UNIVERSITÀ DEGLI STUDI DI GENOVA

SCUOLA POLITECNICA

DIME

Dipartimento di Ingegneria Meccanica, Energetica, Gestionale e dei Trasporti

TESI DI LAUREA MAGISTRALE IN INGEGNERIA MECCANICA
– ENERGIA E AERONAUTICA

Integrating Geometry Tools for Conceptual Aircraft Design in CEASIOMpy

Relatore:
Prof. Ing. Alessandro Bottaro
Correlatore:
Dott. Giacomo Benedetti
Dott. Ing. Jan Vos

Allievo:
Nicolò Perasso

Marzo 2026

In collaboration with:

CFS Engineering

Computational Fluids & Structures Engineering

Acknowledgements

The research presented in this paper has been performed in the framework of the Colossus project (Collaborative System of Systems Exploration of Aviation Products, Services and Business models) and has received funding from the European Union Horizon Europe Programme under grant agreement no 101097120. The Swiss participation in the Colossus project was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 22.00609.

Ringraziamenti

Difficile racchiudere in poche righe cosa rappresenta questa tesi; dietro al lavoro e alle pagine che si susseguono ci sono anni di fatica ma di grande soddisfazione. Tenacia, passione e tanta voglia di fare mi hanno portato ad essere la persona che sono adesso. Nonostante sia impossibile citare tutte le persone che in un modo o nell'altro mi hanno fatto crescere negli anni, il più sentito ringraziamento va al mio relatore Prof Bottaro e ai colleghi della CFS Engineering i quali mi hanno accompagnato in questa bellissima avventura a Losanna. Un ringraziamento va alla mia famiglia sempre col "fucile puntato", come dice la mamma, ma dalla mia parte anche nei momenti piu difficili. E in particolare a Giulia, la quale mi ha fatto riscoprire un parte dentro di me che pensavo ormai persa. Ai miei colleghi dell'università e ai miei amici di Busalla coi quali ho condiviso questi anni, sempre pronti ad uscire e vederci nonostante in alcuni frangenti possa essere stato assente. A tutti i miei parenti, in particolare a mio zio Mattia e Miriam che hanno dato un valore in più a questi sei mesi in Svizzera.

Alla fine non si tratta della destinazione ma del viaggio stesso.

Integrating Geometry Tools for Conceptual Aircraft Design in CEASIOMpy

Sommario

Al fine di rispondere alla crescente esigenza, in ambito ingegneristico aeronautico, di velocizzare le procedure di design preliminare dei velivoli, il software CEASIOMpy rappresenta uno strumento di particolare interesse. CEASIOMpy è un software open–source, sviluppato e mantenuto da CFS Engineering e Airinnova, dedicato al design preliminare di velivoli e in grado di offrire differenti workflow per simulazioni fluidodinamiche a bassa e alta fedeltà. Il lavoro di questa tesi si colloca nell'obiettivo di ampliare le capacità del software, migliorando la generazione di geometrie complesse e non convenzionali attraverso l'implementazione di due moduli dedicati. Il primo modulo, VSP2CPACS, introduce l'integrazione di OpenVSP *(Open Vehicle Sketch Pad)*, un software CAD open–source sviluppato dalla NASA, che fornisce un'interfaccia grafica intuitiva e consente all'utente di iterare rapidamente diverse configurazioni geometriche durante la fase di design. L'integrazione è stata realizzata traducendo i componenti definiti in OpenVSP nei corrispondenti elementi del formato CPACS *(Common Parametric Aircraft Configuration Schema)*. Il secondo modulo, STL2CPACS, implementa una procedura di slicing di componenti in formato STL, dalla quale vengono ricavate nuvole di punti utilizzate per l'identificazione dei profili e dei parametri geometrici necessari alla generazione automatica del file CPACS.

Integrating Geometry Tools for Conceptual Aircraft Design in CEASIOMpy

Abstract

To address the growing demand for efficient preliminary aircraft design workflows in the aerospace sector, the CEASIOMpy framework represents a tool of significant interest. Developed by CFS Engineering and Airinnova, CEASIOMpy is an open-source software designed to facilitate both low and high fidelity fluid dynamics simulations. The objective of this thesis is to extend the software's capabilities, specifically improving geometry generation for complex and non–conventional prototypes through the implementation of two dedicated modules. The first module, VSP2CPACS, introduces the integration of OpenVSP *(Open Vehicle Sketch Pad)*, an open-source parametric geometry tool developed by NASA. This integration makes use of OpenVSP's intuitive user interface to enable rapid iteration of geometric configurations during the design phase, translating OpenVSP components into elements compatible with the CPACS *(Common Parametric Aircraft Configuration Schema)* format. The second module, STL2CPACS, implements a reconstruction procedure for components defined by STL files. By applying a slicing algorithm to detect point clouds at various cross-sections, the module identifies airfoils and geometric parameters to automatically generate the corresponding CPACS file.

Contents

1. Introduction

Designing an aircraft, particularly during the preliminary design stage, is one of the most complex tasks in engineering. As an example, the preliminary design of the Boeing 777 involved approximately 3000 engineers, coordinated through weekly design meetings among 25 lead engineers [7], each representing more than 100 specialists. Figure 1.1 illustrates the conceptual and preliminary design flow of a typical transport aircraft, structured into four main cycles:

- **Cycle 1**: speculative design, exploring a wide design space.
- **Cycle 2**: baseline refinement to demonstrate feasibility.
- **Cycle 3**: configuration development leading to design freeze.
- **Cycle 4**: final hardware design and production.

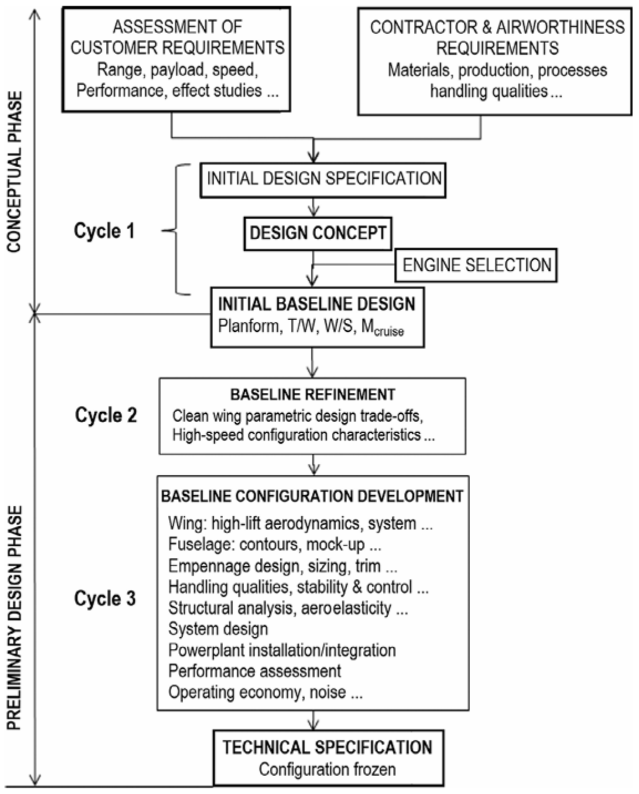

ASSESSMENT OF CUSTOMER REQUIREMENTS
Range, payload, speed, Performance, effect studies ...
CONTRACTOR & AIRWORTHINESS REQUIREMENTS
Materials, production, processes handling qualities ...
CONCEPTUAL PHASE
INITIAL DESIGN SPECIFICATION
Cycle 1
DESIGN CONCEPT
ENGINE SELECTION
INITIAL BASELINE DESIGN
Planform, T/W, W/S, M_{cruise}
Cycle 2
BASELINE REFINEMENT
Clean wing parametric design trade-offs, High-speed configuration characteristics ...
PRELIMINARY DESIGN PHASE
Cycle 3
BASELINE CONFIGURATION DEVELOPMENT
Wing: high-lift aerodynamics, system ...
Fuselage: contours, mock-up ...
Empennage design, sizing, trim ...
Handling qualities, stability & control ...
Structural analysis, aeroelasticity ...
System design
Powerplant installation/integration
Performance assessment
Operating economy, noise ...
TECHNICAL SPECIFICATION
Configuration frozen

Figure 1.1: Conceptual and preliminary design flowchart of a typical transport aircraft

The multidisciplinary nature of preliminary aircraft design highlights the need for advanced simulation and data management environments capable of handling complex workflows. In this context, open-source tools such as CEASIOMpy play a fundamental role in extending advanced design capabilities to a broader user base [10].

1.1 CEASIOMpy Environment

CEASIOMpy is an open-source software written in Python distributed via GitHub or available as web application [11], designed to support conceptual aircraft design. It is structured as a modular environment in which individual tools can be connected and executed according to user-defined workflows. Its capabilities include geometry generation, mesh creation, weight and balance estimation, aerodynamic analysis (Vortex Lattice Method [14] and SU2 [13]), and stability analysis. CEASIOMpy relies on the open–standard CPACS format, developed by the German Aerospace Center (DLR), to enable standardized data exchange between modules [9].

Figure 1.2: CEASIOMpy logo

1.2 CPACS format

CPACS *(Common Parametric Aircraft Configuration Schema)* is an XML–based data definition used to describe aircraft configurations in a structured and hierarchical manner. It allows the representation of geometry, aerodynamic datasets, mission definitions, and other discipline-specific information within a single standardized file. In CPACS, aircraft geometry is defined using a top–down approach, where high–level concepts are progressively decomposed into detailed component descriptions. For example, wing geometry is defined by a sequence of spanwise sections, each associated with an airfoil profile and positioned in the global reference frame through scaling, rotation, and translation [9].

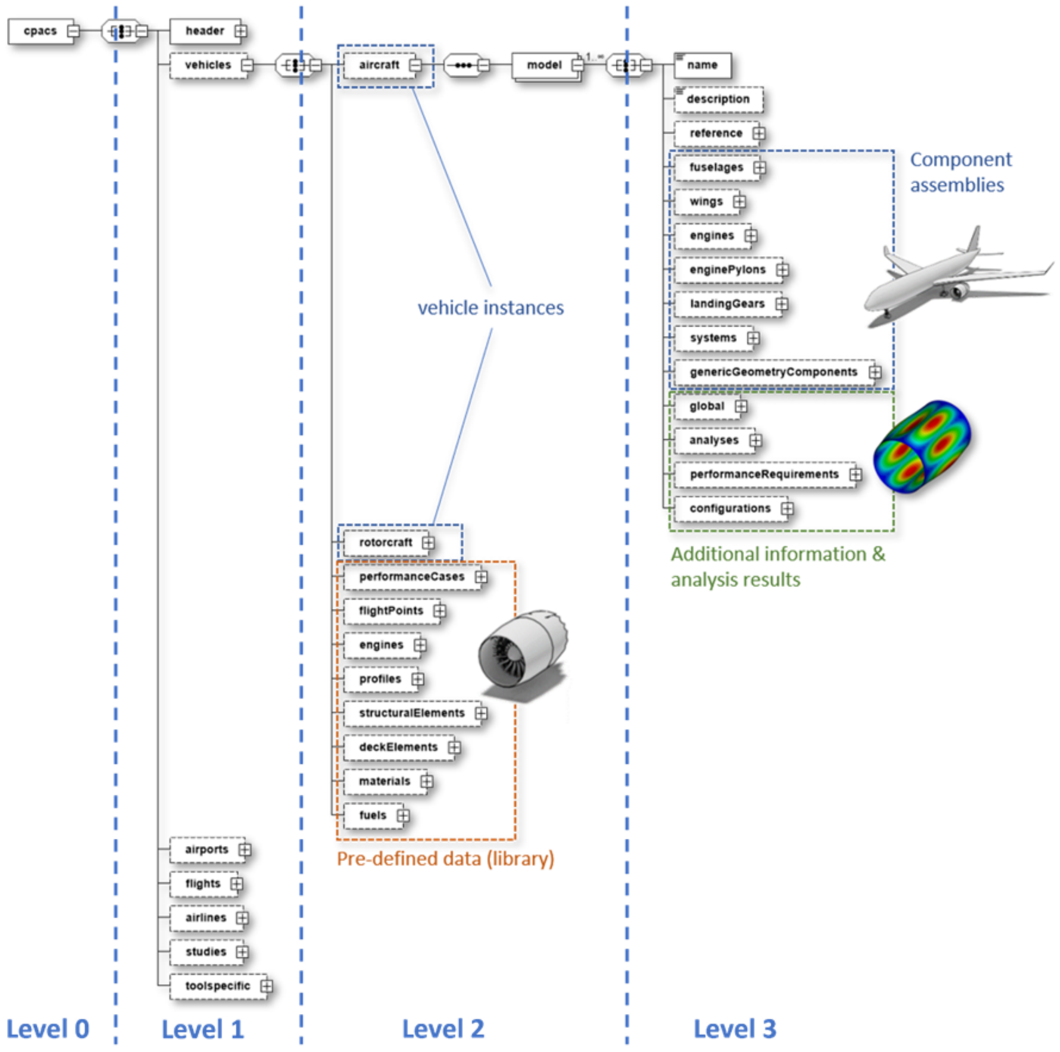


Figure 1.3: CPACS hierarchical structure overview

Level 1 is composed of seven distinct elements, briefly outlined in the following list:

- **header**: within this node, details concerning the dataset's name are found, including its description, and the version of the original CPACS dataset. Additionally, it also provides information about the CPACS version.
- **vehicles**: this node includes two groups of elements, vehicle instances (comprises the nodes utilized to define an aircraft and/or a rotorcraft) and pre-defined data (encompasses additional data that can be referenced within the definition of a vehicle). It includes components, physical properties, mission definition and flight points.
- **airports**: in this node, airports are listed along with details about their locations and runway characteristics, including available runway length and position.
- **flight**: contains all flight definitions.
- **airlines**: within this element, it is possible to define one or more aircraft fleets.

- **studies**: contains optimization data such as definitions of design parameters and design studies for both design of experiments and optimization.
- **toolspecific**: this node's purpose is to store the inputs and outputs generated by the tools employed in the aircraft design process and optimization.

Inside Level 2 — *aircraft* — the focus is on the geometric elements used in the following chapters. This level includes a predefined library containing information about the profiles, where all the airfoils for wing or fuselage components are defined, as well as the engine component. Inside *model* the user must define the aircraft's components. Element *wings* contains all lifting surfaces as wings, horizontal and vertical tailplanes and canards. The *fuselage* element contains all the geometries that can be defined as a fuselage types. There are *Engine* element and other aeronautical feature that will be explained. In this thesis, the focus will be on the three components *wings*,*fuselages* and *engines* where in details are described in the next sub-chapter.

1.2.1 Definition of a wing inside the CPACS

In CPACS, a wing is defined as a hierarchical geometric object that organizes the definition of a lifting surface,such as a main wing, horizontal tailplane, or canard, through a sequence of cross-section, with associated airfoil profiles, using spanwise segments that connect them. The wing element itself is contained within the `<wings>` section of the aircraft model and is identified by a unique `uID`, which is used to reference the wing throughout the file. In addition to this identifier, each wing may carry a name and a description, together with a global transformation that defines its position, orientation, and scaling with respect to the aircraft coordinate system. The core of the wing geometry lies in its sections. Each section corresponds to a specific location along the wing span and is defined by a `<section>` node. Inside each section, one or more `<element>` items are specified; each element associates a 2D airfoil profile to that section by means of an `airfoilUID` that points to a previously defined `<wingAirfoil>` under the `<profiles>` section. Each element may also carry its own local transformation, which can scale the airfoil chord or re-position the section in space, allowing different chord lengths and local offsets along the span. This structure makes it possible to use the same airfoil description at multiple stations while applying different chord scalings, twist, or local offsets as needed. Between these sections, CPACS introduces a second level of parametric control through positioning nodes. Each `<positioning>` element defines the relative vector between two consecutive sections, encoding parameters such as the section length, sweep angle, and dihedral angle. This allows that the wing planform own a parametrization that is both intuitive and consistent with standard aircraft design practice. The segment nodes then complete the definition by explicitly linking pairs of section elements into continuous wing segments. A `<segment>` is defined by a `<fromElementUID>` and a `<toElementUID>`, indicating which section elements at the root and tip are connected, and CPACSCreator, the visualization tool, use this information to interpolate the wing surface between the them. The 2D airfoil profiles are stored separately in the `<profiles>` hierarchy, under `<wingAirfoils>`. Each `<wingAirfoil>` contains a `<pointList>` with vectors `x`, `y`, and `z`, describing a normalized airfoil contour in the local (x, z) plane, where x is along the chord and z is in the thickness direction. CPACS wants a specific order for the airfoil coordinates; the poitns needs to be ordered from the trailing edge along the lower surface to the leading edge, and then back along the upper surface to the trailing edge. By keeping the airfoil definitions in a global profile library and referencing them from the wing elements, CPACS ensures that the wing geometry remains compact, reusable, and consistent across the model. In the example shown in Fig 1.4, this structure is clearly visible: the wing is built from sections. For clearness, j is the index of every section, each of them contains an element, connected by a single segment and controlled by positioning node, with the correspond airfoil. Under the CPACS tree, there is a visualization using CPACSCreator of the wing, in this case is comes from a supersonic conceput aircraft, available in the test files of CEASIOMpy.

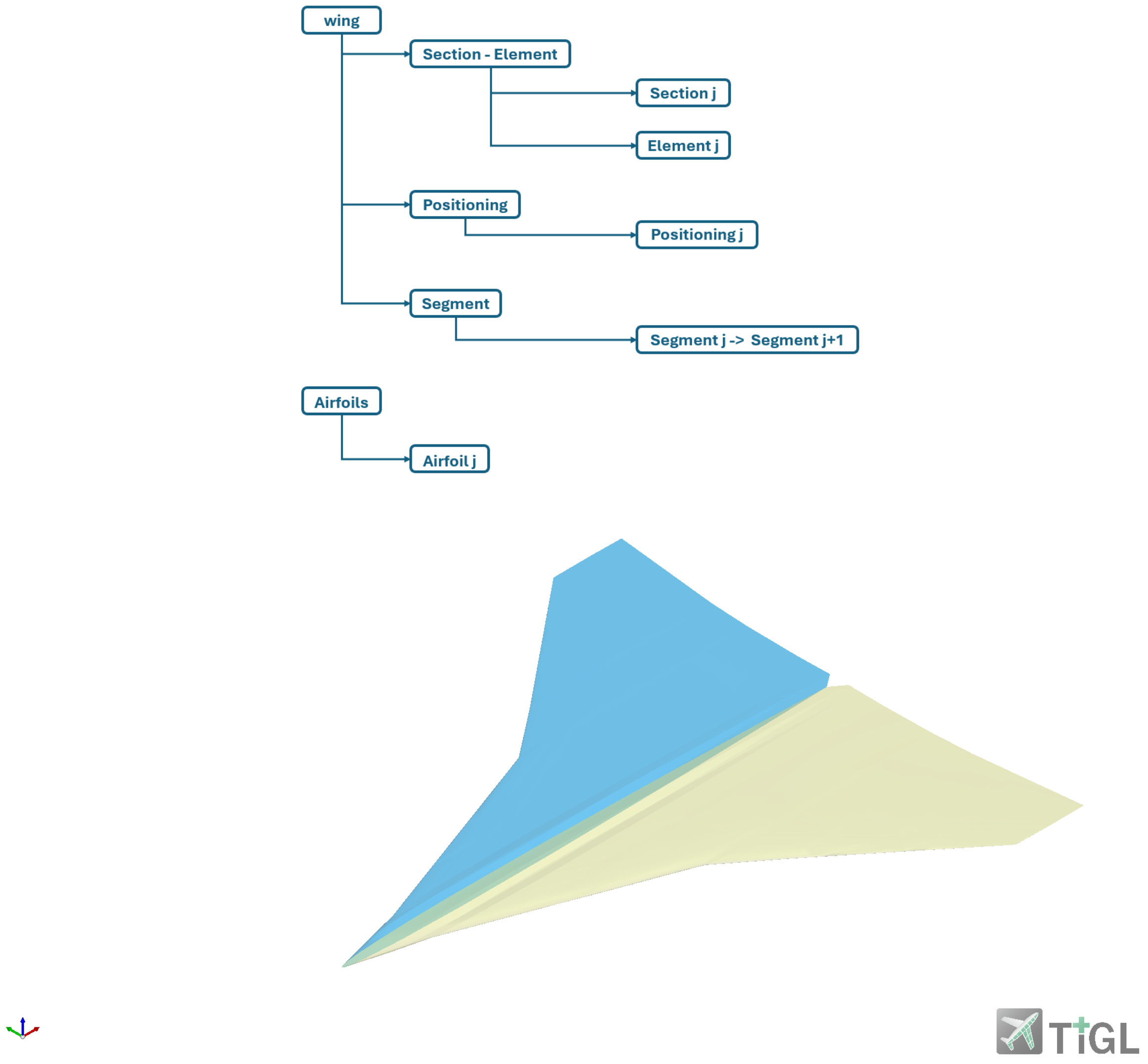


Figure 1.4: Wing definition in CPACS

1.2.2 Definition of a fuselage in CPACS

In CPACS [9], a fuselage is defined as a longitudinal body whose shape is constructed from a sequence of cross-sectional profiles and the segments that connect them along the fuselage axis. The fuselage element itself is under the `<fuselages>` container of the aircraft model and is identified by a `uID`, with an optional name and description, together with a global transformation that positions and orients the complete fuselage structure within the aircraft coordinate system. This high-level structure is conceptually similar to that of a wing, but instead of chordwise airfoils, the fuselage is built from closed 2D profiles in the local (y, z) plane, describing the cross-section of the body at discrete stations along its length. The core of the fuselage definition lies in its sections. Each section represents a cross-section along the fuselage axis and is defined by a <section> node. Within each of them, one or more <element> items are specified; each element references a 2D profile through a `profileUID` that points to a <fuselageProfile> under the <profiles> section. Each element may carry its own local transformation, which scales and repositions the profile at that station and allows the fuselage to vary in cross-sectional size and orientation along the axis. Between these sections, CPACS organizes the fuselage geometry through a set of positioning nodes. Each <positioning> element defines the relative offset between two consecutive sections, encoding the length between them and any angular parameters as sweep and dihedral. The segment nodes then complete the geometric description by connecting pairs of section elements into continuous fuselage segments. A <segment> is specified by a <fromElementUID> and a <toElementUID>,as the wing explained before. The fuselage profiles themselves are stored in the global <profiles> hierarchy, under <fuselageProfiles>, where each <fuselageProfile> contains a vectors `x`, `y`, and `z` that describe the normalized profile. In practice, the x-coordinate is often kept constant or set to zero, and the profile is effectively defined in the (y, z) plane, with the ordering following the CPACS convention for closed profiles (e.g., bottom $\rightarrow$ right side $\rightarrow$ top $\rightarrow$ left side $\rightarrow$ bottom). In Fig 1.5, these is an example of CPACS fuselage, it is the correspond part of the D150,that is shows under the tree.

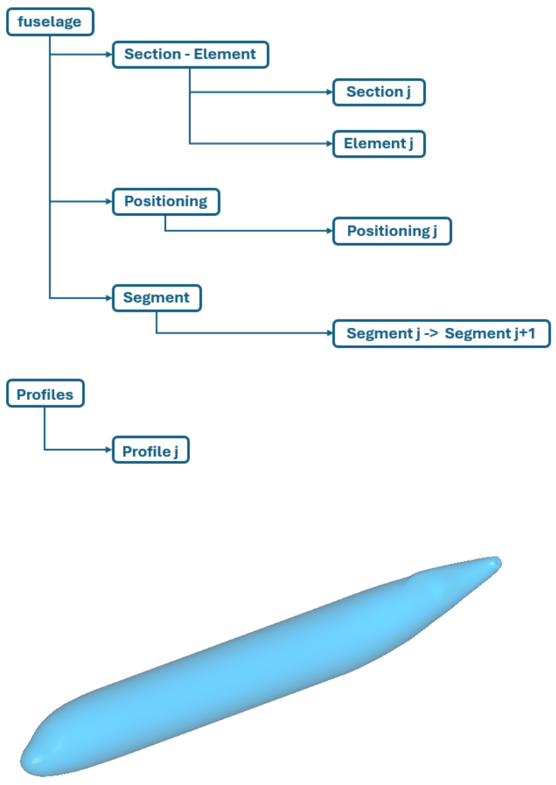


Figure 1.5: Fuselage definition in CPACS

1.2.3 Definition of an engine in CPACS

The engine is one of the most important components of an aircraft, and its definition in CPACS is crucial for accurate performance and aerodynamic analyses. In CPACS, an engine is defined in a more complex manner than the *wing* or *fuselage*; it is defined in two points inside a CPACS file one before the `<aircraft>` node and one inside it. The first *Engine* call specify the structure of it, where requires the definition of the nacelle. This describes the external shape of the propulsion system and it is decomposed into three main elements: the fan cowl, the core cowl and the center cowl. This subdivision is primarily intended for turbofan (bypass) engines. In the case of a non-bypass (turbojet) configuration, the *core cowl* can be omitted. The *center cowl* typically represents the nose cone or spinner region. To clarify the meaning of these elements, the CPACS documentation provides a schematic representation of the fan, core, and center cowls, as shown in Figure 1.6.

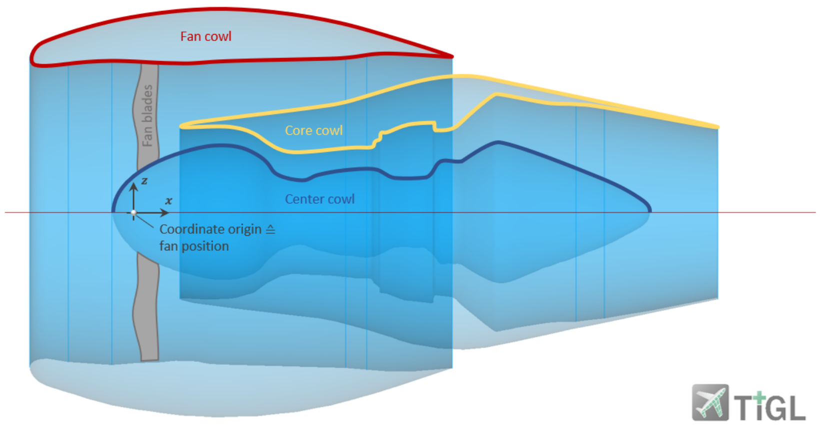


Figure 1.6: Engine cowl components in CPACS

Geometrically, the fan and core cowls are defined by longitudinal sections in the x–z plane. These sections are revolved around the x-axis to generate axisymmetric bodies of revolution. At least one longitudinal section must be defined for each cowl component. Once the section that will be revolved is defined, CPACS gives capabilities to set a shape close to the real life where a propulsion system with compressor, combustion chamber and turbine is present and is not a perfect body of revolution. It means to create the typical convergent-divergent shape as shown in Fig 1.7.

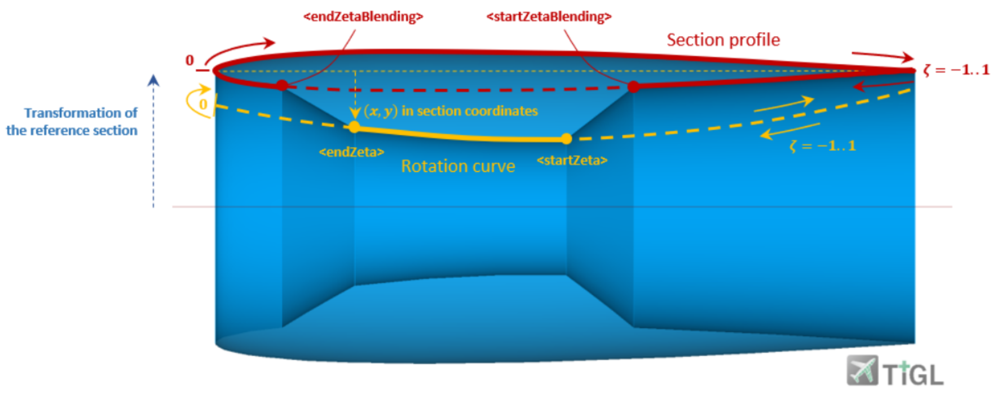


Figure 1.7: rotation curve engine parameters setting

This parametrization is necessary for Fan and Core cowls, while the center cowl is typically defined as a simple body of revolution without additional shaping parameters. Once this part is done, the *Engine* will be called again inside the `<aircraft>` node, where the user must specify the position of the engine with respect to the aircraft reference frame and its orientation. The engine can be positioned using translation parameters along the x, y, and z axes, and oriented through rotation angles around these axes. In Fig. 1.8 there is as same did for wing and fuselage a representation of the engine definition in CPACS, where the tree structure is shown on the top and the corresponding geometry is visualized on the bottom.

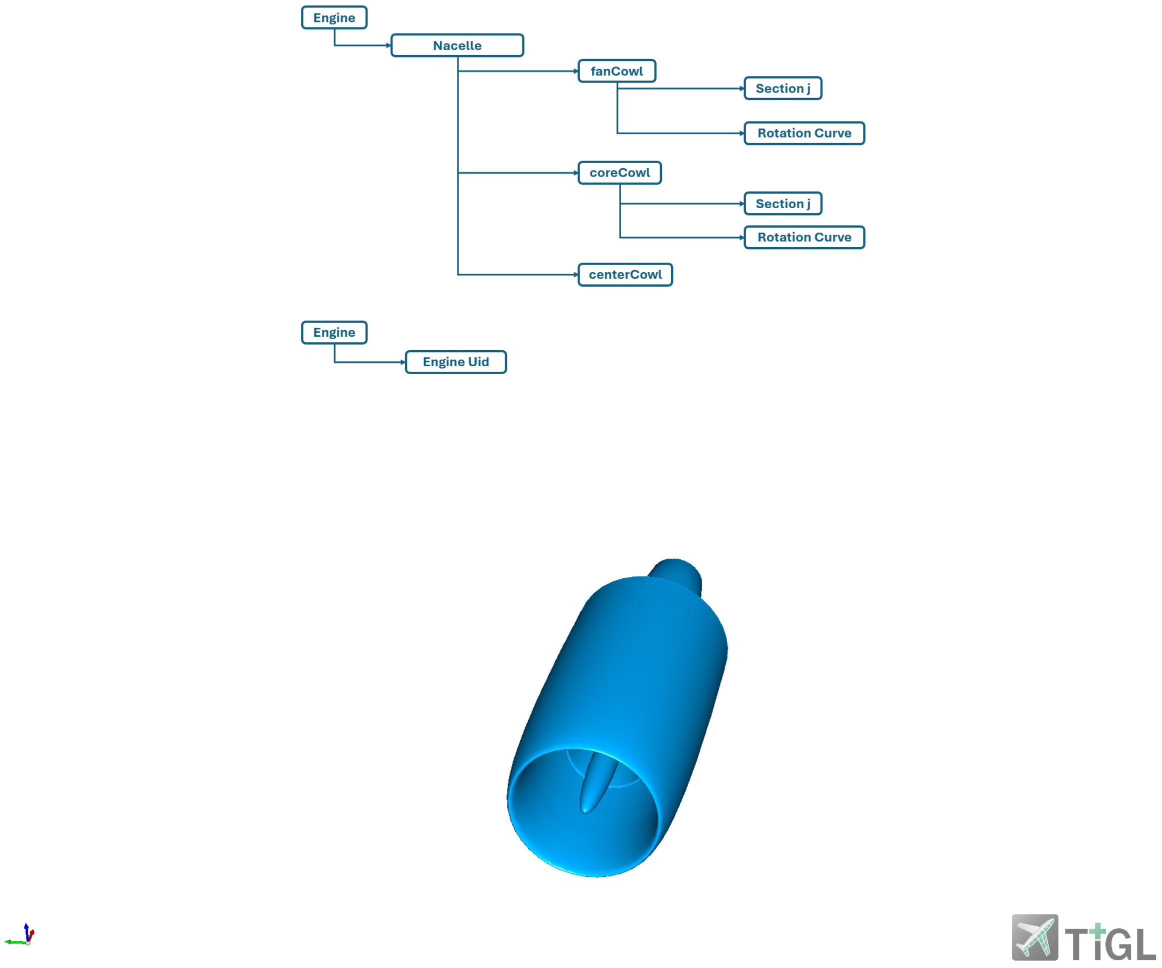


Figure 1.8: Engine definition in CPACS

1.3 CPACS Limitation and OpenVSP Introduction

CPACS presents significant limitations when it comes to geometry creation and modification. Building a new CPACS file with a new geometry means to editing and coding by hand houndreds or thousands of lines with a large number of parameters, this is an extremely time consuming, error prone process that led to think how to overcome these issues. The flexibility of CEASIOMpy from the point of view of a developer, gives the possibility to integrate new module but also external software inside it. In this case, OpenVSP will be employed for geometry creation due to its intuitive and efficient environment for defining aircraft components through high level geometric parameters, making it well suited for rapid design iteration [12]. OpenVSP is a parameteric aircraft geometry tool, originally developed by NASA and it becomes an opensource project since 2012. The advantage of using it is to rapidly create geometric aircraft models without spending much expertise and time in traditional CAD tools. It provides a multitude of basic geometries, common to aircraft modelling, which users modify and assemble to create models. Wings, pods, fuselages, and propellers are available geometries. Advanced components like body of revolution, duct, conformal geometry and such are also available. An illustrative example is presented in

Fig1.9. The upper part of the figure shows a representative aircraft configuration together with the graphical user interface (GUI) panel used to modify its geometric parameters. Through this interface, key aerodynamic and structural features such as sweep angle, dihedral angle, twist distribution, and other characteristic geometric properties can be interactively adjusted. This highlights the parametric flexibility of the modeling environment and its capability to rapidly generate and modify aircraft configurations. The lower part of the figure presents three representative geometries developed within the same framework: from left to right, a Piaggio P.180, a supersonic concept aircraft and a simplified F-16 model. These examples demonstrate the capability of the tool to generate both conventional and unconventional configurations, ranging from propeller-driven aircraft to high-speed jet concepts. All geometries have been entirely constructed from scratch and will serve as reference test cases in the subsequent chapters.

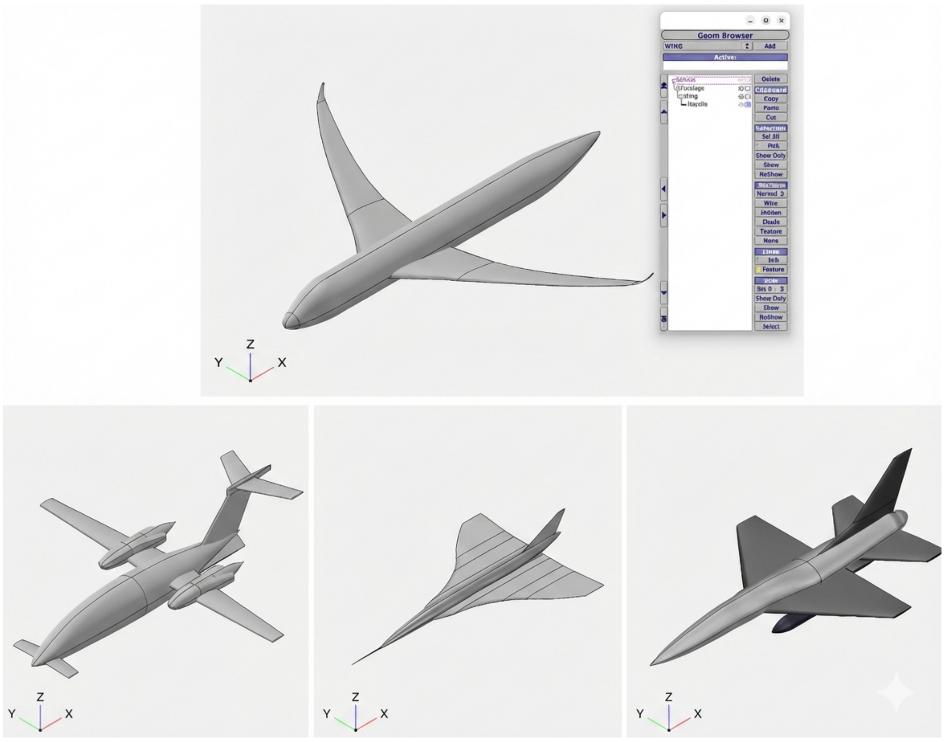


Figure 1.9: OpenVSP test cases

In OpenVSP, aircraft components are generated through a parametric definition based on a limited set of physically meaningful design variables. Take the wing component as an example which represents well how is the modeling philosophy adopted by the software. A wing is defined by specifying global parameters such as total span, reference area and aspect ratio, while local geometric characteristics are controlled through cross-section settings. At each cross-section can be assigned an airfoil profile with independent values of chord length, twist angle, dihedral angle, and thickness-to-chord ratio, allowing for a detailed compact description of the wing geometry. The wing surface is generated by lofting between these cross-sections, ensuring geometric continuity along the span. Additional features such as taper ratio, wing symmetry, and orientation can be introduced directly through the graphical interface, enabling the creation of both conventional and unconventional wing layouts. This is for the wing but works in the same way with different feature for the other components implemented inside the software. OpenVSP can also export the model in various formats from its native file .vsp3 , that is an xml file, to STL/mesh files. The fact is that, in order to use CEASIOMpy the user needs to have a CPACS but OpenVSP doesn't have an export file with the corresponding structure. So, it needs a reliable method for transferring geometry information between the two frameworks and for this reason a dedicated module called VSP2CPACS has been developed with this thesis to bridge the gap between OpenVSP and CPACS.

2. VSP2CPACS

The purpose of this module is to automatically convert OpenVSP components into their corresponding CPACS representations, preserving the geometric structure and key parameters of the aircraft. By transferring the geometry from OpenVSP into CPACS, the model becomes fully compatible with the CEASIOMpy workflow, enabling down-stream analyses such as aerodynamic evaluation and optimization. The VSP2CPACS module thus combines the strengths of both tools: flexibility and parametric control of OpenVSP for geometry generation, and the standardized, multidisciplinary data management capabilities of CPACS. This approach eliminates the need for manual geometry definition in CPACS and significantly reduces the time effort to build a new geometry. The workflow for a CEASIOMpy's user will be, built the geometry in openVSP, after save the file with a specific format and instead of importing a CPACS file inside the CEASIOMpy's GUI, it will be imported the VSP's file. This approach allows users to define and modify geometries without directly interacting with the CPACS file. The module VSP2CPACS takes as input the .vsp3 file and gives as output the .xml file with the CPACS structure. Before the user was able to upload only CPACS file now also file that comes from OpenVSP. The approach fit well on the idea to consider the user experience as first priority because it streamlines the design process and reduces the learning curve for users familiar with OpenVSP. Fig.2.1 summarized the workflow, the path that use OpenVSP could be seen as a a loop for iterative design if the user wants to improve or do some changes in the geometry in a very quickly way.

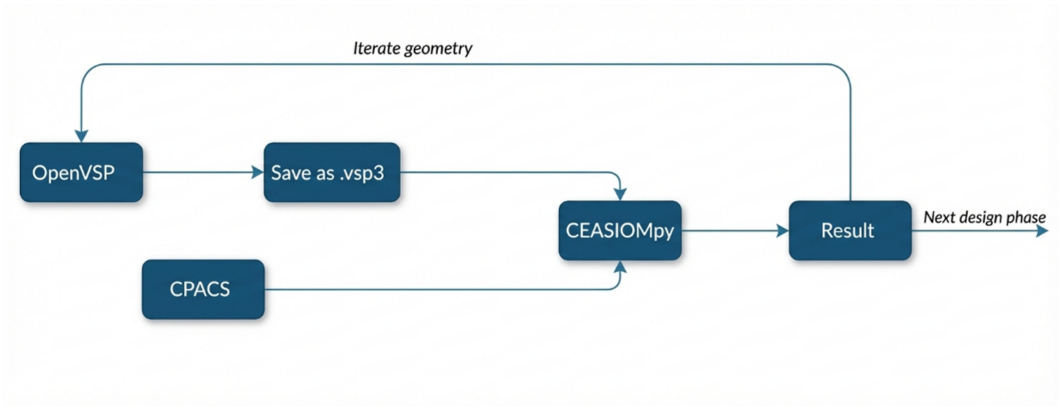


Figure 2.1: User workflow

2.1 Module overview and objectives

The workflow of the VSP2CPACS module is designed to allow the user to define the entire aircraft geometry within OpenVSP. Once the geometry is completed, the model is saved using the native .vsp3 format, which contains the full description of the OpenVSP project. This specific file format, is an XML-based file characterized by a data structure that is specific to OpenVSP and fundamentally different from the CPACS schema. Although the file contains all the information required to reconstruct the geometry, it also includes a large amount of redundant data with OpenVSP specific parameters that are not relevant for the generation of a CPACS model. This file is typically very large, often containing thousands of lines. It stores comprehensive information about the OpenVSP's geometry from GUI settings, individual section parameters to low-level internal variables. Rather than directly interpreting and parsing the XML structure of the .vsp3 file, the VSP2CPACS module relies on the OpenVSP Application Programming Interface (API) to extract the required geometric and parametric information. By using the API, the module accesses the internal, already-processed representation of the geometry, ensuring robustness, consistency, and independence from potential changes in the file format. The main task of the VSP2CPACS module is therefore to query the OpenVSP API to identify and retrieve only the information necessary to define the corresponding CPACS geometry. In particular, the focus is on preserving the geometric structure of the aircraft components,such as wings, fuselages, and control surfaces, while maintaining the same sectional definitions used in OpenVSP.Once the relevant parameters are extracted through the API, they are mapped into the CPACS data structure in a consistent and standardized manner, ensuring full compatibility with the CEASIOMpy workflow. To use the OpenVSP API from an external Python environment, it is necessary to specify the location of the OpenVSP Python bindings. This is done by adding the directory containing the OpenVSP API to the system path, allowing Python to locate and import the OpenVSP module. In practice, setting the EXPORTPATH (or modifying the Python environment path) informs the operating system and the Python interpreter where the OpenVSP API is installed. Without this step, the OpenVSP functions cannot be accessed. For a standard OpenVSP installation, the Python API is in the following directory: . . . /OpenVSP/python/openvsp. The module is built with this structure that will be explained component per component:

```
VSP2CPACS
|_______ vsp2cpacs.py
|_______ __init__.py
|_______ README.md
|_______ func
          |_______ wing.py
          |_______ fuselage.py
          |_______ duct.py
          |_______ pod.py
          |_______ exportcpacs.py
```

Currently, the translation for wing, fuselage, pod and duct components from OpenVSP to CPACS is implemented. The main entry point is the file `vsp2cpacs.py`, which controls the overall conversion workflow. This file receives as input the path to the OpenVSP (`.vsp3`) file generated by the user. For each detected component, the corresponding geometry extraction routine is called. After all relevant geometrical parameters have been collected, the data are passed to the CPACS export routine, which generates the final CPACS output file. The component-specific extraction routines are located in the `func/` directory. Each file within this directory is dedicated to a particular aircraft component. The final stage of the conversion process is handled by `exportcpacs.py`. This file assembles all previously extracted geometrical data into the appropriate CPACS XML structure and writes the resulting configuration to an output file. The export routine ensures that the generated file complies with the structural requirements of the CPACS schema. The file `__init__.py` defines the directory as a Python package and contains module-level configuration parameters.

2.2 Wing component translation

In OpenVSP, the wing component is a parametric geometry element used to define lifting surfaces such as wings and empennage surfaces. The wing geometry is constructed using a section-based approach where each of them represents a spanwise station of the wing and defines the local airfoil shape. The wing component is defined within a local reference system and can be positioned within the global aircraft coordinate system through translation and rotation parameters. The geometric definition of the wing includes both section and global parameters that control the overall planform. The span, chord distribution, taper, sweep, dihedral, and twist are defined either directly through spanwise section positioning or implicitly through parametric relationships between sections. At each wing section a specific airfoil can be associated, which may be selected from predefined airfoil families or imported as a custom profile. Depending on the airfoil representation, additional parameters such as thickness and camber distributions can be controlled. The interpolation between adjacent sections ensures a smooth variation of these properties along the span, resulting in a continuous three-dimensional surface. OpenVSP also provides functionality for applying symmetry conditions, enabling the automatic generation of mirrored wing with respect to a reference plane. The transformation parameters like rotation and translation associated with the wing component play a crucial role in defining the final position and orientation of the wing relative to the aircraft reference frame. Inside wing.py the macro parameters are extracted using this logic; before going into the section, it extracts the global information like symmetry and orientation in space and after iterates in all the sections defined in OpenVSP.

2.2.1 Wing Parameter Translation from OpenVSP to CPACS

The translation of the wing geometry from OpenVSP to CPACS requires a consistent mapping of every geometric parameters. Since both tools rely on a section-based representation, the conversion process focuses on transferring chord, sweep, and spanwise positioning parameters in a coherent and geometrically equivalent manner. So, all the sections defined in OpenVSP will correspond into the CPACS sections one by one. The following list shows the implemented parameters inside VSP2CPACS, where the user can set and being able to preserve the same geometry through the module:

- **Number of Points**: The number of points used to define the airfoil and, consequently, the entire geometry is controlled by the parameter *Tess W*. When VSP2CPACS generates the CPACS from OpenVSP, it is important for the user to note that using the feature EDIT CURVE or importing an external airfoil will *not* alter the number of points. This behavior arises from the data extraction procedure implemented through the OpenVSP API, where the code accesses only the control points or the geometry file originally imported by the user. But for all the other airfoil implemented is able to set the number of points that the user wants.

- **Orientation**: In OpenVSP, a wing can be positioned and oriented in space using the transformation parameters *XLoc*, *YLoc*, *ZLoc* (translations) and *XRot*, *YRot*, *ZRot* (rotations). These parameters are defined in the *Transformation* CPACS and are applied before the cross-section definitions. OpenVSP allows components to be defined either in a global reference system or relative to another component through the *XForm* functionality. In VSP2CPACS, all components are instead exported using a global reference system. This choice was made to generalize the orientation handling and to avoid implementing redundant feature.So, VSP2CPACS computes and assigns the final global position and orientation of each component directly. Doing it, all the OpenVSP's feature inside *XForm - Attach To Parent* are availble because thay change the global orientation where VSP2CPACS has access.

- **Symmetry**: Symmetry is a feature where selecting in *Planar* inside the OpenVSP's GUI, the user is able to mirror the object on the selected plane. This part is written in the CPACS inside on the *Transformation* part before writing the sections.

- **Airfoil**: Each wing or fuselage section in OpenVSP is associated with an airfoil, offering complete freedom in profile selection. The implementation developed in this work focuses on the most common airfoil. (see Tab. 2.1). By default, airfoils such as NACA Four-Digit and Five-Digit (including modified versions) are analytically reconstructed from their defining parameters, ensuring geometric consistency also with the freedom to define how many points the user wants. It is also implemented the OpenVSP's flag to invert the airfoil.

- **Chord**: In OpenVSP, each section defines a local chord value at a given spanwise station. The wing section setting provides the root chord that correspond to the local and tip chord that is referred to the next section. In CPACS, the chord is implemented within the section scaling part. A normalized airfoil profile is associated with each section and subsequently scaled in the x- and z-directions. In this way, the geometric dimensions defined in OpenVSP are preserved.

- **Span**: In OpenVSP, the span of a wing section is defined as the perpendicular distance between consecutive sections. In CPACS, a corresponding parameter exists under the `Positioning` element as `Length`, but the two definitions differ. Specifically, in CPACS, `Length` represents the distance along the segment connecting two sections. If the sweep angle is zero, both definitions coincide. However, when a non-zero sweep angle is applied, the CPACS segment is longer than the perpendicular span. To ensure consistency when transferring geometry from OpenVSP to CPACS, the span must be corrected by dividing by the cosine of the sweep angle:

$$\texttt{Length} = \frac{\texttt{Span}}{\cos(\texttt{Sweep\_angle})}.$$

- **Sweep**: In OpenVSP, the sweep angle is defined for each wing section and geometrically represents the inclination of the section reference line with respect to the spanwise direction. In CPACS, the sweep angle is implemented within the section positioning part, where a dedicated parameter specifies the sweep with respect to the global reference system. During the translation process, the sweep angle is therefore mapped directly into the CPACS section transformation parameters.

- **Sweep Location**: A key difference between OpenVSP and CPACS concerns the reference line used for defining the sweep angle. In CPACS, the sweep is always defined with respect to the leading edge (LE). In contrast, OpenVSP allows the user to define the sweep relative to a generic chordwise location, referred to as the *sweep location*, which ranges from 0 (leading edge) to 1 (trailing edge). Therefore, when the sweep location is different from zero, a geometric correction is required to convert the sweep angle into an equivalent leading edge sweep before transferring it to CPACS. This correction is implemented in the VSP2CPACS module by accounting for the variation of chord along the span. Let c_{root} and c_{tip} be the root and tip chord values of the considered wing segment, and let b be the span of the segment. The chord gradient along the span is defined as

$$\frac{dc}{dy} = \frac{c_{\text{tip}} - c_{\text{root}}}{b}.$$

  If Λ is the sweep angle defined in OpenVSP at a sweep location η (with $\eta \in [0, 1]$), the equivalent leading-edge sweep angle Λ_{LE} is computed as

$$\Lambda_{LE} = \arctan\left(\tan\Lambda - \eta\frac{dc}{dy}\right).$$

  This formulation corresponds to the implementation adopted in the VSP2CPACS code, where the correction term $\eta\frac{dc}{dy}$ accounts for the chordwise shift of the reference line from the specified sweep location to the leading edge. The corrected sweep angle

Λ_{LE} is then converted to degrees and assigned to the corresponding CPACS section positioning parameter. In this way, the geometric equivalence between the OpenVSP and CPACS wing representations is preserved independently of the selected sweep reference location.

- **Dihedral**: The dihedral works in the same way of the sweep, so will be inside the positioning part of the CPACS.
- **Twist and Twist Location**: In OpenVSP, the twist angle is defined at each wing section and represents a rotation of the airfoil profile around a specified chordwise location, referred to as the *twist location*. This location is a nondimensional parameter expressed in the interval $[0, 1]$, where 0 corresponds to the leading edge and 1 to the trailing edge. Unlike CPACS, where twist is typically applied with a y - rotation of the section over the LE, OpenVSP allows the rotation to be performed about an arbitrary point along the chord. The implementation works to upload a normalized but already twisted profile inside the $< WingAirfoil >$ part of the CPACS. It means that the series of points will own inside the applied twist. Let θ be the twist angle extracted from OpenVSP, the rotation is applied in the two-dimensional airfoil using a standard rotation matrix. Considering a clockwise-positive convention, the rotation angle in radians is defined as θ_r The airfoil coordinates $\mathbf{X} = (x, y)^T$ are rotated about the twist location η by first shifting the origin to the rotation point,

  $$\mathbf{X}_s = \mathbf{X} - \begin{pmatrix} \eta \\ 0 \end{pmatrix},$$

  then applying the rotation matrix

  $$\mathbf{R} = \begin{pmatrix} \cos\theta_r & -\sin\theta_r \\ \sin\theta_r & \cos\theta_r \end{pmatrix},$$

  and finally shifting the coordinates back to the original reference system,

  $$\mathbf{X}_{\text{twisted}} = \mathbf{R}\mathbf{X}_s + \begin{pmatrix} \eta \\ 0 \end{pmatrix}.$$

  This implementation ensures that the twist is geometrically equivalent to the definition provided in OpenVSP.
- **LE/TE Tessellation**: The tessellation feature refines the discretization of the airfoil profile by redistributing chordwise points in the leading-edge (LE) and trailing-edge (TE) regions. The refinement intensity is governed by the OpenVSP parameters `LECluster` and `TECluster`, retrieved via the API and constrained within the interval $[1, 9]$ to ensure numerical robustness.For applicable airfoil section types, the algorithm first identifies the leading edge and splits the profile into upper and lower surfaces. A nondimensional chordwise coordinate is then defined in the interval $[0, 1]$, and new point distributions are generated such that the local density near the LE and TE increases proportionally to the tessellation parameters. In practice, lower parameter values result in finer discretization in the LE/TE zones relative to the mid-chord region.Each

surface is subsequently interpolated using cubic interpolation, enabling evaluation of the corresponding y-coordinates at the newly defined x-positions. The refined airfoil is reconstructed by concatenating the mirrored lower and upper surfaces, ensuring the profile remains closed and geometrically continuous. This procedure roughly doubles the total number of points compared to the default OpenVSP discretization. Since `CPACS` interpolation routines are highly sensitive to both the number and spatial distribution of airfoil points, maintaining a smooth and uniform chordwise sampling is essential to avoid numerical oscillations or loss of geometric fidelity.In practical terms, setting both `LECluster` and `TECluster` around 3 yields a balanced distribution, producing a denser clustering from the leading and trailing edges up to approximately 30% of the chord, while preserving moderate spacing in the mid-chord region. This approach improves curvature resolution where necessary without generating excessive refinement that could compromise CPACS compatibility.

The remaining features available in OpenVSP are not yet implemented in the present framework; however, this work represents a significant step forward within the context of preliminary design. Furthermore, the translation between OpenVSP and CPACS is based on a sectional approach, which may lead to minor geometric discrepancies between the original OpenVSP model and its corresponding CPACS representation, primarily due to differences in the interpolation methods used between adjacent sections.

2.2.2 Wing Airfoils

For every section, an airfoil must be defined. OpenVSP allows complete design freedom in this choice. The available profile options in OpenVSP are the same for both wing and fuselage components. Table 2.1 presents the mapping between the airfoil definitions available in the OpenVSP GUI (left column) and the corresponding Python implementations within VSP2CPACS (right column). As will be discussed later, airfoils that are not analytically well defined are not directly implemented. However, by using the `get_coord_edit_curve` function, any airfoil can be accessed by activating the `CONVERT_CEDIT` option.

Table 2.1: Mapping of OpenVSP airfoil definitions to VSP2CPACS implementations

OpenVSP GUI		**Python Implementation**
POINT	→	`get_coord_point`
CIRCLE	→	`get_coord_circle`
ELLIPSE	→	`get_coord_ellipse`
SUPER_ELLIPSE	→	`get_coord_superellipse`
ROUNDED_RECTANGLE	→	`get_coord_roundedrectangle`
FOUR_SERIES	→	`get_coord_naca4`
EDIT_CURVE	→	`get_coord_edit_curve`
AF_FILE	→	`get_coord_from_file`
FOUR_DIGIT_MOD	→	`get_coord_naca4_mod`
FIVE_DIGIT	→	`get_coord_naca5`
FIVE_DIGIT_MOD	→	`get_coord_naca5_mod`
GENERAL_FUSE	→	Not implemented
FUSE_FILE	→	Not implemented
SIX_SERIES	→	Not implemented
BICONVEX	→	Not implemented
WEDGE	→	Not implemented
CST_AIRFOIL	→	Not implemented
KARMAN_TREFFTZ	→	Not implemented
16_SERIES	→	Not implemented
AC25_773	→	Not implemented

The implementation focuses on the most commonly used airfoil definitions. In particular, each section profile is generated by default starting from a zero-based definition, meaning that when the user specifies a NACA four-digit airfoil within OpenVSP, the corresponding airfoil geometry is reconstructed using its defining parameters and generated through the analytical formulation of the airfoil. Although OpenVSP allows the user to select airfoil profiles without constraints, the following chapters describe, with specific reference to the wing component, the airfoil types considered in this chapter. These include: NACA Four-Digit, NACA Five-Digit, Four-Digit Modified, Five-Digit Modified, Edit Curve, and profiles imported from .af files. The others like Circle, Ellipse are more used for fuselage profile and they will be discussed in the following chapter. For analytically defined NACA airfoils (four-digit and five-digit, including their modified versions), the geometric coordinates are computed directly from the corresponding mathematical equations, ensuring consistency and repeatability of the generated profiles. In the case of non-analytical profiles (Edit Curve and .af files), the airfoil geometry is instead obtained from the discretized coordinate data provided by the user. When exporting the wing airfoil profiles to CPACS, specific and mandatory ordering of the profile points is adopted. In accordance with the CPACS standard, the airfoil contour is defined starting from the trailing edge (TE) on the lower surface, proceeding forward along the lower surface up to the leading edge (LE), and then continuing along the upper surface back to the trailing edge, thereby closing the profile. Every function takes as input the section uID, which is used to retrieve the corresponding sectional parameters. In general, the user should be aware that these routines in very extreme or highly skewed parameter combinations may become less robusts and should be checked carefully.

2.2.3 NACA Four digit and NACA Four digit Modified

The generation of wing section profiles in the software relies on two functions: `get_coord_naca4` for standard NACA 4-digit airfoils and `get_coord_naca4_mod` for modified 4-digit profiles. Both functions compute the x and y coordinates of the airfoil and produce a structured representation compatible with CPACS. The Four-Digit formula for a symmetrical airfoil is written in (2.1):

$$y_t = 5t\left[0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1015x^4\right] \tag{2.1}$$

where:

- x = position along the chord from 0 to 1.00,
- y_t = half thickness at a given value of x,
- t = maximum thickness as a fraction of the chord (gives the last two digits in the NACA 4-digit denomination divided by 100),
- $y_u = +y_t$ = coordinates of the upper airfoil surface,
- $y_l = -y_t$ = coordinates of the lower airfoil surface.

Instead, the equation for asymmetric foils is slightly different because it uses the camber line (2.2):

$$y_c = \begin{cases} \frac{m}{p^2}(2px - x^2), & 0 \le x \le p \\ \frac{m}{(1-p)^2}((1-2p) + 2px - x^2), & p < x \le 1 \end{cases} \tag{2.2}$$

$$\theta = \tan^{-1}\left(\frac{dy_c}{dx}\right) \tag{2.3}$$

and the coordinates of the upper and lower surfaces are computed as:

$$x_u = x - y_t \sin(\theta) \tag{2.4}$$

$$y_u = y_c + y_t \cos(\theta) \tag{2.5}$$

$$x_l = x + y_t \sin(\theta) \tag{2.6}$$

$$y_l = y_c - y_t \cos(\theta) \tag{2.7}$$

where:

- m = maximum camber (first digit divided by 100),
- p = location of maximum camber (second digit divided by 10),
- y_u, y_l = coordinates of upper and lower airfoil surfaces, respectively.

The process begins with the extraction of geometric parameters from the OpenVSP section. For the standard NACA 4-digit profile, the function retrieves the maximum camber (m), the location of maximum camber along the chord (p), the maximum thickness (t), the chord length (c), and an optional inversion flag to flip the airfoil vertically. After a cosine-spaced distribution to define the x coordinates along the airfoil, the shape is built. For the Four-Digit Modified profile, the process is slightly adjusted to compute the coordinates:

1. Pick values of x from 0 to the maximum chord c.

2. Compute the mean camber line coordinates using the equations for the Four- or Five-Digit Series.

3. Calculate the thickness distribution above (+) and below (–) the mean line using these equations. The a_x and d_x coefficients are determined from a reference table (derived for a 20% thick airfoil).

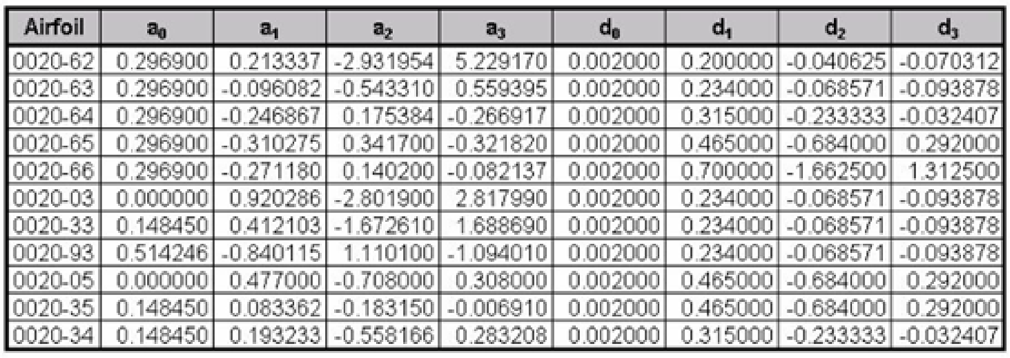
   Ahead of $t_{\max}$:
   $$\pm y_t = a_0\sqrt{x} + a_1 x + a_2 x^2 + a_3 x^3$$

   Aft of $t_{\max}$:
   $$\pm y_t = d_0 + d_1(1-x) + d_2(1-x)^2 + d_3(1-x)^3$$

4. Determine the final coordinates using the same equations as the Four-Digit Series.

5. Scale the airfoil by multiplying the final y coordinates by $\left[\frac{t}{0.2}\right]$ to obtain the desired thickness.

The coefficients used come from the reference table shown in Fig. 2.2.



Airfoil	a_0	a_1	a_2	a_3	d_0	d_1	d_2	d_3
0020-62	0.296900	0.213337	-2.931954	5.229170	0.002000	0.200000	-0.040625	-0.070312
0020-63	0.296900	-0.096082	-0.543310	0.559395	0.002000	0.234000	-0.068571	-0.093878
0020-64	0.296900	-0.246867	0.175384	-0.266917	0.002000	0.315000	-0.233333	-0.032407
0020-65	0.296900	-0.310275	0.341700	-0.321820	0.002000	0.465000	-0.684000	0.292000
0020-66	0.296900	-0.271180	0.140200	-0.082137	0.002000	0.700000	-1.662500	1.312500
0020-03	0.000000	0.920286	-2.801900	2.817990	0.002000	0.234000	-0.068571	-0.093878
0020-33	0.148450	0.412103	-1.672610	1.688690	0.002000	0.234000	-0.068571	-0.093878
0020-93	0.514246	-0.840115	1.110100	-1.094010	0.002000	0.234000	-0.068571	-0.093878
0020-05	0.000000	0.477000	-0.708000	0.308000	0.002000	0.465000	-0.684000	0.292000
0020-35	0.148450	0.083362	-0.183150	-0.006910	0.002000	0.465000	-0.684000	0.292000
0020-34	0.148450	0.193233	-0.558166	0.283208	0.002000	0.315000	-0.233333	-0.032407

Figure 2.2: Coefficients a_x and d_x for the modified NACA 4-digit airfoil.

So, the modified function keeps the standard 4-digit camber definition, but replaces the classical thickness law with a piecewise cubic polynomial whose coefficients are interpolated from a reference table parameterized by leading-edge radius index and thickness-location index.When moving far from the tabulated modified airfoils, the interpolation of the coefficients becomes unreliable,at the edges of the database. In that regime, the piecewise cubic thickness polynomials can produce nonphysical shapes. Consequently, the intended relationship between leading-edge radius, thickness location, and overall profile is no longer guaranteed, and the fixed cosine discretization may fail to resolve sharp or highly shifted features. Therefore, the numerical representation can deviate from the “true” modified family, and the resulting airfoil should be checked or refined before aerodynamic use. In these extreme configurations, a small step may appear at $t_{\max}$ or it follows a lower curvature than the "real" one. To demonstrate the capabilities in Fig. 2.3 is shown a wing with two sections, the root one with a NACA 6410 and the tip one with a NACA 3510-64. The figure shows the comparison between the geometry generated by OpenVSP and the corresponding CPACS representation obtained through VSP2CPACS. The two geometries are visually identical, confirming that the translation process preserves the geometric structure and key parameters of the wing as defined in OpenVSP.

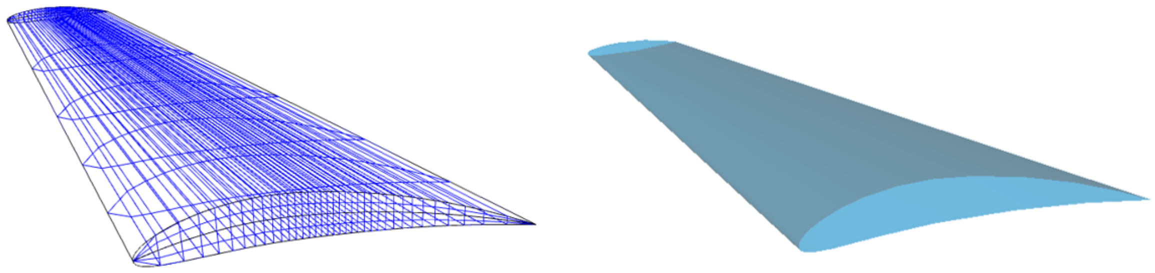


Figure 2.3: Comparison between OpenVSP and CPACS

2.2.4 NACA Five Digit and NACA Five Digit Modified

The generation of wing section profiles relies on two functions: `get_coord_naca5` for standard NACA 5-digit airfoils and `get_coord_naca5_mod` for modified 5-digit profiles. Both functions compute the x and y coordinates of the airfoil and produce a structured representation compatible with CPACS. NACA 5-digit airfoils follow the naming convention LPSTT, where L represents the theoretical optimal lift coefficient $C_{LI} = 0.15L$, P indicates the maximum camber location at $x = 0.05P$, $S = 0$ denotes simple camber, and TT gives maximum thickness as percentage of chord. The implemented function `get_coord_naca5` uses the standard camber line formulation (S=0):

$$y_c = \begin{cases} \frac{k_1}{6}\left(x^3 - 3rx^2 + r^2(3-r)x\right), & 0 \le x < r \\ \frac{k_1 r^3}{6}(1-x), & r \le x \le 1 \end{cases} \tag{2.8}$$

$$\frac{dy_c}{dx} = \begin{cases} \frac{k_1}{6}\left(3x^2 - 6rx + r^2(3-r)\right), & 0 \le x < r \\ -\frac{k_1 r^3}{6}, & r \le x \le 1 \end{cases} \tag{2.9}$$

where r is the break point ensuring maximum camber occurs at $x = p$, and k_1 is interpolated from standard NACA tables (210, 220, 230, 240, 250 profiles) and scaled by $C_{LI}/0.3$. The thickness distribution uses the classical 4-digit formulation, scaled for the desired thickness t:

$$y_t = \frac{t}{0.2}\left[0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1036x^4\right] \tag{2.10}$$

Chordwise points are generated using cosine spacing $x = 0.5(1 - \cos\theta)$ for $\theta \in [0, \pi]$. The upper/lower surface coordinates follow the standard rotation:

$$\theta = \tan^{-1}\left(\frac{dy_c}{dx}\right) \tag{2.11}$$

$$x_u = x - y_t \sin\theta, \quad y_u = y_c + y_t \cos\theta \tag{2.12}$$

$$x_l = x + y_t \sin\theta, \quad y_l = y_c - y_t \cos\theta \tag{2.13}$$

The final airfoil is assembled by concatenating the reversed lower surface with the upper surface, closing at the trailing edge. Optional inversion (via `Invert` flag)is applied. The `get_coord_naca5_mod` function extends the standard 5-digit camber line with a modified thickness distribution controlled by `LERadIndx` (leading edge radius index) and `ThickLoc` (maximum thickness location). The airfoil name follows the format `NACA LPSTT-mm` where `mm` encodes these modification indices. The camber line computation remains identical to the standard 5-digit case. The key difference is the piecewise cubic thickness distribution:

$$\pm y_t(x) = a_0\sqrt{x} + a_1 x + a_2 x^2 + a_3 x^3, \qquad x \leq \texttt{ThickLoc} \tag{2.14}$$

$$\pm y_t(x) = d_0 + d_1(1-x) + d_2(1-x)^2 + d_3(1-x)^3, \qquad x > \texttt{ThickLoc} \tag{2.15}$$

The coefficients a_i and d_i are interpolated from a predefined reference table of 10 base airfoils, see Fig. 2.2, originally derived for 20% thick airfoils. The computational procedure follows these steps that are almost the same for the previous Four Digit Modified:

1. Generate cosine-distributed chordwise points $x \in [0, 1]$.
2. Compute camber line $y_c(x)$ and slope dy_c/dx using standard 5-digit equations.
3. Interpolate thickness coefficients a_i, d_i from the reference table using modification index `mm`.
4. Construct piecewise thickness distribution $y_t(x)$ before/after maximum thickness location.
5. Calculate upper/lower surface coordinates via standard rotation equations.
6. Apply airfoil inversion (if required).

This formulation provides enhanced control over leading-edge radius and thickness location. However, as for the Four Digit MoD, interpolation accuracy degrades near table boundaries (extreme `LERadIndx` or `ThickLoc` values), potentially producing non-physical thickness distributions or curvature discontinuities at $x_{t,\max}$. In Fig. 2.4, a similar example to the previous one is shown, where the root section is defined as a NACA 46104 and the tip section as a NACA 72507.

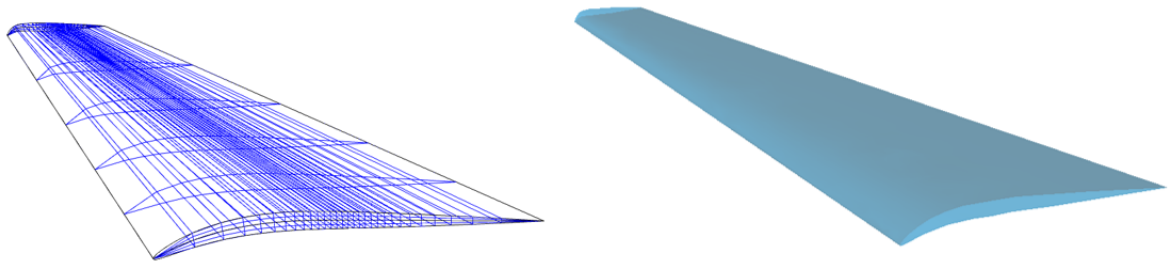


Figure 2.4: Comparison between OpenVSP and CPACS

2.2.5 Edit curve feature

The *Edit Curve* option in OpenVSP allows the user to define an airfoil profile through a set of editable control points rather than through an analytical formulation. Unlike NACA series airfoils, which are reconstructed from closed-form equations, the Edit Curve geometry must be rebuilt from the discrete control points stored in the OpenVSP model. The reconstruction procedure is implemented in the function `get_coord_edit_curve`. This function retrieves the control points associated with the selected cross-section and reconstructs the airfoil geometry according to the curve type specified in OpenVSP. For a given section uID, the function first extracts the geometric scaling parameters:

- `Width`, corresponding to the chordwise dimension,
- `Height`, corresponding to the thickness scaling.

The control points are then obtained using the OpenVSp's API where the coordinates are extracted. Since OpenVSP does not necessarily define the leading edge as the origin of the local coordinate system, a shift must be done to have the leading edge that lies at $x = 0$. OpenVSP allows three different interpolation schemes for the Edit Curve definition:

- Linear interpolation,
- Cubic spline interpolation,
- Cubic Bézier interpolation.

The selected type is retrieved from the OpenVSP parameter `CurveType`, and the corresponding reconstruction routine is called:

- `linear_curve`,
- `spline_curve`,
- `bezier_curve`.

The function `linear_curve` reconstructs the airfoil contour through linear interpolation between consecutive control points. This approach represents the simplest geometric reconstruction strategy and produces a C^0-continuous curve, i.e., continuous in position but not in slope. Given a set of control points $\{\mathbf{p}_i\}_{i=0}^{N}$, each segment between $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ is discretized using a uniform linear interpolation:

$$\mathbf{L}(t) = (1-t)\mathbf{p}_i + t\mathbf{p}_{i+1}, \quad t \in [0,1].$$

The total number of discretization points is distributed approximately uniformly among the segments in order to preserve geometric resolution along the contour. The concatenation of all interpolated segments produces a piecewise linear approximation of the airfoil geometry that strictly follows the control polygon. After the airfoil is re-orderd to follow the CPACS convenction. While computationally inexpensive and geometrically faithful to the control polygon, the linear reconstruction does not guarantee slope continuity at segment junctions. This is something that the user should know regardless. The function `spline_curve` reconstructs a smooth airfoil contour from a discrete set of control points using a centripetal Catmull–Rom cubic spline formulation. Although OpenVSP allows an "arbitrary" number of control points(It depends about what the user selects Circle, Ellipse or Rectangle) to define the airfoil curve, each cubic spline segment is locally constructed using four consecutive control points. The resulting spline is interpolating and ensures smooth first-derivative continuity across segments. The objective of the procedure is to generate a C^1-continuous curve. To properly define boundary segments, the first and last control points are duplicated. The spline parameterization follows the centripetal Catmull–Rom formulation, in which the parametric coordinate is defined as

$$t_{i+1} = t_i + \|\mathbf{p}_{i+1} - \mathbf{p}_i\|^{\alpha}, \quad \alpha = 0.5.$$

The choice $\alpha = 0.5$ corresponds to centripetal parameterization, which mitigates overshooting phenomena typical of uniform splines ($\alpha = 0$) and reduces the risk of self-intersections that may arise in chordal parameterizations ($\alpha = 1$). For each sequence of four consecutive control points $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$, a cubic interpolation segment is constructed over the interval $t \in [t_1, t_2]$. For a given segment defined by four consecutive control points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, the algorithm computes the curve as follows:

- Linear blending (first level): Intermediate points $\mathbf{A}_k(t)$ are computed by linearly interpolating between consecutive points:

  $$\mathbf{A}_1(t) = (1-\tau)\mathbf{p}_0 + \tau\mathbf{p}_1, \quad \mathbf{A}_2(t) = (1-\tau)\mathbf{p}_1 + \tau\mathbf{p}_2, \quad \mathbf{A}_3(t) = (1-\tau)\mathbf{p}_2 + \tau\mathbf{p}_3,$$

  where $\tau \in [0,1]$ is the normalized local parameter along the segment. This step produces points along the edges of the control polygon.

- Quadratic blending (second level): The first-level points are then linearly combined to yield quadratic intermediate points:

  $$\mathbf{B}_1(t) = (1-\tau)\mathbf{A}_1 + \tau\mathbf{A}_2, \quad \mathbf{B}_2(t) = (1-\tau)\mathbf{A}_2 + \tau\mathbf{A}_3.$$

  This step implicitly introduces curvature, forming a C^1-continuous quadratic approximation that starts to respect the desired smoothness between control points.

- Cubic blending (third level): Finally, the quadratic points are blended to produce the cubic spline point on the curve:

  $$\mathbf{C}(t) = (1-\tau)\mathbf{B}_1 + \tau\mathbf{B}_2.$$

  This recursive application of linear interpolations is mathematically equivalent to the classical cubic Hermite formulation, where tangents at $\mathbf{p}_1$ and $\mathbf{p}_2$ are implicitly defined by the neighboring points $\mathbf{p}_0$ and $\mathbf{p}_3$. As a result, the curve is inherently C^1-continuous across segment.

The concatenation of all evaluated segments produces the full airfoil curve. Overall, the implemented centripetal Catmull–Rom spline reconstruction ensures a smooth representation of the Edit Curve airfoil. Compared to linear interpolation, this approach guarantees curvature continuity, while avoiding the oscillatory behavior that may arise from higher-order global interpolation schemes. In the Bézier case, the control points are interpreted in groups of four, each set defining one cubic Bézier segment. A cubic Bézier curve is a parametric curve described by the following equation in Bernstein polynomial form:

$$B(t) = \sum_{i=0}^{3} \binom{3}{i} (1-t)^{3-i} t^i P_i, \qquad t \in [0,1],$$

where $P_i = (x_i, y_i)$ are the control points of the segment. Each $\binom{3}{i}$ term is a binomial coefficient, and the corresponding factors $(1-t)^{3-i}t^i$ are Bernstein basis polynomials of degree three. The curve always starts at P_0 and ends at P_3. The intermediate control points P_1 and P_2 determine the tangent directions at these endpoints:

$$B'(0) = 3(P_1 - P_0), \qquad B'(1) = 3(P_3 - P_2).$$

Thus, altering these intermediate points modifies the tangency and curvature of the curve. In practice, several cubic Bézier segments are concatenated to form the entire airfoil contour. If the control points are $P_0, P_1, \ldots, P_{n-1}$, they are grouped by fours as:

$$(P_0, P_1, P_2, P_3),\ (P_3, P_4, P_5, P_6),\ \ldots$$

Each group defines a local cubic Bézier segment, and adjacent segments share a common endpoint to ensure continuity. The resulting ordered curve defines the final airfoil contour. The Edit Curve implementation guarantees that the reconstructed geometry in CPACS matches the OpenVSP definition in a consistent and reproducible manner. However, the final representation depends on the discretization density chosen for curve evaluation. For highly irregular control-point distributions or extreme geometric configurations, the smoothness and numerical robustness of the reconstructed profile may depend on the selected interpolation method. Among the three procedures; linear curves are simple but produce only C0 continuous profiles, while cubic splines provide smooth C1 continuity through all control points. Cubic Bézier curves offer the greatest flexibility and robustness, taking full advantage of the additional control points provided by OpenVSP. For aerodynamic applications, spline and Bézier interpolation are recommended with Ellipse or Circle initial shape, whereas linear curves may be used only for preliminary or coarse approximations. Furthermore, Edit Curve supports the generation of airfoils that are not analytically defined or pre-implemented within

VSP2CPACS module. When a user selects an existing airfoil and activates the *Convert GEDIT* tool in OpenVSP, the program automatically generates a set of control points that reproduces the selected profile. Compared to analytically defined NACA airfoils, the *Edit Curve* approach offers significantly greater geometric flexibility. For this reason, careful verification of the resulting geometry is recommended when exporting highly customized or non-standard profiles. To show the capabilities, in openVSP when you want to customize a new airfoil using Edit Curve the user can start with a initial shape from these one: Circle, Ellipse, Rectangle which guaranties different number of control points. Take the circle as an example and move the highest points to have an configuration airfoil-like as is shown in Fig 2.5;it has 4 control points for Linear and Spline setting and 8 control points for Bezier. Remember that this procedure wants to obtain the same shape on both the software and inside CPACSCreator, the CPACS visualization tool, the series of points that shows are interpolated with a B-Spline.So, although the user set a custum profile needs to remember that once the CPACS is obtained, CPACSCreator when visualize the geometry interpolate the points with its own logic.

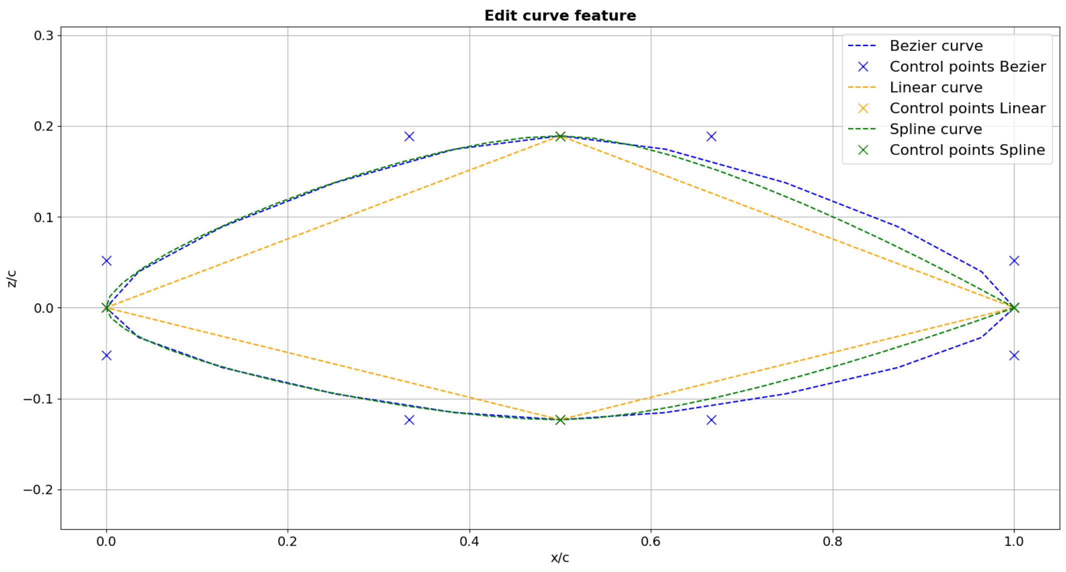


Figure 2.5: Edit curve

2.2.6 Af file

The function `get_coord_from_file` reconstructs an airfoil geometry from an external coordinate file imported into OpenVSP. This approach is typically used when airfoil data are obtained from databases such as the Selig airfoil repository, where upper and lower surface coordinates are explicitly defined.For a given cross-section identifier VSP2CPACS first retrieves the geometric parameters associated with the section, namely the chord length (`Chord`) and implement the features to modify the thickness-chord ration (named T/C inside OpenVSP).These parameters are required to correctly dimensionalize the airfoil profile within the OpenVSP geometry framework. The upper and lower surface coordinates are then extracted separately using the OpenVSP API and each surface is converted into an array.

To construct a closed airfoil contour, the lower surface coordinates are first reversed in order to follow the standard aerodynamic convention from trailing edge to leading edge. This procedure ensures that externally defined airfoils are imported into the VSP2CPACS workflow.In principle, direct access to the upper and lower surface coordinates through the OpenVSP API would allow the import of airfoil files with arbitrary point ordering. However, in practice, OpenVSP probably internall assumes the Selig coordinate convention when parsing `.af` files. If the point ordering does not follow this convention, the airfoil geometry may not be reconstructed correctly because OpenVSP doesn't import the file. This behavior appears to be a limitation within OpenVSP import implementation. For this reason, users are strongly advised to supply airfoil data in standard Selig format to ensure consistent and error-free reconstruction within the OpenVSP–VSP2CPACS workflow.

2.3 Fuse component translation

In OpenVSP, the fuselage component is a parametric geometry element used to define the main body of the aircraft. Unlike the wing component, which is organized primarily along a spanwise direction, the fuselage is constructed along the longitudinal axis of the aircraft. The geometry is generated through a section-based approach in which a sequence of cross-sections is distributed along the axial direction and smoothly lofted to create a continuous three-dimensional surface. Each fuselage section represents a station along the longitudinal axis and defines the local cross-sectional shape. The section shape depends on the selected parameterization. In OpenVSP, the fuselage is typically defined using cross-section parameters such as width, height, area, corner radius, and shape coefficients, allowing flexible control of the body contour. The fuselage component is defined within its own local reference system and can be positioned within the global aircraft coordinate system through translation and rotation parameters. These transformation parameters determine the final placement and orientation of the fuselage relative to the aircraft reference frame. As for the wing component, symmetry conditions may be applied if required. The logic behind the translation into a CPACS component is the same for the wing; export every section,with its parametrization defined in OpenVSP into the correspond CPACS section one by one.Inside `fuselage.py`, the macro parameters are first extracted. These include the global transformation (translation and rotation), symmetry settings, and hierarchical information indicating whether the fuselage is a parent or child component in the overall configuration. Subsequently, the section-level parameters are processed. A loop is implemented from the first section to the last one, extracting the axial position and cross-sectional properties of each station. The relative positioning between consecutive sections is then computed to generate the appropriate CPACS.Actually, only the Monolitic design policy is supported for fuselage modelling.

2.3.1 Fuselage parameters translation from OpenVSP to CPACS

To obtain the correspond CPACS, the fuselage is mapped preserving the sectional definition adopted in OpenVSP. As the same for the wing, the following list shows the implemented parameters inside VSP2CPACS:

- **Number of points**: Similar to the wing, the control parameter that sets the number of points is called `tess_W`. It controls the discretization of each profile, but for the superellipse it is set to 120 due to the higher geometrical complexity of its shape.

- **Orientation**: This part works in the same way of the wing.
- **Symmetry**: Symmetry can be defined as the wing with all the available options or without setting it.
- **Length**: This parameter controls the length of the fuselage. In CPACS there is no single, dedicated parameter equivalent to this one. The distance between nose and tail is defined locally in the `positioning` element, via the `length` entries, which describe the relative distance between consecutive sections. To recover the absolute x-location of each section defined in OpenVSP, VSP2CPACS reads the corresponding section positions and reconstructs the fuselage in CPACS so that the longitudinal layout matches the original OpenVSP model.
- **Section orientation**: Unlike the wing in OpenVSP, fuselage sections can be freely translated in (x, y, z). VSP2CPACS extracts these translations and writes them into the `transformation` part of the CPACS definition, ensuring that each fuselage section is placed at the correct position in space. In OpenVSP, moving the section in space is done in the `XSec` where the parameters in which the user can play are `x`, `y`, `z` to translate and to rotate `Rot_X`, `Rot_Y`, `Rot_Z`.
- **Spin**: The `Spin` parameter in `OpenVSP` generates a helical twist transition between fuselage sections, typically used for nacelles or rotating components. In VSP2CPACS, when a non-zero `Spin` value is detected at a generic section i, it is duplicated and geometrically modified to approximate this helical deformation. Without having access to the OpenVSP implementation, this feature is mainly done using an inverse design approach that, for sure, will be an approximation but it reproduces the same physical behavior: a rotation–scaling transition along the longitudinal axis. To couple twist and local rotation, a normalized helical phase is defined:

$$a = \mathtt{spin} - \frac{\mathtt{x\_rot} \cdot 0.25}{90}, \tag{2.16}$$

$$\mathtt{phase} = a \in [0, 1). \tag{2.17}$$

  Where:

  - `spin`: total helical revolutions (real number, [-1,1]).
  - `x_rot`: section rotation angle [°].
  - $\frac{0.25}{90}$: conversion factor mapping 90° $\longrightarrow$ 0.25 of spin.

  A triangular attenuation function is constructed from the phase:

$$\mathtt{factor} = |2\,\mathtt{phase} - 1|\,.$$

  This produces a periodic triangular wave:

  - `phase` = 0 or 1 $\Rightarrow$ `factor` = 1 (full section size)

- `phase` = 0.5 ⇒ `factor` = 0 (minimum scaling)

The geometric interpretation is a smooth transition over one spin cycle:

$$\text{full} \;\rightarrow\; \text{scaled} \;\rightarrow\; \text{full}$$

In the following table is summarized the values that could be appear, it's important that the choice of building a periodic phase is due only to have value between [0,1) to represent the behaviour in OpenVSP. The axial twist applied to the duplicated section

a	`phase`	`factor`	Relative size
0.00	0.00	1.00	100%
0.25	0.25	0.50	50%
0.50	0.50	0.00	Minimum
0.75	0.75	0.50	50%
1.00	0.00	1.00	100% (periodic)

Table 2.2: Triangular scaling behavior over one normalized spin cycle

is:

$$\texttt{x\_rot}_{\text{new}} = \texttt{x\_rot} - \frac{\texttt{spin} \cdot 90}{0.25}.$$

This ensures the total rotation corresponds to the prescribed number of revolutions. To avoid geometric collapse and CPACS degeneracies, the scaling is bounded:

$$\texttt{y}_{\text{new}} = 1.5 \cdot \max(0.01 * \texttt{y\_base}, \texttt{factor}\,\texttt{y\_base}), \tag{2.18}$$

$$\texttt{z}_{\text{new}} = 1.5 \cdot \max(0.01\,\texttt{z\_base}, \texttt{factor}\,\texttt{z\_base}). \tag{2.19}$$

where:

- `correction_factor` = 1.5 amplifies the transition effect,
- `min_ratio` = 0.01 prevents zero-area sections.

Thus, even at minimum scaling, the section retains a minimal non zero dimension for numerical stability. The Spin parameter must be used with care, since the method used to generate the surface between two sections differs between OpenVSP and CPACS. In OpenVSP, the spin transformation is applied continuously along the geometry, producing a smooth helical transition. In CPACS, however, the surface is reconstructed from discrete cross-sections, and the interpolation between these sections follows a different formulation. As a result, the same Spin value may not produce an identical geometric outcome after conversion. For this reason, the Spin parameter should be applied only when a sufficient number of sections is defined. Increasing the number of sections improves the discretization of the twist and leads to a more accurate representation of the intended helical geometry. In general, the more sections the user introduces in regions affected by Spin, the closer the reconstructed CPACS geometry will match the original OpenVSP model.

The parameters listed above represent the currently implemented features. Although not all available functionalities are fully supported yet, this implementation constitutes a significant step forward, comparable to the level of detail adopted for the wing during the preliminary design phase. Further extensions will progressively incorporate the remaining features. The geometric differences between the OpenVSP and CPACS representations are generally more noticeable for the fuselage than for the wing. This is mainly due to the larger dimensions and stronger curvature variations typically present in fuselage geometries. Although the section definitions themselves may be identical, the way the surface is reconstructed between sections differs between OpenVSP and CPACS. As a result, small discrepancies in interpolation can become more visible over larger or more curved geometries.For example, consider a fuselage nose with a smooth and continuously varying surface. If this region is not described using a sufficient number of cross-sections, the reconstructed geometry in CPACS may deviate from the original OpenVSP shape. Increasing the number of sections in areas with strong curvature gradients improves the fidelity of the converted geometry and reduces these visual and geometric differences.

2.3.2 Fuselage profiles

As is written in the previous chapter for the wing, the fuselage profile could be one of the implemented shapes, shown in Tab. 2.1. In the wing part are described the shapes more used as airfoils; this chapter is focused to explain the typical profile for a fuselage component. The CPACS order for a fuselage's profile is different respect the wing case and the points that define it are typically given in y and z with x set to zero. Starting point should be the lowest point(typically in the symmetry plane), then upwards on the positive y-side up to the highest point and it continues on the negative y-side back down to the lowest point.

2.3.3 Circle and point

The functions `get_coord_point` and `get_coord_circle` are used to generate 2D coordinates for circular cross-sections, which are fundamental in defining fuselage or component profiles in VSP2CPACS. Both functions produce symmetric, closed shapes, but they differ in purpose and scaling. `get_coord_circle` operates creating a circle with a diameter $d = 1$ and discretizes it with the selected number of points. In particular, it generates half circle:

$$\theta = \text{linspace}(0, \pi, n/2), \quad x = \frac{d}{2}\cos\theta, \quad y = \frac{d}{2}\sin\theta.$$

where:

- `n`: is the number of points.

The second half of the circle is generated by mirroring:

$$x_{\text{full}} = [x, -x], \quad y_{\text{full}} = [-y, y].$$

After the profile is closed by ensuring the first and last points coincide. No scaling is applied because, as is done for the wing's airfoils, the profiles are normalized and after scaled in the specific CPACS part. In this case will be the `Diameter` set in OpenVSP that will scaled the profile in y and z. The definition of the `Point` profile works in a way that build a unitarian circle with 40 points, and after is scaled setting a diameter $d = 0$.

2.3.4 Ellipse and super ellipse

The `get_coord_ellipse` function is designed to generate the coordinates of a two-dimensional airfoil-like profile shaped as an ellipse. This profile is fully defined by its height and width, provided by the parameters `Height` and `Width` availble in the OpenVSP's GUI. The semi-major axis a and semi-minor axis b of the ellipse are initialized with unit values and later scaled according to the width and height of the section. The general parametric form of an ellipse in two dimensions is

$$\begin{cases} x(\theta) = \dfrac{a}{2}\cos(\theta) \\ y(\theta) = \dfrac{b}{2}\sin(\theta) \end{cases} \quad \text{with} \quad \theta \in [0, \pi].$$

Since only the upper half of the ellipse is generated at first (from 0 to π), the lower half is obtained by symmetry. The function concatenates both to form a closed profile:

$$x_{\text{full}} = [x, -x], \quad y_{\text{full}} = [-y, y].$$

The function `get_coord_superellipse` generates 2D coordinates for a super-elliptical, airfoil-like cross-section, extending the classical ellipse by introducing shape exponents that control curvature. This allows the creation of smooth, asymmetric profiles with adjustable thickness distribution. The profile is defined by several geometric parameters: `width` (b), `height` (a), shape exponents for the upper and lower part N, `M` and `MaxWLoc` to locally modify the vertical coordinates near the maximum width. The basic parametric equations of a superellipse are:

$$\begin{cases} x(\theta) = \dfrac{b}{2}\,|\sin\theta|^{2/M}, \\ y(\theta) = \dfrac{a}{2}\,|\cos\theta|^{2/N}, \end{cases} \qquad \theta \in \left[0, \frac{\pi}{2}\right].$$

Separate exponents N and M are applied to the upper and lower halves, allowing asymmetric shaping. When $N = M = 2$, the profile reduces to a classical ellipse; larger values produce a more rectangular or rounded contour. To increase local thickness near the mid-chord region, a vertical weighting function $w(x)$ is implemented:

$$w(x) = \begin{cases} 0, & |x| \le x_{\text{core}}, \\ \dfrac{1}{2}\left[1 - \cos\left(\pi \dfrac{|x| - x_{\text{core}}}{(b/2) - x_{\text{core}}}\right)\right], & |x| > x_{\text{core}}, \end{cases}$$

with $x_{\text{core}} = 0.7\,(b/2)$. This function smoothly transitions from zero near the center to one at the edges, producing a gradual curvature. This is done for the upper part and the lower part due to to the different parametrization given from OpenVSP. Also, the vertical coordinates of the upper half are then adjusted to mach the behaviour of `MaxWLoc` :

$$y'_{\text{up}} = y_{\text{up}} + \texttt{MaxWLoc} \cdot w(x_{\text{up}}), \quad y'_{\text{low}} = y_{\text{low}},$$

The full profile is constructed by concatenating the upper and lower parts and closing the shape. Since no closed-form analytical expression exists for the exact super-elliptical deformation with localized thickness modulation, the implementation in `get_coord_superellipse` represents a numerical approximation. This method reproduces the intended curvature behavior with sufficient accuracy for geometric modeling and CPACS export. Below, in Fig. 2.6, a fuselage is shown to illustrate the use of both elliptical and super-elliptical cross-sections. The geometry is composed of the following sections:

Section	Type	Width	Height	MaxWLoc	M	N
0	SuperEllipse	2.5	4	0	5	5
1	SuperEllipse	6	4	0	2	2
2	SuperEllipse	2.5	4	0	10	10
3	Ellipse	2.5	4	–	–	–
4	Point	–	–	–	–	–

The resulting fuselage geometry is shown in Fig. 2.6.

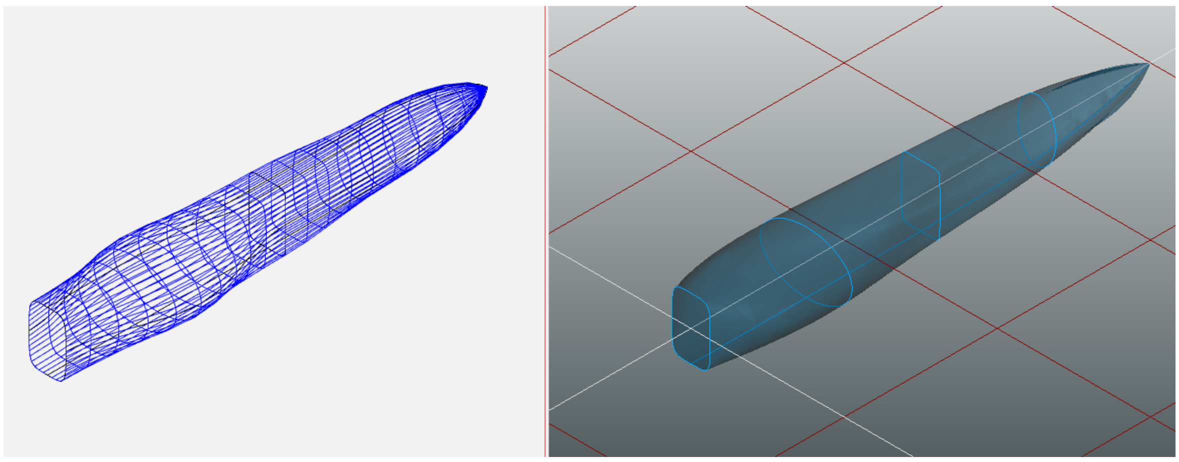


Figure 2.6: Fuselage visualization using ellipse and superellipse cross-sections

2.3.5 Rounded rectangle

The function `get_coord_roundedrectangle` generates a two-dimensional rounded-rectangle. The shape combines straight edges with smoothly blended corners and allows independent control of width, height, skew, vertical skew, keystone taper, and individual corner radii. This parameterization provides a flexible geometric primitive consistent with the rounded-rectangle cross section available in OpenVSP. The construction begins from a normalized reference rectangle of unit width $w = 1$ and height $h = 1$. Four nominal vertices are defined: bottom-left (BL), bottom-right (BR), top-right (TR), and top-left (TL). These vertices are modified through three geometric controls:

- *Skew*: introduces a horizontal displacement between top and bottom edges, generating a rhomboidal deformation.
- *Vertical skew*: shifts the left and right sides vertically in opposite directions, producing a vertical shear.

- *Keystone*: alters the relative length of the top and bottom edges, enabling trapezoidal or triangular-like cross-sections.

The resulting vertex coordinates, written as points P_i(represent with $[y, z]^T$), can be written as:

$$\mathbf{p}_{\text{BL}} = \begin{bmatrix} -\dfrac{\text{skew}}{2} - \text{VSskew} + \left(\text{keystone} - \frac{1}{2}\right)\dfrac{h}{2} \\ 0 \end{bmatrix}, \quad \mathbf{p}_{\text{BR}} = \begin{bmatrix} w - \dfrac{\text{skew}}{2} \\ \text{VSskew} - \left(\text{keystone} - \frac{1}{2}\right)\dfrac{h}{2} \end{bmatrix},$$

$$\mathbf{p}_{\text{TR}} = \begin{bmatrix} w + \text{skew} \\ h + \text{VSskew} + \left(\text{keystone} - \frac{1}{2}\right)\dfrac{h}{2} \end{bmatrix}, \quad \mathbf{p}_{\text{TL}} = \begin{bmatrix} \text{skew} \\ h - \text{VSskew} - \left(\text{keystone} - \frac{1}{2}\right)\dfrac{h}{2} \end{bmatrix}.$$

These four points define a general quadrilateral reflecting the prescribed skew and taper effects. To replace sharp corners with smooth transitions, as is available in OpenVSP at each vertex is assigned a radius r_i. For each corner, the adjacent edge lengths and corresponding unit direction vectors are:

$$\ell_{\text{in}} = \|\mathbf{p}_{\text{corner}} - \mathbf{p}_{\text{prev}}\|, \qquad \ell_{\text{out}} = \|\mathbf{p}_{\text{next}} - \mathbf{p}_{\text{corner}}\|$$

$$\mathbf{d}_{\text{in}} = \frac{\mathbf{p}_{\text{corner}} - \mathbf{p}_{\text{prev}}}{\ell_{\text{in}}}, \qquad \mathbf{d}_{\text{out}} = \frac{\mathbf{p}_{\text{next}} - \mathbf{p}_{\text{corner}}}{\ell_{\text{out}}}$$

Where:

- The corner i has vertex $\mathbf{p}_{\text{corner}}$
- The corner $i - 1$ has vertex $\mathbf{p}_{\text{prev}}$
- The corner $i + 1$ has vertex $\mathbf{p}_{\text{next}}$

When replacing a sharp polygonal corner with a rounded transition, the adjacent edges must first be cut in order to insert a smooth blending curve. Let $\mathbf{p}_{\text{corner}}$ denote the original vertex, and let ℓ_{in} and ℓ_{out} be the lengths of the incoming and outgoing edges, respectively. If a rounding radius r is prescribed, a implementation would trim each edge by a distance equal to r. However, this may lead to geometric degeneracy in two situations:

- If $r > \ell_{\text{in}}$ or $r > \ell_{\text{out}}$, the trimming point would move beyond the adjacent vertex, causing edge inversion or self-intersection.
- If two neighboring corners both use large radii, each may consume up to half of a shared edge, eliminating the straight segment between them and producing a degenerate geometry.

To ensure geometric robustness, the trimming distance is therefore limited:

$$\text{cut} = \begin{cases} 0, & r \le \varepsilon, \\ \min\left(r,\ 0.45\,\ell_{\text{in}},\ 0.45\,\ell_{\text{out}}\right), & r > \varepsilon, \end{cases}$$

where ε is a small numerical tolerance. The factor 0.45 (instead of 0.5) introduces a safety margin that guarantees a finite straight segment remains between adjacent rounded corners. This avoids degenerate configurations and ensures numerical stability during Bézier curve construction and subsequent CPACS export. The trimming procedure is illustrated in Fig. 2.7.

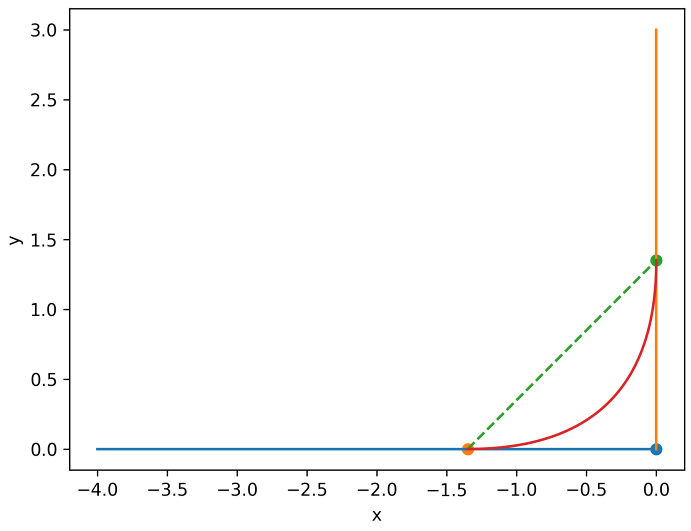


Figure 2.7: Corner trimming procedure with radius limitation. The straight edges are shortened by a distance cut, and the sharp vertex is replaced by a quadratic Bézier curve.

The original sharp corner is replaced by a quadratic Bézier curve connecting the two trimming points, which are obtained by moving a distance cut along the adjacent edges. The straight edges are shortened accordingly, defining the entry and exit points of the rounded corner:

$$\mathbf{p}_{\text{start}} = \mathbf{p}_{\text{corner}} - \mathbf{d}_{\text{in}}\,\text{cut}, \qquad \mathbf{p}_{\text{end}} = \mathbf{p}_{\text{corner}} + \mathbf{d}_{\text{out}}\,\text{cut}.$$

The transition between $\mathbf{p}_{\text{start}}$ and $\mathbf{p}_{\text{end}}$ is generated using a quadratic Bézier curve,

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2, \qquad t \in [0,1],$$

with $\mathbf{P}_0 = \mathbf{p}_{\text{start}}$, $\mathbf{P}_1 = \mathbf{p}_{\text{corner}}$, and $\mathbf{P}_2 = \mathbf{p}_{\text{end}}$. This formulation guarantees a smooth and continuously differentiable corner blending. The straight portions between successive rounded corners are discretized by linear interpolation,ensuring uniform point distribution along each side.

$$\mathbf{p}(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a}), \qquad t \in [0,1],$$

The complete profile is obtained by concatenating bottom, right, top, and left segments, followed by closure of the loop. here there is a fuselage example with a rounded rectangle profiles, as shown in Fig. 2.8.

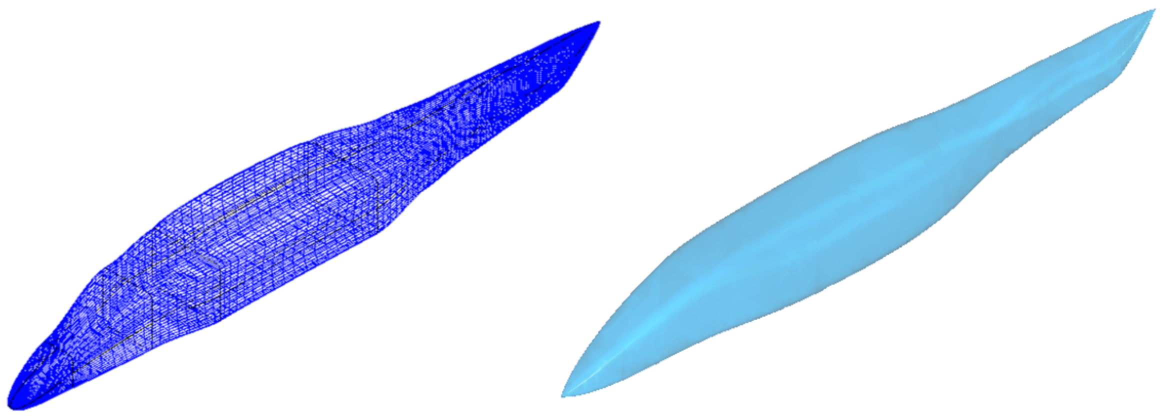


Figure 2.8: Comparison between OpenVSP and CPACS

2.4 POD component translation

In OpenVSP, the POD component represents a simplified axisymmetric body typically used for external stores, engine nacelles, sensor housings, fuel tanks, or generic streamlined geometries. It is parametrically defined through a reduced set of global design variables, making it particularly suitable for conceptual aircraft design studies. The implementation of the POD component within VSP2CPACS was considered essential due to its high flexibility during the aircraft design process. In OpenVSP, the POD is frequently used as a generic streamlined body that can easily represent a wide range of external or embedded components without requiring a complex parameterization. For this reason, supporting the POD translation significantly enhances the capabilities of VSP2CPACS. The primary parameters controlling a POD in OpenVSP are:

- **Length** (L): total longitudinal extent of the body.
- **Fineness ratio** (F): ratio between the total length and the maximum diameter.
- **Tessellation parameter** (n): number of circumferential discretization points used to describe each circular cross-section.

The fineness ratio defines the maximum radius as

$$r_{\max} = \frac{L}{F}.$$

Internally, OpenVSP generates the POD as a body of revolution with circular cross-sections automatically distributed along the longitudinal axis. The user does not explicitly define individual sections; instead, the surface is constructed from the global parameters(see Figure 2.9 for the visualization of the OpenVSP's POD).

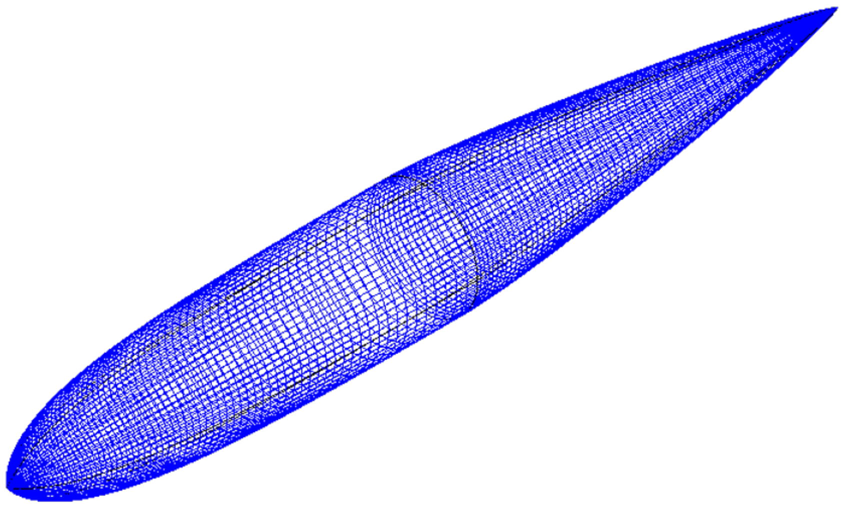


Figure 2.9: POD visualization in OpenVSP

2.4.1 Mapping strategy in VSP2CPACS

Since CPACS does not provide a dedicated POD component, the geometry is mapped to a `fuselage` element. The global parameters (L, F) and the full spatial transformation (translation, rotation, scaling) are extracted from OpenVSP. The tessellation parameter is also retrieved to reconstruct the circular profile consistently. Because the exact analytical surface definition used internally by OpenVSP is not accessible, the translation is performed through an inverse design approach. The POD is approximated by an axisymmetric fuselage composed of circular sections distributed along the x-axis. To avoid an unnecessarily large CPACS file for such a simple geometry, the number of sections is fixed to 11. This choice represents a trade-off between geometric fidelity and model compactness. The sections are distributed as follows:

- 5 sections in the nose region,
- 1 section in the central cylindrical region,
- 5 sections in the rear region.

Let $s = x/L$ be the non-dimensional longitudinal coordinate. Two transition points are defined:

$$s_1 = 0.3, \qquad s_2 = 0.6.$$

The radius distribution $R(x)$ is defined piecewise to approximate the OpenVSP geometry. The radius will be how much VSP2CPACS will scale the circular profile in y and z direction. The distribution is designed to capture the typical POD shape, with a smooth nose curvature, a constant diameter midsection, and a tapered rear end:

$$R(x) = \begin{cases} r_{\max}\sqrt{1-\left(1-\left(1-\dfrac{s}{s_1}\right)\right)^{1.5}}, & 0 \le s \le s_1, \\ r_{\max}, & s_1 < s \le s_2, \\ r_{\max}\sqrt{1-\left(\dfrac{s-s_2}{1-s_2}\right)}, & s_2 < s \le 1. \end{cases}$$

The front region behaves as a quarter superellipse, providing the nose curvature. The central region has constant radius. The rear region tapers smoothly to zero using an elliptical-like decay. The coefficients and exponents were tuned to closely match the OpenVSP-generated POD while ensuring smoothness and numerical robustness. Each section is defined as a circular profile and the profile is discretized using the OpenVSP tessellation parameter. Fig. 2.10 presents a four-view comparison between the OpenVSP POD geometry (blue) and the reconstructed CPACS fuselage representation (green). The overlap demonstrates that, despite the absence of a dedicated POD entity in CPACS, the fuselage-based inverse reconstruction accurately reproduces the original geometry.

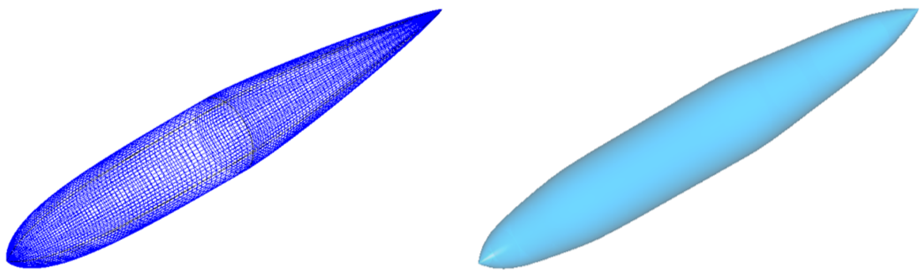


Figure 2.10: Comparison between OpenVSP POD and reconstructed CPACS fuselage

2.5 Duct component translation

In CPACS, an engine is not defined as a simple geometric primitive but rather as a system-level component that integrates geometry with functional and performance-related data. The complete CPACS engine definition involves several interconnected elements such as the nacelle, fan cowl, core cowl, and center body. A detailed description of the CPACS engine structure is provided in the dedicated section of this thesis. OpenVSP does not provide a component explicitly identified as an *engine*. However, during the conceptual design phase the nacelle geometry is typically the dominant external feature that characterizes the propulsion system. Among the available OpenVSP components, the *Duct* provides the closest geometric representation to an engine nacelle. The implementation of the Duct translation in *VSP2CPACS* therefore allows designers to directly convert nacelle-like geometries created in OpenVSP into a consistent CPACS engine representation. This capability significantly simplifies the workflow for users who wish to quickly generate a valid CPACS aircraft model including propulsion components. The target user of this feature is typically an engineer already familiar with CPACS who wants to rapidly construct an engine architecture starting from a conceptual OpenVSP geometry. The conversion routine automatically generates a syntactically correct CPACS engine structure, which can then be further refined or extended by the user if additional fidelity is required.

2.5.1 OpenVSP Duct parameterization

In OpenVSP, the Duct component is defined as a body of revolution generated from a two-dimensional airfoil profile. The user controls both the transversal section of the duct and the longitudinal airfoil shape through a set of intuitive parameters. The parameters extracted and implemented in *VSP2CPACS* are:

- **Width and Height**: define the transversal dimensions of the duct in the y–z plane.
- **Chord, Thickness-to-Chord ratio (T/C), Camber, Camber Location**: define the airfoil profile in the x–z plane. This longitudinal section is subsequently revolved around the axis to generate the nacelle-like body.

A visual representation of the Duct component in the OpenVSP graphical interface is shown in Figure 2.11. The figure illustrates how the generated geometry closely resembles a nacelle, which motivates its selection as the base element for constructing a CPACS engine.

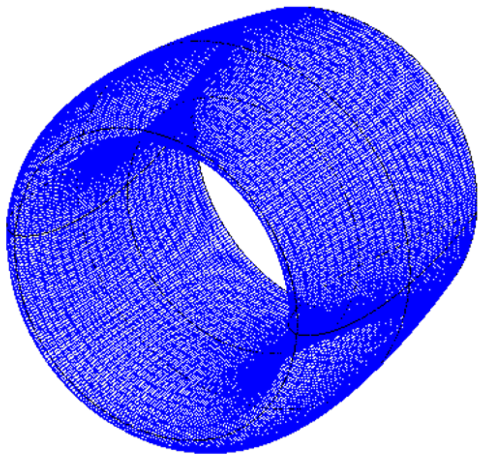


Figure 2.11: Duct component visualization in OpenVSP

2.5.2 Mapping strategy

The translation routine begins when *VSP2CPACS* detects a *Duct* component that defines the external nacelle surface. This outer duct is interpreted as the *Fan Cowl* of the CPACS engine structure. From the parameters extracted from OpenVSP, a four-digit NACA airfoil profile is generated to represent the longitudinal section of the nacelle. The resulting profile is then revolved to reconstruct the nacelle surface. In order to preserve the geometry defined by the user in OpenVSP, a predefined smooth rotation curve is applied. To represent a complete nacelle system, the routine supports the following logical configuration:

- An **outer Duct** representing the *Fan Cowl*
- An **inner Duct** representing the *Core Cowl*
- A **POD component** representing the *Center Cowl* (nose cone)

This configuration allows a simplified but consistent representation of a typical turbofan nacelle geometry. If the user includes an inner duct within the outer nacelle, the routine automatically interprets it as the core cowl. The nose cone is modeled using the POD component described in the previous section. To ensure numerical robustness and smooth surface reconstruction, a set of predefined parameters is used to control the rotation curve applied during the generation of the nacelle geometry. These parameters were tuned to provide a stable geometric reconstruction while remaining consistent with the original OpenVSP shape. The parameters used for both the fan cowl and the core cowl are summarized in Table 2.3.

Feature	**Fan Cowl**	**Core Cowl**
startZeta	−0.28	−0.28
endZeta	−0.25	−0.25
startZetaBlending	−0.30	−0.30
endZetaBlending	−0.23	−0.23

Table 2.3: Parameters used for the reconstruction of the nacelle rotation curve.

The engine component represents one of the most complex structures within the CPACS schema, requiring a large number of parameters compared to simpler components such as wings or fuselages. The implementation of the Duct-to-engine translation in *VSP2CPACS* automates a significant portion of this process by generating a valid CPACS engine structure directly from OpenVSP geometry. This reduces the manual effort required to construct propulsion system definitions and allows designers to quickly obtain a consistent CPACS aircraft model. As a result, this feature facilitates the integration of OpenVSP conceptual design models within CPACS-based workflows and may encourage broader adoption of the CPACS format within the aircraft design community.

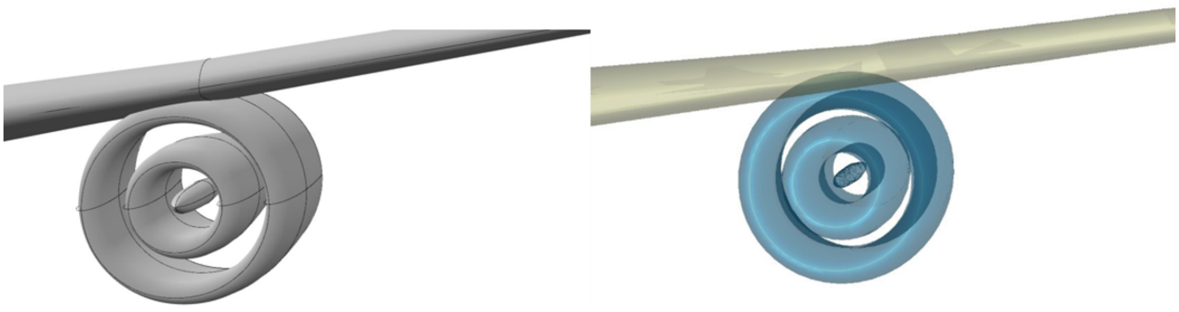


Figure 2.12: Comparison between the OpenVSP Duct geometry and the generated CPACS engine nacelle

2.6 Validation and test cases

To validate the implementation of the VSP2CPACS module, a series of test cases was developed. These cases are available in the folder `VSP files` within the CEASIOMpy repository. The test suite includes a variety of aircraft configurations, ranging from simple wing–body combinations to more complex geometries composed of multiple interconnected components. In this section, two of the most complex and representative geometries are presented to demonstrate the capabilities and robustness of the module. The first test case consists of a generic transport aircraft configuration. The geometry includes a fuselage, a swept and tapered wing with winglets, and a complete engine assembly. The fuselage is defined by 11 elliptical cross-sections, while the wing is composed of four sections. The engine is constructed using a combination of DUCT and POD components in OpenVSP to generate the corresponding CPACS engine definition. Figure 2.13 shows the OpenVSP model (top) and the generated CPACS geometry (bottom).

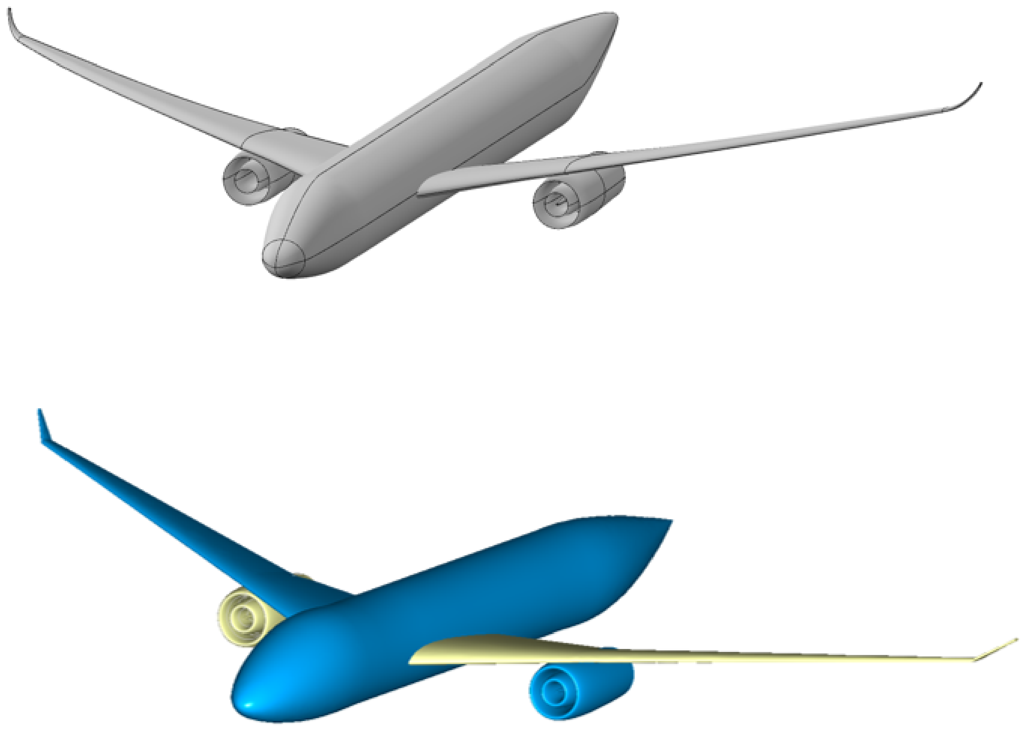


Figure 2.13: Generic transport aircraft geometry in OpenVSP (top) and corresponding CPACS model (bottom)

The second test case represents a more complex configuration based on a P180-type aircraft. This configuration was selected due to its relatively large number of components and unconventional layout. The geometry consists of a main fuselage, a front canard, and rear vertical and horizontal tail planes. The main wing is tapered, and an engine is modeled using fuselage and POD components attached to the wing. This example highlights the flexibility of the VSP2CPACS module. Although in this particular case the engine does not strictly follow the standard CPACS engine syntax, the user retains full freedom to construct custom configurations using the available OpenVSP components.

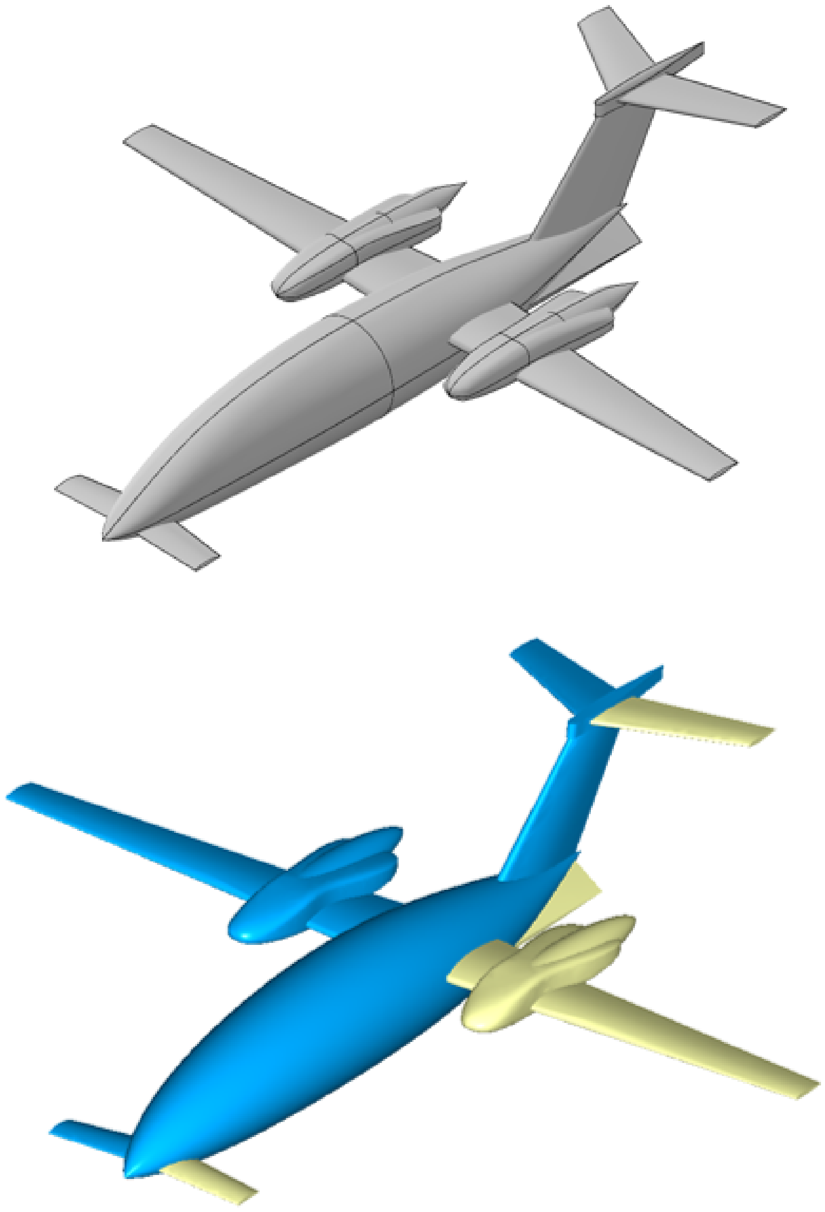


Figure 2.14: P180-type aircraft geometry in OpenVSP (top) and corresponding CPACS model (bottom)

Additional geometries are available in the repository. Users are encouraged to further test the module with different configurations and to report any issues or suggestions for improvement. The provided test cases aim to cover a broad range of geometric scenarios and to demonstrate the robustness and versatility of the VSP2CPACS module in handling diverse aircraft design configurations.

3. STL2CPACS

Modern aircraft design increasingly relies on multidisciplinary workflows where geometrical models must be exchanged between multiple software environments. Within the CEASIOMpy framework, the CPACS format provides a standardized and structured description of aircraft configurations, enabling consistent data exchange between geometry generation, aerodynamic analysis, stability assessment, and performance evaluation modules. A major step toward enabling this interoperability was the development of the `VSP2CPACS` module, which allows parametric aircraft models created in OpenVSP to be automatically converted into CPACS-compatible geometries. Through this interface, designers can define aircraft configurations parametrically in OpenVSP and seamlessly integrate them into CEASIOMpy-based analysis workflows. Despite its effectiveness, this approach presents an inherent limitation: it assumes that the aircraft geometry is available as an OpenVSP parametric model. In practice, however, aircraft geometries are often generated using a wide variety of design tools and methodologies. CAD environments, optimization frameworks, reverse-engineering processes, and external geometry generators frequently produce surface representations in the STL format, where the geometry is described as a triangulated mesh rather than a structured, parametric model. While STL meshes accurately capture the external shape of an aircraft, they do not contain the hierarchical and parametric information required by CPACS. Important structural concepts such as components, sections, or reference parameters are absent, as the geometry is represented solely as a collection of discrete triangular facets. As a consequence, integrating such geometries into CEASIOMpy would require manually rebuilding the aircraft model within a parametric environment such as OpenVSP. This process is time-consuming and may introduce inconsistencies or approximation errors. These limitations highlight the need for a methodology capable of reconstructing a structured CPACS representation directly from triangulated surface data. Addressing this challenge constitutes the primary motivation for the development of the `STL2CPACS` module. The objective of this tool is to bridge the gap between unstructured mesh geometries and the hierarchical aircraft description required by CPACS, enabling aircraft models originating from a wide range of design environments to be integrated into CEASIOMpy workflows.

3.1 Introduction

To address the challenges described above, the `STL2CPACS` module has been developed as a geometry conversion tool capable of transforming triangulated STL aircraft models into structured CPACS representations. Instead of relying on parametric definitions, the module reconstructs the required geometric information directly from the surface mesh. The conversion process is based on a sequence of geometric operations designed to extract and reorganize key information from the raw triangulated data. These operations progressively transform the unstructured mesh into a description compatible with the CPACS aircraft model. More specifically, the `STL2CPACS` workflow involves the following steps:

- Identification and separation of the main aircraft components (e.g., wing, fuselage, and tail surfaces).
- Slicing of the triangulated mesh to generate cross-sectional profiles along the main reference directions.
- Computation of local geometric parameters.
- Reconstruction of CPACS definitions for each identified aircraft component.

Through this automated procedure, the module converts an initially unstructured triangular surface representation into a consistent and reusable parametric aircraft description. In this way, `STL2CPACS` complements the existing `VSP2CPACS` workflow: while the last one remains the preferred pathway when parametric OpenVSP models are available, `STL2CPACS` provides an alternative solution when only triangulated surface geometries exist. The following sections describe the methodology implemented in `STL2CPACS`, detailing the geometric processing techniques used to identify aircraft components, extract sectional information, and reconstruct the CPACS structure from STL surface meshes.

3.2 STL geometry definition and preprocessing

The STL (Stereolithography) [15] file format is a surface representation standard that describes a three-dimensional object exclusively by a tessellation of planar triangular facets. It is one of the most frequently used geometric exchange formats in engineering applications due to its simplicity, robustness, and broad compatibility with CAD systems, meshing tools, and numerical solvers.In an STL file, the geometry of a solid is approximated by a set of triangles. Each triangle is defined by three vertices (given as Cartesian coordinates) and an associated outward normal vector. The format does not contain explicit information about topology (such as a list of unique vertices), parametric surfaces, curves, feature trees, or component hierarchy. Therefore, it provides a purely discrete geometric description of the surface. This is advantageous for visualization, rapid prototyping, and mesh-based numerical simulations, but implies that any higher level interpretation of the geometry must be reconstructed algorithmically from the triangular data. Two different encodings of the STL format exist: ASCII and binary. Both representations store equivalent geometric information but differ in structure and storage efficiency. In the ASCII version, the file is written in a human-readable text format. Each triangular facet is described using keywords such as `facet normal`, `outer loop`, and `vertex`. This representation is convenient for manual inspection, but it typically leads to larger file sizes and slower input/output operations compared to the binary alternative.In the binary STL format, the same geometric data are encoded in a compact binary structure. A binary STL file begins with an 80-byte header, followed by a 4-byte unsigned integer that specifies the number of triangles. Each triangle is then stored in a fixed-length record that contains three 32-bit floating point values for the components of the normal vector, nine 32-bit floating-point values for the three vertices, and a 2-byte attribute field (often unused and set to zero). Binary STL files are significantly smaller and faster to read than their ASCII counterparts, at the cost of not being human-readable without dedicated tools. `STL2CPACS` module, support both encodings. The code reads the initial portion of the file and inspects the first bytes to decide whether the file should be interpreted as ASCII or binary. If the header begins with the keyword `solid`, the file is tentatively treated as ASCII and parsed accordingly; if that attempt fails, the file is reinterpreted as binary. In all cases, the resulting surface is converted into a unified internal representation, an array of shape $N \times 3 \times 3$, where N denotes the number of triangular facets and each facet contains three vertices in three-dimensional space.After parsing, the STL geometry is represented internally as an array containing the coordinates of all triangle vertices. At this stage, the data structure directly reflects the STL philosophy: triangles are stored independently, and vertices shared by adjacent facets are repeated multiple times. This representation is fully sufficient for visualization and simple geometric queries, but it is not optimal for operations such as slicing. To improve computational efficiency, the STL surface is converted into the Cart3D `.tri` format. In contrast to STL, a Cart3D `.tri` file explicitly distinguishes between a list of unique vertices and a list of triangular elements, each defined by indices into the vertex list. In the implementation, the conversion procedure proceeds as follows:

1. All triangle vertices extracted from the STL file are reshaped into a single two-dimensional array of coordinates.

2. A uniqueness filtering operation is applied to identify distinct coordinate points and to associate each triangle with the index of a unique vertices.

3. Each triangle is represented by three integer indices that reference the corresponding vertices in the array of unique nodes.

4. The result is written to disk in Cart3D `.tri` format.

Below, in Fig.3.1 is shown a schematic representation of the two stl format and on the right how is defined the .tri file.

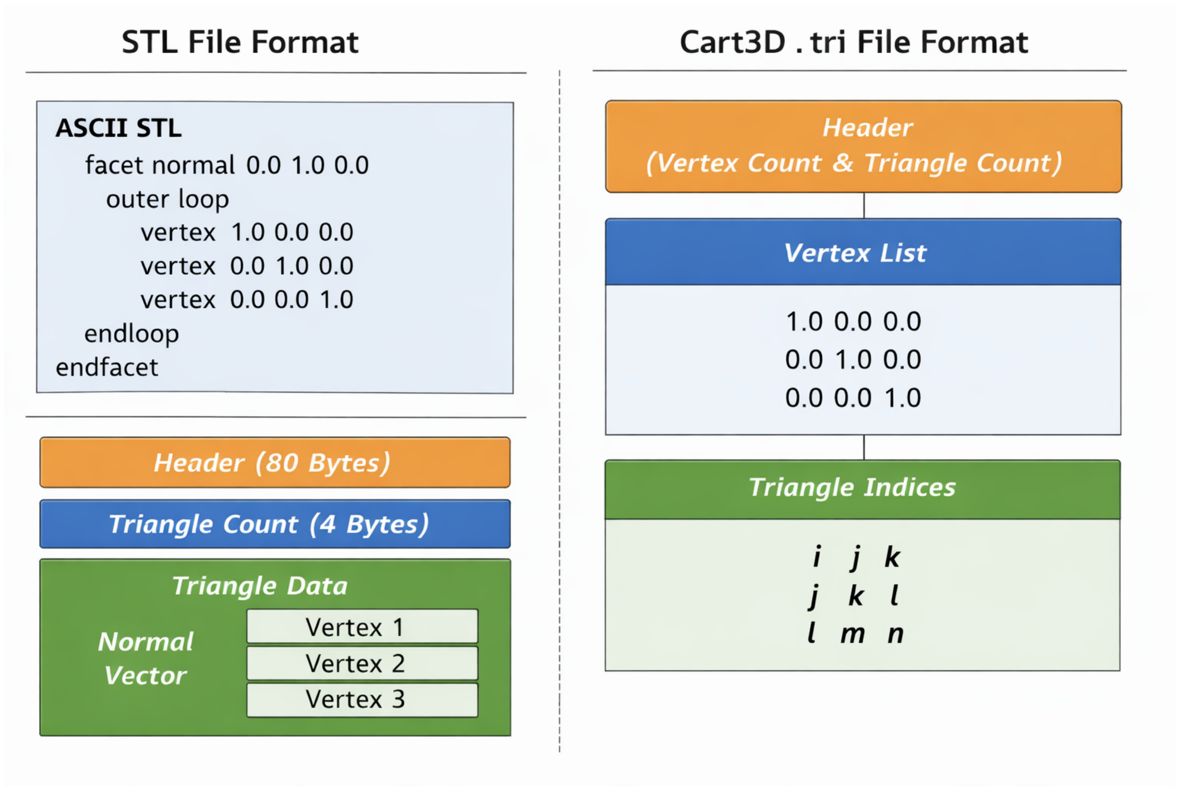


Figure 3.1: STL and Cart3D `.tri` file format comparison

The resulting `.tri` file begins with a header line specifying the total number of vertices and the total number of triangles. It is then followed by the list of vertex coordinates (one vertex per line) and the list of triangle connectivity indices (one triangle per line).This reorganization provides several important advantages: elimination of redundant vertex storage due to duplicated coordinates in the original STL, reduced memory footprint for large meshes, clear separation between geometry (vertex coordinates) and topology (element connectivity), and improved performance for vectorized numerical operations on vertices and elements. Most importantly, the explicit vertex–element structure significantly simplifies geometric algorithms such as plane–triangle intersection, which form the basis of the slicing procedures used later in `STL2CPACS`. This structured mesh representation constitutes the starting point for all subsequent geometric operations within the `STL2CPACS` module, such as slicing, sectional extraction, and airfoil reconstruction.

3.3 Mesh slicing and triangle-plane intersection

To extract geometric cross-sections from arbitrary STL geometries, a mesh slicing procedure has been developed that operates on a triangulated surfaces. This routine is used consistently throughout the STL2CPACS workflow, both for spanwise sections of lifting surfaces (slicing with planes of constant Y) and for fuselage sections (slicing with planes of constant X). In all cases, the algorithm takes as input a set of mesh vertices and triangular connectivity, and returns, for each slicing plane, a discrete point cloud representing the intersection curve. The slicing planes are represented in an analytical form as

$$(\mathbf{x} - \mathbf{p}_0) \cdot \mathbf{n} = 0,$$

where $\mathbf{p}_0$ is a reference point on the plane and $\mathbf{n}$ is the plane normal. For a family of parallel planes, the normal $\mathbf{n}$ is fixed, while $\mathbf{p}_0$ varies with the slicing coordinate. For example, for wing slices at constant $Y = Y_0$ one chooses

$$\mathbf{n} = (0, 1, 0), \qquad \mathbf{p}_0 = (0, Y_0, 0),$$

whereas for fuselage slices at constant $X = X_0$ one uses

$$\mathbf{n} = (1, 0, 0), \qquad \mathbf{p}_0 = (X_0, 0, 0).$$

This formulation makes the algorithm independent of the slicing direction; changing from spanwise sections to longitudinal sections simply corresponds to changing the normal vector and reference point. For each reference point $\mathbf{p}_i$, the signed distance to the current plane is computed as

$$d_i = \mathbf{n} \cdot (\mathbf{p}_i - \mathbf{p}_0),$$

which is positive for points on one side of the plane, negative on the other side, and close to zero for points that lie on the plane within a numerical tolerance. A small numerical tolerance ε (set at 10^{-6} m) is introduced to classify vertices as coplanar when their computed signed distance satisfies $|d_i| < \varepsilon$; without this tolerance, such vertices would be erroneously classified as being slightly above or below the plane, leading to missed intersections. The distance field is evaluated at all vertices and then gathered per triangle. For a given triangular facet with vertices $(\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c)$ and corresponding distances (d_a, d_b, d_c), a fast selection criterion is used to determine whether the triangle may intersect the slicing plane:

$$\min(d_a, d_b, d_c) \leq \varepsilon \quad \text{and} \quad \max(d_a, d_b, d_c) \geq -\varepsilon.$$

If this condition is not satisfied, all three vertices lie strictly on the same side of the plane and the triangle cannot contribute to the intersection. If it is satisfied, the triangle either crosses the plane or lies very close to it and is therefore passed to the intersection routine. Once the algorithm knows what are the triangles that are cutted by the plane is called,the function `intersect_triangle_with_plane_point_normal`, that is the core of the slicing procedure, which computes the intersection points between a single triangle and a given plane. Consider a triangle with vertices $(\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c)$ and the distances of them from the reference plane point (d_a, d_b, d_c). The function processes the three edges

$$(\mathbf{p}_a, \mathbf{p}_b), \quad (\mathbf{p}_b, \mathbf{p}_c), \quad (\mathbf{p}_c, \mathbf{p}_a)$$

one by one. For a generic edge with endpoints $\mathbf{p}_1$ and $\mathbf{p}_2$ and distances d_1 and d_2, the following cases are distinguished inside the function:

1. **Both vertices intersect.** If $|d_1| < \varepsilon$ and $|d_2| < \varepsilon$, the edge lies (within tolerance) entirely on the plane. In this case, both endpoints $\mathbf{p}_1$ and $\mathbf{p}_2$ are added to the list of intersection points for this triangle. This situation typically occurs when the plane coincides with a mesh edge or with an almost flat patch of the surface.

2. **Single coplanar vertex.** If exactly one of the vertices satisfies $|d_i| < \varepsilon$ and the other does not, only the coplanar vertex is added to the intersection list. This corresponds to a case where the plane passes through a triangle vertex but does not intersect the opposite side of the edge.

3. **Proper edge crossing.** If d_1 and d_2 have opposite signs, i.e.

$$d_1\, d_2 < 0,$$

the edge crosses the plane at a single point located strictly between $\mathbf{p}_1$ and $\mathbf{p}_2$. The function computes the intersection parameter

$$t_{\text{int}} = \frac{d_1}{d_1 - d_2},$$

and evaluates the intersection point by linear interpolation along the edge:

$$\mathbf{p}_{\text{int}} = \mathbf{p}_1 + t_{\text{int}}(\mathbf{p}_2 - \mathbf{p}_1).$$

This guarantees that $\mathbf{p}_{\text{int}}$ lies exactly on the line segment between the two vertices and satisfies the plane equation.

To visualize these cases, consider the following schematic figure that shows four case that summarize the possible configurations of a triangle relative to a slicing plane, with the corresponding intersection points.

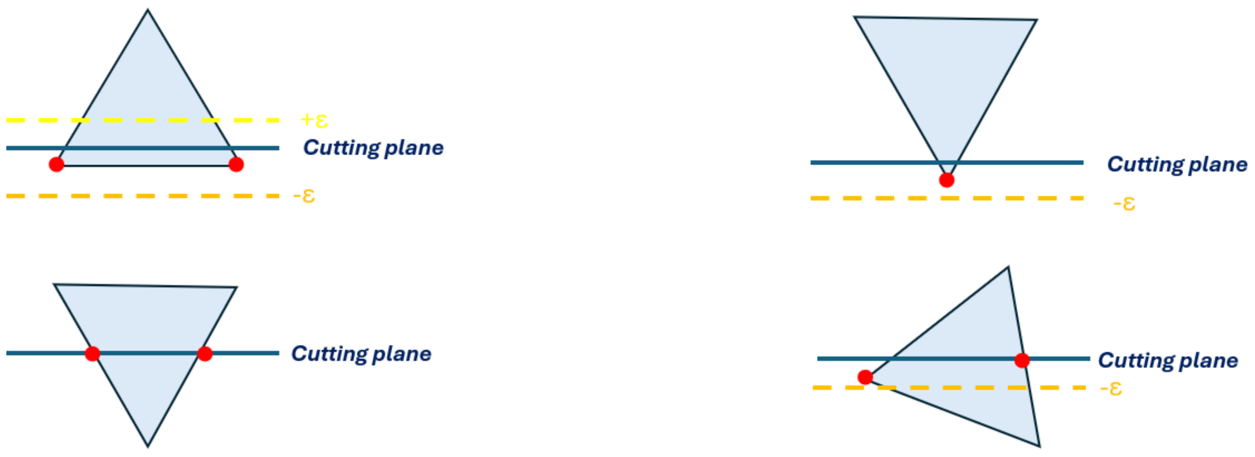


Figure 3.2: Schematic representation of triangle-plane intersection cases. Red markers indicate the intersection points returned by the algorithm.

The function applies this logic to all three edges in sequence, internally collecting all candidate intersection points into a temporary list. Depending on the configuration, the triangle can produce:

- one point (degenerate cases where the plane passes through a vertex only),
- two points (generic case of the plane cutting across the triangle),

After all edges are processed, the function post-processes this list to enforce uniqueness. Two points that are closer than a small threshold (e.g. 10^{-10} in Euclidean norm) are considered duplicates and only one representative is kept. This deduplication step is crucial to avoid artefacts such as zero-length segments or multiple identical points arising from triangles that have at a common vertex. The function returns the final list of unique intersection points for that triangle. For a given slicing plane, the algorithm loops over all triangles previously flagged as intersecting and aggregates all returned points into a single intersection set. If no triangle produces intersection points, the slice is marked as empty and an empty point cloud is stored. Otherwise, all intersection points are stacked into an array. The resulting set of unique points constitutes the discrete cross-section of the geometry with the given plane. So, for each slice STL2CPACS is able to detect the 2D section profile, that correspond to the airfoil dealing with the wing and a profile for the fuselage. This procedure is done from the beginning of the geometry to the end, with a number of slices that is set by default or by an user choice. An example of this procedure is shown in the following figure where on the right there is a V tail wing configuration and on the left a fuselage.

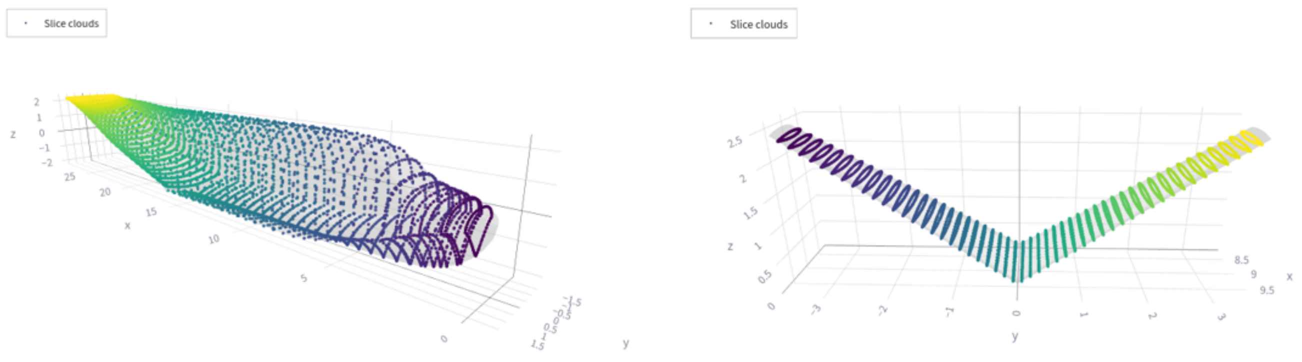


Figure 3.3: Illustration of the slicing procedure applied to different aircraft components.

3.3.1 Computation of sweep and dihedral angles

Before applying the slice filtering and refinement strategy, it is necessary to define how the local sweep and dihedral angles are computed for both the wing and the fuselage. Although the mathematical principle is similar, the geometric reference used for each component differs:

- For the wing, angles are computed from the leading-edge (LE) points.
- For the fuselage, angles are computed from the central reference point of each cross-section.

As shown in the Fig.3.4 where is visualized the reference points for wing and fuselage.

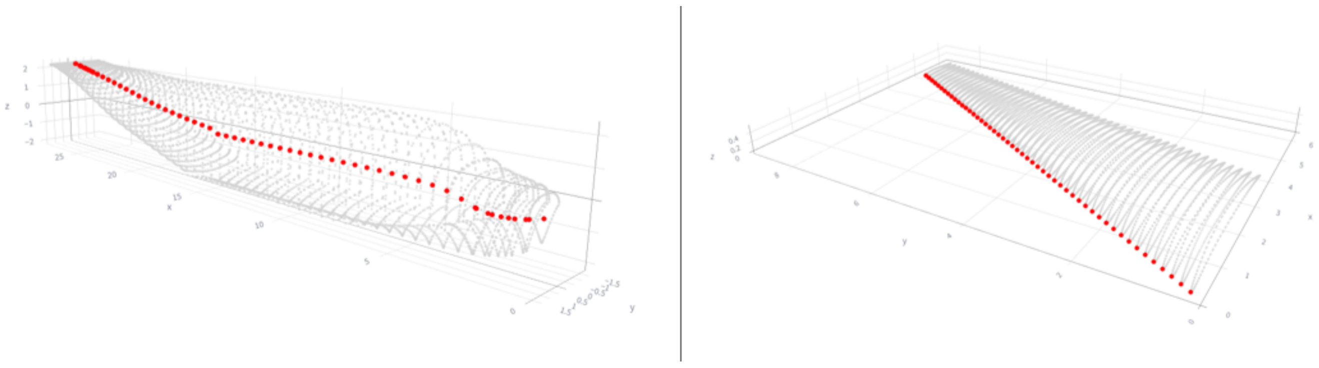


Figure 3.4: Definition of reference points for sweep and dihedral angle computation. For the wing, the leading-edge points of each section are used as reference, while for the fuselage, the central reference point of each cross-section is employed. This distinction reflects the aerodynamic role of the wing and the geometric nature of the fuselage.

This distinction reflects the aerodynamic role of the wing and the geometric nature of the fuselage. Let the reference points of the fuselage sections be:

$$\mathbf{P}_i = (x_i, y_i, z_i),$$

For each pair of consecutive sections, the local direction vector is defined as:

$$\Delta\mathbf{P}_i = \begin{pmatrix} \Delta x_i \\ \Delta y_i \\ \Delta z_i \end{pmatrix} = \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \\ z_{i+1} - z_i \end{pmatrix}.$$

The sweep angle is computed in the horizontal plane (X, Y) as:

$$\Lambda_i = \arctan\left(\frac{\Delta y_i}{\sqrt{\Delta x_i^2 + \Delta z_i^2}}\right).$$

This corresponds to the orientation of the fuselage axis projected onto the horizontal plane. The dihedral angle is computed in the vertical plane (X, Z) as:

$$\Gamma_i = \arctan(\frac{\Delta z_i}{\Delta x_i}).$$

This represents the inclination of the fuselage axis in the vertical direction. For the fuselage, the angles describe the orientation of the centerline of the body. Since the fuselage is primarily aligned along the longitudinal X direction, sweep and dihedral quantify lateral and vertical bending of the centerline. For the wing, the geometry is referenced to the leading-edge points:

$$\mathbf{P}_i = (x_i, y_i, z_i).$$

The local difference vector is:

$$\Delta\mathbf{P}_i = \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \\ z_{i+1} - z_i \end{pmatrix}.$$

Unlike the fuselage, the wing is primarily extended in the spanwise (Y) direction. Therefore, the angle definitions are adapted to reflect aerodynamic conventions. Sweep is computed from the inclination of the leading edge relative to the spanwise direction. It is defined as:

$$\Lambda_i = \arctan\left(\frac{\Delta x_i}{\sqrt{\Delta y_i^2 + \Delta z_i^2}}\right).$$

Dihedral is computed in the (Y, Z) plane as:

$$\Gamma_i = \arctan\left(\frac{\Delta z_i}{\Delta y_i}\right).$$

Remember that CPACS has a section parametrization and once the airfoil/profile is defined i need to know the orientation and the position of the sections between one an other. This is the aim of this step, to compute the local sweep, dihedral angles and orientation that are used to define the CPACS.

3.3.2 Slice filtering

Although both the wing and the fuselage are discretized into sectional slices, their geometric nature is fundamentally different. For this reason, two distinct filtering and interpolation approaches are applied. The wing geometry is primarily defined by variations of sweep angle and dihedral angle along the spanwise direction. In many practical configurations, large regions of the wing exhibit constant sweep and constant dihedral. Retaining all slices inside such constant angle regions would introduce unnecessary discretization without improving geometric fidelity. Let the spanwise stations be defined by y_i, with associated sweep angle Λ_i, dihedral angle Γ_i, and leading-edge position $\mathbf{p}_i$. Two consecutive stations i and $i + 1$ are considered to belong to a constant region if:

$$|\Lambda_{i+1} - \Lambda_i| \leq \varepsilon \quad \text{and} \quad |\Gamma_{i+1} - \Gamma_i| \leq \varepsilon,$$

where ε is a prescribed angular tolerance. In such constant regions, interior slices are removed and only boundary slices are preserved(the first one and the last one). Remember that these slices are eliminated only when there are constant sweep and dihedral region but if there are changes the slicing is preserved. This significantly reduces redundancy while keeping the geometric definition exact. However, is noticed running CFD simulation that in general due to the CPACS section definition the surface shows a peaky behaviour and to fix it, is implemented a feature that add a specific number of slice in the region where there is an angular transition. For a given number n_{insert} of inserted slices, that the user can set, are added slices through a linear interpolation that ensures a smooth evolution across them, avoiding abrupt geometric changes. Therefore, for the wing:

- Constant-angle regions are filtered to remove unnecessary slices.
- Transition regions are refined by interpolation.

In contrast to the wing, the fuselage geometry is characterized by continuous curvature variations, especially in the nose, cabin transition, and tail regions. Even if sweep and dihedral angles remain locally constant, the cross-sectional shape and radius distribution may vary significantly. For this reason, no original fuselage slices are removed. Whenever a transition in sweep or dihedral is detected more slice could be added in order to capture the geometric evoulution. The parameter that control the number of slice is a user setting as for the wing. The value of this parameter should be choosen as a post processing step, because if there are region that needs to be smoother, the user could resolve increasing the number of slices. Adding slice means that the CPACS file will be more heavy, so it must be set in a proper way starting with a low value and increasing it until the user is satisfied with the smoothness of the geometry. As demostrated in the following figure, on the left there is an stl file of a wing with high sweep angle, it is sliced using 50 slices but the result on the upper right visualization shows 5 sections due to the filtering procedure. Also the bottom right figure shows the same wing with a 5 number of slices that are added in every transition region, resulting in a CPACS file with 14 sections but with a smoother geometry.

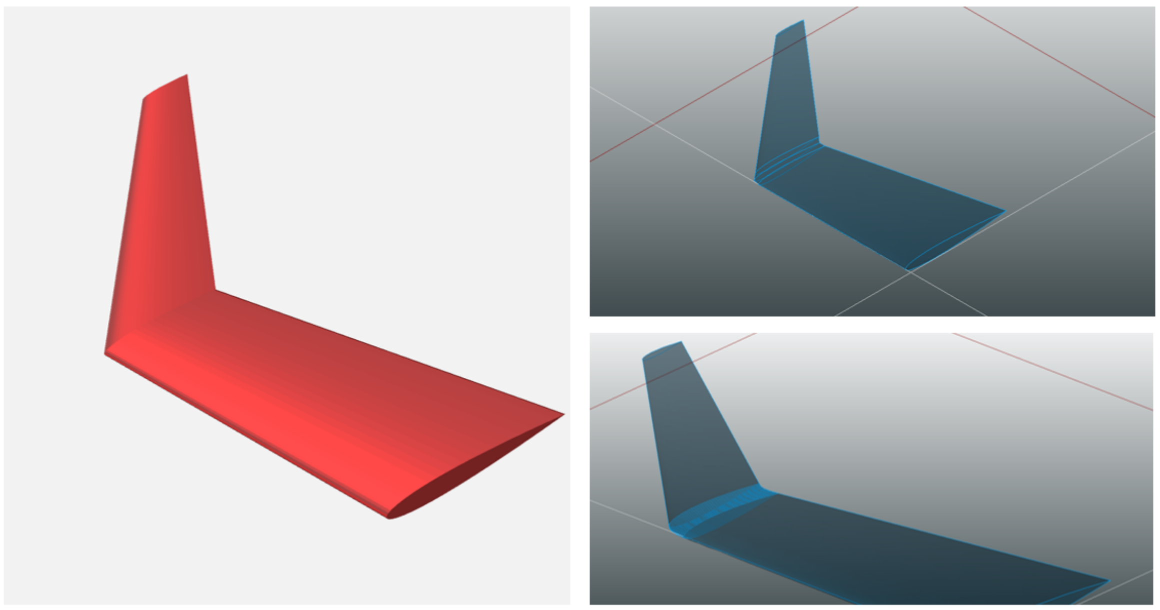


Figure 3.5: Example of the slice filtering and refinement procedure for a wing. Left: STL geometry with high sweep angle. Top-right: initial slicing with 50 planes, where the filtering algorithm retains only 5 sections in regions with constant sweep and dihedral. Bottom-right: refined configuration where additional slices are inserted in transition regions, resulting in 14 sections and a smoother geometric representation in the generated CPACS model.

3.3.3 Rotated slicing strategy for accurate profile extraction

A crucial step in the geometric reconstruction process is the extraction of sectional profiles from the three-dimensional mesh. If slicing is performed using fixed global planes (for example, planes orthogonal to the global Y-axis), the extracted profiles do not correspond to the true geometric cross-sections of the wing or fuselage when sweep or dihedral is present. This leads to distorted airfoils or incorrect fuselage shapes. For this reason, a rotated slicing strategy is introduced. The objective is to align the slicing plane with the *local geometric orientation* of the component. This guarantees that the extracted section corresponds to the physically correct profile. Consider a wing with non-zero sweep and dihedral. If a global plane $Y = \text{const}$ is used the intersection plane is not perpendicular to the local span direction and lead to extract airfoils geometrically skewed. Similarly, for a fuselage with curvature in space, slicing with a fixed global plane produces oblique sections that do not correspond to the true cross-sectional geometry. Therefore, the slicing plane must be constructed *locally*, using the previously computed sweep and dihedral angles. It is important to note that the previous slicing step (non-rotated) is only used to:

- Detect the reference points (leading edge for the wing, central point for the fuselage),
- Compute the local sweep and dihedral angles.

The final geometric extraction is performed using rotated slicing. For each slice i, a reference point $\mathbf{p}_0$ is defined:

- Wing: leading-edge point,
- Fuselage: central section point.

Given the local dihedral angle Γ_i and sweep angle Λ_i, the span direction is constructed. A rotation matrix around the X-axis is defined:

$$R_x(a) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos a & -\sin a \\ 0 & \sin a & \cos a \end{pmatrix}.$$

The nominal spanwise direction in the non rotated configuration is $e_y = (0, 1, 0)$ for the wing and $e_x = (1, 0, 0)$ for the fuselage. The rotated span direction becomes:

$$\mathbf{e}_{\text{span}} = R_x(a)\mathbf{e}_y.$$

Finally, the vector is normalized:

$$\mathbf{e}_{\text{span}} \leftarrow \frac{\mathbf{e}_{\text{span}}}{||\mathbf{e}_{\text{span}}||}.$$

This vector defines the normal direction of the slicing plane. The slicing plane is defined by:

- A point $\mathbf{p}_0$ (reference point of the slice),
- A normal vector $\mathbf{e}_{\text{span}}$.

The slicing procedure is the same of what is describe unsing a plane normal to the slicing direction and once we detect the points that are intersecting the plane, the airfoil/profile that comes is the correct one, because the plane is locally aligned with the geometry. Without this rotation step, the extracted airfoils and fuselage sections would not represent the true geometry, especially in configurations with significant sweep or dihedral. Therefore, rotated slicing is not merely a numerical refinement; it is a geometrically necessary operation to ensure that the reconstructed profiles correspond to the real physical sections of the aircraft components. Imagine that STL2CPACS is applied to a wing with a winglet with an high dihedral angle; what is shown in the following figure is the difference between a non-rotated slicing and a rotated slicing. The non-rotated slicing produces skewed airfoils that do not represent the true geometry, while the rotated slicing correctly captures the airfoil shape.

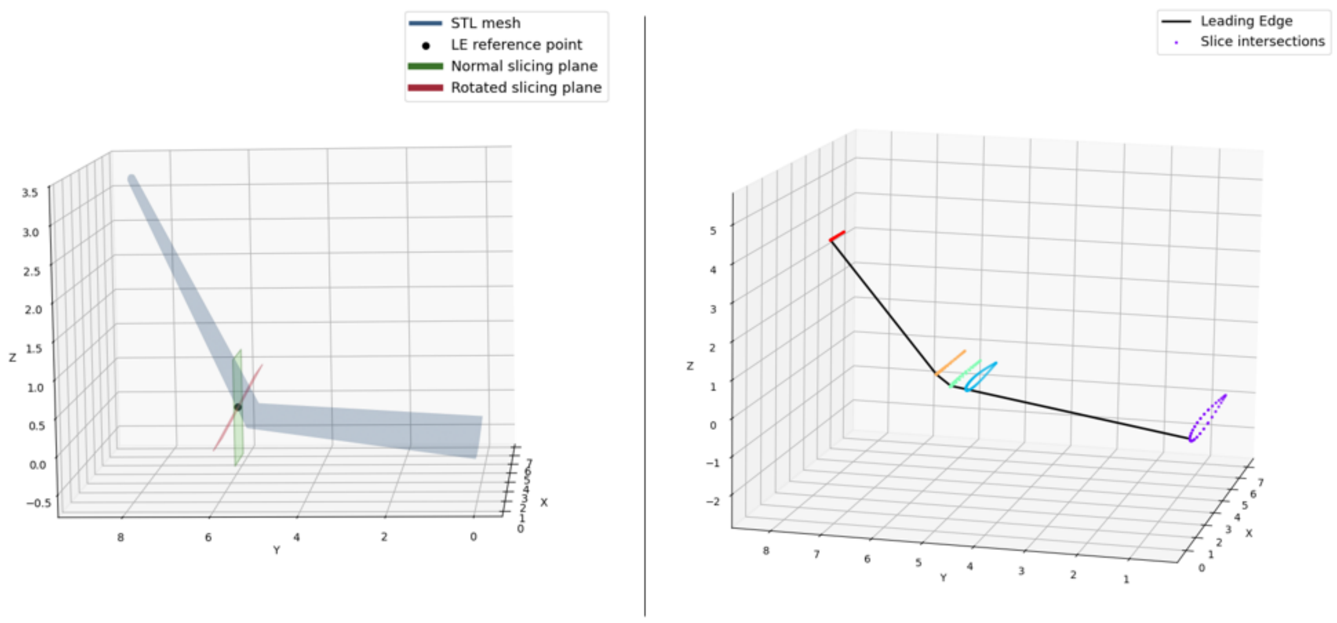


Figure 3.6: Illustration of the rotated slicing strategy used to correctly extract section profiles. Left: comparison between a non-rotated slicing plane and a rotated slicing plane locally aligned with the wing geometry. Using a fixed plane normal to the global slicing direction produces skewed airfoil sections, while rotating the plane according to the local sweep and dihedral angles yields the correct airfoil shape. Right: example of rotated slicing applied along the entire STL geometry, where each slicing plane is locally oriented to follow the wing configuration.

3.4 Airfoil extraction methodology

The automated reconstruction of two-dimensional airfoil-like profiles from unstructured STL surface data is a central step within the `STL2CPACS` module. Depending on the aircraft component—wing or fuselage—the slicing plane and the treatment of the cross-section differ, but both cases share the same underlying philosophy: starting from a cloud of triangulated points, extract a clean, ordered, and resampled closed profile that can be stored as a CPACS point list. The core workflow is done by the `extract_airfoil_surface_local` function, which is called once per section (i.e., per wing or fuselage slice) and returns a normalized 2D profile along with its characteristic length (chord for wings, width and height for fuselages). This function performs three main tasks:

- Extrema detection for leading/ trailing edges (wing) or top/bottom points (fuselage);
- Normalization of the coordinates so that the section is centered and scaled with respect to its reference length;

- Splitting of the point cloud into upper/lower (wing) or left/right (fuselage) surfaces for subsequent resample

In the following sections, the wing and fuselage extraction logics are described separately, highlighting similarities and differences in the treatment of the profiles. For both wings and fuselages, the extraction pipeline starts from a local STL point cloud corresponding to a slice of the component, defined by a reference point P_0 and a normal direction n. The cloud is a 2D plane locally attached to the component, the 2D coordinates are cleaned, normalized and split into two sides depending on the CPACS convetion for wing or fuselage. A final spline resampling step produces a smooth and equi-spaced polyline that can be stored as a CPACS file.

3.4.1 Wing airfoil extraction

The leading and trailing edges are identified by searching for the minimal and maximal x-coordinates, respectively, and the chord length is computed as the distance between them.Once the chord is determined, the coordinates are normalized so that the leading edge lies at $x = 0$ and the trailing edge at $x = 1$, while the z-coordinates are scaled by the same chord length. This normalization ensures that the resulting airfoil is invariant to the absolute size of the component and can be stored in CPACS as a dimensionless profile. The normalized 2D point set is then passed to a splitting function, which performs the critical separation between upper and lower surfaces, according the CPACS ordering for airfoil shape. This routine first applies a preprocessing to remove duplicate or near-duplicate points. The function then constructs a coarse camber line by binning the points along the chordwise direction. So, x is divided into a n numbers of bins and for each bin is computed the mean between the topmost and bottommost points. Doing it for each bin a coarse camber line is obtained. This is done in order to separate the upper and lower surface checking if each point is above or below the camber line. Due to the numerical noise that is present into every procedure and to the fact that the number of points that generate each airfoil could change significantly depending how the original stl is refine; a re sampling step is performed. It's better to visualize what this procedure does, so in the following figure is shown a wing section with the original point cloud and the camber line that is used to separate the upper and lower surface. This is a Five Digit NACA profile high cambered with a sharp trailing edge. The shape is not simple to detect but STL2CPACS shows robustness because the coarse camber line is able to separate the upper and lower surface even in this complex case. The resample is useful to eliminate numerical noise and produce a smooth and equi-spaced profile.

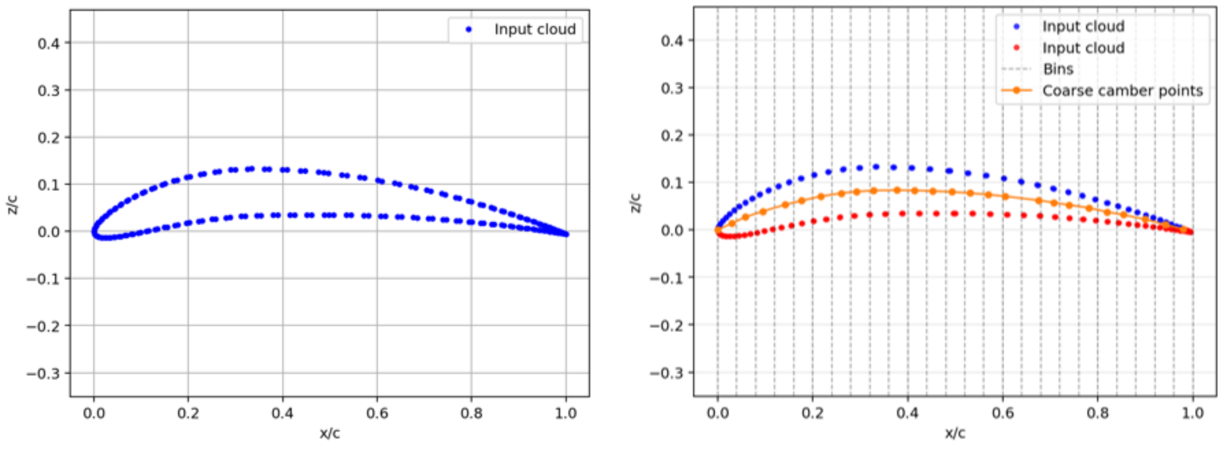


Figure 3.7: Schematic representation of the wing airfoil extraction procedure.

Could happen that the number of points of the original point cloud is not enough to capture the airfoil shape or there are region, as the trailing edge, that has a high curvature with sharp features. In this case the splitting and re-sample procedure is not able to properly generate a correct ordered airfoil for the CPACS convention. To fix this problems the user should play with the number of bins and a parameters called TE_{CUT} where the first one is how the camber line is discretized over the chord. So, more bins needs to have a cloud of points very refine and could capture the airfoil shape better; if the user decrese this number the camber line will be corser and could not be able to split the upper surface to the lower surface. The second parameter shoud be seen as a sensibility about the sharpness of the trailing edge, so if the airfoil has a very sharp trailing edge, the user should increase this parameter, it means that the shape from $1 - TE_{CUT}$ is detected and the last part that correspond to the most difficult region is approximate with a straight line. These post processing setting are useful because the algorithm is working with STL file that can be coarse or not well defined and having these tools to avoid oscillations inside the CPACS file is very useful.So, playing with these parameters is suggest, in the case that the CPACS shows oscillation in the geometry once STL2CPACS has finished.

3.4.2 Fuselage cross-section extraction

For the fuselage, STL2CPACS extracts planar cross-sections orthogonal to the fuselage axis and represents each section in the local (y, z) plane, where y is lateral and z is vertical according to the CPACS axis convention. Given the raw intersection point cloud, a robust detection of the left, right, top, and bottom extrema is then performed. The lateral extent is characterized by the minimum and maximum y values, while the vertical extent is obtained by searching for top and bottom points close to the fuselage symmetry plane. To reduce sensitivity to local noise, the top and bottom locations are determined by averaging all points near the vertical extrema within a small band around the estimated maximum and minimum z values. The cross-section is subsequently normalized in both directions. The lateral coordinate y is scaled so that the left and right sides lie at $y = -0.5$ and $y = +0.5$, respectively, while the vertical coordinate z is scaled such that the bottom and top points are located at $z = -0.5$ and $z = +0.5$. After this step, the profile is centered at the origin and confined to a unit bounding box, which makes different sections directly comparable and independent of the absolute fuselage size at that station. The same normalization factors are stored separately so that the dimensional width and height of each section can be recovered. Before splitting the section into left and right branches, a filtering step is applied to the normalized (y, z) points. Points that are duplicate or closer than a given tolerance are removed and this significantly stabilizes the subsequent geometric operations. This pre-processing is especially important in regions where the triangulated fuselage mesh produces many nearly coincident intersection points. The classification into right and left sides is performed by constructing a coarse centerline with the same wing's logic. The normalized points are first sorted with respect to z, and the interior (non-extreme) points are partitioned into n vertical bins. For each bin, the most lateral points on the positive and negative y sides are identified, and the centerline point is defined as the mean between these two extrema. By repeating this procedure along all bins, a discrete centerline is obtained and then interpolated over the full vertical range. Each interior point is finally classified as belonging to the right or left side depending on whether it lies above or below the local centerline in the (y, z) plane, while the top and bottom extrema are treated separately to preserve the correct CPACS ordering.Once the raw left and right subsets are available, STL2CPACS regularizes the profile through a resampling step. The final CPACS profile is assembled with the correct ordering. As for the wing airfoil extraction, the number of bins used for the centerline construction and the total number of resampling points are exposed as user parameters. Increasing the number of bins refines the centerline description and typically improves the separation between left and right sides, at the cost of requiring a denser and cleaner input point cloud. Conversely, a too coarse binning may fail to capture local shape variations, especially in cross-sections with pronounced bulges or kinks. Therefore, the user may need to adjust these parameters depending on the STL mesh quality and the complexity of the fuselage geometry at each station. To visualize the fuselage cross-section extraction, consider the following figure where a not common shape is analyzed. Although the curvature and the complexity of this fuselage profile, STL2CPACS is able to catch the shape and give a ready to use CPACS profile.

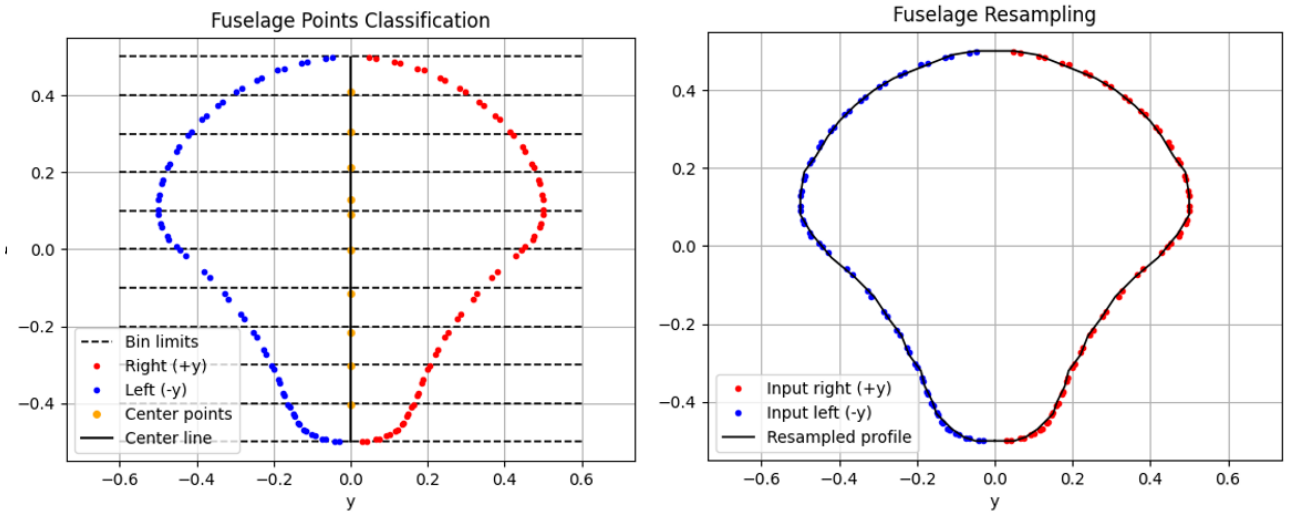


Figure 3.8: Schematic representation of the fuselage cross-section extraction procedure.

3.5 Generate the CPACS file

Once the slicing and the detection of the geometric angles such as sweep and dihedral are completed, `STL2CPACS` possesses all the necessary information to generate the CPACS description. Owing to the flexibility inherited from the integration with OpenVSP, the script `exportcpacs.py` can reconstruct a fully parametric CPACS file component-by-component. For both the wing and the fuselage, `STL2CPACS` follows a structured procedure to rebuild the geometry slice by slice, converting local mesh information into CPACS-compatible transformations and sections. For the wing, the process starts with defining the general transformation block of the wing, which determines its position and orientation within the CPACS global coordinate system:

- `X_Rot`: defines the rotation of the wing; a rotation of 90° around the x-axis is applied if the component corresponds to a vertical tail.
- `X_Trasl`: sets the global translation, corresponding to the first leading-edge point (`le_pts[0]`).

The algorithm then iterates over each slicing plane along the span direction. To summarize the logic, for each station it computes the local dihedral and sweep angles and performs the rotational slicing. This operation yields a locally oriented point cloud, which is converted into a 2D airfoil. Each extracted section is then stored into a dictionary structure. The first section defines the root geometry with a zero span, while subsequent sections compute their local span as it is done in the VSP2CPACS module:

$$\text{Span}_i = \frac{|y_i - y_{i-1}|}{\cos(\text{dihedral}_i)}$$

Scaling factors (`x_scal`, `z_scal`) are proportional to the local chord length, ensuring that the-original surface is accurately captured. The complete set of sections allows `exportcpacs.py` to reconstruct the full parametric wing within the CPACS tree. A similar methodology applies to the fuselage, with slicing performed along the longitudinal direction. The initial transformation part of the dictionary defines the main parameters of the component, including its absolute orientation, reference system, and total length. For each fuselage slice, the script extracts the cross-sectional cloud orthogonal to the main axis and derives its shape and scaling. Each section stores the local offsets (`y_trasl`, `z_trasl`) with respect to the previous slice center, maintaining a continuous centerline along the fuselage body. Through this systematic procedure, `STL2CPACS` transforms unstructured 3D mesh data into a fully parametric CPACS representation. The final dictionaries (`Wing_Dict` and `Fuse_Dict`) encapsulate all necessary geometric and transformation information, which `exportcpacs.py` subsequently writes into the final CPACS file. But,before generating the CPACS model, it is necessary to determine which aircraft component the input STL geometry represents. This step is essential because each component (wing, vertical tail, or fuselage) follows a different geometric parametrization pipeline in `STL2CPACS`. An inaccurate classification would lead to a wrongly interpreted structure in the CPACS hierarchy, ultimately producing an inconsistent or unusable model.

3.5.1 Automatic component detection

Manually specifying the component type for every STL geometry is not desirable from an automation perspective. To ensure robustness and full autonomy, `STL2CPACS` integrates a classification algorithm based on **Principal Component Analysis (PCA)**, a fundamental technique for identifying dominant spatial directions in multidimensional datasets. PCA provides a mathematical framework for extracting the main geometric trends of a surface based solely on its vertex distribution, without requiring any prior knowledge of its orientation or origin. Principal Component Analysis is a linear orthogonal transformation that projects a dataset into a new coordinate system defined by the directions of maximum variance. Given a set of n three-dimensional points:

$$\mathbf{X} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix},$$

the algorithm proceeds as follows:

1. The centroid of the point cloud is subtracted to eliminate the influence of absolute position:
$$\mathbf{X}_c = \mathbf{X} - \bar{\mathbf{x}}, \quad \text{with } \bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i.$$

2. The covariance matrix of the centered data is computed:
$$\mathbf{C} = \frac{1}{n-1}\mathbf{X}_c^{\mathrm{T}}\mathbf{X}_c.$$

3. Solving the eigenvalue problem
$$\mathbf{C}\mathbf{v}_i = \lambda_i \mathbf{v}_i$$
yields the eigenvectors $\mathbf{v}_i$ (principal directions) and eigenvalues λ_i (variance magnitudes).

The eigenvectors define a new orthogonal coordinate system intrinsic to the geometry. Importantly, these principal directions are generally *not aligned* with the global X, Y, or Z axes. Instead, they depend exclusively on the spatial distribution of the vertices. The eigenvalues quantify the variance along these intrinsic directions. To interpret the geometric structure consistently, the eigenvalues are sorted in descending order:

$$\lambda_1 \geq \lambda_2 \geq \lambda_3.$$

This guarantees that:

- λ_1 corresponds to the direction of maximum geometric development (PC1),
- λ_2 corresponds to the second most significant spread (PC2),
- λ_3 corresponds to the smallest spread (PC3).

Sorting is essential because the elongation measure must refer to the dominant principal direction, independent of the global orientation of the STL model. The implemented function, `cpacs_component_detection()`, automates this reasoning. The core procedure comprises the following stages:

1. **Mesh loading and preprocessing.** The STL file is loaded, and its vertices are reshaped into a list of 3D coordinates. Duplicate points are removed to reduce numerical noise.
2. **Centering of the geometry.** The centroid of the point cloud is subtracted to ensure that PCA reflects only the shape distribution.
3. **Principal Component Analysis (PCA).** The covariance matrix is computed, and its eigenvalues and eigenvectors are obtained and sorted in descending order of eigenvalue magnitude.
4. **Axis Alignment Analysis.** After normalization, the first eigenvector (PC1):

$$\mathbf{v}_1 = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T$$

represents the direction of maximum geometric variance. Since $\mathbf{v}_1$ is a unit vector, its components correspond to the direction cosines with respect to the global axes. The alignment measures are defined as

$$\alpha_x = |v_x|, \qquad \alpha_y = |v_y|, \qquad \alpha_z = |v_z|.$$

Each value represents the cosine of the angle between the dominant geometric direction and the corresponding global axis. The absolute value removes sign ambiguity. Large values of α_i indicate strong alignment with axis i.

5. **Variance Ratio (Elongation Measure).**

   To quantify geometric anisotropy, the elongation ratio is defined using the *sorted* eigenvalues:

$$r = \frac{\lambda_1}{\lambda_2}.$$

   Because λ_1 and λ_2 correspond to the two largest intrinsic principal variances, this ratio measures how dominant the primary geometric direction is relative to the second. For strongly elongated bodies such as fuselages:

$$\lambda_1 \gg \lambda_2 \approx \lambda_3,$$

   resulting in $r > 1.2$. In contrast, lifting surfaces (wings and vertical tails) exhibit a more two–dimensional development, leading to smaller values of r.

6. **Classification Criteria.**

   The component type is determined by combining orientation and elongation:

   - If $\alpha_x > 0.75$ and $r > 1.2 \rightarrow$ **Fuselage**
   - If $\alpha_z > 0.6 \rightarrow$ **Vertical tail**
   - If $\alpha_y > 0.6 \rightarrow$ **Wing**

   If none of these thresholds is strictly satisfied, the component is assigned according to the largest alignment value:

$$\text{Component} = \arg\max(\alpha_x, \alpha_y, \alpha_z).$$

The PCA-based classification provides a robust and computationally efficient method for automatic component recognition. Since the procedure depends exclusively on vertex distribution, it is independent of mesh resolution and absolute positioning. The elongation ratio evaluates intrinsic shape properties, while the direction cosines determine global orientation. Every geometry processed by `STL2CPACS` first passes through this PCA-based detection stage. Once the component is identified, the corresponding reconstruction routine is called. As illustrated in Fig.3.9, the wing is identified by its dominant alignment with the global Y-axis, while the fuselage is recognized by its strong elongation along the global X-axis. Here, PC1 denotes the principal component associated with λ_1, whereas PC2 and PC3 correspond to the remaining orthogonal principal directions. This automatic classification ensures that the subsequent CPACS reconstruction process applies the appropriate geometric logic without requiring manual user intervention.

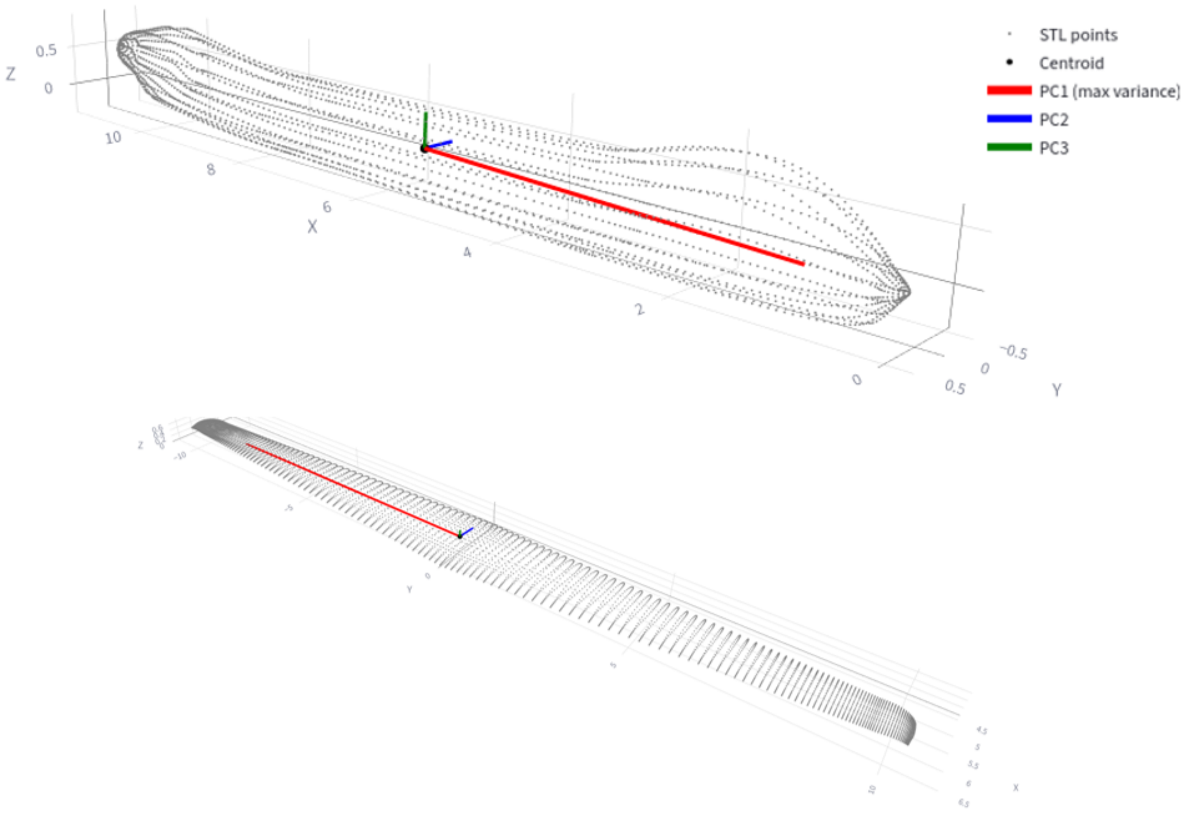


Figure 3.9: Illustration of the PCA-based component detection. The principal direction (PC1) identifies the dominant geometric axis, whose alignment with the global reference frame enables automatic classification of wing and fuselage geometries.

3.5.2 Automatic STL component splitting for Full-Aircraft models

The automatic component detection mechanism described previously assumes that each input STL file represents a single geometric entity. In practical workflows, however, users often possess a single STL file containing the entire aircraft configuration, where all components are merged into one mesh. To improve usability and avoid requiring manual preprocessing, an auxiliary splitting procedure has been implemented to automatically separate such full-aircraft STL models into independent geometric parts. The splitting strategy relies on the observation that most CAD-derived STL assemblies consist of multiple triangulated surfaces that are geometrically disconnected. Even when exported as a single file, different aircraft components typically do not share vertices or edges with each other. Consequently, the mesh can be interpreted as a graph in which triangles are nodes and geometric adjacency defines the connectivity between them. By identifying disconnected subgraphs within this structure, it becomes possible to isolate the different geometric components of the aircraft automatically. The first stage of the procedure consists of loading the STL geometry and converting it into a structured triangle array. Independently of the original STL encoding (ASCII or binary), the geometry is internally represented as an array of size $N \times 3 \times 3$, where N denotes the number of triangles, each triangle contains three vertices, and each vertex is described by its Cartesian coordinates (x, y, z). Once the triangulated mesh is available, the algorithm constructs a triangle adjacency graph. Two triangles are considered adjacent if they share at least one common vertex. In practice, numerical inaccuracies in STL files can cause vertices that should coincide, differ by small numerical deviations. To avoid disconnections caused by such rounding errors, vertex coordinates are first rounded using a small tolerance. This operation effectively merges vertices whose coordinates differ by less than a predefined threshold, ensuring that triangles that should be connected remain linked. So, a mapping between vertices and incident triangles is created. For each vertex, the algorithm records all triangles that reference that vertex. If multiple triangles share the same quantized vertex, they are connected. The result is a connectivity structure in which each triangle stores the list of neighboring triangles that share at least one vertex. After constructing the graph, the mesh is partitioned through a connected-component search. The algorithm traverses the graph using a first exploration strategy, beginning from an unvisited triangle and recursively visiting all neighboring triangles that belong to the same connected region. All triangles discovered during this step generate one geometric component. The procedure is repeated until every triangle in the mesh has been assigned to a component. Each connected group therefore represents a continuous portion of the original geometry. In many cases this vertex-based connectivity analysis is sufficient to identify distinct aircraft components such as wings, fuselage, tail surfaces, or nacelles. However, certain STL exports may contain coincident interfaces or degenerate connections where triangles belonging to different components share isolated vertices or edges. Finally, each component is exported as an independent STL file. An example of this process is shown below, where a complete Sea-Plane, with all its components, is split into individual parts.

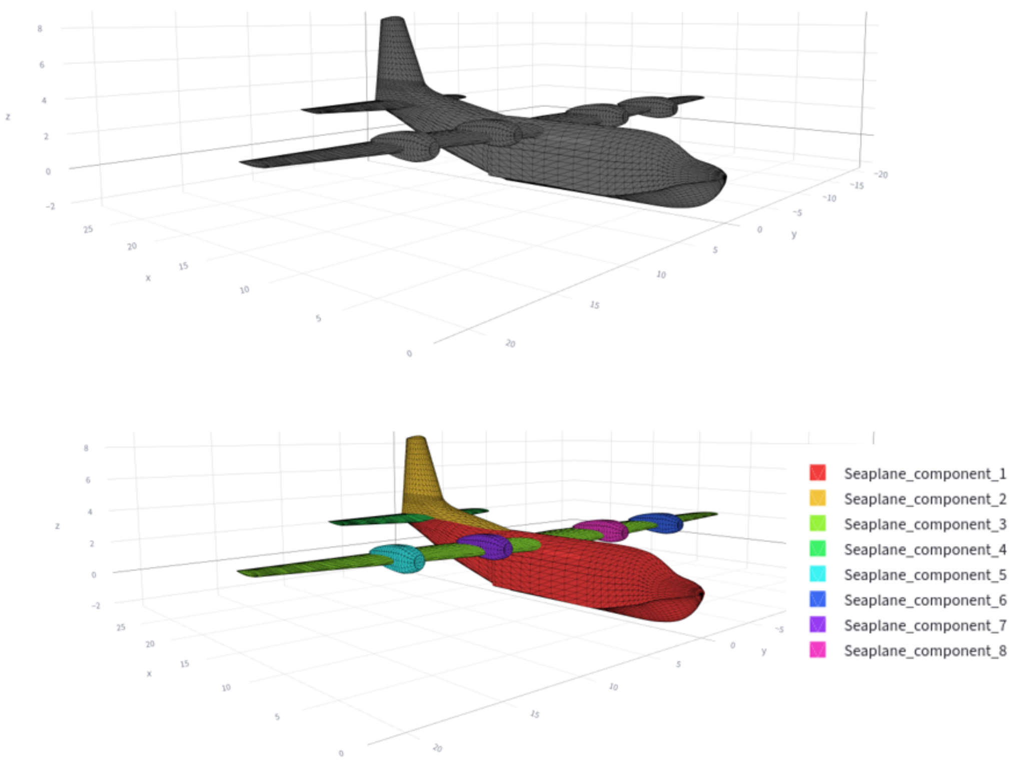


Figure 3.10: Automatic splitting of a full Sea-Plane model into geometrically disconnected components using triangle connectivity analysis.

The resulting files are written using the binary STL format and stored in a dedicated output directory. The components are visualized, named and ordered from the biggest to the smallest file. This ordering facilitates manual verification and simplifies subsequent processing steps. The resulting workflow significantly improves the user experience within the STL2CPACS module. Instead of requiring multiple STL files as input, the user may provide a single full-aircraft geometry. The splitter, automatically decomposes the model into independent geometric components that can then be processed individually by the STL2CPACS conversion pipeline. This capability removes a preprocessing step that could force the user to work outside the CEASIOMpy environment and allows the geometry reconstruction procedure to be applied directly to complex aircraft assemblies.

3.6 GUI and user interaction

To improve usability and simplify the configuration of the STL2CPACS module, a graphical user interface (GUI) has been developed using *Streamlit*, a Python framework designed for the creation of interactive web applications [17]. The GUI provides an intuitive environment where users can execute the STL-to-CPACS conversion workflow while visualizing intermediate results and adjusting relevant parameters. The interface integrates the main functionalities described in the previous sections, including automatic component detection, geometry slicing, and airfoil extraction. The objective of the GUI is to guide the user through the different stages of the reconstruction process while maintaining flexibility for advanced parameter tuning when necessary. The main features of the STL2CPACS GUI are summarized below:

- **File Upload:** Users can upload STL files directly through the interface. Both ASCII and binary STL formats are supported. After uploading, the GUI provides feedback on the file status and size to ensure that the geometry has been correctly loaded.
- **3D Visualization:** Once the STL file is imported, the geometry is rendered in an interactive 3D viewer. This allows the user to visually inspect the model before processing and verify that the geometry has been correctly interpreted by the module. A wireframe visualization mode is also available to better highlight the underlying mesh structure.
- **Geometry Splitting and Component Detection:** The GUI includes a dedicated tool for automatically splitting the STL geometry into individual components. After the splitting operation, each component is displayed with a different color to facilitate visual identification. The automatic component detection routine is then applied, and the detected component type is shown to the user.
- **Slicing Configuration:** The interface provides controls for configuring the slicing procedure used to reconstruct the CPACS geometry. Parameters such as the number of slices, filtering thresholds for slice reduction, and airfoil extraction settings can be adjusted. Default parameter values are provided to ensure robust reconstruction for most geometries, while still allowing user customization when more complex shapes are processed.

- **CPACS Visualization:** After the conversion process is completed, the generated CPACS model can be visualized directly in the GUI. This allows users to immediately inspect the reconstructed geometry and verify the quality of the conversion.

Figure 3.11 shows an example of the STL visualization within the GUI, where the user can inspect the imported model and activate the wireframe mode to better observe the mesh structure.

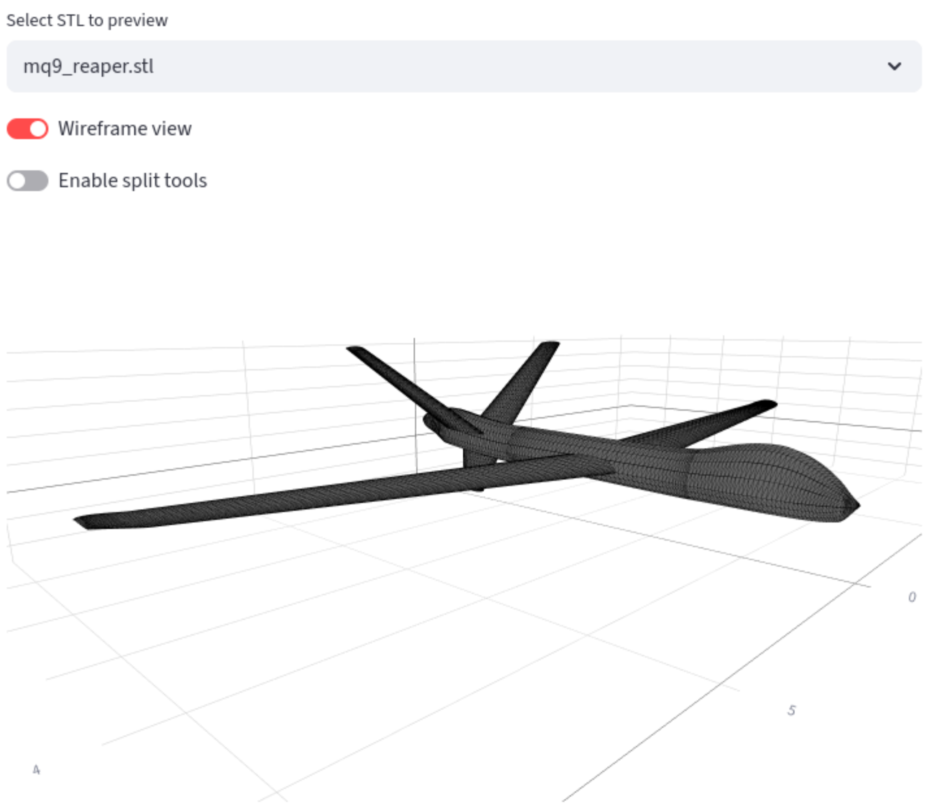


Figure 3.11: Wireframe visualization of the STL model within the GUI.

If the uploaded STL file contains the complete aircraft geometry, the user can activate the splitting tool to automatically separate the mesh into its individual components. Within a few seconds, the GUI displays the detected components, each rendered with a different color to facilitate identification, as shown in Figure 3.12.

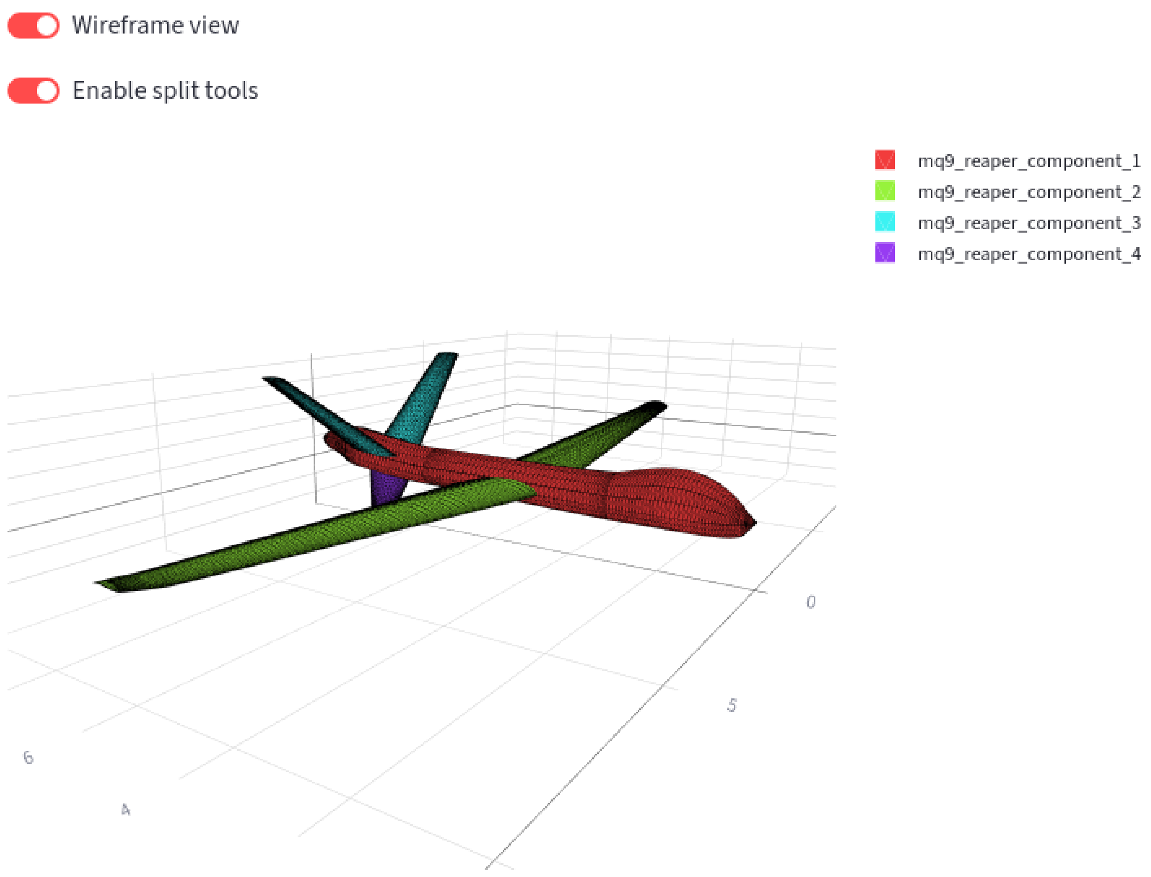
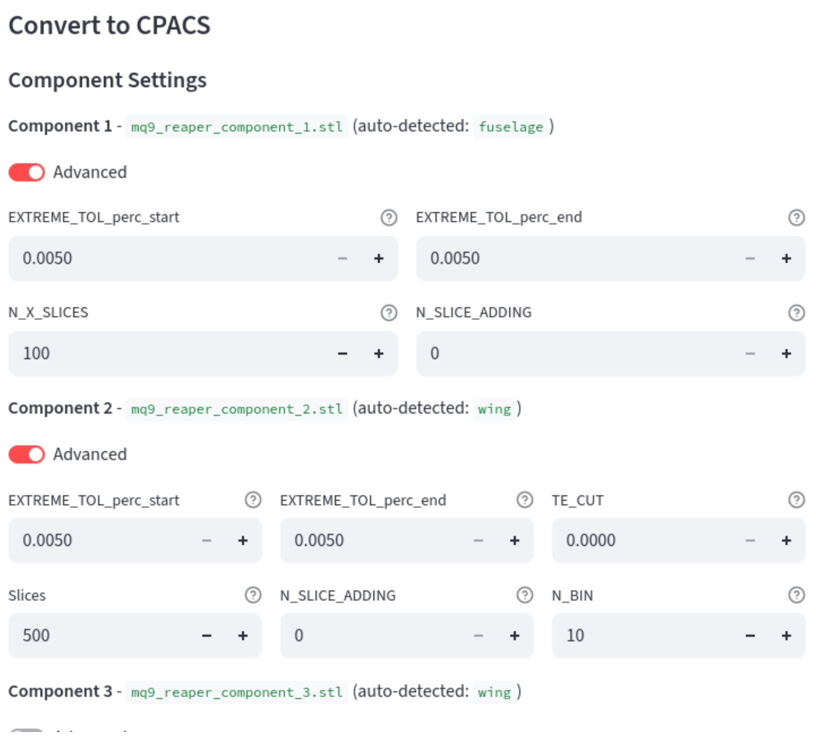


Figure 3.12: Automatic geometry splitting and component visualization.

Further down in the interface, a dedicated configuration panel allows the user to adjust the parameters used for slicing and airfoil extraction. These parameters include the number of slicing planes, filtering thresholds for slice reduction, and additional settings related to the airfoil reconstruction procedure. Default values are provided to ensure a reliable reconstruction for most geometries, while still allowing advanced users to refine the parameters when necessary. Figure 3.13 shows the advanced configuration panel. The parameters correspond to those described previously for wing and fuselage reconstruction, including the number of slices, the trailing-edge cut parameter (*TE_cut*), and the number of bins used to improve airfoil profile detection. As is shown, two additional parameters are introduced for both the wing and the fuselage to control the start and end positions of the slicing procedure. These parameters define small fractional offsets relative to the total span (for the wing) or length (for the fuselage), allowing the slicing process to begin slightly after the geometric root and to terminate slightly before the geometric tip. The motivation for introducing these offsets is primarily robustness. In practical configurations, such as wings equipped with winglets, the geometric tip region often exhibits strong curvature changes or non-planar transitions. If slicing is performed exactly up to the outermost boundary, the detected cross-sections may be distorted or may not correspond to a clean airfoil shape. This is particularly critical during the first slicing step, where the algorithm must identify and parameterize the airfoil geometry. Reducing the slicing range, the algorithm avoids problematic edge regions and instead operates in geometrically well-defined sections. Therefore, these two settings provide a controlled way to fine-tune the effective slicing domain. In certain configurations, adjusting these parameters is necessary to avoid unstable intersections..



Convert to CPACS

Component Settings

Component 1 - `mq9_reaper_component_1.stl` (auto-detected: `fuselage`)

Advanced

EXTREME_TOL_perc_start	EXTREME_TOL_perc_end
0.0050	0.0050
N_X_SLICES	N_SLICE_ADDING
100	0

Component 2 - `mq9_reaper_component_2.stl` (auto-detected: `wing`)

Advanced

EXTREME_TOL_perc_start	EXTREME_TOL_perc_end	TE_CUT
0.0050	0.0050	0.0000
Slices	N_SLICE_ADDING	N_BIN
500	0	10

Component 3 - `mq9_reaper_component_3.stl` (auto-detected: `wing`)

Figure 3.13: Advanced slicing and reconstruction parameters available in the GUI.

Once the parameters are defined, the user can start the conversion process by activating the *Convert to CPACS* button. The STL2CPACS module then executes the reconstruction pipeline. At the end of the process, the generated CPACS file is displayed within the GUI, allowing the user to visually inspect the reconstructed aircraft model.

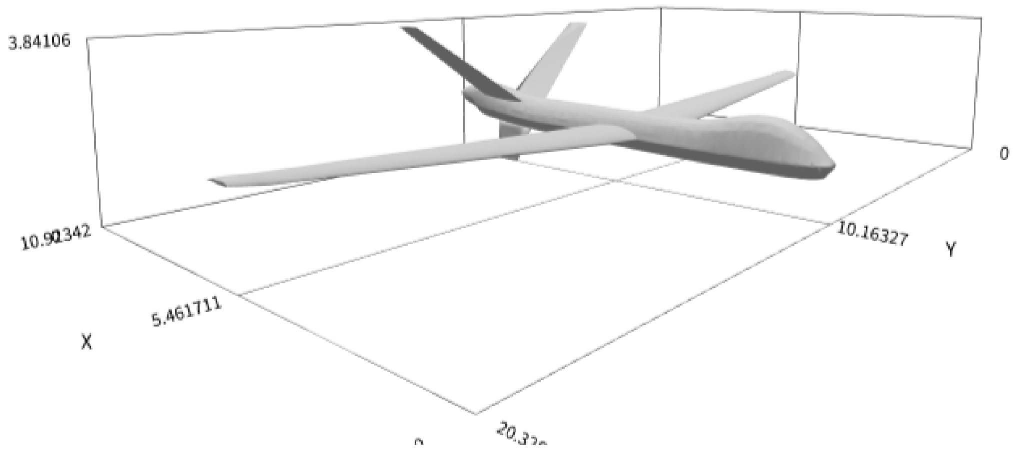


Figure 3.14: Visualization of the generated CPACS geometry within the GUI.

Conclusions

The modules developed in this thesis significantly expand the capabilities of CEASIOMpy from both a user–experience and a methodological perspective. Before this work, introducing a new configuration into the framework typically required building or editing a CPACS file by hand, a process that was time consuming, error prone, and demanding in terms of specific format knowledge. With the introduction of two new access points based on OpenVSP projects and STL geometries, users can now enter the CEASIOMpy environment either from a parametric conceptual model or from an existing triangulated surface, eliminating much of the manual effort previously required. These two access paths are expected to attract a broader user base. On one side, users already familiar with OpenVSP can exploit their existing workflow for parametric geometry definition and transfer models directly into CEASIOMpy, without needing to learn CPACS in detail. On the other side, users who have access only to STL representations—such as meshes from external CAD, legacy datasets, or public repositories—can still participate in CPACS based analyses. The possibility to obtain ready CPACS geometries from widely available formats makes CEASIOMpy accessible to students and researchers who would otherwise be excluded by the complexity of manual CPACS modelling. In practical terms, this integration reduces the time and expertise needed to bring new geometries into CEASIOMpy and encourages the exploration of a wider variety of configurations, including unconventional layouts. Parametric models created in OpenVSP can be iterated quickly and pushed downstream into the analysis chain, while STL reconstructions allow the reuse of industrial or research geometries that were previously disconnected from CPACS workflows. This dual approach not only simplifies geometry creation, but also turns the modules themselves into effective test drivers for CEASIOMpy as a whole, since they can be exercised on a broad set of cases and complexity levels. In the short term, these capabilities increase productivity for individual users and make it easier to perform studies that require many geometry variants. In the longer term, they support larger collaborative projects, where heterogeneous geometries and aerodynamic analyses can be shared and compared across the different modules of CEASIOMpy. Several developments naturally follow from this work. A first direction concerns robustness and coverage, including extending the set of supported components and configurations, improving the behaviour of the tools for highly unconventional layouts, and increasing tolerance to imperfect or incomplete input data. A second direction is improve the integration with mesh generation, fluid dynamics analysis and optimisation, in order that the new geometry can access easily the various analysis tools.

Bibliography

[1] Maierl R., Liersch C. M., Deinert S., Kleinert J., Alder M., Moerland E., 2024, Application of CPACS in Military Aircraft Design, Deutscher Luft- und Raumfahrtkongress 2024, Hamburg (DE).

[2] Jungo A., Zhang M., Vos J. B., Rizzi A., 2016, Benchmarking New CEASIOM with CPACS Adoption for Aerodynamic Analysis and Flight Simulation, Proceedings of the Aerospace Europe Conference.

[3] Alder M., Moerland E., Jepsen J., Nagel B., 2020, Recent Advances in Establishing a Common Language for Aircraft Design with CPACS, Aerospace Europe Conference 2020, Bordeaux (FR).

[4] McDonald R. A., 2016, Advanced Modeling in OpenVSP, Proceedings of the 16th AIAA Aviation Technology, Integration, and Operations Conference, Washington, D.C. (USA).

[5] Yun H., Kim E., Kim D. M., Park H. W., Jun M. B.-G., 2022, Machine Learning for Object Recognition in Manufacturing Applications, International Journal of Precision Engineering and Manufacturing

[6] Anderson J. D., 2017, Fundamentals of Aerodynamics, McGraw-Hill Education, New York.

[7] Rizzi A., Oppelstrup J., 2021, Aircraft Aerodynamic Design with Computational Software, Cambridge University Press, Cambridge.

[8] INDIGO Consortium, 2023, Digital Model Definition in CPACS Format of the Selected Aircraft Configuration, INDIGO Project Deliverable Report.

[9] Nagel B., 2019, CPACS – Common Parametric Aircraft Configuration Schema, German Aerospace Center (DLR), Braunschweig (DE). Available at: www.cpacs.de

[10] CEASIOMpy, CEASIOMpy – Conceptual Aircraft Design Software, Available at: https://github.com/cfsengineering/CEASIOMpy

[11] CEASIOMpy Cloud, Web-based Aircraft Design Environment, Available at: https://www.ceasiompy.com

[12] NASA OpenVSP, OpenVSP – Open Vehicle Sketch Pad, Available at: https://openvsp.org/

[13] SU2 Foundation, SU2 – Open-Source CFD Suite, Available at: https://su2code.github.io/

[14] Pyavl, pyAVL Interface to Athena Vortex Lattice, Available at: https://github.com/pankajp/pyavl

[15] STL, STL File Format, Available at: https://en.wikipedia.org/wiki/STL_(file_format)

[16] Bottaro A., Lecture Notes on Aerodynamics, University of Genoa, Available at: http://www.dicat.unige.it/bottaro/index.html

[17] Streamlit Inc., Streamlit – Open-Source Python App Framework, Available at: https://github.com/streamlit/streamlit

[18] Airinnova, Airinnova – Aircraft Design Services, Stockholm (SE), Available at: https://airinnova.se/

[19] CFS Engineering SA, CFS Engineering – Multidisciplinary Simulation Services, Lausanne (CH), Available at: https://cfse.ch/