Institute of Aircraft Design
Lehrstuhl für Luftfahrtsysteme

Point Cloud Extractor - Manual

3D-Scanning Toolchain

Institute of Aircraft Design

of the Technical University of Munich

Supervised by	Prof. Dr.-Ing. Mirko Hornung
	Sebastian Köberle, Dipl-Ing.
	Institute of Aircraft Design
Authors:	Lina van Brügge

Version number	0.1

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbrevations

ADDAM	Aircraft Design Data Model	$[-]$
ADEBO	Aircraft Design Box	$[-]$
AoA	Angle of Attack	$[rad]$
AVL	Athena Vortex Lattice	$[-]$
COTS	Commercial Off-The-Shelf	$[-]$
GSL	GNU Scientific Library	$[-]$
GUI	Graphical User Interface	$[-]$
IMPULLS	Innovative Modular Payload UAV - TUM LLS	$[-]$
MLS	Moving Least Square	$[-]$
OOBB	Object Oriented Bounding Box	$[-]$
PCD	Point Cloud Data	$[-]$
PCL	Point Cloud Library	$[-]$
SFML	Simple and Fast Multimedia Library	$[-]$
UAV	Unmanned Aerial Vehicle	$[-]$

Greek Symbols

Γ	dihedral	$[rad]$
Λ	sweep	$[rad]$
Θ	twist	$[rad]$

Latin Symbols

α	angle of attack	$[-]$
c	chord length	$[]$
c_d	drag coefficient	$[-]$
c_l	lift coefficient	$[-]$

1 Introduction

This document serves as a short summary of all functions implemented in the point cloud extractor of the Institute of Aircraft Design of the Technical University of Munich. It should give anyone the opportunity to implement further functionalities. Until now, airfoils and fuselages of a complete aircraft can be extracted as well as pre-processed morphing wing structures and propellers. For the complete aircraft extraction process and the morphing wing extraction there are already pre-implemented procedures. For further information on the usage of these, please refer to the read.me in the repository. For detailed information on the scanning process, refer to van Brügge et al. (2021). However, the scanning resolution should be increased in contrast to van Brügge et al. (2021) for being also able to extract the wing tip with a good accuracy. The complete toolchain is shown in Fig. 1.1.

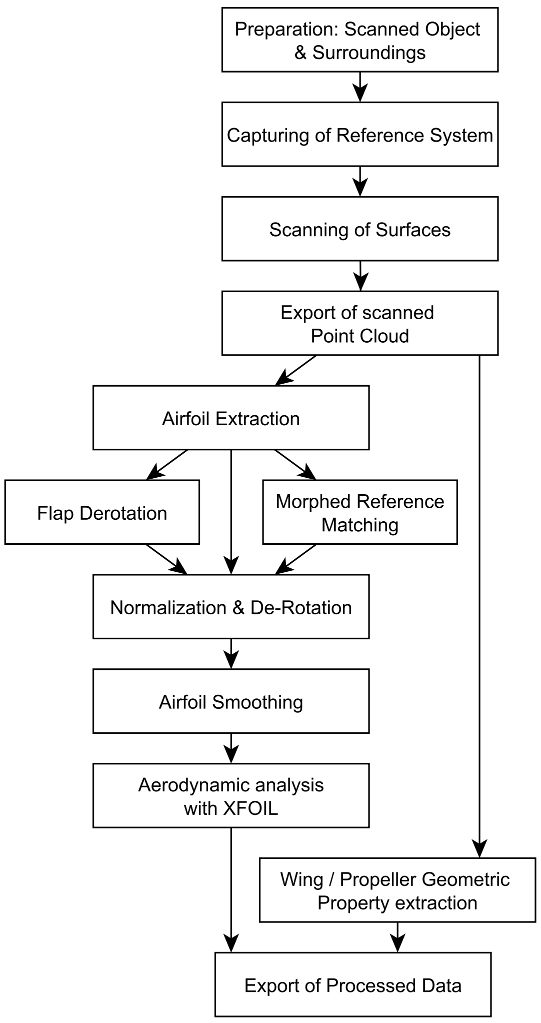

Preparation: Scanned Object & Surroundings
Capturing of Reference System
Scanning of Surfaces
Export of scanned Point Cloud
Airfoil Extraction
Flap Derotation
Morphed Reference Matching
Normalization & De-Rotation
Airfoil Smoothing
Aerodynamic analysis with XFOIL
Wing / Propeller Geometric Property extraction
Export of Processed Data

Figure 1.1: Complete 3D-scanning toolchain (van Brügge et al., 2021, P. 2)

This manual will only focus on the implementation. All functions will be shortly presented in the next chapters. The repository consists of several classes. There are two classes (*Airfoil* and *Fuselage*) to define the separate airfoil and fuselage sections. These can be fitted using the *AirfoilFitter* and *FuselageFitter* classes. The *GeometryExtractor* class is used for all extraction processes while the *PointCloudOperator* class performs all operations on the complete point cloud. Lastly, the *IOHandler* class performs all reading and writing procedures. Separate from this functions is the *UAV* class which is only used to draw a 2D view of the aircraft in the implemented *Graphical User Interface* (GUI) using the *Simple and Fast Multimedia Library* (SFML) (see Gomila (2020)). This GUI is only used to determine the position of the section and can also be skipped by creating a sectioning file (see read.me of the repository and van Brügge (2020)). The point clouds of the aircraft and the sections are handled using the *Point Cloud Library* (PCL) (see Rusu and Cousins (2011)). Matrix operations needed for the fitting processes are provided by *GNU Scientific Library* (GSL) (see Galassi (2018)). An overview of all classes is given in Fig. 1.2.

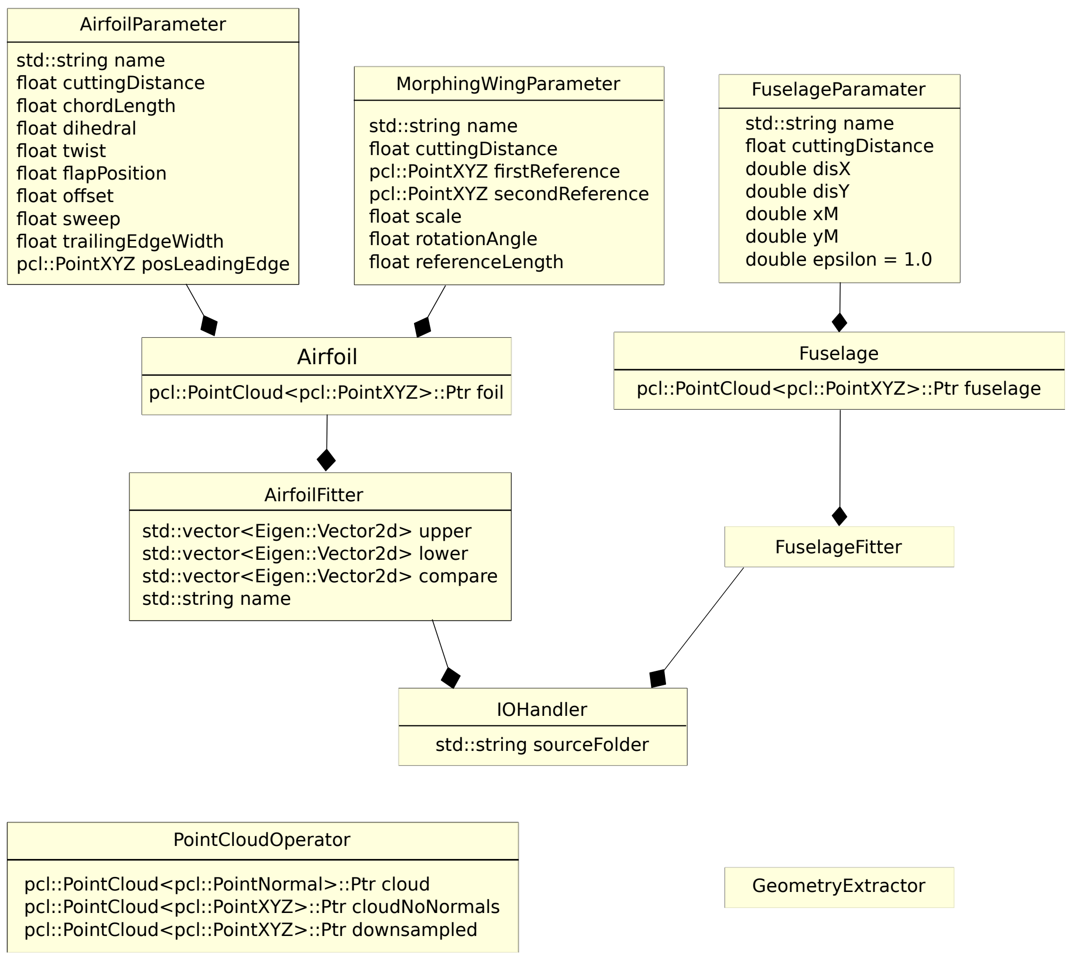


Figure 1.2: Class diagram of the repository including all attributes of the classes

2 Airfoil

The Airfoil class consists of two parameter classes (see sections 2.1 and 2.2 and the point cloud of the airfoil. In the parameter classes all relevant information regarding the airfoil are saved. Note, that so far only one of the two classes contains information depending on the type of the airfoil.

2.1 AirfoilParameter

The AirfoilParameter class does not have any functions itself. It is just used as a structure to combine several attributes. The attributes are listed in Tab. 2.1. Additionally, it contains an enumeration of all parameter names to be able to set them separately.

Parameter Name	Functionality
std::string name	name of the airfoil
float cuttingDistance	defines the position on the wing (segment)
float chordLength	length of the airfoil
float dihedral	the dihedral of the wing
float twist	the twist of the wing at the cutting distance
float flapPosition	relative position of the hinge line (from 0 to 1)
float offset	distance of the LE to the LE of the first section in mm
float sweep	angle computed from the offset and the cutting distance
float trailingEdgeWidth	width of the trailing edge (TE) in mm
pcl::PointXYZ posLeadingEdge	extracted position of the leading edge (LE)

Table 2.1: parameters of the AirfoilParameter class

2.2 MorphingWingParameter

The MorphingWingParameter class does not have any functions itself. It is just used as a structure to combine several attributes. The attributes are listed in Tab. 2.2. Additionally, it contains an enumeration of all parameter names to be able to set them separately.

Parameter Name	Functionality
std::string name	name of the morphing wing
float cuttingDistance	defines the position on the wing (segment)
pcl::PointXYZ firstReference	position of the first reference marker
pcl::PointXYZ secondReference	position od the second reference marker
float scale	factor to scale the airfoil to 1
float rotationAngle	angle used to rotate the airfoil
float referenceLength	appointed distance between the references

Table 2.2: parameters of the MorphingWingParameter class

2.3 Constructor

There are three different constructors defined in the Airfoil class, a default constructor as well as a constructor which sets the airfoil point cloud and either the parameters of a morphing wing or a standard airfoil.

2.4 Get- and Set-Functions

There are several functions for setting and getting the information extracted of the airfoil. It is possible to get the airfoil point cloud and the two parameter classes (see Tab. 2.3).

Function Name	Functionality
getFoil()	returns the saved airfoil
setFoil()	sets a new airfoil
getAirfoilParameter()	returns the complete set of airfoil parameters
getMorphingWingParameter()	see above for morphing wings
setAllAirfoilParameter()	sets a complete set of airfoil parameter
setAllMorphingWingParameter()	see above for morphing wings
setName()	sets a name for the airfoil section

Table 2.3: standard Airfoil get and set functions

For setting values of the parameter classes, it can be chosen if only one specific value should be saved or all the values. For only a single value use the function input *type* to specify the type of the parameter. Possible options for parameter types are specified in Tab. 2.4. The functions are called *setAnyAirfoilParameter()* and *setAnyMorphingWingParameter()*, respectively. As an input the use an enum called *parameterType* and a float value.

Airfoil Parameter	Morphing Wing Parameter
CuttingDistance	CuttingDistance
ChordLength	FirstReferenceX
Dihedral	FirstReferenceY
Twist	FirstReferenceZ
FlapPosition	SecondReferenceX
Offset	SecondReferenceY
Sweep	SecondReferenceZ
TrailingEdgeWidth	Scale
PosLeadingEdgeX	RotationAngle
PosLeadingEdgeY	ReferenceLength
PosLeadingEdgeZ	

Table 2.4: Different parameter types for morphing wing and standard airfoil

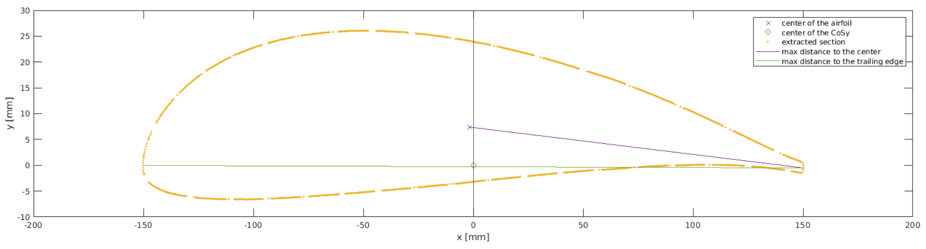


Figure 2.1: Finding the leading and TE of the airfoil

2.5 findTrailingEdge()

This function computes the index of the TE of the airfoil point cloud. For this algorithm, the airfoil section has to be distributed normally. All position are added to find the center of the airfoil. Afterwards a nearest neighbors search is performed for all the data points in the airfoil. The point which has the greatest distance to the computed center is chosen as TE.

2.6 findLeadingTrailingEdge()

Using the index of the *findTrailingEdge()* function, the point with the greatest distance to the TE is chosen as LE. The functionality of the two methods is displayed in Fig. 2.1).

2.7 generateMissingAirfoilParameter()

This function computes all parameters of the airfoil parameter which were not set until now. These parameters are the sweep, the offset, the name and the flap position.

2.8 deleteMorphingWingReferences()

Here, the references of the morphing wing will be deleted. Starting at the saved reference points all points which lay in the computed reference width are deleted on the bottom side of the airfoil. This is done with a small threshold to ensure that the complete reference is deleted.

2.9 computeRotatedFlapPosition()

This function corrects the computed flap position after the derotation of the airfoil around the twist angle.

3 Fuselage

Similar to the Airfoil class, the fuselage class consists of a parameter class and the point cloud of the fuselage.

3.1 FuselageParameter

The FuselageParameter class does not have any functions itself. It is just used as a structure to combine several attributes. The attributes are listed in Tab. 3.1. Additionally, it contains an enumeration of all parameter names to be able to set them separately.

Parameter Name	Functionality
std::string name	name of the fuselage section
float cuttingDistance	relative position of the fuselage section
double disX	distance from the center to the hull in x-direction
double disY	distance from the center to the hull in y-direction
double xM	position of the center in x-direction
double yM	position of the center in y-direction
double epsilon	parameter defining an ellipse (default = 1.0)

Table 3.1: parameters of the FuselageParameter class

3.2 Constructor

There is a default constructor and a constructor which sets the extracted point cloud of the fuselage.

3.3 Get- and Set-Functions

There are several functions for setting and getting the information extracted of the fuselage. It is possible to get the point cloud and the parameter class For setting values of the parameter

Function Name	Functionality
getFuselage()	returns the saved fuselage
setFuselage()	sets a new fuselage
getFuselageParameter()	returns the complete set of fuselage parameters
setAllFuselageParameter()	sets a complete set of fuselage parameter
setName()	sets a name for the fuselage section

Table 3.2: standard Fuselage get and set functions

class, it can be chosen if only one specific value should be saved or all the values. For only a single value use the function input *type* to specify the type of the parameter. Possible options for parameter types are specified in Tab. 3.3. DisX is the width of the fuselage, DisY is the height of the fuselage and XM and YM the center of the fuselage. Epsilon defines the shape of the fitted superellipse (see Sec. **??**). The functions used is called *setAnyFuselageParameter()*. The inputs are an enum called *parameterType* and a float value.

Fuselage Parameter
CuttingDistance
DisX
DisY
XM
YM
Epsilon

Table 3.3: Different parameter types for the fuselage

3.4 computeFuselageParameter()

This function computes the parameters of the fuselage excluding epsilon using a bounding box. The distance and the center of the fuselage is only defined by the bounding box. The shape can be later approximated by a circle, an ellipse or a superellipse (see Sec. 5). This will set the epsilon in the fuselage parameters. Without fitting, the fuselage will be approximated by a circle.

4 AirfoilFitter

The *AirfoilFitter* class is used for smoothing the extracted section. Two options are possible, a spline interpolation and a bernstein polynomial fitting. The class uses the *GNU Scientific Library* (GSL) for the computation of the fitting.

An *AirfoilFitter* contains of an *IOHandler* (see Sec. 6) object which will be used for saving the airfoil. Additionally, the object saves the upper and the lower part of the airfoil in two separate vectors plus a vector which contains an approximation of the skeleton line. This vectors are filled in using the constructor.

4.1 Constructor

The constructor expects an *Airfoil* (see Sec. 2) and a string containing a path for a writeable folder. The folder is used for constructing the *IOHandler* object. For the smoothing process, the airfoil has to be split in half. Therefore, the constructor computes several points lying on the skeleton line of the airfoil. This is done by the function *computeCompareValues()* (see Sec. 4.1.1). Afterwards the function *splitAirfoil()* (see Sec. 4.1.2) is used to split the airfoil in an upper and a lower half, since a complete airfoil can't be fitted. Note, that in this step the orientation of the airfoil is still arbitrary such that the upper half need not correspond to the upper side of the airfoil. The computed skeleton line is shown in Fig 4.1.

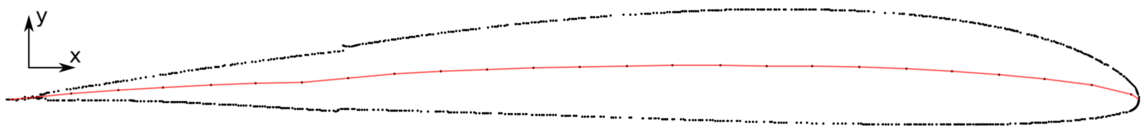


Figure 4.1: Computed skeleton line of an airfoil with arbitrary orientation (van Brügge, 2020, P.34)

4.1.1 computeCompareValues()

The airfoil is filtered equidistantly and the center of each filtered point cloud is computed. This point is saved as a compare value of the skeleton line. Additionally, the furthermost and last point of the airfoil is also saved. An example of a filtered airfoil section is shown in Fig. 4.2.

4.1.2 splitAirfoil()

This function will be called in the constructor of the airfoil fitter. It divides the non-ordered airfoil section in to point clouds presenting the upper and the lower half of the airfoil. Note, these will be still unordered!

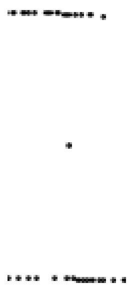


Figure 4.2: Computed center point of a filtered airfoil section

For splitting the airfoil, the skeleton line is needed. An approximate skeleton line is interpolated using the points of the *computeCompareValues()* function. A linear interpolation between 2 neighboring points is used for this. All points of the airfoil section are then iterated. It is tested if a point has a higher or lower y-value than the corresponding point on the skeleton line. Points laying above the skeleton line are saved in a vector containing all points of the upper side and vice versa.

4.2 initiateFitting()

This functions contains the complete procedure of the fitting. In the input, it is specified if the airfoil should be fitted with the Bernstein polynomial fitting or the spline interpolation. The foil is sorted, oriented and fitted. Afterwards it will be saved as a DAT-file using the *IOHandler* class.

4.3 sortUpperAndLowerHalves()

This functions sorts the upper and lower halves of the aircraft in respect to the x-axis. Sorting is done increasingly such that the smallest x-value will be at position 0.

4.4 orientFoil()

This functions checks the orientation of the airfoil and mirrors it at the corresponding axes. Therefore, it uses the functions *checkIfFoilIsUpsideDown()* and *checkPosition-LeadingEdge()*.

4.4.1 checkIfFoilUpsideDown()

Here, the data points of the skeleton line are used to decide weather the airfoil is upside down. The function is based on the assumption that all standard airfoils are either symmetric or concave. If the computed skeleton line is convex, the airfoil has to be mirrored at the x-axis.

This can be checked by comparing the y-values of the center of the skeleton line to the trailing or LE of the airfoil. Is the value smaller at the center, the airfoil is upside down and as to be mirrored.

4.4.2 checkPositionLeadingEdge()

This function assumes that the airfoil is thicker at the LE than the TE. It is important that the airfoil is already sorted increasingly. Then, the y-values of the 20th point is compared to the point which is 20 points before the last point. A greater value at the end indicates a false orientation and the airfoil is mirrored at the y-axis.

4.5 splineInterpolation()

This function uses splines introduced by Steffen (1990), since these splines oscillate rarely even if the point distribution is not evenly. The interpolation functionality is provided by GSL Galassi (2018). The data points are redistributed with a Chebyshev node distribution to ensure a good resolution of the LE. The position of the x-values can be computed using Eq. 4.1. Fitting is done separately for the upper and the lower half with 70 points on each surface.

$$x_k = cos(\frac{2k-1}{2n}\pi),\ k = 1, ..., n \tag{4.1}$$

Note, this interpolation tends to oscillate in some cases when the resolution of the scan is to low (see van Brügge (2020)).

4.6 bernsteinPolynomialFit()

Another fitting option is a polynomial fitting using Bernstein polynomials multiplied by a shape function. This method was introduced by Kulfan and Bussoletti (2006). The shape function (see Eq. 4.2) approximates a standard NACA airfoil including a infinite gradient at the LE and a pointy TE.

$$f(x) = \sqrt{x} \cdot (1-x) \tag{4.2}$$

Since Bernstein polynomials always sum up to one (see Grant Timmerman (2014)), the key features of the shape function are retained. The characteristics of the airfoil like thickness and camber are presented by the coefficients β_d The complete fitting rule including the shape function is presented in Eq. 4.3.

$$y_k \;=\; \Sigma_{i=0}^{d} \binom{d}{i} \beta_d \cdot \cdot x_k^i \cdot (1 - x_k)^{d+1-i} \cdot \sqrt{x_k} \tag{4.3}$$

The degree (*d*) chosen is 8 since this proved sufficient in Kulfan and Bussoletti (2006) to achieve wind tunnel accuracy. *(x,y)* is the corresponding position of the *k* points used to described the airfoil.

The Bernstein polynomial coefficients β_d are computed by the function *getBernsteinPolynomialCoeff()* (see Sec. 4.6.3). For the fitting 100 points on each (lower and upper) surface are computed using a Chebyshev node distribution (see Eq. 4.1) and the y-value of the Bernstein polynomial function (see Eq. 4.3).

For a successful fitting, the airfoil has to be oriented beforehand and points on the TE have been deleted (see Fig. 4.3). It is also mandatory, that it was derotated such that the trailing and leading edges are on the x-axis. Due to manufacturing tolerances, this can't be guaranteed for every airfoil section. Therefore, the width of the TE is calculated and the airfoil is fitted twice, once with the LE on the x-axis and once with the TE on the x-axis. Afterwards these two fittings will be combined if the TE is too thick for a single fitting approach (see Fig. 4.4). Errors which can occur in the fitting process are listed in Lina van Brügge (2022).

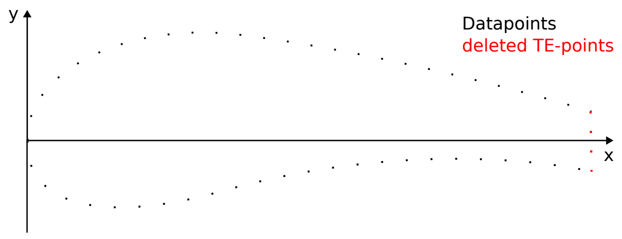


Figure 4.3: Example of an acceptable airfoil for the fitting process

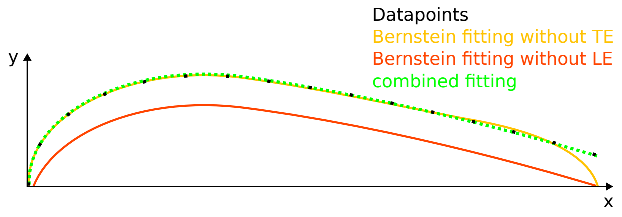


Figure 4.4: Bernstein fitting method for airfoils with a thick TE

4.6.1 binomialCoeff()

This functions computes the binomial coefficient n over k. It returns a *long*.

4.6.2 getBernsteinPolynomialValue()

This function returns an approximated y-value from a given set of coefficients and a x-value. The equation is 4.3.

4.6.3 getBernsteinPolynomialCoeff()

This function computes the coefficients β_d for the polynomial fitting. It uses a basic approximation scheme shown in Eq. 4.4. b are the y-values of the points of the airfoil. The matrix **A** is a $(d+1) \times n$-matrix. It is computed using Eq. 4.5.

$$\vec{\beta} = \mathbf{A}\vec{b} \tag{4.4}$$

$$a_{i,j} = \binom{d}{j} \cdot x_i^j \cdot (1-x_i)^{d+1-j} \cdot \sqrt{x_i},\ j = 0, ..., d+1 \tag{4.5}$$

The solution is computed using a QR solver from GSL.

4.7 replaceMorphedFlap()

This function is solely used for morphing wing demonstrators. It was necessary since the demonstrators missed a deflectable flap. So, in order to be able to do computations in e.g. XFOIL (see Drela and Youngren (4.3.2017)), the flap of the reference profile has to be added to the scan. The difference between the morphed reference profile and the non-deflectable flap of the demonstrator can be seen in Fig. 4.5. The airfoil parts are sorted increasingly.

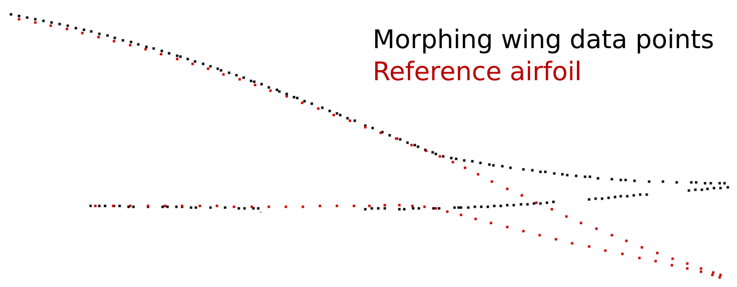


Figure 4.5: Flaps of the morphing wing demonstrator compared to the reference profile

Since the airfoil is already oriented such that the leading edge is at the front. It doesn't have to be completely oriented. However, it can be positioned upside down. In this case, the airfoil is mirrored at the x-axis. Afterwards, the airfoil part is downsized if the number of points of one part of the airfoil exceeds 100 points. After this steps all points of the reference foil which have a higher x-value than the morphed airfoil will be inserted. The foil is then saved using the *IOHandler* object.

4.7.1 downsizeAirfoil()

This function reduces the size of the input vector to approximately 100 data points. Intermediate points will be deleted.

5 FuselageFitter

The *FuselageFitter* class is used to approximate the given shape of the aircraft fuselage by a defined geometry. Possible approximation options are circular, ellipse and superellipse fitting. A fuselage can only be fitted, if the section does not contain parts of the wing or tail or any other mounted devices like cameras. The fitting options were chosen since most of the common aerodynamic tools like XFLR5 (see *XFLR5* (3.4.2022)) use circles or ellipses to model a fuselage. The superellipse fitting was added to be able to approximate a greater variety of fuselages.

The *FuselageFitter* class contains a *IOHandler* (see Sec. 6) object which is used for saving the fitted fuselage and a *Fuselage* object. This object will be modified in the fitting process.

5.1 Constructor

The inputs for the constructor are similar to the constructor of the *AirfoilFitter* class. A *Fuselage* object is needed as well as a string which contains a writeable path for the results. This path is then used to create the *IOHandler* object.

5.2 circularFit()

For the circular fit, a bounding box is placed around the fuselage. Its center is used as the center of the circle. The radius r is approximated by the averaged height and width of the bounding box. 360 points are then placed as a fuselage using Eq. 5.1.

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_M \\ y_M \end{pmatrix} + r \cdot \begin{pmatrix} cos(\frac{k}{180} \cdot \pi) \\ sin(\frac{k}{180} \cdot \pi) \end{pmatrix}, k = 0, ..., 359 \tag{5.1}$$

This approximation is saved as well as the value 1.0 for the fuselage parameter epsilon. The fitting procedure is shown in Fig. 5.1.

5.3 ellipsoidalFit()

This function works similar to the *circularFit()*-function (see Sec. 5.2). The only difference is, that the height and width of the bounding box correspond directly to the half axes a and b of the ellipsoidal fitting. The positioning of the points are described using Eq. 5.2.

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_M \\ y_M \end{pmatrix} + \begin{pmatrix} a \cdot cos(\frac{k}{180} \cdot \pi) \\ b \cdot sin(\frac{k}{180} \cdot \pi) \end{pmatrix}, k = 0, ..., 359 \tag{5.2}$$

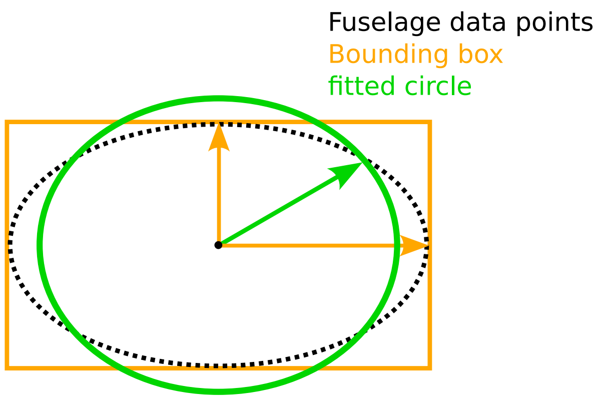


Figure 5.1: Circular Fitting method

The points are saved and epsilon is set to 2.0.

5.4 superellipsoidalFit()

To be able to approximate more possible fuselage shapes, a superellipsoidal fitting was added. A superellipse is described by Eq. 5.3 under the condition $S(x, y) = 1$.

$$S(x,y) = |\frac{x}{a}|^{\frac{2}{\epsilon}} + |\frac{y}{b}|^{\frac{2}{\epsilon}} \tag{5.3}$$

For $\lim_{\epsilon \to \infty}$ the equation describes a rectangle, for $\epsilon = 1$ a diamond, and for $\epsilon < 1$ a cross. Fitting a superellipse is always a difficult optimization problem (see Rosin (2000)). To avoid the complex implementation, another method was chosen for the fitting. To get rid of the modulus, all points are shifted to the first quadrant (see Fig. 5.2). Then the bisection method is used to fit a superellipse to the data points and minimize the absolute difference. The loop procedure is enumerated below:

1. Insert the position of the data points in the equation of the superellipse (see Eq. 5.3) with a chosen ϵ
2. Compute the difference of the points to the ideal superellipse and average it
3. Select new boundaries for epsilon
4. Compute new ϵ

As initial conditions 0.0 is used as a right boundary condition and 50.0 as the left boundary condition. The ϵ chosen for the next step is defined by Eq. 5.4.

$$\epsilon = \frac{4}{(bound_{right} + bound_{left})} \tag{5.4}$$

The loop will be interrupted if either the loop exceeds 25 iterations or the difference of the points to the fitted superellipse is lower than 10^{-3}.

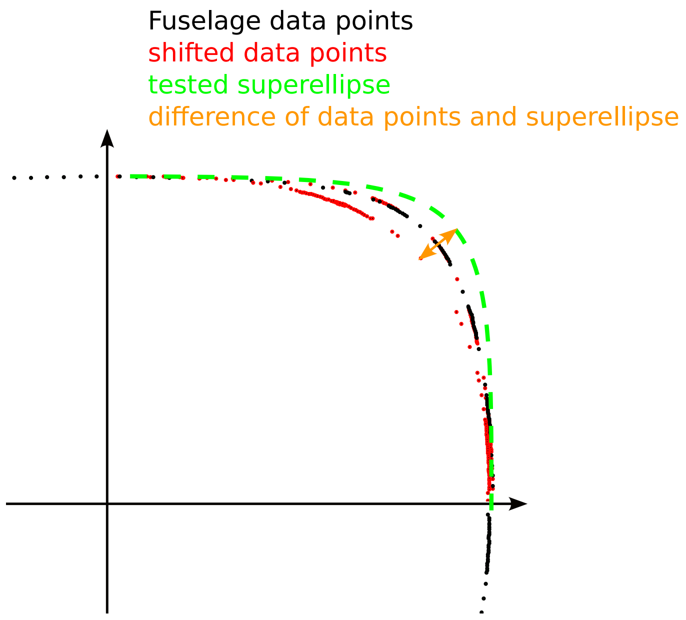


Figure 5.2: Fitting of the superellipse to the data points

6 IOHandler

The *IOHandler* is used for all functionalities which write or read data from a file. The data is always not edited, all inputs to file writing functions are constant. Its single variable is a string.

6.1 Constructor

The *IOHandler* has to constructors. The first constructor gets a writeable and existing filepath where all results will be written in. The other one is the default constructor.

6.2 writingPointCloud()

This section writes an point cloud to a DAT-file. The two inputs are the point cloud data as a vector of 2D points and the file name. The filename is added to the path of the result folder. If it can't be open because e.g. the folder does not exist, an error will be printed in the console.

6.3 writingWingDataInCSV()

This function writes in a CSV-file all aircraft parameters which could be generated during the sectioning process. As input it expects an outstream, the saved *AirfoilParameter*, a string for the type of section and the number of sections. The outstream is needed that all data can be written in the same CSV-file. The section type can either be *wing*, *horizontal_tail* or *vertical_tail*. The columns are:

- Filename
- Sectioning Distance [mm]
- Chord Length [mm]
- Dihedral [deg]
- Twist [deg]
- Flap Position [%]
- Offset [mm]
- Sweep [deg]
- Trailing Edge Width [mm]
- x-Position Leading Edge [mm]
- y-Position Leading Edge [mm]
- z-Position Leading Edge [mm]

6.4 writingMorphingWingDataInCSV()

For saving morphing wing data in a CSV file, the function *writingMorphingWingDataInCSV()* is used. This function expects the same outstream as the *writingWingDataInCSV* (see Sec. 6.3) and additionally the saved *MorphingWingParameter* and the number of sections. The written columns are:

- Filename
- Sectioning Distance [mm]
- Scale []
- Rotation Angle [deg]

6.5 writingFuselageDataInCSV()

This function saves all *FuselageParameter* which where generated during the sectioning. It's inputs are similar to the inputs of the *writingMorphingWingDataInCSV()* function (see Sec. 6.4). The columns of the table are:

- Filename
- Sectioning Distance [mm]
- Half-axis X [mm]
- Half-axis Y [mm]
- Center X [mm]
- Center Y [mm]
- epsilon []

6.6 readSectionFile()

This functions reads the sectioning file and copies the cutting distances in the corresponding vectors. The generation of the sectioning file is explained in the *read.me* of the project. A possible sectioning file is presented in Fig. 6.1.

The names in the beginning of each line specifies which direction will be sliced and is used to decide in which vector the variables will be saved. The function used for extracting the values of the line in the TXT-file is called *readLineInVector()*.

In this function a stringstream is used to push all the values in a double value. This is then appended to the sectioning vector given as input.

```
h
-800
fuselage 1140 980 800 500 180 50 -900
horizontal_tail 70 150 300 410
wing 120 135 700 1450 2000 2875 2915 2975
vertical_tail -200 -135 -90 75 170
```

Figure 6.1: Exemplary section generation file

6.7 convertTXTToPCDFile()

This function can be used if the scanned aircraft is instead of a point cloud data file (PCD) a ASCII point cloud. A string containing the path to the point cloud file is used as input. Note, that this file has to be in a writeable folder, since the PCD file couldn't be generated in any other case.

A PCD-file is the standard export format of PCL. It is a ASCII point cloud extended by a header which specifies e.g. the data format (see Rusu and Cousins (2011)). The function opens a new file, adds the header and saves then all the points of the point cloud to the new file.

6.8 readAirfoilDATFile()

This function reads an airfoil from a DAT file and saves the points in a vector. A while loop containing a stringstream for pushing the values in a varaible is used for this procedure.

7 Sectioning GUI

In this chapter, the procedure of the sectioning GUI will be explained. For a detailed description how to use the GUI see van Brügge (2020). The GUI is generated using SFML. Therefore, most of the functionalities are depending on it.

7.1 Class UAV

The *UAV* class is a transformable and drawable class. It is used for drawing the point cloud of the aircraft in the window and scale it accordingly.

7.1.1 Parameters

The parameters of the *UAV* class are listed below:

- windowWidth (int)
- windowHeight (int)
- minPt (Eigen::Vector2f)
- maxPt (Eigen::Vector2f)
- scaling (float)
- m_vertices (private sf::VertexArray)
- m_texture (private sf::Texture)

The variables *minPt* and *maxPt* are used to scale the picture of the aircraft in the GUI and *scaling* saves the scaling factor.

7.1.2 Constructor

The input of the constructor are the number of points in the point cloud and the width and height of the GUI. It allocates memory for the points inside the point cloud and sets the scaling factor to 1.

7.1.3 loadUAV()

This function is used for loading the points of the point cloud into the drawable vertex array *m_vertices*. Therefore it needs the point cloud and the two dimensions which should be printed (e.g. X-Y). This is necessary since the GUI is 2D and therefore needs 2D-vertices as input. Additionally, yellow is set as color for the vertices and the maximum and minimum values of the point cloud are computed. The values corresponding to the dimensions of the point cloud are saved in the variables *minPt* and *maxPt*.

7.1.4 getScalingFactor()

This function computes the scaling factor. The scaling factor guarantees that the aircraft is completely visible when it is drawn the first time. Depending on the distance between the maximum and minimum point the aircraft will be either scaled using the horizontal scaling factor s_h or the vertical scaling factor s_v (see Eq. 7.1). The constant term were added to add space between the window edges and the points of the aircraft.

$$\begin{aligned} s_h &= \frac{height_{window}}{|minPt_1 - maxPt_1| + 200} \\ s_v &= \frac{width_{window}}{|minPt_2 - maxPt_2| + 100} \end{aligned} \tag{7.1}$$

7.1.5 resize()

This function resizes the drawable vertices *m_vertices* using the given integer.

7.1.6 translation()

This function is used to adjust the position of the UAV in the window. If the UAV was filtered to show just a part of it, it will be printed near the left edge of the window, otherwise it is printed in the center of the window.

7.1.7 draw()

This function draws the UAV in the window. First of all, a transformation matrix is generated. There a translation is added such that the UAV will be drawn in the center of the window. Additionally, it is scaled, that the greater dimension fits completely in the window. This transformation is then used for drawing the vertices.

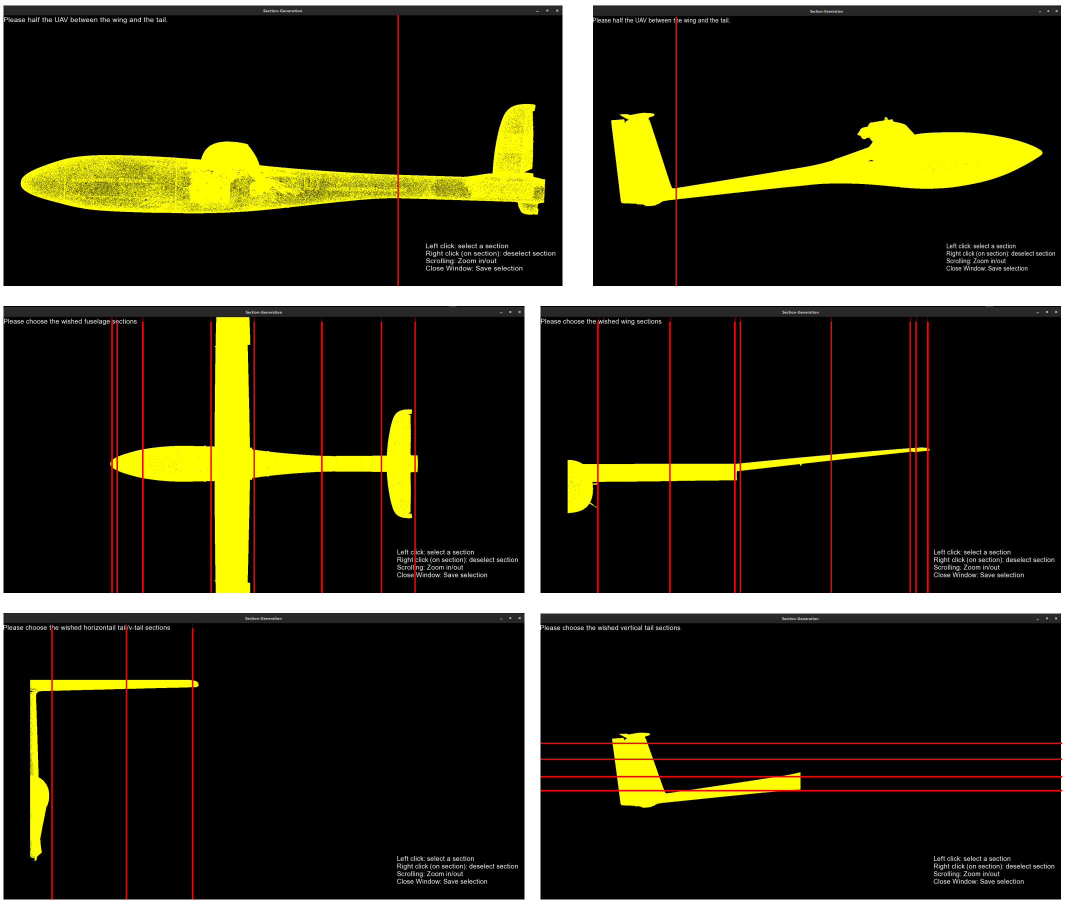
7.2 sectionGenerationGUI()

This function is used for executing all needed procedures which are needed for the section generation file. There are two common tail configurations for aircraft: a conventional tail or a v-tail. To be able to generate sections for both aircraft types, an user input is required at the beginning. In the command line, the user has to input if the scanned aircraft is an aircraft with a v-tail or not. This will be saved also in the first line of the section generation file for the later extraction process since it define how many different parts of the aircraft have to be extracted. Afterwards, the output on the GUI will be generated. It consist of:

- Instructions what to do in the current step

- Drawn aircraft in different views
- Small help on how to use the GUI
- Already selected sections

In Fig. 7.1 the different steps of the section generation GUI are depicted. Note, that the orientation of the aircraft is not completely certain as shown in the Fig. in the pictures at the top.



Please half the UAV between the wing and the tail.

Left click: select a section
Right click (on section): deselect section
Scrolling: Zoom in/out
Close Window: Save selection

Please half the UAV between the wing and the tail.

Left click: select a section
Right click (on section): deselect section
Scrolling: Zoom in/out
Close Window: Save selection

Please choose the wished fuselage sections

Left click: select a section
Right click (on section): deselect section
Scrolling: Zoom in/out
Close Window: Save selection

Please choose the wished wing sections

Left click: select a section
Right click (on section): deselect section
Scrolling: Zoom in/out
Close Window: Save selection

Please choose the wished horizontail tail/v-tail sections

Left click: select a section
Right click (on section): deselect section
Scrolling: Zoom in/out
Close Window: Save selection

Please choose the wished vertical tail sections

Left click: select a section
Right click (on section): deselect section
Scrolling: Zoom in/out
Close Window: Save selection

Figure 7.1: Exemplary steps for the generation of a section generation file (van Brügge, 2020, P.25)

As can be seen, the first two required sections are done without any filtering. The first section determines a value between the wing and the tail. This is needed to be able to section both, the wing and the tail without sectioning the other part, too. Afterwards, the sections for the fuselage are selected. For printing the UAV for the wing and tail sections it is filtered first. It is halved at the symmetry axis that there will be just positive values in the selected sections and the tail is also filtered using the first selected value. Since the orientation is not completely certain, it is also possible that the sections for the horizontal tail will be selected before the wing sections. The last sectioning will just be executed if the aircraft has a conventional tail, otherwise it will be skipped.

The selection which part of the UAV has to be drawn is managed by a switch-case condition executed in a for-loop. Afterwards, everything will be drawn in the window. The drawing is a while loop which isn't stopped until the window is closed. To be able to select every part of the aircraft for sectioning, the function will open for every selection a new window. Whenever the window is closed, the selected values will be saved in the section generation file.

In the drawing loop, there is a second loop which is triggered every time when a event occurs. These events will be described in the following paragraphs.

Close window After this event, the window will be closed and all selected sections will be saved in the sectioning file. Since the drawing of the aircraft is scaled and translated, the values have to be transformed and re-scaled first. The splitting position between tail and wing is saved additionally in a variable to be able to filter the aircraft later.

Left mouse button pressed A line will be added at the clicked position - most of the time horizontally, if the sections or the vertical tail will be selected vertically. The location of the line is computed on the non-scaled and non-transformed aircraft to be able to shift it afterwards, if the aircraft is zoomed.

Right mouse button pressed If a line is at the position of the mouse click, it will be deleted from the array.

Scrolling Scrolling will adjust the scaling factor of the UAV to be able to select the sections more accurately.

In the end of the drawing loop, all text, the UAV and the selected positions (as lines) will be drawn.

8 PointCloudOperator

This class is used for creating point cloud objects and editing them. The functionalities are depending on PCL (see Rusu and Cousins (2011)). It has three different point cloud variables. The first is a point cloud which consists of 3D-data points and surface normal vectors corresponding to every data point. The second is the same point cloud without the normal vectors. The third is a downsampled version of the second point cloud to speed up the computation of the registration.

8.1 Constructor

There are three different constructor available in for this class. The first constructor assumes an already registered point cloud with normal vectors. The input is just saved as attribute and the point cloud is copied to get a point cloud without normal vectors for the second variable.

The second constructor expects a point cloud as input as well as two booleans. The first is true if the dimension of the fuselage is larger as the dimension of the wing and the second one is true if the scan is a complete aircraft and not just a part of it. If the size of the point cloud is bigger than 10^6, it will be downsized (see Sec. 8.5). Afterwards it will be aligned (see Sec. 8.4 and normal vectors will be computed (see Sec. 8.3).

The third constructor works similar to the second constructor, however, the point cloud is not loaded already. Instead the path of the PCD file is set as an input. This file is then loaded and the cloud will be downsized and aligned accordingly. Then, normal vectors will be estimated.

8.2 Get-Functions

There are two get-functions implemented. The first returns the point cloud with computed normal vectors (*getPointCloudWithNormals()*) and the other one with returns the point cloud without normal vectors (*getPointCloudWitoutNormals()*). The downsampled point cloud is just used internally.

8.3 estimateNormals()

There are two functions which are used to compute the normal vectors of the aircraft. Both use the same *Moving Least Square* algorithm of PCL. The difference between the two functions is that the first computes the normal vectors of the *cloudNoNormals* object of the *PointCloudOperator* class and the other one gets a point cloud as an input which will be used. The normal estimation is based on a surface generation with radius 2.5 mm using a *KDTree* object of PCL (see Rusu and Cousins (2011)).

8.4 aligningPointCloud()

This functions aligns the aircraft. A feature estimation of PCL is used to obtain the smallest bounding box. It is rotated afterwards such that the center of the bounding box becomes the origin and the axis of the aircraft become the main axis of the coordinate system. The biggest dimension is chosen as X and the smallest as Z. So, for a standard aircraft, the yz-plane is the symmetry plane of the aircraft (see Fig. 8.1). To speed up the feature estimation, the downsampled point cloud will be used for this step.

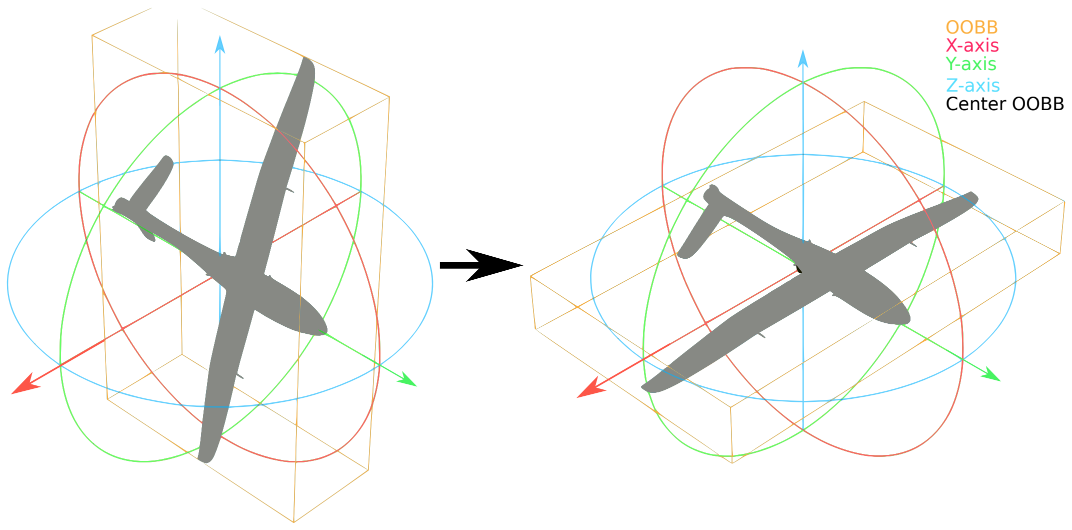


Figure 8.1: Aligning process of the aircraft (van Brügge, 2020, P.27)

The sectioning process assumes this orientation of the aircraft. If the length of the fuselage is greater than the wingspan, the orientation has to be corrected. If this boolean is set *true*, the aircraft will be rotated after the orientation step about $\pi/2$ around the z-axis to guarantee the correct orientation of the aircraft for the sectioning. Depending on the aircraft shape, it is also possible, that the fuselage is not completely horizontally. That can occur if the aircraft has a Y-tail (see the red aircraft in Fig. 8.2).

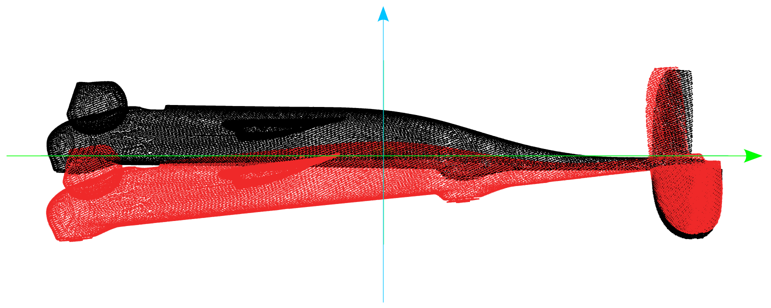


Figure 8.2: Alignment error for aircraft with Y-tail

Therefore, an additional correction is needed. This will be triggered by the second boolean. The correction angle will be computed using the function *getAngleXZPlane()* (see Sec. 8.6). Afterwards the point cloud will be rotated around the x-axis using the computed angle.

8.5 downsize()

This function downsamples the point cloud. In a circle with a radius of 5 mm around one point all other points of the point cloud will be filtered.

8.6 getAngleXZPlane()

For computing the correction angle, the point cloud is filtered around the origin. It is assumed that there are no objects attached in this area to guarantee that the small bounding box will be minimized. It is a iterative procedure. The initial step size is 1 $^\circ$. The algorithm is shown in Fig. 8.3.

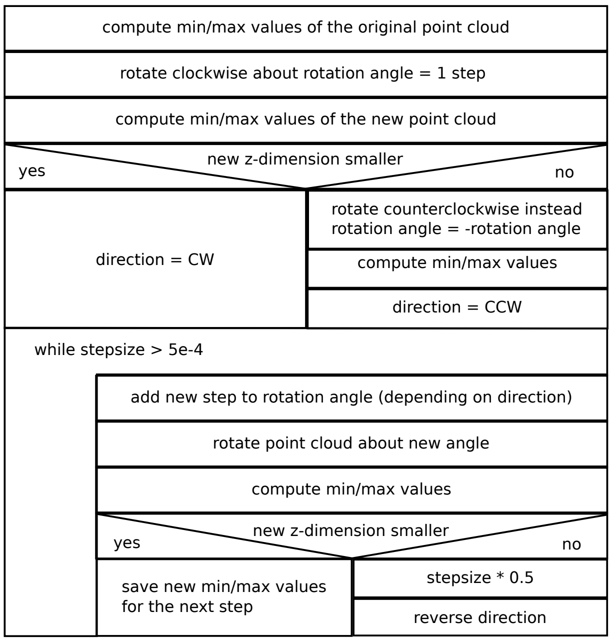

compute min/max values of the original point cloud
rotate clockwise about rotation angle = 1 step
compute min/max values of the new point cloud
new z-dimension smaller
yes
no
direction = CW
rotate counterclockwise instead rotation angle = -rotation angle
compute min/max values
direction = CCW
while stepsize > 5e-4
add new step to rotation angle (depending on direction)
rotate point cloud about new angle
compute min/max values
new z-dimension smaller
yes
no
save new min/max values for the next step
stepsize * 0.5
reverse direction

Figure 8.3: Procedure to compute the correction angle

8.7 splitCloudInWingAndTail()

This function filters the point cloud and saves it in three smaller point clouds. It uses the splitting distance of the sectioning file. If the x-dimension of the filtered point cloud is smaller than the original one, the point cloud is saved as vertical tail. It will be additionally filtered at the XZ-plane to speed up the computations. This will be saved as horizontal tail. The other part of the aircraft is considered as the wing and is also filtered at the XZ-plane.

9 GeometryExtractor

The *GeometryExtractor* class doesn't have any attributes. It is used for extracting the different sections of the point clouds and further editing steps of the sections like translating and rotating.

9.1 sectioningCloudX()

This function returns an *Airfoil* object which was sectioned at the cutting distance. An integer containing the sectioning type will specify the further processing steps of the airfoil section. It is impossible to just section a plane parallel to the YZ-plane since most of the aircraft feature a dihedral. In order to get the correct airfoil shape, this angle has to be computed first and the sectioning plane rotated about it (see Fig. 9.1).

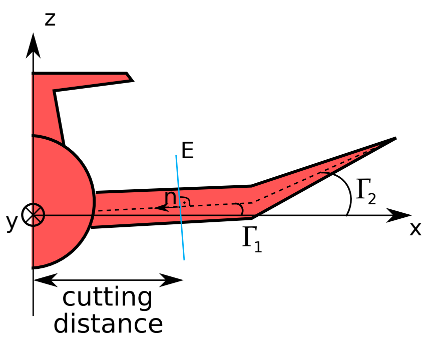


Figure 9.1: Creation of the sectioning plane (van Brügge, 2020, P.29)

Since it is only possible to filter point clouds in direction of one of the main axes, the complete aircraft will be rotated about the computed angle and filtered afterwards using a plane parallel to the YZ-plane (see Fig. 9.2). Therefore, a new cutting distance has to be computed.

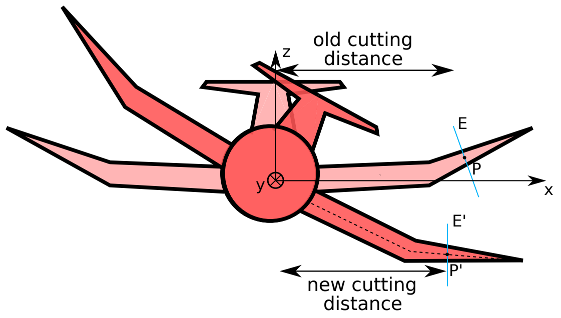


Figure 9.2: Rotation of the aircraft for sectioning (van Brügge, 2020, P.31)

The algorithm follows these steps:

1. filter point cloud near the cutting distance ($\pm 0.5\ mm$)
2. compute an averaged surface normal

3. save the index of the leading edge point of the section in the complete point cloud
4. compute the normal vector of the sectioning plane using the cross product
5. compute the dihedral
6. rotate the point cloud around the dihedral
7. generate the sectioning plane using the new distance of the saved point
8. filter the point cloud using the sectioning plane $\pm 0.1 * k$ (k guarantees that enough points will be present in the section)
9. project all filtered points on the sectioning plane
10. post-process section depending on the sectioning type

The post-processing can be either to derotate a flap (*sectioningType* == 1) (see Sec. 9.10, scale and derotate a morphing wing (*sectioningType* == 2) (see Sec. 9.7) or just saving the dihedral and the section as an *Airfoil* object (*sectioningType* == 0).

9.2 sectioningCloudY()

This function returns a *Fuselage* section which was generated at the cutting distance. Since the fuselage can be sectioned with a plane parallel to the XZ-plane no rotation is needed (see Sec. 9.1). The remaining steps are:

1. creating a sectioning plane at the cutting distance
2. filter the point cloud using the sectioning plane $\pm 0.1 * k$ (k guarantees that enough points will be present in the section)
3. project points on the sectioning plane
4. set fuselage parameter and save the section

9.3 sectioningCloudZ()

This function is a combination of the sectioning procedures in x- and y-direction (see Sec. 9.1 and 9.2). The section does not have to be rotated since it can be represented by a plane parallel to the XY-plane. The algorithm can be described by:

1. copy steps 1-3 of Sec. 9.2 just in z-direction
2. rotate the gained section about 90° around the y-axis
3. post-process the point cloud (derotate flap or save *Airfoil* object)

9.4 derotateSection()

Using this function, the twist angle of the wing can be computed. Additionally, the aircraft will derotated such that the leading and trailing edge will be parallel to the x-axis (see Fig. 9.3). For the derotation the indexes of the LE and TE has to be found. This is done using the function *findLeadingTrailingEdge()* (see Sec. 2.6). A vector is created pointing from the TE to the LE and the angle Θ to the XY-plane is calculated. The airfoil is then rotated clockwise (CW) and counterclockwise (CCW) since it is not possible to be certain of the rotational direction. Minima and maxima points of both rotated airfoils are extracted and the one which has a smaller z-dimension will be selected as the derotated airfoil.

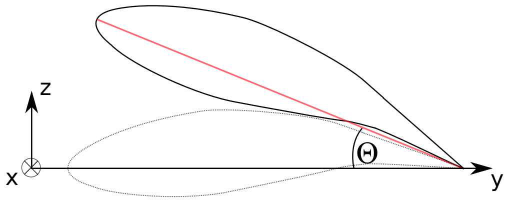


Figure 9.3: Derotation of the airfoil around the twist angle (van Brügge, 2020, P.34)

Note, that the airfoil should be translated to the origin first if the LE and TE should be on the x-axis after the derotation.

9.5 translateSection()

This method translates the airfoil such that the point laying in the middle of the airfoil become the origin. Since all position information will be deleted in the process, the original position of the LE is saved in the *AirfoilParameter* object of the *Airfoil*.

The translation vector is determined by the half distance between the leading and trailing edge. If the airfoil was extracted with a flap then the flap position will be also translated accordingly.

9.6 deleteTrailingEdge()

With this function, points laying on the TE will be deleted. This can be the case when the TE is very thick due to e.g. manufacturing tolerances. The fitting will fail if any points remain on the vertical line connecting the upper and lower airfoil surface. Thus, these points have to be deleted.

The algorithm filters first of all the point cloud near the trailing edge to compute the z-dimension of the TE, saves it as an *AirfoilParameter* and determine the orientation of the airfoil. Afterwards, it will be filtered in the distance from the TE specified as input (*distanceFromTrailingEdge*). The result of the algorithm can be seen in Fig. 9.4.

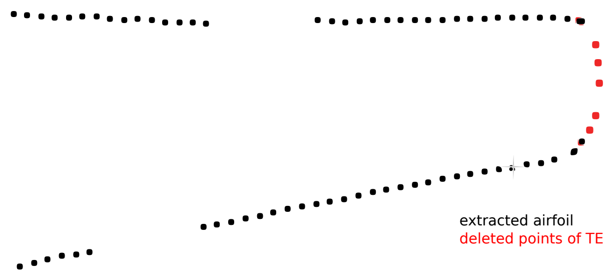


Figure 9.4: Airfoil with deleted trailing edge

9.7 findingMorphedReferencePoints()

This function will be used to find the references on the morphing wing. The references are profiles which where attached before the scanning process (see Kloiber (2021)). Please note, that the scaling is highly depending on an exact measurement of the position of the two references and can be slightly of whenever the scanning resolution is to low or the position or not exactly known.

The algorithm is shown in the diagram of Fig. 9.5. The references will be deleted using the *deleteMorphingWingReferences()* of the *Airfoil* class (see Sec. 2.8).

9.8 derotateToReferencePoints()

This function rotates, scales and translates the morphing wing on the two reference points. an illustration is shown in Fig. 9.6.

Since the original reference points were deleted with the references itself, the first step is to search for the two nearest points which remained on the airfoil. These will be then used to compute the rotation angle between the reference airfoil and the generated section. This has the benefit that it is sure that the angle contains no small error term if the found reference point was on the reference itself and not the wing. The foil is then rotated CW and CCW. The rotation with the smallest z-dimension is used as the derotated airfoil. Also the saved reference points of the section will be transformed accordingly. The scale is computed depending on the

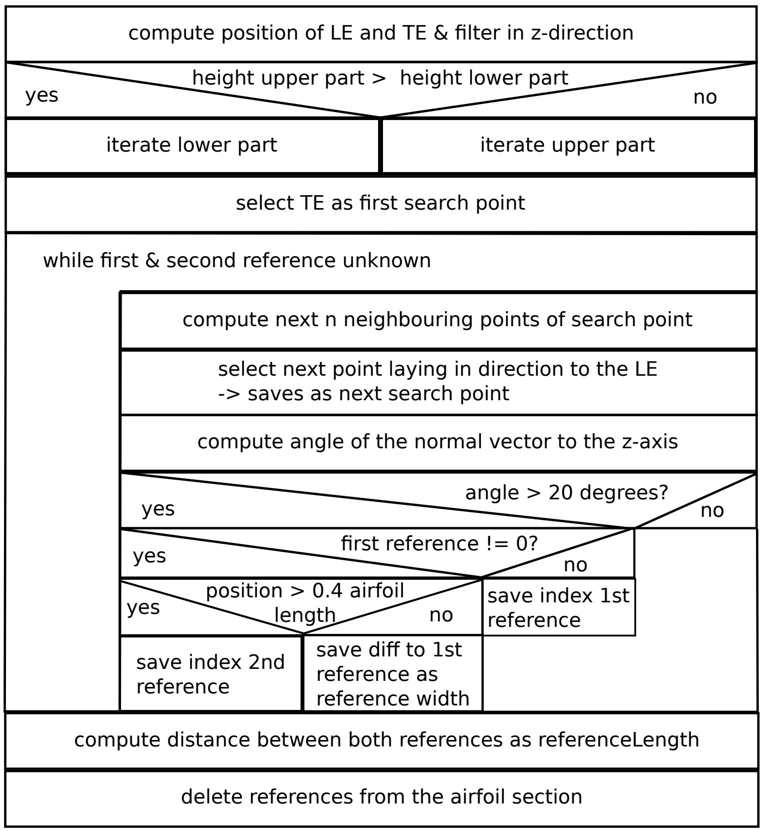

compute position of LE and TE & filter in z-direction
height upper part > height lower part
yes
no
iterate lower part
iterate upper part
select TE as first search point
while first & second reference unknown
compute next n neighbouring points of search point
select next point laying in direction to the LE
-> saves as next search point
compute angle of the normal vector to the z-axis
angle > 20 degrees?
yes
no
first reference != 0?
yes
no
position > 0.4 airfoil length
yes
no
save index 1st reference
save index 2nd reference
save diff to 1st reference as reference width
compute distance between both references as referenceLength
delete references from the airfoil section

Figure 9.5: Procedure for finding the reference points of the morphing wing

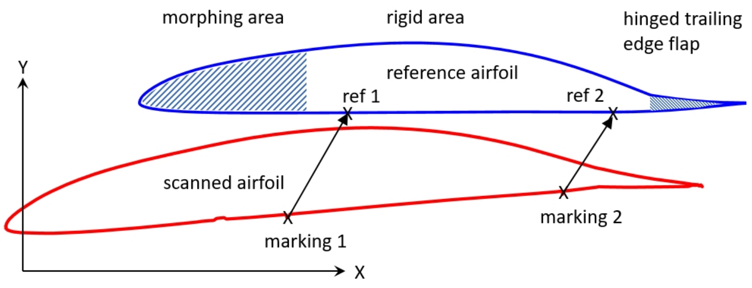


Figure 9.6: Coordinate transformation of scanned morphing airfoil with respect to reference airfoil Kloiber (2021)

given reference points and the *referenceLength* of the *MorphingWingParameter* class. The arifoil is scaled and afterwards translated using the function *translateSectionToReference()* (see Sec. 9.9).

9.9 translateSectionToReference()

The first found reference on the section will be translated to the first reference point. The y-positions will be deleted.

9.10 derotateFlap()

This function contains the complete derotation procedure. It will also call *getIndexFlapPosition()*. After a hinge line was successfully found, the flap will be filtered, derotated and concatenated (see Fig. 9.10).

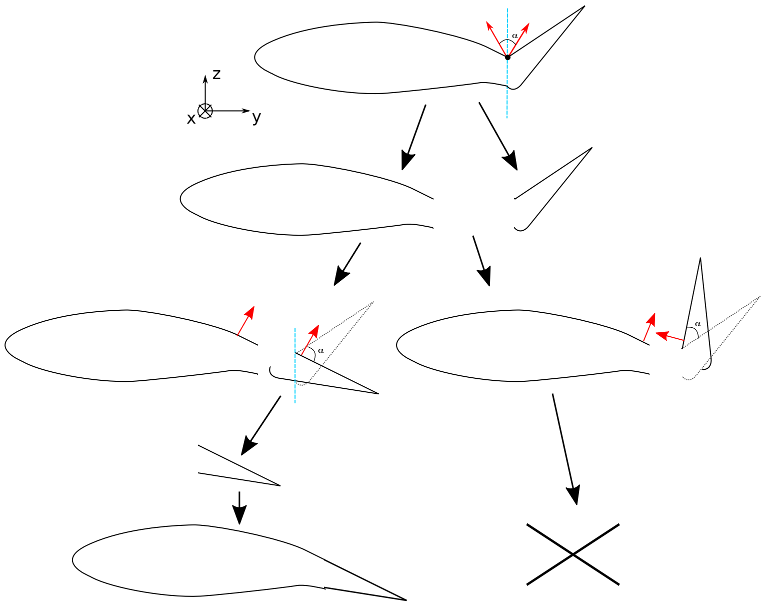


Figure 9.7: Procedure for derotating the flap (van Brügge, 2020, P.34)

The steps which have to be executed are also enumerated below in more detail:

1. find index of the hinge line
2. filter the point cloud at the hinge line
3. compute average normal vectors in direction of the TE and the LE
4. compute the angle between these normal vectors
5. find the index of the flap position in the filtered point cloud and save original position
6. rotate flap around computed angle CW and CCW
7. decide which rotation is correct using a comparison of surface normal vectors
8. filter overlapping points of the rotated flap
9. compute translation vector to concatenate the flap and the foil
10. save computed airfoil parameters

To compute the translation vector the function *getOriginalPosition()* is used.

9.10.1 getIndexFlapPosition()

This algorithm searches for the hinge line of the flap. To use it, the flap has to be actuated in direction of the hinge lie during the scanning process. The algorithm works similar to the one for finding the reference points of the morphing wing (see Sec. 9.7). The surface with the hinge line will be iterated until a difference in the angle of the normal vectors is found (see Fig. 9.8). The complete algorithm is illustrated in Fig. 9.9. The angle has to be relatively low since most flaps can just be actuated in a certain range.

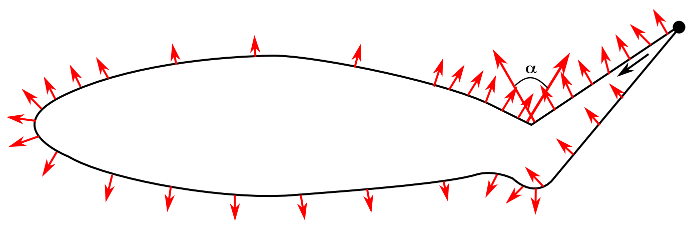


Figure 9.8: Finding hinge line using normal vectors (van Brügge, 2020, P.33)

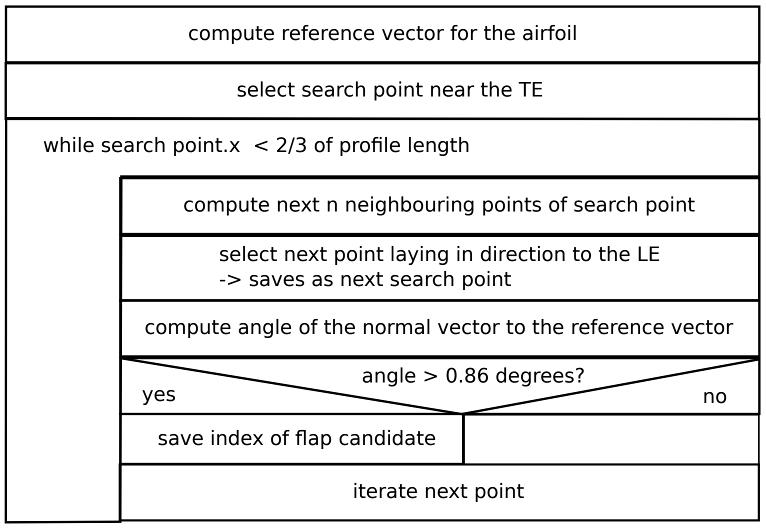

compute reference vector for the airfoil
select search point near the TE
while search point.x < 2/3 of profile length
compute next n neighbouring points of search point
select next point laying in direction to the LE
-> saves as next search point
compute angle of the normal vector to the reference vector
angle > 0.86 degrees?
yes
no
save index of flap candidate
iterate next point

Figure 9.9: Algorithm for finding the hinge line position of the flap

9.10.2 computeAverageNormalOfFoil()

This function uses a nearest neighbor search to iterate through the points beginning at a specific location. It can be chosen if it should be iterated in direction of the LE or TE. Every normal vector in this direction will be summed up. Afterwards the normalized normal vector will be returned as averaged surface normal.

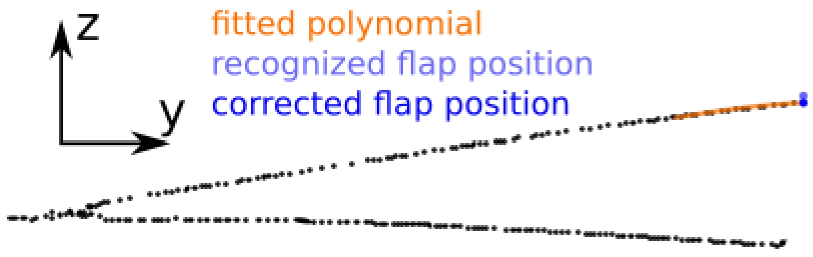


Figure 9.10: Correction of the flap position

9.10.3 getOriginalPosition()

This function is used to reconstruct the curvature of the airfoil before concatenating the flap to the foil. Therefore a polynomial approximation of GSL will be used. Firstly, using a nearest neighbor search the 10 next points will be selected. These will be then used to compute a polynomial of order 3. The coefficients will be obtained using the function 9.10.4. The z-value is then corrected using the fitted polynomial (see Fig. 9.10).

This correction is needed since it is possible that the recognized flap position does not fit exactly to the remaining foil (see e.g. Fig. 4.1).

9.10.4 getPolynomialCoeff()

This function is a fitting function for a polynomial of a specified order (input). It uses a QR solver of GSL to compute the coefficients of the polynomial. The coefficients for the fitting matrix are computed using Eq. 9.1.

$$a_{ij} = x_i^j \tag{9.1}$$

Bibliography

Drela, M. and Youngren, H. (4.3.2017), ‘https://web.mit.edu/drela/Public/web/xfoil’.
URL: *https://web.mit.edu/drela/Public/web/xfoil/* [Accessed on: 10.3.2020]

Galassi, M. e. a. (2018), ‘GNU Scientific Library Reference Manual’. Software. v2.6.
URL: *https://www.gnu.org/software/gsl/* [Accessed on: 2020-10-04]

Gomila, L. (2020), ‘Simple and Fast Multimedia Library’. Software. v2.5.1.
URL: *https://www.sfml-dev.org/* [Accessed on: 2020-10-04]

Grant Timmerman (2014), ‘Approximating Continuous Functions and Curves using Bernstein Polynomials’.

Kloiber, F. (2021), Geometrieanalyse von Flügelsegmenten mit formvariabler Vordersektion für ein Hochleistungssegelugzeug mittels 3D-Laserscan und Photogrammetrie, Semester thesis, Technical University Munich.

Kulfan, B. M. and Bussoletti, J. E. (2006), "fundamental" parametric geometry representations for aircraft component shapes, *in* ‘11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference’. doi: 10.2514/6.2006-6948.

Lina van Brügge (2022), ‘Refinement of the 3D-Scanning Toolchain for the Characterization of Aerospace Structures’.

Rosin, P. (2000), ‘Fitting superellipses’, *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **22**, 726 – 732. doi: 10.1109/34.865190.

Rusu, R. B. and Cousins, S. (2011), 3D is here: Point Cloud Library (PCL), pp. 1–4. Proceedings in IEEE International Conference on Robotics and Automation 2011. doi: 10.1109/ICRA.2011.5980567.

Steffen, M. (1990), ‘A simple method for monotonic interpolation in one dimension.’, *A&A* **239**, 443–450. doi: https://ui.adsabs.harvard.edu/abs/1990A&A...239..443S.

van Brügge, L., Çetin, K., Köberle, S., Thiele, M., Sturm, F. and Hornung, M. (2021), Application of 3D-Scanning for Structural and Geometric Assessment of Aerospace Structures, *in* 'Deutscher Luft- und Raumfahrtkongress 2021'.

van Brügge, L. (2020), Development and Implementation of a 3D-Scanning-Toolchain for the Geometric Characterization of UAVs, Bachelor's thesis, Technical University Munich.

XFLR5 (3.4.2022).
URL: *http://www.xflr5.tech/xflr5.htm* [Accessed on: 3.4.2022]