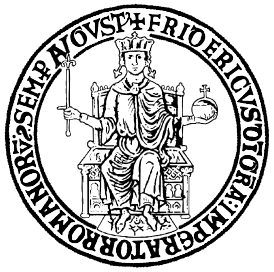

Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE
Corso di Laurea in Ingegneria Aerospaziale

Tesi di Laurea Magistrale
in
Ingegneria Aerospaziale

Aircraft Solid Modeling with JPAD and Automatic Workflows for High-Fidelity Numerical Aerodynamic Simulations

Candidato:
Mario Di Stasio
Matricola M53/241

Relatore:
Prof. Agostino De Marco

Anno Accademico 2017/2018

Ai miei genitori

Abstract

The main goal of this thesis work is to give an overview on the development of a Java-based library, to be integrated in JPAD (a software dedicated to preliminary aircraft design), whose purpose is dealing with the automatic production of complete or partial aircraft CAD models. So generated 3D models can be imported and visualized in a JavaFX window, to be implemented in the JPAD GUI, or imported into a CFD tool, in order to perform high-fidelity numerical analyses.

A brief introduction on the JPAD software architecture is followed by a detailed description of the JPADCAD package, which contains all the classes and the utility functions designed to generate CAD models. Afterwards, the methodologies and the functions through which the aircraft components (fuselage, wing, horizontal and vertical tail, canard, fairings, control surfaces) are translated from XML data file to 3D models are described. The last chapter deals with the classes and utilities designed and used in order to import JPAD generated CAD files into CFD analysis software CD-adapco STARCCM+, in order to perform workflows involving high-fidelity numerical solutions.

Sommario

Lo scopo del presente lavoro di tesi è di fornire una panoramica circa lo sviluppo, in linguaggio Java, di una libreria, da integrare al software di preliminary design JPAD, per la generazione automatica di modelli CAD di velivoli completi o parti di essi. I modelli così generati possono essere visualizzati in una finestra JavaFX da integrare nella GUI di JPAD, oppure importati in un tool di CFD, per analisi numeriche ad alto livello di accuratezza.

Ad una breve introduzione circa JPAD e la sua architettura, segue una descrizione accurata del pacchetto JPADCAD, in cui attualmente risiedono le classi e le utility atte alla generazione di modelli CAD. Successivamente vengono descritte le metodologie e le funzioni mediante cui le componenti dell'aereo (fusoliera, ala, piano orizzontale e verticale di coda, canard, fairing, superfici di controllo), a partire da file di input di tipo XML, vengono tradotte in modelli 3D. Conclude il lavoro un excursus sulle classi e le utility create per l'importazione dei file CAD generati in JPAD nel software di analisi CFD CD-adapco STARCCM+, al fine di creare dei cicli automatizzati di lavoro.

Contents

Chapter 1

JPAD: AN OVERVIEW

The conceptual and preliminary design phases play a very important role for the development of the future transport aircraft. A computational framework capable of finding an optimal configuration satisfying several basic requirements would be an essential tool for industrial aircraft designers. Such software should be developed around all those basic principles and approaches to aircraft preliminary design well described in several textbooks on the subject. [19][21][23]

A modern preliminary aircraft design tool should be characterized by a certain level of accuracy and reliability, the capability to perform multidisciplinary analyses, and reasonably short computational times. Because of the particular relevance of production costs, noise, emissions, maintenance, and operative costs in the commercial success of a transport aircraft, a modern software framework should be developed with a Multi-disciplinary Design Optimization (MDO) approach in mind. Another important aspect is the user-friendliness of the interface that should allow the user to interact with the design framework in an easy, fast and efficient way. Of the same importance is the possibility to include in the software multiple fidelity analysis methods or to modify and develop new semi-empirical models to achieve better accuracy. It shoul also be possible to export the aircraft configuration geometry (e.g., as a Computer-Aided Design (CAD) model) in one or more standard formats and to execute high-fidelity analyses with external tools (e.g., Computational Fluid Dynamics (CFD) or Finite Element Method (FEM) solvers).

The present chapter gives an overview of Java Program toolchain for Aircraft Design (JPAD), a Java-based desktop application for aircraft designers. The aim of JPAD, which eventually will be released as open-source software, is to provide a library and a set of companion tools based on modern software technology as a support for typical preliminary design studies. The software has been conceived to be used in an industrial environment across conceptual and preliminary design phases. In these phases, a lot of different configurations have to be considered, and so the proposed software relies mostly on semi-empirical analysis methods and is capable to quickly provide results.

1.1 Java choice

The main challenge in developing from scratch a computational ecosystem to be used by aircraft designers is the choice of the programming language and its related software technology. On the domain modeling side, another challenge is the abstraction of the aircraft, with all its subsystems and subcomponents, as well as the interface with a wide range of analysis tools (related to a number of different disciplines and with different levels of fidelity). So, the programming language has to be chosen according to several considerations. Regarding JPAD, Java has been selected for the following reasons:

- it is widely supported,
- it promotes the use of open-source libraries (especially for I/O tasks and for complex mathematical operations),
- there are several Integrated Development Environment (IDE) providing a selection of widely supported Graphical User Interface (GUI) frameworks and GUI visual builders,
- it supports the object-oriented design pattern and promotes modularity,
- it naturally allows the development of portable software.

Java is currently backed by Oracle and by a huge community of developers, and so it is continuously updated. Also, advanced and freely available IDEs (such as Eclipse, Netbeans and IntelliJ) allow programmers to streamline and simplify the development process. According to the TIOBE index, in terms of popularity, during the last years the Java language has proven to be the most use among the developers community, as shown in figure 1.1.

With Java being a pure object-oriented programming language, it greatly encourages and simplifies modularization. Each module (package) can be programmed quite independently so that it is relatively easy to divide the work among several programmers. This is essential because the amount of classes and calculation needed to abstract, manage, and analyze the entire aircraft is very large. For such a reason, the establishment of common practices and the adherence to fundamental principle of software development (namely, *do not repeat yourself*, *separation of concerns*, *agile software development*) are equally important.

The abstraction of the aircraft geometry, its subcomponents (fuselage, lifting surfaces, nacelles, undercarriage, etc.), and their inherent features (weight breakdown, aerodynamic loads, engine deck, etc.) should be effective and general at the same time. This is a typical problem of domain modeling, which has been solved with the object-oriented design pattern naturally provided by the Java language.

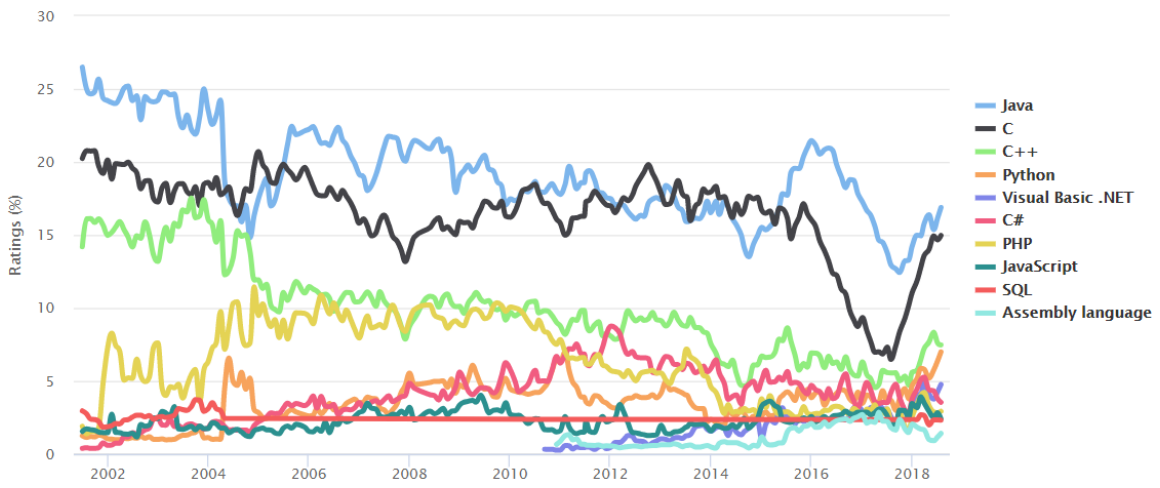

Figure 1.1 TIOBE Programming Community Index (www.tiobe.com)

1.2 Software architecture

The following paragraphs deal with the organization of the code. In particular, the first paragraph describes how aircraft data is stored in XML files, while the second one focuses on the packages which constitutes JPAD. There are currently six main packages and in the following chapters attention will be particularly focused on two of them: `JPADCAD` and `JPADCADSandbox`. The last section, instead, gives a brief overview on how and which kind of analyses are performed in JPAD.

1.2.1 Input files

As previously mentioned, the input file type for JPAD is the XML. In this way, aircraft data, operating conditions, and analyses requirements can be read by the software. XML stands for *eXtensible Markup Language*. It is defined *Markup Language* due to the use of tags that describe the content but, unlike other languages of the same type, the tags created in XML are self explanatory and the user is free to define his own tags, hence the *eXtensible* definition. For this reason documents encoded using the XML format are both human-readable and machine-readable. XML support is provided by many programming languages (including Java) to create and process XML data. Simplicity, portability, platform independence, and usability are some of the key features that have resulted in the increasing popularity of the use of XML based standards. [3]

More specifically, the XML file format can be summarized by the following concepts:

- tag,
- attribute,
- tree structure.

Each of the above mentioned points is reported in figure 1.2: tags, which are contained between two brackets, are completely personalizable, can be completed by some attributes

(such as units of measure) and can be innested, in order to obtain a convenient tree structure suitable for encoding typical data structures.

```
<out_tag>
   <inn_tag1 attribute="value1"> content1 </inn_tag1>
   <inn_tag2 attribute="value2"> content2 </inn_tag2>
</out_tag
```

Figure 1.2 A simple XML example

In order to represent through the use of XML files the typical aircraft data structure, the input file setup designed by the Stanford University for their SUAVE project [22] has been taken into account, deriving from it the folder tree represented in figure 1.3. As the figure shows, there's not just one input file storing all the data related to the aircraft. Instead, there's a XML file for each aircraft component and, as shown below, each component can call other XML files (e.g., lifting surfaces XMLs call for airfoil XMLs). The setup of a hierarchy among the XML files helps to reduce dramatically the dimension of each file and makes working with configuration files much easier, especially at high-level. In fact, the user can work fast and simply, assembling custom aircrafts through combination of chosen components XML files. An example of this capability is shown in listing 1.1, 1.2 and 1.3. Figure 1.3 also shows that a similar approach has been followed on the analysis side, splitting analysis XML in multiple files, each one dealing with a specific type of task (see also listing 1.4).

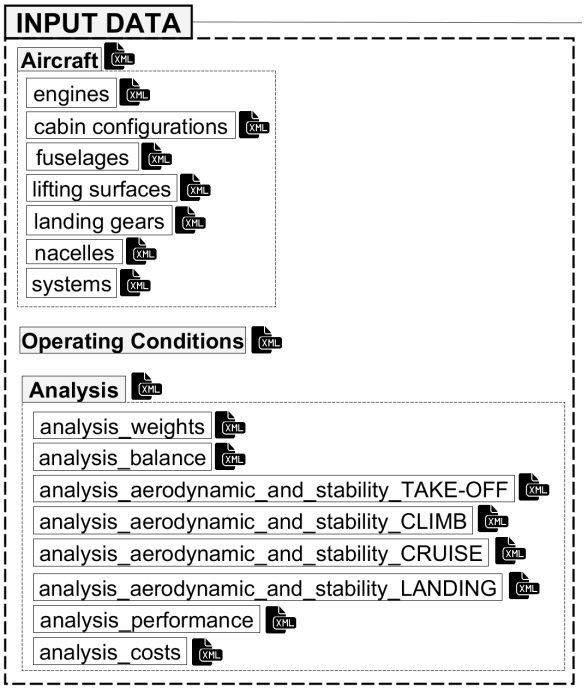

Figure 1.3 JPAD input folder tree

```
1  <jpad_config>
2     <aircraft id="ATR-72" regulations="FAR_25" type="TURBOPROP">
3        <lifting_surfaces>
4           <wing file="wing_ATR72.xml">
```

```
            <position>
               <x unit="m">10.8</x>
               <y unit="m">0.0000</y>
               <z unit="m">1.6</z>
            </position>
            <rigging_angle unit="deg">3.0000</rigging_angle>
         </wing>
         <vertical_tail file="vtail_ATR72.xml">
            <position>
               <x unit="m">21.6</x>
               <y unit="m">0.0000</y>
               <z unit="m">1.3</z>
            </position>
            <rigging_angle unit="deg">0.0000</rigging_angle>
         </vertical_tail>
         <horizontal_tail file="htail_ATR72.xml">
            <position>
               <x unit="m">25.3</x>
               <y unit="m">0.0000</y>
               <z unit="m">5.7374</z>
            </position>
            <rigging_angle unit="deg">1.0000</rigging_angle>
         </horizontal_tail>
      </lifting_surfaces>
      <fuselages>
         <fuselage file="fuselage_ATR72.xml">
            <position>
               <x unit="m">0.0000</x>
               <y unit="m">0.0000</y>
               <z unit="m">0.0000</z>
            </position>
         </fuselage>
      </fuselages>
   </aircraft>
</jpad_config>
```

Listing 1.1 Example of JPAD aircraft XML file

```
<jpad_config>
   <horizontal_tail id="Horizontal tail" mirrored="true">
      <panels>
         <panel id="ATR72 horizontal tail">
            <span unit="m">3.6548</span>
            <dihedral unit="deg">0.0000</dihedral>
            <sweep_leading_edge unit="deg">13.4400</sweep_leading_edge>
            <inner_section>
               <chord unit="m">2.0440</chord>
               <airfoil file="naca0012.xml"/>
               <geometric_twist>0.000000</geometric_twist>
            </inner_section>
            <outer_section>
               <chord unit="m">1.1652</chord>
               <airfoil file="naca0012.xml"/>
```

```
16                <geometric_twist unit="deg">0.0000</geometric_twist>
17             </outer_section>
18          </panel>
19       </panels>
20       <symmetric_flaps>
21          <symmetric_flap id="Elevator" type="PLAIN">
22             <inner_station_spanwise_position>0.1</inner_station_spanwise_position>
23             <outer_station_spanwise_position>0.9</outer_station_spanwise_position>
24             <inner_chord_ratio>0.3000</inner_chord_ratio>
25             <outer_chord_ratio>0.3000</outer_chord_ratio>
26             <min_deflection unit="deg">-25.0</min_deflection>
27             <max_deflection unit="deg">5.0</max_deflection>
28          </symmetric_flap>
29       </symmetric_flaps>
30    </horizontal_tail>
31 </jpad_config>
```

Listing 1.2 Example of JPAD lifting surface XML file

```
1  <jpad_config>
2      <airfoil family="NACA_4_Digit" name="NACA0012" type="CONVENTIONAL">
3          <geometry>
4              <thickness_to_chord_ratio_max>0.1200</thickness_to_chord_ratio_max>
5              <radius_leading_edge_norm unit="m">0.0158</radius_leading_edge_norm>
6              <x_coordinates>[...]</x_coordinates>
7              <z_coordinates>[...]</z_coordinates>
8          </geometry>
9          <aerodynamics>
10             <alpha_zero_lift unit="deg">0.0000</alpha_zero_lift>
11             <alpha_end_linear_trait unit="deg">11.0000</alpha_end_linear_trait>
12             <alpha_stall unit="deg">20.1000</alpha_stall>
13             <Cl_alpha_linear_trait unit="1/rad">6.9200</Cl_alpha_linear_trait>
14             <Cl_at_alpha_zero>0.000000</Cl_at_alpha_zero>
15             <Cl_end_linear_trait>1.2300</Cl_end_linear_trait>
16             <Cl_max>1.8600</Cl_max>
17             <Cd_min>0.005500</Cd_min>
18             <Cl_at_Cdmin>0.000000</Cl_at_Cdmin>
19             <laminar_bucket_semi_extension>0.000000</laminar_bucket_semi_extension>
20             <laminar_bucket_depth>0.000000</laminar_bucket_depth>
21             <K_factor_drag_polar>0.003500</K_factor_drag_polar>
22             <Cm_alpha_quarter_chord unit="1/deg">0.0000</Cm_alpha_quarter_chord>
23             <Cm_ac>0.000000</Cm_ac>
24             <Cm_ac_at_stall>-0.090000</Cm_ac_at_stall>
25             <x_ac_normalized>0.2500</x_ac_normalized>
26             <mach_critical>0.7340</mach_critical>
27             <x_transition_upper>0.1200</x_transition_upper>
28             <x_transition_lower>0.1200</x_transition_lower>
29         </aerodynamics>
30     </airfoil>
31 </jpad_config>
```

Listing 1.3 Example of JPAD airfoil XML file

1.2.2 Libraries

JPAD is currently made of six main packages, each one of them dealing with specific tasks. Moreover, each package can contain several sub-packages (organized in classes and utilities) pertinent to more specialized assignments. This kind of organization, called modularity, helps in terms of code mainteinance: part of the system can be updated in case of issues without a need to make large-scale changes. Besides, modularity makes working with the code much easier in an ever changing team.

The six packages that constitutes JPAD are: `JPADCore` (actually, `JPADCore_v2` due to the intense changes occurred during the last months), `JPADConfigs`, `JPADCommander`, `JPADCAD`, `JPADSandBox` (actually, `JPADSandBox_v2`), and `JPADCADSandBox`. The next sections describe each package individually, summarizing the characteristics and the tasks of each one of them.

`JPADCore`

`JPADCore`, as the name suggests, is JPAD main package. It currently contains the following sub-packages.

- `aircraft`, which contains sub-packages dedicated to aircraft components (fuselage, lifting surfaces, nacelles, power plant, cabin configuration) and classes managing landing gears and systems mounted on board. As previously mentioned for the input XML structure side, the lifting surfaces package also contains the classes managing the airfoils.
- `analyses`, which contains the managers for five types of analysis that can be performed on aircrafts: weights, balance, aerodynamics and stability, performance, and costs. Additionally, `analyses` also contains sub-packages (one for each of the before mentioned aircraft components) enclosing component's manager classes for weigths, balance and aerodynamic calculation. Finally, classes managing the operating conditions are stored in this package too.
- `calculators`, actually containing all the methods and the formulas needed to perform all the previously mentioned analyses.
- `database`, containing the classes used in order to read data from HDF files (reader can refer to [4] for more information about). In particular, these classes manage the data reading process from aerodynamics and engine databases.
- `standaloneutils`, which encloses several utility classes, i.e., classes containing methods that can be accessed in a static way. These utilities cover the most various needings, from Standard Atmoshere properties calculation to XML file reading.
- `writers`, finally, enclosing classes useful in order to store results.

JPADConfigs

`JPADConfigs` is the package containing all the enumeration classes used in JPAD. In computer programming, enumerations consist of a set of named values. The enumerator names are usually identifiers that behave as constants in the language. In the Java programming language, enumeration types are defined using the `enum` keyword; because values assumed by enumeration type variables are constant, the names of an `enum` type fields are in uppercase letters. [11]

Like the `standaloneutils` classes, enumerations contained in `JPADConfigs` cover very different fields: some such as `ComponentEnum` and `AirplaneType` regard the aircraft and its characteristics, while others, such as `MethodEnum` and `AnalysisTypeEnum`, deal with the analyses.

JPADCommander

`JPADCommander` contains the classes which allow the JPAD GUI (simply called JPADCommander in the rest of the paragraph) to work.

JPAD comes with a simple yet very intuitive GUI. In order to simplify the management of all the operations required for the definition of the aircraft parametric model and its analysis, JPADCommander guides the user through a path which starts with the definition of the working folders and ends with the management and visualization of the output. In particular, JPADCommander interfaces with the user by means of four main windows. The first one is a configuration window, through which the user defines:

- working directory,
- input directory,
- output directory,
- database directory.

The second screen gives the user the possibiity to choose between *Input*, *Analysis* and *Results*. As soon as the user has defined the previously mentioned folders, the only option available is the one regarding the input, with the analysis button becoming available just after the input phase has been performed. The same happens for the results button, which becomes clickable after the user has performed the analysis phase. The third screen, the one regarding the Input Manager, is shown in figure 1.4. Here, the user has three possibilities:

- loading a default aircraft,
- importing aircraft data from file,
- generating a new aircraft filling in all the fields in the Input Manager screen.

Figure 1.4 also shows that the Input Manager screen has several additional pages, dedicated to each of the aircraft components mentioned in the input file section. After completing the input phase, clicking the *Home* button the user will have access to the Analysis Manager screen, which is shown in figure 1.5. In this module, the user can define which analyses he wants

to perform. As for the Input Manager, also in this case the user has the possibility to load a previously saved analysis or fill in the available fields. Finally, as soon as the analyses have been performed, results become available from the main screen.

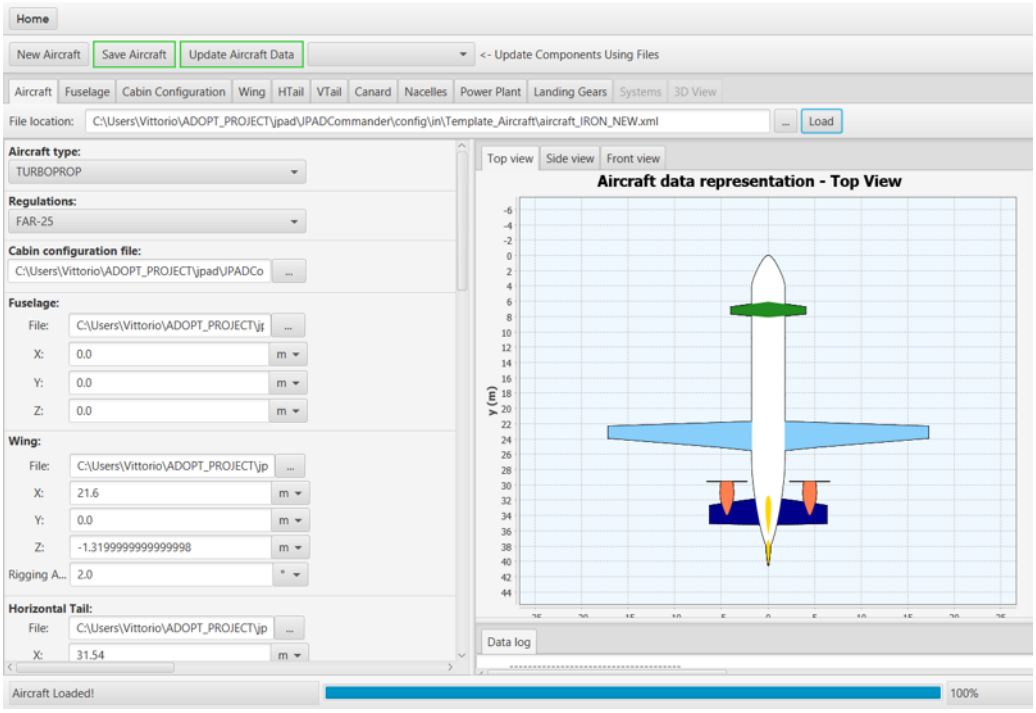

Figure 1.4 JPADCommander Aircraft Input Manager screen

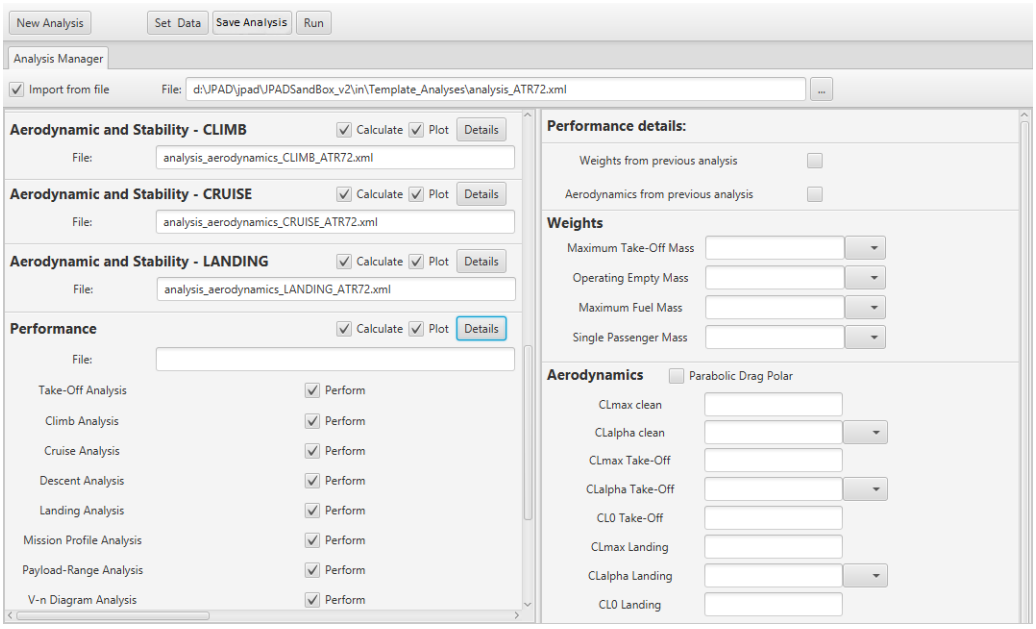

Figure 1.5 JPADCommander Analysis Manager screen

JPADSandBox

This package contains all the test classes, i.e., classes (mostly containing main methods) used in order to test new methods or functionalities, before adding them to JPAD main libraries. It encloses several packages, one for each developer, in which individual tests can be written and

performed. Besides, it also contains the folders in which template aircraft and analysis XMLs are stored.

JPADCAD - JPADCADSandBox

Generating CAD models can be an important achievement for a software such as JPAD for several reasons.

- It gives an immediate feedback about the data provided to the application: in case of mistakes, the CAD model helps detecting them almost immediately.
- It allows the user to run a CFD analysis using an external tool. In fact, JPAD generated CAD models are ready to be meshed without any further adjustment.
- It provides an accurate estimate of the wet surface of each component.

`JPADCAD` and `JPADCADSandbox` are the packages which currently contain all the classes and utility classes necessary to the creation of geometric and topological entities. Shapes (i.e., basic topological entities) can be exported into various file formats (e.g., STEP or IGES) and opened in any desired CAD software. Generating the most various shapes is made possible by the use of the Open CASCADE Technology (OCCT) classes, which are actually written in C++, but can be used in JPAD thanks to the OCCT Java Wrapper, a tool which allows the use of OCCT classes from within Java applications. The static methods which actually perform the translation from the aircraft geometric data (stored, as previously mentioned, in the XML file format) to CAD file format currently reside in `AircraftUtils`, a `JPADCADSandbox` utility class. The same package also contains all the tests performed for this thesis work.

1.2.3 Analyses

The main purpose of JPAD is to perform different analyses and to provide the data necessary for the comparison of different aircraft or different configurations of the same aircraft. The previously described XML structure easily allows this. Besides, the same hierarchical structure has been replicated for the analyses, as reported in listing 1.4, giving the user the possibility to perform a complete analysis (i.e., executing all the analyses contained in `JPADCore`), or specific ones, combining different analysis files. This allows an easier evaluation of a generic cost function during optimization tasks, resulting in a reduced amount of computational costs required for this kind of operations.

```
1  <jpad_config>
2      <!-- Input data, shared across analysis tasks -->
3      <global_data>
4          <positive_limit_load_factor>2.5</positive_limit_load_factor>
5              <negative_limit_load_factor>-1.0</negative_limit_load_factor>
6      </global_data>
7      <!-- Required analysis tasks - comment the tag of the analysis if not needed -->
8      <analyses id="ATR-72 analysis" iterative_loop="FALSE">
9              <weights
10             file="analysis_weights_ATR72.xml"
```

```
11                plot="TRUE"
12                create_CSV="TRUE"
13          method_fuselage=""
14          method_wing=""
15          method_htail=""
16          method_vtail=""
17                method_canard=""
18                method_nacelles=""
19                method_power_plant=""
20                method_landing_gears=""
21                method_APU=""
22                method_air_conditioning_and_anti_icing=""
23                method_instruments_and_navigation_system=""
24                method_hydraulic_and_pneumatic_systems=""
25                method_electrical_systems=""
26                method_furnishings_and_equipments=""
27                method_control_surfaces=""
28          />
29          <balance
30                file="analysis_balance_ATR72.xml"
31                plot="TRUE"
32                create_CSV="TRUE"
33          method_fuselage=""
34          method_wing=""
35          method_htail=""
36          method_vtail=""
37                method_canard=""
38                method_nacelles=""
39                method_power_plant=""
40                method_landing_gears=""
41          />
42          <!--
43          <aerodynamic_and_stability
44                plot="FALSE"
45                create_CSV="FALSE"
46                take_off_condition="TRUE"
47                file_take_off_condition="analysis_aerodynamics_ATR72_takeoff.xml"
48                climb_condition="TRUE"
49                file_climb_condition="analysis_aerodynamics_ATR72_climb.xml"
50                cruise_condition="TRUE"
51                file_cruise_condition="analysis_aerodynamics_ATR72_cruise.xml"
52                landing_condition="TRUE"
53                file_landing_condition="analysis_aerodynamics_ATR72_landing.xml"
54          />
55          <performance
56                file="analysis_performance_ATR72.xml"
57                plot="FALSE"
58                create_CSV="FALSE"
59                take_off="FALSE"
60                climb="FALSE"
61                cruise="FALSE"
62                descent="FALSE"
63                landing="FALSE"
```

```
64                  payload_range="FALSE"
65                  V_n_diagram="FALSE"
66                  mission_profile="TRUE"
67            />
68            <costs
69                  file="analysis_costs_ATR72.xml"
70                  plot="FALSE"
71                  create_CSV="FALSE"
72                  doc_capital="TRUE"
73                  doc_capital_method="AEA"
74                  doc_crew="TRUE"
75                  doc_crew_method="AEA"
76                  doc_fuel="TRUE"
77                  doc_fuel_method="AEA"
78                  doc_charges="TRUE"
79                  doc_charges_method="ILR_AACHEN"
80                  doc_maintenance="TRUE"
81                  doc_maintenance_method="ATA"
82            />
83            -->
84      </analyses>
85  </jpad_config>
```

Listing 1.4 JPAD analyses XML file, not needed analyses are commented

JPAD currently allows five types of analysis.

- Weights analysis: estimates the aircraft weight breakdown starting from a first guess maximum take-off weight and specific mission requirements. In particular, it evaluates each aircraft component mass using well-known semi-empirical methods.
- Balance analysis: estimates the centre of gravity position related to each weight condition and draws the balance diagram.
- Aerodynamics and Stability analysis: the aerodynamics module estimates all the aerodynamic characteristics in terms of lift, drag and moment coefficients, at different operating conditons and for each aircraft component (wing, tails, fuselage, and nacelles); whereas the stability module reports data concerning the aircraft static stability.
- Performance analysis: evaluates aircraft most important performances, such as payload-range diagram, mission profile, cruise flight envelope, ground performance, climb performance, and cruise grid chart.
- Costs analysis: estimates the Direct Operative Cost (DOC) breakdown.

In addition to multidisciplinary analyses, JPAD is also able to perform parametric studies and MDO. In both cases it is possible to define a set of parameters to be varied, typically geometric, and the input manager will provide for the generation of all the test aircrafts thanks to the combination of different files for the individual components. Once all the required configurations are ready, it will be possible to choose whether to carry out a parametric study or an optimization. It will be necessary to define one or more objectives to be monitored

and one or more constraints to be respected; the latter can be selected from a dedicated list. Once all the parameters have been defined, JPAD will launch all the analyses necessary for the evaluation of the objectives and the required constraints, with the possibility of exploiting serial or parallel analyses (i.e., executed on more threads) in order to reduce calculation times. Once all the required calculation have been completed, one or more response surfaces will be obtained which, on the one hand, constitute the output of the parametric study, while on the other they can be used as input for optimization (figure 1.6).

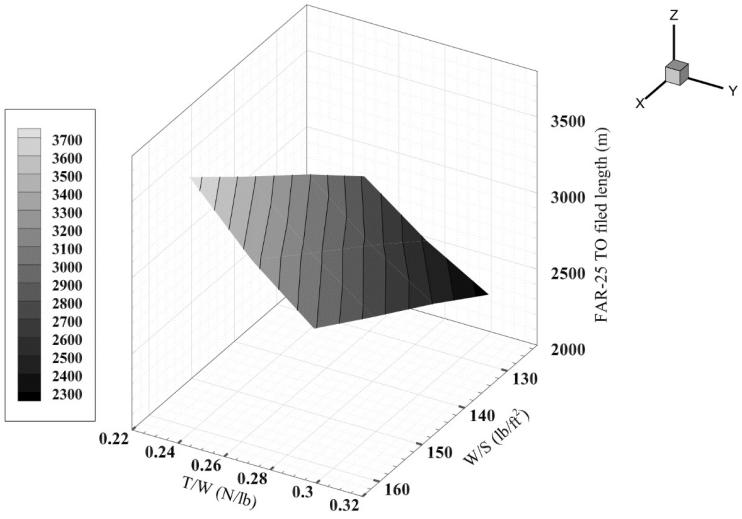

Figure 1.6 Example of parametric study performed in JPAD on a Boeing B747-100B

Figure 1.7 shows JPAD typical flowchart, summarizing the concepts expressed in the previous paragraphs. In particular, it clearly makes a distinction between the input (aircraft and analysis data) and the core (the analysis itself).

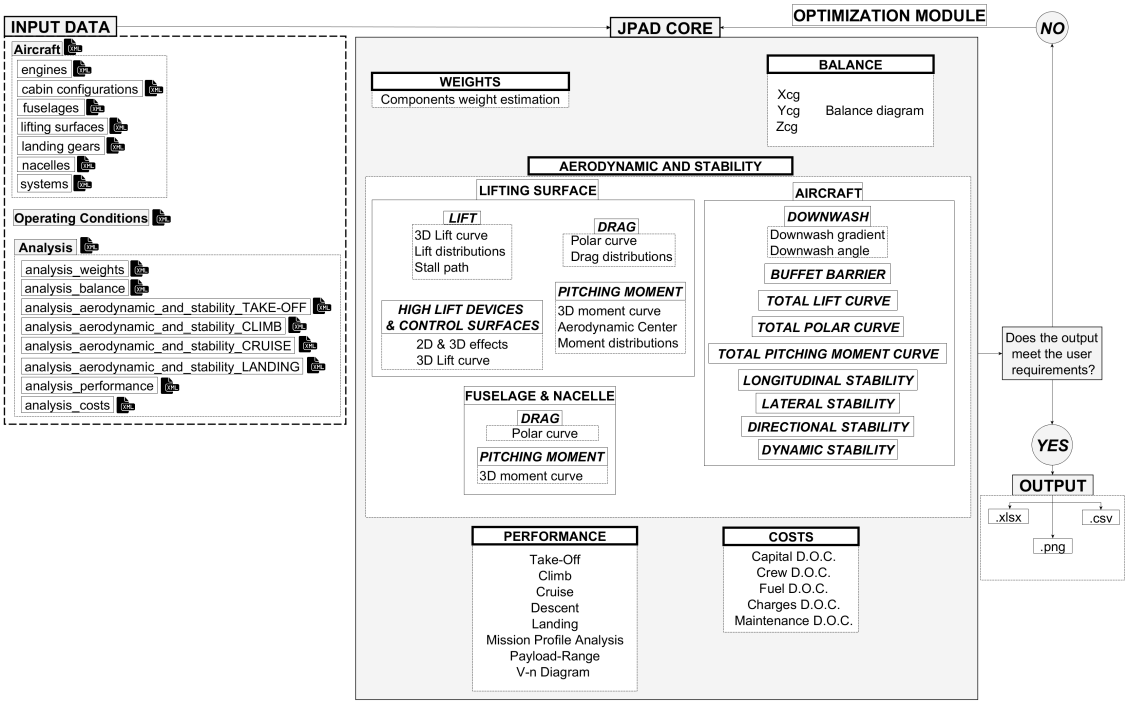

Figure 1.7 JPAD flowchart

Chapter 2

JPADCAD MODULE

2.1 CAD: motivations and approaches

In paragraph 1.2, it was mentioned why the generation of a 3D CAD model in a software such as JPAD could prove particularly useful. In this section the attention will be focused even more on the motivations that may lead to the need to generate a complete 3D model of the aircraft even in the initial design stages.

CFD has increasingly become, over the last three decades, an indispensable tool for aerodynamic studies in aircraft design. Nowadays, viscous CFD simulations are carried out in ever increasing numbers and at all stages of the design. CFD is particularly useful in terminal project stages, when simulations are required in the most disparate phases of flight (such as take-off or landing). With the possibility of producing unstructured mesh, the spectrum of CFD applications has expanded further, and, at the moment, there seem to be no limits to the geometric complexity that can be fed to CFD solvers. This also happens thanks to the continouos reduction of the time needed to create the mesh and perform the calculations. Engineers can now perform complex simulations using a simple desktop computer, when, on the other hand, the same simulations would have required a cluster for parallel computing only a few years ago. This growing computing power has given engineers the ability to carry out most of the design and analysis work of complex systems without the need to resort to physical models or prototypes.

Nowadays there is an ever increasing demand in the aerospace industry to use CFD analysis even in the preliminary and conceptual phases of aircraft design. This is due to the fact that CFD simulations can predict the compressible and viscous flow fields around the aircraft taking into account all the important aerodynamic phenomena. The result is the achievement of forces on points on the surfaces or integral forces for each of the components of the aircraft. These forces will take into account viscous and transonic phenomena and interferences between components, with a reliable accuracy. Moreover, in the cases in which it is desired to analyze new and uncommon configurations, the estimation of certain aerodynamic characteristics through the traditional techniques of the preliminary design phase could be difficult or completely

impossible. This is because many of these methods, of an empirical or semiempirical nature, have been conceived and calibrated for use in the case of classical or ordinary configurations. In this case, the CFD analysis would come to the rescue, limiting the risks of making gross errors in the early design phases. [20]

But aerodynamics is not the only domain that needs to be investigated. In a multi-disciplinary design context, such as that of aircraft design, engineers have to deal with different disciplines at the same time, and necessitate to make sure the solutions they come up to are the closest possible to the optimum ones. Traditionally, in the early stages of design, engineers are in possess of what is defined as the Outer Mold-Line (OML) of the aircraft, and the details of other subsystems, such as structural layouts, are not examined until later design phases. But, in general, the configuration's OML geometry design is impacted by these subsystems, and it may be not discovered until high-fidelity analyses are performed in later development phases. Besides, further complications could arise whenever the geometry definition is scattered over many different analysis methods, which can span different disciplines as well as different fidelity levels within a discipline. What could happen is that geometry adjustments from redesign based on one method are likely to be inconsistent with the other geometry definitions. One solution to these difficulties and complications is, again, to generate a 3D model of the aircraft much earlier during the design study, as during the preliminary or even the conceptual phase, and to use this 3D model for all the possible analyses. Having a 3D model at disposal during conceptual design phase and performing on it higher fidelity analyses (alongside low-fidelity ones) would lead to a greater confidence in calculated data and for configuration down-select. Besides, there would be no need to generate new and more precise geometries as the design process progresses, because the 3D model of the aircraft would mature during the design process itself. [16]

What appears quite clear from the previous examination is that the performing of high-fidelity analyses is something really auspicable even during the first stages of the aircraft design process. But, in order to perform these analyses, an adequate geometry generator must be used. At this stage several approaches can be considered, especially when parametrization and MDO need to be taken into account.

A Free-Form Deformation (FFD) volume approach for CAD-free geometry parametrization can be followed, as presented in [18]. The FFD approach is borrowed directly from soft object animation in the computer graphics field and allows to perform high level modifications to complex objects by simply embedding them in a rubber-like volume, which is stretched and twisted in order to obtain, indirectly, the desired final shape for the object (figure 2.1). One key observation that needs to be made about FFD is that it parametrizes not the geometry itself, but the geometry change. This brings to an efficient computation of analytic derivatives for gradient-based optimization.

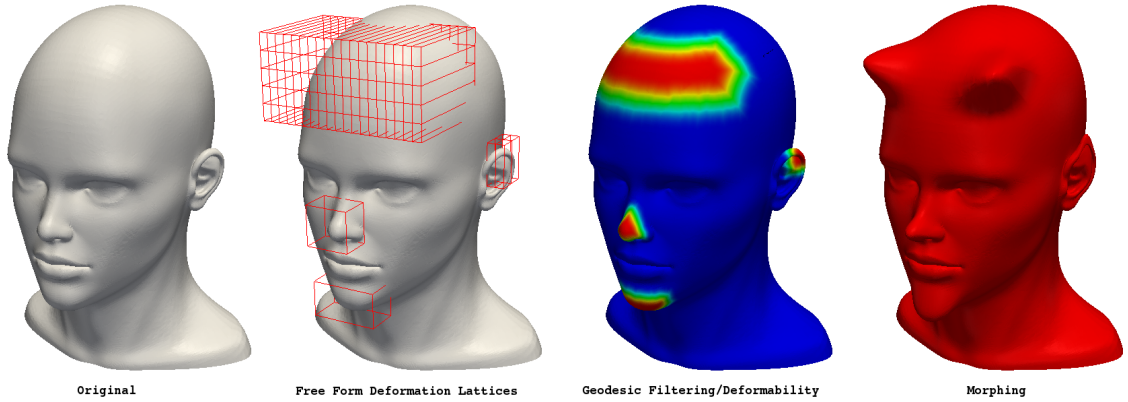

Figure 2.1 Free-form deformation example (www.optimad.it/wp-content/uploads/ffdprocedure.png)

A further approach to the problem is described in [20], in which the author examines the potential of parametric CAD systems and the help they can offer during the early stages of aircraft design. In a parametric CAD software, contrary to what happens in conventional CAD systems, at each step the user has the possibility to name the entity that is being created and to make a parameter one or more values of the entity itself. In the event that one of the parameters is changed, all the child entities of the element to which the mutated parameter belongs are updated.

One last possible approach is the one used in [16], which involves the generation of a general geometry Application Programming Interface (API) in order to create solids-based geometries. In particular, the API created by the authors (named EGADS) provides a solid modeling geometry kernel that supports both *Bottom-Up* construction (i.e., the process of producing a solid model from its constituent geometric entities) as well as the ability to perform Constructive Solid Geometry (CSG) operations. Both capabilities are necessary in order to build closed 3D models, which are essentials for high-fidelity analyses. The core of EGADS is Open CASCADE, a fully functional open-source solid modeling geometry kernel, which incorporates all the methods and operations necessary to perform shape manipulation. Since the approach is *Bottom-Up* and programmatic (each of the geometric and topological entities is assigned one or more manageable variables) the 3D CAD models produced will be, to a certain extent, parametrized *a priori*. As it will be more clear from the following paragraphs, the approach used for JPAD is pretty similar to the latter.

2.2 Open CASCADE Technology

As anticipated in section 2.1, the `JPADCAD` module helps the user to build from scratch, in a *Bottom-Up* approach, aircraft solid components ready to be passed to a CFD solver, in order to perform high-fidelity analysis. This would have not been possible without the adoption of an external library such as OCCT.

Open CASCADE Technology (OCCT) is an object-oriented C++ class library designed for rapid production of sophisticated domain-specific CAD applications. A typical application developed using OCCT deals with 2/3D geometric modeling in general-purpose or specialized

CAD systems, manifacturing or analysis applications, simulations applications, or even illustration tools. OCCT provides classes for several purposes, such as:

- basic data structures (e.g., geometric modeling, visualization, interactive selection);
- modeling algorithms;
- working with mesh;
- data interoperability with neutral formats, such as IGES and STEP.

The C++ classes are located into packages, which are organized into toolkits, which are finally grouped into seven modules [5] (figure 2.2):

- **Foundation Classes** module underlies all other OCCT classes;
- **Modeling Data** module supplies data structures to represent 2D and 3D geometric primitives and their compositions into CAD models;
- **Modeling Algorithms** module contains a wide range of geometrical and topological algorithms;
- **Mesh** module implements tessellated representation of objects;
- **Visualization** module provides mechanisms for graphical data representation;
- **Data Exchange** module interacts with popular data formats and relies on **Shape Healing** module to improve compatibility between different CAD software;
- **Application Framework** module offers solutions for handling application-specific data and commonly used functionality (e.g., save/restore, undo/redo, copy/paste).

In addition, Open CASCADE comes equipped with a convenient testing tool, called **Test Harness** or **Draw**, that can be used to test and prototype various algorithms before building an entire application.

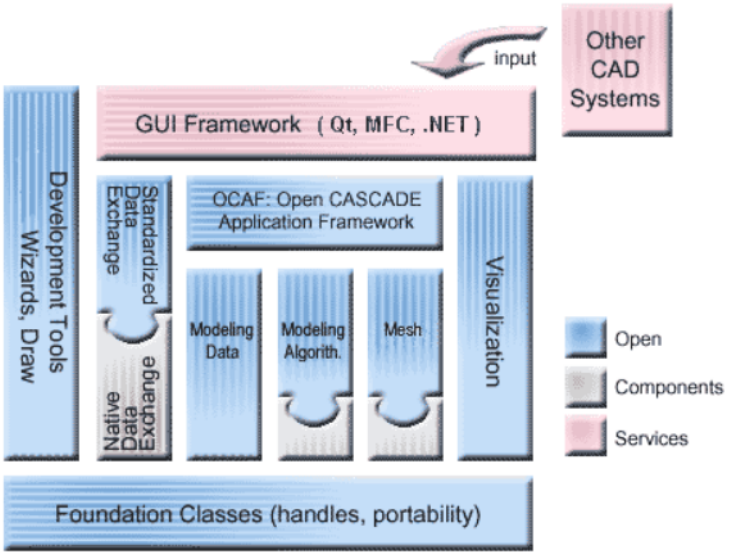

Figure 2.2 OCCT module structure

In the following subsections more will be said about the Modeling Data module, the Modeling Algorithms module, and the Data Exchange module, because these are the packages which group the majority of the classes used within `JPADCAD`.

2.2.1 Modeling Data and Algorithms

Modeling Data module supplies data structures to implement Boundary Representation (BRep) of objects in three dimensions. In BRep each shape is an aggregation of geometry within topology. The geometry is the mathematical description of a shape, and it is given by curves and surfaces, which can be of different types (e.g., Bezier, B-Spline, Non-Uniform Rational B-Spline (NURBS)). The topology is a data structure binding geometrical objects together.

On the other hand, Modeling Algorithms module groups a wide range of topological and geometric algorithms used in modeling. These algorithms can be divided in two major groups. The first one comprehends high-level routines used in the real design, such as primitives construction (e.g., boxes, prisms, cylinders), kinematic modeling (e.g., prisms, pipes,), Boolean operations, and algorithms for local modifications (such as hollowing and shelling). The second one groups low-level mathematical support functions, low-level geometric tools, and topological tools providing algorithms to tesselate shapes, convert shapes to NURBS geometry, sew connected topologies, etc.

Geometry

Geometric entities are grouped in two packages: `Geom2d` and `Geom`. As the names suggest, `Geom2d` package defines geometric objects in 2D space, while `Geom` package deals with geometric entities in three dimensions. These packages provide classes for:

- description of points, vectors, curves and surfaces (just `Geom`, for obvious reasons);
- their positioning in the plane/space using coordinates systems;
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following geometric objects are available in OCCT, with the last four entities pertaining to the `Geom` package:

- point,
- vector,
- direction,
- axis,
- line,
- conic (circle, ellipse, hyperbola, parabola),
- bounded curve (trimmed curve, Bezier curve, NURBS curve),

- offset curve,
- elementary surface (plane, cylinder, cone, sphere, torus),
- bounded surface (rectangular trimmed surface, Bezier surface, NURBS surface),
- swept surface (surface of linear extrusion, surface of revolution),
- offset surface.

One peculiar characteristic of `Geom` curves and surfaces is that they are parametrized. For this reason each class comes equipped with functions that allow the user to work with the parametric equation of the curve or surface and to compute the point of a certain parameter u on a curve or the points of parameter (u, v) on a surface, together with the derivatives (in form of vectors) of every order at this point.

Topology

OCCT topology allows accessing and manipulating data of objects without dealing with their 2D or 3D representation. Whereas OCCT geometry provides a description of objects in terms of coordinates or parametric values, topology describes data structures of objects in parametric space. These description use location in and restriction of parts of this space. Abstract topological data structure describes a basic entity (i.e., a shape), which can be divided into the following component topologies.

- **Vertex**: a zero-dimensional shape which correspond to a point in geometry.
- **Edge**: a shape corresponding to a curve, and bound by a vertex at each extremity.
- **Wire**: a sequence of edges connected by their vertices.
- **Face**: part of a surface bounded by a closed wire.
- **Shell**: a collection of faces connected by some edges of their wire boundaries.
- **Solid**: a part of 3D space bound by a shell.
- **Solids Compound**: a collection of solids.

OCCT topology is provided by six packages. In the following list, the first three packages describe the topological data structure used in Open CASCADE Technology, while the remaining three provide tools to access and manipulate this topology.

- `TopAbs` package groups all the necessary resources for topology-driven applications. It contains enumerations that are used to describe basic topological notions, like topological shape, orientation and state. It also provides methods to manage these enumerations.
- `TopLoc` package provides resources useful to handle 3D local coordinates systems.
- `TopoDS` package contains classes to model and build data structures that are purely topological.

- **TopTools** package provides basic tools to use on topological data structures.
- **TopExp** package provides classes to explore and manipulate the topological data structures contained in the TopoDS package.
- **BRepTools** package provides classes to explore, manipulate, read and write BRep data structures. These more complex data structures combine topological descriptions with additional geometric information, and include rules for evaluating equivalence of different possible representation of the same object.

Geometry Utilities

In order to manage geometry, there are several utilities which provide the following services. Just a few of these tools is contained in the Modeling Data package, while the majority is located in the Modeling Algorithms one.

- **Creation of shapes by interpolation and approximation**. In modeling, it is often required to approximate or interpolate points into curves and surfaces. In interpolation, the process is complete when the curve or surface passes through all the points; in approximation, when it is as close to these points as possible. In Open CASCADE, these processes can be performed both in 2D and in 3D, providing the interpolator/approximator method a set of 2D or 3D points. In particular, packages Geom2dAPI and GeomAPI provide simple methods for approximation and interpolation with minimal programming. Different types of approximator/interpolator curves can be obtained, such as B-Spline, Bezier and NURBS, by simply selecting the right options.
- **Direct construction of shapes**. Direct construction methods from gce, GC and GCE2d packages provide algorithms to build elementary geometric entities, such as lines, circles and curves. They complement the reference definitions provided by the gp, Geom and Geom2d packages. The algorithms implemented by gce, GC and GCE2d are simple: there is no creation of objects defined by advanced positional constraints. For example, to construct a circle one could simply provide a point and a radius. The above mentioned simple entities that can be created comprehend:
  - 2/3D lines,
  - 2/3D conics (circles, ellipses, hyperbolae, parabolae),
  - arcs (of circle, ellipse, parabola, hyperbola),
  - planes,
  - cylinders,
  - cones,
  - transformations.
- **Conversion of curves and surfaces to B-Spline curves and surfaces**. The conversion to and from B-Spline has two distinct purposes. The first one is to provide a homogeneous formulation which can be used to describe any curve or surface. The second one is that

it can be used to divide a B-Spline curve or surface into a series of curves or surfaces, thereby providing a higher degree of continuity. All these utilities are grouped in two packages: `Geom2dConvert` and `GeomConvert`.

- **Computation of the coordinates of points on 2D and 3D curves**. OCCT gives the user the possibility to use complex algorithms that compute points on a 2D or 3D curve. For example, the following characteristic points exist on parametrized curves in 3D space:
  - points equally spaced on a curve,
  - points distributed along a curve with equal chords,
  - a point at a given distance from another point on a curve.

  Even more options become available by the use of dedicated packages, such as `GCPnts`.

- **Calculation of extrema between shapes**. The classes to calculate the minimum distance between points, curves, and surfaces, both in 2D and 3D, are provided by `Geom2dAPI` and `GeomAPI` packages. These packages give the user the capability to calculate the extrema of distance between:
  - a point and a curve,
  - a point and a surface,
  - two curves,
  - a curve and a surface,
  - two surfaces.

- **Calculation of intersections between shapes**. Four different types of intersection can be calculated in OCCT:
  - the intersections between two 2D curves;
  - the self-intersections of a 2D curve;
  - the intersection between a 3D curve and a surface;
  - the intersection between two surfaces.

  Intesection between curves in 2D space is managed by the `Geom2dAPI_InterCurveCurve` class, while, in 3D space, the intersection between a curve and a surface and two different surfaces is managed, respectively, by the `GeomAPI_InterCS` class and the `GeomAPI_InterSS` classes.

- **Construction of 2D lines and circles from constraints**. OCCT provides methods to build, in 2D space, circles and lines using numeric or geometric constraints in relation to other curves. These constraints can impose the following:
  - the radius of a circle,
  - the angle that a straight line makes with another straight line,

- the tangency of a straight line or circle in relation to a curve,
- the passage of a straight line or circle through a point,
- the circle with center in a point or curve.

These algorithms are much more complex than the ones described in the Direct Construction section.

- **Construction of curves and surfaces from constraints**. High level functions can be used in OCCT in order to create faired and minimal variation 2D curves (`FairCurve` package), ruled, pipe and filling of surfaces (`GeomFill` package), and plate surfaces (`GeomPlate` package).
- **Computation of projections**. OCCT provides algorithms in order to compute:
  - the projections of a 2D point onto a 2D curve;
  - the projections of a 3D point onto a 3D curve or surface;
  - the projection of a 3D curve onto a surface.

  The classes providing these services are grouped in the `Geom2dAPI` and `GeomAPI` packages.

Topology Tools and Algorithms

In order to generate topological entities starting from geometric objects or other entities of topological type, the Modeling Algorithms package is provided with different sub-packages, each one specialized in a specific domain. These packages are as follows.

- **`BRepBuilderAPI`** - Provides classes in order to create BRep topology data structure. The `BRepBuilderAPI_MakeShape` class is the root of all `BRepBuilderAPI` classes, which build shapes. In order to generate an edge element, for example, a `BRepBuilderAPI_MakeEdge` object will be created, providing several methods to build an edge from a curve. The `BRepBuilderAPI` package also provides classes which allow the creation of connected topology (shells and wires) from a set of separate topological elements (faces and edges). The sewing algorithm is implemented in the class `BRepBuilderAPI_Sewing`, which provides methods to load initial data, set customization parameters (such as tolerances), apply analisys methods, and sew the loades shapes.
- **`BRepFilletAPI`** - Provides classes in order to create fillets and chamfers. A fillet is a smooth face replacing a sharp edge and the class which allows filleting a shape is `BRepFilletAPI_MakeFillet`. To produce a fillet it is necessary to define the filleted shape at the construction of the class and add fillet descriptions (such as the radius) using the `Add` method. Fillet with changing radius can be also created. On the other hand, a chamfer is a rectilinear edge replacing a sharp vertex of the face and can be performed by using the `BRepFilletAPI_MakeChamfer` class. The procedure to produce a chamfer is almost equal to the one for the fillet construction.
- **`BRepOffsetAPI`** - It groups classes providing the following services:

- creation of offset shapes and their variants, such as hollowing, shelling and lofting;
- creation of tapered shapes using draft angles;
- creation of sweeps (i.e., objects obtained by sweeping a profile along a path).

- **`BRepPrimAPI`** - Provides classes for primitives construction, such as boxes, cones, cylinders and prisms. These classes also allow to create partial solids, such as a sphere limited by longitude, and build specific sub-parts by sweeping along a profile (altough for swept constructions along complex profiles such as B-Spline curves the `BRepOffsetAPI` package shoul be preferred).

- **`BRepAlgoAPI`** - This package mainly deals with Boolean operations. Boolean operations are used to create new shapes from the combinations of two shapes. These operations are the most convenient way to create real industrial parts. As most industrial parts consist of several simple elements such as gear wheels, arms, holes, ribs, tubes and pipes, it is usually easy to create those elements separately and then to combine them by Boolean operations in the whole final part. Four operations can be performed, provided by five main classes, of which one (`BRepAlgoAPI_BooleanOperation`) is the root:

  - `BRepPrimAPI_Fuse` performs the fuse operation;
  - `BRepPrimAPI_Common` performs the common operation;
  - `BRepPrimAPI_Cut` performs the cut operation;
  - `BRepPrimAPI_Section` performs the section operation, whose result is a compound made of edges.

  In general, Boolean operations can be performed in OCCT for various types of arguments. For arguments having the same shape type, for example, the result is a compound, containing shapes of this type. In case of different shape type arguments, some boolean operations can not be performed. For example, the fuse operation between a shell and a solid can not be done, while the cut operation can be performed, provided that the shell is the argument (i.e., the object of the operation) and the solid is the tool.

- **`BRepFeat`** - This package helps the user for creation and manipulation of form and mechanical features that go beyond the classical boundary representation of shapes. The form features are depressions or protrusions including types such as cylinders, prisms and pipes.

2.2.2 Data Exchange

Data Exchange module allows developing OCCT-based applications that can interact with other CAD systems by writing and reading CAD models to and from external data. The exchanges run smoothly regardless of the quality of external data or requirements to its internal representation, for example, to the data types, accepted geometric inaccuracies, etc. Data Exchange module is organized as a set of interfaces that comply with various CAD formats: IGES, STEP, STL, VRML, etc. These interfaces allow software based on OCCT to exchange data with various CAD software packages, mantaining a good level of interoperability.

- **Standardized Data Exchange** interfaces allow querying and examining the input file, converting its contents to a CAD model and running validity checks on a fully translated shape. The following formats are currently supported:
  - STEP (AP203: Mechanical Design; AP214: Automotive Design),
  - IGES (up to 5.3),
  - VRML and STL meshes.
- **Extended Data Exchange** (XDE) allows translating additional attributes attached to geometric data (colors, layers, names, materials, etc.).
- **Advanced Data Exchange Components** are available in addition to Standard Data Exchange interfaces to support interoperability and data adaptation with CAD software using the following proprietary formats:
  - ACIS SAT,
  - Parasolid,
  - DXF.

2.2.3 Open CASCADE Technology Java Wrapper

The OCCT classes are natively written in C++, while JPAD is a program written in Java. In order to be able to use the OCCT classes in JPAD, it is necessary to use a tool that links the two languages. For this purpose, Open CASCADE offers its own Java Wrapper, a tool for wrapping Open CASCADE Technology C++ classes to Java language, to allow their use within Java applications. In particular, the wrapping is made using the Simplified Wrapper and Interface Generator (SWIG) tool [7]. SWIG is an interface compiler that connects programs written in C and C++ with languages such as Perl, Python, Java, etc. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that other languages need to access the undelying C/C++ code. The Open CASCADE Java Wrapper comes equipped with the SWIG interface file (occtypes.i), which contains definitions of basic OCCT types for SWIG and defines several SWIG macros facilitating wrapping of other OCCT types. For example, the `WRAP_AS_ENUM_INCLUDE(name)` macro is uded to wrap enumerations, while `WRAP_AS_PACKAGE(name)` is used to wrap package methods. The use of these macros allows blind yet customizable approach to wrapping, facilitating controllable but easy wrapping of everything what is needed. The OCCT Software Development Kit (SDK) currently in use along with the Java wrapper is the 7.0.0 version, altough, at the moment this document is being written, the most recent version is the 7.3.0.

2.3 JPADCAD structure

The Open CASCADE Technology library is extremely useful for various reasons, which can be listed as follows:

- supports Manifold and Non-Manifold Geometry;

- gives the user the possibility to perform *Bottom-Up* construction;
- has both CSG operations and other abstract *feature-like* construction methods;
- can read and write IGES, STEP and native file formats;
- is a fully object-oriented API with thousands of methods and millions of lines of code.

The last point is both an advantage and a disadvantage. Having a plethora of methods to chose from surely gives the users all they necessitate to realize the most various applications. On the other hand, the level of programming complexity with the huge suite of methods makes the use of Open CASCADE rather a difficult undertaking. Moreover, its lack of a complete documentation adds to the enormous task of understanding this large but capable software suite.

All of these issues provide the motivation for the design and the development of a system of classes, interfaces and utilities which wrap the original OCCT classes, allowing the current and future JPAD developers to deal with a lowest number of well documented methods and functions, ready to be used in aerospace CAD modeling. The following paragraphs deal with the structure of this system, which, it is good to clarify, is still under costant development. For this reason, some of the most advanced features (for example, those concerning Boolean operations) still need to be adequately wrapped and integrated into the JPADCAD utilities superstructure.

2.3.1 Abstract Factory pattern

Most of the classes involved in the OCCT library wrapping are in the shape domain. As for the OCCT library, many topological entities must be managed at the same time in order to build solid models ready to be imported into external tools. The necessity to deal with many different entities all pertinent to the same domain as the possibility to employ, in the next future, geometric modeling kernels other than OCCT, has led to the adoption, for the development of the JPADCAD superstructure, of a particular Design Pattern.

By definition, Design Patterns are reusable solutions to commonly occurring problems in the context of software design [8]. Design Patterns were started as best practices that were applied again and again to similar problems encountered in different contexts. They became popular after they were collected, in a formalized form, in the Gang of Four book in 1994. The pattern adopted for JPADCAD is the one that goes under the name of Abstract Factory pattern. The Abstract Factory pattern offers the interface for creating a family of related objects, without explicitly specifying their classes [1]. The classes that participate to the Abstract Factory pattern are the followings.

- **`AbstractFactory`** - declares an interface for operations that create abstract products.
- **`ConcreteFactory`** - implements operations to create concrete products.
- **`AbstractProduct`** - declares an interface for a type of product object.

- **`Product`** - defines a product to be created by the corresponding `ConcreteFactory`; it implements the `AbstractProduct` interface.
- **`Client`** - uses the interfaces declared by `AbstractFactory` and `AbstractProduct` classes.

The `AbstractFactory` class is the one that determines the actual type of the concrete object and creates it, but it returns an abstract pointer to the concrete object just created. This determines the behavior of the `Client` that asks the factory to create an object of a certain abstract type and to return the abstract pointer to it, keeping the `Client` from knowing anything about the actual creation of the object. The fact that the factory returns an abstract pointer to the created object means that the client doesn't have knowledge of the object's type. This implies that there is no need for including any class declarations relating to the concrete type, the client dealing at all times with the abstract type. The objects of the concrete type, created by the factory, are accessed by the `Client` only through the abstract interface. Figure 2.3 contains an Unified Modeling Language (UML) diagram which shows how the pattern works.

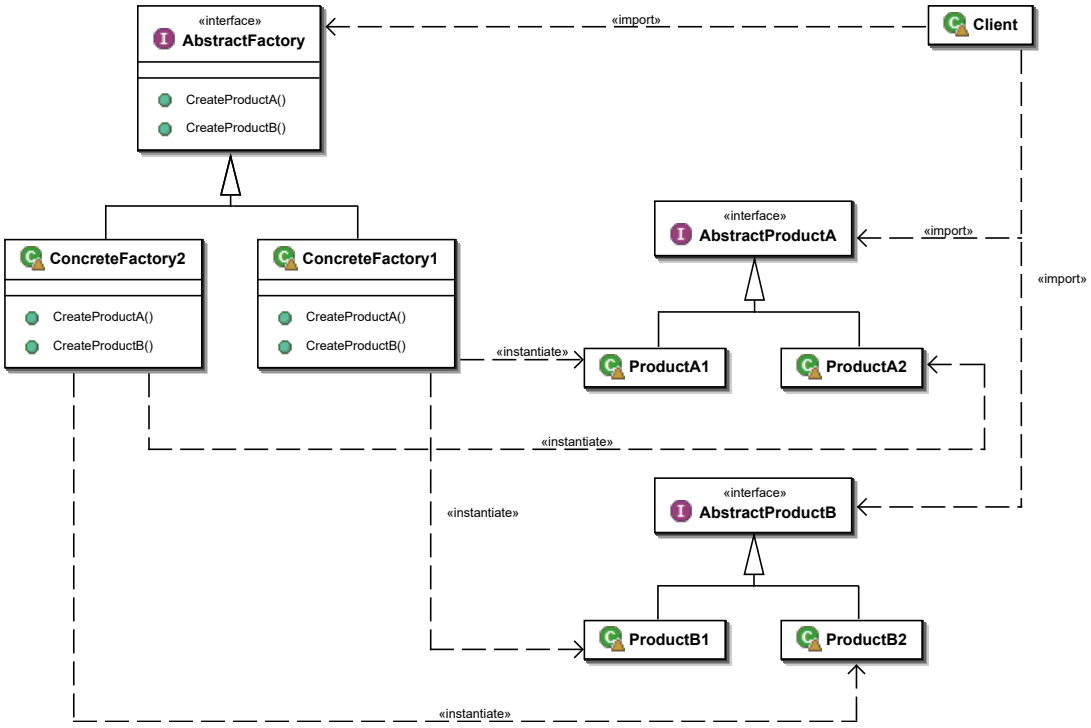

Figure 2.3 Abstract Factory pattern UML

As figure 2.3 shows, the `Client` class that requires the `ProductA` and `ProductB` objects doesn't instantiate the `ProductA1` and `ProductB1` classes directly. Instead, the `Client` refers to the `AbstractFactory` interface for creating objects, which makes the `Client` independent of how the objects are created (i.e., which concrete classes are instantiated). The `ConcreteFactory1` class implements the `AbstractFactory` interface by instantiating the `ProductA1` and `ProductB1` classes. Obviously the same can be said for `ConcreteFactory2`, and `ProductA2` and `ProductB2` classes.

2.3.2 JPADCAD classes and utilities

The organizational structure of the JPADCAD package follows the Abstract Factory pattern paradigm. In the specific case under examination, the abstraction from which all the classes of the package derive consists in considering the package itself as the superstructure of a generic (not better specified) kernel for CAD modeling. Therefore, all the classes of the package that have the CAD prefix are abstract classes (interface or abstract class reference types in Java) which therefore need to be implemented. Back to the Abstract Factory pattern, in the case of JPADCAD the abstract factory is the abstract class CADShapeFactory. As specified in the previous paragraph, the motivation that led to the use of the Abstract Factory pattern is the possibility that, in the future, more than just one kernel for geometric modeling could be used. Therefore, at present, being OCCT the only CAD library underlying JPAD, the abstract factory CADShapeFactory has a single implementation, consisting of the OCCShapeFactory class. In general, all the JPADCAD classes having the OCC prefix are concrete classes related to the implementation of the OCCT geometric modeling library.

JPADCAD currently comprises almost fourty classes, most of which are related to topological entities. More specifically, these classes can be distinguished as in table 2.1.

Class Type	Characteristics
Geometry classes	Define geometric entities (curve, surface)
Shape classes	Define topological entities (vertex, edge, etc.)
Factory classes	Provide factory methods
Explorer classes	Define topological data structure explorers
Iterator classes	Define iterators on sub-shapes
Shape type classes	Provide type-safe enumeration of CAD types
Utility classes	Provide methods to build/manipulate CAD entities
JavaFX export classes	Provide data structure to import shapes into a JavaFX window

Table 2.1 JPADCAD classes and characteristics

Shape classes

Shape classes featured in JPADCAD follow those offered by OCCT, with table 2.2 showing a comparison between the topology classes of Open CASCADE and the JPAD ones. As the table reports, each topological entity has both an interface and its implementation. Moreover, each class specific to a particular topological entity (both interface and concrete class) extends a generic shape class, which is CADShape for interface classes and OCCShape for concrete classes.

Open CASCADE	JPAD	
	Abstract Topology	Concrete Topology
TopoDS_Shape	CADShape	OCCShape
TopoDS_Vertex	CADVertex	OCCVertex
TopoDS_Edge	CADEdge	OCCEdge
TopoDS_Wire	CADWire	OCCWire
TopoDS_Face	CADFace	OCCFace
TopoDS_Shell	CADShell	OCCShell
TopoDS_Solid	CADSolid	OCCSolid
TopoDS_CompSolid	CADCompSolid	OCCCompSolid
TopoDS_Compound	CADCompound	OCCCompound

Table 2.2 JPADCAD topology classes overview

Figure 2.4 gives a complete overview on how topology classes are related to each other. Being CADShape a generic shape interface, it features basic methods common to all type of shapes. These methods are then specified by OCCShape, using the entities and the algorithms of the OCCT library. Table 2.3 presents a list of all the methods featured in CADShape, with a brief explanation for each one. CAD-type topology interfaces implement the methods in table 2.3 yet adding their own ones, specific to the entity they represent. For example, CADEdge interface extends CADShape and adds to it methods like range(), which returns an array containing minimum and maximum values of the parameter describing the geometry of an edge, and vertices(), that returns the extremities of an edge in terms of CADVertex entities. In the same way, CADSolid interface presents its own method which is getVolume(), that returns the volume of a solid entity.

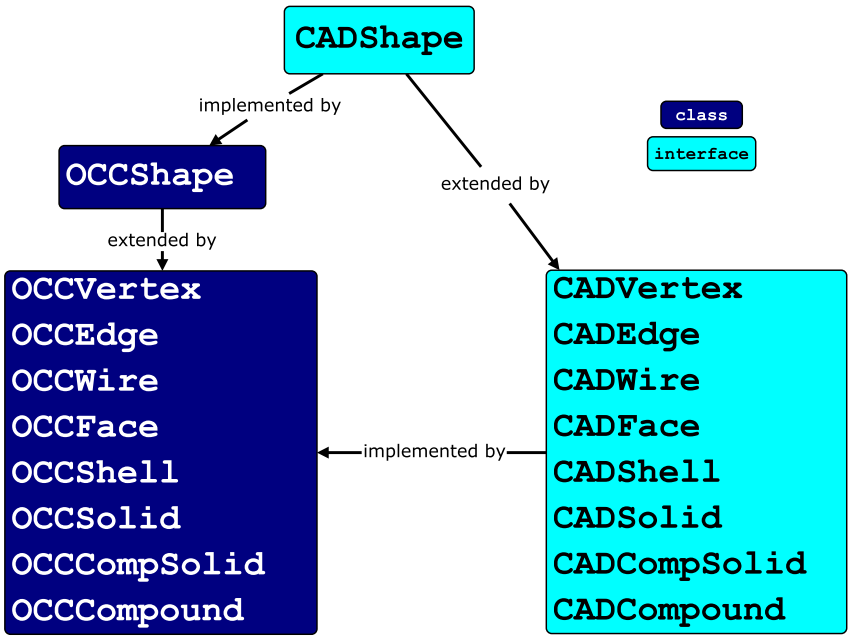

Figure 2.4 Relationship diagram for JPADCAD topological classes and interfaces

Method	Description
`boundingBox()`	Returns the bounding box of the shape in a `double` array format
`reversed()`	Returns a reversed instance of the shape
`orientation()`	Returns `int` value 1 if the shape is forward oriented
`isOrientationForward()`	Returns `boolean` value `true` if the shape is forward oriented
`equals(Object o)`	Returns `boolean` value `true` if `o` has same orientation and geometry as the shape
`isSame(Object o)`	Returns `boolean` value `true` if `o` has same geometry as the shape
`writeNative(String file)`	Writes the shape into the native format
`hashCode()`	Returns a hash code matching the equals method

Table 2.3 `CADShape` methods

As previously stated, `OCCShape` generic class implements the `CADShape` interface, adding to it also an empty constructor, a shape attribute (of the `TopoDS_Shape` type), and setter and getter for the shape attribute. Each `OCC` topology class that extends `OCCShape` then specifies its own constructors. Listing 2.1 has been included as an example of how shell-type elements can be built, as it reports a partial list of the constructors belonging to this class, plus a private method which provides the actual algorithm through which all the constructors work. The `myShape` variable reported in this listing is the previously mentioned shape attribute that each class that extends the `OCCShape` one acquires. One thing though should be highlighted: none of the constructors is actually used directly to build new shape objects (as for the shell-type elements as for the others). Following the Abstract Factory pattern, new elements are built through the use of factories and utility classes, as reported in the next paragraphs.

```
1  public class OCCShell extends OCCShape implements CADShell
2  {
3    // Shell area
4    private double area = 0.0;
5
6    // Shell builder attributes.
7    private static long defaultMakeSolid = 0;
8    private static long defaultMakeRuled = 0;
9    private static double defaultPrec3D = 1.0e-06;
10
11   // Setters and getters
12   public static boolean isDefaultMakeSolid() {
13       return (defaultMakeSolid == 1);
14   }
15
16   public static void setDefaultMakeSolid(boolean value) {
17       OCCShell.defaultMakeSolid = (value) ? 1 : 0;
18   }
19
20   public static boolean isDefaultMakeRuled() {
21       return (defaultMakeRuled == 1);
```

```
22      }
23
24      public static void setDefaultMakeRuled(boolean value) {
25          OCCShell.defaultMakeRuled = (value) ? 1 : 0;
26      }
27
28      public static double getDefaultPrec3D() {
29          return defaultPrec3D;
30      }
31
32      public static void setDefaultPrec3D(double defaultPrec3D) {
33          OCCShell.defaultPrec3D = defaultPrec3D;
34      }
35
36      @Override
37      public double getArea() {
38          return area;
39      }
40
41      // CONSTRUCTORS
42
43      /**
44       * Default empty constructor
45       */
46      public OCCShell() {
47      }
48
49      /**
50       * Builds a shell through a list of curves
51       */
52      public OCCShell(List<CADGeomCurve3D> cadGeomCurveList) {
53
54          myShape = OCCShellThruSections(
55              null, cadGeomCurveList, null,
56              defaultMakeSolid, defaultMakeRuled, defaultPrec3D
57              );
58      }
59
60      /**
61       * Builds a shell making it pass through a list of curves and an initial vertex
62       */
63      public OCCShell(OCCVertex v0, List<CADGeomCurve3D> cadGeomCurveList) {
64
65          myShape = OCCShellThruSections(
66              v0, cadGeomCurveList, null,
67              defaultMakeSolid, defaultMakeRuled, defaultPrec3D
68              );
69      }
70
71      /**
72       * Builds a shell making it pass through a list of curves and a final vertex
73       */
74      public OCCShell(List<CADGeomCurve3D> cadGeomCurveList, OCCVertex v1) {
```

```
75
76          myShape = OCCShellThruSections(
77              null, cadGeomCurveList, v1,
78              defaultMakeSolid, defaultMakeRuled, defaultPrec3D
79              );
80      }
81
82      /**
83       * Builds a shell making it pass through a
84       * list of curves and initial and final vertices
85       */
86      public OCCShell(
87          OCCVertex v0, List<CADGeomCurve3D> cadGeomCurveList, OCCVertex v1) {
88
89          myShape = OCCShellThruSections(
90              v0, cadGeomCurveList, v1,
91              defaultMakeSolid, defaultMakeRuled, defaultPrec3D
92              );
93      }
94
95      /**
96       * Actual shell builder, based on OCCT low-level algorithms
97       */
98      private TopoDS_Shape OCCShellThruSections(
99            OCCVertex v0, List<CADGeomCurve3D> cadGeomCurveList, OCCVertex v1,
100           long solid, long ruled, double pres3d) {
101       ...
102     }
103 }
```

Listing 2.1 `OCCShell` constructors and methods

Geometry classes

`JPADCAD` currently comprises six geometry classes, dealing with both 2D and 3D geometric entities. As well as for the shape classes, the `CAD` prefix indicates the interfaces dealing with abstract geometry, while the `OCC` prefix indicates classes relative to the OCCT kernel implementation. Table 2.4 gives an overview on the geometry classes contained in `JPADCAD`. Another class should be added to this list: `OCCDiscretizeCurve3D`. This class, as the name suggests, is pertinent to the OCCT implementation and offers support for 3D curves discretization, giving the user the possibility to manipulate curve points.

Open CASCADE	JPAD	
	Abstract Geometry	Concrete Geometry
Geom2dAdaptor_Curve	CADGeomCurve2D	OCCGeomCurve2D
GeomAdaptor_Curve	CADGeomCurve3D	OCCGeomCurve3D
Geom_Surface	CADGeomSurface	OCCGeomSurface

Table 2.4 `JPADCAD` geometry classes overview

Basic interfaces contain just the methods, which are then implemented by the OCC classes by using the OCCT functions. These methods are specific for each geometric entity. As an example of this, table 2.5 shows the methods contained in CADGeomSurface, along with a brief description of the operations intended by each one of them.

Method	Description
`dinit(int degree)`	Initializes the degree of the surface
`value(double u, double v)`	Gets 3D coordinates from (u, v) coordinates
`lowerDistance(double[] p)`	Returns the distance of a point from the surface
`setParameter(double u, double v)`	Sets the (u, v) coordinates used for derivative and curvature operations
`d1U()`	Returns the u first derivative vector at the coordinates set by `setParameter`
`d1V()`	Returns the v first derivative vector at the coordinates set by `setParameter`
`d2U()`	Returns the u second derivative vector at the coordinates set by `setParameter`
`d2V()`	Returns the v second derivative vector at the coordinates set by `setParameter`
`dUV()`	Returns the cross derivative vector at the coordinates set by `setParameter`
`normal()`	Returns the normal to the surface at the coordinates set by `setParameter`
`minCurvature()`	Returns the minimum curvature at the coordinates set by `setParameter`
`maxCurvature()`	Returns the maximum curvature at the coordinates set by `setParameter`
`gaussianCurvature()`	Returns the Gaussian curvature at the coordinates set by `setParameter`
`meanCurvature()`	Returns the mean curvature at the coordinates set by `setParameter`
`curvatureDirections()`	Returns the direction of maximum and minimum curvature at the coordinates set by `setParameter`

Table 2.5 CADGeomSurface methods

Constructors are implemented by the OCC classes. The most important of these constructors reside in OCCGeomCurve3D, since the construction of main aircraft components 3D models starts from a series of curves, quite always provided by a good selection of points read from file. OCCGeomCurve3D actually features three types of constructor, which are partially reported in listing 2.2 along with some of the attributes and methods of the class. These constructors use classes belonging to the OCCT library. In particular, the constructors based on provided point lists perform interpolation through a crucial OCCT class, which is GeomAPI_Interpolate. This class is used to interpolate a B-Spline curve passing through an array of points, with a C2 continuity if tangency is not requested at any middle point. If tangency is requested at some point the continuity will be C1. If periodicity is requested the curve will be closed and the junction will be the first point given.

```
1  public class OCCGeomCurve3D implements CADGeomCurve3D
2  {
3    // Attributes
4    private CADEdge cadEdge = null;
5    private GeomAdaptor_Curve myCurve = null;
6    private final double[] range = new double[2];
7    private OCCDiscretizeCurve3D discret = null;
8    private double length = 0.0;
9
10   /**
11    * Returns a point on the curve
12    */
13   public double[] value(double p) {
14       ...
15   }
16
17   /**
18    * Discretizes myCurve splitting it in n arcs. The result is put into discret
19    */
20   public void discretize(int n) {
21       ...
22   }
23
24   /**
25    * Returns myCurve range
26    */
27   public double[] getRange() {
28       return range
29   }
30
31   /**
32    * Returns myCurve length
33    */
34   public double length() {
35       return length;
36   }
37
38   /**
39    * Returns the edge element represented by myCurve
```

```
40      */
41     public CADEdge edge() {
42         return cadEdge;
43     }
44 
45     /**
46      * Returns myCurve
47      */
48     public GeomAdaptor_Curve getAdaptorCurve() {
49         return myCurve;
50     }
51 
52     /**
53      * Returns the discretized curve obtained from discretize method
54      */
55     public OCCDiscretizeCurve3D getDiscretizedCurve() {
56         return discret;
57     }
58 
59     // CONSTRUCTORS
60 
61     /**
62      * Builds a 3D curve from an edge object
63      */
64     public OCCGeomCurve3D(CADEdge E) {
65         ...
66     }
67 
68     /**
69      * Builds a 3D curve from a list of double[] points. The second
70      * parameter tells the method whether the curve is periodic or not.
71      */
72     public OCCGeomCurve3D(List<double[]> pointsList, boolean isPeriodic) {
73         ...
74     }
75 
76     /**
77      * Builds a 3D curve from a list of double[] points. The second parameter tells
78      * the method whether the curve is periodic or not, while the third and fourth
79      * parameter set the initial and final tangents, respectively. The last parameter
80      * tells the method whether the tangents need to be scaled or not.
81      */
82     public OCCGeomCurve3D(List<double[]> pointList,
83         boolean isPeriodic,
84         double[] initialTangent, double[] finalTangent,
85         boolean doScale) {
86         ...
87     }
88 }
```

Listing 2.2 `OCCGeomCurve3D` constructors

Factory and Utility classes

Factories and utilities are the classes that are actually exploited by JPADCAD users in order to instantiate new geometric or topological entities. These classes use the aforementioned constructors to build new CAD objects. The abstract factory CADShapeFactory provides the methods that the concrete factory classes must implement. Currently, OCCShapeFactory is the only implementation provided in JPADCAD, allowing the construction of 3D CAD entities by the use of the OCCT classes. Below there is a list of the methods provided by the CADShapeFactory, along with a description of what they do and the constructors they make use of in the OCCT implementation.

- **getFactory, setFactory** - Getter and setter for the CADShapeFactory attribute factory. This attribute needs to be instantiated one time, whenever factory methods are needed.
- **newExplorer, newWireExplorer** - Methods through which new instances for generic topology explorer (CADExplorer) and wire explorer (CADWireExplorer) are created. These explorers are intended for topological data structure investigation. In case of a generic explorer, the shape to explore and the type of the shapes to be found must be provided.
- **newIterator** - Creates a CADIterator object, intended for sub-shapes iteration.
- **newShape** - Methods that generate CADShape generic entities. The input provided to the method can be:
  - a generic Object of the underlying implementation (e.g., a TopoDS_Edge object);
  - a String standing for the path to a file, from which CAD shapes need to be loaded;
  - a pair of CADShape objects and a char, in order to perform Boolean operations.

  In the OCCT implementation, Boolean operations are performed thanks to the classes contained in the BRepAlgoAPI package. The char parameter allows the user to select the type of the operation to be performed.
- **newCurve2D** - Creates a CADGeomCurve2D object. Currently the factory contains just one method for 2D curves construction, which needs as input the edge owning the 2D curve and the face (i.e., the plane) containing the same curve.
- **newCurve3D** - Methods used to create CADGeomCurve3D objects, exploiting the constructors reported in listing 2.2. These constructors (and so these methods) allow to generate 3D curves starting from:
  - an edge owning the 3D curve;
  - a pair of points in double[] format (in order to generate a segment);
  - an indefinite number (at least two elements) of points in double[] format;
  - a list of points in double[] format;
  - a list of points in double[] format, and initial and final tangents in terms of double[];
  - a list of gp_Pnt (an OCCT class used to represent points in 3D space);

- a list of `PVector` (a class, part of the Java Processing package, which describes two or three dimensional vectors [6]).

All the aforementioned methods that generate a 3D curve from points assignment also allow the user to set the curve as periodic.

- **`newVertex`** - Generates new vertices by providing coordinates in terms of `double`.
- **`newFacePlanar`** - Methods that allow to generate a `CADFace` as a planar triangle connecting three vertices. The vertices can be provided both in terms of `double[]` points and `OCCVertex` entities.
- **`newShell`** - These methods help the user to build a shell, making it pass through a selected list of curves, and initial and final vertices eventually. The `OCCShell` constructors actually allowing the necessary operations have been reported in listing 2.1. In particular, the `newShell` methods enable the user to generate shell elements from the following inputs:
  - a list of `CADGeomCurve3D`;
  - an initial `CADVertex`, a list of `CADGeomCurve3D`, one final `CADVertex`.

  The OCCT class `BRepOffsetAPI_ThruSections` provides the algorithms to perform `newShell` underlying operations. This class describes functions to build a loft (a shell or a solid entity) passing through a set of sections in a given sequence. It is necessary to point out that none of these methods is used directly to generate shell elements: the `OCCUtils` utility class actually contains the methods used in practice, as it will be clear shortly.
- **`newShellFromAdjacentFaces`** - It allows to build a shell by connecting adjacent faces. Faces can be provided both in an array of indefinite lenght and in a list. The algorithms that perform the operations in the OCCT implementation are provided by the `BRepBuilderAPI_Sewing` class.
- **`newSolidFromAdjacentFaces`** - Performs the same operations described above, producing a solid instead of a shell. In the OCCT implementation, the solid is produced by the use of the `BRepBuilderAPI_MakeSolid` class.

The concrete factory class `OCCShapeFactory`, as long as its methods, are not intended to be instantiated and exploited directly by `JPADCAD` users. The utility class `OCCUtils`, instead, is intended to be used in order to generate new factory instances through which factory methods can be performed. In particular, `OCCUtils` gives the user the capability to make the following operations.

1. Generate a new `OCCShapeFactory` instance and assigning it to its own `theFactory` attribute, which belongs to the `CADShapeFactory` type.
2. Access factory methods through `theFactory` attribute, performing the operations on shapes listed above.

In addition, OCCUtils contains several methods covering different necessities that can be encountered while managing CAD entities. Though some of these functions just make use of factory methods by simply rearranging the way their parameters are set (as for the methods making use of newShell factory functions), others allow the users to perform completely new manipulation operations on shapes. Obviously, the methods contained in this utility, as well as the utility class itself, are inherent to the OCCT implementation. The following list offers an overview on the methods contained in OCCUtils.

- **write** methods allow to write produced shapes to file. In order to be exported, shapes must be OCCShape generic instances. The methods currently contained in OCCUtils allow to write shapes just in BRep format. Other formats, such as STEP and IGES, are actually covered in another utility class, AircraftUtils, which, at the moment, is contained in JPADCADSandbox, and whose methods will be explained in the next chapters.

- **reportOnShape** and **numberOfShapes** methods enable to investigate shapes, providing a report (viewable on console) on the sub-shapes and their number.

- **pointProjectionOnCurve** methods enable to project points onto a 3D curve, providing CADVertex entities as result. The OCCT class GeomAPI_ProjectPointOnCurve is behind this operation.

- **splitEdge** functions allow to split edges or curves at one or more locations, returning a list of CADEdge entities as result. The Open CASCADE class underlying the operations performed by these methods is BOPAlgo_PaveFiller.

- **makeFilledFace** methods allow the user to generate filling surfaces, i.e., surfaces that can be used to fill empty gaps between faces. These methods make use of the OCCT class BRepOffsetAPI_MakeFilling and newShape factory methods.

- **makePatchThruSections** routines, finally, make use of newShell factory methods, in order to enhance user possibilities in terms of which type of parameters can be provided as input in order to generate a shell element from a sequence of points and curves.

The following paragraph shows some simple examples on how JPADCAD classes are used in order to produce basic shapes. The next chapter will deal on how all the aforementioned methods can be used in order to generate shapes representing more complex geometries, such as those behind main aircraft components.

2.3.3 Examples

The following example shows how to generate B-Spline curves and loft surfaces in JPADCAD. The methods used below are those descripted in the previous paragraph. Some particular attention has been put on such methods because they are among the most used for aircraft components CAD models production.

As previously explained, in order to be able to actually generate CAD shapes in JPAD it is necessary to instantiate a factory. For this reason, the first method called through OCCUtils

is `initCADShapeFactory`. After making sure that a `OCCShapeFactory` instance actually exists, the first thing to do is creating points for the curves. In the listing below, these points are created in `double[]` format, and passed to the `newCurve3D` method, along with a `boolean` value that states that the curve we want to generate is not periodic. The application of the previous method returns a generic `CADGeomCurve3D`, as required by the use of the Abstract Factory pattern. Exploiting the `edge` method provided by the `CADGeomCurve3D` interface, the `CADEdge` entity underlying the B-Spline curve can be obtained.

```
1   if(OCCUtils.theFactory == null)
2     OCCUtils.initCADShapeFactory();
3
4   // Create a list of points to be passed to the
5   // factory methods for 3D curves generation
6   double[] pntA = new double[] {0.0, -1.0, 0.0};
7   double[] pntB = new double[] {0.0,  0.0, 1.0};
8   double[] pntC = new double[] {0.0,  2.0, 0.0};
9
10  // Generate a B-Spline curve from the previous points
11  CADGeomCurve3D curve1 = OCCUtils.theFactory.newCurve3D(false, pntA, pntB, pntC);
12
13  // Get the edge entity underlying the previous curve
14  CADEdge edge1 = curve1.edge();
```

Listing 2.3 B-Spline curve creation

In order to produce other curves and to show how `newCurve3D` can actually manage more than just one set of parameters (thanks to Java polymorphism), the previous points are added to a Java List [13] and manipulated (after being inserted in the list), in order to generate the successive curves at a certain distance from the previous one. In this way, two more curves are made, with the last one being also imposed two tangency constraints, at the initial and final point respectively. Once all the curves are made, the `write` method is called by the use of `OCCUtils`, allowing to write the desired shapes to a BRep file. The result is shown in figure 2.5.

```
1   // Passing points to the B-Spline curves generator method through a Java List
2   List<double[]> points1 = new ArrayList<double[]>();
3
4   points1.add(pntA);
5   points1.add(pntB);
6   points1.add(pntC);
7   points1.forEach(d -> d[0] = 2.0);
8
9   CADGeomCurve3D curve2 = OCCUtils.theFactory.newCurve3D(points1, false);
10
11  // Get the edge entity underlying the previous curve
12  CADEdge edge2 = curve2.edge();
13
14  // Give to the B-Spline factory method initial and final tangent too
15  List<double[]> points2 = new ArrayList<double[]>();
```

```
16
17      points2.addAll(points1);
18      points2.forEach(d -> d[0] = 10.0);
19
20      double[] iTang = new double[] {0.0, 1.0, 0.0};
21      double[] fTang = iTang;
22
23      CADGeomCurve3D curve3 = OCCUtils.theFactory.newCurve3D(points2,
24          false,
25          iTang, fTang,
26          false
27          );
28
29      // Get the edge entity underlying the previous curve
30      CADEdge edge3 = curve3.edge();
31
32      // Writing the generated edges to BRep file
33      System.out.println("Have been the shapes correctly written to file? " +
34          OCCUtils.write("EdgeExample_01.brep",
35              (OCCEdge) edge1,
36              (OCCEdge) edge2,
37              (OCCEdge) edge3
38              ));
```

Listing 2.4 B-Spline curve creation with different `newCurve3D` methods

The last part of the example focuses on `newShell` and `makePatchThruSections` methods. The curves used as sections for the loft are those created in the first part of the example. Three lofts are made, exploiting three `makePatchThruSections` available variants. The example also shows how to generate `CADVertex` entities by use of a `double` array. Generated shapes are then transcripted to file and reported in figures 2.6, 2.7 and 2.8.

```
1       // Making a shell pass through the curves
2       OCCShape shell1 = OCCUtils.makePatchThruSections(curve1, curve2, curve3);
3
4       // Writing the generated shell to BRep file
5       System.out.println("Have been the shapes correctly written to file? " +
6           OCCUtils.write("ShellExample_01.brep", shell1));
7
8       // Making a shell pass through the above generated curves and an initial vertex
9       List<CADGeomCurve3D> curves = new ArrayList<CADGeomCurve3D>();
10
11      curves.add(curve1);
12      curves.add(curve2);
13      curves.add(curve3);
14
15      CADVertex iVtx = OCCUtils.theFactory.newVertex(new double[] {-10.0, 0.0, 0.0});
16
17      OCCShape shell2 = OCCUtils.makePatchThruSections(iVtx, curves);
18
19      // Writing the generated shell to BRep file
20      System.out.println("Have been the shapes correctly written to file? " +
```

```
21          OCCUtils.write("ShellExample_02.brep", shell2));
22
23      // Making a shell pass through initial and final vertices too
24      CADVertex fVtx = OCCUtils.theFactory.newVertex(new double[] {20, 0.0, 0.0});
25
26      OCCShape shell3 = OCCUtils.makePatchThruSections(iVtx, curves, fVtx);
27
28      // Writing the generated shell to BRep file
29      System.out.println("Have been the shapes correctly written to file? " +
30          OCCUtils.write("ShellExample_03.brep", shell3));
```

Listing 2.5 Lofts creation by use of `makePatchThruSections` methods

Figure 2.5 Edges obtained by use of `newCurve3D` methods

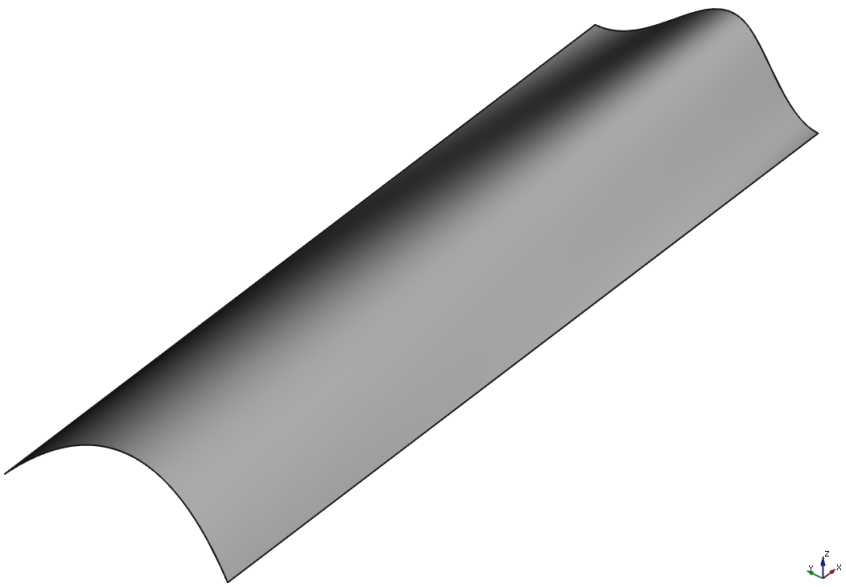

Figure 2.6 Loft passing through three curves

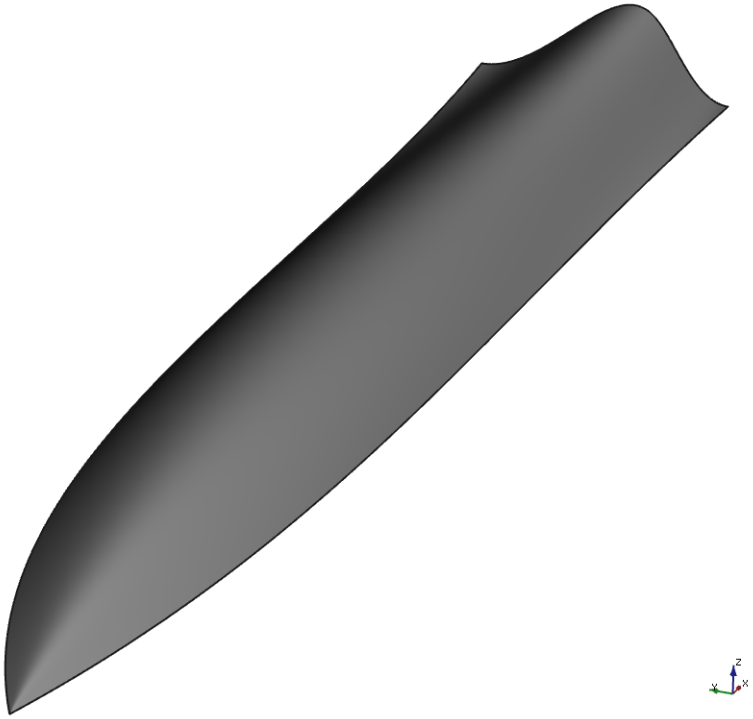

Figure 2.7 Loft passing through three curves and an initial vertex

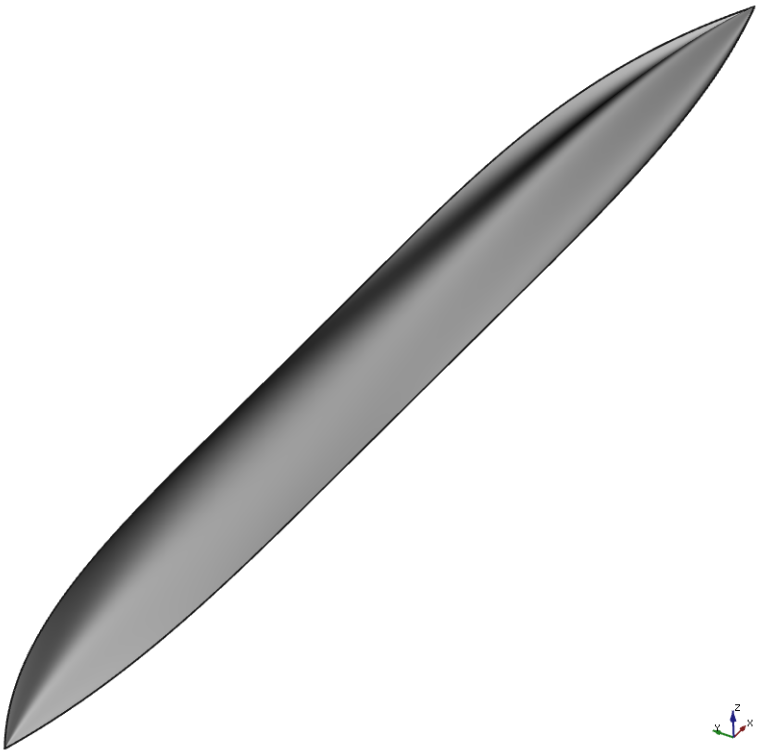

Figure 2.8 Loft passing through three curves and initial and final vertices

Chapter 3

A utility class for aircraft modeling: `AircraftUtils`

3.1 Introduction

Once the `JPADCAD` library was developed, providing basic methods for the construction of CAD shapes, these same methods have been used to create functions that, starting from the geometric characteristics of a given aeronautical component supplied in input, automatically generate the CAD model of the aircraft, complete of all of its components or just a given number of them. As stated in Chapter 1, JPAD comes equipped with classes and methods that allow to read data regarding the aircraft and its components from appropriately formatted XML files. Reading this data determines the creation of an instance of the JPAD `Aircraft` class. This class is the one delegated to the storage of all the data related to the aircraft, and also contains all the methods that allow access to the different sub-components of the airplane. These sub-components are treated in JPAD by the use of different classes. As for the sub-components currently modeled through the use of `JPADCAD`, the `Fuselage` and `LiftingSurface` classes are those delegated to the description, respectively, of the fuselage and all the different types of aerodynamic components in general. For this reason, the `LiftingSurface` class can be used, for example, to describe both wings and tail surfaces. The modeling of the CAD entities of the different components of an aircraft starts from the instances of these classes, as it will be more evident in the next paragraphs.

3.2 `AircraftUtils` overview

In order to develop and test methods allowing the construction, from scratch, of aircraft principal components, a new package has been added to the JPAD three. This package, named `JPADCADSandbox`, has been conceived with the intent to build some sort of *playground* for all the activities inherent to CAD modeling. For this reason, this package not only contains the test

classes produced to check the correct functioning of the methods contained in JPADCAD. On the contrary, at present, it also contains all the methods used to build the 3D counterparts of the instances of the Fuselage and LiftingSurface classes. These methods have all been collected in one class, along with others that will be listed in the following, which serves as an utility class, i.e., a class that does not need to be instantiated and provides several methods for multiple other classes. These methods can then be accessed in a static way, providing an approach to the solution of a problem which is not properly object oriented (like should be when using a programming language such as Java), but that produces easy to debug/mantain code, which is an important prerequisite especially when developing new functionalities. This utility class is called AircraftUtils and, along with the methods that have been referred very briefly above, it also provides functions that can be used to:

- load an aircraft from XML file;
- write aircraft solid components to file;
- get the whole aircraft shapes.

All these methods are set to public, meaning that they can be actually accessed by external classes. Furthermore, AircraftUtils contains several private methods (i.e., functions that can be used only by classes and methods internal to the class in which they are defined), that provide support when dealing, for example, with the wing tip building. Finally, it also includes some enumeration classes, dealing with file extension and spacing type. These enumerators are mainly used internally AircraftUtils, and more will be said about them and all the public extra methods in paragraph 3.5.

The next two paragraphs, instead, deal with the methodologies and the functions used to build main aircraft components: the fuselage and the lifting surfaces. In order to build these components properly, some sort of construction strategy must be followed. This is a crucial point, especially when dealing with the necessity of using the same code for the construction of the most various shapes and configurations. For this reason, the typical modeling strategies for aircraft geometric design were adopted [20], by making use of all the primary functions generally used when dealing with object such as fuselages and wings. Classically, typical aircraft components are modeled by means of loft functions. This type of functions, that have been described in paragraph 2.3.2 along with their implementation in JPADCAD, usually requires planar section curves, plus guide curves in some cases (actually not the underlyng OCCT algorithm used for the ThruSections methods), to build shapes that could not be obtained by use of simple extrude, revolve or sweep functions. When dealing with aircraft geometric modeling, loft functions are undoubtedly the most used ones. In order to actually use these functions, it is necessary to construct a wireframe first. This wireframe consists of a set of primary shapes (basically curves or wires) that helps building the final 3D model and provides a first glimpse at the result we coming up with. In aircraft modeling, two different types of primary shapes exist: wing-type shapes and fuselage-type shapes. In case of wing-type shapes the defining sections are parallel to the flow direction and are tipically airfoils, while in case of fuselage-type shapes the defining sections are normal to the flow direction and are symmetrical. Once the wireframes have been defined lofting operations can begin, giving

life to the succession of steps that will lead to the generation of the definitive 3D model of the aeronautical component. The methods described below, which return the shapes of primary aircraft components, follow the just described methodology. The next paragraphs therefore provide a focus on how these steps are followed and which classes and methods the construction functions make use of. Moreover, when explaining the algorithms behind the creation of the 3D model of a lifitng surface, a particular emphasis will be given to the methodology followed for the generation of the wing tip, which is the element of greater complexity of the 3D wing produced by JPAD.

3.3 Fuselage CAD method

In order to help the subsequent lofting operations, it was first necessary to conceive a subdivision of the surface of the fuselage. Thankfully JPAD `Fuselage` class helps doing this by providing a schematization of the external surface which can be easily used for CAD purposes. This scheme splits the fuselage in five principal parts, which can be listed as follows:

- nose cap,
- nose trunk,
- cylindrical trunk,
- tail trunk,
- tail cap.

When coming to the CAD generation problem, one could imagine each of the parts mentioned above having its own patch being produced by means of a lofting method. The union of these patches (obtained by the use of sewing algorithms) determines the final shape of the fuselage (figure 3.1). This subdivision not only helps building lofts representing the external surface, making the patching through sections phase much easier. It also provides an useful scheme that has been actually used to fix the parameters of the algorithm for the fuselage construction. The method doing this is called `getFuselageCAD` and the parameters it accepts are the followings.

- **`fuselage`** - An instance of the `Fuselage` type containing all the geometric information about the component. It provides easy access to the lists of points defining side curves and outlines. It also contains methods returning the x-coordinates of the sections delimiting one part of the fuselage from another.
- **`noseCapSectionFactor1`, `noseCapSectionFactor2`** - Entries defining the limits, in terms of x-coordinates, of the nose cap patch. In particular, these factors (which are expressed as Java `double`) multiply the nose cap offset length normalized with respect to the nose length, thus returning the normalized initial and terminal x-coordinates of the sections defining the shape of the nose cap.
- **`numberNoseCapSections`** - An `int` fixing the number of nose cap sections to patch through.

- **numberNosePatchSections** - An `int` defining the number of nose trunk sections.
- **spacingTypeNosePatch** - An instance of one of the Java `enum` classes mentioned above, defining the type of spacing to be adopted for the sections of the nose trunk.
- **numberTailPatchSections** - An `int` value defining the number of sections to be created for the tail trunk.
- **spacingTypeTailPatch** - The spacing type for the tail trunk sections.
- **tailCapSectionFactor1, tailCapSectionFactor2** - What has been said for the nose cap sections applies here. These factors simply fix the x-coordinates of the tail cap initial and terminal sections.
- **numberTailCapSections** - An `int` value fixing the number of tail cap sections to patch through.
- **exportSupportShapes, exportLofts, exportSolids** - Entries of the Java `boolean` type, allowing the user to select which kind of shapes he wants to generate and export, with the shapes being respectively: support shapes, such as those defining the wireframe of the fuselage; loft shapes, generated by patching through the fuselage sections; solid shapes, obtained by sewing the lofts.

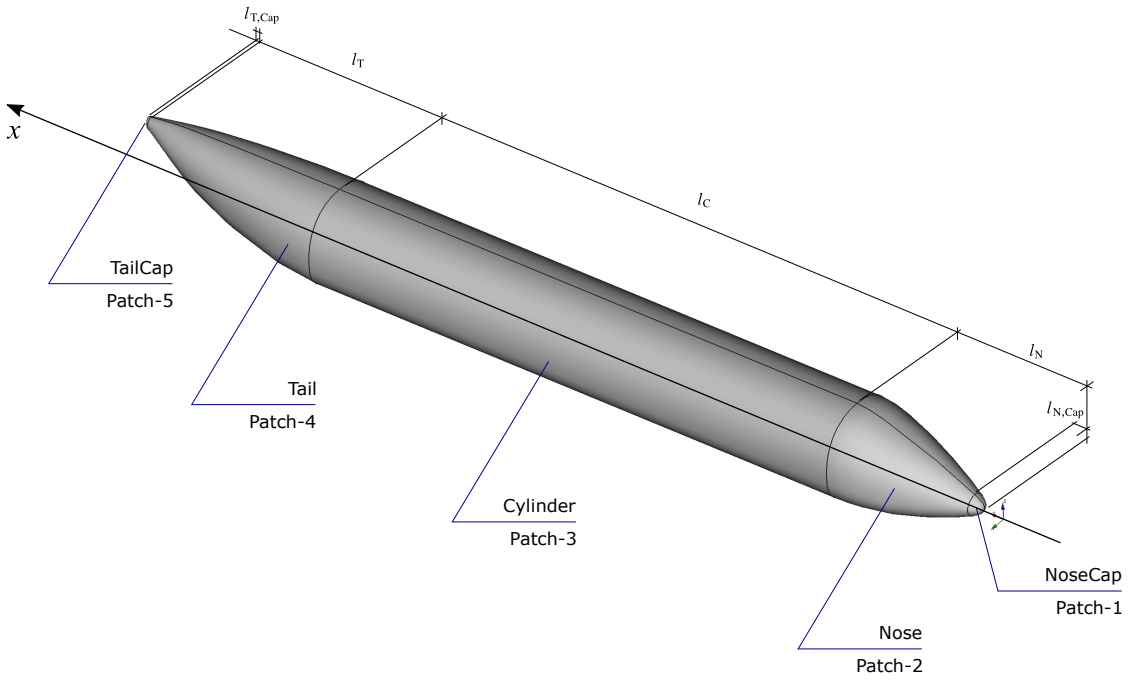

Figure 3.1 Fuselage schematization for CAD purposes

The algorithm can be split in two sections, with the first one providing for the generation of the lofts and the solid, while the second one is mainly dedicated to the production of the outline curves of the fuselage, completing the list of the extra shapes to be exported in case the `exportSupportShapes` variable has been set to `true`. After collecting essential geometric variables (such as those related to the lengths of the parts in which the fuselage has been split),

the construction of the five patches that compose the fuselage begins. The building process clearly starts from the nose cap and continues till the terminal section of the fuselage.

In order to determine the curves to patch through, the x-coordinates of the sections must be first calculated. As anticipated, the `noseCapSectionFactor1` and `noseCapSectionFactor2` parameters fix the position of the initial and terminal section of the nose cap patch. In particular, when respectively fixed to 0.0 and 1.0, first and last nose cap patch sections x-coordinates coincide with the actual initial and teminal x-coordinates of the nose cap. In order to avoid a degenerate section, the first parameter must be set to a value higher than 0.0 (usually a value equal to 0.15 seems to work fine). The number of nose cap sections provided to the algorithm determines how many x-coordinates must be calculated between initial and terminal ones, with the spacing between the sections being set to half-cosine type (with higher density towards the terminal section). JPAD utility class `MyArrayUtils` provides methods for spacing. Once all the x-coordinates have been determined, it is finally possible to acquire section points, by means of the `Fuselage.getUniqueValuesYZSideRCurve` method. It requires the x-coordinates of the sections expressed in terms of `Amount` (a Java class, part of the JScience package, providing support for measurements [15]) values, and returns a list of `PVector` entities, representing the coordinates of the fuselage side curve points. As the name of the method suggests, just the right curve points are returned, given the simmetry of the fuselage. For this reason, mirroring operations must be performed on lofts in order to obtain the complete shape of the fuselage, as it will be clarified in the following. Once all the lists of `PVector` points have been collected, the patch for the nose cap can be finally obtained, by means of one of the `OCCUtils` methods listed in Chapter 2. Since the patch needs to be made pass through one initial vertex and a set of curves defined by means of `PVector` entities, an appropriate variant of the `makePatchThruSections` methods has been used. Listing 3.1 contains pieces of the actual code used to generate nose cap shapes. Variables containing the term `bar` in their name hint at quantities which have been normalized with respect to the whole nose length, as specified above.

```
1  // Determine a List containing the sections x-coordinates normalized with
2  // respect to the nose length. xbarNoseCap is the normalized nose cap length.
3  List<Double> xbars1 = Arrays.asList(
4          MyArrayUtils
5            .halfCosine2SpaceDouble(
6            noseCapSectionFactor1*xbarNoseCap, noseCapSectionFactor2*xbarNoseCap,
7            numberNoseCapSections)
8          );
9
10
11 // Fill a List of PVectors for each section
12 List<List<PVector>> sections1 = new ArrayList<List<PVector>>();
13 xbars1.forEach(x -> sections1.add(
14           fuselage.getUniqueValuesYZSideRCurve(noseLength.times(x)))
15     );
16
17
18 // Generate the patch for the nose cap
19 PVector noseTipVtx = new PVector(0.0f, 0.0f, (float) zNoseTip.doubleValue(SI.METER))
```

```
OCCShape patch1 = OCCUtils.makePatchThruSectionsP(
              noseTipVtx,
              sections1
              );

// Actually generate the supporting curves and add them to the ones to be exported
sections1.stream()
      .map(sec -> OCCUtils.theFactory
          .newCurve3D(
              sec.stream()
              .map(p -> new double[] {p.x, p.y, p.z})
              .collect(Collectors.toList()),
              false)
          )
      .map(crv -> (OCCEdge) ((OCCGeomCurve3D) crv).edge())
      .forEach(e -> extraShapes.add(e));
```

Listing 3.1 Nose cap shapes building process

The patch for the nose trunk is the second to be built. The procedure is almost identical to the one previously described. The main difference consists in the fact that in this case the user is given the possibility to actually choose which kind of spacing he wants to use for the supporting curves. The `spacingTypeNosePatch` variable is the one allowing this. This variable belongs to the `XSpacingType` enumeration class, which will be explained in its details in the following, and being an instance of a Java `enum` class it can only assume values in a close range of constant ones. Each of these constant values is associated to one of the possible spacings that can be used for a distribution of points, and each one also implements its own version of the `calculateSpacing` abstract method, which belongs to the enumeration class too. The following listing contains the code used to generate the loft for the nose trunk and the supporting curves. As previously said, the variables containing the word `bar` have been normalized with respect to the total length of the nose, while the ones with the word `mt` (standing for *meter*) are dimensional variables. Figure 3.2 shows the final result for the nose cap and the nose trunk shapes, highlighting the differences between the two lofts in terms of spacing and number of supporting sections.

```
// Determine a List containing the sections x-coordinates normalized with respect
// to the nose length. xbarNoseCap is the normalized nose cap length.
List<Double> xbars2 = Arrays.asList(
    spacingTypePatch2.calculateSpacing(
        noseCapSectionFactor2*xbarNoseCap, 1.0,
        numberNosePatch2Sections
    ));

// Generate the supporting curves for the nose
// trunk and make them available for the export
List<Double> xmtPatch2 = new ArrayList<>();
xbars2.forEach(x ->
    xmtPatch2.add(x*noseLength.doubleValue(SI.METER))
    );

```

```
List<CADGeomCurve3D> cadCurvesNoseTrunk = new ArrayList<>();
xmtPatch2.stream()
    .map(x -> Amount.valueOf(x, SI.METER))
    .forEach(x -> cadCurvesNoseTrunk.add(
        OCCUtils.theFactory
            .newCurve3DP(fuselage.getUniqueValuesYZSideRCurve(x), false)
        ));

cadCurvesNoseTrunk.stream()
    .map(c -> (OCCGeomCurve3D) c)
    .map(crv -> (OCCEdge) (crv.edge()))
    .forEach(e -> extraShapes.add(e));

// Patch through the nose trunk supporting sections in order to obtain a loft
OCCShape patch2 = OCCUtils.makePatchThruSections(cadCurvesNoseTrunk);
```

Listing 3.2 Nose trunk shapes building process

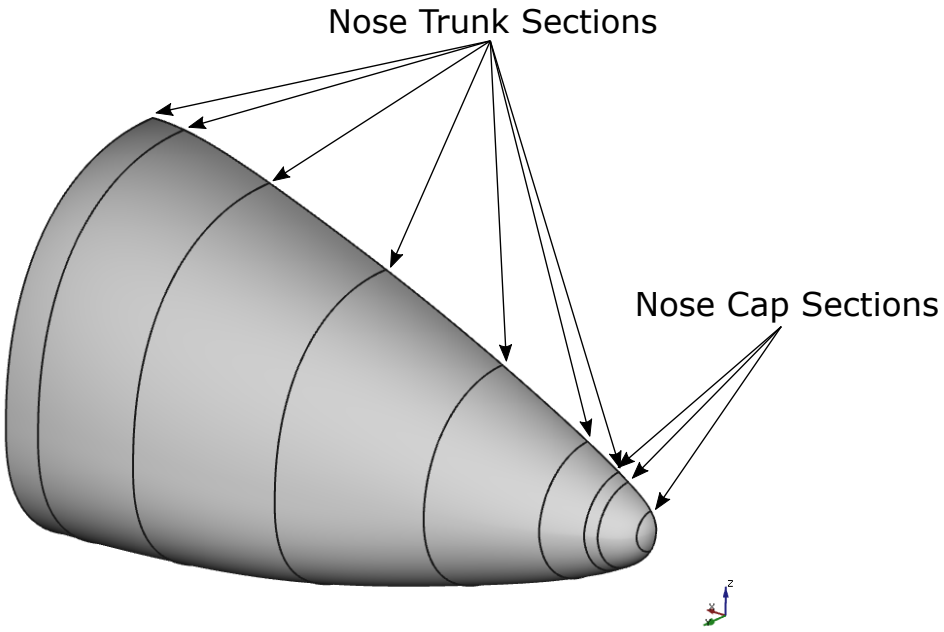

Figure 3.2 Nose cap and nose trunk patches with supporting sections

The next step consists in generating the loft for the cylindrical section of the fuselage. This one is the easiest to obtain, since no particular attention must be paid to the number of supporting sections and the spacing between them. In fact, the `getFuselageCAD` method doesn't even allow to choose the values for these parameters. Instead, the number of supporting sections is constantly set to three, with a linear spacing being used to separate them. The following lines of code describe the process through which cylindrical patch shapes have been built, with figure 3.3 showing the result.

```
// Determine the dimensional x-coordinates for the cylinder cross sections
List<Double> xmtPatch3 = Arrays.asList(
        MyArrayUtils.linspaceDouble(
            noseLength.doubleValue(SI.METER),
            noseLength.plus(cylinderLength).doubleValue(SI.METER),
            3)); // number of cross sections

// Generate supporting curves and make them available for export
List<CADGeomCurve3D> cadCrvCylinder = new ArrayList<>();
cadCrvCylinder.add(OCCUtils.theFactory.newCurve3DP(
    fuselage.getUniqueValuesYZSideRCurve(noseLength), false));
cadCrvCylinder.add(OCCUtils.theFactory.newCurve3DP(
    fuselage.getUniqueValuesYZSideRCurve(
        noseLength.plus(cylinderLength.times(0.5))),false));
cadCrvCylinder.add(OCCUtils.theFactory.newCurve3DP(
    fuselage.getUniqueValuesYZSideRCurve(
        noseLength.plus(cylinderLength)), false));
cadCrvCylinder.forEach(crv ->
    extraShapes.add((OCCEdge) ((OCCGeomCurve3D) crv).edge())) ;

// Generate the cylindrical loft by patching through the just created 3D curves.
OCCShape patch3 = OCCUtils.makePatchThruSections(cadCrvCylinder);
```

Listing 3.3 Fuselage cylinder building steps

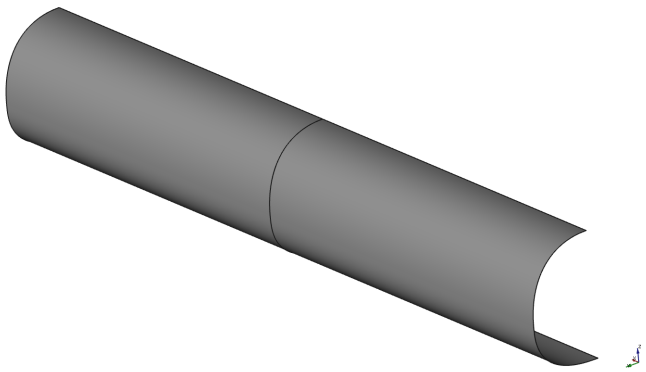

Figure 3.3 Fuselage cylindrical section

The generation of the tail shapes follows the same steps listed for the nose ones. The main difference consists in the different spacing adopted for the tail cap: in this case the half cosine spacing adopted forces an higher density towards the initial section of the list (so, in the negative x direction). For the sake of completeness, the listing 3.4 reports part of the actual code used to generate tail shapes (including supporting sections), while figure 3.4 shows the just generated shape once being imported into a CAD suite.

```
// Generate a list containing all tail trunk cross section x-coordinates
List<Double> xmtPatch4 = Arrays.asList(
        spacingTypeTailPatch.calculateSpacing(
            noseLength.plus(cylinderLength).doubleValue(SI.METER),
```

```
            fuselageLength.minus(
                tailCapLength.times(tailCapSectionFactor1)).doubleValue(SI.METER),
            numberTailPatchSections)
            );

// Generate a list containing tail cap cross section x-coordinates
List<Double> xmtPatch5 = Arrays.asList(
        MyArrayUtils.halfCosine1SpaceDouble(
            fuselageLength.minus(
                tailCapLength.times(tailCapSectionFactor1)).doubleValue(SI.METER),
            fuselageLength.minus(
                tailCapLength.times(tailCapSectionFactor2)).doubleValue(SI.METER),
            numberTailCapSections)
        );

// Generate tail trunk cross section curves and make them available for export.
List<CADGeomCurve3D> cadCurvesTailTrunk = new ArrayList<>();
xmtPatch4.stream()
    .map(x -> Amount.valueOf(x, SI.METER))
    .forEach(x -> cadCurvesTailTrunk.add(
        OCCUtils.theFactory
            .newCurve3DP(fuselage.getUniqueValuesYZSideRCurve(x), false)
        ));

cadCurvesTailTrunk.stream()
    .map(crv -> (OCCEdge) ((OCCGeomCurve3D) crv).edge())
    .forEach(e -> extraShapes.add(e));

// Generate tail cap cross section curves and make them available for export
CADVertex vertexTailTip = OCCUtils.theFactory.newVertex(
        fuselageLength.doubleValue(SI.METER), 0, zTailTip.doubleValue(SI.METER));

List<CADGeomCurve3D> cadCurvesTailCap = new ArrayList<>();
xmtPatch5.stream()
    .map(x -> Amount.valueOf(x, SI.METER))
    .forEach(x -> cadCurvesTailCap.add(
        OCCUtils.theFactory
            .newCurve3DP(fuselage.getUniqueValuesYZSideRCurve(x), false)
        ));

cadCurvesTailCap.stream()
    .map(crv -> (OCCEdge) ((OCCGeomCurve3D) crv).edge())
    .forEach(e -> extraShapes.add(e));

// Generate the patch for the tail trunk
OCCShape patch4 = OCCUtils.makePatchThruSections(cadCurvesTailTrunk);


// Generate the patch for the tail cap
OCCShape patch5 = OCCUtils.makePatchThruSections(
    cadCurvesTailCapTrunk, vertexTailTip);
```

Listing 3.4 Tail trunk and cap building process

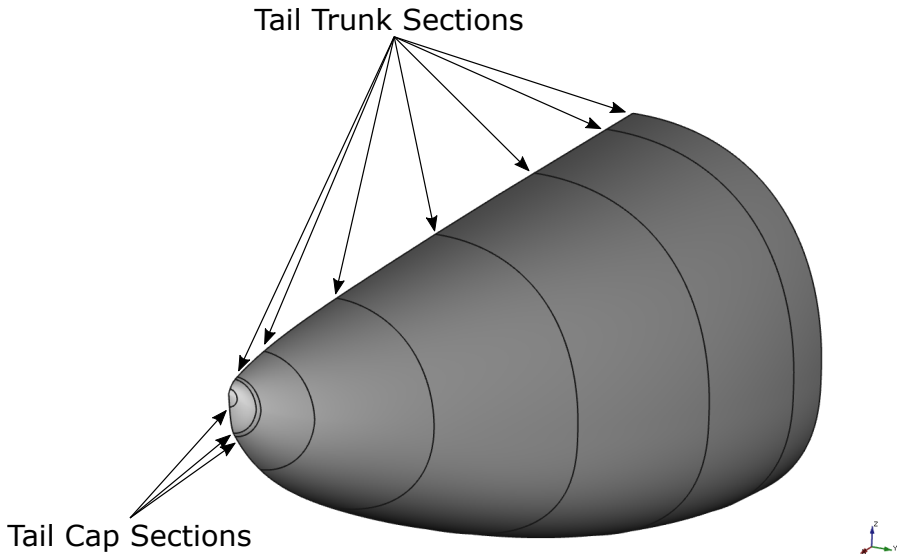

Figure 3.4 Tail cap and tail trunk patches with supporting sections

Once all the lofts have been created, they need to be sewed, in order to get one single shell. The OCCT library provides the class to perform this kind of operation: it is called `BRepBuilderAPI_Sewing` and provides methods in order to:

- create an empty builder,
- add elements (generic shapes) to the builder,
- set the tolerance,
- compute the operation,
- return the resulted shapes.

The `CADShapeFactory` actually contains methods in order to perform sewing operations, but starting from `CADFace` instances. Since the shape we need to sew are actually shells, and sewing methods managing shell-type elements still need to be implemented in `JPADCAD`, the actual code makes direct use of the OCCT class. This is not the only case (as it will be clear in the next paragraphs and chapters), since `JPADCAD` is still under constant development and testing.

```
1  // Generate a new instance of the sewmaker class
2  BRepBuilderAPI_Sewing sewMaker = new BRepBuilderAPI_Sewing();
3
4  // Initalize the sewmaker and provide it with the patches.
5  // Finally, perform the sewing operation.
6  sewMaker.Init();
7  sewMaker.Add(patch1.getShape());
8  sewMaker.Add(patch2.getShape());
9  sewMaker.Add(patch3.getShape());
10 sewMaker.Add(patch4.getShape());
11 sewMaker.Add(patch5.getShape());
```

```
12 sewMaker.Perform();
13
14 // Get the resulting TopoDS_Shape entity
15 TopoDS_Shape tds_shape = sewMaker.SewedShape();
```

Listing 3.5 Lofts sewing process

Once the sewing operation has been performed, the returned shape must be explored, in order to find shell-type entities. This operation is performed by means of the `CADShapeFactory` method `newExplorer`, which relies on the `OCCExplorer` class constructors and allows to generate some sort of *seeker* for desired shapes. This operation is fundamental, since the shape returned by the sewing algorithm belongs to the abstract OCCT `TopoDS_Shape` class.

The next operation consists in mirroring the fuselage right shell obtained at the previous step. As mentioned in Chapter 2, the OCCT library offers the possibility to perform basic geometry transformations. The class supervising this type of operation is called `gp_Trsf`, which actually allows to perform:

- translation,
- rotation,
- scale,
- reflection with respect to a point, a line, a plane.

The type of the operation can be choosed by means of several `set` methods, that act directly on an empty instance of the `gp_Trsf` class. In our case, a plane (the plane of symmetry of the fuselage) must be defined in order to perform the mirroring operation. The OCCT class `gp_Ax2` helps defining a plane by means of a point (the origin, expressed in terms of `gp_Pnt`), and two directions (both expressend in terms of instances of the OCCT class `gp_Dir`), identifying the normal to the plane and a direction contained in the same plane. The following listing shows how the operation has been coded.

```
1  // Define a new instance of the transformation class
2  gp_Trsf mirrorTransform = new gp_Trsf();
3
4  // Define the plane with respect to the shapes must be mirrored
5  gp_Ax2 mirrorPointPlane = new gp_Ax2(
6      new gp_Pnt(0.0, 0.0, 0.0), // Origin
7      new gp_Dir(0.0, 1.0, 0.0), // Y direction
8      new gp_Dir(1.0, 0.0, 0.0)  // X direction
9      );
10
11 // Select the the transformation to perform, add to it the just
12 // created plane and generate a new instance of the actual OCCT
13 // class executing transformation on shape geometries
14 mirrorTransform.SetMirror(mirrorPointPlane);
15 BRepBuilderAPI_Transform mirrorBuilder =
16     new BRepBuilderAPI_Transform(mirrorTransform);
```

```
17
18 // Finally perform the mirroring operation
19 mirrorBuilder.Perform(sewedShell.getShape(), 1);
20 TopoDS_Shape mirroredShape = mirrorBuilder.Shape();
```

Listing 3.6 Fuselage right shell mirroring operations

In this case too it is necessary to go through the shapes produced by the mirroring operations in order to find shell-type entities. The operation performed is the same described above. Once the shells for the right and left side of the fuselage have been obtained, another sewing operation can be executed, returning one single shell, which will be the one to be used in order to create a solid entity. Altough the `CADShapeFactory` contains methods generating solids, the ones currently implemented do not allow the production starting from shells. For this reason, at present, the code performs the creation of solid entities by using low level OCCT classes. In particular, the class that implements this type of operation is called `BRepBuilderAPI_MakeSolid` which provides methods for:

- defining and implementing the construction of a solid,
- consulting the results.

Once the shell has been added to an empty instance of the solid builder class, the user just needs to use the `Build` method in order to perform the construction. Figure 3.5 shows different views of the same solid fuselage, with the one depicted belonging to the ATR-72.

```
1  // Generate a CADSolid empty instance
2  CADSolid solidFuselage = null;
3
4  // Generate a new solidmaker empty object
5  BRepBuilderAPI_MakeSolid solidMaker = new BRepBuilderAPI_MakeSolid();
6
7  // Add the sewed halves of the fuselage to the solidmaker
8  // and perform the building operation
9  solidMaker.Add(TopoDS.ToShell(sewedShells.getShape()));
10 solidMaker.Build();
11
12 // Finally generate the fuselage CADSolid object by means of the
13 // OCCUtils newShape method, which accept a generic object of
14 // the underlying implementation to build CAD-type ones
15 if(solidMaker.IsDone() == 1) {
16     solidFuselage = (CADSolid) OCCUtils.theFactory.newShape(solidMaker.Solid());
17 }
```

Listing 3.7 Fuselage solid building step

The `getFuselageCAD` method implemented in `AircraftUtils` allows not only to export the final solid shape of the fuselage. It also gives the user the possibility to choose between which types of shape he wants to export and write to file. For example, by setting to `true` the `exportSupportShapes` parameter and `false` the remaining ones, just the wireframe of the fuselage

would be exported. For sake of completeness, figure 3.6 shows the wireframe (completed by the outline curves, whose calculation is performed by the code but not actually used for loft generation purposes) of the fuselage of the ATR-72.

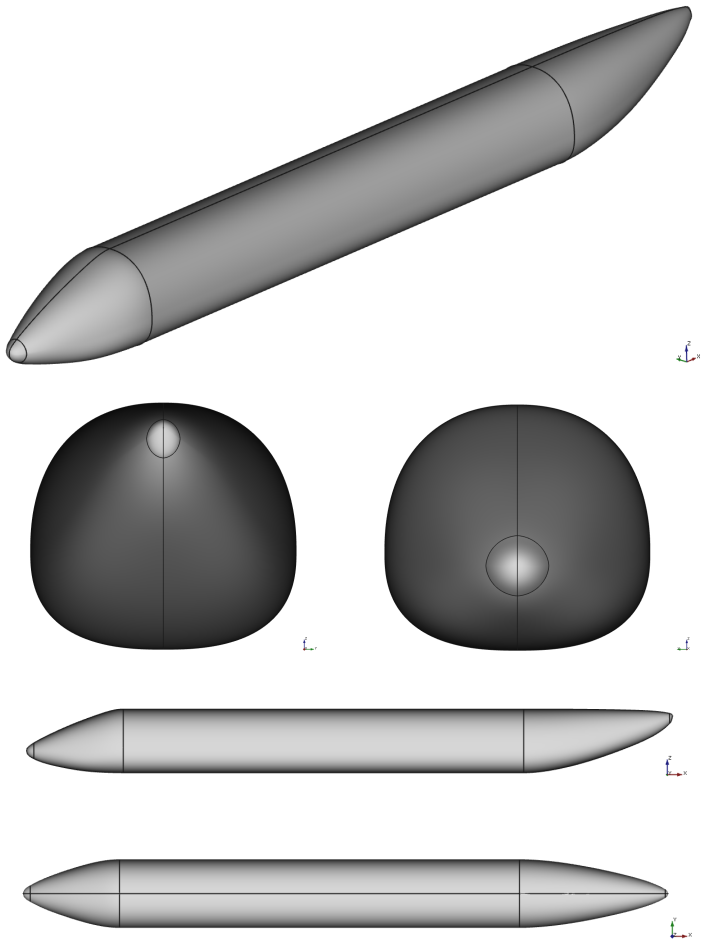

Figure 3.5 ATR-72 solid fuselage, different views

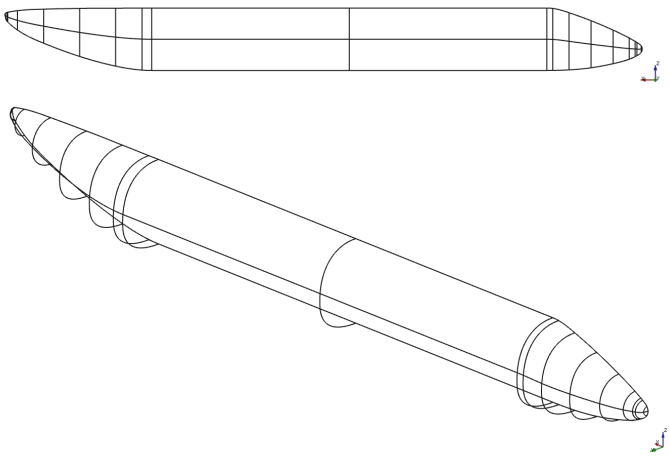

Figure 3.6 ATR-72 fuselage wireframe, different views

3.4 Lifting surface CAD method

The construction of the shapes representing a generic lifting surface revolves around three main stages, tipically:

1. generate one loft for each panel of the wing;
2. generate a shell for the tip of the wing;
3. reflect the resulting shapes with respect to a symmetry plane (whether necessary).

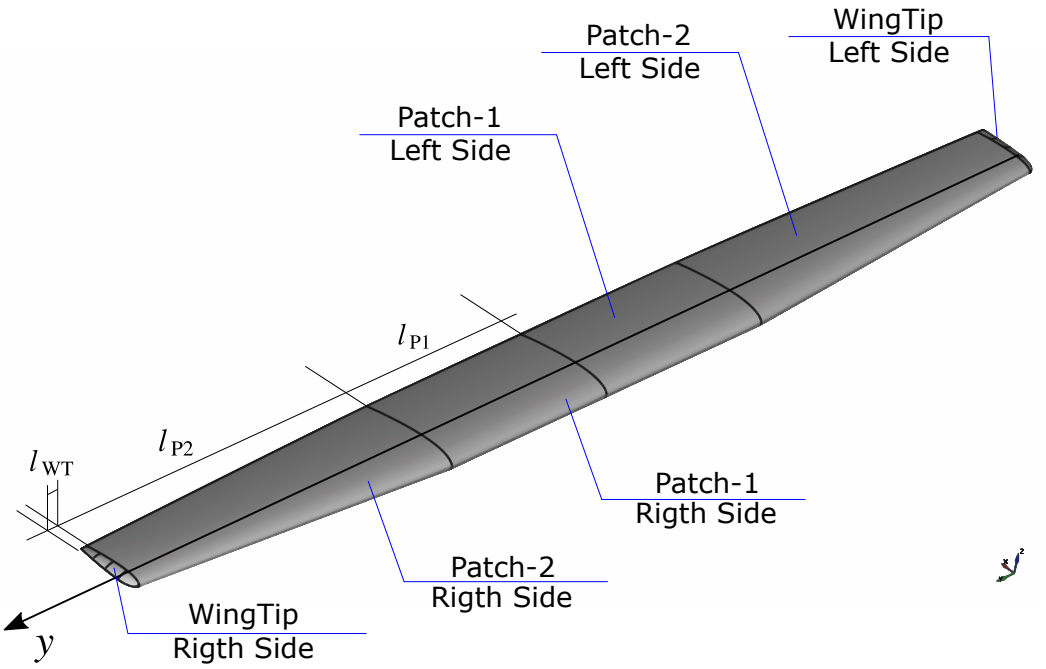

Figure 3.7 Wing patches for a JPAD wing

The `AircraftUtils` method that provides for the construction of lifting surfaces is called `getLiftingSurfaceCAD`. This method accepts the following inputs.

- **`liftingSurface`** - An instance of the JPAD `LiftingSurface` class, that is used in order to describe and collect geometric, aerodynamic and mass characteristics of a generic lifting surface. It provides also the methods to access all these information.
- **`typeLS`** - An instance of the JPAD `ComponentEnum` class, which provides support for aircraft components enumeration. The `getLiftingSurfaceCAD` algorithm makes use of this argument in order to distinguish between different typologies of lifting surfaces. It is necessary for the code to know with which kind of surface it is dealing with, in order to perform the necessary checks and to activate the right flags for certain types of operations.
- **`tipTolerance`** - A `double` entry fixing the tolerance for the process of sewing the tip of the lifting surface (which is modeled separately) with the rest of the wing.

- **exportLofts, exportSolid, exportSupportShapes** - Entries of the `boolean` type, that allow to export, respectively, the shells (wing and tip), the solid, and the wireframe of the lifting surface.

First things the algorithm executes regard data collection (by means of the methods provided by the `LiftingSurface` class) and the wing wireframe construction. One of the datum that must be collected is the one related to the number of panels that constitute the wing. This number is tipically equal to two for proper wings, one for the other lifting surfaces such as the tail ones and the canard. Once all the necessary data has been collected, the construction of the wireframe of the wing can begin. For this purpose, leading edge, trailing edge, and airfoil curves must be created. It has to be noted that, as for the fuselage construction, some of the wireframe shapes generated are not intended to be used for lofting operations. Leading and trailing edge segments, for example, just serve for graphical purposes, and are not actually used as guide curves when performing patching. Furthermore, being generated by means of just two points (obtained by wing breakpoints coordinates), these edges are just segments and do not follow the actual profile of the curves they are meant to represent. On the contrary, airfoil curves are actually used to generate lofts, by patching through them using the same algorithms described in the previous paragraph and in Chapter 2. The `liftingSurface` object passed to the method allows access to the list of airfoils located at the wing breakpoints. This list contains objects of the `Airfoil` type, which is the class that in JPAD manages aerodynamic and geometric characteristics of airfoils, and provides methods that give access to the x and z coordinates of its points. Since these points just define the *absolute* airfoil (the airfoil with its chord aligned with the x axis and length equal to 1), they need to be modified in order to correctly represent the actual airfoils at some precise locations along the lifting surface. The `AircraftUtils` class has then been provided with a private method called `populateCoordinateList`. This method receives:

- a `double` representing the y station at which the airfoil points must be calculated;
- an object of the `Airfoil` class, which provides the basic airfoil points;
- an object of the `LiftingSurface` type, that provides wing characteristics (in terms of chord length, twist angle, and leading edge coordinates) at precise locations along its span;

and returns a List of `double[]` points, representing the actual coordinates of the airfoil. This method also performs several checks on the points it is supplied to. A first check is made on the points imported from the `Airfoil` object by controling there are no duplicates among the list, in order to avoid problems once the spline interpolation needs to be performed. Another check is made on the trailing edge of the airfoil, which has to be opened in case the z coordinates of the first and last points of the list coincide. This is a necessary operation, especially when the import of the CAD model of the wing into a CFD suite is something planned. Currently the method does not allow to adjust the value of the gap between the points: the code automatically opens the trailing edge if the distance between the first and the last point of the basic airfoil is less than 10^{-5}m in the z direction, setting it to 10^{-4}m.

```
if(Math.abs(zCoords[0] - zCoords[nPoints - 1]) < 1e-5) {
      zCoords[0] += 5e-4;
      zCoords[nPoints - 1] -= 5e-4;
    }
```

Listing 3.8 Opening creation at the trailing edge

Once the points have been obtained, they can be interpolated by means of one of the factory methods for 3D curves seen in Chapter 2. These methods interpolate a list of points by using a NURBS curve and provide a `CADGeomCurve3D` object representing the curve itself. Figure 3.8 shows an example of an airfoil obtained by applying the aforementioned methods. In particular, the figure depicts a NACA 23018 airfoil.

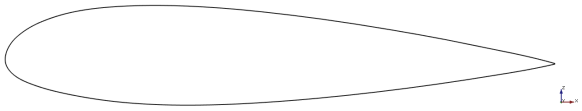

Figure 3.8 NACA 23018 airfoil curve generated in JPAD

The method described above allows to build airfoils at every desired location along the wing span. A first group of airfoils is generated at the breakpoints, while secondary groups are created between them. The reason behind having multiple airfoils between breakpoint ones resides in the necessity to create wing lofts, one for each panel, by patching through more than just two airfoil curves at a time, in order to reduce the risk of having unexpected results and to increase final shapes adherence to real counterparts. Since the number of panels is not something that can be known *a priori* when applying CAD generating methods for lifting surfaces, the user is not currently enabled to fix the number of cross sections for each patch. However, the code automatically sets it, based on the length of each panel respet to the total wing span. At the end of this process, a list of lists of `CADGeomCurve3D` is filled, with each of the sub-lists associated to a specific panel. Figure 3.9 shows the wireframe created to this point, with breakpoints airfoils also provided with their chord segment.

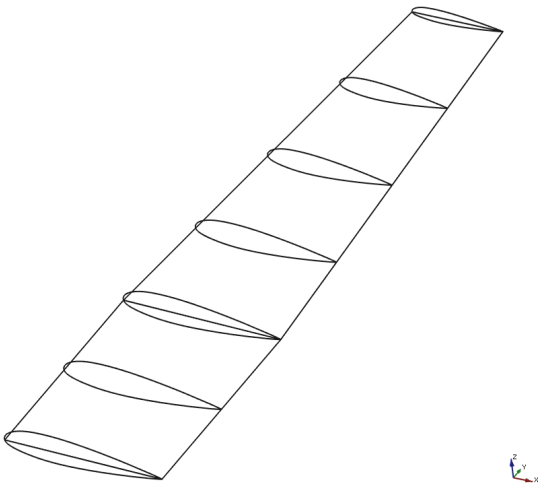

Figure 3.9 Wing wireframe, with no sketching curves for the tip

As mentioned before, the `getLiftingSurfaceCAD` method also generates shapes for the wing tip. These shapes are created by following the same steps used for the construction of fuselage and wing patches. The main difference in this case consists in the fact that the curves to patch through needs to be created from scratch, since JPAD `LiftingSurface` class does not contain a description for the tip. The very first operation consists in splitting the tip and penultimate airfoils (i.e., the last and the second-to-last of the airfoils depicted in figure 3.9) in two halves at their leading edge, by means of the wing wireframe generated at the previous step. These operations are necessary for two reasons:

- allow to better shape the leading edge of the wing tip by splitting its filling process in two steps, one regarding the upper side and one regarding the lower one;
- the algorithm that has been implemented in order to generate the cross sections for the wing tip necessitates both the upper and the lower side curves of the tip and pre-tip (penultimate) airfoils.

Besides, since it is quite likely that the leading edge of the tip airfoil does not precisely pass through the points of the leading edge of the wing wireframe, a further manipulation is necessary. This operation consists in forcing the two halves of the tip airfoil to pass through the last of the points describing the leading edge wireframe, and, in order to make sure the two halves preserve their original shape as much as possible, a much larger number of points is used to discretize them. This operation is quite necessary, since in case the condition described above is not satisfied the filling operations for the wing tip leading edge could encounter some problematics while are executed.

In order to build the necessary supporting curves for the tip shell, some construction must be performed first. For one thing, some sort of supporting plane is built, by simply extending the leading and trailing edge segments of the wing wireframe in the positive y direction. The amount of this extension is related to the thickness of the tip airfoil. This extension generates the points A and B (figure 3.12), and the segments a, b, and c, that define the actual construction plane for the wing tip. These operations are performed by means of `PVector` objects, that prove to be really useful especially when managing directions and point definitions. In order to obtain an authentic plane at the previous step, point B z coordinate is slightly changed and adjusted (whether the lifting surface being built is not a vertical tail) in order to make it coincide with the one of point A. Once the construction plane has been obtained, it is possible to build the first of the supporting curves for the tip. The points for this curve are obtained by using the `PVector` class too, which provides static methods to perform linear interpolation between one vector and another. In particular, the point C is the point at the 25% of the segment b (that has been oriented from A to B, so in the positive x direction), while the point D is at the 75% of the same segment. Point E is at the 75% of the segment connecting the leading and trailing edge of the tip airfoil, while F is at the 90% of the edge that connects the points D and E. Finally, point G is obtained as the point at the 25% of the segment c.

```
1 double[] mainVSecVector = {0.25, 0.75};
2 PVector cPnt = PVector.lerp(aPnt, bPnt, (float) mainVSecVector[0]);
```

```
PVector dPnt = PVector.lerp(aPnt, bPnt, (float) mainVSecVector[1]);
PVector ePnt = PVector.lerp(le2, te2, (float) mainVSecVector[1]);
PVector fPnt = PVector.lerp(ePnt, dPnt, 0.90f);
PVector gPnt = PVector.lerp(bPnt, te2, 0.25f);
```

Listing 3.9 Points for the in-plane tip construction curve

In order to shape the in-plane construction curve correctly, tangent vectors must be assigned. In particular, the curve is assigned three tangency constraints: one in A, one in C, and the last one in G. These tangent vectors are obtained by means of several `PVector` entities manipulations. Since the OCCT algorithm underlying the `newCurve3D` factory methods allow to assign tangency constraints only at the initial and final points of a curve, two in-plane construction curves are actually built, with the second one being split (at the point F) by means of the `OCCUtils.splitEdge` method, in order to facilitate the subsequent necessary manipulations (listing 3.10).

```
// Point lists for the in-plane construction curves
List<double[]> constrPlaneGuideCrv1Pnts = new ArrayList<>();
constrPlaneGuideCrv1Pnts.add(new double[] {le2.x, le2.y, le2.z});
constrPlaneGuideCrv1Pnts.add(new double[] {cPnt.x, cPnt.y, cPnt.z});

List<double[]> constrPlaneGuideCrv2Pnts = new ArrayList<>();
constrPlaneGuideCrv2Pnts.add(new double[] {cPnt.x, cPnt.y, cPnt.z});
constrPlaneGuideCrv2Pnts.add(new double[] {fPnt.x, fPnt.y, fPnt.z});
constrPlaneGuideCrv2Pnts.add(new double[] {gPnt.x, gPnt.y, gPnt.z});

// Generate tangent vectors
PVector leVector = PVector.sub(le2, le1); // Vector representing the LE segment
PVector aVector = PVector.mult(leVector, aLength);
PVector cVector = PVector.sub(cPnt, aPnt);
PVector gVector = PVector.sub(gPnt, fPnt);

// Weights for the tangency constraints
double tanAFac = 1;
double tanCFac = (cVector.mag()/aVector.mag())*0.75;
double tanGFac = tanCFac;

// Tangent vectors normalization
aVector.normalize();
cVector.normalize();
gVector.normalize();

// Actual tangent vectors for the construction curves.
double[] tanAConstrPlaneGuideCrv = MyArrayUtils.scaleArray(
    new double[] {aVector.x, aVector.y, aVector.z}, tanAFac);
double[] tanCConstrPlaneGuideCrv = MyArrayUtils.scaleArray(
    new double[] {cVector.x, cVector.y, cVector.z}, tanCFac);
double[] tanGConstrPlaneGuideCrv = MyArrayUtils.scaleArray(
    new double[] {gVector.x, gVector.y, gVector.z}, tanGFac);

// Points interpolation
```

```
CADGeomCurve3D constrPlaneGuideCrv1 = OCCUtils.theFactory.newCurve3D(
        constrPlaneGuideCrv1Pnts,
        false,
        tanAConstrPlaneGuideCrv,
        tanCConstrPlaneGuideCrv,
        false);
CADGeomCurve3D constrPlaneGuideCrv2_0 = OCCUtils.theFactory.newCurve3D(
        constrPlaneGuideCrv2Pnts,
        false,
        tanCConstrPlaneGuideCrv,
        tanGConstrPlaneGuideCrv,
        false);

// Split constrPlaneGuideCrv2_0 for further manipulations
List<OCCEdge> constrPlaneGuideCrvs2 = OCCUtils.splitEdge(
    constrPlaneGuideCrv2_0,
    new double[] {fPnt.x, fPnt.y, fPnt.z});

CADGeomCurve3D constrPlaneGuideCrv2 =
    OCCUtils.theFactory.newCurve3D(constrPlaneGuideCrvs2.get(0));
CADGeomCurve3D constrPlaneGuideCrv3 =
    OCCUtils.theFactory.newCurve3D(constrPlaneGuideCrvs2.get(1));
```

Listing 3.10 In-plane construction curves building steps

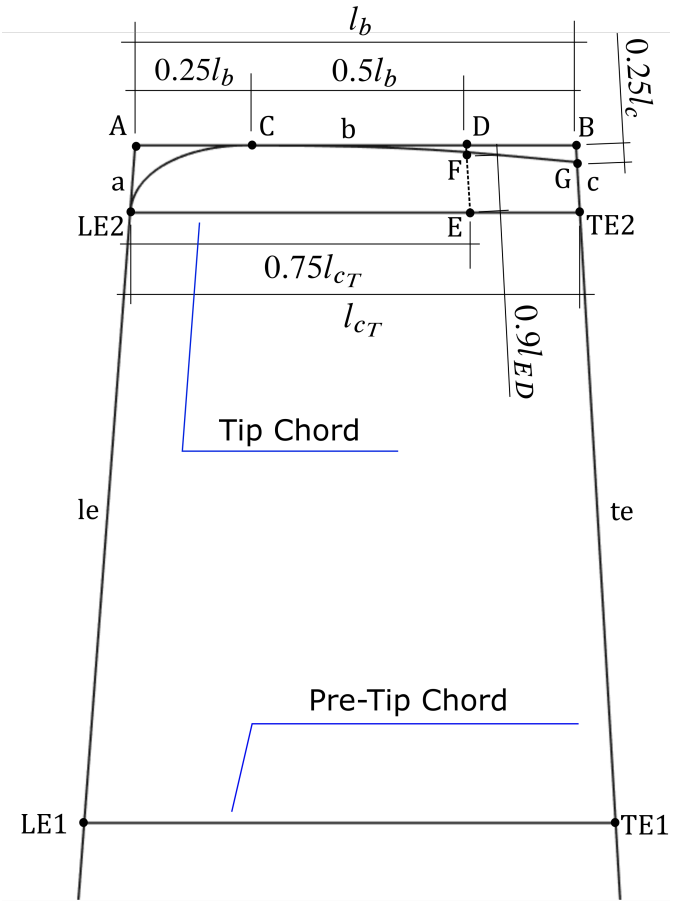

Figure 3.10 Wing tip construction plane, complete with point definitions and edge dimensions

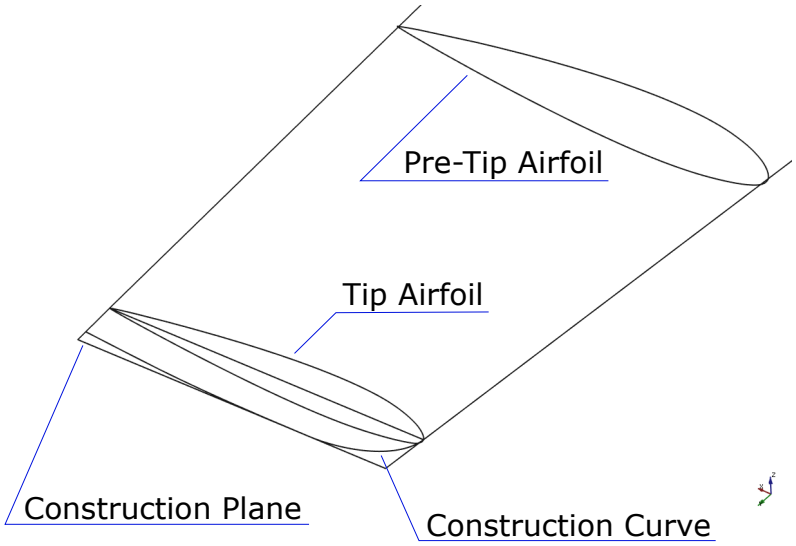

Figure 3.11 Wing tip in-plane construction curve

The in-plane construction curves serve as some sort of guide for the actual supporting curves of the wing tip, by providing points for their construction. First cross section curves to be generated are those passing for the points C, F, and G. `AircraftUtils` contains a private static method, called `createVerCrvsForTipClosure`, that generates cross section curves for the tip (in the form of `CADGeomCurve3D` objects) by means of the following arguments:

- an instance of the `LiftingSurface` class representing the lifting surface;
- two Java Lists of `OCCEdge` objects, containing respectively the upper and the lower edges of the tip and pre-tip airfoil;
- two `PVector` entities, representing the directions of the leading and trailing edge of the wing wireframe, respectively;
- a `double` entry, whose value is between 0.0 and 1.0, and states at which station, in terms of the tip chord fraction, the supporting curve needs to be created;
- 3D point coordinates in the form of a `double` array, representing the point on the in-plane construction curve that the supporting curve needs to pass through.

This algorithm uses the entries listed above in order to build curves that start on the upper side of the wing tip airfoil and end on the lower one, passing through the in-plane construction curve. In order to get the necessary points on the tip airfoil curves, the method makes use of the `PVector cross` static function, which allows to calculate the cross product between vectors. The cross product one is interested in in this case is the one between the vector representing the chord (or a fraction of it) of the wing tip airfoil and one of the vectors belonging to the construction plane (such as the leading and trailing edge vectors passed to the method), in order to generate segments belonging to the tip airfoil plane and perpendicular to its chord. These segments are then used to calculate points on the airfoil curves, once being multiplied by a scalar quantity representing the semi-airfoil (upper or lower side) height with respect

to its chord. These scalar quantities are provided by a private static method, still contained in `AircraftUtils` and called `getThicknessAtX`, which requires an object belonging to the JPAD `Airfoil` class (representing the basic airfoil) and a `double` value standing for the chord fraction at which the heights (both in the up and down direction) of the airfoil curve must be calculated. The segments, once scaled, finally provide the points on the airfoil curve, by means of the static method `pointProjectionOnCurve` contained in `OCCUtils`.

Points are not the only thing needed in order to generate the cross section curves of the wing tip. Three tangency constraints need to be imposed onto these curves: one at the starting point, on the upper side of the tip profile; the second one at the intersection with the in-plane curve; the last one at the ending point, on the lower side of the tip curve. These tangency constraints (the first and the last one in particular) are calculated still by means of the curves of the last and penultimate airfoils. In fact, in order to obtain a smooth transition from the last panel shell to the one of the wing tip, tangent vectors are calculated by simply connecting corresponding points (i.e., points located at the same chord fraction) onto the penultimate and last airfoil. Points on the pre-tip airfoil curve are calculated in the same way as described above. Regarding the second tangent vector, instead, it is simply imposed coincident with the normal vector of the wing tip construction plane. In order to obtain the right shape for the supporting curves, tangent vector magnitudes are adjusted according to some parameters, which depend on the tip airfoil thickness and on the distance between the in-plane construction curve and tip airfoil plane at the cross section at which the curve is being calculated. The piece of code below (listing 3.11) just gives a glimpse at the operations behind the search of the appropiate magnitudes for the tangent vectors. Once all these operations are accomplished, the algorithm finally performs the curves calculation. For the same reasons specified above, the code produces two supporting curves at a time, that are collected into a `CADGeomCurve3D` array and returned to the calling method. Figure 3.12 shows the result obtained at this stage.

```
1  // Determine tip airfoil thicknesses with respect to its chord segment
2  double thickUpp = PVector.sub(pntOnTipAirfoilUCrv, pntOnTipChord).mag();
3  double thickLow = PVector.sub(pntOnTipAirfoilLCrv, pntOnTipChord).mag();
4
5  // Determine construction curve height with respect to
6  // the tip airfoil chord segment
7  double crvHeight = PVector.sub(
8      new PVector(
9          (float) guideCrvPnt[0],
10         (float) guideCrvPnt[1],
11         (float) guideCrvPnt[2]),
12     pntOnTipChord).mag();
13
14 // Determine tangent magnitudes, both for the upper and lower
15 // supporting curves, based on the just calculated values
16 double tanUppVSecCrvFac = 1;
17 double tanLowVSecCrvFac = 1*(-1);
18 double tanUppHalfVSecCrvFac = Math.pow(thickUpp/crvHeight, 0.60)*(-1);
19 double tanLowHalfVSecCrvFac = Math.pow(thickLow/crvHeight, 0.60)*(-1);
```

Listing 3.11 Tip supporting curves tangent vectors magnitudes

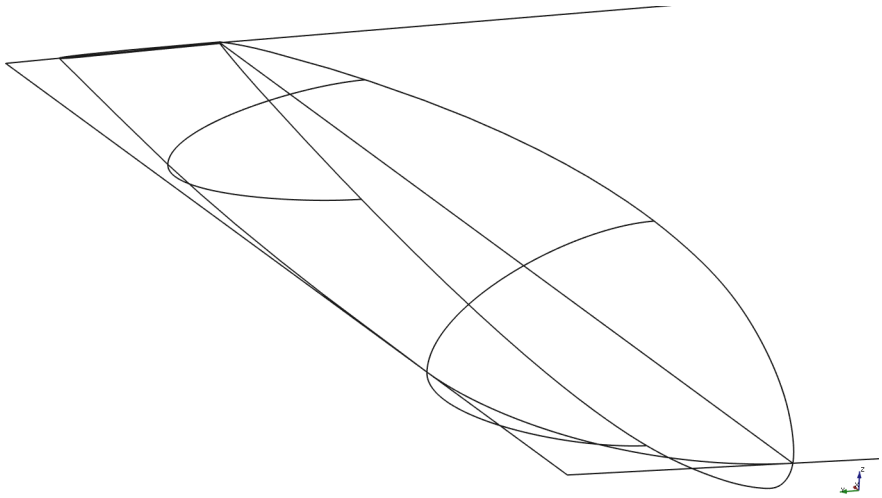

Figure 3.12 Wing tip wireframe main supporting curves

For the main supporting curves, namely those crossing the tip airfoil at the 25%, 75% and 100% in terms of its chord fraction, it is not necessary to calculate anything for the last entry of the `createVerCrvsForTipClosure` method, since those points are the ones that have been used to determine the shape of the in-plane construction curve. When it comes to supporting curves between the main ones (which are necessary in order to make sure the tip patching process is successfully accomplished) some calculation is needed instead. First thing, spacings between these curves are established by means of three arrays (reported in listing 3.13), one for each of the sections the wing tip has been split into by the previous step. These spacings are used directly to determine points on the in-plane construction curves by employing their parametric definition and their range. But once these points have been successfully obtained, what it is necessary to do, in order to use the same method described above in order to generate cross section curves, is determine their position in terms of wing tip chord fraction. This operation is quite simple to accomplish and involves just linear interpolation (provided by the JPAD utility class `MyMathUtils`) and some basic geometry consideration, and is reported in listing 3.13. Once the necessary chord fractions have been obtained, the `createVerCrvsForTipClosure` method can be used, and the resulting cross section curves are arranged in three different arrays, in order to be used later in the code. The `double` arrays defining these sections follow a peculiar spacing, especially for the first one. These spacings have been chosen after performing several tests, while searching for the best results in terms of returned shapes. Figure 3.13 shows the final result for the wing tip wireframe.

```
1  // Sub cross sections arrays
2  double[] subVSecP1Vector = {0.10, 0.15, 0.40, 0.55, 0.60, 0.75, 0.90};
3  double[] subVSecP2Vector = {0.25, 0.50, 0.75};
4  double[] subVSecP3Vector = {0.25, 0.50, 0.75};
5
6  List<double[]> subVSecVector = new ArrayList<>();
7
8  subVSecVector.add(subVSecP1Vector);
9  subVSecVector.add(subVSecP2Vector);
10 subVSecVector.add(subVSecP3Vector);
```

```
11
12  List<CADGeomCurve3D[]> subVSecP1 = new ArrayList<>();
13  List<CADGeomCurve3D[]> subVSecP2 = new ArrayList<>();
14  List<CADGeomCurve3D[]> subVSecP3 = new ArrayList<>();
15  List<List<CADGeomCurve3D[]>> subVSec = new ArrayList<List<CADGeomCurve3D[]>>();
16
17  // Iterate through the wing tip sections, in order
18  // to generate additional  supporting curves
19  for(int i = 0; i < 3; i++) {
20    int idx = i;
21    subVSec.add(Arrays
22      .stream(subVSecVector.get(i))
23      .mapToObj(f -> {
24        double[] crvRange = constrPlaneGuideCrvs.get(idx).getRange();
25        double[] pntOnGuideCurve = constrPlaneGuideCrvs.get(idx)
26                .value(f*(crvRange[1]-crvRange[0])+crvRange[0]);
27        double interpCoord;
28        double chordFraction;
29        double x = pntOnGuideCurve[0];
30
31        // Determine the corresponding chord fraction for the just
32        // calculated point lying on the construction curve
33        if(!typeLS.equals(ComponentEnum.VERTICAL_TAIL)) {
34          interpCoord = pntOnGuideCurve[1];
35          double xLE = MyMathUtils.getInterpolatedValue1DLinear(
36            new double[] {le2.y, aPnt.y},
37            new double[] {le2.x, aPnt.x},
38            interpCoord
39                );
40              double xTE = MyMathUtils.getInterpolatedValue1DLinear(
41                new double[] {te2.y, bPnt.y},
42                new double[] {te2.x, bPnt.x},
43                interpCoord
44                );
45              chordFraction = (x - xLE)/(xTE - xLE);
46        } else {
47              interpCoord = pntOnGuideCurve[2];
48              double xLE = MyMathUtils.getInterpolatedValue1DLinear(
49                new double[] {le2.z, aPnt.z},
50                new double[] {le2.x, aPnt.x},
51                interpCoord
52                );
53              double xTE = MyMathUtils.getInterpolatedValue1DLinear(
54                new double[] {te2.z, bPnt.z},
55                new double[] {te2.x, bPnt.x},
56                interpCoord
57                );
58              chordFraction = (x - xLE)/(xTE - xLE);
59        }
60
61        // Generate the supporting curve
62        CADGeomCurve3D[] subVSecCrvs = createVerCrvsForTipClosure(
63              liftingSurface,
```

```
64              airfoilTipCrvs,
65              airfoilPreTipCrvs,
66              new PVector[] {le1, le2},
67              new PVector[] {te1, te2},
68              chordFraction,
69              pntOnGuideCurve
70              );
71          return subVSecCrvs;
72      })
73      .collect(Collectors.toList())
74      );
75  }
76
77  // Collect the additional supporting curves in three different arrays
78  subVSecP1.addAll(subVSec.get(0));
79  subVSecP2.addAll(subVSec.get(1));
80  subVSecP3.addAll(subVSec.get(2));
```

Listing 3.12 Supporting curves between main wing tip cross sections

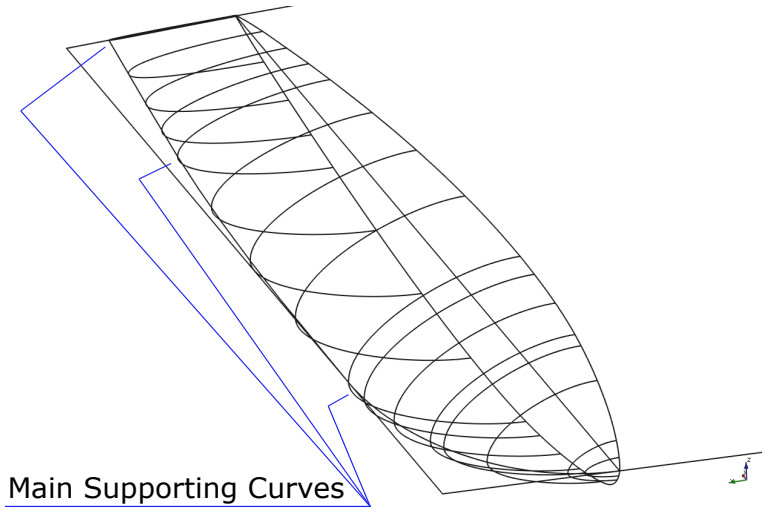

Figure 3.13 Wing tip wireframe supporting curves

Once the definitive wireframe has been built, it is finally possible to generate wing patches. The first ones to be created are those for the wing panels. The procedure is quite similar to the one used for the fuselage. The aforementioned list of lists, called `cadCurveAirfoilList` and containing `CADGeomCurve3D` entities representing airfoil curves, is used along with the `makePatchThruSections` method in order to generate a loft for each panel. The piece of code below briefly explains the procedure, while figure 3.14 depicts the results.

```
1  List<OCCShape> patchWing = new ArrayList<>();
2  patchWing.addAll(cadCurveAirfoilList.stream()
3                          .map(OCCUtils::makePatchThruSections)
4                          .collect(Collectors.toList()));
```

Listing 3.13 Spacings for supporting curves between main wing tip cross sections

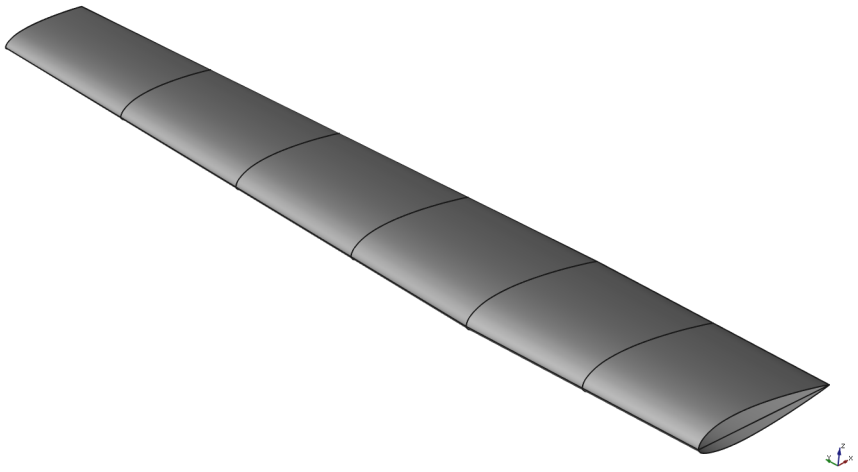

Figure 3.14 Main wing patches

Then it's the turn for the wing tip. The patching process can be divided in four stages, with the first one regarding the forward part of the wing tip. In order to obtain a smoother surface than the one that would be returned by simply using the `makePatchThruSections` algorithm (the one involving curves and vertices), it has been decided to employ a different method and some different OCCT classes. In particular, the choice has fallen on the `BRepOffsetAPI_MakeFilling` class, which allows to generate filling surfaces by providing a set of curves bounding the face we want to generate, and a set of points defining some constraints the support face has to satisfy. In order to provide the bounding edges the algorithm requires, it has been necessary to split the first of the in-plane construction curves and the upper and lower curves defining the wing tip airfoil at the extremities of the second pair of supporting curves of the wing tip wireframe first section. This task has been accomplished by means of the `OCCUtils splitEdge` method. Then several points on the first of the supporting curves have been used as constraints for the final surface. In order to obtain the best result possible, two filling surfaces have actually been created, one for the upper side and one for the lower one. The following listing shows how the procedure has been coded, while figure 3.15 shows the final result.

```
1  // Splitting the tip airfoil curves and the construction
2  // curve #1 in order to fill the wing tip LE correctly
3  List<OCCEdge> airfoilUpperCrvs = new ArrayList<>();
4  airfoilUpperCrvs.addAll(OCCUtils.splitEdge(
5      OCCUtils.theFactory.newCurve3D(airfoilTipCrvs.get(0)),
6      subVSecP1.get(1)[0].edge().vertices()[0].pnt()
7      ));
8
9  List<OCCEdge> constPlaneGuideCrvs1 = new ArrayList<>();
10 constPlaneGuideCrvs1.addAll(OCCUtils.splitEdge(
11     constrPlaneGuideCrv1,
12     subVSecP1.get(1)[0].edge().vertices()[1].pnt()
13     ));
14
15 // Creating a filler surface at the wing tip leading edge, upper side.
16 // The procedure for the lower side pretty identical.
```

```
17 double[] contrCrvUppRng = subVSecP1.get(0)[0].getRange();
18 double[] contrPntUpp1 = subVSecP1.get(0)[0].value(
19     0.25*(contrCrvUppRng[1] - contrCrvUppRng[0]) + contrCrvUppRng[0]);
20 double[] contrPntUpp2 = subVSecP1.get(0)[0].value(
21     0.50*(contrCrvUppRng[1] - contrCrvUppRng[0]) + contrCrvUppRng[0]);
22 double[] contrPntUpp3 = subVSecP1.get(0)[0].value(
23     0.75*(contrCrvUppRng[1] - contrCrvUppRng[0]) + contrCrvUppRng[0]);
24
25 BRepOffsetAPI_MakeFilling fillerP1Upp = new BRepOffsetAPI_MakeFilling();
26
27 fillerP1Upp.Add(
28     airfoilUpperCrvs.get(1).getShape(),
29     GeomAbs_Shape.GeomAbs_C0);
30 fillerP1Upp.Add(
31     ((OCCEdge)((OCCGeomCurve3D)subVSecP1.get(1)[0]).edge()).getShape(),
32     GeomAbs_Shape.GeomAbs_C0);
33 fillerP1Upp.Add(
34     constPlaneGuideCrvs1.get(0).getShape(),
35     GeomAbs_Shape.GeomAbs_C0);
36
37 fillerP1Upp.Add(new gp_Pnt(contrPntUpp1[0], contrPntUpp1[1], contrPntUpp1[2]));
38 fillerP1Upp.Add(new gp_Pnt(contrPntUpp2[0], contrPntUpp2[1], contrPntUpp2[2]));
39 fillerP1Upp.Add(new gp_Pnt(contrPntUpp3[0], contrPntUpp3[1], contrPntUpp3[2]));
40
41 fillerP1Upp.Build();
```

Listing 3.14 Wing tip leading edge filling code, upper side

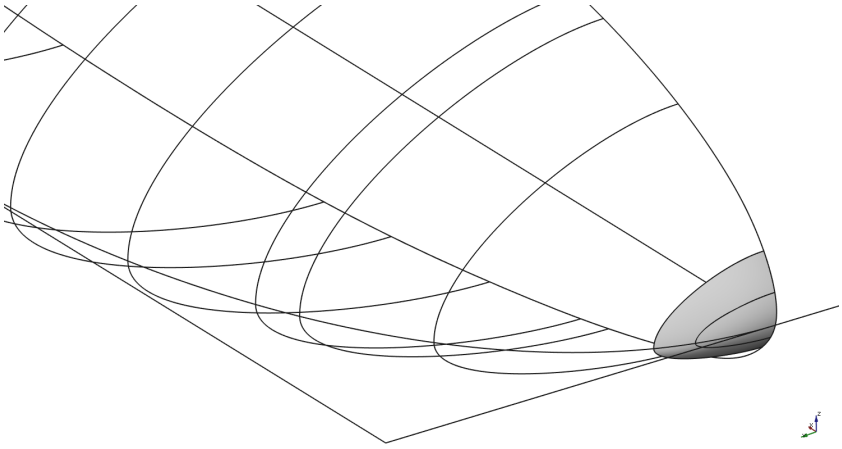

Figure 3.15 Wing tip leading edge filling surface

The remaining sections of the wing tip are completed by means of the `makePatchThruSections` algorithm. It has to be said that the second section (the one going from point C to point F, to be clear), has not been obtained in a single step (i.e., by patching through all the cross sections at the same time), but splitting the patching in two, in order to obtain the best result. This is the reason why, in the figures in which the final shell and solid shapes are depicted, the wing tip shell presents five patches rather than four, including the one obtained by the use of the filling algorithm. Figure 3.16 shows all the patches along with the supporting curves.

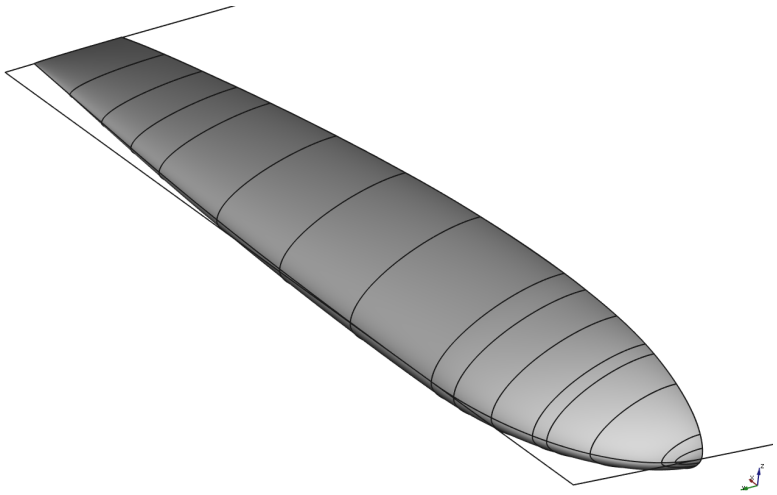

Figure 3.16 Wing tip patches and supporting curves

Before sewing all the patches together in order to obtain one single shell, another step is needed. In fact, the aforementioned operation performed by the `populateCoordinateList` method, consisting in opening the trailing edge of the airfoil whether necessary, has left some sort of hole on the back of the wing that must be filled. This operation has to be performed both for the panel patches and the wing tip, and the method that allows to do this is one of those listed in Chapter 2: the `OCCUtils` static method `makeFilledFace`. The OCCT algorithm behind this method is provided by the same class used above, in order to fill the leading edge of the wing tip: `BRepOffsetAPI_MakeFilling`. However, the way the algorithm is implemented in `OCCUtils` allows just to determine the flat surface bounded by the curves the method is provided with. So, the method accepts an array of indefinite length of `CADGeomCurve3D` entities, and returns the surface between the curves (which necessaryly must provide a closed wire). Figure 3.17 shows the filling surface for the trailing edge of the wing tip.

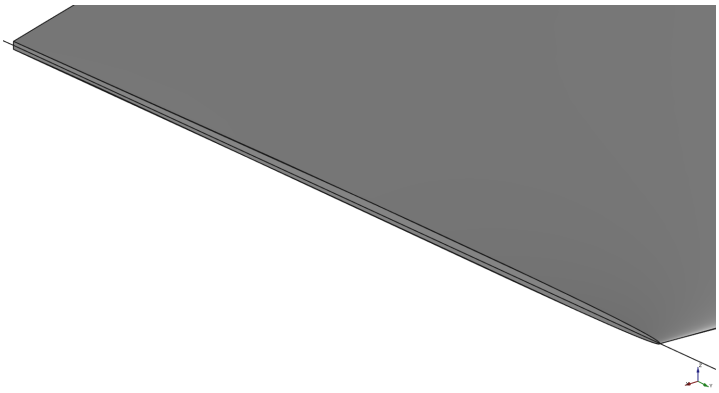

Figure 3.17 Wing tip trailing edge filling surface

The next steps consist in sewing all the patches and the faces created till now. First, wing panel patches are sewed, along with the trailing edge filling surfaces mentioned above. Then it is necessary to sew together the wing tip patches. This operation is much more delicate than the preceding one, since in this case the curves that define the borders of the patches we want to sew do not exactly coincide. This is due to the use of the edge splitting algorithm,

which sligthly modify the returning curves respect to the original one. In order to overcome this problem, the default tolerance of the sewing algorithm has been changed, and set to the value which is provided as input to the `getLiftingSurfaceCAD` method. For the same reasons, the default tolerance for the sewing algorithm needs to be set to a different (higher) value when sewing the wing tip shell with the one obtained by sewing the panel lofts. In this case, in particular, the difference between the two shells is much more evident, due to the fact that they have been obtained by patching through sections following different (orthogonal) directions.

The following operations are pretty much the same conducted for the fuselage solid construction. The main difference consists in the fact that mirroring operations are not necessary in case the lifting surface is a vertical tail. In fact, in this case, the only necessary operation consists in closing the side of the shell which is still open (i.e., the side opposite to the one where the wing tip has been built). The following piece of code (listing 3.15) shows how this operation has been implemented in the `getLiftingSurfaceCAD` algorithm. Again, the method this operation makes use of is the `OCCUtils` static function `makeFilledFace`.

```
1  // Check whether the lifting surface is a vertical tail
2  if(typeLS.equals(ComponentEnum.VERTICAL_TAIL)) {
3      CADShape faceRoot = OCCUtils.makeFilledFace(
4
5          // Get the first of the airfoil curves
6          cadCurveAirfoilList.get(0).get(0),
7
8          // Create a segment representing the trailing
9          // edge of the first of the airfoil curves
10         OCCUtils.theFactory.newCurve3D(
11             cadCurveAirfoilList.get(0).get(0).edge().vertices()[0].pnt(),
12             cadCurveAirfoilList.get(0).get(0).edge().vertices()[1].pnt()
13             )
14         );
15
16     // Add this shape to the ones to be sewed
17     sewMakerWing.Add(((OCCShape) faceRoot).getShape());
18 }
```

Listing 3.15 Vertical tail filling operations

In case the lifting surface is not a vertical tail, mirroring, sewing, and solid generating operations are performed by means of the same methods and classes listed above, for the fuselage CAD problem. In this case too, the method allows to export just the desired shapes: by setting the previously mentioned `boolean` arguments, the user can decide which types of shapes he wants to extract. Figure 3.18 shows the final result in terms of solid shapes. In particular, the depicted wing is the one of the ATR-72. The following paragraphs will also provide more examples of lifting surfaces produced by means of the same algorithm.

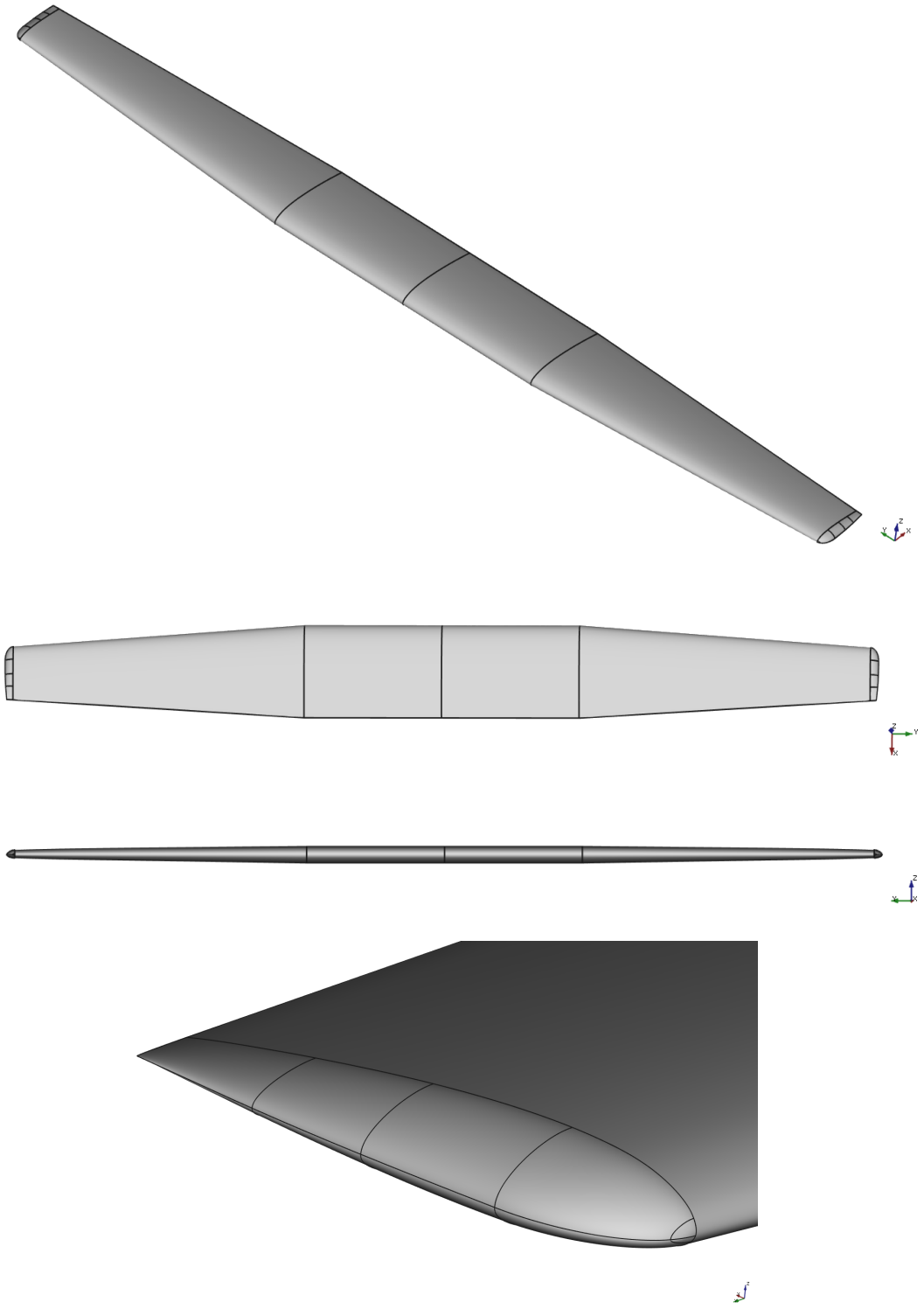

Figure 3.18 ATR-72 solid wing, with a close-up on the wing tip

3.5 AircraftUtils additional methods

The methods analyzed in the previous paragraphs provide just the necessary tools to build CAD shapes out of data stored in JPAD aircraft classes. In order to actually be able to import aircraft data from files and write generated CAD shapes to specific file formats, AircraftUtils needs to be filled with additional methods. This paragraph gives an overview on the extra

functions (and sub-classes) contained in AircraftUtils, providing basic information on how they work and the classes they make use of.

3.5.1 importAircraft

The importAircraft method is the tool that has been used in JPADCADSandbox tests in order to actually import aircraft data from XML files and populate instances of the JPAD aircraft class. The operations performed by this method rely on an important Java library: args4j [17]. In short, args4j library provides classes that allow to parse command line options. In this way, the paths to the directories and the files containing the necessary information to build new aircraft instances can be easily managed and passed to the JPAD utilities that provide support to XML file reading operations. In particular, the importAircraft method accepts just one parameter, of the Java String[] type, that contains the list of all the necessary directories. This list must be written according to the same notation used for the definition of the options of a supporting class (which in the present case has been called ArgumentsJPADCADSandbox) that, along with the args4j CmdLineParser class, actually performs the parsing of the command line. Once the input array has been parsed, it is finally possible to populate the aircraft object, by means of the JPADXmlReader class. An example of usage of this method is given in paragraph 3.6.

3.5.2 enum classes

The AircraftUtils class contains two Java enum classes, which have yet been mentioned in the previous paragraphs. What follows focuses on their definitions and their usage.

FileExtension

This enum class is used to enumerate all the possible file formats, supported by the OCCT library, that can be used to save CAD entities. In particular, this class is used by another AircraftUtils method, which is explained just below, in order to select the right operations to perform for the purpose of writing CAD shapes to a specific file format. Listing 3.16 reports the structure of this enum class.

```
1 public enum FileExtension {
2     BREP,
3     STEP,
4     IGES,
5     STL;
6 }
```

Listing 3.16 FileExtension class

XSpacingType

The XSpacingType class is used to enumerate the spacings that it is possible to employ once being provided an interval and the number of points that must be contained in the interval. In

addition to the definitions of the elements of the class, it also contains an abstract method, named `calculateSpacing`, that it is implemented by each one of the enumerators (listing 3.17). This `enum` class is used by the `getFuselageCAD` method in order to define the spacing for the cross sections of the nose and the tail trunks.

```
1  public enum XSpacingType {
2      UNIFORM {
3          @Override
4          public Double[] calculateSpacing(double x1, double x2, int n) {
5              Double[] xSpacing = MyArrayUtils.linspaceDouble(x1, x2, n);
6              return xSpacing;
7          }
8      },
9      COSINUS {
10         @Override
11         public Double[] calculateSpacing(double x1, double x2, int n) {
12             Double[] xSpacing = MyArrayUtils.cosineSpaceDouble(x1, x2, n);
13             return xSpacing;
14         }
15     },
16     HALFCOSINUS1 { // finer spacing close to x1
17         @Override
18         public Double[] calculateSpacing(double x1, double x2, int n) {
19             Double[] xSpacing = MyArrayUtils.halfCosine1SpaceDouble(x1, x2, n);
20             return xSpacing;
21         }
22     },
23     HALFCOSINUS2 { // finer spacing close to x2
24         @Override
25         public Double[] calculateSpacing(double x1, double x2, int n) {
26             Double[] xSpacing = MyArrayUtils.halfCosine2SpaceDouble(x1, x2, n);
27             return xSpacing;
28         }
29     };
30
31     public abstract Double[] calculateSpacing(double x1, double x2, int n);
32 }
```

Listing 3.17 `XSpacingType` class

3.5.3 CAD methods overload

One of the most important characteristics of a language such as Java is the possibility to perform methods overloading. Method overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. For this reason, it is possible to define various `getLiftingSurfaceCAD` and `getFuselageCAD`, differing from each other for the arguments they support. The following listing reports all the different versions for the aforementioned methods currently contained in `AircraftUtils`.

```
1  public static List<OCCShape> getFuselageCAD(Fuselage fuselage,
2        double noseCapSectionFactor1, double noseCapSectionFactor2,
3        int numberNoseCapSections,
4        int numberNosePatch2Sections, XSpacingType spacingTypePatch2,
5        int numberTailPatchSections, XSpacingType spacingTypeTailPatch,
6        double tailCapSectionFactor1, double tailCapSectionFactor2,
7        int numberTailCapSections,
8        boolean exportLofts, boolean exportSolids, boolean exportSupportShapes) {
9
10       ...
11 }
12
13 public static List<OCCShape> getFuselageCAD(Fuselage fuselage,
14       boolean exportSupportShapes) {
15
16    return getFuselageCAD(fuselage,
17          0.15, 1.0, 3,
18          7, XSpacingType.COSINUS,
19          7, XSpacingType.COSINUS,
20          1.0, 0.15, 3,
21          true, true, exportSupportShapes);
22 }
23
24 public static List<OCCShape> getFuselageCAD(Fuselage fuselage,
25       boolean exportLofts, boolean exportSolids, boolean exportSupportShapes) {
26
27    return getFuselageCAD(fuselage,
28          0.15, 1.0, 3,
29          7, XSpacingType.COSINUS,
30          7, XSpacingType.COSINUS,
31          1.0, 0.15, 3,
32          exportLofts, exportSolids, exportSupportShapes);
33 }
34
35 public static List<OCCShape> getFuselageCAD(Fuselage fuselage,
36       int numberNosePatch2Sections, int numberTailPatchSections,
37       boolean exportLofts, boolean exportSolids, boolean exportSupportShapes) {
38
39    return getFuselageCAD(fuselage,
40          0.15, 1.0, 3,
41          numberNosePatch2Sections, XSpacingType.COSINUS,
42          numberTailPatchSections, XSpacingType.COSINUS,
43          1.0, 0.15, 3,
44          exportLofts, exportSolids, exportSupportShapes);
45 }
46
47 public static List<OCCShape> getLiftingSurfaceCAD(LiftingSurface liftingSurface,
48       ComponentEnum typeLS,
49       double tipTolerance,
50       boolean exportLofts,
51       boolean exportSolids,
52       boolean exportSupportShapes) {
```

```
53
54        ...
55  }
56
57  public static List<OCCShape> getLiftingSurfaceCAD(LiftingSurface liftingSurface,
58        ComponentEnum typeLS,
59        boolean exportSupportShapes) {
60
61      return getLiftingSurfaceCAD(liftingSurface,
62          typeLS,
63          1e-3,
64          true,
65          true,
66          exportSupportShapes);
67  }
```

Listing 3.18 CAD generating methods overloading in `AircraftUtils`

`AircraftUtils` also contains a method, called `getAircraftShapes`, that allows to perform more than just one CAD generating operation at a time, meaning that fuselage and lifting surface shapes can be generated at the same time by using one single method. In particular, the method accepts an object representing the aircraft, and a list containing values of the JPAD `ComponentEnum` class, specifying which type of shapes (i.e., fuselage, wing, tail, canard) must be produced. Then the method returns a list containing all the generated `OCCShape` objects (listing 3.19).

```
1   public static List<OCCShape> getAircraftShapes(
2         Aircraft theAircraft,
3         List<ComponentEnum> components
4         ) {
5       List<OCCShape> allShapes = new ArrayList<>();
6
7       if(theAircraft.getFuselage() != null &&
8          components.contains(ComponentEnum.FUSELAGE)) {
9         allShapes.addAll(getFuselageCAD(theAircraft.getFuselage(), true));
10      }
11      if(theAircraft.getWing() != null &&
12         components.contains(ComponentEnum.WING)) {
13        allShapes.addAll(getLiftingSurfaceCAD(theAircraft.getWing(),
14          ComponentEnum.WING, 1e-3, true, true, true));
15      }
16      if(theAircraft.getHTail() != null &&
17         components.contains(ComponentEnum.HORIZONTAL_TAIL)) {
18        allShapes.addAll(getLiftingSurfaceCAD(theAircraft.getHTail(),
19          ComponentEnum.HORIZONTAL_TAIL, 1e-3, true, true, true));
20      }
21      if(theAircraft.getVTail() != null &&
22         components.contains(ComponentEnum.VERTICAL_TAIL)) {
23        allShapes.addAll(getLiftingSurfaceCAD(theAircraft.getVTail(),
24          ComponentEnum.VERTICAL_TAIL, 1e-3, true, true, true));
25      }
```

```
26      if(theAircraft.getCanard() != null &&
27         components.contains(ComponentEnum.CANARD)) {
28        allShapes.addAll(getLiftingSurfaceCAD(theAircraft.getCanard(),
29          ComponentEnum.CANARD, 1e-3, true, true, true));
30      }
31      return allShapes;
32    }
```

Listing 3.19 `getAircraftShapes` method

3.5.4 `getAircraftSolidFile`

The `getAircraftSolidFile` method allows to write solid shapes to file. It requires a Java List containing generic `OCCShape` entities, a `String` standing for the name of the file we want to write, and an element of the `FileExtension` class, which specifies the format of the file to be written. The first operation the method performs consists in filtering the input shapes, collecting all the solids found in one single list. This operation is made through the use of another `AircraftUtils` static method, called `getAircraftSolid`. Once all the solids have been obtained, the algorithm automatically chooses, by use of a switch statement, the right operations to perform in order to obtain a file in the format specified by the user. Currently the method supports writing in the following extensions:

- BRep,
- STEP,
- IGES,
- STL.

All the writing operations rely on particular OCCT classes, each one dedicated to a specific file format.

3.6 Example

The following example gives an idea of how the previously described methods can be used to automatically generate the CAD model of an aircraft. For this purpose, a simple testing class has been created in `JPADCADSandbox`. This class contains a main method that receives as input a `String[]`, containing the paths to the directories that contain the XML files from which we want to read the aircraft data. The first operation consists in importing this data and fill in the *fields* of a new `Aircraft` object. This operation is conducted by using the `importAircraft` method, by providing it with the aforementioned `String` array. The next step consists in generating new instances of the `Fuselage` and `LiftingSurface` classes by making use of the methods of the `Aircraft` class. Once they have been obtained, CAD generating methods can be launched, by selecting one of the methods seen in section 3.5.3. In order to obtain the solid of the aircraft and write it to file (for the purpose of using it later, for example, in a CAD or a CFD software),

the `getAircraftSolidFile` method described above can be used, by providing it with the shapes we want to write, and the name and the format of the file (listing 3.20).

```
1  public class Test {
2
3    public static void main(String[] args) {
4
5      System.out.println("JPADCADSandbox Test");
6
7      // Import aircraft data from XML files
8      Aircraft aircraft = AircraftUtils.importAircraft(args);
9
10     // Generate instances of the JPAD
11     // classes describing aircraft components
12     Fuselage fuselage = aircraft.getFuselage();
13     LiftingSurface wing = aircraft.getWing();
14     LiftingSurface hTail = aircraft.getHTail();
15     LiftingSurface vTail = aircraft.getVTail();
16
17     // Generate CAD shapes for all the aircraft components
18     // and add them to a single List, in order to be exported
19     List<OCCShape> allShapes = new ArrayList<>();
20
21     List<OCCShape> fuselageShapes = AircraftUtils.getFuselageCAD(
22         fuselage, 7, 7, true, true, true);
23
24     List<OCCShape> wingShapes = AircraftUtils.getLiftingSurfaceCAD(
25         wing, ComponentEnum.WING, 1e-3, true, true, true);
26
27     List<OCCShape> hTailShapes = AircraftUtils.getLiftingSurfaceCAD(
28         hTail, ComponentEnum.HORIZONTAL_TAIL, 1e-3, true, true, true);
29
30     List<OCCShape> vTailShapes = AircraftUtils.getLiftingSurfaceCAD(
31         vTail, ComponentEnum.VERTICAL_TAIL, 1e-3, true, true, true);
32
33     allShapes.addAll(fuselageShapes);
34     allShapes.addAll(wingShapes);
35     allShapes.addAll(hTailShapes);
36     allShapes.addAll(vTailShapes);
37
38     // Filter the shape list in order to collect the solid
39     // ones only, and write them to STEP file format
40     AircraftUtils.getAircraftSolidFile(allShapes, "aircraft", FileExtension.STEP);
41   }
42 }
```

Listing 3.20 Usage of `AircraftUtils` methods

Instead of generating CAD shapes for each component at a time, one could have chosen to use the `getAircraftShapes` method and provide it with the list of desired aircraft components (listing 3.21). The results would have been exactly the same.

```
1       List<ComponentEnum> componentList = new ArrayList<>();
2       componentList.add(ComponentEnum.FUSELAGE);
3       componentList.add(ComponentEnum.WING);
4       componentList.add(ComponentEnum.HORIZONTAL_TAIL);
5       componentList.add(ComponentEnum.VERTICAL_TAIL);
6
7       allShapes.addAll(AircraftUtils.getAircraftShapes(
8           aircraft, componentList));
```

Listing 3.21 Usage of `getAircraftShapes` method

Figure 3.19 shows the CAD solid models of four different aircrafts obtained by using the same test along with the same methods listed above. It has to be noted how the same algorithm, `getLiftingSurfaceCAD`, succeeds at building CAD shapes for very different lifting surfaces, revealing to be quite versatile.

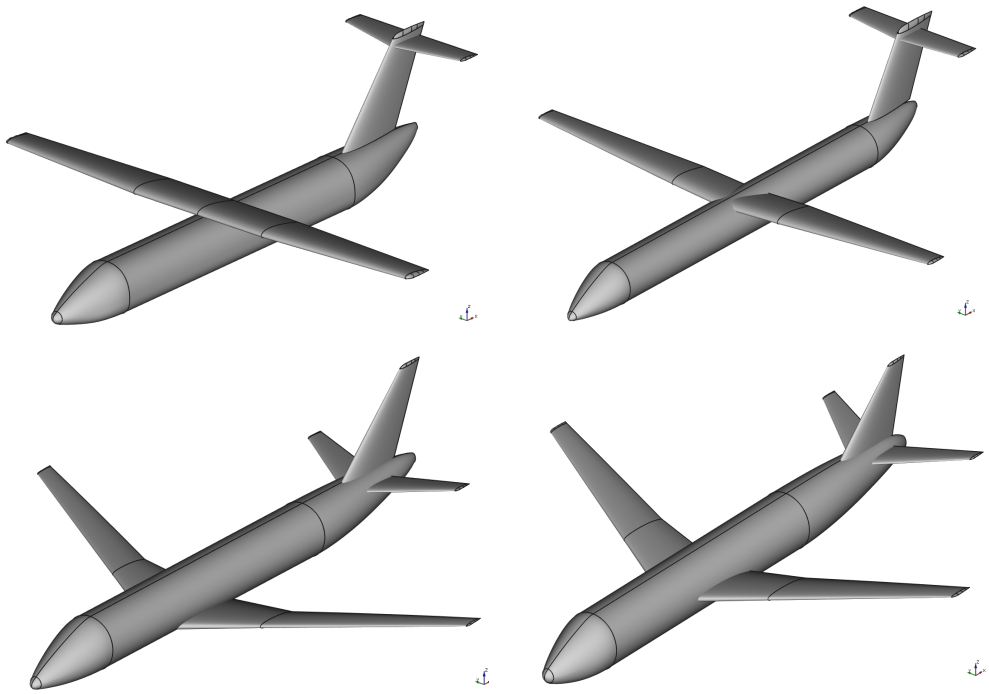

Figure 3.19 Comparison between different aircraft CAD models obtained by using the same test class

Chapter 4

Automatic workflows for aerodynamic simulations

4.1 Introduction

Since the very beginning, the main purpose behind the development of a library devoted to the automatic generation of 3D CAD models of aircrafts has been the possibility of using the same models for aerodynamic analyses, carried out by the use of a CFD software. For this reason, the models produced by JPAD come equipped with solutions (such as the modeling of the wing tip, or the automatic gap enhancement of the aifoils trailing edge in case of necessity) that make them ready to be imported and used into a suite for aerodynamics investigations. Moreover, since the models are generated without the necessity of human intervention during the process, and considering that the classes of JPAD offer the possibility to make changes to the geometric characteristics of aircraft components once imported from data file, what comes to mind is the possibility to integrate the `JPADCAD` module, along with the `AircraftUtils` functionalities, in a comprehensive development cycle involving CFD analyses. By use of the aforementioned functions and classes, in fact, it becomes possible to automatically produce several geometries of the same aircraft, differing in some characteristics (e.g., the wing aspect ratio, or the dihedral angle). These models should be then imported, one at a time and in an automatic way, into a software for Computational Fluid Dynamics, in order to perform on them all the necessary analyses and produce the requested results. What becomes necessary at this point is some sort of *bridge*, connecting JPAD and all its modules to some existing CFD software. This software, however, based on what has been said above, must fulfill the following requirements:

- must provide support for recording and playing macros,
- must give the possibility to run macros in batch mode.

These macros are going to collect all the operations that will be performed by the CFD software, from importing the CAD models to collecting results. In this way, one can think

to launch a complete workflow from a simple test class (i.e., a class containing just a main method) located in JPAD, which manages all the operations, from establishing the type of the analysis to actually running the software and the macro.

Therefore a choice needs to be made about the CFD software. CD-adapco STAR-CCM+ meets the characteristics listed above. In addition, STAR-CCM+ macros are written in Java, with this representing a great advantage, since JPAD is written in Java too. In fact, in order for our Java macros to run properly and perform operations as we want, it is necessary to provide them support by means of dedicated classes and utilities. These classes not only are used by the macros, but can also be employed by the test class in JPAD in order to define the characteristics of the simulations, as it will be clearer in the following paragraphs.

4.2 STAR-CCM+ overview

STAR-CCM+ is not just a tool for CFD analyses. More in general, it is a Computer-aided Engineering (CAE) solution for solving multidisciplinary problems in both fluid and solid continuum mechanics, within a single integrated interface. The STAR-CCM+ environment offers all the tools required in order to execute engineering analyses. These tools allows the following operations:

- import and creation of geometries,
- mesh generation,
- solution of the governing equations,
- analysis of the results,
- automation of the simulation workflows for design exploration studies,
- connection to other CAE software for co-simulation analysis.

For what concerns geometries, STAR-CCM+ can read geometric data saved in neutral formats (such as IGES and STEP) or triangulated ones (such as STL). Besides, it also offers a built-in capability for modifying and creating CAD geometries directly. Its 3D CAD tool is a parametric feature-based modeler that is built upon the Parasolid kernel. Among its extensive suite of CAD functions it includes important operations, such as Boolean actions, automatic removal of small bodies or surfaces irrelevant to simulation, and flow domain extraction. Regarding mesh operations, STAR-CCM+ provides a complete set of capabilities for both surface and volume meshing operations. Its mesh framework provides a flexible environment and, most importantly, repeatable processes. The principal features of the framework consist in allowing serial, concurrent per part, and parallel mesh generation, permitting the sequencing and re-ordering of mesh operations, and providing the capability to perform local mesh modifications. Being a CAE software, STAR-CCM+ solves systems of equations derived from the fundamental laws of physics and spanning several different fields, from fluid to solid mechanics, passing through aeroacoustics and reacting flows. In order to solve these systems of equations, built from the chosen models and their boundary conditions, it makes use of numerical algorithms, based on finite volume and finite element methods. [9]

4.3 A supporting package: MacroExtras

MacroExtras is the package containing all the *extra* classes and utilities used by our STAR-CCM+ macros and by the test launching classes. These classes are arranged as figure 4.1 shows, while the tasks they fulfill can be listed as follows:

- provide support to XML writing and reading operations;
- handle the entities and the parameters of the simulation;
- provide enumerations in order to easily manage the entities and the parameters of the simulation.

The following paragraphs give a description of each of the classes that is contained in MacroExtras, focusing on the Java classes used for accomplishing their tasks and on the eventual Design Pattern behind the way they have been written.

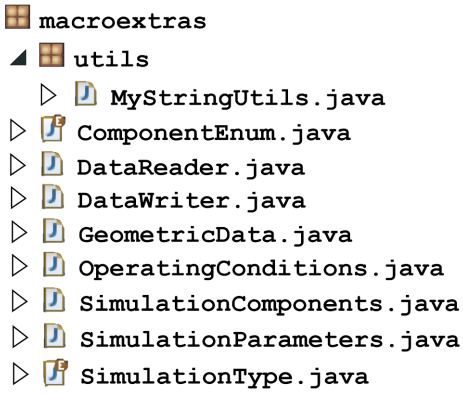

Figure 4.1 MacroExtras package structure

4.3.1 enum classes

The MacroExtras is provided with two Java enum classes, that are intended for easily managing:

- all the different types of aircraft components that we are able to generate, in CAD form, by the use of JPADCAD and AircraftUtils;
- the types of simulation that our STAR-CCM+ macros allow us to perform.

These enum classes are called, respectively, ComponentEnum (as the enum class used by JPAD to manage aircraft components, but with actually fewer elements) and SimulationType (listing 4.2 and 4.1).

```
1 public enum SimulationType {
2     EULER,
3     VISCOUS
4 }
```

Listing 4.1 SimulationType enumeration class

```
1 public enum ComponentEnum {
2     FUSELAGE,
3     WING,
4     HORIZONTAL,
5     VERTICAL,
6     CANARD;
7 }
```

Listing 4.2 `ComponentEnum` enumeration class

4.3.2 Simulation data classes

In order to provide the simulation with the data necessary for its execution, it is necessary to create some classes whose main purpose is to store and manage essential information. These classes handle three different types of data, which are the following:

- data related to the geometry of the aircraft, subject of the simulation;
- data regarding the operating conditions for the simulation;
- simulation parameters.

Given the large amount of data that needs to be stored in these classes (or at least for two of them), which involves the need to handle a large number of different attributes for each class, following some special strategy to facilitate the creation of new instances immediately appeared to be an opportune and advantageous choice. As previously mentioned (Chapter 2), in software engineering there are several Design Patterns that provide a general repeatable solution to commonly occurring problems in software design. In this specific case, the problem is related to the object creation and the Design Pattern that usually comes to the rescue in cases like this is the Builder Pattern. What the Builder Pattern does is to allow to create instances of very complex objects in an easiest way [2]. Constructors in Java are used to create objects, and what they typically do is to require a certain number of parameters which, in some cases, can be equal to the number of attributes of the class they work for. Since this number can be very high, mantaining and using such a class can become an overwhelming task. Through the use of the Builder Pattern, one can create an inner class (inside the original one) which will be the actual Builder. This inner class uses the same fields of the outer class and has several `set` methods, that help the building process of new instances of the original class by filling in all the fields of the Builder class. The Builder class, finally, has also a `build` method, that actually builds new instances of the outer class, by providing its constructor with the inner class attributes, which are the same of the original class and have been set by the use of the methods cited above. The following sections analyze each of the simulation data classes, providing examples of how the code has been organized by means of the just mentioned pattern.

`GeometricData` class

The `GeometricData` class is the one in charge of collecting data related to the components of the aircraft (their type and number), and the geometric characteristics of some of these

components (such as the length of the fuselage and the surface of the wing). The majority of this data is used by the macros in order to populate expression reports inside STAR-CCM+ simulations. These reports are in turn used in order to create further different reports (such as aerodynamic coefficient ones) or to provide support when defining the dimensions of the fluid domain. Listing 4.3 shows part of the actual code of the class, drawing attention on the particular pattern adopted for it and the other classes of this group. The attributes managed by the class are summarized in table 4.1, along with a brief description for each of them. All the quantities in the class related to lenghts and areas are expressed/need to be expressed in meters and square meters.

Attribute	Description
`cadUnits`	A `String` that specifies the length unit adopted for the CAD components definition
`components`	A Java List containing enumerators specifying which aircraft components are going to be imported into STAR-CCM+
`componentsNumber`	A `int` array that defines the number of occurrences for each of the components specified above
`fuselageLength`	A `double` value representing the length of the fuselage (l_{fus}) if present among the components
`meanAerodynamicChord`	A `double` value representing the mean aerodynamic chord (MAC) of the wing
`wingSurface`	The attribute which stores the value of the wing planform surface (S_{wing}) in `double` format
`wingSpan`	The attribute that stores the value of the wing span (s_{wing}) in `double` format
`momentPoleXCoord`	A `double` value representing the x coordinate of the point to be used for aerodynamic moment coefficients calculation by STAR-CCM+

Table 4.1 `GeometricData` attributes

```
1  public class GeometricData {
2
3      // GeometricData class attributes
4      private String cadUnits;
5      private List<ComponentEnum> components;
6      private int[] componentsNumber;
7      private double fuselageLength;
8      private double meanAerodynamicChord;
9      private double wingSurface;
10     private double wingSpan;
```

```
11      private double momentPoleXCoord;
12
13      // GeometricData class constructor
14      public GeometricData(
15            String cadUnits,
16            List<ComponentEnum> components,
17            int[] componentsNumber,
18            double fuselageLength,
19            double meanAerodynamicChord,
20            double wingSurface,
21            double wingSpan,
22            double momentPoleXCoord
23          ) {
24          this.cadUnits = cadUnits;
25          this.components = components;
26          this.componentsNumber = componentsNumber;
27          this.fuselageLength = fuselageLength;
28          this.meanAerodynamicChord = meanAerodynamicChord;
29          this.wingSurface = wingSurface;
30          this.wingSpan = wingSpan;
31          this.momentPoleXCoord = momentPoleXCoord;
32      }
33
34      // GeometricData class Builder
35      public static class GeometricDataBuilder {
36
37          // GeometricDataBuilder class attributes
38          private String cadUnits;
39          private List<ComponentEnum> components;
40          private int[] componentsNumber;
41          private double fuselageLength;
42          private double meanAerodynamicChord;
43          private double wingSurface;
44          private double wingSpan;
45          private double momentPoleXCoord;
46
47          // Setters for the GeometricDataBuilder class
48          public GeometricDataBuilder setCadUnits(String cadUnits) {
49              this.cadUnits = cadUnits;
50              return this;
51          }
52          ...
53
54          public GeometricDataBuilder setMomentPoleXCoord(double momentPoleXCoord) {
55              this.momentPoleXCoord = momentPoleXCoord;
56              return this;
57          }
58
59          // Actual GeometricData class builder
60          public GeometricData build() {
61              return new GeometricData(
62                  cadUnits,
63                  components,
```

```
64                  componentsNumber,
65                  fuselageLength,
66                  meanAerodynamicChord,
67                  wingSurface,
68                  wingSpan,
69                  momentPoleXCoord
70                  );
71          }
72      }
73  }
```

Listing 4.3 `GeometricData` class overview

`OperatingConditions` class

The `OperatingConditions` class, as the name suggests, consists in attributes and methods managing the state in which the aircraft is operating. This class manages an even larger number of attributes with respect to the previous one (which was the main reason behind the Builder Pattern choice): it includes variables taking into account the angles between reference lines of the aircraft and the flow direction (such as the angle of attack (AOA)), attributes relative to dimensionless quantities such as the Mach and the Reynolds numbers, and variables regarding the operating altitude and all the air quantities that depend from it according to the International Standard Atmosphere (ISA), such as the pressure, the temperature, the density, etc. These quantities, as the ones mentioned for the `GeometricData` class, are used by our STAR-CCM+ macros in order to define vectors referring to important directions (such as lift and drag directions), aerodynamic coefficients (whose definition is supplied by the just mentioned vectors too), and the boundary conditions. Table 4.2 provides the list of the attributes owned by the `OperatingConditions` class along with their units, with no further explanations since the names are quite intelligible. The only thing that needs to be specified concerns the angles: they are calculated with respect to the body axes of the aircraft. Finally, listing 4.4 gives an overview on how the class is organized.

Attribute	Symbol	Units
`angleOfAttack`	α	$^\circ$
`sideslipAngle`	β	$^\circ$
`machNumber`	M	
`reynoldsNumber`	Re	
`altitude`	h	ft
`pressure`	p	Pa
`density`	ρ	$\mathrm{kg\,m^{-3}}$
`temperature`	T	K
`speedOfSound`	a	$\mathrm{m\,s^{-1}}$
`dynamicViscosity`	μ	Pa s
`velocity`	V	$\mathrm{m\,s^{-1}}$

Table 4.2 List of `OperatingConditions` class attributes

```java
public class OperatingConditions {
    private double angleOfAttack;
    private double sideslipAngle;
    ...
    private double velocity;

    public OperatingConditions(
        double angleOfAttack,
        double sideslipAngle,
        ...
        double velocity
        ) {
      this.angleOfAttack = angleOfAttack;
      this.sideslipAngle = sideslipAngle;
      ...
      this.velocity = velocity;
    }

    public static class OperatingConditionsBuilder {
        private double angleOfAttack;
        private double sideslipAngle;
        ...
        private double velocity;

        public OperatingConditionsBuilder setAngleOfAttack(double angleOfAttack) {
            this.angleOfAttack = angleOfAttack;
            return this;
        }

        ...

        public OperatingConditionsBuilder setVelocity(double velocity) {
            this.velocity = velocity;
            return this;
        }

        public OperatingConditions build() {
            return new OperatingConditions(
              angleOfAttack,
              sideslipAngle,
              ...
              velocity
              );
        }
    }
}
```

Listing 4.4 `OperatingConditions` class overview

SimulationParameters class

The last of the simulation data classes is the one dedicated to the collection of the parameters for the simulation. This class currently contains just three attributes, listed in table 4.3 along with a brief description, but it is probably intended to accomodate several more as the development of the package proceeds, in order to guarantee the user greater control over the settings and the actions performed during the simulation. One could think, for example, to give the user the possibility to change the settings for the operations regarding the mesh, in order to refine it or make it coarser. Listing 4.5 contains excerpts from the original code, showing how the class has been structured.

Attribute	Description
`simulationType`	An instance of the `SimulationType` enumeration class that specifies the type of the simulation we want to perform in STAR-CCM+
`isSymmetrical`	An attribute of the `boolean` type that sets whether the simulation to be performed is symmetrical or not (with respect to the plane of symmetry of the aircraft)
`executeMesh`	Another attribute of the Java `boolean` type that tells STAR-CCM+ whether to execute or not the operations related to the mesh generation and actually run the simulation

Table 4.3 `SimulationParameters` class attributes

```
1  public class SimulationParameters {
2      private SimulationType simulationType;
3      private boolean isSymmetrical;
4      private boolean executeMesh;
5
6      public SimulationParameters(
7            SimulationType simulationType,
8            boolean isSymmetrical,
9            boolean executeMesh
10           ) {
11         this.simulationType = simulationType;
12         this.isSymmetrical = isSymmetrical;
13         this.executeMesh = executeMesh;
14     }
15
16     public static class SimulationParametersBuilder {
17         private SimulationType simulationType;
18         private boolean isSymmetrical;
19         private boolean executeMesh;
20
21         public SimulationParametersBuilder setSimulationType(
22               SimulationType simulationType) {
23             this.simulationType = simulationType;
24             return this;
```

```
25          }
26
27          public SimulationParametersBuilder setIsSymmetrical(boolean isSymmetrical) {
28              this.isSymmetrical = isSymmetrical;
29              return this;
30          }
31
32          public SimulationParametersBuilder setExecuteMesh(boolean executeMesh) {
33              this.executeMesh = executeMesh;
34              return this;
35          }
36
37          public SimulationParameters build() {
38              return new SimulationParameters(
39                simulationType,
40                isSymmetrical,
41                executeMesh
42                );
43          }
44      }
45  }
```

Listing 4.5 `SimulationParameters` class overview

4.3.3 XML operations classes

The classes described in section 4.3.2 do not act as mere *containers* for the parameters and the settings to be passed to the simulation. On the contrary, their main purpose is to be directly used in order to support the reading and the writing operations regarding the XML data file that it is actually used by STAR-CCM+ macros in order to *populate* the simulation. For this purpose, `MacroExtras` has been provided with two dedicated classes, `DataWriter` and `DataReader`, as well as a utility class, called `MyStringUtils`, which provides several static methods used by the aforementioned classes in order to execute some basic operations on strings and arrays of strings.

`DataWriter` class

The `DataWriter` class is the one that actually writes all the data discussed in section 4.3.2 to XML file. As mentioned before, Java greatly supports XML reading and writing operations by means of dedicated packages. The `DataWriter` class owns four attributes, three of which belong to the classes discussed in the previous paragraph, while the fourth one is an instance of the Java `Document` interface, which represents the entire XML document that it is produced and handed back to the user. The class constructor accepts three objects of the aforementioned classes and by means of them sets its own attributes. Besides, it generates an instance of the Java `Document` interface and starts to populate it in order to create a tree structure, with each branch representing one of the simulation data classes. Then the constructor starts appending *leaves* to the branches, with each *leaf* representing an attribute of the class the branch belongs to. Listing 4.6 shows an excerpt of the actual code in which the branch for the

operating conditions is being populated. Once all these operations have been completed, the final document is passed to the class `Document` attribute, ready to be used by the class method `write`, that actually transfers the document to file once provided with a Java `String` standing for desired file path and the name to be given to the data file (listing 4.7).

```
// Generate a Document instance inside the constructor
DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
Document doc = docBuilder.newDocument();

// Generate the root of the file
Element rootElement = doc.createElement("data");
doc.appendChild(rootElement);

// Generate the branch for the operating conditions
Element operatingConditionsElement = doc.createElement("operating_conditions");

// Append this branch to the root
rootElement.appendChild(operatingConditionsElement);

// Create elements for the operating conditions
Element angleOfAttack = doc.createElement("angle_of_attack");
Element sideslipAngle = doc.createElement("sideslip_angle");
...
Element velocity = doc.createElement("velocity");

// Fill these elements with the data provided by the OperatingConditions class
angleOfAttack.appendChild(
  doc.createTextNode(Double.toString(operatingConditions.getAngleOfAttack())));
sideslipAngle.appendChild(
  doc.createTextNode(Double.toString(operatingConditions.getSideslipAngle())));
...
velocity.appendChild(
  doc.createTextNode(Double.toString(operatingConditions.getVelocity())));

// Append these elements to the branch they belong to
operatingConditionsElement.appendChild(angleOfAttack);
operatingConditionsElement.appendChild(sideslipAngle);
...
operatingConditionsElement.appendChild(velocity);
```

Listing 4.6 `DataWriter` class constructor

```
public void write(String filepath) {

    try {

      // The doc variable used by the method is the
      // Document instance populated by the constructor
      TransformerFactory transformerFactory = TransformerFactory.newInstance();
      Transformer transformer = transformerFactory.newTransformer();
      DOMSource source = new DOMSource(doc);
```

```
10
11        StreamResult result = new StreamResult(new File(filepath));
12        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
13        transformer.transform(source, result);
14      }
15
16      catch (TransformerException tfe) {
17        tfe.printStackTrace();
18      }
19    }
```

Listing 4.7 `DataWriter` class `write` method

Listing 4.8 shows the XML file actually produced through the use of the class `write` method. This is the file that the STAR-CCM+ macros are handed over, which requires a class managing the process of reading data from it.

```
1  <data>
2        <operating_conditions>
3              <angle_of_attack>2.0</angle_of_attack>
4              <sideslip_angle>0.0</sideslip_angle>
5              <Mach>0.64</Mach>
6                    <Reynolds>1.826622621962677E7</Reynolds>
7              <altitude unit="ft">30000.0</altitude>
8              <pressure unit="Pa">30148.62980549334</pressure>
9              <density unit="kg/m^3">0.45904041459985995</density>
10             <temperature unit="K">228.79937393459855</temperature>
11             <speed_of_sound unit="m/s">303.23012560617843</speed_of_sound>
12             <dynamic_viscosity unit="Pa*s">1.4875263548207093E-5</dynamic_viscosity>
13             <velocity unit="m/s">194.0672803879542</velocity>
14       </operating_conditions>
15       <geometric_data>
16             <CAD_units>mm</CAD_units>
17             <aero_components>[FUSELAGE, CANARD, WING]</aero_components>
18             <components_number>[1, 1, 1]</components_number>
19             <fuselage_length unit="m">38.03999999999999</fuselage_length>
20             <wing_MAC unit="m">3.050073169187992</wing_MAC>
21             <wing_S unit="m^2">98.59823999999995</wing_S>
22             <wing_span unit="m">34.34116884366766</wing_span>
23             <moment_pole_Xcoord unit="m">23.04147231852965</moment_pole_Xcoord>
24       </geometric_data>
25       <simulation_parameters>
26             <type>EULER</type>
27             <symmetrical>true</symmetrical>
28             <execute_automesh>false</execute_automesh>
29       </simulation_parameters>
30 </data>
```

Listing 4.8 STAR-CCM+ macro input data file

DataReader class

The DataReader class is the one dedicated to reading the XML data file. Its three attributes belong to the classes explained in section 4.3.2 and are the ones that get populated by means of the class constructor. It accepts one single parameter in the String format, standing for the path to the file to be read. Once got the file, it parses it and sorts through its elements by means of their tag name, obtaining three nodes, each one specific to one of the simulation data classes. Then each of the nodes gets parsed in turn, in order to populate the aforementioned attributes. Listing 4.9 reports an excerpt from the original code, showing how the constructor works in order to populate the attributes related to the operating conditions. Obviously, the actions performed for the remaining attributes are pretty much the same. Once the constructor has finished its work and the attributes of the class have been succesfully populated, the same can be accessed, in order to be used by means of simple getter methods.

```
1  public class DataReader {
2
3    // Attributes
4    private OperatingConditions operatingConditions = null;
5    ...
6
7    // Constructor
8    public DataReader(String filePath) {
9      try {
10
11       // Generate a new Document instance by
12       // using the path to the actual XML data file
13       File xmlFile = new File(filePath);
14       DocumentBuilderFactory docBuilderFactory =
15           DocumentBuilderFactory.newInstance();
16       DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
17       Document doc = docBuilder.parse(xmlFile);
18
19       doc.getDocumentElement().normalize();
20
21       // Get each XML file branch by means of its tag name
22       Node node1 = doc.getElementsByTagName("operating_conditions").item(0);
23       ...
24
25       // Populate the OperatingConditions attribute
26       if(node1.getNodeType() == Node.ELEMENT_NODE) {
27         Element element1 = (Element) node1;
28         this.operatingConditions = new OperatingConditions
29           .OperatingConditionsBuilder()
30             .setAngleOfAttack(
31               Double.parseDouble(
32                 element1
33                 .getElementsByTagName("angle_of_attack").item(0).getTextContent()))
34             .setSideSlipAngle(
35               Double.parseDouble(
36                 element1.
37                 getElementsByTagName("sideslip_angle").item(0).getTextContent()))
```

```
38              ...
39              .build();
40        }
41        ...
42      }
43
44      catch(Exception exception) {
45        exception.printStackTrace();
46      }
47    }
48
49    // Getter for the OperatingConditions attribute
50    public OperatingConditions getOperatingConditions() {
51      return this.operatingConditions;
52    }
53    ...
54  }
```

Listing 4.9 `DataReader` class overview

`MyStringUtils` class

In order for the `DataReader` class to parse the XML file correctly, avoid excessive code duplication, and even provide support for the operations taking place in the macro, an utility class, called `MyStringUtils`, has been added to the `MacroExtras` package, that supplies methods in order to:

- strip the extension from file name strings,
- get an array of strings from a single string representing an array with brackets,
- convert arrays of strings to arrays of integers,
- convert arrays of strings to lists of `ComponentEnum` objects,
- convert strings to `SimulationType` entities,
- getting just the file name from a string representing the whole file path.

4.3.4 CAD files management: `SimulationComponents` class

The macro, in order to work, does not need the link just to the XML file containing all the simulation data. The macro also has to import CAD parts into the simulation, and, in order to perform this operation, it necessitates both data coming from the XML file (such as the components and their number) and the path to the directory which actually contains the CAD parts on which the simulation has to be executed. The class that supervises the operations related to the collection of each single file path pointing at some specific component, linking them with instances of the `ComponentEnum` class, is called `SimulationComponents`. What this class makes is building a Java Map [14] containing instances of the `ComponentEnum` class as keys, associated to Java Lists of strings standing for the paths to the actual CAD files (saved in

STEP format). This Map is extremely important, since it is the object that it is directly used by the macro in order to import aircraft fuselages and lifting surfaces into the active simulation, and name them properly. The class constructor requires three parameters:

- an array of Java Files [10], obtained by means of the paths to each single CAD STEP file;
- a Java List containing the aircraft components (in terms of `ComponentEnum` instances);
- an array of integers, which indicates the number of files present for each of the components.

What provides the `SimulationComponents` constructor with the last two parameters is an instance of the `GeometricData` class, once one has been generated in the macro after the XML data file has been read. Listing 4.10 shows the operations made by the constructor in order to obtain the aforementioned result. What needs to be highlighted is the fact that it makes use of the methods of `MyStringUtils` in order to perform basic operations on strings referring to CAD file paths. In order for the constructor to actually work it is necessary for the CAD files to be saved with an appopriate name:

- the name of the file, in upper case, must reflect the name of the component it contains;
- valid names are those that correspond to the string versions of the elements of the `ComponentEnum` class (so, a CAD file containing a vertical tail must be named VERTICAL);
- whether there is more than one single CAD file for a certain aircraft component (e.g., an aircraft with a twin tail) the name must be followed by an underscore and an index;

The naming rules listed above are the ones that need to be followed in the class that launches the macro, as the following paragraphs explains.

```
1  public class SimulationComponents {
2
3    // Attributes
4    private HashMap<ComponentEnum, List<String>> componentsMap;
5    private List<ComponentEnum> notRequiredComponents;
6
7    // Constructor
8    public SimulationComponents(
9        File[] files,
10       List<ComponentEnum> requiredComponents,
11       int[] componentsNumber) {
12
13     HashMap<ComponentEnum, List<String>> fileMap
14         = new HashMap<ComponentEnum, List<String>>();
15     HashMap<ComponentEnum, Integer> componentMap
16         = new HashMap<ComponentEnum, Integer>();
17
18     // Generate a Map taking into account the number of each component
19     for(int i = 0; i < requiredComponents.size(); i++) {
20       fileMap.put(requiredComponents.get(i), new ArrayList<String>());
```

```
21        componentMap.put(requiredComponents.get(i), componentsNumber[i]);
22      }
23
24      // Generate a Map taking into account the file path to each component
25      for(File file : files) {
26        String completeFileName = MyStringUtils.stripExtension(file.getName());
27        String fileName = MyStringUtils.deleteFromUnderscoreOn(completeFileName);
28        int fileIndex = MyStringUtils.getIndexAfterUnderscore(completeFileName);
29
30        if(requiredComponents.contains(
31            MyStringUtils.convertStringToComponentEnum(fileName))) {
32          switch(fileName) {
33          case "FUSELAGE":
34            if(fileIndex <= componentMap.get(ComponentEnum.FUSELAGE))
35              fileMap.get(ComponentEnum.FUSELAGE).add(file.getAbsolutePath());
36            break;
37          case "WING":
38            if(fileIndex <= componentMap.get(ComponentEnum.WING))
39              fileMap.get(ComponentEnum.WING).add(file.getAbsolutePath());
40            break;
41          case "HORIZONTAL":
42            if(fileIndex <= componentMap.get(ComponentEnum.HORIZONTAL))
43              fileMap.get(ComponentEnum.HORIZONTAL).add(file.getAbsolutePath());
44            break;
45          case "VERTICAL":
46            if(fileIndex <= componentMap.get(ComponentEnum.VERTICAL))
47              fileMap.get(ComponentEnum.VERTICAL).add(file.getAbsolutePath());
48            break;
49          case "CANARD":
50            if(fileIndex <= componentMap.get(ComponentEnum.CANARD))
51              fileMap.get(ComponentEnum.CANARD).add(file.getAbsolutePath());
52            break;
53          default:
54            break;
55          }
56        }
57      }
58      this.componentsMap = fileMap;
59      this.notRequiredComponents = getComplementaryList(requiredComponents);
60    }
61
62    // Getter for the Map containing all the file paths
63    public HashMap<ComponentEnum, List<String>> getComponentsMap() {
64      return componentsMap;
65    }
66    ...
67  }
```

Listing 4.10 `SimulationComponents` class overview

One final note needs to be added about the package. In order for the STAR-CCM+ macros that make use of the classes and utilities of the package to actually work, it is necessary to

provide the software with the path to the Java ARchive (JAR) file of the package, that can be obtained once it has been correctly compiled. The way to link the JAR file consists in going to the STAR-CCM+ options menu and provide the software environment with the classpath (figure 4.2).

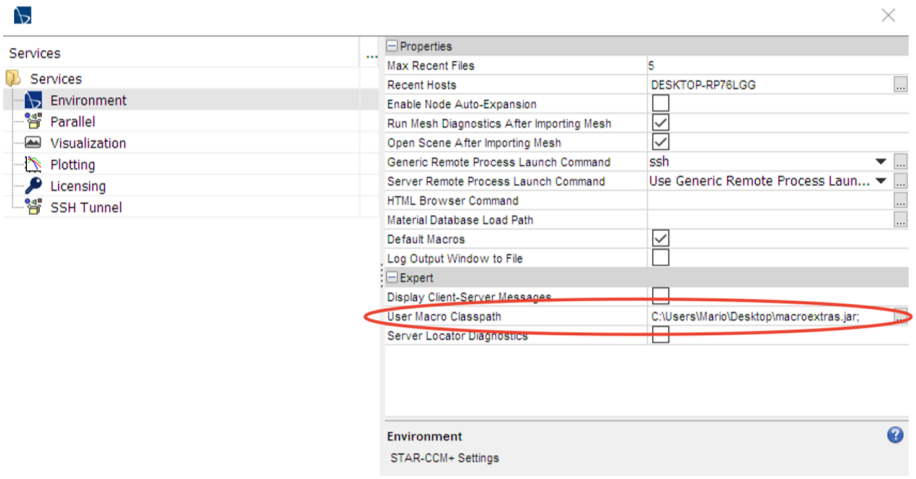

Figure 4.2 `MacroExtras` classpath setting inside STAR-CCM+

4.4 Running STAR-CCM+ macros from inside JPAD

The classes of the `MacroExtras` package listed in the previous paragraph just serve as a support for the Java files in charge of the operations that actually allow to automatize the process of analysis of a certain aircraft configuration by means of a CFD tool. The aforementioned Java files are two in number and consist of:

- a simple class providing a main method that imports some aircraft data from XML files, eventually modifies some of the components of the aircraft by means of the methods supplied by JPAD libraries, generates a CAD file for each of the components employing the functions offered by `AircraftUtils`, writes the XML file containing all the data related to the simulation, and finally runs STAR-CCM+;
- a STAR-CCM+ macro, which is a particular Java class that makes use of the STAR-CCM+ Macro API classes, along with the ones described in section 4.3, in order to perform all the operations requested before and after running an aerodynamic simulation, from importing CAD parts into the active simulation to saving the simulation once all the calculations have been completed.

The following paragraphs give an overview on how these two files have been realized and on the methods and classes they make use of. The last one, in particular, just provides an example of how the main method launching the workflow can be organized, since part of its structure depends on the type of experiment/simulation that one wants to perform.

4.4.1 The macro

A STAR-CCM+ macro is a Java program that is compiled and executed within the STAR-CCM+ workspace. In order for STAR-CCM+ to actually run macros, it comes equipped with the Java SDK (the 1.7 version, so that Java 8 improvements are not available), that allows the software to compile the Java macros. The macros that one can write or record are standard Java code, meaning that it is possible to have access to all the programming constructs of the language, such as loops and conditional constructs. In addition, the STAR-CCM+ server exposes a certain number of classes, that it is possible to instantiate and manipulate in order to carry out some required sequence of tasks. Macros can be written from scratch, but that would require up-front knowledge about all the classes, attributes, and methods that the server exposes. Instead, it is more effective to use the workspace to record the actions to be performed and edit the Java file using a text editor in order to get the exact required effect.

The STAR-CCM+ macro that is currently used in order to perform tests and simulations on CAD aircraft parts produced by the use of JPAD is called `MultipleExecute`. It consists in a sequence of actions, performed in proper order, that allow to simulate fluid flow around the aforementioned aircraft components. As all the STAR-CCM+ macros, it extends the abstract class `StarMacro`, and it has been included into a Java project having STAR-CCM+ JARs and class folders on the build path, in order for the STAR-CCM+ macros contained in this package to actually import and use STAR-CCM+ classes. For the same reason, also the path to the `MacroExtras` package has been included in the build path. The `MultipleExecute` macro consists of several void methods (i.e., methods that do not return anything directly), each one devoted to a specific task, and some attributes, that correspond to variables that can be used by more than one single method during the execution of the macro. As for the global variables, they are the following.

- **`dataFolderPath`** - A `String` variable that stands for the path to the folder in which all the files the simulation needs (the XML data file and the CAD files in STEP format) have been stored.

- **`simFolderPath`** - A `String` variable that stores the path to the directory in which the user wants to save the STAR-CCM+ simulation files. Both this and the previous variable need to be fixed by the user, in order for the macro to retrieve and save data in the proper locations.

- **`theSimulation`** - An instance of the STAR-CCM+ Macro API class `Simulation`. It is the most important of the global variables of the macro, since it stores all the informations on the active simulation and it is used by all the methods across the class. It gets initialized at the very beginning of the macro, once the launching class has started running the software and played the macro.

- **`theOperatingConditions`, `theGeometricData`, `theSimulationParameters`** - Instances of the `MacroExtras` classes `OperatingConditions`, `GeometricData`, and `SimulationParameters` respectively. They are the variables devoted to storing all the data coming from the XML input file.

- **thePartsMap** - A `HashMap` (an implementation of the Java Map interface that permits null values and null keys) having keys of the `ComponentEnum` type and Java Lists of strings as values. It gets initialized by means of the `SimulationComponents` getter method `getComponentsMap`, so it stores the file paths of each of the components that need to be imported into the active simulation. As the global variables at the previous points, it is initialized at the beginning of the macro, by the first of the methods in which `MultipleExecute` macro has been split.

- **theAircraftParts** - A Java List containing entities of the STAR-CCM+ API class `GeometryPart`. This class, in general, manages CAD parts once they have been imported into the simulation. So `theAircraftParts` is a list containing all the imported aircraft components.

Listing 4.11 shows an overview of the macro, highlighting the attributes and the methods contained in the class. As can be seen from the listing, the macro consists of one main `execute` method, which calls all the other methods present in the class in a proper sequence. Since the macro can be used also in a stand-alone way, without launching it from JPAD but simply playing it from inside STAR-CCM+, one can think of commenting part of the tasks executed by the code, in order to have a more step-by-step approach. The following sections provide a more detailed description of some of the methods contained in the macro, especially for the ones that make greater use of Java programming constructs.

```
1  public class MultipleExecute extends StarMacro {
2
3      // Global variables
4      public static final String dataFolderPath = "C:\\Users\\...";
5      public static final String simFolderPath = "C:\\Users\\...";
6      public static Simulation theSimulation;
7      public static OperatingConditions theOperatingConditions;
8      public static GeometricData theGeometricData;
9      public static SimulationParameters theSimulationParameters;
10     public static HashMap<ComponentEnum, List<String>> thePartsMap;
11     public static List<GeometryPart> theAircraftParts;
12
13     // Macro main method
14     public void execute() {
15         initializeSimulation();
16         importCadParts();
17         createFluidDomain();
18         assignPartsToRegion();
19         createReferenceValuesReports();
20         createDirectionFieldFunctions();
21         createMesh();
22         physicsSetup();
23         createAerodynamicsCoefficients();
24         createBoundaryConditions();
25         runSimulation();
26         saveSimulation();
27         killSimulation();
28     }
```

```
29
30      // Macro methods
31      public void initializeSimulation() {
32      ...
33      }
34
35      ...
36
37      public void killSimulation() {
38      ...
39      }
40  }
```

Listing 4.11 `MultipleExecute` macro overview

`initializeSimulation`

The first of the macro methods has the fundamental task of populating global variables such as the ones related to the operating conditions, the geometric data, the simulation parameters, and the simulation components CAD files. The method filters the files contained in the directory indicated by `dataFolderPath`, distinguishing between STEP files and XML data file, and creates two distinct `File` arrays, one dedicated to the CAD components, the other one for the simulation data. The latter is used by the `DataReader` class in order to initialize the global variables: `theOperatingConditions`, `theGeometricData`, and `theSimulationParameters`. The other `File` array is used by the `SimulationComponents` class in order to generate a Map to the files containing all the different CAD components (`thePartsMap`).

`importCADParts`

Once the paths to the CAD files have been collected, it is necessary to import them into the active simulation. The STAR-CCM+ API class allowing this is `PartImportManager`. Once provided with the path to the actual file and the required options (such as coincidence tolerance, merging options, and tessellation density) it imports the component, which is collected into a Java List (`theAircraftParts`) after being casted to `GeometryPart`. After being imported, it is also possible to set a presentation name for the component and its surfaces, according to the name of the file from which it has been extracted. This is really helpful, since it allows in the following to get from the simulation parts list the desired part just by means of the presentation name set on it at this point. The method also automatically scales the parts according to the value set for the `OperatingConditions` `cadUnits` attribute. Since the OCCT STEP writer class saves shapes in mm by default, it is necessary to scale them properly.

```
1  public void importCadParts() {
2
3      LabCoordinateSystem labCoordinateSystem =
4          theSimulation.getCoordinateSystemManager().getLabCoordinateSystem();
5      PartImportManager partImportManager =
6          theSimulation.get(PartImportManager.class);
7
```

```
8       int partIndex = 0;
9       List<GeometryPart> aircraftParts = new ArrayList<>();
10      for(Iterator<ComponentEnum> comp =
11          thePartsMap.keySet().iterator(); comp.hasNext(); ) {
12
13        // Collect simulation parts into a list
14        ComponentEnum component = comp.next();
15        for(String path : thePartsMap.get(component)) {
16          partImportManager.importCadPart(
17              resolvePath(path),
18              "SharpEdges", 30.0, 2, true, 1.0E-5, true, false, false, false
19              );
20          String partName = MyStringUtils.getFilenameFromFilepath(path);
21          GeometryPart aircraftPart =
22              ((List<GeometryPart>) partImportManager .getParts()).get(partIndex);
23
24          // Assign a proper name to the imported part and its surfaces
25          aircraftPart.setPresentationName(partName);
26          ((List<PartSurface>) aircraftPart.getPartSurfaces())
27                      .get(0)
28                      .setPresentationName(partName);
29
30          // Scale parts whether necessary
31          if(theGeometricData.getCADUnits().equals("mm")) {
32            partImportManager.scaleParts(
33                Collections.singletonList(aircraftPart),
34                new DoubleVector(new double[] {1000, 1000, 1000}),
35                labCoordinateSystem
36                );
37          }
38          aircraftParts.add(aircraftPart);
39          partIndex++;
40        }
41      }
42
43      // Initialize theAircraftParts global variable
44      theAircraftParts = aircraftParts;
45    }
```

Listing 4.12 `importCADParts` method

`createFluidDomain`

The next step consists in creating the fluid domain in which the simulation takes place. For this purpose, a new geometry part gets created. This part is a simple block, whose dimensions are automatically set to be multiples of one of the aircraft reference length. In particular, the code set this reference length to be the greatest between the fuselage lenght and the wing span. Besides, depending on the value of the `SimulationParameters` `isSymmetrical` attribute, the block domain extends from the symmetry plane of the aircraft to both (positive and negative) y directions or just the positive one. In addition, the front extension of the block is set to be lower than the back one, in order to let the fluid settle after having swept the aircraft. Once the

block has been created, it's time to split its surface, in order to correctly assign the boundary conditions later in the code. The way the block surface gets split depends on:

- whether the simulation to be performed is symmetrical or not,
- whether the simulation involves an inviscid flow or not,

with the latter condition depending on the value of the `SimulationParameters simulationType` attribute. Listing 4.13 shows how the block surface gets split and how the different surfaces obtained through the process get named according to the boundary conditions to be applied onto them later. The final operation performed by the method consists in creating the actual fluid domain by means of a Boolean subtraction involving the block and the aircraft parts. The result of this operation is a new geometry part (called simply FLUID) and it is the one that it is actually used in the rest of the macro and on which operations, such as meshing and boundary conditions assignment, are performed. Its surfaces preserve the names assigned, during the previous steps, to the surfaces of the parts that have been used to generate it (i.e., the block domain and the aircraft parts), so that it is easy to retrieve them by means of the same.

```
1  // Get the block surface in order to split it
2  PartSurface simpleBlockPartSurfaces =
3      (PartSurface) simpleBlockPart.getPartSurfaceManager()
4                    .getPartSurface("Block Surface");
5
6  // Split the block surface according to the simulation requirements
7  if(!theSimulationParameters.isSimulationSymmetrical()) {
8
9      // In case the simulation to be performed is not symmetrical
10     if(theSimulationParameters.getSimulationType().equals(SimulationType.EULER)) {
11
12         // In case of inviscid fluid, the back surface of the fluid block is
13         // named OUTLET, while all the remaining ones are named INLET
14         simpleBlockPart.splitPartSurfaceByPatch(
15           simpleBlockPartSurfaces, new IntVector(new int[] {6}), "OUTLET");
16         simpleBlockPartSurfaces.setPresentationName("INLET");
17     } else
18
19         // In case of viscid fluid, all the block surfaces get named FAR-FIELD
20         simpleBlockPartSurfaces.setPresentationName("FAR-FIELD");
21 } else {
22
23     // In case the simulation to be performed is symmetrical
24     if(theSimulationParameters.getSimulationType().equals(SimulationType.EULER)) {
25
26         // In case of inviscid fluid, the back surface of the fluid block is named
27         // OUTLET, the symmetry one is simply named SYMMETRY PLANE, and
28         // the remaining ones are named INLET
29         simpleBlockPart.splitPartSurfaceByPatch(
30           simpleBlockPartSurfaces, new IntVector(new int[] {6}), "OUTLET");
31         simpleBlockPart.splitPartSurfaceByPatch(
32           simpleBlockPartSurfaces, new IntVector(new int[] {4}), "SYMMETRY PLANE");
33         simpleBlockPartSurfaces.setPresentationName("INLET");
```

```
34      } else {
35
36          // In case of viscid fluid, the block symmetry surface is named SYMMETRY
37          // PLANE, while the remaining ones are named FAR-FIELD
38          simpleBlockPart.splitPartSurfaceByPatch(
39            simpleBlockPartSurfaces, new IntVector(new int[] {4}), "SYMMETRY PLANE");
40          simpleBlockPartSurfaces.setPresentationName("FAR-FIELD");
41      }
42  }
```

Listing 4.13 Fluid domain outer surfaces creation and naming operations

`createReferenceValuesReports` - `createDirectionFieldFunctions`

The `createReferenceValuesReports` method generates several instances of the STAR-CCM+ Macro API `ExpressionReport` class, used to represent constant values that can be used throughout the macro in order to define other reports (such as those relative to aerodynamic coefficients) and to generate user defined field functions. These reports, in particular, get defined by means of the attributes of the simulation data classes `OperatingConditions` and `GeometricData`, with the latter being used in order to define reference lengths and surfaces. As for the user defined field functions mentioned above, they are produced by the `createDirectionFieldFunctions` method and consist in directions that help defining aerodynamic coefficients. In particular, they identify the lift, drag, and side force directions by means of the angle of attack and the sideslip angle specified in the operating conditions. Both the expression reports and the direction field functions are given a specific presentation name, in order to easily retrieve them later by using it. For the sake of completeness, listing 4.14 provides a sample of how direction field functions are created.

```
1  // Instantiate a new field function
2  UserFieldFunction dragDirectionFieldFunction =
3      theSimulation.getFieldFunctionManager().createFieldFunction();
4
5  // Set the field function to VECTOR type
6  dragDirectionFieldFunction.getTypeOption()
7            .setSelected(FieldFunctionTypeOption.Type.VECTOR);
8
9  // Define the direction field function by means of its components, using
10 // the expression reports for the angle of attack and the sideslip angle
11 dragDirectionFieldFunction.setDefinition(
12   "[cos(${AngleOfAttack(deg)Report}/57.3)*cos(${SideslipAngle(deg)Report}/57.3)," +
13   "sin(${SideslipAngle(deg)Report}/57.3)," +
14   "sin(${AngleOfAttack(deg)Report}/57.3)*cos(${SideslipAngle(deg)Report}/57.3)]"
15   );
16
17 // Set an appropriate presentation name
18 dragDirectionFieldFunction.setFunctionName("DragDirection");
19 dragDirectionFieldFunction.setPresentationName("DragDirection");
```

Listing 4.14 Drag direction field function

createMesh

The createMesh method generates the discretized representation of the computational domain, which the physics solvers use to provide a numerical solution. The meshing strategy adopted by the MultipleExecute macro is parts-based, which is a pretty common strategy for unstructured meshing: it detaches the meshing from the physics and provides a flexible and repeatable meshing pipeline. The first thing the method does is instantiating a new AutoMeshOperation object, with AutoMeshOperation being the STAR-CCM+ class that supervises meshing operations. The way this object is instantited depends on the type of simulation to be performed, whether it involves viscid or inviscid fluid flow. So the code contains an *if* statement, whose condition is based on the SimulationParameters simulationType attribute. If the type of the simulation has been set to SimulationType.EULER, the code selects the following options for the AutoMeshOperation object.

- **Surface Remesher** - Remeshes the initial surface to provide a quality discretized mesh that is suitable for CFD.
- **Automatic Surface Repair** - Provides an automatic procedure for correcting a range of geometric problems that can exist in the remeshed surface once the surface remeshing process is complete.
- **Polyhedral Mesher** - Generates a volume mesh that is composed of polyhedral-shaped cells.

In addition, when the simulationType attribute is set to SimulationType.VISCOUS, the **Prism Layer Mesher** option gets activated. It adds prismatic cell layers next to wall boundaries, in order to capture viscous gradients at the wall. Once the AutoMeshOperation object has been created and associated to the fluid domain part generated by the createFluidDomain method, it is possible to set values for some of its parameters. The macro currently sets values for the following properties.

- **Base Size** - A characteristic dimension of the model that can be set before using any relative values. As for the MultipleExecute macro, this value has been set equal to the greater between the fuselage length and the wing span.
- **Growth Factor** - A property that defines the growth of the mesh size, whose value should be set between 1.0 and 2.0. Increasing its value results in larger mesh sizes in the transition from boundary size to the size away from the boundary. The macro currently sets its value to 1.3.

In case of viscid fluid flow, the macro also automatically sets values regarding the Prism Layer Mesher. In particular, it sets the number of prism layers (currently fixed to 10) and the absolute prism layer thickness, to calculate the value of which the macro uses the formula for turbulent boundary layers over a flat plate:

$$\delta \approx 0.37x/\mathrm{Re}_x^{1/5}$$

with the Reynolds number based on the reference length expression report created by the createReferenceValuesReports method (and equal to the wing mean aerodynamic chord). The

last operations performed by the method consist in generating custom mesh controls for the aircraft part surfaces the fluid domain is made up of. In general, surface custom mesh controls override any default controls for the volume mesher, allowing to refine or coarsen the mesh for part surfaces. The custom controls the macro makes use of are the following.

- **Target Surface Size** - Defines the edge length that the mesher aims for in absence of any mesh refinement. Decreasing the target surface size generates more refined and detailed surface mesh.
- **Minimum Surface Size** - Specifies the lower limit of edge lengths on the surface mesh. Decreasing the minimum surface size allows more refinement in regions where the mesher is trying to capture complex features.

The `MultipleExecute` macro specifies the controls listed above for each aircraft surface and as relative to the aforementioned base size. Besides, the code automatically detects the type of the surface the controls are being set for, and uses different values depending on which part surfaces belong to. For example, the values used for the fuselage are five times the ones used for the lifting surfaces in general (target surface size equal to 0.1 of the base size, minimum surface size equal to 0.01 of the base size), since the latter typically require much more mesh refinement.

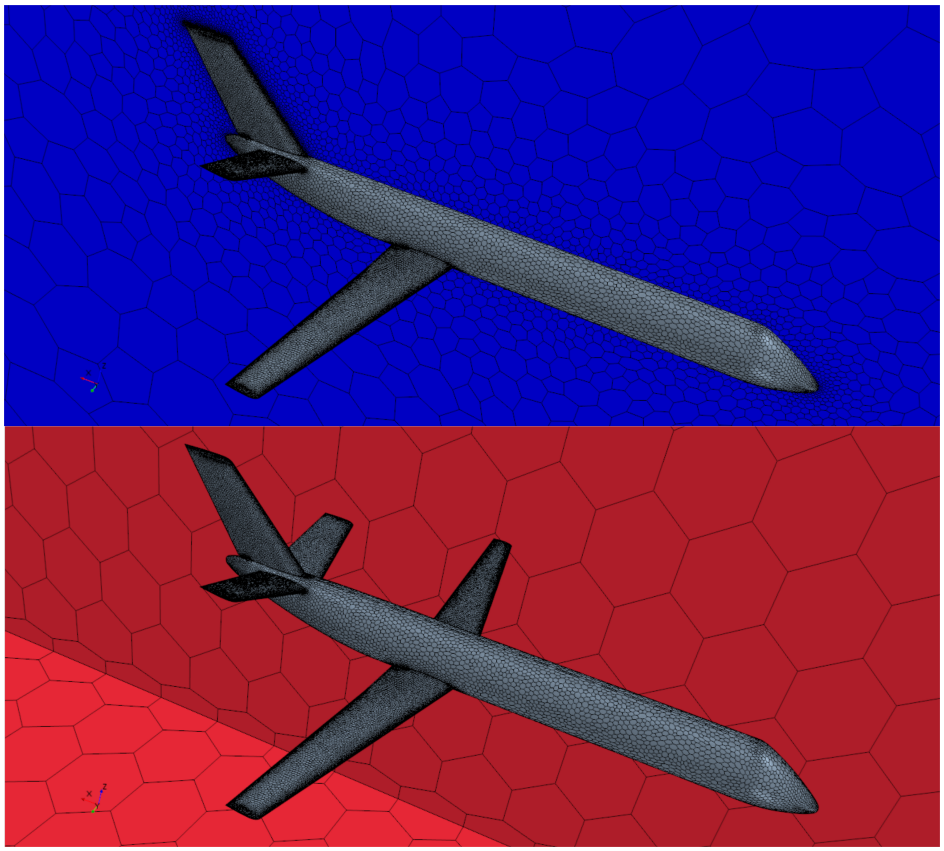

Figure 4.3 Automated meshes for symmetrical/non-symmetrical simulations

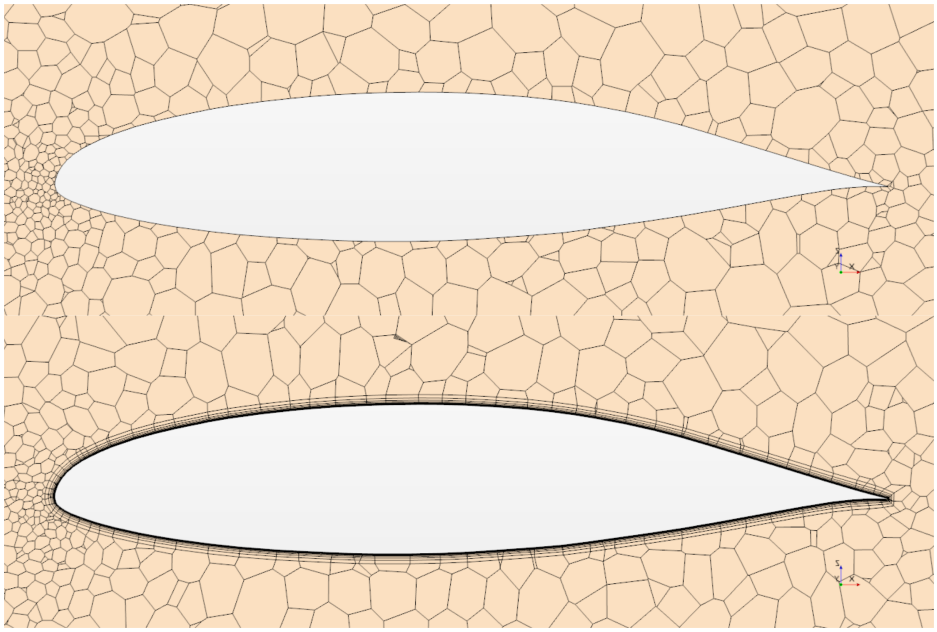

Figure 4.4 Automated meshes for non-viscous/viscous fluid flows

physicsSetup

Once the meshing operations have been defined (altough the actual mesh still needs to be generated), the macro arranges for the set up of the physics models. These define the primary variables of the simulation and what mathematical formulation is used to generate the solution. An appropriate combination of models is necessary for the complete definition of a physics continuum. After generating a new `PhysicsContinuum` object, with `PhysicsContinuum` being the class that manages physics models definition, the method enables several physics models, common to both viscid and inviscid flow simulations. Then an *if* statement allows to select different models depending on the value of the `SimulationParameters simulationType` attribute. In case the attribute is equal to `SimulationType.EULER`, the Inviscid Model is enabled. If not, several models relative to turbulence and its treatment get implemented. Listing 4.15 reports the whole method, along with the list of physics models enabled by the macro.

```
1  public void physicsSetup() {
2
3      // Generate a new instance of PhysicsContinuum
4      PhysicsContinuum physicsContinuum =
5          theSimulation.getContinuumManager().createContinuum(PhysicsContinuum.class);
6      physicsContinuum.setPresentationName("Physics");
7
8      // Enable some basic physics models
9      physicsContinuum.enable(ThreeDimensionalModel.class);
10     physicsContinuum.enable(SteadyModel.class);
11     physicsContinuum.enable(SingleComponentGasModel.class);
12     physicsContinuum.enable(CoupledFlowModel.class);
13     physicsContinuum.enable(IdealGasModel.class);
14     physicsContinuum.enable(CoupledEnergyModel.class);
15
16     // Enable more specific physics models
17     if(theSimulationParameters.getSimulationType().equals(SimulationType.VISCOUS)) {
```

```
18        physicsContinuum.enable(TurbulentModel.class);
19        physicsContinuum.enable(RansTurbulenceModel.class);
20        physicsContinuum.enable(SpalartAllmarasTurbulence.class);
21        physicsContinuum.enable(SaTurbModel.class);
22        physicsContinuum.enable(SaAllYplusWallTreatment.class);
23      } else {
24        physicsContinuum.enable(InviscidModel.class);
25      }
26
27      // Set a value for the reference pressure
28      physicsContinuum.getReferenceValues()
29            .get(ReferencePressure.class)
30            .setValue(theOperatingConditions.getPressure());
31  }
```

Listing 4.15 `physicsSetup` method

`createAerodynamicsCoefficients`

The expression reports and the direction field functions created by the methods described above are mostly used in order to define aerodynamic coefficients. `ForceCoefficientReport` and `MomentCoefficientReport` are the STAR-CCM+ classes that help creating this kind of entities. These coefficients, in order to be properly defined, need to be associated to the surfaces on which they are calculated. All the coefficients defined at this point are calculated on the whole aircraft. Listing 4.16 shows how the lift coefficient and the pitch moment coefficient reports are defined. It has to be noted that the reference surface area gets automatically adjusted in case the simulation to be performed is symmetrical. In addition to the reports for the aerodynamic coefficients, the method also provides the definitions for the pressure coefficient (by means of the `PressureCoefficientFunction` class) and, in case `simulationType` is set to `VISCOUS`, the skin friction coefficient (by means of the `SkinFrictionCoefficientFunction` class).

```
1  // Generate a new force coefficient instance
2  ForceCoefficientReport liftCoefficientReport
3      = theSimulation.getReportManager().createReport(ForceCoefficientReport.class);
4
5  // Set a direction by means of the field function created before
6  liftCoefficientReport.getDirection().setDefinition("$${LiftDirection}");
7
8  // Set reference quantities by means of the expression reports created above
9  liftCoefficientReport.getReferenceDensity().setDefinition(
10     "${DensityReport}");
11 liftCoefficientReport.getReferenceVelocity().setDefinition(
12     "${ReferenceVelocityReport}");
13 liftCoefficientReport.getReferenceArea().setDefinition(
14     "${ReferenceAreaReport}/" + Integer.toString(symmetryFactor));
15
16 // Set the surface
17 liftCoefficientReport.getParts().setObjects(aircraftBoundaries);
18
19 // Set the presentation name
```

```
20 liftCoefficientReport.setPresentationName("CL");
21
22 // Generate a new moment coefficient instance
23 MomentCoefficientReport pitchMomentCoefficientReport =
24     theSimulation.getReportManager().createReport(MomentCoefficientReport.class);
25
26 // Set a direction by using components
27 pitchMomentCoefficientReport.getDirection().setComponents(0.0, 1.0, 0.0);
28
29 // Set reference quantities by means of the expression reports created above
30 pitchMomentCoefficientReport.getReferenceDensity().setDefinition(
31     "${DensityReport}");
32 pitchMomentCoefficientReport.getReferenceVelocity().setDefinition(
33     "${ReferenceVelocityReport}");
34 pitchMomentCoefficientReport.getReferenceRadius().setDefinition(
35     "${ReferenceLengthReport}");
36 pitchMomentCoefficientReport.getReferenceArea().setDefinition(
37     "${ReferenceAreaReport}/" + Integer.toString(symmetryFactor));
38
39 // Set the pole
40 pitchMomentCoefficientReport.getOrigin().setDefinition(
41     "[${ReferenceMomentPoleReport}, 0.0, 0.0]");
42
43 // Set the surface
44 pitchMomentCoefficientReport.getParts().setObjects(aircraftBoundaries);
45
46 // Set the presentation name
47 pitchMomentCoefficientReport.setPresentationName("CM");
```

Listing 4.16 Force and moment coefficients creation

`createBoundaryConditions`

Before finally being able to run the simulation, boundary conditions must be applied. Since the fluid domain surface has already been split by one of the previous methods, at this point it's just necessary to impose proper conditions, based on the name given to the surfaces by the same method. For this purpose, `createBoundaryConditions` performs a first check on the symmetry of the problem: if so, a symmetry boundary condition is applied on the fluid domain surface previously named SYMMETRY PLANE (listing 4.13). Then, an *if* statement, whose condition depends on the `simulationType` attribute, helps the code deciding which boundary conditions should be applied on the remaining surfaces. If the simulation to be performed involves inviscid fluid, velocity inlet and pressure outlet boundary conditions are applied on the surfaces previously named INLET and OUTLET respectively. Otherwise, a free-stream condition is applied on the FAR-FIELD surfaces of the fluid domain. As for the aircraft surfaces, a wall boundary condition is applied automatically, so there's no need to modify anything. In order to set values for the quantities that it is necessary to define for each of the boundary conditions mentioned above, expression reports and direction field functions are used. Listing 4.17 shows the entire method, along with the quantities defined for the boundary conditions.

```
1  public void createBoundaryConditions() {
2
3    // Instantiate a new BoundaryManager object
4    BoundaryManager boundaryManager
5        = theSimulation.getRegionManager().getRegion("Region").getBoundaryManager();
6
7    // Checking on symmetry
8    if(theSimulationParameters.isSimulationSymmetrical()) {
9
10     // Get the symmetry surface by means of its
11     // presentation name and set its boundary type
12     boundaryManager.getBoundary("FLUID.Block.SYMMETRY PLANE")
13       .setBoundaryType(
14         (SymmetryBoundary) theSimulation.get(ConditionTypeManager.class)
15                   .get(SymmetryBoundary.class));
16   }
17
18   // Simulation type check
19   if(theSimulationParameters.getSimulationType().equals(SimulationType.EULER)) {
20
21     // In case of inviscid fluid flow
22     Boundary inflow = boundaryManager.getBoundary("FLUID.Block.INLET");
23     Boundary outflow = boundaryManager.getBoundary("FLUID.Block.OUTLET");
24
25     inflow.setBoundaryType(
26       (InletBoundary) theSimulation.get(ConditionTypeManager.class)
27                   .get(InletBoundary.class));
28     outflow.setBoundaryType(
29       (PressureBoundary) theSimulation.get(ConditionTypeManager.class)
30                   .get(PressureBoundary.class));
31
32     // Set definitions for velocity inlet BC quantities
33     inflow.getConditions().get(FlowDirectionOption.class)
34             .setSelected(FlowDirectionOption.Type.COMPONENTS);
35     inflow.getValues().get(FlowDirectionProfile.class)
36             .getMethod(ConstantVectorProfileMethod.class)
37             .getQuantity().setDefinition("$${DragDirection}");
38     inflow.getValues().get(StaticTemperatureProfile.class)
39             .getMethod(ConstantScalarProfileMethod.class)
40             .getQuantity().setDefinition("${TemperatureReport}");
41     inflow.getValues().get(VelocityMagnitudeProfile.class)
42             .getMethod(ConstantScalarProfileMethod.class)
43             .getQuantity().setDefinition("${ReferenceVelocityReport}");
44
45     // Set definitions for pressure outlet BC quantities
46     outflow.getValues().get(StaticTemperatureProfile.class)
47             .getMethod(ConstantScalarProfileMethod.class)
48             .getQuantity().setDefinition("${TemperatureReport}");
49   } else {
50
51     // In case of viscous fluid flow
52     Boundary freeStream = boundaryManager.getBoundary("FLUID.Block.FAR-FIELD");
```

```
53
54      freeStream.setBoundaryType(
55        (FreeStreamBoundary) theSimulation.get(ConditionTypeManager.class)
56                    .get(FreeStreamBoundary.class));
57
58      // Set definitions for free-stream BC quantities
59      freeStream.getValues().get(FlowDirectionProfile.class)
60              .getMethod(ConstantVectorProfileMethod.class)
61              .getQuantity().setDefinition("$${DragDirection}");
62      freeStream.getValues().get(MachNumberProfile.class)
63              .getMethod(ConstantScalarProfileMethod.class)
64              .getQuantity().setDefinition("${MachNumberReport}");
65      freeStream.getValues().get(StaticTemperatureProfile.class)
66              .getMethod(ConstantScalarProfileMethod.class)
67              .getQuantity().setDefinition("${TemperatureReport}");
68    }
69  }
```

Listing 4.17 `createBoundaryConditions` method

`runSimulation` - `saveSimulation` - `killSimulation`

The three last methods allow, respectively, to generate the mesh and run the simulation (whether the `executeMesh` attribute has been set to `TRUE`), to save the simulation with a proper name, and to kill the simulation. The last step is fundamental when playing the macro in batch mode, because it allows to properly close the active simulation, in order to start a new one or simply end the process.

4.4.2 The launching class

The last piece of the puzzle consists of the Java class which, equipped with a main method and a series of attributes used as global variables, allows to launch a simulation (or a cycle of simulations) from within JPAD. It is necessary to point out that the class described below provides only an example of how the main method should be structured for simulation purposes. In particular, the class described in the following corresponds to the one used for testing out the macro described in the previous paragraph and the system of classes provided by `MacroExtras`.

First thing, the launching class requires some attributes to be set. These attributes, corresponding to instances of the Java `String` class, are the following.

- **`workingFolderPath`** - Defines the path to the folder in which CAD files are stored, along with the XML file containing all the information about the simulation to be run.
- **`jpadCADFolder`** - Defines the path to the folder in which JPAD CAD files are generated and initially stored.
- **`macroPath`** - Contains the path of the folder in which the custom macros are stored (`MultipleExecute` is one of them).
- **`macroName`** - A string standing for the name of the macro to be run.

- **starExePath** - Defines the path to the STAR-CCM+ .exe file.
- **starOptions** - A Java `String` containing a set of option for STAR-CCM+ execution, like the path to the software license and the number of processors that must be used during the simulation.

As for the main method of the class, the first operation performed consists in loading an aircraft from XML data file. For this purpose, the `AircraftUtils.importAircraft` method is used, once the launching class run configuration has been provided with an appropiate set of arguments (see section 3.6). After the aircraft has been correctly imported, what the code makes is building a Map, called `aeroMap`, similar to the one produced by the `SimulationComponents` class: it contains `MacroExtras ComponentEnum` constants as keys to Java Lists of generic objects, which can be instances of JPAD `Fuselage` and `LiftingSurface` classes. The actual Map implementation used in the code is the `ConcurrentHashMap`, which allows concurrent modification of the Map and allows it to be modified during iteration. Listing 4.18 shows how this Map is created. It is important to note that each key is associated to a list and not to just a single entity, in accordance with the way in which the `MacroExtras` classes and the `MultipleExecute` macro have been arranged. This means that, for example, one can think of chosing one of the aircraft components present in `aeroMap`, generate a copy of the original one and modify it by means of the methods offered by the JPAD library, and finally add it to `aeroMap`, as listing 4.19 shows.

```
1  // Import aircraft data from file
2  Aircraft aircraft = AircraftUtils.importAircraft(args);
3
4  // Generate the Map and populate it with keys
5  // and empty Lists of generic Java Object entities
6  ConcurrentHashMap<ComponentEnum, List<Object>> aeroMap
7      = new ConcurrentHashMap<ComponentEnum, List<Object>>();
8  aeroMap.put(ComponentEnum.FUSELAGE, new ArrayList<Object>());
9  aeroMap.put(ComponentEnum.WING, new ArrayList<Object>());
10 aeroMap.put(ComponentEnum.CANARD, new ArrayList<Object>());
11 aeroMap.put(ComponentEnum.HORIZONTAL, new ArrayList<Object>());
12 aeroMap.put(ComponentEnum.VERTICAL, new ArrayList<Object>());
13
14 // Iterate through the Map, adding Fuselage and LiftingSurface
15 // objects to the initially empty Lists whether necessary
16 for(Iterator<ComponentEnum> comp = aeroMap.keySet().iterator(); comp.hasNext(); ) {
17
18   ComponentEnum component = comp.next();
19   switch(component) {
20
21   // Add fuselages to the Map
22   case FUSELAGE:
23     if(!(aircraft.getFuselage() == null))
24       aeroMap.get(ComponentEnum.FUSELAGE).add(aircraft.getFuselage());
25     else {
26       System.out.println("There's no FUSELAGE component for the selected aircraft");
27       aeroMap.remove(component);
28     }
29
```

```
30      break;
31
32    // Add wings to the Map
33    case WING:
34      if(!(aircraft.getWing() == null))
35        aeroMap.get(ComponentEnum.WING).add(aircraft.getWing());
36      else {
37        System.out.println("There's no WING component for the selected aircraft");
38        aeroMap.remove(component);
39      }
40
41      break;
42
43    ...
44    }
45  }
```

Listing 4.18 Aircraft Map creation

```
1   // Chose to add a custom canard, if the original aircraft has one
2   ComponentEnum customComponentEnum = ComponentEnum.CANARD;
3   if(aeroMap.containsKey(customComponentEnum)) {
4
5     // Return in case it has been decided to add a custom fuselage
6     if(customComponentEnum.equals(ComponentEnum.FUSELAGE)) return;
7
8     // Initialize the original component and the custom component
9     LiftingSurface originalComponent = (LiftingSurface) aeroMap
10                      .get(customComponentEnum).get(0);
11    LiftingSurface customComponent = null;
12
13    // Reimport the component to be modified
14    switch(customComponentEnum.name()) {
15
16    case "WING":
17      customComponent = AircraftUtils.importAircraft(args).getWing();
18      break;
19
20    ...
21
22    }
23
24    // Modify the chosen component by means of JPAD methods and criteria
25    customComponent.adjustDimensions(
26        originalComponent
27            .getAspectRatio()*1.2,
28        originalComponent.getEquivalentWing().getPanels().get(0)
29            .getChordRoot().doubleValue(SI.METER)*1.1,
30        originalComponent.getEquivalentWing().getPanels().get(0)
31            .getChordTip().doubleValue(SI.METER)*0.9,
32        originalComponent.getEquivalentWing().getPanels().get(0)
33            .getSweepLeadingEdge(),
34        originalComponent.getEquivalentWing().getPanels().get(0)
```

```
35              .getDihedral(),
36          originalComponent.getEquivalentWing().getPanels().get(0)
37              .getTwistGeometricAtTip(),
38          WingAdjustCriteriaEnum.AR_ROOTCHORD_TIPCHORD
39          );
40
41      // Set the airfoil list for the custom component
42      customComponent.setAirfoilList(originalComponent.getAirfoilList());
43
44      // Set the coordinates for the construction axes origin of the custom component
45      customComponent.setXApexConstructionAxes(
46          originalComponent.getXApexConstructionAxes()
47          .plus(Amount.valueOf(3, SI.METER)));
48      customComponent.setYApexConstructionAxes(
49          originalComponent.getYApexConstructionAxes());
50      customComponent.setZApexConstructionAxes(
51          originalComponent.getZApexConstructionAxes()
52          .plus(Amount.valueOf(0.1, SI.METER)));
53
54      // Finally add the custom component to the Map
55      aeroMap.get(customComponentEnum).add(customComponent);
56  }
```

Listing 4.19 Adding custom components to the original Map

Once all has been set on the aircraft components side, it's necessary to fix the simulation data by means of the classes provided by `MacroExtras`. For this purpose, three new instances of the classes listed in section 4.3.2 are created. As for the geometric data, the getter methods provided by the `Fuselage` and `LiftingSurface` classes are used. Besides, a check is made on the components contained in the aircraft Map: if no fuselage is present, the fuselage length attribute of the `GeometricData` class is automatically set to zero (so that the macro showed in the previous paragraph always choses the wing span as the reference length), while if no wing is present among the keys list of `aeroMap` the code flow is stopped and the main method of the STAR-CCM+ macro launching class returns nothing. As for the operating conditions, instead, the launching class makes use of two of JPAD utility classes, `AtmosphCalc` and `StdAtmos1976`, which provide methods for calculating the atmospheric properties of the International Civil Aviation Organization (ICAO) 1976 Standard Atmosphere to an arbitrary altitude. Once everything is ready, the builders of the simulation data classes provide the methods to be used in order to generate the objects to be passed to the `DataWriter` class (listing 4.20 and 4.21).

```
1  // Define the operating conditions
2  double angleOfAttack = 2.0;
3  double sideslipAngle = 0.0;
4  double machNumber = 0.64;
5  double altitude = 30000;
6  double altitudeM = Amount.valueOf(altitude, NonSI.FOOT)
7              .doubleValue(SI.METER);
8
9  StdAtmos1976 atmosphere = AtmosphereCalc.getAtmosphere(altitudeM);
10 double pressure = atmosphere.getPressure();
```

```
11 double density = atmosphere.getDensity()*1000;
12 double temperature = atmosphere.getTemperature();
13 double speedOfSound = atmosphere.getSpeedOfSound();
14 double dynamicViscosity = AtmosphereCalc.getDynamicViscosity(altitudeM);
15 double velocity = speedOfSound*machNumber;
16 double reynoldsNumber = density*velocity*wingMAC/dynamicViscosity;
17
18 // Generate the operating conditions object
19 OperatingConditions operatingConditions =
20     new OperatingConditions.OperatingConditionsBuilder()
21         .setAngleOfAttack(angleOfAttack)
22         .setSideslipAngle(sideslipAngle)
23         ...
24         .setVelocity(velocity)
25         .build();
```

Listing 4.20 Instantiate a new `OperatingConditions` object

```
1 // Create the data file writer
2 DataWriter writer = new DataWriter(
3     operatingConditions,
4     geometricData,
5     simulationParameters
6     );
```

Listing 4.21 Instantiate a new `DataWriter` object

The XML data file is not the only file that the launching class needs to generate. By means of the utilities listed in Chapter 3, it also generates the CAD files, in STEP format, containing the solid counterparts of the components listed in `aeroMap`. What the launching class does, in particular, is to iterate through the keys of the aircraft components Map, chosing for each key the right method and the proper options to be set for it, depending on the type of the component to be converted to CAD file. Listing 4.22 shows an excerpt of the original code in which the operations just described are performed. It has to be noted that also the name each file is given is chosen precisely, because from it depends the right functioning of the macro, as described in the previous paragraph.

```
1  // Create aircraft CAD files
2  List<String> cadNames = new ArrayList<>();
3
4  // Iterate through aircraft components Map keys
5  for(Iterator<ComponentEnum> comp = aeroMap.keySet().iterator(); comp.hasNext(); ) {
6    ComponentEnum component = comp.next();
7
8    switch(component.name()) {
9
10   // Transfer fuselage shapes to CAD file
11   case "FUSELAGE":
12     int nFus = aeroMap.get(component).size();
13     for(int i = 0; i < nFus; i++) {
```

```
14        String cadName = (nFus > 1) ? ("FUSELAGE_" + (i + 1)) : "FUSELAGE";
15        cadNames.add(cadName);
16        AircraftUtils.getAircraftSolidFile(
17          AircraftUtils.getFuselageCAD(
18            (Fuselage) aeroMap.get(component).get(i),
19            7, 7, true, true, false),
20          cadName,
21          FileExtension.STEP);
22      }
23      break;
24
25    // Transfer wing shapes to CAD file
26    case "WING":
27      int nWng = aeroMap.get(component).size();
28      for(int i = 0; i < nWng; i++) {
29        String cadName = (nWng > 1) ? ("WING_" + (i + 1)) : "WING";
30        cadNames.add(cadName);
31        AircraftUtils.getAircraftSolidFile(
32          AircraftUtils.getLiftingSurfaceCAD(
33            (LiftingSurface) aeroMap.get(component).get(i),
34            ComponentEnum.WING, 1e-3, false, true, false),
35          cadName,
36          FileExtension.STEP);
37      }
38      break;
39    ...
40  }
```

Listing 4.22 Aircraft components CAD files generation

At this point, all the necessary preparations for the launch of the macro are almost over. What remains to be done is to make copies of the CAD files generated at the previous step and move them to the folder in which all the simulation files are stored (`workingFolderPath`). Besides, it is necessary to actually write to XML file format the simulation data. This operation can be performed by using the `write` method supplied by the `DataWriter` class. Finally, the macro can be played by running the STAR-CCM+ application in batch mode. The Java classes allowing this are `Runtime` and `Process`. By means of the former the application (i.e., the launching class) is allowed to interface with the environment in which it is running. The `getRuntime` method of this class simply returns the runtime object associated with the current Java application, which can be used in order to execute commands provided to the `Runtime` class `exec` method in the form of strings. The usage of this method produces an instance of the `Process` class, which allows to perform input from the process, perform output to the process, wait for the process to complete, check the exit status of the process, and destroy (kill) the process. Listing 4.23 shows precisely how the procedure described above has been coded.

```
1  // Copy the CAD files and move them to the working folder
2  cadNames.forEach(s -> {
3    try {
4      Files.copy(
5        Paths.get(jpadCADFolder + "\\" + s + ".step"),
```

```
6         Paths.get(workingFolderPath + "\\" + s + ".step"),
7         StandardCopyOption.REPLACE_EXISTING
8         );
9     }
10    catch (IOException e) {
11      e.printStackTrace();
12    }
13  });
14
15  // Write simulation data to XML file format
16  writer.write(workingFolderPath + "\\Data.xml");
17
18  // Run STAR-CCM+ application in batch mode
19  // and play MultipleExecute Java macro
20  try {
21    // Get the runtime object
22    Runtime runtime = Runtime.getRuntime();
23
24    // Execute commands
25    Process runTheMacro = runtime.exec(
26      "cmd /c cd\\ && cd " + macroPath + " && dir && " + // change directory
27      "\"" + starExePath + "\" " +  // run the application
28      starOptions + " " +  // set license and settings
29      "-new -batch " + macroName // start new simulation in batch mode
30      );
31
32    // Get the input from the running process and print it to console
33    BufferedReader input =
34      new BufferedReader(new InputStreamReader(runTheMacro.getInputStream()));
35
36          String line = null;
37    while((line = input.readLine()) != null) System.out.println(line);
38
39    // Exit the process once all the operations prescribed
40    // by the previous step have been completed
41    int exitVal = runTheMacro.waitFor();
42          System.out.println("Exited with error code " + exitVal);
43  }
44
45  catch(Exception e) {
46    System.out.println(e.toString());
47    e.printStackTrace();
48  }
```

Listing 4.23 Launching class final steps

The steps listed above show how to launch a single simulation starting from a dedicated class contained in JPAD, making use of the `MacroExtras` classes and a STAR-CCM+ macro specifically written for the purpose. At this point it is immediate to think that such an operation can be launched several times consecutively, employing the loop statements offered by the Java language, for the purpose of, for example, studying the contribution to the aerodynamics of the aircraft offered by lifting surfaces with different characteristics or simply positioned

in an alternative manner (and it has been seen right above, listing 4.19, how the `Fuselage` and `LiftingSurface` classes through their methods easily allow to act on aircraft components parameters). Figure 4.5 summarizes these concepts, offering an overview of how a workflow involving the JPAD library and the STAR-CCM+ analysis tools could be arranged.

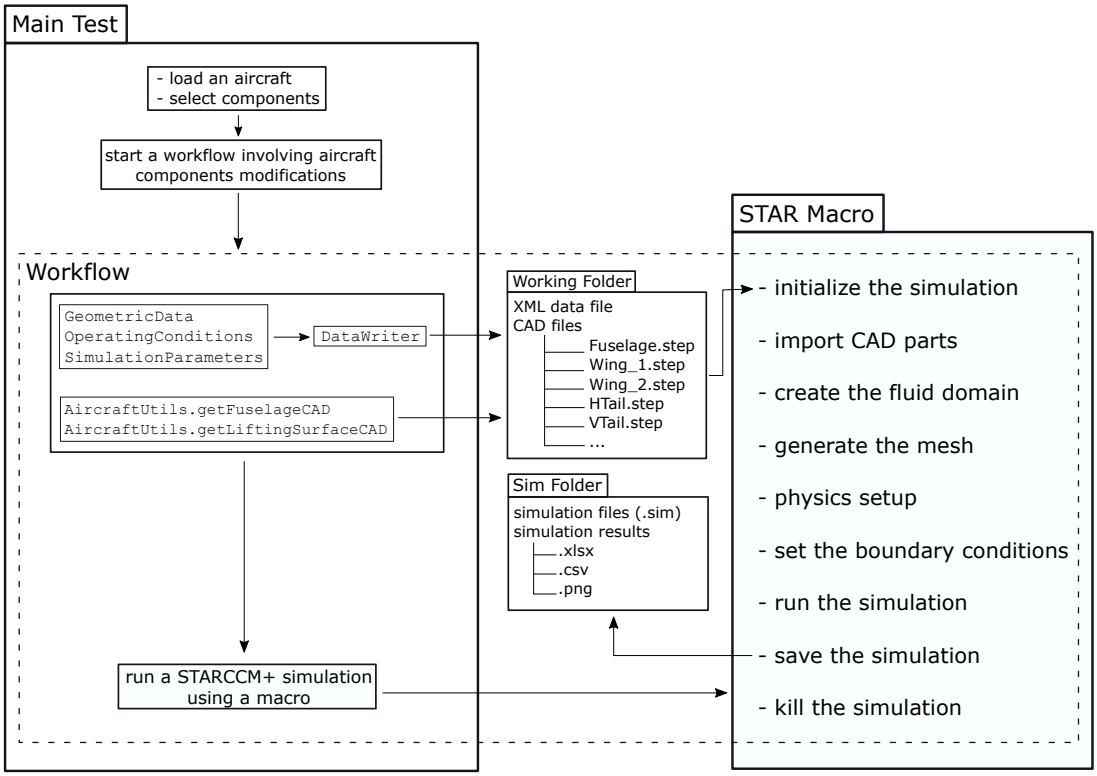

Figure 4.5 JPAD - STAR-CCM+ workflow overview

4.4.3 Interoperability example

The following tables and pictures show an example of usage of the framework described in this chapter. The simulations (involving an inviscid fluid flow) have been performed on an innovative configuration developed by the DAF research group as part of the IRON Project. The code explained in the previous chapter has been used in order to modify the aircraft configuration. More in general, CFD analyses performed on CAD components generated by means of `JPADCAD` and `AircraftUtils` have returned plausible results, and in accordance with the ones obtained for more refined CAD models. However, still more tests are needed in this sense, especially for simulations involving viscous flow.

Operating Conditions	
α	$2\,^{\circ}$
β	$0\,^{\circ}$
M	0.64
h	30000 ft
V	194,070 m s^{-1}

Table 4.4 Operating conditions for both the tests

	Geometric data
Fuselage	
Total Length	38,040 m
Cylinder Diameter	3,510 m
Wing	
Planform Surface	98,600 m^2
$\mathcal{R}$	11.96
Span	34,340 m
Mean Aerodynamic Chord	3,050 m
Twist at Root	0,000 °
Twist at Tip	−2,000 °
Horizontal Tail	
Planform Surface	39,910 m^2
$\mathcal{R}$	4.261
Span	13,040 m
Vertical Tail	
Planform Surface	24,450 m^2
$\mathcal{R}$	1.366
Span	5,780 m
Canard	
Planform Surface	10,974 m^2
$\mathcal{R}$	5.125
Span	7,500 m

Table 4.5 Aircraft data

Part	C_L	C_D	C_M
Fuselage	0.04792	0.009314	0.07679
Wing	0.56609	0.016831	-0.12323
Horizontal Tail	-0.09956	-0.002894	0.31572
Vertical Tail	0.00261	-0.001193	-0.01532
Total	0.51707	0.022058	0.25395

Table 4.6 C_L, C_D, C_M breakdown, WBH configuration

Part	C_L	C_D	C_M
Fuselage	0.04004	0.010309	0.05960
Wing	0.54700	0.017558	-0.12706
Horizontal Tail	-0.11549	-0.005141	0.36480
Vertical Tail	0.00229	-0.000894	-0.01394
Canard	0.05354	0.002412	0.28061
Total	0.52737	0.024244	0.56401

Table 4.7 C_L, C_D, C_M breakdown, WBCH configuration

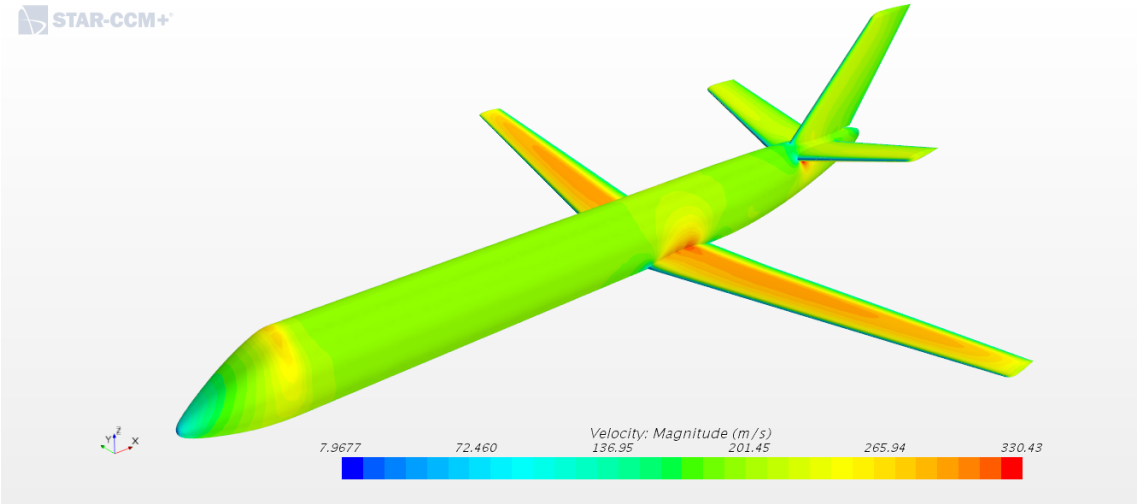

Figure 4.6 Velocity magnitude scalar scene for the no-canard configuration

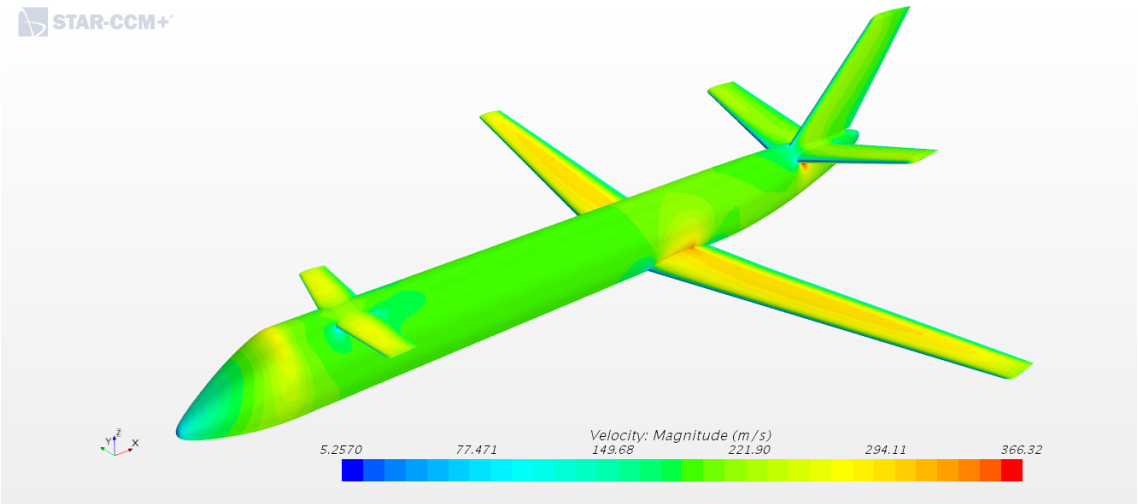

Figure 4.7 Velocity magnitude scalar scene for the canard-equipped configuration

Appendices

Appendix A

Advanced aircraft CAD modeling

In Chapter 3 the methodologies and the functions used by JPAD to model the aircraft main components have been introduced. Altough these methods help to provide a quite satisfactory representation of the aircraft, it is not enough. More effort can be put in trying to produce functions capable to model additional aircraft parts, such as engines, control surfaces and fairings. This appendix just gives an overview of the tests made in this regard, by providing a description of the adopted methodologies and images of the results. It has to be noted that the functions described in the following are still incomplete and require some work before being actually added to the JPAD library.

A.1 Wing-Fuselage fairing modeling

Several tests have been performed in order to produce a method providing a satisfactory modeling of the wing to fuselage fairing, whatever the position of one relative to the other. Currently, the method allows to generate ventral fairings (located on the underside of the fuselage, between the main wings) and wing to fuselage fairings in case of high wing, even though still more effort needs to be put on this side in order to obtain decent results. For this reason, the following descriptions and figures refer mostly to the first typology.

Since the JPAD aircraft classes do not provide a description for the fairings, all the supporting curves have been designed from scratch, generating the points the curves have to pass through (figures A.1 and A.2) by means of a set of ad hoc parameters. The fairing generator algorithm currently makes use of the following arguments:

- **`fuselage`** - An instance of the JPAD `Fuselage` class, providing all the necessary geometric data regarding the fuselage.
- **`wing`** - An instance of the JPAD `LiftingSurface` class, providing all the necessary geometric data regarding the wing, like its position with respect to the fuselage.

- **`frontLengthFactor`** - A `double` quantity fixing how much the fairing extends forward (in the negative x direction) in terms of lifting surface root chord.
- **`backLengthFactor`** - A `double` quantity fixing how much the fairing extends backward (in the positive x direction) in terms of lifting surface root chord.
- **`sideSizeFactor`** - A `double` quantity standing for how large the fairing should be compared to the fuselage diameter, and currently the code does not allow to make the fairing larger than the fuselage. Besides, in case the lifting surface is not detached from the fuselage, there is a minimum side size that the fairing must respect, in order to be sure it *wraps* the lifting surface, which is the fairing main purpose; this limit is automatically imposed by the code.
- **`fairingHeightFactor`** - A `double` quantity that fixes how much the fairing extends above or below:
  - the fuselage, in case the lifting surface is completely contained in it,
  - the wing, in case the lifting surface upper/lower surface exceeds the limits imposed by the fuselage.

  The actual height is then calculated in terms of root chord thickness.
- **`filletRadiusFactor`** - A `double` quantity that fixes the radius of the fillet to be applied to the fairing shapes. The actual radius is obtained by multiplying this factor by the distance between the points A and G shown in figure A.1, in order to make sure the fillet operation can be actually performed, otherwise the fillet algorithm, provided by the OCCT library, would encounter some problematics.

Other two factors are actually provided to the method but they are not as relevant as the aforementioned ones, or, at least, they could be fixed and excluded from the final fairing function arguments list. These factors are the **`heightBelowContactFactor`** and the **`heightAboveContactFactor`**. The contact to which they make reference is the point in which the fairing, with its side size, actually *touches* the fuselage and crosses its surface. Being the fuselage comparable to some sort of cylinder, two contacts point can be actually distinguished. In order to make sure the fairing is correctly shaped, it is necessary it doesn't exceeds some limits fixed by these contact points. Points A and G in figure A.1 are obtained by the use of these factors. In case of ventral fairing, point G z coordinate value is automatically set greater than the one of the lower contact line (how much greater depends on the value of the `heightAboveContactFactor`), but less than the one of the upper contact line, in order for the fairing to not exceeds fuselage limits. Point A z coordinate value, instead, is automatically set less than the one of the lower contact line, but greater than the fuselage lower outline one. The values of these factors are just relevant for the fillet actual radius, since the distance between the points A and G is used as a reference lenght for it.

```
1 public static List<OCCShape> getFairingShapes(
2       Fuselage fuselage,
3       LiftingSurface wing,
```

```
        double frontLengthFactor,
        double backLengthFactor,
        double sideSizeFactor,
        double fairingHeightFactor,
        double heightBelowContactFactor,
        double heightAboveContactFactor,
        double filletRadiusFactor
        ) {
        ...
}
```

Listing A.1 Fairing CAD method

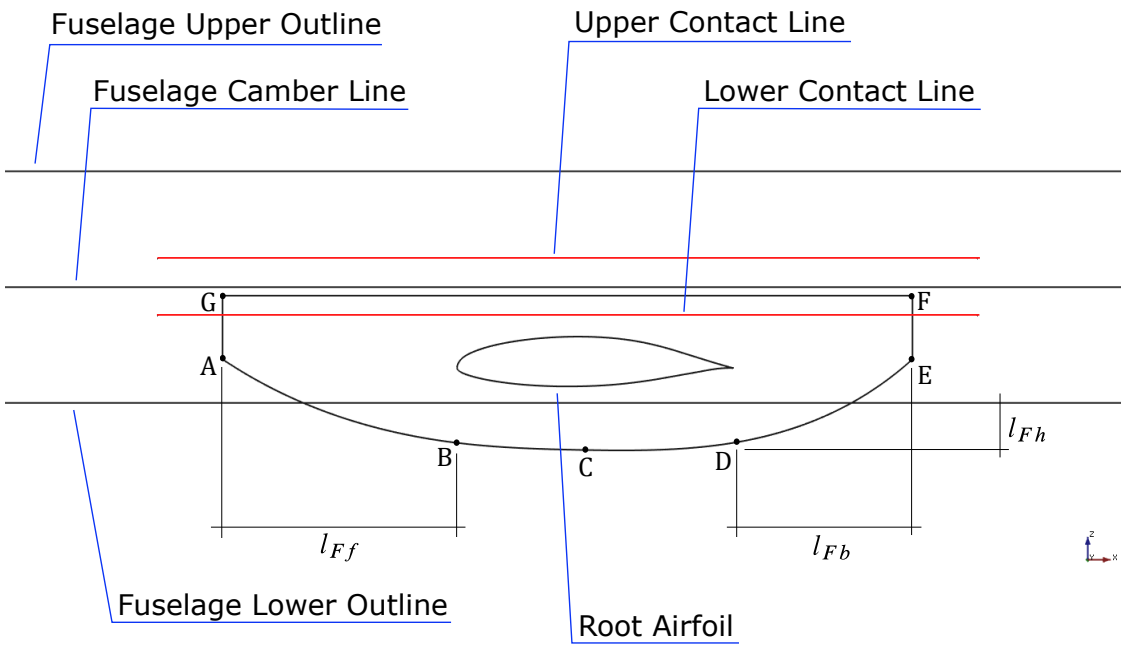

Figure A.1 Ventral fairing root profile curves

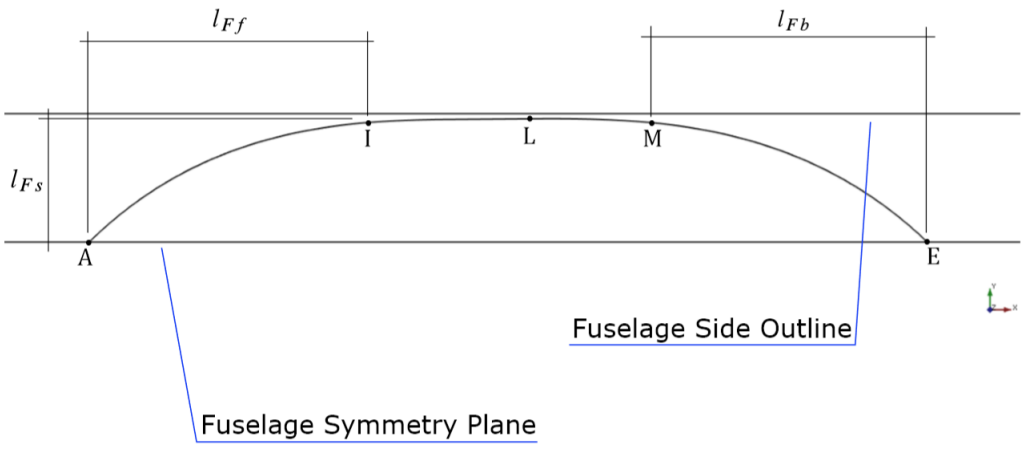

Figure A.2 Ventral fairing side curve

Length	Description
l_{Ff}	The fairing front length, obtained by multiplying the wing root chord by the `frontLengthFactor`
l_{Fb}	The fairing back length, obtained by multiplying the wing root chord by the `backLengthFactor`
l_{Fs}	The fairing side length, obtained by multiplying half the fuselage width at the wing apex x coordinate by the `sideSizeFactor`
l_{Fh}	The fairing height, obtained by multiplying the wing root airfoil thickness by the `fairingHeightFactor`

Table A.1 JPAD fairing characteristic lengths

By means of the factors listed above, and using the coordinates of the points belonging to the wing root airfoil and the fuselage outlines, the points shown in the previous figures are determined and used in order to generate the wireframe of the fairing. In particular, the side supporting curve depicted in figure A.2 serves as a reference, and is used to build C-shaped elements supporting the following construction of the surfaces belonging to the fairing. In fact, these C-shaped wires, shown in figure A.3, extend in the y direction by a quantity provided by the side curve. The edges that form the depicted wires are then used to generate the surfaces of the fairing, by means of the `OCCUtils.makePatchThruSections` method (figure A.3). It has to be noted that, at this stage, only the right shapes of the fairing are being produced, since the OCCT library provides the methods to perform the necessary transformations, as seen in Chapter 3.

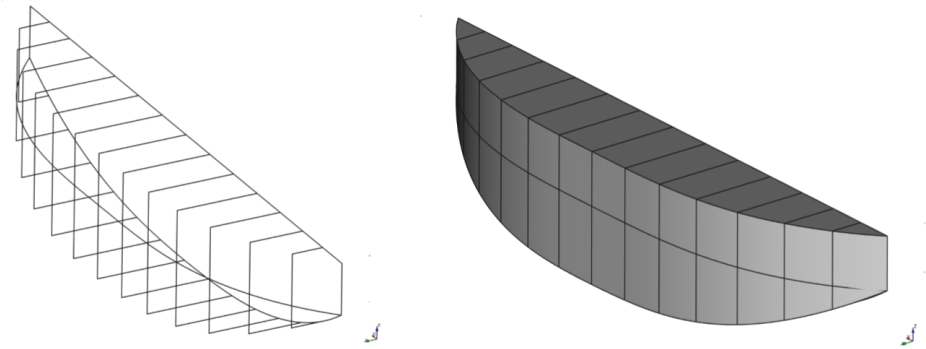

Figure A.3 Fairing complete wireframe (left) and fairing surfaces (right)

Once the patches on the right side of the fairing have been obtained, they are sewed together by means of the same OCCT class used in Chapter 3 for the same purpose. The final result is a single shell, onto which a fillet can be applied in order to smooth the edges of the fairing that actually emerge from the fuselage solid. The OCCT library provides a class for this operation, called `BRepFilletAPI_MakeFillet`, which simply requires the shape on which the operation should be performed, the edge to be smoothed out, and the radius of the fillet. Figure A.4 shows the final result, once the filleted half fairing shell has been mirrored and the solid has been produced.

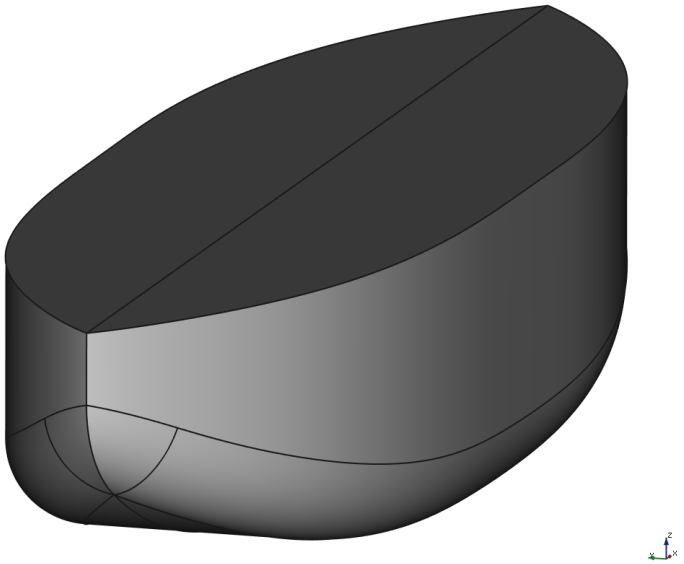

Figure A.4 Fairing solid

As mentioned before, the same algorithm can be used to generate different types of fairing. The following pictures show a business aircraft concept equipped with a canard. The function used to generate both the fairings (the one between the canard and the fuselage and the one for the main wing and the fuselage) is exactly the same, with just the values provided to the arguments of the `getFairingShapes` function being different (listing A.2).

```
1  // Import the aircraft from the XML data file and populate
2  // JPAD aircraft dedicated classes instances
3  Aircraft aircraft = AircraftUtils.importAircraft(args);
4
5  Fuselage fuselage = aircraft.getFuselage();
6  LiftingSurface wing = aircraft.getWing();
7  LiftingSurface horTail = aircraft.getHTail();
8  LiftingSurface verTail = aircraft.getVTail();
9  LiftingSurface canard = aircraft.getCanard();
10
11 // Generate CAD shapes for the aircraft components and the fairings
12 List<OCCShape> fuselageShapes = getFuselageCAD(
13     fuselage, 7, 7, true, true, false);
14 List<OCCShape> wingShapes = getLiftingSurfaceCAD(
15     wing, ComponentEnum.WING, 1e-3, false, true, false);
16 List<OCCShape> wingFairingShapes = getFairingShapes(fuselage, wing,
17     1.15, 1.15, 0.50, 0.15, 0.50, 0.50, 0.75);
18 List<OCCShape> horTailShapes = getLiftingSurfaceCAD(
19     horTail, ComponentEnum.HORIZONTAL_TAIL, 1e-3, false, true, false);
20 List<OCCShape> verTailShapes = getLiftingSurfaceCAD(
21     verTail, ComponentEnum.VERTICAL_TAIL, 1e-3, false, true, false);
22 List<OCCShape> canardShapes = getLiftingSurfaceCAD(
23     canard, ComponentEnum.CANARD, 1e-3, false, true, false);
24 List<OCCShape> canardFairingShapes = getFairingShapes(fuselage, canard,
25     1.50, 0.85, 0.55, 0.20, 0.20, 0.20, 0.75);
```

Listing A.2 Fairing CAD method application

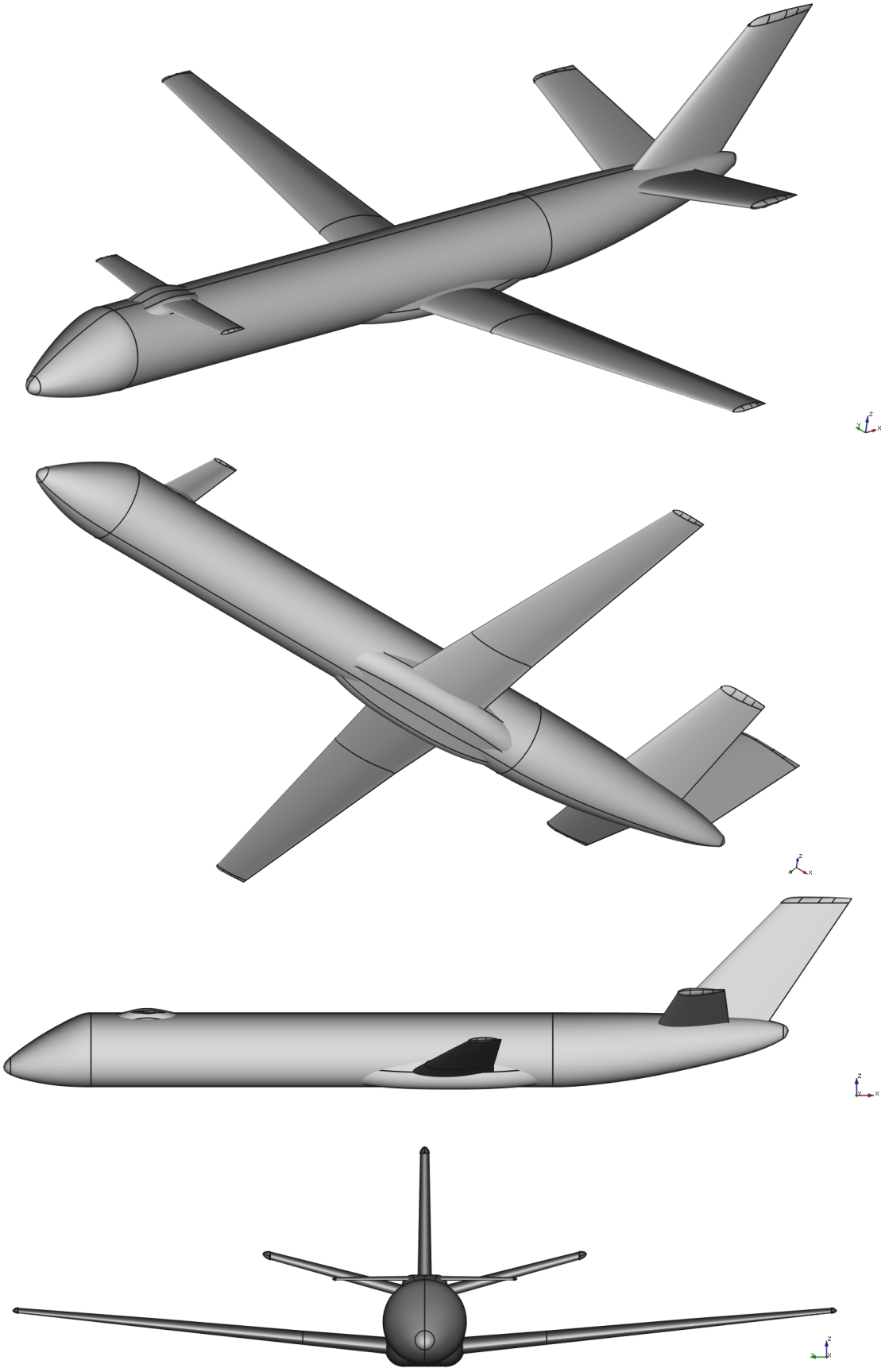

Figure A.5 Canard and main wing to fuselage fairings

A.2 Control surfaces modeling

Several tests have been performed in order to evaluate the capabilities of the Boolean operations provided by the OCCT library, for the purpose of using the same as the main tool to generate control surfaces. OCCT, in fact, comes equipped with a package dedicated to Boolean operations, which is `BRepAlgoAPI`, that contains `BRepAlgoAPI_Cut`, which is the class providing the methods to perform shell/solid cutting operations. The images and the descriptions below mostly refer to the tests made for Fowler type flaps, altough the same Java code, along with the adopted methodologies, can be easily adapted to generate other typologies of flaps and control surfaces in general.

The algorithm implemented in the testing classes first requires a solid shape to begin with, which is generated by the use of the `getLiftingSurfaceCAD` methods provided by `AircraftUtils`, and the geometric data related to the control surface to be produced, such as the breakpoints and the chords. This data is acquired by the means of the `get` methods for symmetric/antysimmetric flaps implemented by JPAD `LiftingSurface` class. Then, in order to generate the flapped wing, Boolean operations are needed in order to:

1. generate the cut wing, i.e., the wing once cut but with no flaps;
2. generate the flap itself.

These operations require the construction of some dedicated solid entities, which need to be designed from scratch using the data regarding the flaps that has been previously acquired. The first of these *cutting* solids are built by using another `BRepAlgoAPI` class, which is `BRepAlgoAPI_Section`, that produces a section operation between an argument (the wing solid) and tools (vertical planes lineary distributed along the wing). The result of this operation is a set of airfoils, which can be used in order to produce wires to patch through for the purpose of creating the wing cutting solids (figure A.6)

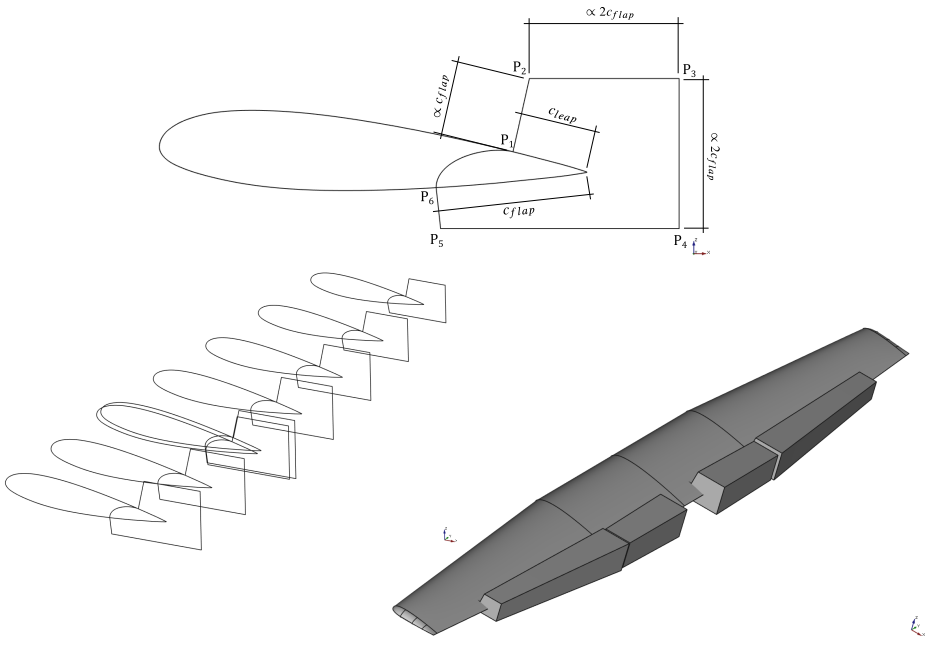

Figure A.6 Wing cutting wires (above and left) and wing cutting solids (right) once mirrored

The second set of solids for cutting operations, instead, is produced starting from one of the edges of the wire generated at the previous step, the one that actually intersects the wing shapes. Points are taken on this edge and wires are generated in a similar fashion as the wing cutting ones (figure A.7). The solids that result from the patching operation through these wires are used for the flap leading edge adjustment: in fact, the leading edge of the flap must be rounded after the cutting operations for the wing have been performed. This operation could actually be avoided by using the functionalities provided by the `BRepFilletAPI_MakeFillet` class instead, as seen for the fairing. But some problems could arise during the filleting operations in case a double edge has been generated at the flap leading edge by the Boolean cut executed on the wing. In order to prevent this possibility, the code performs the rounding of the leading edge of the flap in the aforementioned manner.

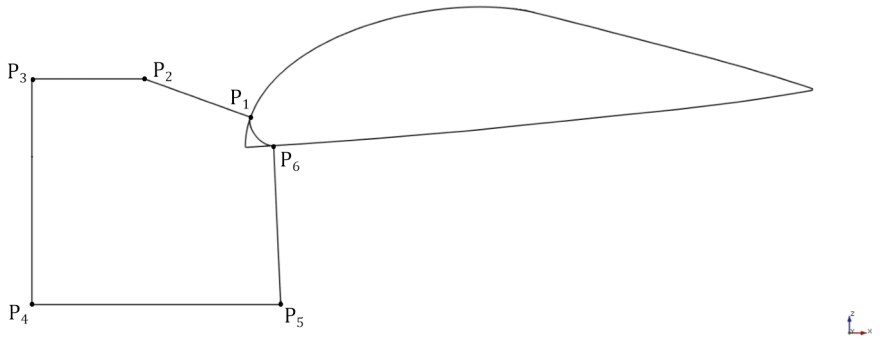

Figure A.7 Flap cutting wire

Once the supporting solids have been produced, Boolean cutting operations can be finally executed. First, cuts are performed on the wing by subtracting wing cutting solids from it one by one. At the end of each cut the argument of the Boolean operation is updated, in order to obtain the so-called cut wing (figure A.8). Turning to the flaps, auxiliary cut wings must be created first (at the center in figure A.9). These solids are generated at the same way of the previous ones, but without updating the argument of the Boolean operation, so that it stays equal to the original wing. Then the auxiliary cut wings are subtracted one by one from the wing, in order to obtain the flaps, whose leading edge is adjusted by means of the aforementioned flap cutting solids (figure A.10).

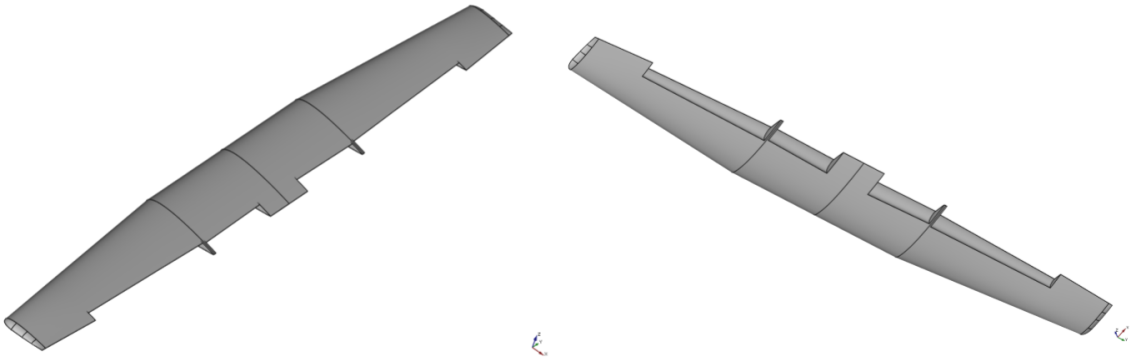

Figure A.8 Cut wing

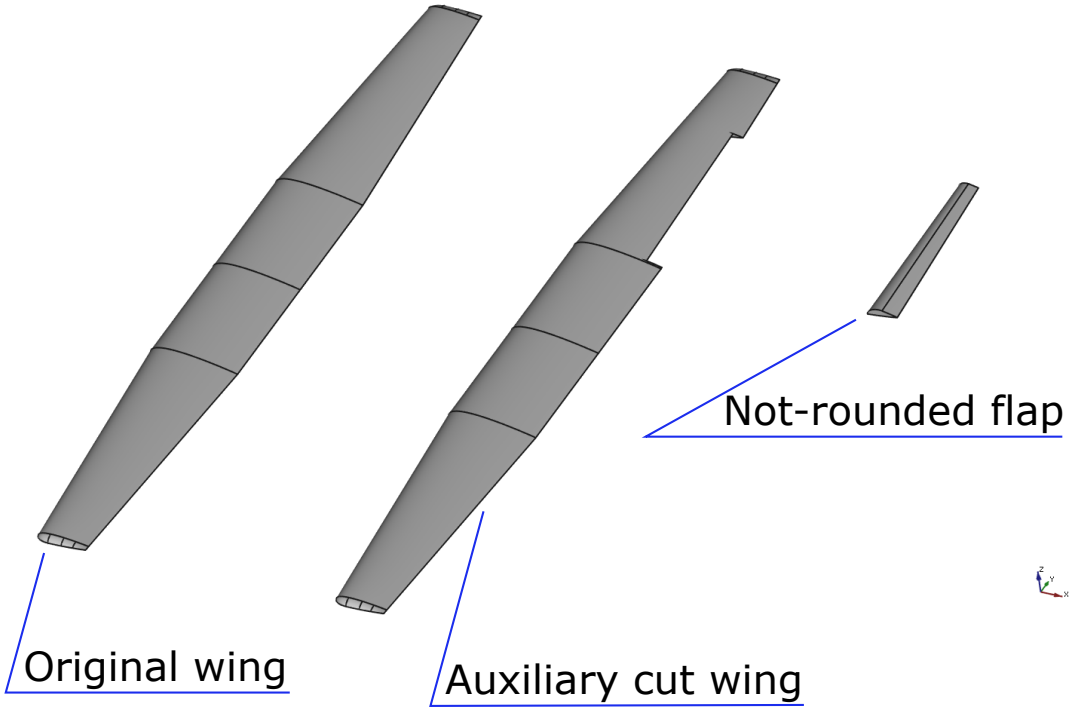

Figure A.9 Flap generation steps

Figure A.10 Flap leading edge adjustment

The final step consists in moving the flaps with respect to the cut wing. As mentioned before, the OCCT library provides also the classes to perform transformations such as translation and rotation. However, it has to be noted that the current algorithm for control surfaces motion still needs some work and adjustment, especially regarding the Fowler flaps. The following figures show the final result for different wings. In particular, figures A.11 and A.12 show the wings of the ATR-72 and CS300 respectively, equipped with inboard and outboard Fowler flaps. Figure A.13, instead, shows the horizontal and vertical stabilizers of the ATR-72, equipped with elevator and rudder treated as plain flaps.

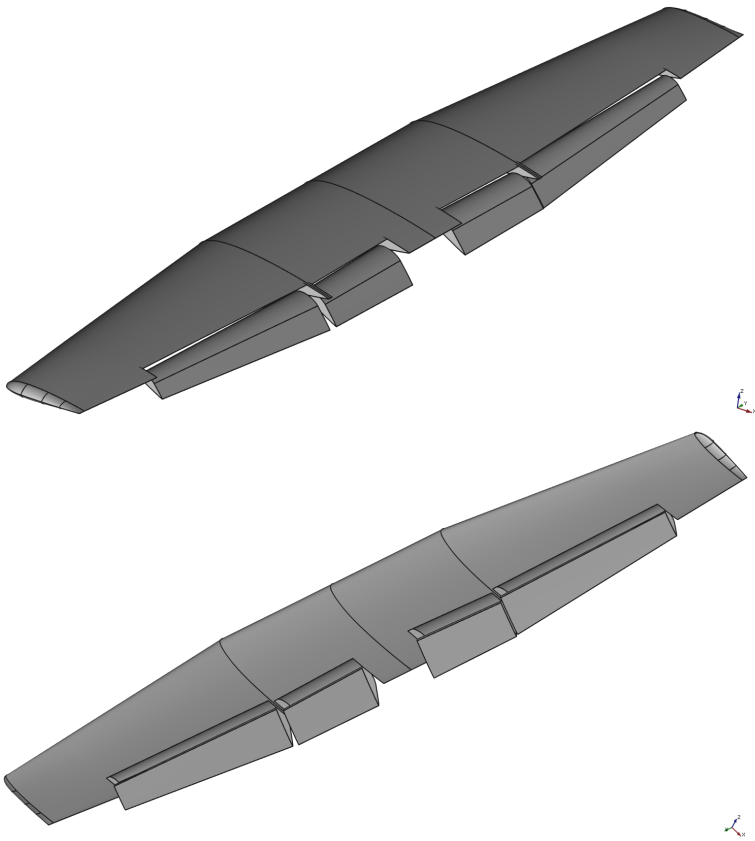

Figure A.11 ATR-72 wing, with inboard and outboard Fowler flaps

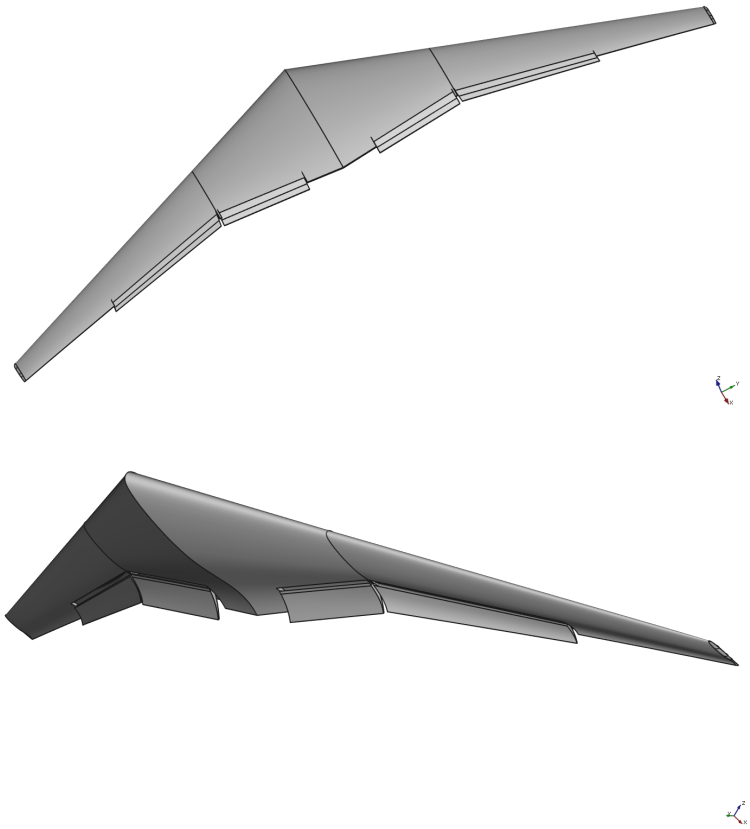

Figure A.12 CS300 wing, with inboard and outboard Fowler flaps

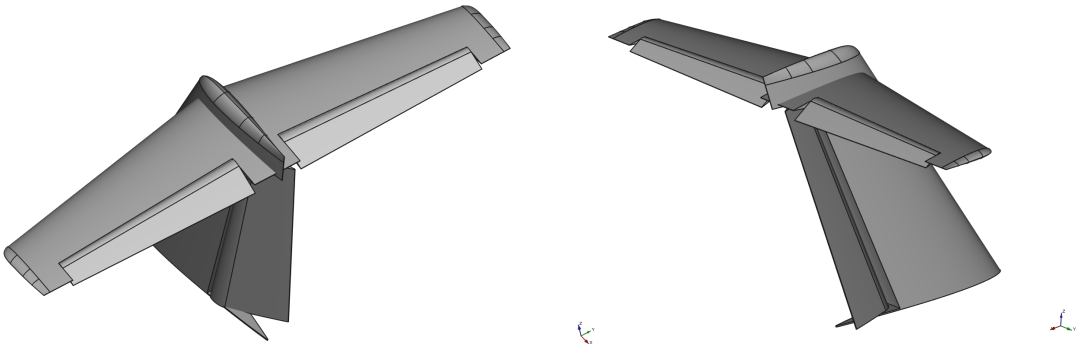

Figure A.13 ATR-72 tail, with rotated elevator and rudder

Appendix B

Importing CAD models into a JavaFX window

JPAD comes equipped with a simple yet very intuitive GUI, which provides the user with the possibility to import aircraft data in several ways and perform analyses. Altough the JPAD GUI currently offers different 2D views of the aircraft (made mostly from the outlines of each of the aircraft components), allowing the user to check the actual configuration and realize quite instantly the changes made to it after modifying some of the available parameters, it would be quite interesting and useful to give also the possibility to view the 3D model of the aircraft, built by means of the `JPADCAD` package and the utility functions contained in `AircraftUtils`.

The Java language offers the possibility to design and create applications such as a simple window for 3D models view and exploration by means of the JavaFX platform [12]. JavaFX is a Java library that can be used to build complex applications, that can run consistently across multiple platforms and on various devices, such as desktop computers, mobile phones, TVs, tablets, etc. In general, a JavaFX application has three major components, namely *Stage*, *Scene*, and *Nodes*. The stage (a window) contains all the objects of a JavaFX application, and is represented by the `Stage` class. The primary stage is automatically created by the platform itself and is passed as an argument to the `start` method of the `Application` class, the abstract class from which all the JavaFX applications extend. A scene, instead, represents the physical contents of a JavaFX application. It contains the *Scene Graph*, which is a tree-like data structure, whose root, branches, and leaves are represented by nodes. A node, in general, may include:

- geometrical objects, such as circles, rectangles, polygons, etc.;
- controls, such as buttons, check box, choice box, text area, etc.;
- containers, such as border pane, grid pane, flow pane, etc.;
- media elements, such as audio, video, and image objects.

A branch node (also called parent node) is a node with child nodes. There are several types of parent nodes, the most used of which is the *Group* one. A group node is a collective node and whenever the group node is rendered, all its child nodes are rendered in order. Any transformation applied on the group is applied to all the child nodes.

The objects that we want to import into a JavaFX window are much more complicated than the ones that can be created by using the three built-in 3D shape classes that come with JavaFX: `Cylinder`, `Box`, and `Sphere`. For more complex objects, the JavaFX library provides the user with the `TriangleMesh` class, which allows to create objects based on a connected series of triangles. In order to create the necessary `TriangleMesh` objects, which should represent the 3D shapes of the aircraft obtained by the use of `JPADCAD`, it is crucial some sort of conversion from `TopoDS_Shape` (the generic OCCT class managing all sort of shapes) to `TriangleMesh` entities. `OCCDataProvider` and `OCCFXMeshExtractor` are the `JPADCAD` classes providing the instruments for this operation. In particular, to generate a `TriangleMesh` instance three collections must be defined: one for the points, one for the faces, and one for the texture coordinates. The `OCCFXMeshExtractor` class, along with its `FaceData` sub-class (which extends the generic `OCCDataProvider` one), allows to automatically convert `TopoDS_Face` objects to `TriangleMesh` entities, by employing the face object underlying mesh data structure (handled by the OCCT `BRepMesh_IncrementalMesh` class). In this way, the aircraft 3D model (its face-type shapes) can be converted into a list of sub-lists of `TriangleMesh` entities, with each sub-list being relative to a particular aircraft component (listing B.1).

```
// Import the aircraft into the application
Aircraft theAircraft = AircraftUtils.importAircraft(args);

// Generate selected aircraft solid shapes
List<ComponentEnum> comps = new ArrayList<>();
comps = Arrays.asList(new ComponentEnum[] {
    ComponentEnum.FUSELAGE,
    ComponentEnum.WING,
    ComponentEnum.HORIZONTAL_TAIL,
    ComponentEnum.VERTICAL_TAIL,
    ComponentEnum.CANARD
});

List<OCCShape> allShapes = AircraftUtils.getAircraftShapes(theAircraft, comps);
List<TopoDS_Solid> solids = AircraftUtils.getAircraftSolid(allShapes);

// Extract the triangle meshes
List<List<TriangleMesh>> triangleMeshes = solids.stream()
  .map(s -> (new OCCFXMeshExtractor(s)).getFaces().stream()
    .map(f -> {
      FaceData faceData = new FaceData(f, true);
      faceData.load();
      return faceData.getTriangleMesh();
    })
    .collect(Collectors.toList()))
  .collect(Collectors.toList());
```

Listing B.1 Solid shapes conversion to JavaFX triangle meshes

The converted faces are then added to a group, after being given a specific color based on the aircraft surface they belong to. This group, in turn, is added to another one, which acts as a camera for the final scene. The result is shown in the following figure, which depicts four different aircrafts imported into a JavaFX window by the means of the same application. The next step, obviously, will consist in adding the so-created window to the JPAD GUI.

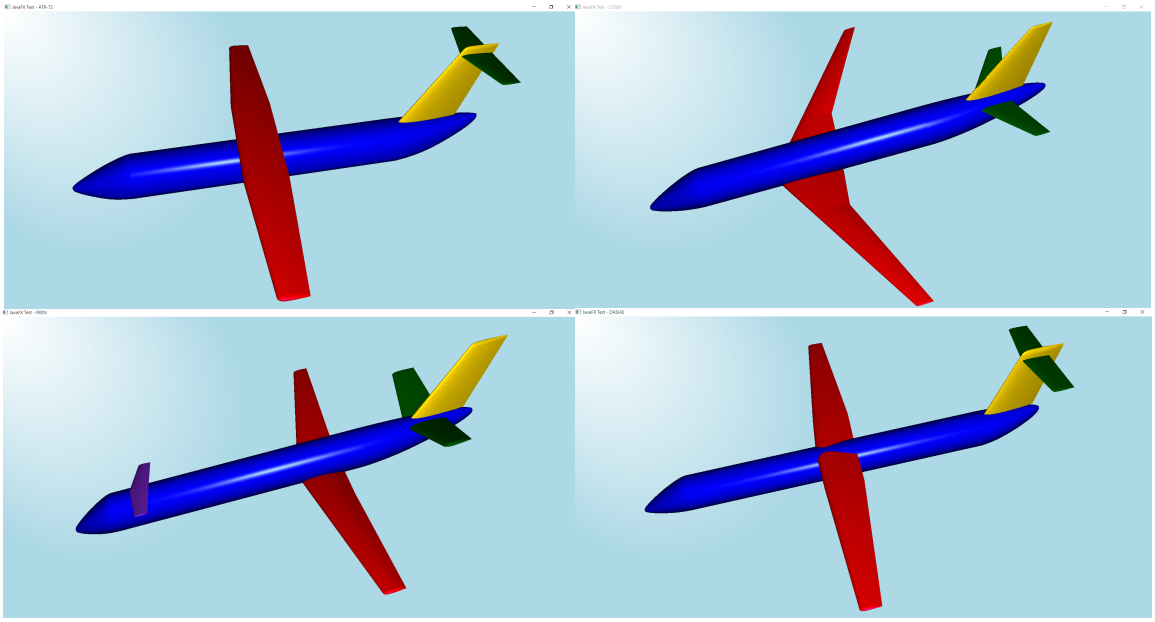

Figure B.1 `JPADCAD` aircrafts imported into the JavaFX application

Bibliography

[1] Various Authors. *Abstract Factory Pattern*. URL: https://en.wikipedia.org/wiki/Abstract_factory_pattern.

[2] Various Authors. *Builder Pattern*. URL: https://en.wikipedia.org/wiki/Builder_pattern.

[3] Various Authors. *eXtensible Markup Language (XML)*. URL: https://en.wikipedia.org/wiki/XML.

[4] Various Authors. *Hierarchical Data Format (HDF)*. URL: https://en.wikipedia.org/wiki/Hierarchical_Data_Format.

[5] Various Authors. *Open CASCADE Technology (OCCT)*. URL: https://www.opencascade.com/content/overview.

[6] Various Authors. *Processing Foundation*. URL: https://processing.org/reference/PVector.html.

[7] Various Authors. *Simplified Wrapper and Interface Generator (SWIG)*. URL: http://www.swig.org/.

[8] Various Authors. *Software Design Patterns*. URL: https://en.wikipedia.org/wiki/Software_design_pattern.

[9] Various Authors. *STAR-CCM+ v12.04 User Guide*. Siemens PLM Software, 2017.

[10] Oracle Corporation. *File*. URL: https://docs.oracle.com/javase/8/docs/api/java/io/File.html.

[11] Oracle Corporation. *Java Enum Types*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

[12] Oracle Corporation. *JavaFX*. URL: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm.

[13] Oracle Corporation. *List*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/List.html.

[14] Oracle Corporation. *Map*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html.

[15] JScience Group. *Amount*. URL: http://jscience.org/api/org/jscience/physics/amount/Amount.html.

[16] R. Haimes and M. Drela. *On The Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design*. Nashville, Tennessee: 50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition.

[17] K. Kawaguchi. *Args4j*. URL: http://args4j.kohsuke.org/.

[18] G.K.W. Kenway, G.J. Kennedy, and J.R.R.A. Martins. *A CAD-Free Approach to High-Fidelity Aerostructural Optimization*. Fort Worth, Texas: 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference.

[19] L.M. Nicolai and G. Carichner. *Fundamentals of Aircraft and Airship Design*. AIAA education series. American Institute of Aeronautics and Astronautics, 2010.

[20] A. Ronzheimer. *CAD in Aerodynamic Aircraft Design*. Braunschweig, Germany: Deutscher Luft- und Raumfahrtkongress, 2017.

[21] J. Roskam. *Airplane Design*. Airplane Design pt. 1-6. DARcorporation, 2000.

[22] Aerospace Design Lab at Stanford University. *SUAVE - An Aerospace Vehicle Environment for Designing Future Aircraft*. URL: http://suave.stanford.edu/.

[23] E. Torenbeek. *Advanced Aircraft Design*. Wiley, 2013.

GLOSSARY

Boundary Representation In solid modeling and computer-aided design, boundary representation (BRep) is a method for representing shapes using the limits. A solid is represented as a collection of connected surface elements, the boundary between solid and non-solid. Boundary representation of models are composed of two parts: topology and geometry (points, curves and surfaces).

Computational Fluid Dynamics Computational fluid dynamics (CFD) is a branch of fluid mechanics tha uses numerical analysis and data structures to solve and analyze problems tha involve fluid flows. Computers are used to perform the calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. With high-speed supercomputers better solutions can be achieved.

Computer-Aided Design Computer-aided design (CAD) is the use of computer systems to aid in the creation, modification, analysis or optimization of a design. CAD software is used to increase the productivity of the designer, improve the quality of design, improve communications through documentation, and to create a database for manifacturing.

Computer-aided Engineering Computer-aided engineering (CAE) is the use of computer software to simulate performance in order to improve product designs or assist in the resolution of engineering problems for a wide range of industries. This includes simulation, validation and optimization of products, processes, and manufacturing tools. CAE areas covered include: stress analysis (FEA), thermal and fluid flow analysis (CFD), multibody dynamics and kinematics..

Constructive Solid Geometry Constructive solid geometry (CSG) (formerly called computational binary solid geometry) is a tecnique used in solid modeling which allows a modeler to create a complex surface or object by using boolean operators to combine simpler objects.

Design Pattern In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Direct Operative Cost The totality of aircraft costs directly connected to the aircraft flight. It can be seen as the amount of money necessary to carry 1 ton of payload upon 1 km.

File A Java class representing files and directory path names in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc. The File object created by means of the File class represents the actual file/directory on the disk.

Finite Element Method Finite element method (FEM) is a numerical method for solving problems of engineering and mathematical physics. Typical problem areas of interest include structural analysis, heat transfer, fluid flow, mass transport, and electromagnetic potential.

Integrated Development Environment An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools and a debugger. Most modern IDEs have an intelligent code completion. Some IDEs contain a compiler, interpreter, or both, such as NetBeans and Eclipse; others do not, such as SharpDevelop and Lazarus. The boundary between an integrated development environment and other parts of the broader software development environment is not well-defined. Sometimes a version control system, or various tools to simplify the construction of a Graphical User Interface (GUI), are integrated. Many modern IDEs also have a class browser, an object browser, and a class hierarchy diagram, for use in object-oriented software development.

Java ARchive A JAR (Java ARchive) is a package file format typically used to aggregate many Java class files and associated metadata and resources into one file for distribution. They are built on the ZIP format and typically have a .jar file extension.

Java Program toolchain for Aircraft Design Collection of libraries and classes with the aim of providing complete aircraft preliminary design analyses through the use of several semi-empirical formulas tested against experimental data.

List A Java interface, extending the Collection one, that gives users the possibility to generate ordered collections. The user of this interface has control over where in the list each element is inserted. Elements of the list can be accessed using their integer index (i.e., their position in the list). It also allows the user to search for elements in the list.

Manifold Geometry Manifold is a geometric topology term meaning that disjoint lumps are allowed to exist in a single logical body. On the other hand, non-manifold means that all disjoint lumps must be their own logical body. In practice, manifold essentially means manufacturable, while non-manifold means non-manufacturable. Disconnected vertices and edges, internal faces, and areas with no thickness, are all examples of non-manifold geometry.

Map A Map is an object that maps keys to values. Each key and value pair is known as an entry. Its is extremely useful in case there is need to search, update or delete elements on the basis of key. The Java general Map interface doesn't allow duplicate keys, but it is possible to have duplicate values. Some implementations of the Map interface allows null keys and values.

Multi-disciplinary Design Optimization Multi-disciplinary design optimization (MDO) is a field of engineering that uses optimization methods to solve design problems incorporating a number of disciplines. MDO allows designers to incorporate all relevant disciplines simultaneously. The optimum of the simultaneous problem is superior to the design found by optimizing each discipline sequentially, since it can exploit the interactions between the disciplines. However, including all disciplines simultaneously significantly increases the complexity of the problem.

Open CASCADE Technology Open CASCADE Technology (OCCT) is an open-source software development platform for 3D CAD, CAM, CAE, etc. that is developed and supported by Open CASCADE SAS.

Simplified Wrapper and Interface Generator The Simplified Wrapper and Interface Generator (SWIG) is an open-source software tool used to connect computer programs or libraries written in C or C++ with languages such as Java.

Acronyms

AOA angle of attack.

API Application Programming Interface.

BRep Boundary Representation.

CAD Computer-Aided Design.

CAE Computer-aided Engineering.

CFD Computational Fluid Dynamics.

CSG Constructive Solid Geometry.

DOC Direct Operative Cost.

FEM Finite Element Method.

FFD Free-Form Deformation.

GUI Graphical User Interface.

ICAO International Civil Aviation Organization.

IDE Integrated Development Environment.

ISA International Standard Atmosphere.

JAR Java ARchive.

JPAD Java Program toolchain for Aircraft Design.

MDO Multi-disciplinary Design Optimization.

NURBS Non-Uniform Rational B-Spline.

OCCT Open CASCADE Technology.

OML Outer Mold-Line.

SDK Software Development Kit.

SWIG Simplified Wrapper and Interface Generator.

UML Unified Modeling Language.