ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

CFS Engineering
Computational Fluids & Structures Engineering

Computer-Aided Design for Aircraft

Master Thesis Report

Author:
Malo Drougard

EPFL Supervisor:
Professor Doctor Mark Pauly

CFSE Supervisors:
Doctor Jan Vos
Aidan Jungo

Lausanne, 31-08-2018

Abstract

In this project, a framework[1] was developed that permits to edit the geometrical data given in the CPACS (Common Parametric Aircraft Configuration Schema [2]) aircraft format. For historical reasons, the geometrical CPACS data has a complex hierarchical structure with multiple dependencies and multiple alternative to describe the aircraft geometry. In this project we first developed a set of tools to handle the complexity of the geometry description. Then we developed a framework to edit the geometrical CPACS data that does not require the need to understand the underlying CAPCS description. In the third step, the wing description was translated into high level parameters commonly used in aircraft design. These parameters have the particularity to be also applicable for unconventional aircraft configurations and considerably simplifies the possibilities to modify a wing. Finally, we developed a user friendly interface that can be used with our framework. To accomplish this we integrated the framework in the graphical interface called TIGLViewer [3].

Contents

1 Abbreviations

AGILE	= Aircraft 3rd Generation MDO for Innovative Collaboration of Heterogenous Team of Experts
API	= Application Programming Interface
DLR	= German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt)
CAD	= Computer-Aided Design
CFD	= Computational Fluid Dynamics
CPACS	= Common Parametric Aircraft Configuration Schema
GUI	= Graphical User Interface
MDO	= Multidisciplinary Design Optimization
UML	= Unified Modeling Language
XML	= Extensible Markup Language
XSD	= XML Schema Definition

2 Introduction

The design process of a modern aircraft involves many disciplines. Structures, aerodynamics, controls, systems, noise, emissions are only some of them. According to many text books [4] [5], the aircraft design process can be described in 3 phases: Conceptual Design, Preliminary Design, Detailed Design. The goal of Conceptual Design Phase is to select one, or very few workable concepts and optimize them as much as possible. The main characteristics of the aircraft that the best meets the requirements are defined. In the Preliminary Design Phase, the selected concept is studied into more detail. Analysis and simulations are carried out to fine-tune the geometry, while at the same time all sub-systems begin to be shaped. The goal of this phase is to completely define the aircraft that is going to be manufactured and to 'freeze' its design. The final step of the design process is the Detail Design Phase during which all components and parts are sized in details. During this phase, all manufacturing documentation is produced.

The three different design phases and the disciplines involved in each of them require different knowledges and tools. There are many teams of experts across the world involved in the design process, and one of the main challenge in this process is to integrate all the different disciplines, tools and teams. They need to exchange data, but also to work together to find the best solution. The design process of an aircraft can be view as a Multidisciplinary Design and Optimization process (MDO).

The high complexity of the MDO process creates difficulties for companies and institutions to create and analyze unconventional aircrafts such as a strut-braced wing, a box wing or blended wing a body, see figure **??**. This is mainly because there is a lack of knowledge and empirical data for such aircrafts. However, the transport sector is growing rapdily and there is high pressure to reduce prices, improve quality and make a greener transportation system. The aviation industry therefore needs to explore and analyze such unconventional solutions that have a large potential to reduce emissions.

(a) strut-braced aircraft

(b) blended wing aircraft

(c) box wing aircraft

Figure 1: non-conventional aircraft geometry

The AGILE project, funded by the European Commission, started in May 2015 and running until November 2018, has as the objective to develop the next generation of aircraft by using Multidiscplinary Design and Optimization processes [6]. The goal is to develop a framework that allows to explore unconventional aircrafts and significantly reduce the costs and time to market, leading to more cost effective and greener aircraft solution.

The CPACS Format that we will present in section 3 was developed prior to the AGILE project by the DLR to exchange data between the different disciplines, and was adopted by the AGILE project to exchange data. Actually, different tools and softwares store their data in this format. There exist, as we will see in section 4, a framework called TIGL that processes and displays the CPACS geometrical data. But there is no framework that supports the editing of the geometrical data of CPACS, and at the start of this master project the geometrical data of the aircraft has to be edited manually. The goal of this project is to a create a geometry editor for CPACS. Combine with TIGLViewer, the user will have a Computer-Aided Design (CAD) like software to edit and display the geometry of aircrafts. The code of this project can be find on GitHub of CFSE[1].

3 Description of CPACS

Most of the disciplines involved in aircraft design need to exchange data between each other. If each pair of disciplines needs to define an ad hoc interface to exchange data, this will require $(N+1)N/2$ interfaces, where N is the number of disciplines. The Common Parametric Aircraft Configuration Scheme (CAPCS) is a data definition for air transportation system that aims to solve this issue. The idea is to use CPACS as a central data model between all these disciplines [7]. Each discipline would ideally exchange data only via CPACS and thus have only one interface, see figure 2. CPACS can hold various informations ranging from analyses on climate impact to the structural component of a wing. We will present in the next sub section the aircraft geometric data of CPACS, that is only a small subset of the format. A complete documentation of CPACS can be found on their Github [2] as a compiled HTML file format.

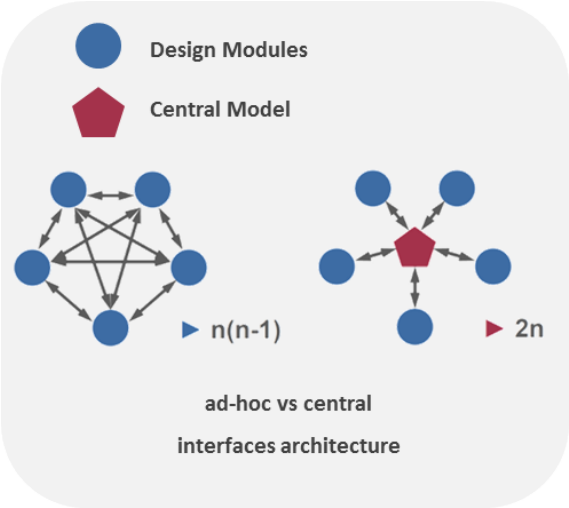


Figure 2: Illustration of the advantage of a central model, image from [2]

The CPACS format uses the XML language to store the data. The structure of the CPACS file is described in the XML Schema definition Language (XSD). To define the structure of the XML document, the XSD language define types for XML elements and defines the structure of the elements in the document.

3.1 Notations

Words in *italic* are CPACS XSD type defined in the CPACS XSD. We generally omit the suffix "Type" of the type name to avoid overcharging the syntax. For example, the type *transformationType* becomes *transformation*. We also often use the type name to speak about a instance of this type. For example, in the sentence; "This *wingElement* is valid", we mean, "This *wingElement* instance is valid"; There are many CPACS XSD types that differ only in one "s". For example, there is *positionings* type and *positioning* type. To make the differentiation between the plural and the type, we use a uppercase S for type names that end with s and a lowercase s for the plural. For example: *positionings* is multiple *positioning* instances and a *positioningS* is one instance of the type *positionings*.

3.2 CPACS Geometry

The aircraft geometrical data is a small part of a CPACS file. Most geometrical data are contained in the type *model*, see figure 3. The *model* appears in the node "/cpacs/vehicles/aircraft/" of a CPACS

XML tree. The *aircraft* can contain one or multiple *models*, but generally it contains only one *model*. One *model* describes an aircraft shape, its wings, its fuselages, etc. All the data we need for this project are contained in *models*. There is one exception to this rule, some elements of the model reference *profileGeometry*. These *profileGeometries* define a set of points that represent a curve and appear in the node "/cpacs/vehicles/profiles/" of a CPACS XML tree.

We will present in detail how a wing is build to understand the logic of CPACS. We let the reader discover the other parts of the aircraft in the CPACS documentation[2].

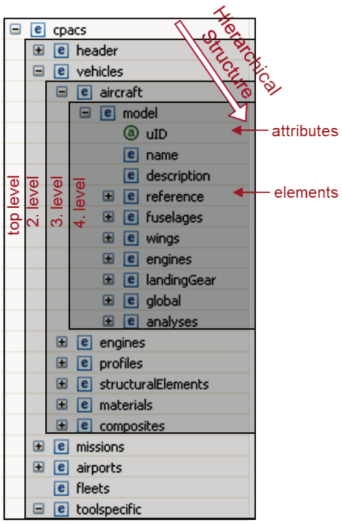


Figure 3: CPACS structure, image from [2]

3.2.1 CPACS Wing

We will explain the wing geometrical description in CPACS using a bottom-up approach, then we will take a look at some available options and make some comments.

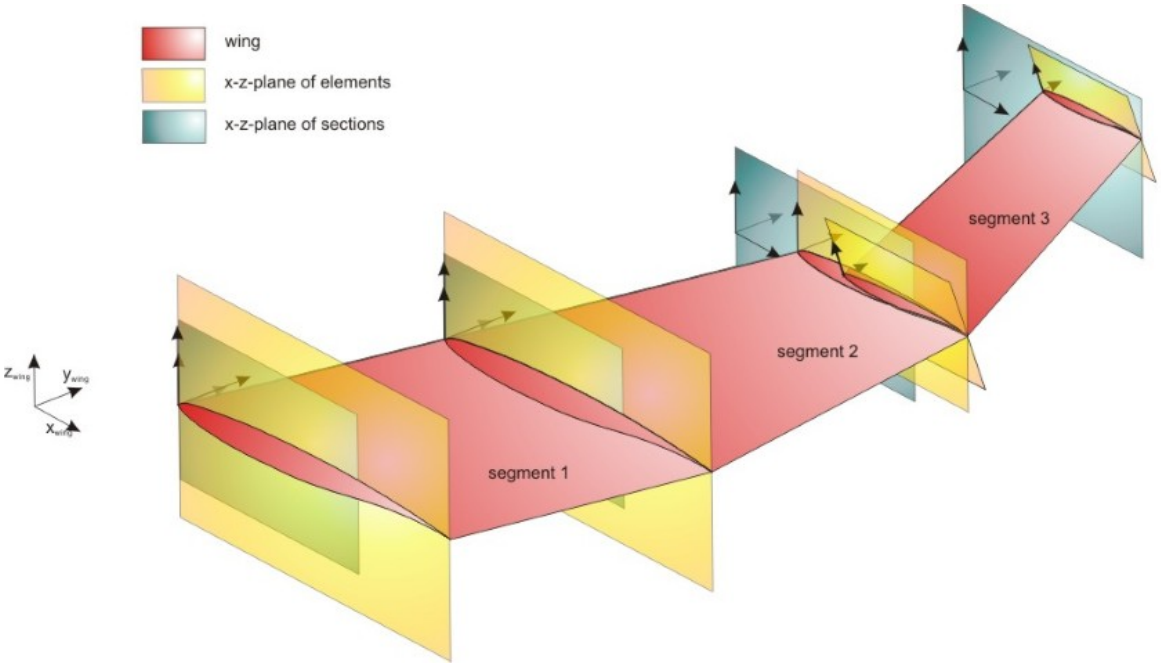


Figure 4: Illustration of the components of a CPACS wing, image from [2]

First, we need to define a *wingAirfoil* element in *profiles*. A *wingAirfoil* is a *profileGeometry* instance. Therefor the *wingAirfoil* contains a list of points that describes a curve. This curve represent the profile of the wing. You can imagine it as a a normalized slice of the wing. The CPACS format allows to define *wingAirfoil* that are not on a 2D plane, but TIGL, that is the main existing tool for CPACS as we will seen in section 4.2, imposes that *wingAirfoil* point lies on the XZ plane. Therefore we will assume that *wingAirfoils* are always described in the XZ plane.

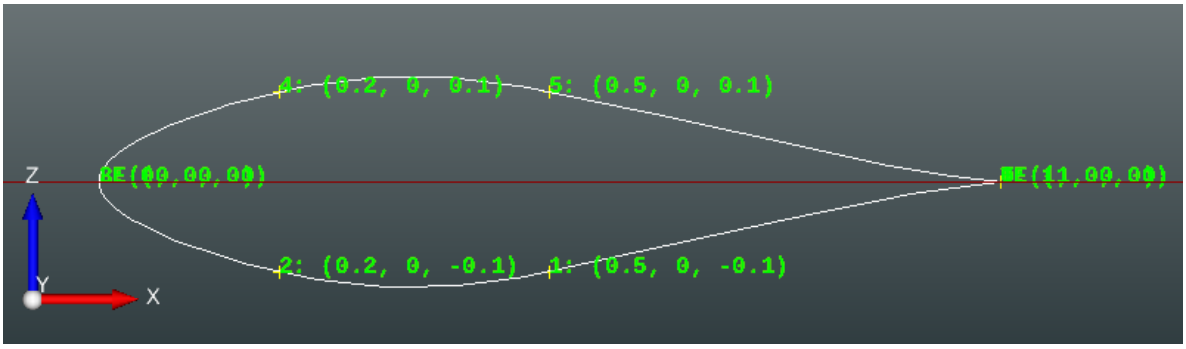


Figure 5: A airfoil shown in TIGLViewer, the yellow crosses are the points define in the CPACS file

Then, the *wingAirfoil* element is referenced in a *wingElement* element. Each *wingElement* must contain exactly one *wingAirfoil* reference. This *wingElement* contains a *transformation*. This *transformation* can scale, rotate and translate the *wingElement*. The *wingAirfoil* referenced by this *wingElement* lives now in this space and, thus can be transformed in the same way.

Then, one or multiple *wingElement* are contained in *wingElementS*. *WingElementS* is just a container for *wingElements*. The container can not perform any transformation on *wingElement*. Generally, one *wingElementS* holds only one *wingElement*.

Then the *wingElementS* is itself contained in a *wingSection* element. A *wingSection* can scale, rotate and translate its *wingElementS* via a *transformation*. This transformation is affecting all the *wingElements* in *wingElementS*.

Then, all the *wingSection* elements are contained in a *wingSectionS* element. This *wingSectionS* is just a container and no *transformation* can be perform on it.

Finally, the *wingSectionS* is contained in the *wing*. The *wing* can scale, rotate and translate the *wingSectionS* via a *transformation*. Figure 6 provides an overview of all the rotations, scalings and translations that can be applied on the *wingAirfoil*.

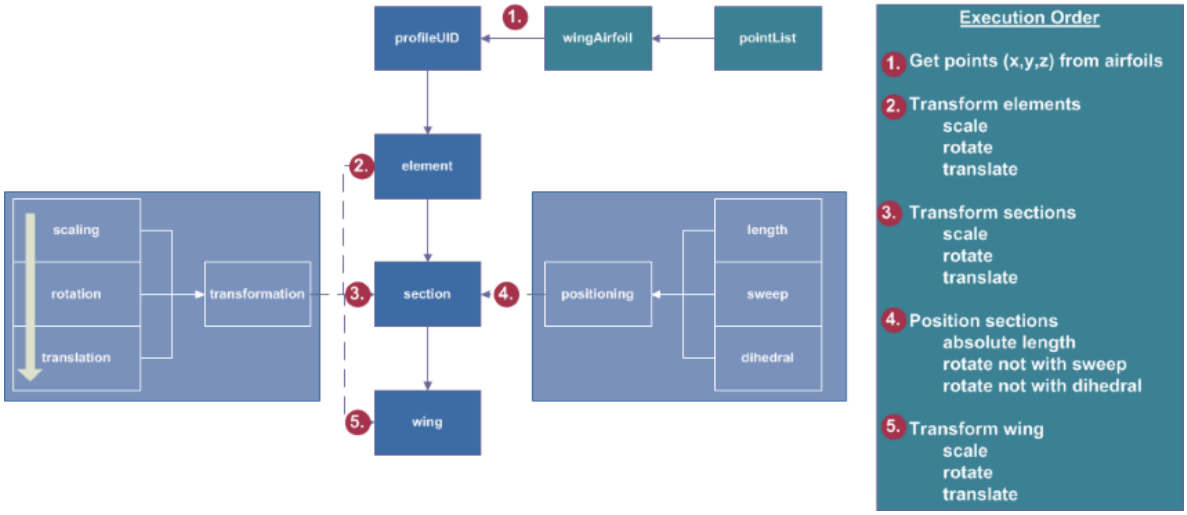


Figure 6: Wing transformations order, image from [2]

At this point, we have, basically, scaled airfoils positioned in 3D space. We will often use the term of wire to speak about these scaled positioned airfoils. Now we need to connect these wires together to create a wing structure. To connect wires together we use *wingSegment* elements. *WingSegment* elements take two *wingElements* and connect them together.

Then, all the *wingSegments* are contained in a *wingSegmentS* and the *wingSegmentS* elements is contained in the *wing* element. See figure 7 for a UML diagram of the CPACS wing and see figure 4 to see a schema of different components that compose a wing.

The wing structure described above is the most simple structure to create a wing in CPACS. However, there is an optional element in *wing* called, *positioningS* that can also modify the global shape of a wing. *PostioningS* is a container for *positionings*. *Postioning* permits to add a translation to *wingSection*. The translation of the *positioning* is described in terms of sweep angle, dihedral angle and length. This translation is added to the translation defined in *wingSection* and takes place after the *wingSection transformation* is applied. The translation must be defined relatively to another *positioning* or to the wing origin. *Positioning* does not allow to rotate or scale *section*. See figure 6 for the execution order of the different rotations, scalings and translations that apply on *wingAirfoils*.

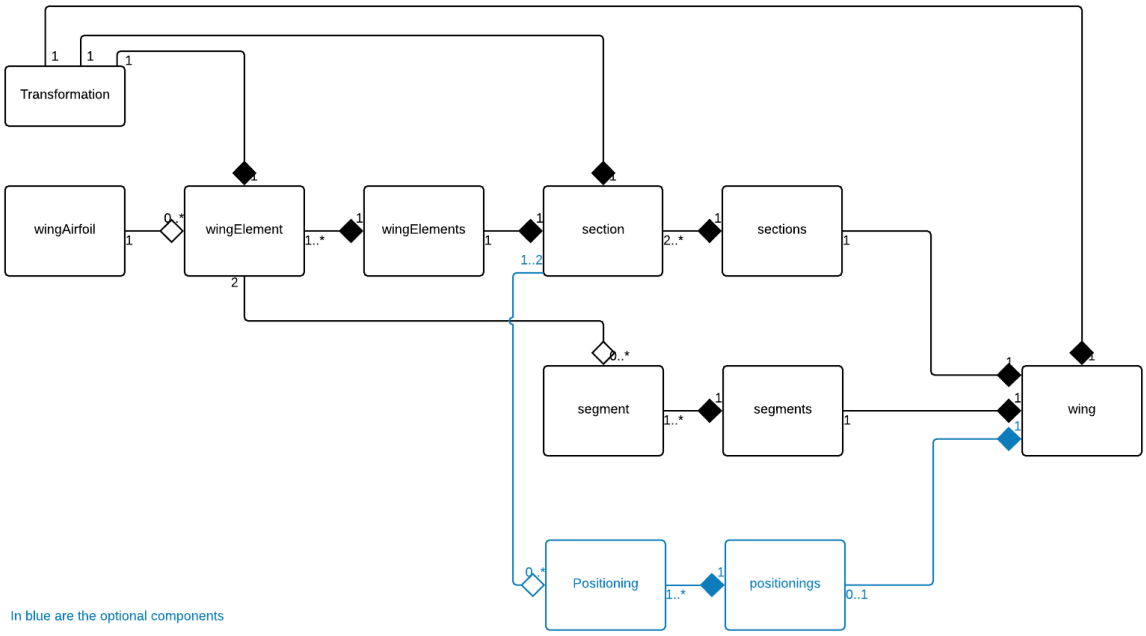


Figure 7: UML of CPACS Wing complex type

The *wing* contains another element, called *componentSegment*. *componentSegment* contains information about the wing structure, its control surfaces, the wing fuel tanks and the wing fuselage attachment. Unfortunately, due to time constraints this part of the wing will not be covered in this project.

Remark that we never speak about how the points of the *wingAirfoil* are connected together nor how the the surfaces between the airfoils are generated. This is because the CPACS format does not specify how this generation should be done. This task is left to the CPACS interpreter as for example TIGL. TIGL generates the curves between the *wingAirfoil* points using splines. These curves are then connected together using loft surfaces.

4 Existing Tools

In this section we will present the existing framework for CPACS, namely, TIXI, TIGL, TIGLViewer.

4.1 TIXI

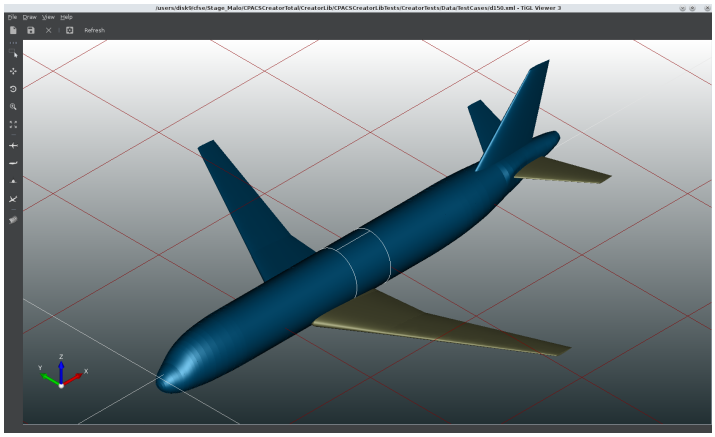
The TIXI [8] Library provides a simple API to read and write data in a XML structure. The goal was to help developers to perform simple tasks like writing an integer or reading a text value in a XML structure. The library provides functions to read, modify and create basic XML elements. It also provides validation functions to check if the XML structure respect a given XML-schema definition (XSD). To describe the target element, TIXI uses a restricted subset of XPath expressions. In particular, the specified path has to be unique. The users of the library can not modify multiples elements at once. TIXI is a C library build on top of the popular and stable XML-library libXML2 [9]. Language interfaces for C++, Fortran and Python are provided. TIXI is released under Apache2 license and was developed by DLR.

4.2 TIGL

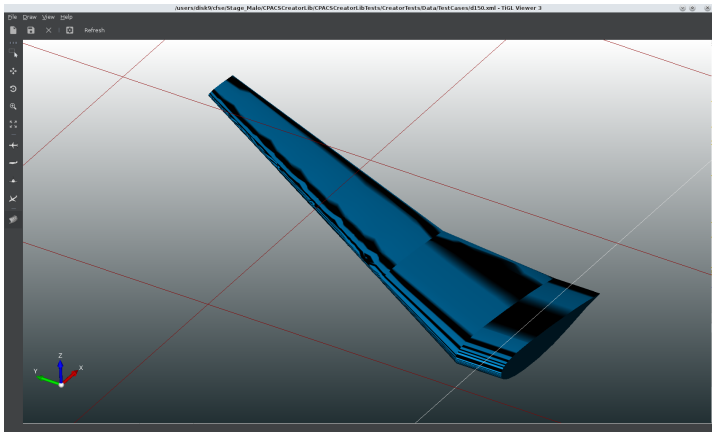
The TIGL [3] Geometry Library provides an API to query and process CPACS geometrical data. For example, the library has functions to detect how many wings the current aircraft configuration contains, to compute the planform area of a wing or to find the lower point on a fuselage segment. The library can also export the geometry in other formats as for example IGES, STEP, STL and VTK format. It is important to mention that no function to modify the CPACS data is available. TIGL is written is C++ and makes heavy use of the OpenCASCADE [10] framework to represent the geometry by B-spline surfaces. TIGL uses the TIXI library to access the XML Structure. It provides external interfaces for C, C++, Python, MATLAB and FORTRAN. The code is released under Apache2 license and developed by the DLR.

4.3 TIGLViewer

TIGLViewer is a software to visualize the CPACS geometrical data, see figure 8. TIGLViewer can build a three dimensional representation of the entire aircraft described in a CPACS file. It can also display only a particular wing or fuselage. The displayed element can be represented as wire frames, surfaces or surfaces with reflection lines. TIGLViewer is written in C++ and uses Qt5 framework for the GUI. It makes a heavy use of TIGL and OpenCASCADE libraries to build the three dimensional representation. TIGLViewer comes with TIGL library in the following GitHub repository [3]. TIGLViewer is released under the Apache2 license and is developed by DLR.



(a) TIGLViewer displaying a whole aircraft



(b) TIGLViewer displaying a wing with reflection lines

Figure 8: TIGLViewer interface

5 Overview of Creator Framework

The goal was to create an editor to permit the modification of the geometrical data of CPACS. As the reader can see in section 3 the CPACS format has a low-level description in the sense that the aircraft is described in terms of sections and connections. But there are no high level parameters as the wing span, wing area or global length of the fuselage. The idea was to abstract these lower components in terms of higher level parameters and to enable the editing of these higher level parameters. Due to time constraints, our framework focuses on the wing only. During the aircraft conceptual and preliminary design phases the main parameters of the wing are the the wing profiles, the wing span, the wing aspect ratio, the area of the wing, the sweep angle and the dihedral angle. Therefore, we have decided to select these parameters. The precise definition of these parameters are not so clear and multiple versions of them are used. We will fix and define these parameters for our framework in section 6. Such a parametric model view has many advantages [11]. These parameters can be used for rapid external performance analyzes, as for example the evaluation of the parasite drag. Compared to traditional computer aided design packages, the time and expertise to learn the software is fast. They allow the designer to quickly change the global shape of the aircraft without going in the detail of each components. Our goal can be split into two smaller goals:

- Modifying CPACS geometry with high level parameters.
- Having a user friendly interface to modify them.

One of the first architecture design choices we made was to keep the first goal separated from the second one. The CFS Engineering team and their partners use and develop many different tools and scripts to perform analyzes and optimization studies. It will be beneficial for them to have a library that performs modification of high level aircraft parameters. This library can then be used by other software packages. In particular, the library can be used in the optimization process to generate automatically multiple different geometries. A use case of this library is shown in appendix B where we generated multiple geometries to perform CFD simulations on the super-computer Beskow [12]. Another benefit of a clean separation of the two tasks is to keep the logic away from the presentation, we focus on only one purpose at the time. Thus, we came up with two modules, a library to modify CPACS geometry, and we will call this library CPACS creator library or simply CPACSCreatorLib, and a GUI, called CPACSCreator, that uses CPACSCreatorLib.

In the rest of the document, we will first define the high level aircraft parameters. Then we will present the main challenges to work with the CPACS format and the algorithm that we used. Following that we will present the CPACSCreator GUI. Finally we will take a look to implementation and possible future works.

6 CPACSCreator Definitions

The following definitions play a central role in this project. These definitions will define the high level parameters of our framework and will be present explicitly or implicitly in the final user interface. As example, the result of the function 'getWingSweep' returns the value defined as wing sweep below. Moreover they help us to clarify and think about the structure of CPACS. During the project, the precise definitions have evolved by discussion with aircraft designers, realizations about the CPACS format and implementation constraints. In this subsection, we will first present the final definitions that we end up with, then we will explain the main reasons that led to our choices.

6.1 Raw Definitions

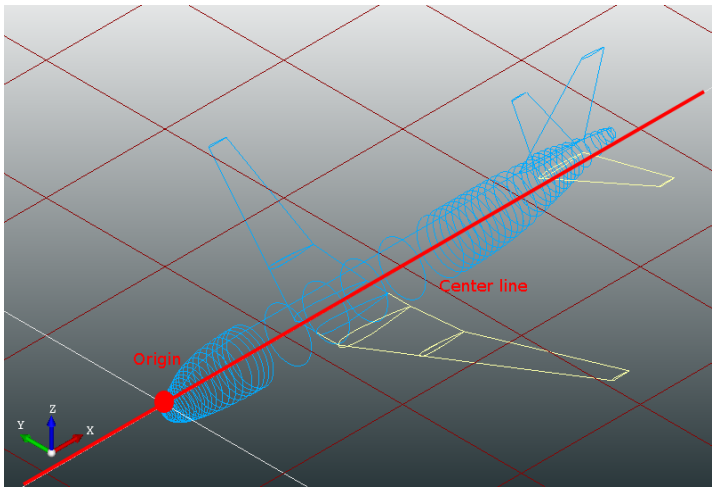


Figure 9: Conventional position of the aircraft in world coordinate system

World coordinate system: The world coordinate system is a right-handed three-dimensional Cartesian coordinates system. This is the same coordinate system as in CPACS and TIGL.

Wing coordinate system: The wing coordinate system is the world coordinate system transformed by the inverse of the wing *transformation*. We will use this coordinate system for the wing parameters.

Assumption 1: Position of the aircraft in the world coordinate system: We assume that the aircraft is described with respect to the CPACS convention. This mean that the nose of the aircraft is at the origin, that the positive X-axis direction is from nose to tail, that the positive Y-axis goes from symmetry plane to the right wingtip and that the positive Z-axis goes from landing gear to the tip of the vertical tail plane, see figure 9.

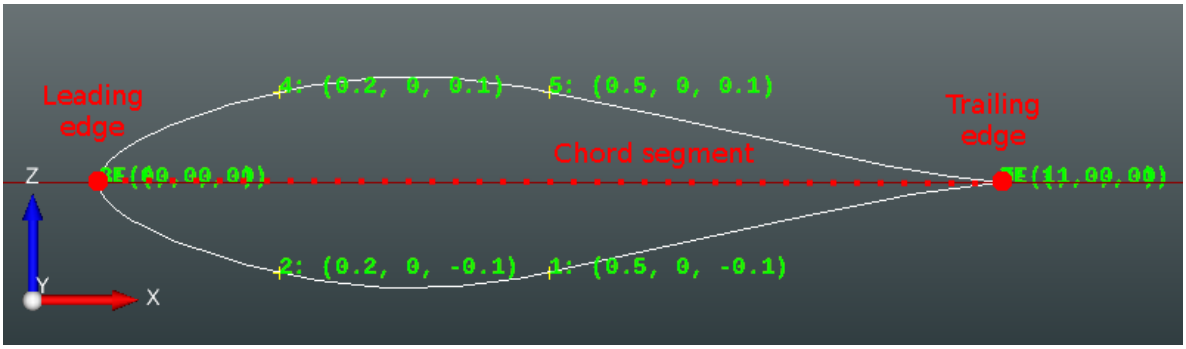


Figure 10: Airfoil profile generated from the points in table 1

X	1	0.5	0.2	0	0.2	0.5	1
Y	0	0	0	0	0	0	0
Z	0	-0.1	-0.1	0	0.1	0.1	0

Table 1: List points for the airfoil of figure 10

Trailing edge of a *wingAirfoil* ($\mathcal{T}_{aifoiUID}$): In CPACS a wing airfoil is described as a set of points that represents a curve. The trailing edge is simply the first point in this description. This is the same definition as in CPACS.

Leading edge of a *wingAirfoil* ($\mathcal{L}_{airfoiUID}$): In CPACS a wing airfoil is described as a set of points that represents a curve. The leading edge is the point in the description that is the most distant of the trailing. This is the same definition as in CPACS.

Chord segment of a *wingAirfoil*: The segment starting from the leading edge to the trailing edge of this *wingAirfoil*.

Trailing edge of a *wingElement* ($\mathcal{T}_{elementUID}$): A well formed *wingElement* generates a closed curve in the world space. This curve is constructed from the *wingAirfoil* and the affine transformations apply on the *wingElement*. As shown in section 3.2, these affine transformations are given by the *wingElement*, *wingSection*, *positioning* and *wing*. The trailing edge point of the *wingElement* is the trailing edge point of the associated *wingAirfoil* transformed by the same affine transformations as for the *wingElement*. This definition is the same definition as in TIGL. If we speak about the trailing edge of a *wingElement* in the wing coordinate system, it's simply the trailing edge of a *wingElement* in the wing coordinate system. This means that we do not apply the *wing transformation* on the *wingAirfoil*.

Leading edge of a *wingElement* ($\mathcal{L}_{elementUID}$): A well formed *wingElement* generates a closed curve in the world space. This curve is build from the *wingAirfoil* and the affine transformations applied on the *wingElement*. As shown in section 3.2, these affine transformations are given by the *wingElement*, *wingSection*, *positioning* and *wing*. The leading edge point of the *wingElement* is the leading edge point of the associated *wingAirfoil* transformed by the same affine transformations as for the *wingElement*. This is the same definition as in TIGL. If we speak about the leading edge of a *wingElement* in the wing coordinate system, it's simply the leading edge of a *wingElement* in the wing coordinate system. This mean that we do not apply the *wing transformation* on the *wingAirfoil*.

Chord segment of a *wingElement*: The segment starting from the leading edge to the trailing edge of this *wingElement*.

Chord percent point $\mathcal{C}_{percent,elementUID}$ of *wingElement*: The point on the chord that is at the given percentage of the segment. The percentage given is expressed between 0 and 1. The percentage

0 is assigned to the leading edge and the percentage 1 is assigned to the trailing edge. For example: $\mathcal{C}_{0,a} = \mathcal{L}_a$, $\mathcal{C}_{1,a} = \mathcal{T}_a$ and $\mathcal{C}_{0.5,a}$ is the point on the middle of the chord.

Wires: We often want to speak about the curve generated by the scaled airfoils positioned in 3D space. We use the term of wire to reference it, because these curves generate the wire-frame of our aircraft. This is more a syntactic sugar than really a definition.

End *wingElements*: The wing contains generally more than two *wingElements*. Some of them are connected to one other element and some of them are connected to multiples other *wingElement* by *wingSegment*. We define "end *wingElement* " as the *wingElements* that are connected to one and only one other *wingElement*. This means basically that a end *wingElement* is at one extremity of the wing.

Root *wingElement*: The end *wingElement* of the wing that has the minimal distance in Y to the X axis of the wing coordinate system. The minimal distance in Y is simply the absolute value of the Y coordinate of $\mathcal{C}_{0.5,}$ in the wing system coordinate. Notice that we use the wing system coordinate and not the world coordinate system. We call, sometimes, the root *wingElement* simply the root. In the case of multiple end *wingElement* with equal distances. We chose the *wingElement* that appears first in the CPACS file.

Tip *wingElement*: The *wingElement* of the wing that has the maximal distance in Y from the root of the wing in the wing coordinate system. The distance as for the root is evaluated at $\mathcal{C}_{0.5,}$. Notice that the tip is not necessarily an end *wingElement*. As for the root, in the case of multiple end *wingElement* with equal distances. We chose the *wingElement* that appears first in the CPACS file.

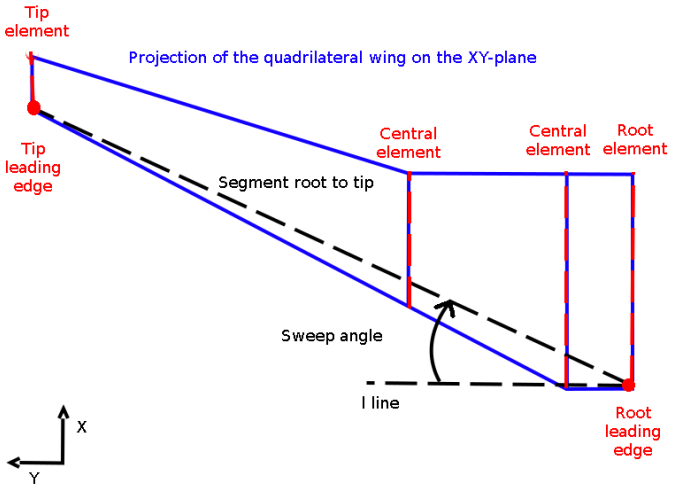


Figure 11: Sweep angle example

Sweep Angle of a wing: The sweep angle is basically the horizontal angle between the wing and the fuselage. We define this parameter in the wing coordinate system. So, all the following axis, planes, $\mathcal{L}$ and $\mathcal{T}$ are living in the wing coordinate system. Let $\mathcal{L}_{root}$ be the leading edge point of the root, and $\mathcal{L}'_{root}$ be the projection $\mathcal{L}_{root}$ on the the XY plane, let $\mathcal{L}_{tip}$ be the leading edge point of the tip and $\mathcal{L}'_{tip}$ the projection of $\mathcal{L}_{tip}$ on the XY plane, let $\overline{\mathcal{L}'_{root}\mathcal{L}'_{tip}}$ be the the segment form $\mathcal{L}'_{root}$ to $\mathcal{L}'_{tip}$, let l be a line parallel to the Y axis passing by $\mathcal{L}'_{root}$. We define the sweep angle as the angle between the segment $\overline{\mathcal{L}'_{root}\mathcal{L}'_{tip}}$ and the line l. See figure 11.

Sweep Angle of a wing at Chord x Percent: This is a generalization of the sweep angle definition. Instead of using the leading edges to define the sweep angle, we use the chord point at the given percentage x. The percentage given is expressed between 0 and 1. The percentage 0 is assigned to the

leading edge and the percentage 1 is assigned to the trailing edge. A formal definition of the 'Sweep Angle of a wing at Chord x Percent' can be obtained by replacing all the $\mathcal{L}$ by $\mathcal{C}_x$, in the sweep angle definition.

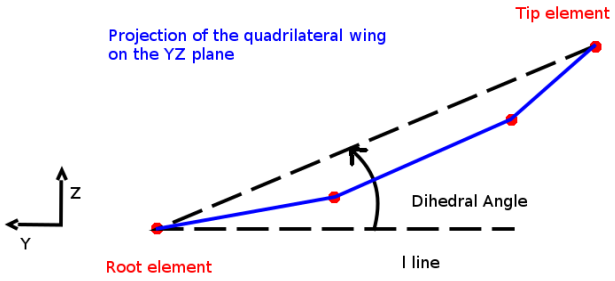


Figure 12: Dihedral angle example

Dihedral Angle of a wing: The dihedral angle is basically the vertical angle between the wing and horizontal plane. The dihedral angle is defined in the same way as the sweep angle. We define the dihedral parameter in the wing coordinate system. So, all the following axis, planes, $\mathcal{L}$ and $\mathcal{T}$ are living in the wing coordinate system. Let $\mathcal{L}_{root}$ be the leading edge point of the root, and $\mathcal{L}'_{root}$ be the projection $\mathcal{L}_{root}$ on the the YZ plane, let $\mathcal{L}_{tip}$ be the leading edge point of the tip and $\mathcal{L}'_{tip}$ the projection of $\mathcal{L}_{tip}$ on the YZ plane, let $\overline{\mathcal{L}'_{root}\mathcal{L}'_{tip}}$ be the the segment form $\mathcal{L}'_{root}$ to $\mathcal{L}'_{tip}$, let l be a line parallel to the Y axis passing by $\mathcal{L}'_{root}$. We define the dihedral angle as the angle between the segment $\overline{\mathcal{L}'_{root}\mathcal{L}'_{tip}}$ and the line l. See figure 12.

Dihedral Angle of a wing at Chord x Percent: This is a generalization of the dihedral angle definition. Instead of using the leading edges to define the dihedral angle, we use the chord point at the given percentage x. The percentage given is expressed between 0 and 1. The percentage 0 is assigned to the leading edge and the percentage 1 is assigned to the trailing edge. A formal definition of the 'Dihedral Angle of a wing at Chord x Percent' can be obtained by replacing all the $\mathcal{L}$ by $\mathcal{C}_x$, in the dihedral angle definition.

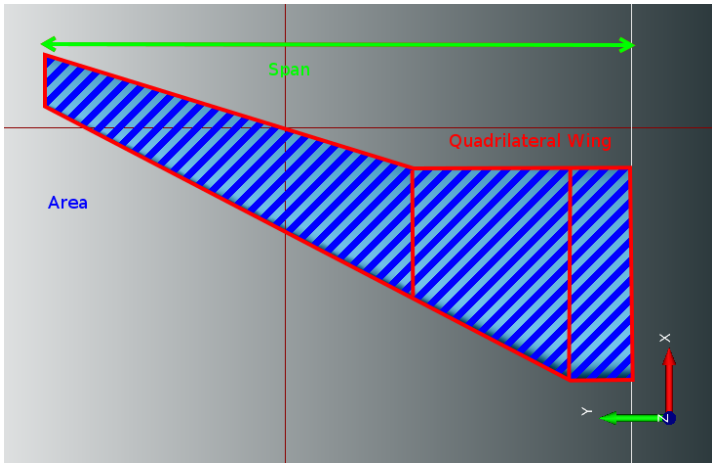


Figure 13: Quadrilateral Wing, Wing Span and Wing Top Area

Wing Span (b): The span is basically the width of the wing, see figure 13. We define it precisely as the distance in Y between the root leading edge and the tip leading edge in the wing coordinate system. This means:

$$b = abs(\mathcal{L}_{tip}[y] - \mathcal{L}_{root}[y])$$

Quadrilateral Wing: The simplified version of the wing composed only by the quadrilaterals connecting the chord segment of *wingElement* together, see figure 13

Wing Top Reference Area (A_{xy}): The area of the quadrilateral wing projection on the XY-plane in the wing coordinate system. Be careful, this definition diverges of the result of the TIGL function 'tiglWingGetReferenceArea(TIGL_X_Y_PLANE)' that returns the quadrilateral area in the world coordinate system. We will see in the next subsection why we selected this definition over the TIGL definition.

Wing Front Reference Area (A_{yz}): The area of the quadrilateral wing projection on the YZ-plane in wing coordinate system. Be careful, this definition diverges of the result of the TIGL function 'tiglWingGetReferenceArea(TIGL_Y_Z_PLANE)' that returns the quadrilateral area in the world coordinate system.

Wing Lateral Reference Area (A_{xz}): The area of the quadrilateral wing projection on the XZ-plane in the wing coordinate system, see figure 13. Be careful, this definition diverges of the result of the TIGL function 'tiglWingGetReferenceArea(TIGL_X_Z_PLANE)' that returns the quadrilateral area in the world coordinate system.

Aspect Ratio (AR): The Aspect ratio is a wing parameter that captures some aspect of the wing shape. A long, narrow wing has a high aspect ratio, whereas a short, wide wing has a low aspect ratio. The precise definition of the aspect ratio is:

$$AR = b^2/A_{xy}$$

.

6.2 Discussion on definitions

One of the principles that led to our choices for the definitions was to be consistent with CPACS and TIGL. We did not want to end up with two definitions living in the same space. This principle was typically applied to the trailing edge and to the leading edge. In the aeronautic field the trailing edge and the leading edge are often defined in terms of curvature. But, if we haven taken this definition, the global framework of CPACS, TIGL and CPACSCreator would have been more complicated to use and understand. For example, if a user writes manually a CPACS file, he has in mind that the first point of the airfoil is the trailing edge, but when he opens the file with CPACSCreator, the trailing edge can be at other points.

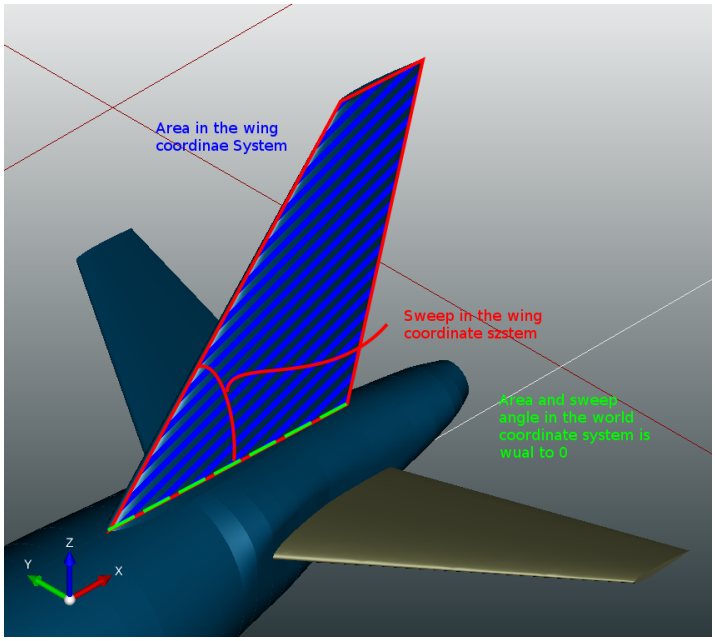


Figure 14: Representation of the sweep and area in the world coordinate system and in the wing coordinate system

Another important decision was to chose if we want to define the wing parameter in the wing coordinate system or in the world coordinate system. Imagine that we selected the world coordinate system. In the case of a vertical wing the sweep angle will be 0, the top area will be 0, the span will be 0 and the aspect ratio will generate a division by zero exception, see figure 14. We agree that these parameters are weird. The user would probably prefer to obtain these parameters as if the wing is rotated by 90 degrees, so the modification of the parameters are possible and make sense.
Apart from this, we observed that the vertical wing is often constructed in the following way: first the wing is described in the XY plane of the world coordinate system and then the wing is rotated by 90 degrees around the X axis without any additional scaling by the wing *transformation*. In this particular case of a wing *transformation* of 0 or 90 degrees with a trivial scaling, if we define the parameters in the wing coordinate system, the wing parameter seems reasonable for both a horizontal wing and a vertical wing. This is the main reason behind the decision to define the parameters in the wing coordinate system and not in the world coordinate system.

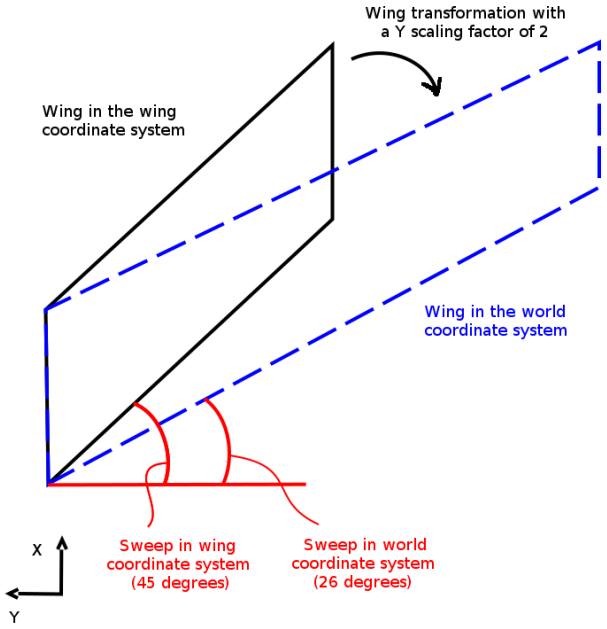


Figure 15: Difference between sweep in the world coordinate system and the wing coordinate system in the case of a wing transformation with a Y scaling factor of 2

Remark that the scaling component of the wing *transformation* must be trivial, otherwise the parameters will be distorted. Take as example a wing with a sweep angle of 45 degrees in the wing coordinate system and a scaling of 2 in Y in the wing *transformation*. The wing will be stretched in Y and the wing displayed in TIGL would have a visual sweep angle of 26 degrees, see figure 15.

To summarize the wing parameter definitions in the wing coordinate system is consistent with the displayed wing if the wing transform has a trivial scaling and a trivial rotation or a rotation of 90 degrees around the X axis. At a first view, the constraint on the wing *transformation* seems strong, but as we will see in the section 8, there always exist an equivalent CPACS description of a wing that meets this constraint.

There is a second reason that we have selected the wing coordinate system over the world coordinate system. As shown in section 3.2 the local sweep angle and dihedral in *positionings* live in the wing coordinate system. So it makes sense that both definitions live in the same context. Of course, we can argue that the area definition is not anymore consistent with the definition of area by the TIGL function 'tiglWingGetReferenceArea', but this is just a function of TIGL that is less used as the CPACS *positionings* definitions.

For the global wing sweep angle, there exists no unique definition in the aircraft design field. The sweep angle is well defined in the case of a really simple wing with two sections. But even for this case there are multiple definitions, sometimes the angle is measured at the leading edge, sometimes at 25% of the cord, sometimes at the trailing edge as shown in figure 16. We cover all these different percentage definitions by generalizing our definition to support any chord percentage. By default we will use the leading edge definition. But in the case of a wing with multiple kinks or for unconventional aircraft as the blended wing body or the box wing there is no convention for the wing sweep. It is even difficult to find references with a proper definition of the sweep angle. We have evaluated two alternatives: define the global sweep by the wing root and the wing tip, or define the global sweep as an average of all the different segments of the wing. The idea for the average sweep is to weight the sweep of each segment by the area of this segment. We have chosen to define the wing sweep using

the wing root and the wing tip because this definitions seems to be more appropriate for the box wing and unconventional wing in general. For example for the box wing aircraft shown in figure 17, the averaged sweep definition results in a computed sweep of 90 degrees. But when we will look at the wing, the sweep angle that we observe is more in the order of 30 degrees.
It is important to notice that one of the keys for this natural definition of sweep angle is that we take the tip *wingElement* and not the other end of the wing to define the sweep. This notion of tip *wingElement* plays an important role. Because it represents what most engineers have in mind when they think about the tip of wing. They think about the extremity of the wing and not its structural end. This is not the case for the root, because the root is generally attached to the fuselage and in that sense its representation is attached to a end.

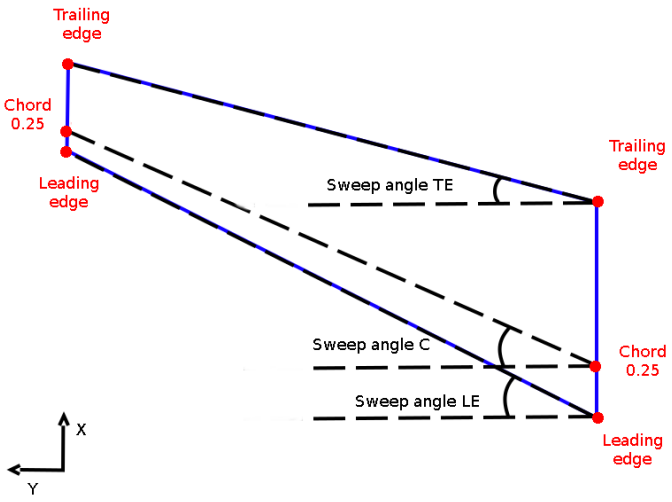


Figure 16: Different alternative for the sweep angle of a simple wing

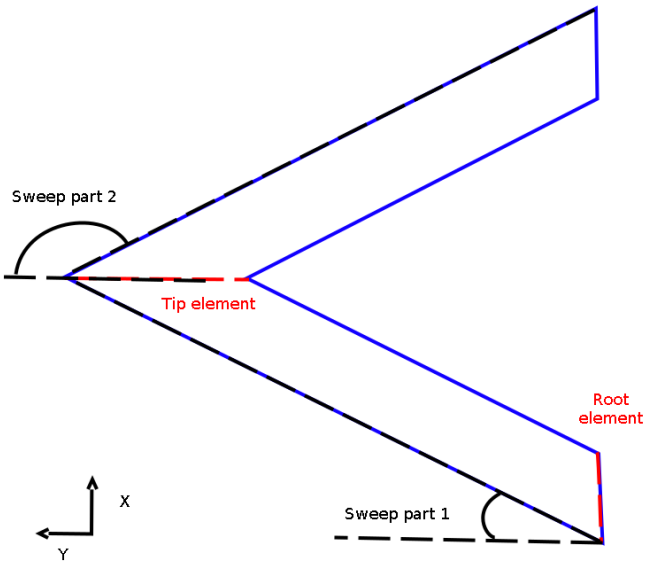


Figure 17: Box wing sweep

7 CPACSCreator Tools

7.1 CPACS Geometry Complexity

If we take a closer look at the CPACS format, we realize it has a highly structured hierarchy with many dependencies. There are several alternatives to describe two identical geometries. The CPACS format has also a high degree of freedom for some of its components. Originally, the geometry description of CPACS was developed with manual edition in mind. The hierarchical structure of CPACS, its dependencies and alternatives facilitate this manual edition. For example, by modifying the wing *transformation*, we can modify the whole wing. Aircraft engineers that are used to define the wing segments position in term of sweep and dihedral can use the optional *positionings* element and the aircraft engineers that prefer to use translation to position the wing segments can use *transformation*. But, once we started to use the format in a software context, the many alternatives to facilitate manual edition become complexities. We will present in this subsection some of the issues that appeared when we started to use the CPACS format in our framework. Then we will present two tools that we have developed to get rid off some of the complexities of the format.

The first issue was to deal with the hierarchical structure and the high degree of freedom in the *wingAirfoil* description. To retrieve the final wire position and scaling, we need to parse the entire wing structure. As shown in section 3.2, the final wire is given by many different element, the *wingAirfoil*, the *wingElement*, the *wingSection*, the *positioning*, the *wing*. To get the complete information about a wire and to be able to modify it, we need to analyze all these different elements. In particular we need to analyze the *wingAirfoil*. The *wingAirfoil* type is really lax about the constraints of its airfoil description. The *wingAirfoil*, as seen in section 3.2, is simply a set of points that represent a curve. But this curve can be of any size and can can be placed any where in the XZ-plane. For example, in one of the CPACS aircraft examples we found that the *wingAirfoils* were scaled and placed in the XZ-plane such that the *wingElements* needs only to place them by a translation in Y. Once we started to modify this CPACS wing, the modification did not had the expected result. For example if we scale the *wingElement* of a such wing, the final wire of this *wingElement* is also modified in its X and Z position. In fact the scale is applied on the whole set of points, so a shift is induced, see figure 18. This means that if we want to modify properly the final wire, we need to analyze its *wingAirfoil* and not only the *transformations* apply on it. This can be tricky because the *wingAirfoil* is a set of points and not an affine transform as the other components.

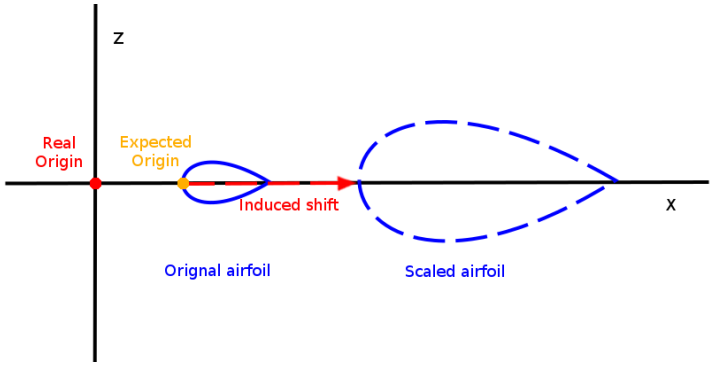


Figure 18: The effect of scaling a *wingAirfoil* where the leading edge is not on the axis origin

The second issue was to deal with the dependencies between CPACS elements and its alternatives. Each time we modify an element, we need to keep in mind its possible dependencies and alternatives. For example, in most of the cases a *wingSection* has only one *wingElement*, so you may want to modify the *wingSection* to modify a particular wire. But sometimes there is a other *wingElement* in the *wingSection*, so it is possible that another wire is modified too. The *positioning* is another good example of these dependencies and alternatives. As shown in section 3.2, the positioning can be

relative to the wing coordinate system or to another positioning. So, if you modify the positionings sometimes it acts only on one *wingSection* and sometimes on multiples *wingSections*.

The third issue was to understand how the *wingElements* are connected. CPACS does not define any constraints on the connections between *wingElements*. A *wingSegment* can connect any two *wingElements* of the wing. These connections can be described in CPACS in any order. As shown in figure 19, CPACS can contain wings that are unconnected and element that are connected to multiple other elements. It is possible to have a *wingElement* that is not used in any *wingSegment*. This means the *wingElement* appears in *wingElementS* but is finally never used. So, when we process the *wingSegmentS* part of the wing we have only the connections between the *wingElements*, but we have no idea how these connections are related between each others.

To summarize, each time we want to handle the final wires generated by the wing we need to be aware of all the components of the wing structure. We need to deal with all the possible alternatives that could generates the wires and we need to take into account the possible side effect of modifying an element.

7.2 CPACS Geometry to Augmented Matrices

The first tool that we have developed is to represents the CPACS *transformations* and *positionings* in terms of augmented matrices for homogeneous coordinate systems. This representation of affine transformation has made its proof in computer graphics and allows us to use represent affine transformation as standard matrices. The idea was to be able to deal only with one matrix for one *wingElement* instead of 4 different elements.

If we look at the CPACS *wing*, see section 3.2, we first draw a set of points in the XZ-plane in a *wingAirfoil*, then these points are placed in the 3D space by a set of affine transformations. And finally this set of points are connected together by *wingSegments*. Since all transformations are regular affine transformations there is no reason that we could not express them as augmented matrices.

More precisely, each *wingAirfoil* is referenced by a *wingElement* that is placed by 3 *transformations* and a optional 1 *positioning*, see figure 7 and figure 6.

Each *transformation* is composed by a scaling S, a rotation R and a translation T. The rotation is described as 3 Euler angles in XYZ order. This means that the first rotation is around the X-axis, the second rotation is around the rotated Y-axis (Y') and the third rotation is around the two times rotated Z-axis (Z"). We first perform the scaling, then the rotation and then the translation. Each of these operations are affine transformation which can be expressed as a matrix in homogeneous coordinate system. So we can express a *transformation* by one augmented matrix M of the form:

$$M = TRS$$

A *positioning* is basically just a translation, but the trick is that the translation can start at a point given by another *positioning*. To get the final translation of positioning, we need to dereference recursively the *positionings* and then add together all the used *positionings*. Once we have the global translation, we can express it as an augmented matrix.

As seen above, each component which places a *wingElement* is an affine transformation that can be expressed as augmented matrices. Therefore, all the transformations that affect a *wingElement* can be expressed as a single matrix G of the form:

$$\begin{aligned} G = WPSE, \text{ where } & W \text{ is the transform matrix of the wing} \\ & P \text{ is the transform matrix of positioning} \\ & S \text{ is the transform matrix of the section} \\ & E \text{ is the transform matrix of the element} \end{aligned}$$

Remark that each *wingElement* has its own G matrix and a single change in the CPACS file can affect all the G matrices. It should be mentioned that the order is also crucial and that we can not permute affine transformations. In the rest of this document, if nothing is specified, G, W, P, S, E will refer to these matrices.

We have implemented a function called $getGlobalPositioningTranslationForSection$ that extracts the translation done by *positioning* components to a particular *wingSection*. We have implemented a function called $getTransformationChainForOneElement$ that extracts each affine matrices involved in a particular *wingElement* placement. We have implemented a function called $getGlobalTransformMatrixOfElement$ that computes the global matrices for a particular *wingElement*. We have also implemented a function that decompose a matrix M into a scaling S, a rotation R and a translation T.

Remarks that with this representation we manage all the affine transforms but not the *wingAirfoils*.

7.3 Wing as a graph

The idea was to represent the connections between the *wingElement* as an undirected graph. The graph is build from the *wingSegment*. Each *wingElements* used in a *wingSegment* becomes a vertex and each connection becomes a edge, see figure 20. With this representation we transform the set of unrelated connections into a graph. This representation really facilitates the processing of the wing structure. We can detect if the wing has unconnected parts or over connected parts, and we can find the end *wingElements*. We use this representation to determine the root *wingElement* between all the different *wingElements*, see algorithm 1. Remark that for the most cases, the wing is in one part and has at most two connections per *wingElement*. This is equivalent to say that the graph is connected, acyclic and have at most two edges per vertex. Basically, the graph becomes a path. In this case we say that the wing has well-formed classical structure. We can assign a order to *wingElements*. We start from the root and follow the path. This order follow the connections of the wing and is called the graph order. As we will see later on, this representation helps us a lot in the Creator standard.

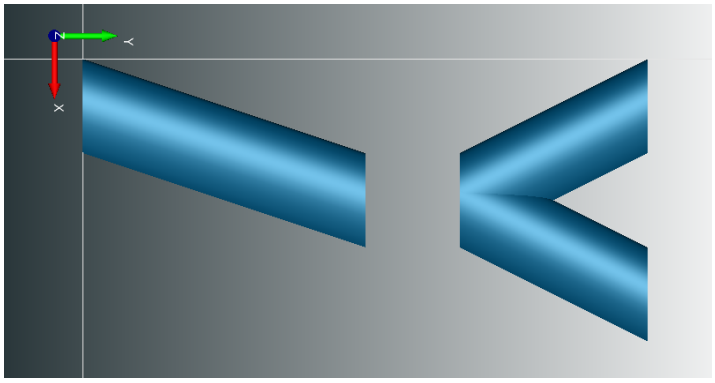


Figure 19: A CPACS wing with segments between *wingElement* 1 and 2, between *wingElement* 3 and 4 and between *wingElement* 4 and 5

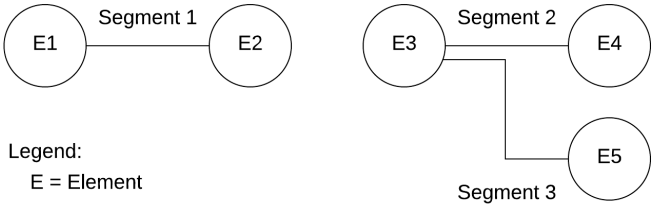


Figure 20: The graph representation of the wing show in figure 19

Algorithm 1 Get Wing Root

1: **function** GETWINGROOT($wingUID$)
2: Create a graph from the *wingSegments*
3: Extract all the leafs from the graph.
4: Compute the $\mathcal{C}_{0.5,}$ of each leaf.
5: $minPoint \leftarrow$ Find the point $\mathcal{C}_{0.5,}[y]$ with the minimal distance to 0
6: **return** The *wingElement* who generate the $minPoint$
7: **end function**

8 CPACSCreator Standardization

As we have discussed in Section 7, there are many alternatives in CPACS to describe two identical geometries. In this section, we will present our preferred way to describe a wing. We will call this particular way of describing the wing the CPACS Creator standard. The Creator standard adds the following constraints on the description of the wing.

wingAirfoils: The wing airfoils should have the trailing edge at (1,0,0) and the leading edge at (0,0,0).

wingElements: The *wingElement transformation* should contain only a scaling component, the rotation and translation component of the transformation should be trivial. There is one exception to this: if the wing has not a well-formed classical structure, in the sense of section 7.3, then the *wingElement transformation* can also have a translation component.

wingSections: The wing *wingSection* should contain exactly one *wingElement* and the *transformation* should contain only a rotation component. The scaling and translation part of the *transformation* should be trivial.

wing: The *wing transformation* should have a trivial scaling, the rotation should be trivial or 90 degrees around the X axis, the translation should move the wing root at its final position.

positionings: The *positioning* of the *wingSections* should describe the translations of the wires in the wing coordinate system. The first *positioning* of the root *wingSection* should be null. Then the *positioning* of the other *wingSections* should be relative to the previous *wingSection* in the graph order. There is one exception to this: if the wing has not a well-formed classical structure, then the graph order is not defined. In this case, no *positionings* are present and the translations of the wires are set in the *wingElement transformation*.

These constraints add a lot of desirable properties to the CPACS wing. First, if two *wingAirfoils* have the same leading edge and trailing edge point, we can switch between the two *wingAirfoils* in the *wingElement* by changing only the reference while maintaining the quadrilateral wing equal. Second the *wingSection* has only one *wingElement*, so we can use the *positionings* to place the *wingElement*. Remember that the *positioning* acts on the *wingSection*, so if *wingSection* has multiple *wingElements*, these *wingElement* move together by *positioning*. We hesitated about putting the translations in *positionings* or putting in *wingElement* and delete all *positionings*. We finally decided to store the translation into the *positioning* if the wing has a well-formed structure for two reasons. First, if the *positioning* holds the whole translation and the other elements satisfy the Creator standard, the *sweep* and *dihedral* elements of the *positioning* become indeed the sweep and dihedral of the segment, see figure 21. The second reason is that the CPACS users that I discussed with, like and use *positioning*. It should be mentioned that each wing that we have found during our project has a well-formed classical structure. The only wings that we have seen that have not this well-formed structure, were the wings that we have developed ad hoc to test our framework. Last but no least, the wing transformation constraint guarantees that our wing parameters are consistent with the displayed wing (see also section 6.2). In a sense, the *wing transformation* becomes an anchor that sets the position of the wing.

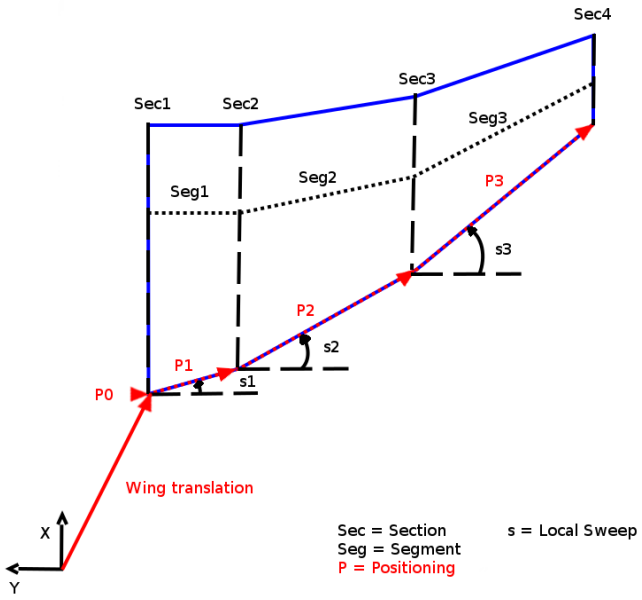


Figure 21: In Creator standard, *positioning sweep* and *dihedral* overlap the sweep and dihedral of the segment

Aside of adding desirable properties, the creator standard stores all the information in the same structure. Once a file is standardized, the user is certain where the data is. It is thus not needed to analyze the whole wing to know where the desired data is.

8.1 Creator Standardization Algorithms

In this subsection we will present the algorithm that we used to convert any CPACS wing into a Creator standard wing. This means that we can create for any wing an equivalent wing description, with the same final wires and connections as the original wing, but that follows the Creator standard discussed above.

8.1.1 Standardization of *wingAirfoils*

The idea was to apply an affine transform on the *wingAirfoil* points to standardize them and then to apply the inverse of this transform on the *wingElement* so that the final shape is unchanged. See figure 22 and the algorithms 2 and 3.

Algorithm 2 Standardize one *wingAirfoil*

1: **function** STANDARDIZEWINGAIRFOIL($airfoilUID$)
2: $\mathcal{L}_{\leftarrow}$ Get the airfoil leading edge
3: $\mathcal{T}_{\leftarrow}$ Get the airfoil trailing edge
4: $t \leftarrow \mathcal{L}$
5: Translate all points of the *wingAirfoil* by t
6: $r \leftarrow$ The rotation such that the $\mathcal{T}$ is on the X axis
7: Rotate all points of the *wingAirfoil* by r
8: $s \leftarrow$ The scaling such that the length form $\mathcal{L}$ to $\mathcal{T}$ is equal to 1
9: Scale all points of the *wingAirfoil* by s
10: **return** The Augmented matrix that represent the affine transformation described by t, then r, then s
11: **end function**

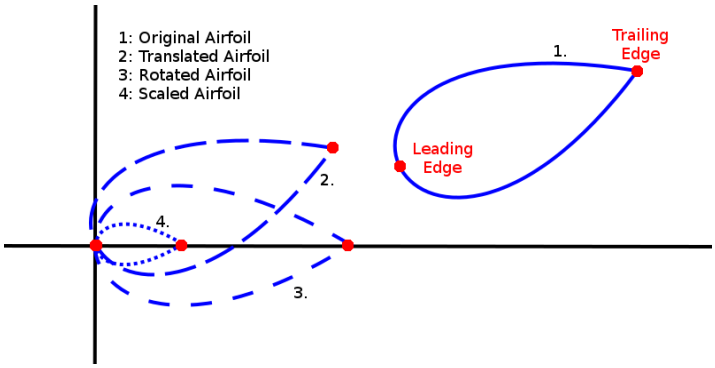


Figure 22: The affine transforms applied on the airfoil during the standardization

Algorithm 3 Standardize all *wingAirfoils* used in a wing

1: **function** STANDARDIZEWINGAIRFOILS($wingUID$)
2: **for** all *wingElement* e in the wing **do**
3: $a \leftarrow$ Get the associated *wingAirfoil*
4: $m \leftarrow$ standardizeWingAirfoil(a.uid)
5: $mi \leftarrow$ get the inverse of m
6: Apply mi on the *wingElement transformation*
7: **end for**
8: **end function**

Remark that in the real implementation of the algorithm 3 at line 4, we do not override the existing *wingAirfoil* because another wing may use it, but we create an new *wingAirfoil* and change the reference in the *wingElement*.

8.1.2 One *wingElement* per *wingSection* standardization

The algorithm 4 modify the wing such that there is one *wingElement* per *wingSection*. If a new section is needed, we create it and make sure that the final wires and connections remain the same.

Algorithm 4 One *wingElement* per *wingSection*

```
1: function oneElementPerSection(wingUID)
2:     for all wingSection s in the wing do
3:         n ← The number of wingElement in s
4:         if n = 0 then
5:             delete the wingSection
6:         else if n > 1 then
7:             for all wingElement e in s except the first one do
8:                 s_new ← Create a new wingSection in the wing
9:                 e_new ← Create a new wingElement in s_new
10:                m ← the global affine matrix applied on e
11:                Set s_new transformation with a trivial transformation
12:                Set e_new transformation such that e_new has the same global matrix of m
13:                e_new.uid ← e.uid
14:                delete e
15:            end for
16:        end if
17:    end for
18: end function
```

Since we keep the same *wingElement* uid we are certain that the connection between *wingElement* are the same as in the original wing. Note that we guarantee that the final wire is at the same position by setting the *transformation* in the new *wingElement*. This implies that maybe the rotation and translation part of the transformation is not so trivial as in the standard. But, we will manage the repartition of the affine transformations in the different CPACS component later in 'splitGloblMatrix'.

8.1.3 Standardization of *wing transformation*

We want to set the wing *transformation* such that its scaling is trivial, its rotation is trivial or has a unique rotation of 90 degrees around X and such that the translation brings the leading edge of the root to its final position. To achieve this, we first determine the needed wing *transformation* (algorithm 5, then we it and modify the *wingElements* such that the final shape stays the same (algorithm 6).

Algorithm 5 Determine the ideal wing *transformation*

```
1: function determineWingTransformation(wingUID)
2:     t ← Create a trivial transformation
3:     dihedral ← Compute the world dihedral of the wing
4:     if dihedral > 45 then
5:         Set t rotation to 90 degree around X
6:     end if
7:     L_root ← Get the leading edge of the root in the world coordinate system
8:     Set t translation to L_root
9:     return t
10: end function
```

The trick is to use the world coordinate system to find the values of the ideal *transformation*. The world dihedral uses the same definition as the definition of the dihedral but in the world coordinate system. We can compute in the same way as the dihedral angle, see section 9.

Algorithm 6 Set the *transformation* of the wing

1: **function** SETWINGTRANSFORMATION($wingUID$)
2: $W_{new} \leftarrow$ determineWingTransformation($wingUID$)
3: **for** all *wingElement* e **do**
4: $chain[W, P, S, E] \leftarrow$ getTransformationChain(e)
5: $m_{new} \leftarrow S^{-1} * P^{-1} * W_{new}^{-1} * W * P * E$
6: Set the *transformation* of e with m_{new}
7: **end for**
8: Set *wing transformation* with W_{new}
9: **end function**

At line 5, we find a new *wingElement transformations* that correct the effect of the new *wing transformation*. We know that the final affine transform should be the same, so we can use the following equivalence to find the new *wingElement transformation* E_{new}

$$W * P * S * E = W_{new} * P * S * E_{new}$$
$$S^{-1} * P^{-1} * W_{new}^{-1} * W * P * S * E = E_{new}$$

8.1.4 Split global affine transform standardization

The goal is to assign to each component a specific part of the affine transform. The *wingElement* will be responsible for the scaling part, the *wingSection* for the rotation part and the *positioning* for the translation part.

Algorithm 7 Split global matrix

1: **function** SPLITGLOBALMATRIX($wingUID$)
2: $W \leftarrow$ get the *wing transformation*
3: Delete all positionings
4: $graph \leftarrow$ Get the wing graph
5: $previousSec \leftarrow$ null
6: $previousT = 0$
7: **for** all *wingElement* e in graph order **do**
8: $G \leftarrow$ getGlobaltransformMatrix(el)
9: $SemiG \leftarrow W^{-1} * G$
10: Decompose $SemiG$ into Translation t, Rotation r, Scaling s
11: Set e *transformation* with s
12: $sec \leftarrow$ Retrieve the parent section of e
13: Set sec *transformation* with r
14: $pos \leftarrow$ Create a positioning for sec
15: **if** $previousSec! = null$ **then**
16: $pos.from \leftarrow previousSec.uid$
17: **end if**
18: $pos.to \leftarrow sec.uid$
19: $deltaT \leftarrow t - previousT$
20: Set pos with $deltaT$
21: $previousSec = sec$
22: $previousT = t$
23: **end for**
24: **end function**

There are two requirements for this algorithm. First there must be one *wingElement* per *wingSection*. Otherwise, it is impossible to guarantee the translation only using a *positioning*. Secondly, the wing

must have a graph order. In the case of the wing without a well well-structured wing (see section 7.3), we use a similar algorithm, but we put the translation into the *wingElements*.

8.1.5 Complete standardization

Once we implemented the functions above it is straightforward to transform any wing into the Creator standard. We simply call the functions in the following order:

‘standardizeWingAirfoils‘, ‘oneElementPerSection‘, ‘setWingTransformation‘, ‘SplitGlobalMatrix‘.

9 High level aircraft parameters algorithms

In this section we will discuss the algorithms developed to retrieve and to modify the parameters discussed in section 6. These algorithms and functions work with any CPACS file. But if the CPACS file is not standardized, the *wing transformation* might have a scaling factor and the parameters might not have the values expected by the user. We therefore strongly recommend to use these algorithms with a standardized *wing transformation*.

Remember that the wing parameters are defined in the wing coordinate system. When we speak about position, scaling or rotation in the algorithm or in the text, we implicitly speak about the position, scaling or rotation in the wing coordinate system if nothing else is specified.

9.1 Sweep Angle

9.1.1 Get Wing Sweep

To retrieve the wing sweep, the following algorithm can be deduced from the definition.

Algorithm 8 Get Wing Sweep Angle

1: **function** GETWINGSWEEP($wingUID$)
2: $\mathcal{L}_{root} \leftarrow$ Get the leading edge of the root
3: $\mathcal{L}_{tip} \leftarrow$ Get the leading edge of the tip
4: $rootToTip \leftarrow \mathcal{L}_{tip} - \mathcal{L}_{root}$
5: $rootToTip' \leftarrow$ Project $rootToTip$ on the XY plane
6: $sweep \leftarrow$ Compute the angle between the $rootToTip'$ and the unit vector Y.
7: **return** *sweep*
8: **end function**

In step 2 and 3, we use the wing graph representation to find the root and the tip *wingElement*. Then we use a TIGL function to get the leading edge and trailing edge of this *wingElement*.

This algorithm can be easily generalized to support the computation of the sweep at a given chord percentage. We simply use the $\mathcal{C}_{x,}$ instead of the $\mathcal{L}$. In fact, all the algorithms for sweep and dihedral can be generalized by replacing $\mathcal{L}$ by $\mathcal{C}_{x,}$ in the right place. We will not mention it again, but the reader should keep in mind that a straight forward generalization version of the algorithm for sweep and dihedral exist.

9.1.2 Set Wing Sweep

According to our definition of the sweep angle there is only one possible sweep angle per wing. However, there are multiple ways to change the position of *wingElements* to obtain a desired sweep angle. Here we will present 2 methods that we implemented to set the sweep angle. For both methods we decided to keep the root leading edge at the same position and the Y and Z coordinates of the tip leading edge at the same value. With these two constraints there is only one possible position for the leading edge tip. But the other parameters are still free. In fact, our definition of the sweep angle only requires that the tip leading edge is correctly set. For example, if there are *wingElements* between the root and the tip, they can be placed randomly. Or the root *wingElement* can be randomly scaled and rotated as long as the leading edge stays at the correct position.

Method 1: keep the scale and rotation of each wire

The idea was to keep the shape of the wires and to change only their position in the 3D space. Thus we keep the same global scaling and the rotation factors for each wire. This implies that the root does not change at all. Furthermore, this implies that the final tip position and scaling is unique. The leading edge tip moves in X, but not in Y and Z, so we can interpret this motion as a X shear. For the central wires we decided to place their leading edges by performing the same shearing as for the leading tip. Thus we have now for each wire its position, its scale and its rotations. It's important to note that the shearing is done only on the leading edge and not on the whole wing.

Algorithm 9 Set Wing Sweep Angle By Translations

1: **function** SETWINGSWEEPBYTRANSLATION(*wingUID*, *sweepAngle*)
2: $\mathbb{E}\mathbb{s} \leftarrow$ Find all *wingElement* used in this wing
3: $\mathbb{L}\mathbb{s} \leftarrow$ For each *wingElement* in $\mathbb{E}\mathbb{s}$ get its leading edge point $\mathcal{L}$
4: $\mathcal{L}_{root} \leftarrow$ Get the root leading edge.
5: $\mathcal{L}_{oldTip} \leftarrow$ Get the tip leading edge.
6: $\mathcal{L}_{newTip} \leftarrow$ Compute the new tip leading edge point based on the *sweepAngle*, the $\mathcal{L}_{root}$ and the $\mathcal{L}_{oldTip}$.
7: $Shear \leftarrow$ Compute the shearing needed to move $\mathcal{L}_{oldTip}$ to $\mathcal{L}_{newTip}$
8: **for** all *wingElement* e in $\mathbb{E}\mathbb{s}$ **do**
9: $\mathcal{L}_{old} \leftarrow$ the current leading edge associated with e
10: $\mathcal{L}_{new} \leftarrow Shear * \mathcal{L}_{old}$
11: $\mathcal{O}_{old} \leftarrow$ Get the current origin of e.
12: $\mathcal{O}_{new} \leftarrow \mathcal{L}_{newTip} - (\mathcal{L}_{oldTip} - \mathcal{O}_{oldTip})$.
13: Set the *transformation* of the *wingElement* e such that its origin $\mathcal{O}_e$ becomes $\mathcal{O}_{new}$
14: **end for**
15: **end function**

In step 7, the shearing is done relatively to an axis parallel to X that goes through the $\mathcal{L}_{root}$, because we want to keep $\mathcal{L}_{root}$ at the same place. In fact $Shear$ is of the form: $T^{-1}ShearT$, where T is a translation that bring the X axis to $\mathcal{L}_{root}$.

For step 11 it should be mentioned that in section 7.2 we discussed that all transformations performed on a *wingElement* can be expressed in the form $WPSE$. To find out the position of the origin of the *wingElement*, $\mathcal{O}_e$, in the wing coordinate system, we can simply calculate: $\mathcal{O}_e = PSE * 0$, where $0 = [0, 0, 0, 1]^T$ is the homogeneous origin.

In step 12, since the rotation and scaling factors in the wing remain the same, the vector from $\mathcal{O}_{old}$ to $\mathcal{L}_{old}$ will also be the same as the vector from $\mathcal{O}_{new}$ to $\mathcal{L}_{new}$. We use this to compute the $\mathcal{O}_{new}$.

In step 13 we need to find the *transformation* of a *wingElement* e such that the *wingElement* origin, $\mathcal{O}_e$, is at the desired position, say $\mathcal{O}_{new}$. We also want to keep the rotation and scaling factors. If we use a homogeneous coordinates system, we have:

$$\mathcal{O}_{new} = PSE'_e * 0 \tag{1}$$
$$S^{-1}P^{-1}\mathcal{O}_{new} = E'_e * 0 \tag{2}$$
$$S^{-1}P^{-1}\mathcal{O}_{new} = T'_e R'_e S'_e * 0 \tag{3}$$
$$S^{-1}P^{-1}\mathcal{O}_{new} = T'_e * 0 \tag{4}$$
$$\mathcal{O}'_{new} = t_{new} \tag{5}$$
$$\Rightarrow E'_e = T_{new} R_e S_e \tag{6}$$
$$\tag{7}$$

Where E'_e is the new wanted matrix for e and 0 is the homogeneous origin, $0 = [0, 0, 0, 1]^T$. As shown in 2, we are certain that the inverses exist because all matrices are valid affine transformations. In 3,

E'_e can be decomposed as a scaling S'_e, a rotation R'_e and a translation T'_e by construction of E_e in the CPACS format. In 4, R'_e and S'_e have no effect on 0, because 0 is the origin. Therefore, we can deduce that the required translation t_{new} and since the R_e and S_e do not change we have all the components of the new *transformation* of the *wingElement* e.

Properties of method 'setWingSweepByTranslation'

The method described above has some interesting properties. First of all, since the root does not move, we are certain that the wing stays correctly attached to the fuselage. Secondly, the span is unchanged. The span is the distance in Y between the two extremities. But by construction, the *wingElements* can only be translated in the X direction and as a result the wing span is unchanged. But the wing area, A_{xy} might change, and this implies that the aspect ratio can also change. In fact, a proper shearing conserves the A_{xy}, but we have performed the shearing only on the leading edges and not on the whole wing. However, the special wing configuration where the chords of the *wingElement* are parallel to the X axis, keeps the A_{xy}, because the shearing effect is reproduced by the translation. The front area, A_{yz}, is conserved since only a translation on X axis is performed.

Method 2: Keep A_{xy} area

One of the issues of the first method is that the wing area A_{xy} is not alway conserved. A_{xy} and the aspect ratio play a very important role in the generation of lift of the aircraft, and the aircraft designer often wants to keep the modification of the sweep independent of the modification of the A_{xy}. In this method we try to set the wing sweep by a shearing of the whole wing. We know that a shearing along X would keep A_{xy}. If the CPACS format would have the possibility to have a shearing on the *wing transformation*, we would be basically done. But this is not the case. The order, the number and the type of the available affine transformations are very strictly defined in CPACS. In this method we present a way to keep the A_{xy} area by performing a shearing on the quadrilateral wing.

Algorithm 10 Set Wing Sweep Angle By Shearing

1: **function** SETWINGSWEEPBYSHEARING($wingUID$, $sweepAngle$)
2: $\mathbb{E}\mathbb{s} \leftarrow$ find all *wingElement* used in this wing
3: $\mathbb{L}\mathbb{s} \leftarrow$ for each *wingElement* in $\mathbb{E}\mathbb{s}$ get its leading edge point $\mathcal{L}$
4: $\mathcal{L}_{root} \leftarrow$ Get the root leading edge.
5: $\mathcal{L}_{oldTip} \leftarrow$ Get the tip leading edge.
6: $\mathcal{L}_{newTip} \leftarrow$ Compute the new tip leading edge point based on the $sweepAngle$, the $\mathcal{L}_{root}$ and the $\mathcal{L}_{oldTip}$.
7: $Shear \leftarrow$ Compute the shearing needed to move $\mathcal{L}_{oldTip}$ to $\mathcal{L}_{newTip}$
8: **for** all *wingElement* e in $\mathbb{E}\mathbb{s}$ **do**
9: $chain[WPSE] \leftarrow$ getTransformationChain(e)
10: $\mathcal{L}_{old} \leftarrow$ the current leading edge associated with e
11: $\mathcal{L}_{new} \leftarrow Shear * \mathcal{L}_{old}$
12: $\mathcal{T}_{old} \leftarrow$ the current trailing edge associated with e
13: $\mathcal{T}_{new} \leftarrow Shear * \mathcal{L}_{old}$
14: $\mathcal{L}_{ini} \leftarrow E^{-1} * S^{-1} * P^{-1} * \mathcal{L}_{old}$
15: $\mathcal{T}_{ini} \leftarrow E^{-1} * S^{-1} * P^{-1} * \mathcal{T}_{old}$
16: $\mathcal{P}_{ini} \leftarrow \mathcal{L}_{ini} + UnitVectorZ \times (\mathcal{T}_{ini} - \mathcal{L}_{ini})$
17: $\mathcal{L}'_{new} \leftarrow S^{-1}P^{-1} * \mathcal{L}_{new}$
18: $\mathcal{T}'_{new} \leftarrow S^{-1}P^{-1} * \mathcal{T}_{new}$
19: $\mathcal{P}' \leftarrow S^{-1}P^{-1} * Shear * P * S * E * \mathcal{P}_{ini}$
20: $E_{new} \leftarrow$ call FindTransformation($\mathcal{L}'_{new}$,$\mathcal{L}_{ini}$,$\mathcal{T}'_{new}$,$\mathcal{T}_{ini}$,$\mathcal{P}'$, $\mathcal{P}$)
21: *transformation* of the *wingElement* $e \leftarrow E_{new}$
22: **end for**
23: **end function**

Until step 7, the algorithm is similar to the previous algorithm 9.

In step 11 and 13, we obtain the new desired values for the quadrilateral wing.

We want to modify only the *transformation*, E, of *wingElements* to obtain the desired transformation. First notice that we can isolate the effect of E. For a point $\mathcal{X}_{init}$ of the *wingAirfoil*, we would ideally have:

$$\begin{aligned}
\mathcal{X}_{new} &= PSE^i_{new} * \mathcal{X}_{ini} \\
\& \ \mathcal{X}_{new} &= Shear * \mathcal{X}_{old} \\
\& \ \mathcal{X}_{old} &= PSE_{old} * \mathcal{X}_{ini} \\
&\Rightarrow \\
\mathcal{X}_{new} &= Shear * PSE_{old} * \mathcal{X}_{ini} \\
PSE^i_{new} * \mathcal{X}_{ini} &= Shear * PSE_{old} * \mathcal{X}_{ini} \\
E^i_{new} * \mathcal{X}_{ini} &= S^{-1}P^{-1} * Shear * PSE_{old} * \mathcal{X}_{ini}
\end{aligned}$$

We have thus the ideal transformation matrix, E^i_{new}, and we can calculate the effect of this transformation on each point. This is the mathematical background for the calculation of the points from step 14 to 19.

But we also know that the E^i_{new} contains probably a none null shearing and E^i_{new} must be decomposable in $E^i_{new} = TRS_e$ where T is a translation, R is a rotation and S_e is a scaling. These two constraints are impossible to satisfy at the same time. On another side, the points of the *wingAirfoil* are only described in XZ-plane, thus all the interesting points $\mathcal{X}_{ini}$ are on the XZ plane. So, we relax the constraints in the following way to create the *transformation* E_{new} in step 20. The two particular points $\mathcal{L}_{ini}$ and $\mathcal{L}_{ini}$ must satisfy the equation $E^i_{new}\mathcal{X}_{ini} = E_{new}\mathcal{X}_{ini}$ (1) and the points on the XZ plane are on the image of the XZ plane (2), but not necessary at the correct position. To find the transformation $E_{new} = TRS_e$ that satisfy (1) and (2), we have to construct E_{new} by the following algorithm:

Algorithm 11 Find transformation

1: **function** FINDTRANSFORMATION($\mathcal{L}'$,$\mathcal{L}$,$\mathcal{T}'$,$\mathcal{T}$, $\mathcal{P}'$,$\mathcal{P}$)
2: $C' \leftarrow$ center of segment form $\mathcal{L}'$ to $\mathcal{T}'$
3: $C \leftarrow$ center of segment form $\mathcal{L}$ to $\mathcal{T}$
4: $T_1 \leftarrow$ translation from C' to origin
5: $T_2 \leftarrow$ translation from C to origin
6: apply T_1 on $\mathcal{L}'$, $\mathcal{T}'$ and $\mathcal{P}'$ becoming $\mathcal{L}^{ap}$, $\mathcal{T}^{ap}$ and $\mathcal{P}^{ap}$
7: apply T_2 on $\mathcal{L}$, $\mathcal{T}$ and $\mathcal{P}$ becomming $\mathcal{L}^a$, $\mathcal{T}^a$ and $\mathcal{P}^a$
8: $N' \leftarrow$ vector perpendicular of the plane formed by $\mathcal{L}'$, $\mathcal{T}'$ and $\mathcal{P}'$
9: $N \leftarrow$ vector perpendicular of the plane formed by $\mathcal{L}$, $\mathcal{T}$ and $\mathcal{P}$
10: $R_1 \leftarrow$ find a rotation such that N' is bring to N
11: $R_2 \leftarrow$ find a rotation around the N axis such that the vector for origin to $R_1 * \mathcal{L}^{ap}$ is bring to the vector form origin to $\mathcal{L}^a$.
12: Find the scaling, S, that bring the point $R_2R_1 * \mathcal{L}^{ap}$ to $\mathcal{L}^a$
13: $S_f \leftarrow S^{-1}$
14: $R_f \leftarrow R_1^{-1}R_2^{-1}$
15: $t_f \leftarrow t_1^{-1} + R_fS_ft_2$ where t are the transformation vector of transformation matrix T
16: $E \leftarrow$ new *transformation* with t_f, R_f, S_f
17: **return** E
18: **end function**

In step 15 we use the method presented in appendix A.1.

Properties of method 'setWingSweepByShearing'

Since each leading edge and trailing edge of *wingElement* are placed with the same shearing along X, the area A_{xy} does not change. The area A_{yz} remains also unchanged. We are quite happy because this was the goal of this function. But it is important to mention that the thickness of the wing might change. In particular by the fact that the airfoil is scaled at step 12 of the algorithm 11 to get the correct $\mathcal{L}$ and TLE without any other consideration. Also the root can be detached of the fuselage when the root is not parallel to the X axis, because the shearing can create an unwanted gap. The span is conserved as for method 1.

9.2 Dihedral Angle

9.2.1 Get Wing Dihedral

To retrieve the wing dihedral, the following algorithm can be deduced from the definition.

Algorithm 12 Get Wing Dihedral

1: **function** GETWINGDIHEDRAL($wingUID$)
2: $\mathcal{L}_{root} \leftarrow$ Get the leading edge of the root
3: $\mathcal{L}_{tip} \leftarrow$ Get the leading edge of the tip
4: $rootToTip \leftarrow \mathcal{L}_{tip} - \mathcal{L}_{root}$
5: $rootToTip' \leftarrow$ Project $rootToTip$ on the YZ plane
6: $dihedral \leftarrow$ Compute the angle between the $rootToTip'$ and the unit vector Y.
7: **return** $dihedral$
8: **end function**

9.2.2 Set Wing Dihedral

As for the wing sweep there are multiple ways to modify the *wingElements* to obtain the desired dihedral angle. It seems reasonable to use a similar method as for the wing sweep, the root leading edge will stay in the same position and the tip leading is sheared to create the desired angle. Then the central leading edges are placed using the same Z shear. The leading edge positions are obtained only by translations on the wires, no scaling nor rotations are performed on them.

Algorithm 13 Set Wing Dihedral Angle By Translation

1: **function** SETWINGDIHEDRALBYTRANSLATION($wingUID$, $dihedralAngle$)
2: $\mathbb{Es} \leftarrow$ Find all *wingElement* used in this wing
3: $\mathbb{Ls} \leftarrow$ For each *wingElement* in $\mathbb{Es}$ get its leading edge point $\mathcal{L}$
4: $\mathcal{L}_{root} \leftarrow$ Get the root leading edge.
5: $\mathcal{L}_{oldTip} \leftarrow$ Get the tip leading edge.
6: $\mathcal{L}_{newTip} \leftarrow$ Compute the new tip leading edge point based on the $dihedralAngle$, the $\mathcal{L}_{root}$ and the $\mathcal{L}_{oldTip}$.
7: $Shear \leftarrow$ Compute the shearing needed to move $\mathcal{L}_{oldTip}$ to $\mathcal{L}_{newTip}$
8: **for** all *wingElement* e in $\mathbb{Es}$ **do**
9: $\mathcal{L}_{old} \leftarrow$ the current leading edge associated with e
10: $\mathcal{L}_{new} \leftarrow Shear * \mathcal{L}_{old}$
11: $\mathcal{O}_{old} \leftarrow$ Get the current origin of e.
12: $\mathcal{O}_{new} \leftarrow \mathcal{L}_{newTip} - (\mathcal{L}_{oldTip} - \mathcal{O}_{oldTip})$.
13: Set the *transformation* of the *wingElement* e such that its origin $\mathcal{O}_e$ becomes $\mathcal{O}_{new}$
14: **end for**
15: **end function**

Properties of method 'setWingDihedralByTranslation'

The root wire remains exactly the same and thus we are certain that the wing stays correctly attached to the fuselage. The span is unchanged. Since all the chord points of each wire are only translated in the Z direction, the top area A_{xy} and the aspect ratio remain unchanged. There is no need to perform a real shearing on the quadrilateral wing for the dihedral angle.

9.3 Wing Area, wing Span and wing Aspect Ratio

9.3.1 Get Wing Reference Area

Varignon[13] discovered that the area of a simple quadrilateral can be expressed in therm of the cross product of the diagonals. A quadrilateral Θ described by the points $\mathcal{L}_2, \mathcal{T}_2, \mathcal{L}_1, \mathcal{T}_1$, see figure 23, has an area of:

$$A_\Theta = 1/2 * ||(\mathcal{T}_2 - \mathcal{L}_1) \times (\mathcal{L}_1 - \mathcal{T}_1)||$$

So to compute the reference area of the wing on a plane we first project the quadrilateral wing on this plane. Then, we compute for each projected quadrilateral segments the area and finally we sum the areas. Remark for self-intersecting quadrilaterals, the Varignon formula is equal to the difference in area of the two regions the quadrilateral bounds. The implementation does for the moment not cover this case and simply uses the same formula. It should be mentioned that this case is very rare for a wing.

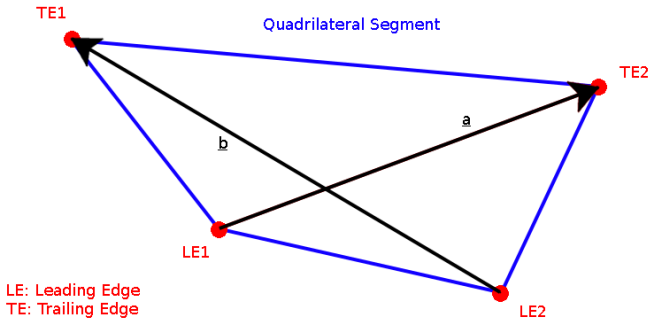


Figure 23

9.3.2 Get Wing Span

According to the definition it is very simple to get the span. First, we get the root and the tip leading edge, then we calculate the difference between these two points in Y.

9.3.3 Get Wing Aspect Ratio

The aspect ratio derives from the span and the top area. Once we have the span and the area we have the aspect ratio.

9.3.4 Edition of the Aspect Ratio, Top Area, Span

We observe that these three parameters influence each others. We have:

$$
\begin{aligned}
AR &= b^2/A_{XY} \\
A_{XY} &= b^2/AR \\
b &= \sqrt{AR * A_{XY}}
\end{aligned}
$$

If we want to modify the AR then at least one of the b or A_{XY} needs to change. It is thus impossible to modify one of these parameter independently from the two others. Ideally, we would keep the modification of one parameter decoupled from the other parameter. Since it is impossible for these three parameters, we decided to set the parameter value by changing one of the two other parameters and to keep the other constant. Thus we have for each parameter two modifications functions:

- 'setWingAreaKeepAR'
- 'setWingAreaKeepSpan'
- 'setWingSpanKeepAR'
- 'setWingSpanKeepArea'
- 'setWingARKeepSpan'
- 'setWingARKeepArea'

9.3.4.1 Set Wing Area Keep Aspect Ratio

If we scale the wing uniformly by a scale factor of s in the wing coordinate system, then the new area A_{XY} can be calculated from:

$$
\begin{aligned}
A_{XY,new} &= \sum_{u \in \{segments\}} A_{XY,new,u} && (8) \\
A_{XY,new} &= \sum_{u \in \{segments\}} 0.5 * ||(\bar{c}_{u,new}) \times (\bar{d}_{u,new})|| && (9) \\
A_{XY,new} &= \sum_{u \in \{segments\}} 0.5 * ||(s * \bar{c_u}) \times (s * \bar{d_u})|| && (10) \\
A_{XY,new} &= \sum_{u \in \{segments\}} s^2 * 0.5 * ||(\bar{c_u} \times \bar{d_u})|| && (11) \\
A_{XY,new} &= s^2 \sum_{u \in \{segments\}} 0.5 * ||(\bar{c_u} \times \bar{d_u})|| && (12) \\
A_{XY,new} &= s^2 A_{XY,old} && (13) \\
& && (14)
\end{aligned}
$$

Where the c and d are the projections of the two diagonals of a quadrilateral segment on the XY plane. Variables without the explicit flag "new", are variables of the wing before the scaling.

For a uniform scaling the aspect ratio AR remains the same, we have:

$$AR_{new} = b_{new}^2 / A_{XY,new} \tag{15}$$
$$AR_{new} = (s * b)^2 / s^2 A_{XY,old} \tag{16}$$
$$AR_{new} = s^2 * b^2 / s^2 A_{XY,old} \tag{17}$$
$$AR_{new} = b^2 / A_{XY,old} \tag{18}$$
$$AR_{new} = AR_{old} \tag{19}$$
$$\tag{20}$$

In 15, the span is the norm of a vector, so the new span is simply the old span scaled.

The sweep angle and the dihedral angle are also preserved. Both can be viewed as vector. When making a uniform scaling the vectors keep the same direction, having the same angles relatively to a plane.

To summarize, by performing a uniform scaling of the entire wing the aspect ratio, the sweep angle and the dihedral angle remain the same. Therefore we decided to set the area by scaling the entire wing. To find the scale factor, we use:

$$A_{XY,new} = s^2 A_{XY,old} \tag{21}$$
$$s = \sqrt{A_{XY,new} / A_{XY,old}} \tag{22}$$

The algorithm to set the wing area keeping the aspect ratio is then straightforward.

Algorithm 14 Set Wing Area Keep Aspect Ratio

```
1: function setWingAreaKeepAR(wingUID, newArea)
2:     oldArea ← GetWingArea(wingUID, planeXY)
3:     s ← √(newArea/oldArea)
4:     ScaleWingUniformly(s)
5: end function
```

To scale uniformly the wing in the wing coordinate system, we use the following algorithm.

Algorithm 15 Scale Wing Uniformly by s

```
1: function ScaleWingUniformly(wingUID, s)
2:     ScaleM ← s * IdentyM
3:     for each wingElement e in wing do
4:         chain[WPSE] ← getTransformationChain(e)
5:         newG ← W * ScaleM * PSE
6:         newE ← S^-1 P^-1 W^-1 * newG
7:         set transformation of e with newE
8:     end for
9: end function
```

9.3.4.2 Set Wing Area Keep Span

The area of a segment u can be expressed by its leading edges and its trailing edges:

$$A_{XY,u} = 0.5 * ||(\mathcal{T}_1 - \mathcal{L}_2) \times (\mathcal{T}_2 - \mathcal{L}_1)|| \tag{23}$$

To avoid overcharging the syntax we omit the fact that the $\mathcal{L}_i$ and $\mathcal{T}_i$ are projected on the XY plane. In fact we only evaluate $\mathcal{L}_i$ and $\mathcal{T}_i$ in X and Y and we set Z = 0.

If we define (see figure 24):

$$a = \mathcal{L}_2 - \mathcal{L}_1 \tag{24}$$
$$b = \mathcal{T}_1 - \mathcal{L}_1 \tag{25}$$
$$c = \mathcal{T}_1 - \mathcal{L}_1 \tag{26}$$
$$\tag{27}$$

We have :

$$A_{XY,u} = 0.5 * ||(\mathcal{T}_1 - \mathcal{L}_2) \times (\mathcal{T}_2 - \mathcal{L}_1)|| \tag{28}$$
$$A_{XY,u} = 0.5 * ||(\mathcal{L}_1 + b - \mathcal{L}_1 - a) \times (\mathcal{L}_1 + a + c - \mathcal{L}_1)|| \tag{29}$$
$$A_{XY,u} = 0.5 * ||(b - a) \times (a + c)|| \tag{30}$$
$$A_{XY,u} = 0.5 * ||(b - a) \times a + (b - a) \times c|| \tag{31}$$
$$A_{XY,u} = 0.5 * ||b \times a - a \times a + b \times c - a \times c|| \tag{32}$$
$$A_{XY,u} = 0.5 * ||b \times a + b \times c - a \times c|| \tag{33}$$
$$A_{XY,u} = 0.5 * ||b \times a + b \times c + c \times a|| \tag{34}$$

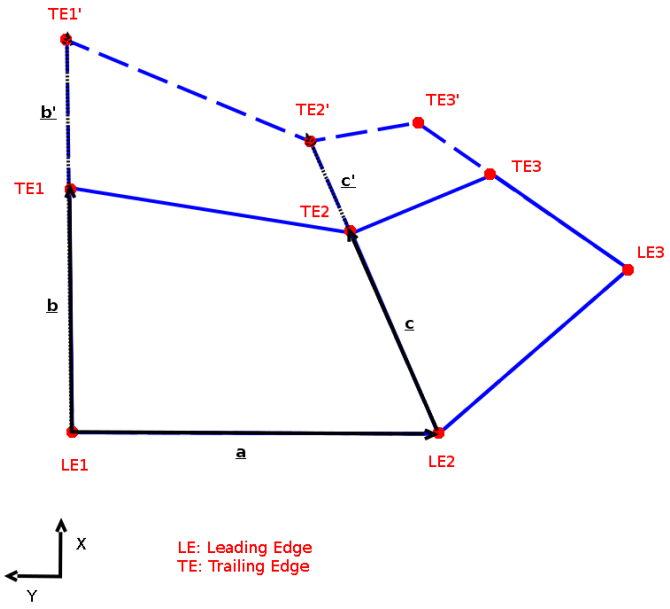


Figure 24: Vector between leading edges and trailing edges

The idea is to keep the leading edges at the same position and scale the chord by a unique factor θ that brings the area to the desired value. Be keeping the leading edges at the same position, we are certain that the span, the sweep angle and the dihedral angle remain the same. The scale factor θ, will scale the value of each chord. The chord is represented by the vectors b and c. In figure 24, the factor θ transforms b into b' and c into c'. We solve the following equation to find θ:

$$A_{XY} = \sum_{u \in S} 0.5 * ||(\theta b \times a) + (\theta b \times \theta c) + (\theta c \times a)|| \tag{35}$$

$$2A_{XY} = \sum_{u \in S} ||\theta(b \times a) + \theta^2(b \times c) + \theta(c \times a)|| \tag{36}$$

$$2A_{XY} = \sum_{u \in S} ||\theta((b + c) \times a) + \theta^2(b \times c)|| \tag{37}$$

$$2A_{XY} = \sum_{u \in S} ||\theta v + \theta^2 w|| \tag{38}$$

$$2A_{XY} = \sum_{u \in S} \theta ||v + \theta w|| \tag{39}$$

$$2A_{XY} = \theta || \sum_{u \in S} \{v + \theta w\}|| \tag{40}$$

$$2A_{XY} = \theta || \sum_{u \in S} \{v\} + \theta \sum_{u \in S} \{w\}|| \tag{41}$$

$$2A_{XY} = \theta ||g + \theta h|| \tag{42}$$

$$2A_{XY} = \theta \sqrt{(g_x + \theta h_x)^2 + (g_y + \theta h_y)^2 + (g_z + \theta h_z)^2} \tag{43}$$

$$(2A_{XY})^2 = \theta^2[(g_x + \theta h_x)^2 + (g_y + \theta h_y)^2 + (g_z + \theta h_z)^2] \tag{44}$$

$$(2A_{XY})^2 = \theta^2[(g_x^2 + g_y^2 + g_z^2) + \theta(2g_x h_x + 2g_y h_y + 2g_z h_z) + \theta^2(h_x^2 + h_y^2 + h_z^2)] \tag{45}$$

$$(2A_{XY})^2 = \theta^2[B + \theta C + \theta^2 D] \tag{46}$$

$$(2A_{XY})^2 = \theta^2 B + \theta^3 C + \theta^4 D \tag{47}$$

$$0 = D\theta^4 + C\theta^3 + \theta^2 B - 4A_{XY}^2 \tag{48}$$

In step 38, 42,46 we simply merge the terms that we know. In step 40, we observe that b,c,a are on the XY plane, so v and w will be collinear of the unit vector Z, and the area is defined in the same sense. These two facts imply that the sum of the norms of the vectors are equal to the norm of the sum of the vectors.

With the equation above we can express the searched scale factor θ in the form of a quartic equation. Quartic equations are a well-known problem that we can solve using Descartes-Euler method. We use a code implemented by Sergey Khashin to find the roots. These roots can be complex and negative, but for our problem we are only interested by positive and real roots. But, it is possible to have multiple available roots. For not simple quadrilateral, the Varignon formula (used to compute the area) returns the difference in area of the two regions of the quadrilateral. So the equation can find weird solutions where the chords are crossing each others. The appropriate root for our solution is the smallest real and positive root.

We now have the desired scale factor θ. This θ is the scale factor to be apply on each projected chord. We want to keep the direction of the chords in wing coordinate system. To keep the direction of the chords, we use the fact that the ratio between the projected chord on XY plane and the Z component of the chord needs to remain the same.

Finally we can set the wing area while keeping the span by the following algorithm.

Algorithm 16 Set Wing Area keep Span

1: **function** SetWingAreaKeepSpan($wingUID$, $area$)
2: $\theta \leftarrow$ find scale factor using quartic equation
3: **for** each *wingElement* e in wing **do**
4: $\mathcal{L}_e \leftarrow$ Get the leading in wing Coordinate system
5: $\mathcal{T}_e \leftarrow$ Get the trailing in wing Coordinate system
6: $r \leftarrow \mathcal{L}_e[y]/\sqrt{\mathcal{L}_e[x]^2 + \mathcal{L}_e[y]^2}$
7: $c \leftarrow projectionOnXY(\mathcal{T}_e - \mathcal{L}_e)$
8: $c' \leftarrow \theta * c$
9: $\mathcal{T}'_e[x] \leftarrow \mathcal{L}_e[x] + c'[x]$
10: $\mathcal{T}'_e[y] \leftarrow \mathcal{L}_e[y] + c'[y]$
11: $\mathcal{T}'_e[z] \leftarrow ratio * ||c'||$
12: Set *transformation* e such that $\mathcal{T}_e$ becomes $\mathcal{T}'_e$
13: **end for**
14: **end function**

In step 12, we have the new $\mathcal{T}$ and the new $\mathcal{L}$ and we can use a similar method as in the algorithm 10 to set the *transformation* of the *wingElement*.

9.3.4.3 Set Wing Span Keep Aspect Ratio

In the previous paragraphs we have seen that if we scale the wing uniformly, the aspect ratio, the sweep angle and the dihedral angle are preserved. As for 'setWingAreaKeepAR', the idea is to scale the wing such that the parameters obtain the desired value. By scaling uniformly the wing by s, we have:

$$b_{new} = s * b_{old} \tag{49}$$
$$s = b_{new}/b_{old} \tag{50}$$

So we simply call the algorithm 'ScaleWingUniformly' with the argument s.

9.3.4.4 Set Wing Span Keep Area

The idea is to first set the span by a scaling of the wing followed by setting the area to the correct value.

Algorithm 17 Set Wing Span keep Area

1: **function** SetWingSpanKeepArea($wingUID$, $span$)
2: $area \leftarrow$ GetWingArea($wingUID$)
3: SetWingSpanKeepAR($wingUID$, $span$)
4: SetWingAreaKeepSpan($wingUID$, $area$)
5: **end function**

9.3.4.5 Set Wing Aspect Ratio Keep Span

We have:

$$AR = b^2/A_{XY}$$

If we want to set the aspect ratio without changing the span b, we need to set the area A_{XY}.

$$A_{XY} = b^2/AR$$

Algorithm 18 Set Wing Aspect Ratio keep Span

1: **function** SetWingARKeepSpan($wingUID$, $newAR$)
2: $b \leftarrow$ GetWingSpan($wingUID$)
3: $newArea \leftarrow b^2/newAR$
4: SetWingAreaKeepSpan($wingUID$, $newArea$)
5: **end function**

9.3.4.6 Set Wing Aspect Ratio Keep Area

We have:

$$AR = b^2/A_{XY}$$

If we want to set the aspect ratio without changing the area A_{XY} we need to set the span:

$$b = \sqrt{AR * A_{XY}}$$

We omitted the negative option because we want a positive span.

Algorithm 19 Set Wing Aspect Ratio keep Area

1: **function** SetWingARKeepSpan($wingUID$, $newAR$)
2: $area \leftarrow$ GetWingArea($wingUID$)
3: $newB \leftarrow \sqrt{newAR * area}$
4: SetWingSpanKeepArea($wingUID$, $newB$)
5: **end function**

10 CPACSCreator GUI

The goal is to have a user friendly interface to edit the aircraft geometrical data. We chose to modify the exiting TIGLViewer software to reduce the development time as we will see in section 11. The idea was to add two more widgets to TIGLViewer, see figure 25. One widget will show a simplified Aircraft structure in the form of a tree. The other widget has an interface to modify a selected part of the aircraft structure. The user should be able to select one element of the tree by clicking on it. Each time the selection changes, CPACScreator analyzes the selection and display the available modifiable parameters and high level functions in the modification widget, see the UML use case on figure 26.

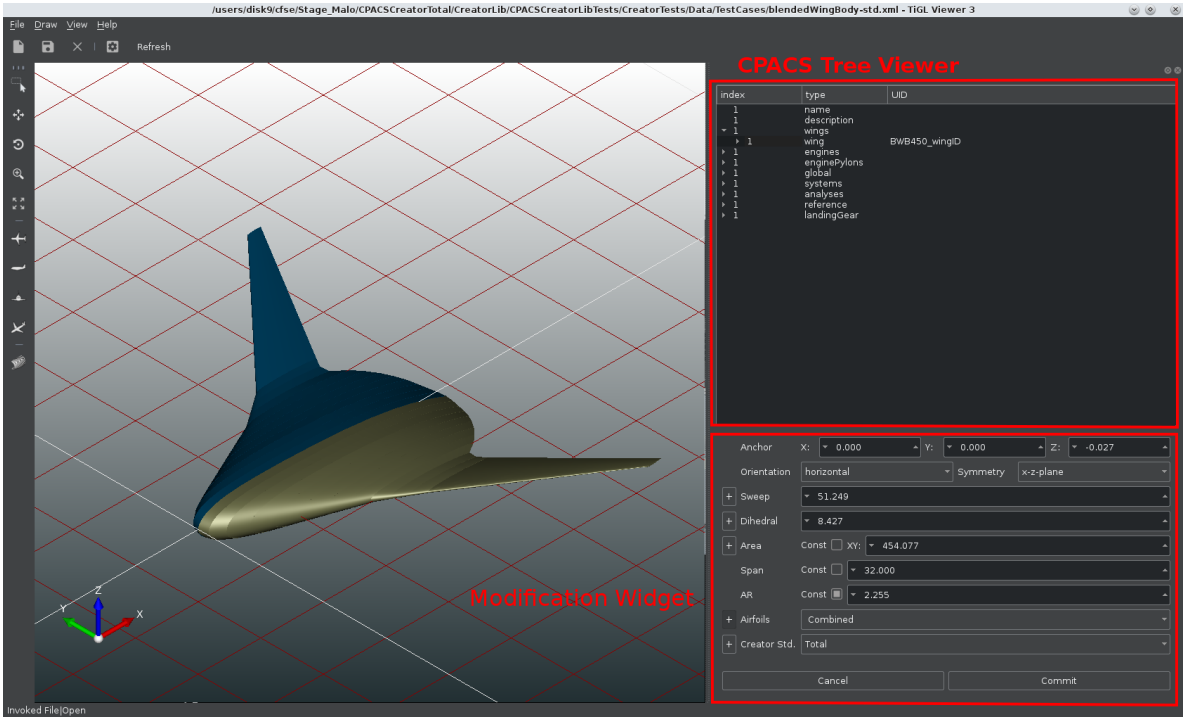


Figure 25: CPACSCreator interface

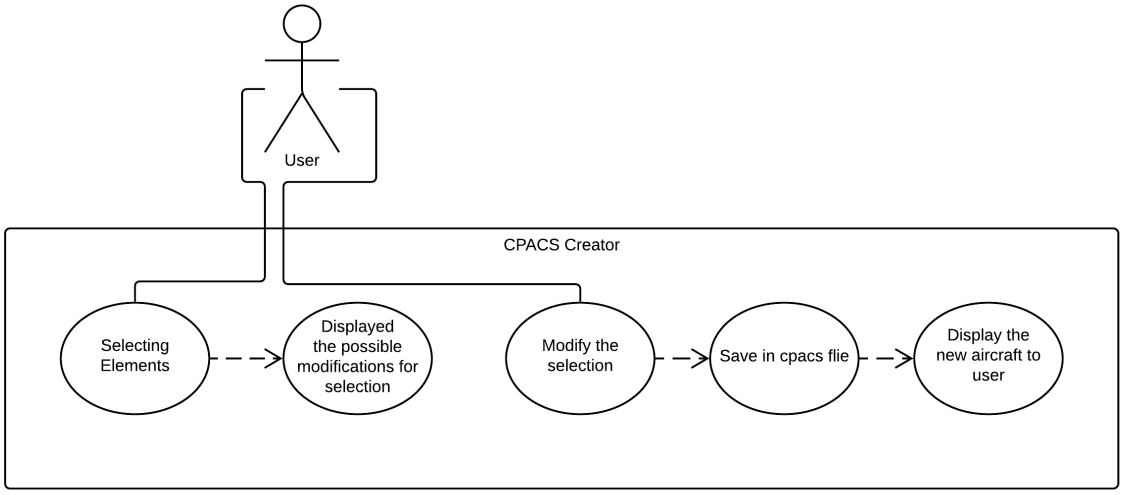


Figure 26: UML Use Case for modifications in CPACSCreator

During this project we focused our attention on the wing modification. Most of the editing functions are available by clicking on a wing. The modification widget for a wing is shown in figure 27. The idea was to have a clean, powerful and simple interface for the wing. The interface should be easy to use for a non expert aircraft designer who does not know the CPACS format. To not saturate the user with rarely used options, we mask some of the options in a first time. So when the user arrive on the wing modification widget only the principal argument of each parameters are displayed. The user can then click on the "+" button to see more available options. The figure 28 shows the modification widget for a wing with all available options.

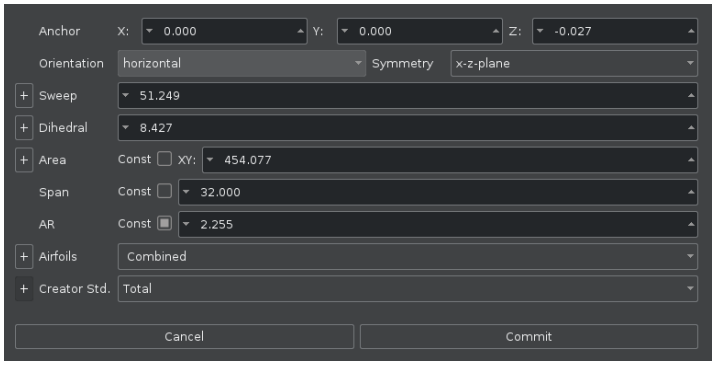


Figure 27: CPACS Creator wing modification widget

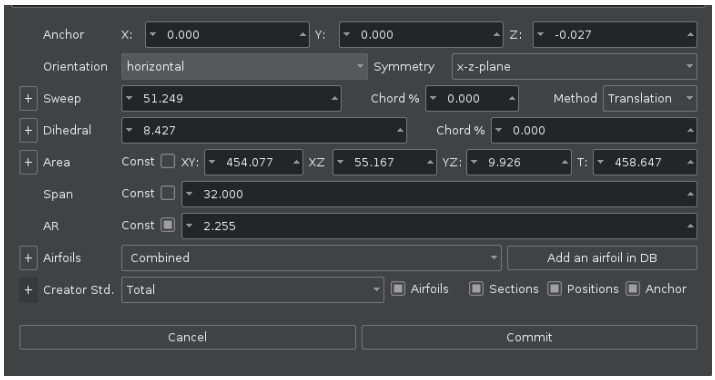


Figure 28: CPACS Creator wing modification widget with all options expended

To help the user to edit the parameter, we allow the user to modify only the parameter in a reasonable way. For example, as we have seen in section 9 the aspect ratio edition is always coupled with the span and area edition. To represent this behavior, we add the 'Const' check boxes. At every moment, one and only one 'Const' check boxes is set and the user can modify only the two other parameters and not the constant parameter. An other example of this is the creator Standard options. If we expend the creator standard option, we can not unselect a already select creator option, because there is no function to unstandardize a creator standard option.

There is currently two more modification widgets: modification widget for *transformation* elements and for *positioningS* elements. The modification widget for *transformation* simply displayed the raw data of a *transformation* and allow to edit them. Figure 29 shows the modification widget for *transformation*. The modification widget for *positioningS* retrieve all the *positioning* children and display their values. For a standardize wing, this *positioning* are in graph order and their sweep and dihedral correspond to the sweep and dihedral of the segments. So, the user can edit the sweep and dihedral for each segment. The figure 30 shows the positionings modification widget for a standardized wing.

Figure 29: CPACS Creator transformation modification widget

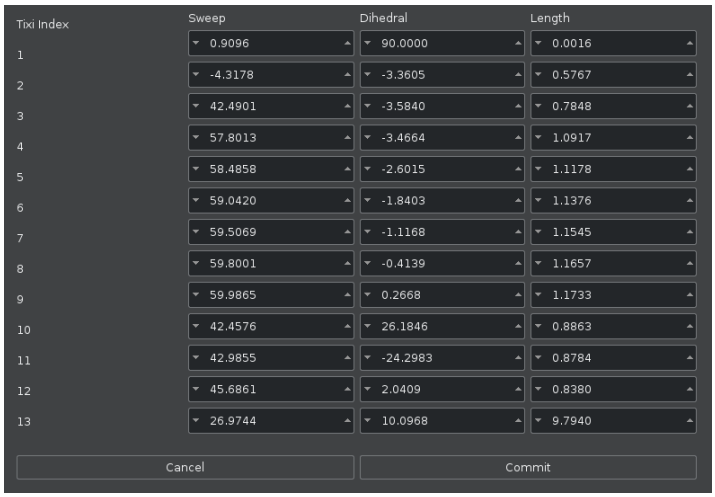


Figure 30: CPACS Creator positionings modification widget

As the fuselage is define int the same way as the wing, those two widgets become also available for the fuselage. The code architecture presented in section 11 is generic enough to automatically extend these two widgets to the fuselage.

11 Implementation

As seen in section 5, the Creator framework is composed of two modules: CPACSCreatorLib and CPACSCreator.

CPACSCreatorLib use the TIXI and TIGL libraries API. CPACSCreator is a modified version of TIGLViewer which uses the CPACSCreatorLib.

The main motivation to reuse these existing libraries was to reduce the development time. This is particularly true for the CPACSCreator GUI. If we had developed the CPACSCreator from scratch, all the generation process to build the shape would have been reimplemented. For the CPACSCreatorLib, we hesitated to directly modify the TIGL library or to use the API only. We decided to use the API only because the TIGL library is quite complex and has a strong code structure that is not really suited for what we want to achieve. Furthermore by keeping our library separated from the TIGL and TIXI libraries by the API, we can integrate future updates of the TIGL and TIXI libraries without a large effort. In fact, our CPACSCreator code has only a weak dependency to the TIGL library. We use only some functions to retrieve the chord positions in the world coordinates system. This dependency can be removed in the future by implementing a few set of functions.

The choice of modifying the TIGLViewer to create CPACSCreator leads us to write our code in C++ under Apache2 license. The main condition of the Apache2 license is that the already written code remains under the Apache2 license. Since the expertise of CFSE is in aerodynamic simulation and not in software CPACS development, we decided that our code can be also published under Apache2 license to avoid any license conflict. TIGLViewer is written in C++ therefore CPACSCreator is also written in C++. The CPACSCreatorLib could have been developed in another language with a C++ interface. We realized it is easier to allow CPACSCreator to access deep components of the CPACSCreatorLib and not only the API. So we decided to write CPACSCreatorLib also in C++ under the Apache2 license.

Figure 31 shows dependencies diagram of CPACSCreator/TIGL framework.

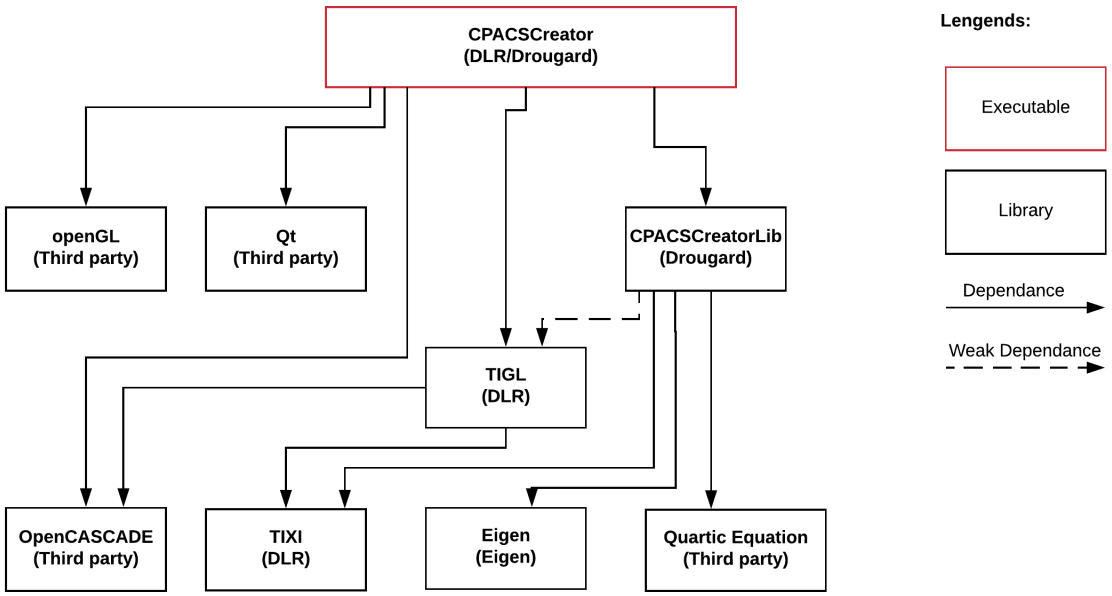


Figure 31: CPACSCreator dependencies diagram

11.1 CPACSCreatorLib Implementation

The CPACSCreatorLib implements the algorithms described in section 8 and section 9. This means that CPACSCreatorLib implements the functions to compute and edit the high level parameters and to standardize the CPACS data into the Creator Standard.

The CPACSCreatorLib code can be divided into 3 main components:

CPACSFile Class: The CPACSFile class open a CPACS file using the TIXI library functions and allows to retrieve and modify the data of the file. It can be viewed as a extension of the TIXI library. It can read some part of CPACS file and transforms the data into Basic CPACS Structure Classes instances (see the next component). CPACSFile instance can also receive Basic CPACS Structure Classes instances and write it into the CPACS file. This class is dummy in the sense that no logic is implemented in it. It only reads and writes CPACS file data and transforms it into other structures.

Basic CPACS Structure Classes: These classes represent a CPACS XSD type. They basically store a subset of CPACS data in a C++ structure and manage the representation of these data. A typical example of this is the CPACSTransformation class. This class takes the data of a *transformation* instance, stores them and allows to express the transformation as a augmented matrix. Currently there are such classes for *positioning*, *wingSegment* and *wingAirfoil*. These classes instances have no references to their position in the CPACS file.

AircraftTree Class: Aircraft tree class is the core component of the library. Aircraft tree implements the logic of the library and the algorithm discussed in section 9 and section 8. AircraftTree basically builds an aircraft tree out of a CPACS file. This tree starts at one *model* element of the CPACS file and contains all its children elements. Each node of the tree contains the XSD Type of the node, the XPath of the node and the UID of the node if the node has a UID. Then AircraftTree instance has functions to modify the tree and the referenced CPACS elements. It use a CPACSFile instance to read and write the data into the CPACS file. It use Basic CPACS Structure Classes to work with the CPACS data.

The tree structure permits us to navigate though the CPACS structure. The CPACSFile allows us to retrieve the data from CPACS and transforms it into a suitable representation. Basically a high level function call on a AircraftTree instance, navigate though the tree structure to find the need components, retrieve and transform this components into basic CPACS structures instances with its CPACSFile instance, perform some logic on the basic CPACS structures instances and the relations between them and eventually write changes in the CPACS file with its CPACSFile instance.

This code architecture is flexible, in the sense that new functions can be easily add. To add a new function the developer needs to insert the function in the AircraftTree class and implements the function with the help of the existing function. The code architecture is robust with respect to possible modifications of the CPACS format. When, for example, a new XSD type is introduced to the CPACS format the code will continue to work as long as the new type does not influence the older ones. The instances of the new XSD type will be inserted in the tree but no algorithm will use it. The code architecture is also robust with respect to not well formed CPACS file. The AircraftTree class only retrieves from CPACS the values when they are needed, so as long as the function uses a well formed part of the CPACS file, the function will continue to work correctly. There is a small drawback to this code architecture, each time a function is called it performs all the needed operations as reading some data and transform it into desired structures. These operations are often the same for multiple function calls, so the code is not really optimized. However, one function call execution time is still really reasonable because the amount of needed operations is low.

We use the Eigen library [14] to perform matrix and quaternion operations. We also use some code from Sergei Khashin [15] to solve quartic equation.

We developed a test suite using Google Test Framework [16]. For each high level function we have a test that uses different CPACS configurations.

11.2 CPACSCreator Implementation

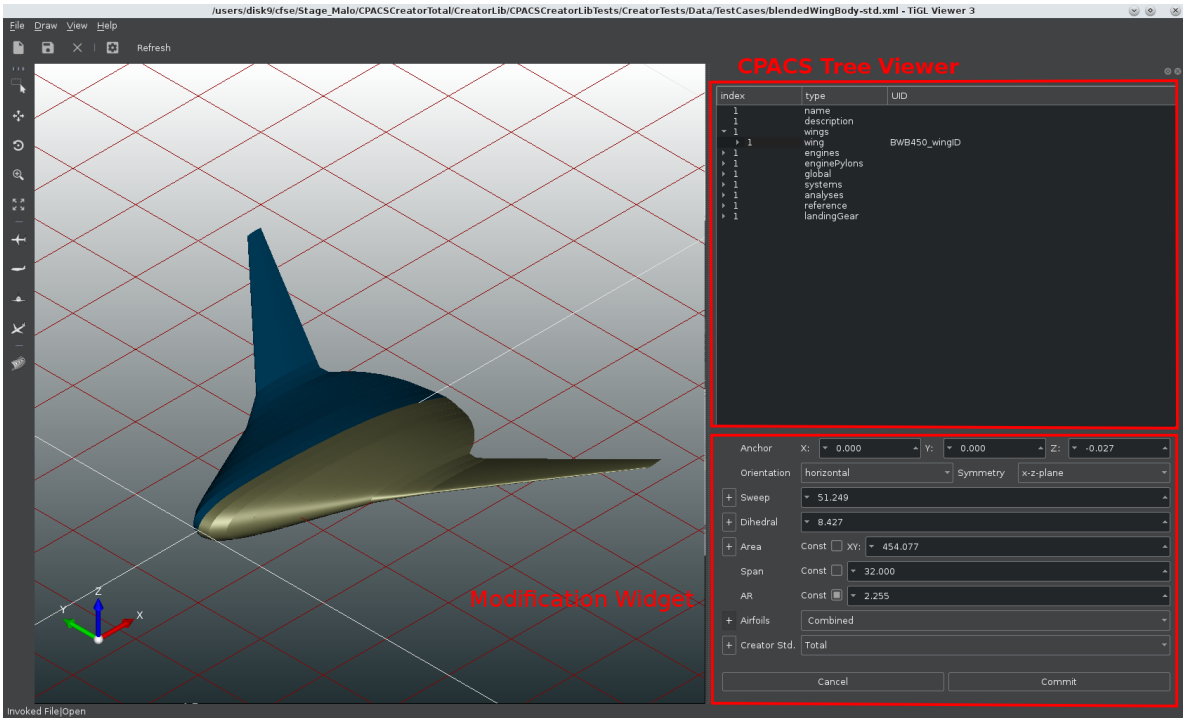


Figure 32: CPACSCreator Interface

As discussed in section 4, TIGLViewer uses the TIGL library to process CPACS data, the Qt5 framework to develop the graphical user interface and the OpenCASCADE library to build the 3 dimensional representation. We added to these 3 libraries our CPACSCreator library. The main challenge was to combine our library and code to work together with all the other codes and libraries.

The basic idea was to have a AircraftTree for each *model* of the open file. Once the AircraftTree is build the CPACS Tree Viewer displays the different elements of the tree. When the user clicks on a element of the tree, the element is analyzed and the correct Modification Widget is displayed. The Modification Widget display the possible high level functions and the current values of the parameters are retrieved from the AircraftTree. If a modification is committed the displayed aircraft needs to be rebuild and sometimes the tree needs to be updated.

On one hand the CPACS Tree Viewer needs to access the AircraftTree to display it and the Modification Widgets needs to call functions on it. On the other hand, the AircraftTree need to be synchronized with the 3D representation.

To achive this we created 3 main new classes and added an object of each class in the main window of the CPACSCreator. These classes are:

CPACSCreatorAdapter: This class embeds an AircraftTree and manages function calls on the AircraftTree. The result and arguments of a call to AircraftTree is transformed into the appropriate format and the values are checked. This class can also have no AircraftTree for a while.

CPACSAbstractModel: This class is the base class to have the tree displayed in the CPACS Tree Viewer. The Tree Viewer is a QTreeView Widget. The QTreeView widget needs a QAbstractItemModel model that stores and mange the elements fot the QTreeView. So, the CPACSAbstractModel class inherits from the QAbstractItemModel and is the model for the QTreeView. CPACSAbstractModel holds a reference to the CPACSCreatorAdapter and uses it to retrieve the AircraftTree nodes.

ModificatorManager: This class manages the different modification widgets. This class sets the appropriate modification widget. A modification widget is a object of the new class called ModificatorWidget or form its derivatives. Curently there are 3 derivatives: TIGLViewerWingWidget, TIGLViewerTransformationWidget ,TIGLViewerPositioningsWidget. These 3 classes are the modification widget for respectively, *wing* node of the tree, *positionings* node of the tree and *transformation* of node of the tree.

The Qt5 frame work uses the signals and slots mechanisms for communication between objects. A signal is emitted when a particular event occurs. A slot is a function that is called in response to a particular signal. We can connect the signal and slot at every moment, but we mainly initialize them at the start of the program or when an object is created. We use them to connect the different part of the program.

The program has basically the following behavior. When a new document is opened, a signal is emitted and a slot that updates the CPACSCreatorAdapter object is run. This slot builds a new Aircraft tree. Then the CPACSAbstractModel object receives the updated CPACSCreatorAdapter and updates the QTreeView. When a element of the QTreeView is selected, a signal is send and a slot of ModificatorManager having the selected element as argument is run. The ModificatorManager analyzes the element and if there is a ModificatorWidget for the selected element, the ModificatorManager initializes the ModificatorWidget and displays it. When the commit button is pressed, the ModificatorWidget checks if there are changes in parameters values. To do this it compares the values of the interface with the current values of the aircraft. If there is a change the appropriate function is called on the CPACSCreatorAdapter. During the modification of the aircraft we need to be certain that the CPACSAbstractModel does not access the AircraftTree because the tree structure can be modified and a null pointer exception might occur. To manage this case we disconnect the CPACSCreatorAdapter from the QTreeView before any modifications. When the modification has been performed, we reset the connection between CPACSCreatorAdapter and QTreeView. Once the modification is finished, a signal is emitted and a slot that updates the displayed aircraft from the CPACS file is run.

Figure 33 shows the UML class diagram for the principal classes used in the CPACS Creator framework.

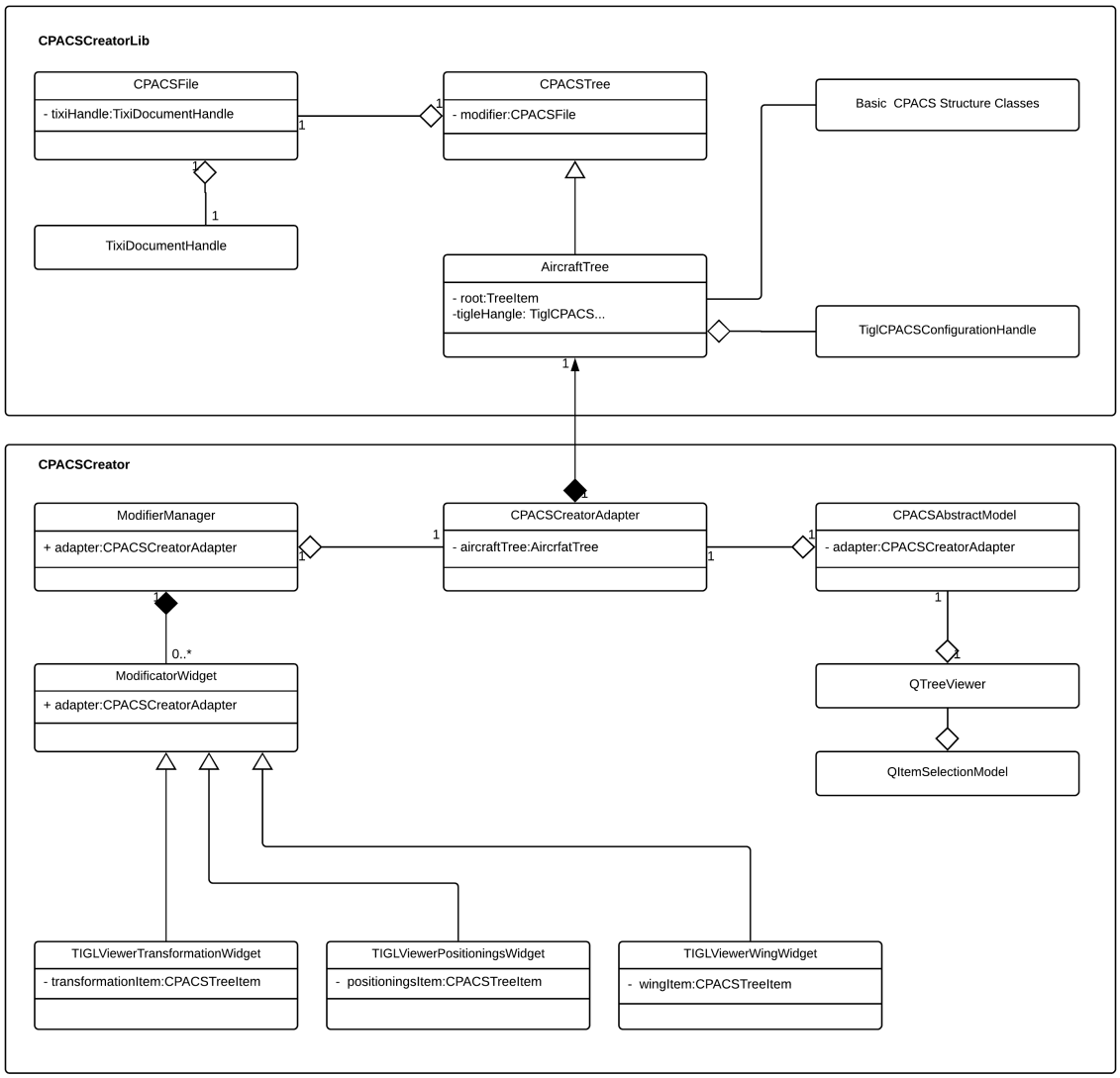


Figure 33: UML Class Diagram of The Creator Framework

12 Further Works

Our Creator framework resolved the main challenges to work with CPACS and it provides basic functions to handle CPACS geometrical data. But there is still a lot of work to be done to transform the CPACS Creator framework into a complete CAD for aircraft. In this section we will present the possible further work for this project.

For the CPACSCreator library, the next step will be to extend it to support other parts of the geometrical CPACS format. The library could be extended to support the following functions:

- Modify fuselages with high level parameters
- Create fuselages
- Create wings
- Modify trailing-edge-devices of a wing
- Add twist parameter for the wing
- Modify ribs and spars of a wing

One key aspect of this extension would be to be able to create an aircraft from scratch. The parameters to give to a function that create a wing or fuselage need to be study. Algorithms to create wings and fuselages geometry based on these parameters also needs to be written. Such algorithms would be a huge step towards a complete aircraft CAD software.

Another useful extension for the CPACSCreatorLib would be to create a interface for Python. Python is a really powerful and popular language that is heavily used by the CFSE team and their partners. For the moment the framework can be easily used by other C++ packages. But the code needs to be compiled each time. The main advantage to have a Python interface is that no compilation is required. So, the CFSE team can create some scripts for the purpose of a particular use case and then slightly change the script for another use case.

For the CPACSCreator library, the next step would be to create the interface to support the new functions above. During the project we had several interface and feature ideas but we unfortunately did not have the time to implement them.

One of the observation was that the parameters are only numbers, and numbers are not really user friendly. As human we need to process them to find out there meaning. This observation leads us to two interface ideas.

The first idea is to allow editing directly in the 3D scene. Such interfaces are popular in 3D modeling software, as the 3Ds Max studio [17]. By clicking on the wing, the user will be able to move and rotate it. We can also think about hot keys to set for example span and dihedral angle. This interface can be really user friendly but it is also a real challenge to implement. The principal challenge is that the 3D scene is completely separated from our CPACSCreator library. To allow such an editing a bridge between the 3D scene and the library must be build. One solution is to incorporate markers in the 3D scene that link the scene to AircraftTree. When the user selects an element in the scene these markers will appear and actions can be performed on it. Figure 34 shows the potential markers for a wing and the effect of dragging them. If a such interface is created, a particular attention must be paid on how the mouse act with the 3D object and move them. It is easy to create an interface where is difficult to move a points on the wanted position.

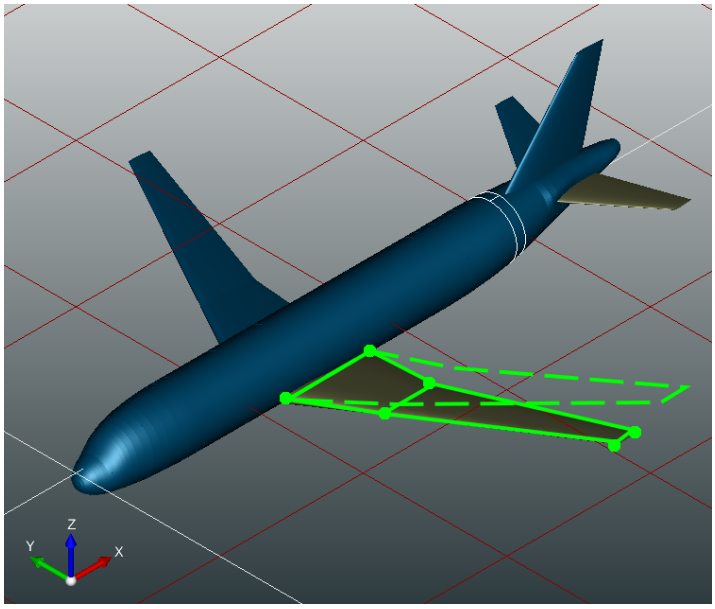


Figure 34: CPACSCreator Marker Interface proposal for the wing

The second interface idea is to represent the *positioning* sweep angles, dihedral angles and length as a graph. Currently the positionings modification widget can quickly becomes overcharged. The figure 35 shows a aircraft with multiple positionings and its modification widget. If we represent these data as a graph we can see some pattern in the data and the data can becomes more readable. The figure 36 show the same data as in the positionings widget in graphs. The graph shape can be analyzed by the user and editing would be possible with a few set control points or its equation. Such a representation of the *positionings* seems promising for aircraft optimization. The main challenge to implement this interface is to find or build the graph library. After a small research on a such library we did not found an available library for Qt5 that is compatible with the Apache2 license.

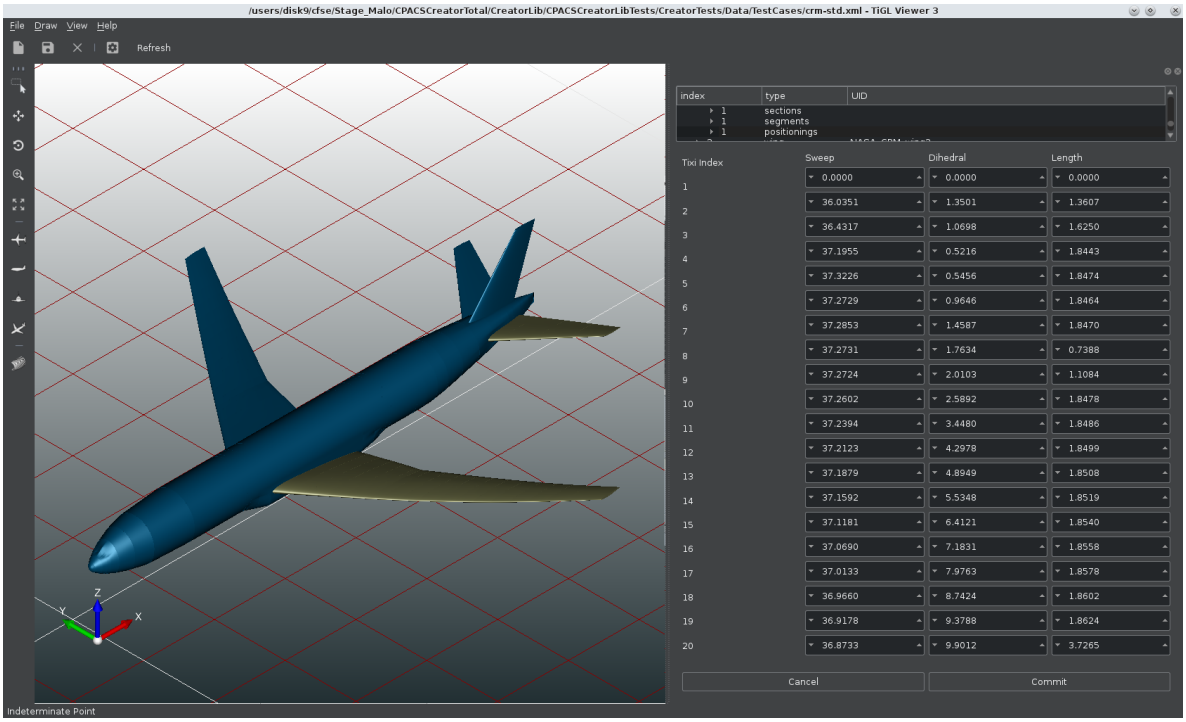
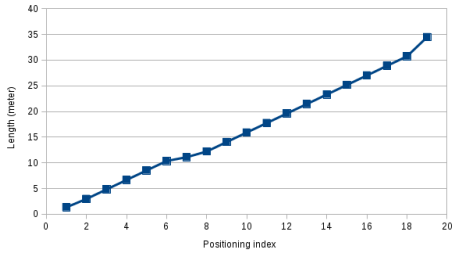
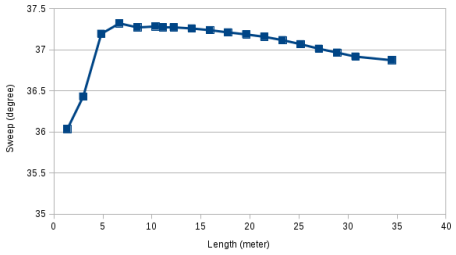


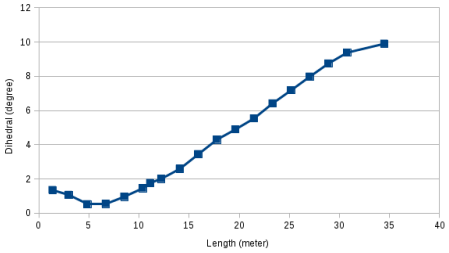
Figure 35: The positionings modification widget for an aircraft with multiple positionings



(a) Length of the positionings



(b) Sweep angles of the positionings



(c) Dihedral angles of the positionings

Figure 36: Positionings data of the aircraft shown in figure 35 as graphs

We estimate the time to implement the major extensions to 6 moth. This estimation does not include the editing in the 3D scene

12.1 Towards to a new definition of CPACS

In this subsection we will present several suggestions for the wing description, that might be taken up in a possible future release of CPACS. These suggestions are coming from the work done in this project, thus they are really only from the point of view of computer graphics and in some smaller measure of the aerodynamic perspective. We are not taking into account other fields involved in aircraft design and we are not aware of their requirements on the CPACS geometry structure.

We suggest to describe a wing with the following elements:

Normalized wing airfoil: The airfoil will describe a normalized slice of the wing. It will be really close to the *wingAirfoil* used in the current CPACS format. But we will add requirements on it such that the origin of the airfoil is always defined at the same place and such that the chord length is always 1.

Wing element: The wing element will reference a profile and apply a affine transform on it. This affine transformation will be the only affine transform that can be applied on the profile. The affine transform will be described in term of an augmented matrix to allow the shearing.

Directed graph: The directed graph will represent the connection between the wing elements. It should start from the root be connected and acyclic. It will play the role of the *wingSegment* of the current CPACS. But information is added about the root and the end element of the wing.

The wing will the be build in a similar way as in the current CPACS. First we define airfoils, then we transform them by wing elements and finally we connect the wing elements by the graph.

The main benefits of this structure is that the dependencies between elements are removed, we do not need to open the profile to have the quadrilateral wing, the affine transformation is in one place only and the structure of the wing is clearly given by the graph. We can observe that the main issues occurred in this project to work with CPACS format will disappear if the wing would be described in this manner. Of course, this hypothetical wing description was develop with the particular needs of the Creator project. Once we would start to real use it, particularly in other context some issues will probably occur.

The main drawback of this description is that the XML is more hermetic to read and edit for a human, in particular the augmented matrices. But if a framework such as CPACSCreator is developed and complete, very few people will edit manually the file.

13 Conclusions

During this project we spend a considerable amount of time to understand the CPACS format and the codes of the existing tools. We solved most of the challenges to handle the complexity of this format.

We developed a framework that permits to edit the geometrical part of the CPACS data that does not require the need to understand the underlying CAPCS description. The wing description was translated into high level parameters commonly used in aircraft design. This has considerably simplified the possibilities to change a wing. The user does not need to know the details of the CPACS description of the wing, he can simply edit these high level parameters using simple function calls.

We demonstrated the usability of our framework to generate various aircraft geometries and to run CFD simulation on the super-computer Beskow.

The framework was integrated in the TIGLViewer which offers a user friendly interface that permits to directly see the modifications by the user.

There are still several steps needed to be able to modify all geometrical data of an aircraft described in the CPACS format. However, we can conclude that the theoretical foundation and the code architecture to modify the geometrical data in CPACS has been successfully developed in this master thesis project.

14 Acknowledgements

The work presented in this thesis has been performed in the framework of the AGILE project (Aircraft 3rd Generation MDO for Innovative Collaboration of Heterogeneous Teams of Experts) and has received funding from the European Union Horizon 2020 Programme (H2020-MG-2014-2015) under grant agreement no 636202. The Swiss participation in the AGILE project was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) undercontract number 15.0162.

The work on the super-computer Beskow in Sweden has been performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the author gratefully acknowledges the support of Airinnova [18] and the computer resources and technical support provided by PDC center of the KTH.

I would personally like to thank Jesper Oppelstrup and Mengmeng Zhang who provided me with theoretical and technical support during my stay in Sweden. The CFSE team for their warm welcome in their office and in particular Jan Vos and Aidan Jungo who supervised my thesis. Along the project they made constructive remarks, gave me previous advices and support to write the report. I would also like to thank Professor Mark Pauly who supervised this thesis on the EPFL side and provided the administrative structure for this master thesis.

A Math

A.1 Combined translations

Imagine the case where a first translation is performed before a linear transformation and a second translation is performed after the linear transformation. We are interested to combine these two translations into a single translation.

Let T_1 and T_2 be augmented translation matrices and let M be an augmented linear transformation matrix. This implies that they are respectively of the form:

$$T_n = \begin{bmatrix} I^{(3\times3)} & t_n^{(3\times1)} \\ 0^{(1\times3)} & 1^{(1\times1)} \end{bmatrix}, M = \begin{bmatrix} A^{(3\times3)} & 0^{(3\times1)} \\ 0^{(1\times3)} & 1^{(1\times1)} \end{bmatrix}$$

Now, lets develop the following multiplication using block matrices arithmetic.

$$\begin{aligned}
T_1 M T_2 &= \begin{bmatrix} I & t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & t_2 \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} I & t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} [AI + 0 \cdot 0] & [At_2 + 0 \cdot 1] \\ [0 \cdot I + 0 \cdot 1] & [0 \cdot t_2 + 1 \cdot 1] \end{bmatrix} \\
&= \begin{bmatrix} I & t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A & At_2 \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} [IA + t_1 \cdot 0] & [IAt_2 + t_1 \cdot 1] \\ [0A + 1 \cdot 0] & [0At_2 + 1 \cdot 1] \end{bmatrix} \\
&= \begin{bmatrix} A & At_2 + t_1 \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} I & At_2 + t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix} = T_{left} M \\
&= \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & t_2 + A^{-1} t_1 \\ 0 & 1 \end{bmatrix} = M T_{right}
\end{aligned}$$

So, we can indeed combine these two translations into a single translation and the explicit formula is given by the two last rows.

B CPACS on Beskow

In this section we will present a framework, called cpacsOnBeskow [19], that we have developed to quickly launch a series of computational fluid dynamics (CFD) simulations on the super-computer Beskow. This framework takes as input a set of CPACS files and a set of CFD stimulation parameters, and produces VTK flow files.

This framework has three main benefits for our project. Firstly, we understand better in which environment the CPACSCreator software will take place. Secondly, we can validate the high level functions that we have implemented. With validation, we mean that a generated CPACS file is coherent and can be used as input for a CFD simulation. Thirdly, we give to our users an easy way to use CPACS with the CFD solver SU2 [20].

To perform a CFD simulation over an aircraft we need a volume mesh of the geometry. The meshing can be performed by SUMO [21]. However, SUMO does not accept CPACS files as input. A software, called CPACS2SUMO [22], developed by Aidan Jungo, converts a CPACS file to a SUMO DMX file. The simulation itself is performed by SU2. The main task of our scripts is to connect these 3 softwares together.

Figure 37 provides an overview of the whole process. We generate a set of CPACS files with the CPACSCreator library (step 1 in the figure). For example, we can vary only one variable of the aircraft geometry as the sweep angle or the dihedral angle. We transform these geometries in DMX files for SUMO using the CPACS2SUMO software. SUMO generates the SU2 meshes. These two steps are contained in script *cpacs2Su2.bash* (Step 2 in the figure). On another side, we generate SU2 configuration files with the *generateSu2Cfg* script (Step 3 in the figure). 'GenerateSu2Cfg.bash' script takes a list of Mach values, a list of angles of attack and generates for every two possible pairs a configuration file for SU2. Finally, the script *runOnBeskow.bash* runs for each meshes all the different CFD simulations parametrized by the different configuration files (step 4 in the figure).

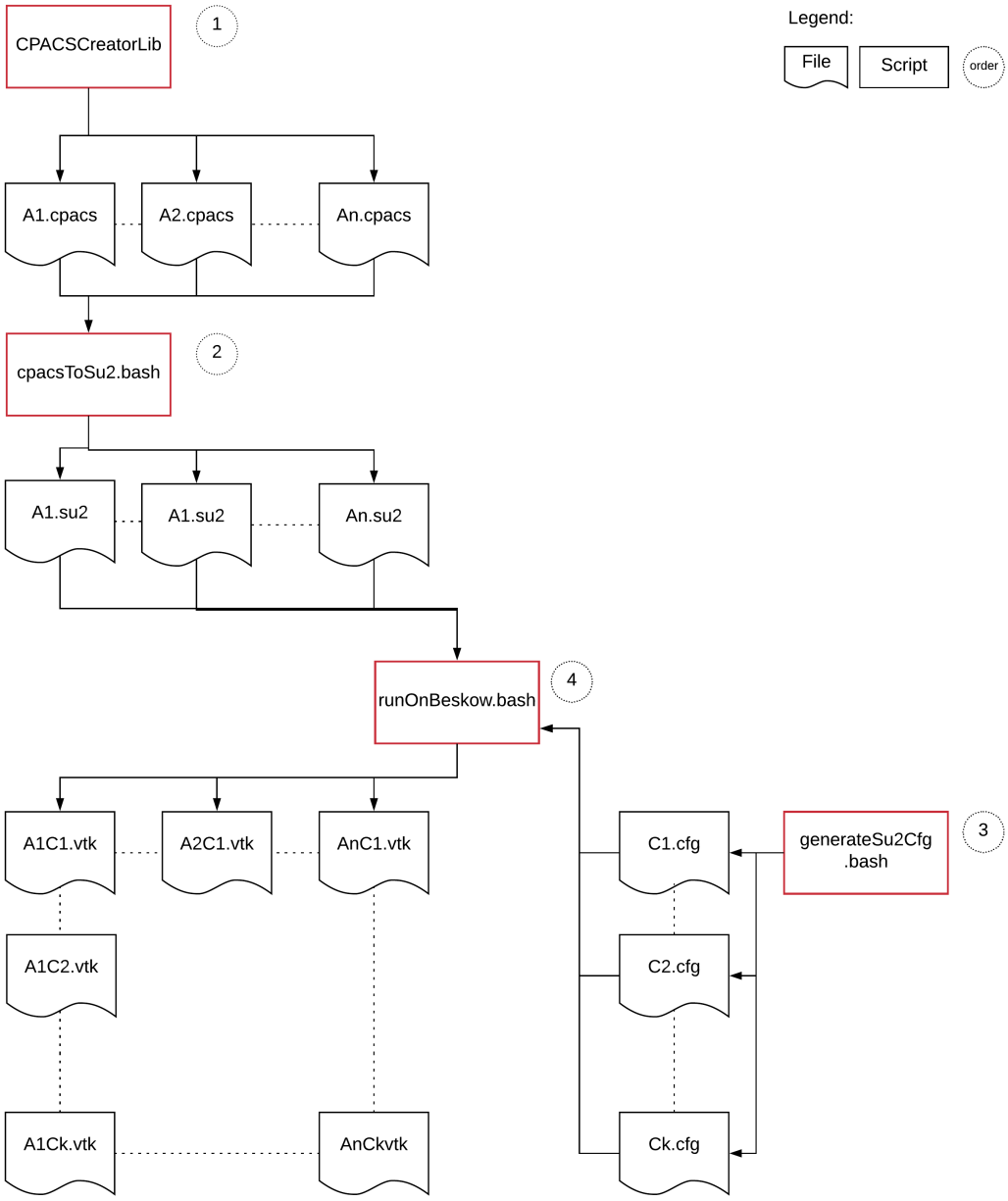


Figure 37: Overview of the cpacsOnBeskow framework

With these 3 scripts, we are able to launch the same series of simulations for slightly different aircrafts. Of course, the computational time can quickly explode. Each parameter multiplies the number of simulations to run. For example, if we have 5 CPACS input files, 3 Mach numbers and 3 angles of attack, we got already 5 * 3 * 3 = 45 simulations to run. If each simulation costs 20 minutes, the total time needed to run these 45 simulations will be 15 hours.

To optimize our framework we wanted to know until when the parallelization makes sense using SU2 on Beskow. We ran two different CFD simulations using a wide range of cores. The first configuration, A, has 1.5 million of cells in the mesh and requires 9000 iterations to converge. The second configuration, B, is smaller. It has only 70'000 cells in the mesh and needs 300 iterations to converge. In a perfect world, each time we double the cores number, the processing time is divided by 2. This rarely the case, because the code needs to prepare the parallelization and the different cores need to communicate

with each other during the simulation. The chart 38 and 39 shows the processing time over the core number. We have concluded that for small simulation (configuration B), it's reasonable to run the calculation up to 64 cores and for larger simulations (configuration A) it's reasonable to use up to 512 cores. We added a setting in *runOnBeskow.bash* to set the number of cores that SU2 will use.

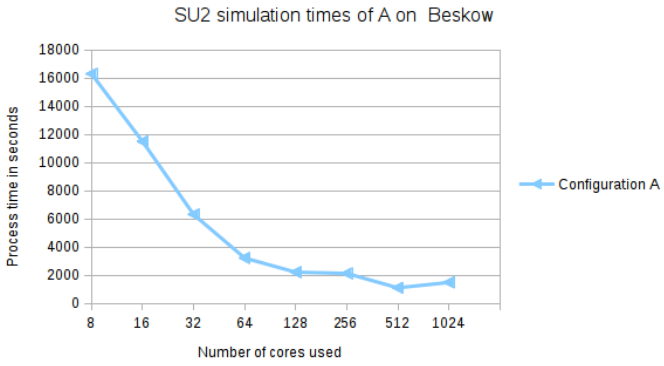


Figure 38

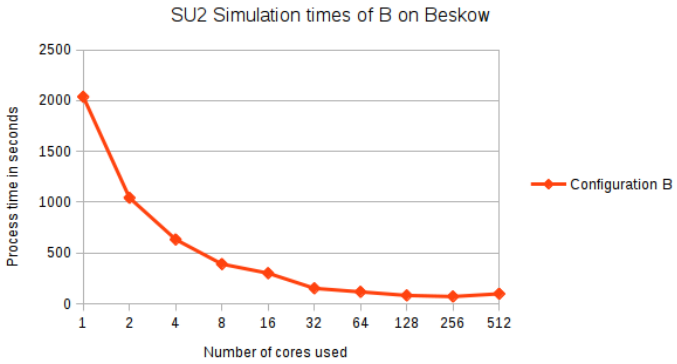


Figure 39

This framework is a prototype and some optimization and improvements can be made. Particularly, we have seen that the parallelization make sense only until a quite small amount of the available cores resources of Beskow. But each simulation is independent, so if we want to have our results more quickly, we could ask for a large amount of cores and run different simulations in parallel. For example, we could ask for 2048 cores and run 32 simulation of 64 cores in parallel. So our previous example of 15 hours will take only 2 hours.

References

[1] Malo Drougard. Cpacscreator github. https://github.com/cfsengineering/CPACSCreator.

[2] DLR. CPACS GitHub. https://github.com/DLR-LY/CPACS.

[3] DLR. TIGL GitHub. https://github.com/DLR-SC/tigl.

[4] John J. Bertin Ray Whitford Steven A. Brandt, Randall J. Stiles. *Introduction to Aeronautics: A Design Perspective*. AIAA Education Series, 2015.

[5] D. P. Raymer. *Aircraft Design: A Conceptual Approach*. AIAA Education Series, 1992.

[6] German Aerospace Center (DLR) Pier Davide Ciampa, German Aerospace Center (DLR); Björn Nagel. the agile paradigm: the next generation of collaborative mdo. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference Denver, Colorado*, 2017.

[7] Björn Nagel Daniel Böhnke P Saquet Arthur Rizzi, Mengmeng Zhang. Towards a unified framework using cpacs for geometry management in aircraft design. 01 2012.

[8] DLR. TIXI GitHub. https://github.com/DLR-SC/tixi.

[9] The GNOME Project. libXML2. https:https://github.com/aidan-cfse/CPACS2SUMO.

[10] OPENCASCADE. OPENCASCADE Web Site. https://www.opencascade.com/. Accessed : 2018-03-07.

[11] Guillermo Costa Nachiket Deshpande Mark Moore Edric San Miguel Alison Snyder William Fredericks, Kevin Antcliff. Conceptual design using vehicle sketch pad. 01 2010.

[12] KTH. Beskow reference page. https://www.pdc.kth.se/hpc-services/computing-systems/beskow-1.737436.

[13] Varignon. Varignon theorem. https://en.wikipedia.org/wiki/Varignon%27s_theorem.

[14] Eigen. Eigen library web page. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[15] Sergei Khashin. Quartic equation solver. http://math.ivanovo.ac.ru/dalgebra/Khashin/poly/index.html.

[16] Google. Google test framework. https://github.com/google/googletest.

[17] Autodesk. 3ds max studio web page. https://www.autodesk.eu/products/3ds-max/overview.

[18] Airinnova. Airinnova web site. http://airinnova.se/.

[19] Malo Drougard. cpacsOnBeskow. https://github.com/MaloDrougard/cpacsOnBeskow.

[20] SU2. SU2. https://su2code.github.io/.

[21] Larosterna. SUMO. https://www.larosterna.com/products/open-source.

[22] Aidan Jungo. CPACS2SUMO. http://www.xmlsoft.org/.