit, TE_final_target, n_flaps);
    if norm(F_final) > 1e-4
        warning('Solveur non convergé - résidu = %.4e', norm(F_final));
    end

    slot_gap(end) = params(1);
    overlap(end)  = params(2);

    fprintf('→ slot_gap(%d) = %.6f mm\n',   n_flaps, slot_gap(end));
    fprintf('→ overlap(%d)  = %.6f mm\n\n', n_flaps, overlap(end));
end
```

```
% Plot original vs coupé - Main Element
figure; hold on; axis equal; grid on;
title(sprintf('Main Element - Original vs Coupé %.1f mm | corde=%.1f mm | AoA=%.1f°', ...
    te_target(1), chord(1), AoA_main_deg));
xlabel('X [mm]'); ylabel('Y [mm]');

% Original profile: already scaled by chord in profiles_raw, but NOT rotated/translated
% Re-apply the same LE-anchor translation so it sits in the same coordinate frame
x_orig = profiles_raw{1}(:,1) + dx_fix;
y_orig = profiles_raw{1}(:,2) + dy_fix;
plot(x_orig, y_orig, 'b-', 'LineWidth', 2, 'DisplayName', 'Original (non-rotaté)');

% Processed profile: TE-cut + rotated + translated
plot(x_main, y_main, 'r--', 'LineWidth', 2, ...
    'DisplayName', sprintf('Coupé %.1f mm | AoA=%.1f°', te_target(1), AoA_main_deg));

% Mark LE and TE upper for reference
plot(x_le_main_final, y_le_main_final, 'ko', 'MarkerSize', 8, ...
    'MarkerFaceColor', 'k', 'DisplayName', 'LE');
plot(x_te_main, y_te_upper_main, 'gs', 'MarkerSize', 8, ...
    'MarkerFaceColor', 'g', 'DisplayName', 'TE upper');

legend('Location', 'best');


% ============================================================
%  BOUCLE DE PLACEMENT DES FLAPS (tous)
% ============================================================
flap_profiles   = cell(n_flaps, 1);
real_slot_gap   = zeros(1, n_flaps);
real_overlap    = zeros(1, n_flaps);
prev_x          = x_main;
prev_y          = y_main;
te_upper_placed = cell(n_flaps, 1);
le_placed       = cell(n_flaps, 1);
x_ref           = x_te_main;
y_ref           = y_te_upper_main;

for i = 1:n_flaps

    fprintf('=== FLAP %d  (pivot : %s | corde=%.1fmm) ===\n', i, pivot_mode{i}, chord(i+1));

    if i == 1
        vec_chord = [x_te_main - x_le_main_final; y_te_upper_main - y_le_main_final];
```

```
    else
        vec_chord = [te_upper_placed{i-1}(1) - le_placed{i-1}(1); ...
                     te_upper_placed{i-1}(2) - le_placed{i-1}(2)];
    end
    vec_chord_unit = vec_chord / norm(vec_chord);
    vec_perp_unit  = [vec_chord_unit(2); -vec_chord_unit(1)];

    [x_placed, y_placed, x_te_up, y_te_up, x_le_fin, y_le_fin] = place_flap( ...
        profiles_raw{i+1}, te_target(i+1), te_nb_point, ...
        AoA_flap_deg(i), pivot_mode{i}, ...
        x_ref, y_ref, vec_chord_unit, ...
        slot_gap(i), overlap(i));

    flap_profiles{i}   = [x_placed, y_placed];
    te_upper_placed{i} = [x_te_up, y_te_up];
    le_placed{i}       = [x_le_fin, y_le_fin];

    if i == 1
        x_te_prev_ref = x_te_main;
        y_te_prev_ref = y_te_upper_main;
    else
        x_te_prev_ref = te_upper_placed{i-1}(1);
        y_te_prev_ref = te_upper_placed{i-1}(2);
    end

    real_slot_gap(i) = -dot([x_le_fin - x_te_prev_ref; y_le_fin - ...
        y_te_prev_ref], vec_perp_unit);
    real_overlap(i)  = -dot([x_le_fin - x_te_prev_ref; y_le_fin - ...
        y_te_prev_ref], vec_chord_unit);

    min_dist = inf;
    for pt = 1:length(x_placed)
        dists    = sqrt((prev_x - x_placed(pt)).^2 + (prev_y - ...
            y_placed(pt)).^2);
        min_dist = min(min_dist, min(dists));
    end

    x_ref  = x_te_up;
    y_ref  = y_te_up;
    prev_x = x_placed;
    prev_y = y_placed;

    fprintf('LE placé        → X = %.4f | Y = %.4f\n', x_le_fin, y_le_fin);
    fprintf('TE upper        → X = %.4f | Y = %.4f\n', x_ref, y_ref);
    fprintf('Slot gap input  = %.4f mm  |  réel = %.4f mm\n', slot_gap(i), ...
        real_slot_gap(i));
    fprintf('Overlap  input  = %.4f mm  |  réel = %.4f mm\n', overlap(i), ...
        real_overlap(i));
    fprintf('Distance min    = %.4f mm\n\n', min_dist);
end
```

```
310
311 fprintf('=== TE FINAL ===\n');
312 fprintf('TE dernier flap → X = %.4f | Y = %.4f\n', x_ref, y_ref);
313 fprintf('TE cible       → X = %.4f | Y = %.4f\n', TE_final_target(1),
    ↪ TE_final_target(2));
314 fprintf('Écart          → dX = %.4e | dY = %.4e mm\n\n', ...
315     x_ref - TE_final_target(1), y_ref - TE_final_target(2));
316
317 % ============================================================
318 %  EXPORT XLSX
319 % ============================================================
320 all_profiles    = cell(n_elements, 1);
321 all_profiles{1} = [x_main, y_main];
322 for i = 1:n_flaps
323     all_profiles{i+1} = flap_profiles{i};
324 end
325
326 xlsx_file      = 'Reference_Sheet_Matlab_Final_Out.xlsx';
327 sheet_names    = cell(1, n_elements);
328 sheet_names{1} = 'Main';
329 for i = 1:n_flaps
330     sheet_names{i+1} = sprintf('Flap%d', i);
331 end
332
333 for k = 1:n_elements
334     prof  = all_profiles{k};
335     n_pts = size(prof, 1);
336     out_table = array2table( ...
337         [ones(n_pts,1), (1:n_pts)', prof(:,1), prof(:,2), zeros(n_pts,1)], ...
338         'VariableNames', {'ones', 'index', 'X', 'Y', 'zeros'});
339     writetable(out_table, xlsx_file, 'Sheet', sheet_names{k}, 'WriteRowNames',
    ↪ false);
340     fprintf('Onglet %-8s → %d points exportés\n', sheet_names{k}, n_pts);
341 end
342 fprintf('Export XLSX → %s (%d onglets)\n\n', xlsx_file, n_elements);
343
344 % ============================================================
345 %  EXPORT CSV
346 % ============================================================
347 max_len    = max(cellfun(@(p) size(p,1), all_profiles));
348 out_matrix = NaN(max_len, 2 * n_elements);
349 for k = 1:n_elements
350     prof  = all_profiles{k};
351     n_pts = size(prof, 1);
352     out_matrix(1:n_pts, 2*k-1) = prof(:,1);
353     out_matrix(1:n_pts, 2*k)   = prof(:,2);
354 end
355
356 headers    = cell(1, 2 * n_elements);
357 headers{1} = 'x_main';
358 headers{2} = 'y_main';
```

```
for i = 1:n_flaps
    headers{2*i+1} = sprintf('x_flap%d', i);
    headers{2*i+2} = sprintf('y_flap%d', i);
end

fid = fopen('Reference_Sheet_Matlab_Final_Out.csv', 'w');
fprintf(fid, '%s,', headers{1:end-1});
fprintf(fid, '%s\n', headers{end});
for row = 1:max_len
    vals = out_matrix(row, :);
    for col = 1:length(vals)-1
        if isnan(vals(col))
            fprintf(fid, ',');
        else
            fprintf(fid, '%.6f,', vals(col));
        end
    end
    if isnan(vals(end))
        fprintf(fid, '\n');
    else
        fprintf(fid, '%.6f\n', vals(end));
    end
end
fclose(fid);
fprintf('Export CSV → Reference_Sheet_Matlab_Final_Out.csv (%d éléments)\n\n',
↪  n_elements);

% ============================================================
%  FIGURE FINALE
% ============================================================
y_min_plot = min(y_main);
for i = 1:n_flaps
    y_min_plot = min(y_min_plot, min(flap_profiles{i}(:,2)));
end

figure; hold on; axis equal; grid on;
title(sprintf('Multi-Element Wing - %d éléments', n_elements));
xlabel('X [mm]'); ylabel('Y [mm]');

if show_reg_box
    plot([0 886 886 0 0], [0 0 400 400 0], 'w-', 'LineWidth', 2, ...
        'DisplayName', 'Regulation Box 886×400mm');
end

plot(x_main, y_main - y_min_plot, 'b-', 'LineWidth', 2, ...
    'DisplayName', sprintf('Main | AoA=%.1f° | pivot=LE | corde=%.1fmm',
    ↪  AoA_main_deg, chord(1)));

if show_chord_line
    plot([x_le_main_final, x_te_main], ...
         [y_le_main_final - y_min_plot, y_te_upper_main - y_min_plot], ...
```

```
        'b--', 'LineWidth', 1, 'HandleVisibility', 'off');
end

if show_slot_circles
    theta_circ = linspace(0, 2*pi, 100);
    r_main = slot_gap_circle(1);
    plot(x_te_main + r_main * cos(theta_circ), ...
         y_te_upper_main - y_min_plot + r_main * sin(theta_circ), ...
         'b:', 'LineWidth', 1, 'HandleVisibility', 'off');
end

colors = {'r', 'm', 'c', 'y'};
for i = 1:n_flaps
    col = colors{mod(i-1, 4)+1};
    x_p = flap_profiles{i}(:,1);
    y_p = flap_profiles{i}(:,2) - y_min_plot;

    plot(x_p, y_p, 'Color', col, 'LineWidth', 2, ...
        'DisplayName', sprintf( ...
        'Flap %d | AoA=%.1f° | pivot=%s | corde=%.1fmm | gap=%.2f/%.2fmm |
        ↪  ovlp=%.2f/%.2fmm', ...
        i, AoA_flap_deg(i), pivot_mode{i}, chord(i+1), ...
        slot_gap(i), real_slot_gap(i), overlap(i), real_overlap(i)));

    if show_chord_line
        plot([le_placed{i}(1), te_upper_placed{i}(1)], ...
             [le_placed{i}(2) - y_min_plot, te_upper_placed{i}(2) -
             ↪  y_min_plot], ...
             '--', 'Color', col, 'LineWidth', 1, 'HandleVisibility', 'off');
    end

    if show_slot_circles
        if ~exist('theta_circ', 'var'), theta_circ = linspace(0, 2*pi, 100);
        ↪  end
        r_f = slot_gap_circle(i+1);
        plot(te_upper_placed{i}(1) + r_f * cos(theta_circ), ...
             te_upper_placed{i}(2) - y_min_plot + r_f * sin(theta_circ), ...
             ':', 'Color', col, 'LineWidth', 1, 'HandleVisibility', 'off');
    end
end

plot(LE_main(1), LE_main(2) - y_min_plot, 'wo', 'MarkerSize', 10, 'LineWidth',
↪  2, ...
    'DisplayName', sprintf('LE fixe (%.0f, %.0f)', LE_main(1), LE_main(2)));
plot(TE_final_target(1), TE_final_target(2) - y_min_plot, 'w+', 'MarkerSize',
↪  12, 'LineWidth', 2, ...
    'DisplayName', sprintf('TE cible (%.0f, %.0f)', TE_final_target(1),
    ↪  TE_final_target(2)));
plot(x_ref, y_ref - y_min_plot, 'w^', 'MarkerSize', 10, 'LineWidth', 2, ...
    'DisplayName', sprintf('TE final (%.2f, %.2f)', x_ref, y_ref));

```

```
453 legend('Location', 'best');
454
455
456 % ========================================================
457 %  FONCTIONS LOCALES  (doivent être APRÈS tout le code script)
458 % ========================================================
459
460 function [x_out, y_out, x_te_val, y_te_upper, y_te_lower, x_te_upper_raw,
    ↪  y_te_upper_raw] = ...
461         cut_trailing_edge(x, y, target, te_nb_point)
462 % Coupe le TE du profil à l'épaisseur cible.
463
464     seuil = min(x);
465     [~, idx_le] = min(x);
466
467     xu = flipud(x(1:idx_le));
468     yu = flipud(y(1:idx_le));
469     xl = x(idx_le+1:end);
470     yl = y(idx_le+1:end);
471
472     idx_seuil_u = find(xu > seuil, 1);
473     idx_seuil_l = find(xl > seuil, 1);
474
475     xu_te = xu(idx_seuil_u:end);  yu_te = yu(idx_seuil_u:end);
476     xl_te = xl(idx_seuil_l:end);  yl_te = yl(idx_seuil_l:end);
477
478     x_min_c  = max(min(xu_te), min(xl_te));
479     x_max_c  = min(max(xu_te), max(xl_te));
480     x_common = linspace(x_min_c, x_max_c, te_nb_point);
481
482     yu_interp = interp1(xu_te, yu_te, x_common);
483     yl_interp = interp1(xl_te, yl_te, x_common);
484     thickness = yu_interp - yl_interp;
485
486     idx_cut = find(abs(thickness) <= target, 1, 'first') - 1;
487     x_cut   = x_common(idx_cut);
488
489     x_te_val   = x_common(idx_cut);
490     y_te_upper = yu_interp(idx_cut);
491     y_te_lower = yl_interp(idx_cut);
492
493     mask_u_cut     = xu <= x_cut;
494     xu_kept        = xu(mask_u_cut);
495     yu_kept        = yu(mask_u_cut);
496     x_te_upper_raw = xu_kept(end);
497     y_te_upper_raw = yu_kept(end);
498
499     mask_l_cut = xl <= x_cut;
500     xl_kept    = xl(mask_l_cut);
501     yl_kept    = yl(mask_l_cut);
502
```

```
503     if x(1) > x(end)
504         x_out = [flipud(xu_kept); xl_kept];
505         y_out = [flipud(yu_kept); yl_kept];
506     else
507         x_out = [xu_kept; flipud(xl_kept)];
508         y_out = [yu_kept; flipud(yl_kept)];
509     end
510 
511     mask_nan = ~isnan(x_out) & ~isnan(y_out);
512     x_out    = x_out(mask_nan);
513     y_out    = y_out(mask_nan);
514 end
515 
516 
517 function [x_placed, y_placed, x_te_up_out, y_te_up_out, x_le_out, y_le_out] =
    ↪  ...
518         place_flap(profiles_raw_i, te_target_i, te_nb_point, ...
519                    AoA_i, pivot_i, ...
520                    x_ref_in, y_ref_in, vec_chord_unit_in, ...
521                    sg, ov)
522 % Coupe, tourne et positionne un flap selon slot_gap sg et overlap ov.
523 
524     [x_f, y_f, ~, ~, ~, x_te_raw, y_te_raw] = ...
525         cut_trailing_edge(profiles_raw_i(:,1), profiles_raw_i(:,2),
            ↪  te_target_i, te_nb_point);
526 
527     [~, idx_le_f] = min(x_f);
528     x_le_f = x_f(idx_le_f);
529     y_le_f = y_f(idx_le_f);
530 
531     if strcmp(pivot_i, 'LE')
532         x_piv = x_le_f;    y_piv = y_le_f;
533     else
534         x_piv = x_te_raw;  y_piv = y_te_raw;
535     end
536 
537     theta = deg2rad(-AoA_i);
538     R = [cos(theta), -sin(theta); sin(theta), cos(theta)];
539 
540     coords_rot  = R * [(x_f - x_piv)'; (y_f - y_piv)'];
541     x_rot = coords_rot(1,:)' + x_piv;
542     y_rot = coords_rot(2,:)' + y_piv;
543 
544     vec_perp_unit = [vec_chord_unit_in(2); -vec_chord_unit_in(1)];
545 
546     x_le_target = x_ref_in - ov * vec_chord_unit_in(1) - sg * vec_perp_unit(1);
547     y_le_target = y_ref_in - ov * vec_chord_unit_in(2) - sg * vec_perp_unit(2);
548 
549     le_rot_f = R * [x_le_f - x_piv; y_le_f - y_piv];
550     dx = x_le_target - (le_rot_f(1) + x_piv);
551     dy = y_le_target - (le_rot_f(2) + y_piv);
```

```
552
553     x_placed = x_rot + dx;
554     y_placed = y_rot + dy;
555
556     te_rot_f    = R * [x_te_raw - x_piv; y_te_raw - y_piv];
557     x_te_up_out = te_rot_f(1) + x_piv + dx;
558     y_te_up_out = te_rot_f(2) + y_piv + dy;
559
560     x_le_out = le_rot_f(1) + x_piv + dx;
561     y_le_out = le_rot_f(2) + y_piv + dy;
562 end
563
564
565 function res = residual_te(params, profiles_raw, te_target, te_nb_point, ...
566         AoA_flap_deg, pivot_mode, x_ref_last, y_ref_last, ...
567         vec_chord_last_unit, TE_final_target, n_flaps)
568 % Résidu Newton : écart entre TE placé du dernier flap et TE_final_target.
569
570     sg_try = params(1);
571     ov_try = params(2);
572     [~, ~, xte_try, yte_try, ~, ~] = place_flap( ...
573         profiles_raw{n_flaps+1}, te_target(n_flaps+1), te_nb_point, ...
574         AoA_flap_deg(n_flaps), pivot_mode{n_flaps}, ...
575         x_ref_last, y_ref_last, vec_chord_last_unit, ...
576         sg_try, ov_try);
577     res = [xte_try - TE_final_target(1); yte_try - TE_final_target(2)];
578 end
579
```

5.5 Solver method simuations for 2D workflow

subsubsectionRANS equation recall Before discussing the different approaches considered, a brief reminder of the RANS equations is introduced.
The Reynolds-Averaged Navier–Stokes (RANS) equations are obtained by decomposing each flow variable into a mean component and a fluctuating turbulent component. Applying this decomposition to the Navier–Stokes equations and performing a time average yields a set of equations describing the behaviour of the mean flow field. This approach significantly reduces the computational cost compared to directly resolving all turbulent scales, making it particularly suitable for engineering aerodynamic applications.
The averaging process introduces additional unknown terms, known as Reynolds stresses, which require a turbulence closure model to complete the system of equations. Turbulence models provide approximations for these terms and differ in complexity, accuracy, and computational cost. Among the various available approaches, RANS-based turbulence models remain the most widely used framework in industrial aerodynamics due to their balance between computational efficiency and physical fidelity.

5.5.0.1 Options Tested

Four solver configurations are compared on the initial refined mesh, where the reference configuration established by the aerodynamics division which is a coupled solver with second order spatial discretisation and serves as the baseline for this comparison. The discretisation methods listed in the table below, first order and second order are used for momentum and turbluent kinetic enregy while keeping a second order one for pressure for every options considered.

Case	Coupling	Discretisation
1	COUPLED	2nd order upwind
2	COUPLED	1st order upwind
3	SIMPLE	1st order upwind
4	SIMPLE	2nd order upwind

Table 5.1: Solver configurations compared in the method study.

5.5.0.2 Mathematical Differences

The four configurations stem from the combination of two discretisation orders and two pressure-velocity coupling schemes, each presenting a different trade-off between stability, accuracy, and computational cost.

Discretisation order. First-order schemes are unconditionally stable and converge easily, making them useful as a starting point. However, they tend to smooth out sharp flow gradients, which can reduce the accuracy of the solution in critical regions such as the boundary layer and the slots. Second-order schemes significantly reduce this diffusion and are the standard choice for aerodynamic simulations requiring accurate resolution of flow gradients, at the cost of slightly harder convergence.

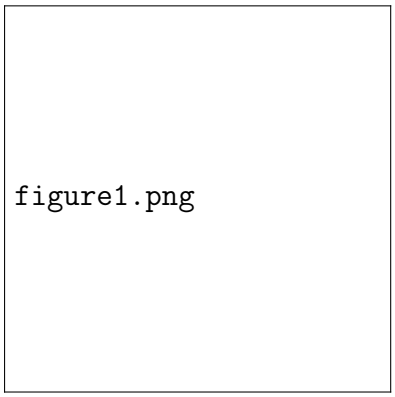
Pressure-velocity coupling. The coupled scheme solves the momentum and pressure equations simultaneously, offering faster convergence in strongly coupled regions such as narrow slots, at a higher cost per iteration. The SIMPLE scheme uses a segregated pressure-correction approach, with cheaper iterations but slower overall convergence and reduced stability in strongly coupled flows.

5.5.0.3 Results and Justification

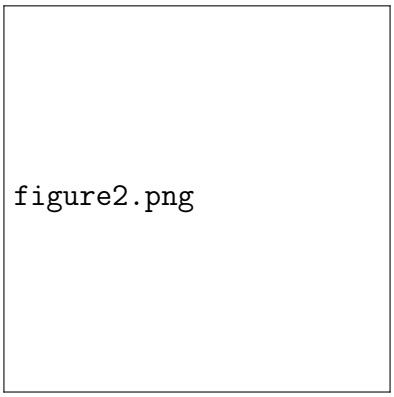
The solver configurations are evaluated on the same geometry and boundary conditions, so that the comparison isolates the effect of the numerical scheme alone. The selection criteria are the convergence of the residuals, the stability of the lift and drag coefficients, and the velocity field in the critical regions of the domain. The results of this comparison are presented below.

Configuration	C_L	C_D	C_L/C_D
Coupled — First order			
Coupled — Second order			
SIMPLE — First order			
SIMPLE — Second order			

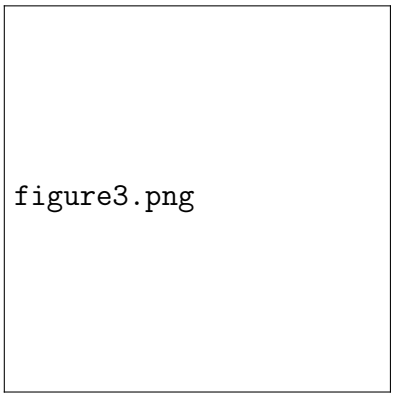
Table 5.2: Lift and drag coefficients for each solver configuration



(a) Residuals

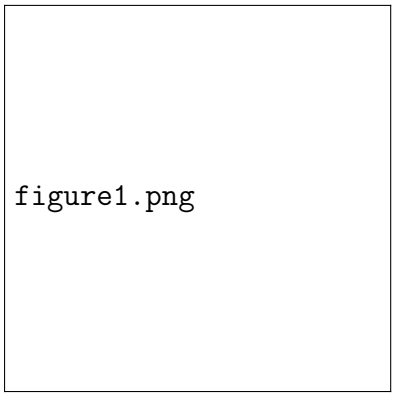


(b) Cl / Cd plot



(c) Velocity Contours

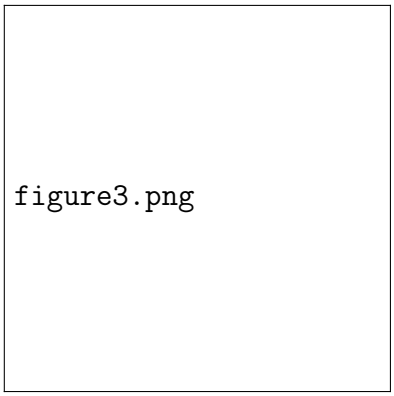
Figure 5.1: Option 1



(a) Residuals

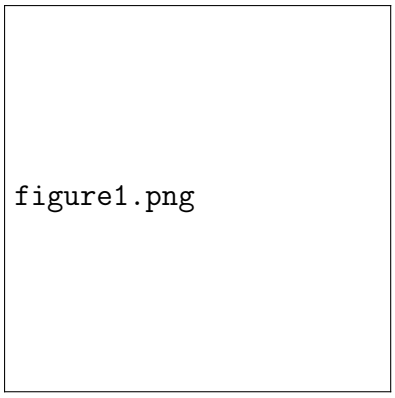


(b) Cl / Cd plot

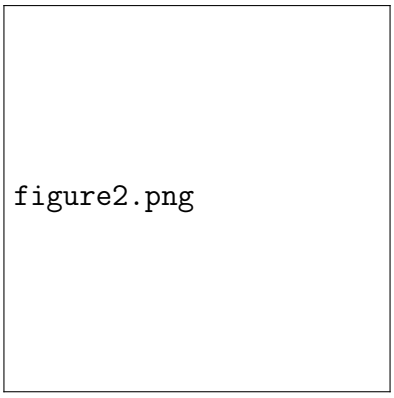


(c) Velocity Contours

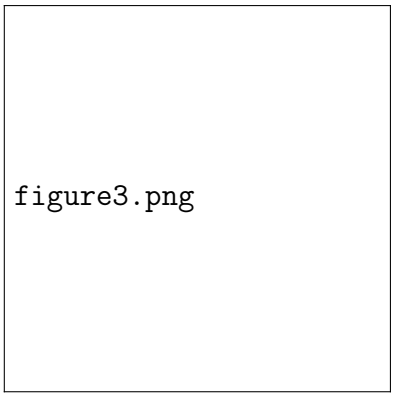
Figure 5.2: Option 2



(a) Residuals

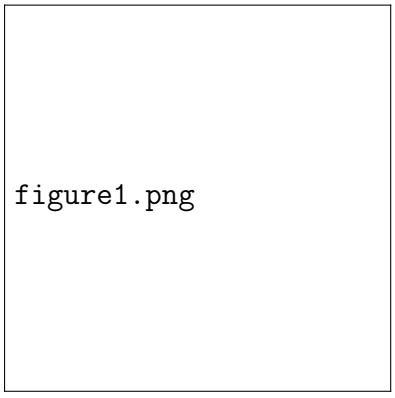


(b) Cl / Cd plot

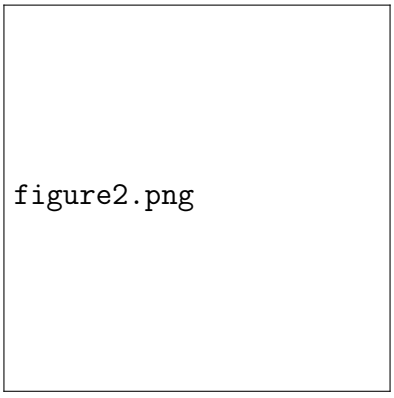


(c) Velocity Contours

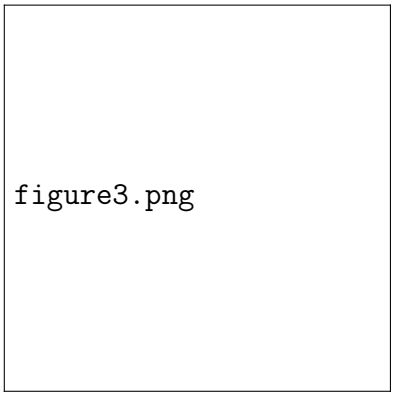
Figure 5.3: Option 3



(a) Residuals



(b) Cl / Cd plot



(c) Velocity Contours

Figure 5.4: Option 4

The results clearly show that the SIMPLE-based configurations fail to resolve the flow correctly in the slot regions, making them incompatible with the requirements of this study. The coupled approach is therefore the only viable option for this configuration. Between the two coupled configurations, the first-order discretisation scheme is selected.

The second-order scheme exhibited convergence difficulties near the slot regions due to locally high skewness introduced by the adaptive refinement process, while the first-order scheme demonstrated stable convergence at a reasonable computational cost. This loss of accuracy is accepted since the primary objective is to rank geometric configurations relative to each other, rather than to predict absolute aerodynamic values.

The coupled solver with first-order discretisation is therefore retained as the numerical method for all subsequent simulations in this study.

Bibliography

[1] Formula Student Germany, "Outlook for 2027 – aerodynamics (draft, subject to changes)," FSG Academy slides, Oct. 2025, draft only, subject to changes.

[2] Ansys, "Fluent user's guide," https://ansyshelp.ansys.com/public/account/secured?returnurl=/Views/Secured/corp/v261/en/flu_ug/flu_ug.html, accessed: 2026-06-01.

[3] ——, "Fluent text command list," https://ansyshelp.ansys.com/public/account/secured?returnurl=/Views/Secured/corp/v261/en/flu_tcl/flu_tcl.html, accessed: 2026-06-01.

[4] ——, "Fluent customization manual," https://ansyshelp.ansys.com/public/account/secured?returnurl=/Views/Secured/corp/v261/en/flu_udf/flu_udf.html, accessed: 2026-06-01.

[5] M. Javurek, "Scheme-programmierung in FLUENT 5 & 6," unpublished / internal document.

[6] LIPS Scheme, "Introduction to scheme," https://lips.js.org/docs/category/introduction-to-scheme, accessed: 2026-06-01.

[7] J. D. Anderson, *Fundamentals of Aerodynamics*, 5th ed. McGraw-Hill, 2011.

[8] Y. Liang, L. Zhang, E. Li, X. Liu, and Y. Yang, "Design considerations of rotor configuration for straight-bladed vertical axis wind turbines," *Advances in Mechanical Engineering*, vol. 2014, pp. 1–15, 2014.

[9] J. D. Anderson, *Introduction to Flight*, 8th ed. McGraw-Hill Education, 2016.

[10] Embry-Riddle Aeronautical University. (n.d.) Boundary layers. Accessed: 2026-06-01. [Online]. Available: https://eaglepubs.erau.edu/introductiontoaerospaceflightvehicles/chapter/boundary-layers/

[11] *Prediction and Validation Technologies of Aerodynamic Force and Heat for Hypersonic Vehicle Design*. Amazon Digital Services LLC, 2021, accessed: 2026-06-01. [Online]. Available: https://www.amazon.com/Prediction-Validation-Technologies-Aerodynamic-Hypersonic-ebook/dp/B08Y7NGGN4

[12] J. G. Leishman, *Introduction to Aerospace Flight Vehicles*. Daytona Beach, FL: Embry-Riddle Aeronautical University, 2023, accessed: 2026-05-30. [Online]. Available: https://eaglepubs.erau.edu/introductiontoaerospaceflightvehicles/

[13] I. H. Abbott and A. E. von Doenhoff, *Theory of Wing Sections: Including a Summary of Airfoil Data*. Dover Publications, 1959.

[14] J. Katz, *Race Car Aerodynamics: Designing for Speed*. Bentley Publishers, 1995.

[15] D. P. Coiro, F. Nicolosi, and F. Grasso, "Design and testing of multi-element airfoil for short-takeoff-and-landing ultralight aircraft," *Journal of Aircraft*, 2010.

[16] D. Izquierdo, "Multi-element wing configuration and boundary layer treatment sketch," Online publication figure, n.d., fig. 5, uploaded publication; content may be subject to copyright; own elaboration mentioned in source.

[17] J. D. C. Zerihan, "An investigation into the aerodynamics of wings in ground effect," Ph.D. dissertation, University of Southampton, School of Engineering Sciences, 2001.

[18] Hanley Innovations. (n.d.) Me-foil software and innovation panel. Accessed: 2026-06-01. [Online]. Available: https://www.hanleyinnovations.com/mefoil.html

[19] A. Spentzos, G. N. Barakos, K. J. Badcock, and B. E. Richards, "Investigation of three-dimensional dynamic stall using computational fluid dynamics," *AIAA Journal*, vol. 43, no. 5, pp. 1023–1033, 2005.

[20] Formula Student Germany, "Formula student rules 2026, version 1.1," https://www.formulastudent.de/fileadmin/user_upload/all/2026/rules/FS-Rules_2026_v1.1.pdf, Nov. 2025, accessed: 2026-05-23.

[21] M. Abid, H. A. Wajid, M. Z. Iqbal, S. Najam, A. Arshad, and A. Ahmad, "Design and analysis of an aerodynamic downforce package for a formula student race car," *IIUM Engineering Journal*, vol. 18, no. 2, pp. 212–224, 2017. [Online]. Available: https://www.researchgate.net/publication/321888416_DESIGN_AND_ANALYSIS_OF_AN_AERODYNAMIC_DOWNFORCE_PACKAGE_FOR_A_FORMULA_STUDENT_RACE_CAR

[22] SAE International, "Vehicle dynamics terminology," SAE International, Warrendale, PA, USA, Standard J670_200801, 2008.

[23] X. Zhang and J. Zerihan, "Aerodynamics of a double-element wing in ground effect," *AIAA Journal*, vol. 41, no. 6, pp. 1007–1016, 2003.