EdFoil
User Guide

Version 0.3

Miguel A. Valdivia-Camacho

Contents

1 Introduction

Setting up a numerical model is often the most time-consuming stage of any analysis. Preparing the geometry, applying material definitions, and defining boundary conditions can take considerably longer than running the simulation itself. This user guide introduces the **EdFoil** framework designed to accelerate the modelling process for composite blades. By automating the generation of airfoils, stations, and laminate sections, the tool aims to reduce setup time and ensure consistency across different models. In addition, it takes advantage of modern computing hardware, allowing engineers and researchers to start modelling full three-dimensional composite blades more efficiently than with traditional two-dimensional workflows. **EdFoil** is built with Qt, the C++ framework for software development.

2 Installation

EdFoil offers three ways to be used, depending on the needs of the user. It can either be installed in the machine, initialised straight from an executable, or cloned from GitHub. The software installer is recommended for most users, as it is the simplest method to run the program.

Note: The GitHub repository is the most up-to-date version of **EdFoil** as new features are added to the base framework first. Thorough validation is required before any future feature is implemented in the application.

2.1 Software Installer

This section provides a step-by-step guide for installing the **EdFoil** software through the installer package. The process follows a standard installation wizard, and each step is illustrated with a screenshot for clarity.

Note: Users can complete the installation and launch the program without difficulty by pressing "**Next**" for all the steps.

The installer starts with a welcome window. It is recommended to close all other applications before continuing.

Click "**Next**" to proceed.

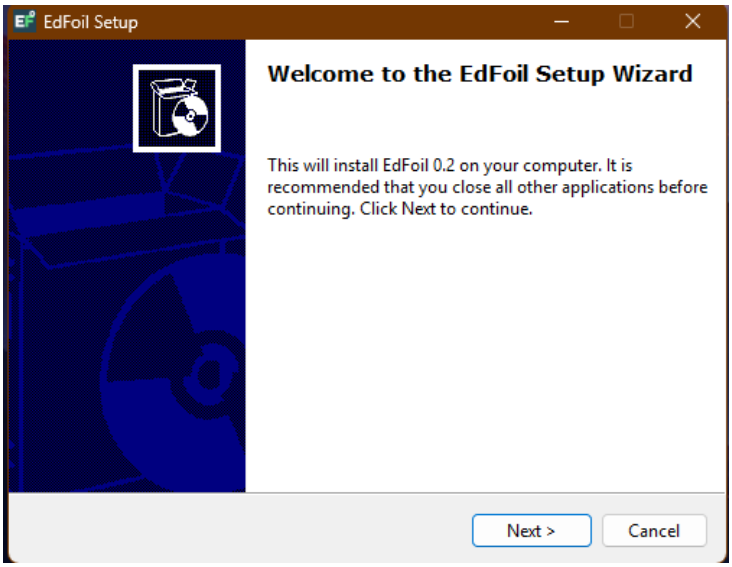


Figure 1: Welcome screen (Step 1)

Next, the license agreement is displayed. Read through the GNU General Public License and accept it by selecting "I accept the agreement." Then click **"Next"**.

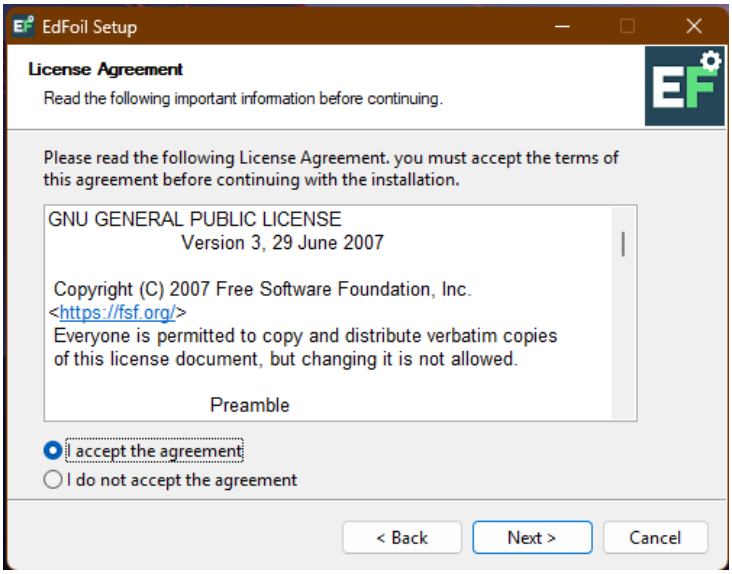


Figure 2: License agreement (Step 2)

Choose the installation folder. By default, **EdFoil** will be installed in `"C:\Program Files (x86)\EdFoil"`.

You can change the directory if required, then click "**Next**".

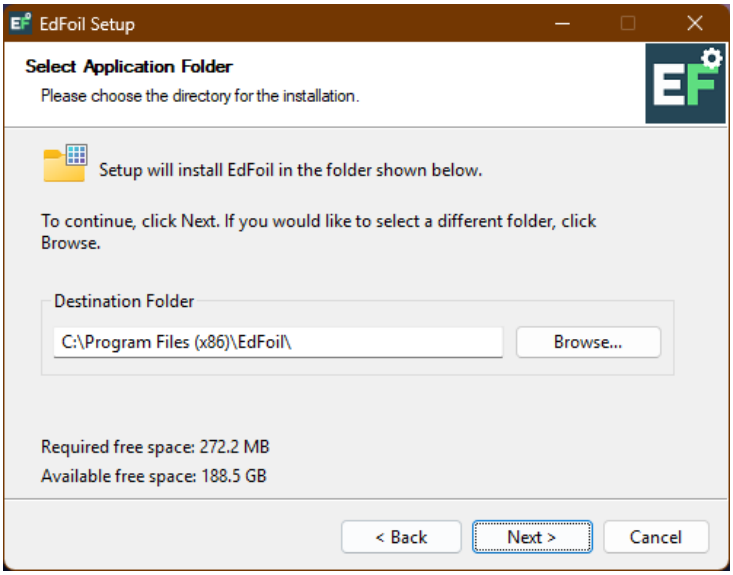


Figure 3: Installation folder (Step 3)

Select any additional tasks you would like the installer to perform, such as creating a desktop icon or a start menu folder. Once selected, click "**Next**".

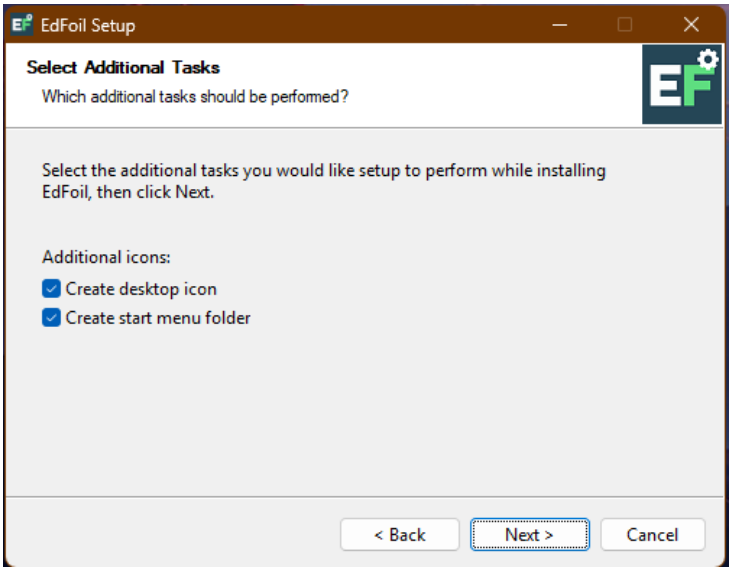


Figure 4: Additional tasks (Step 4)

The installation process begins. Files are copied to the selected directory and installation

should take up to two minutes.

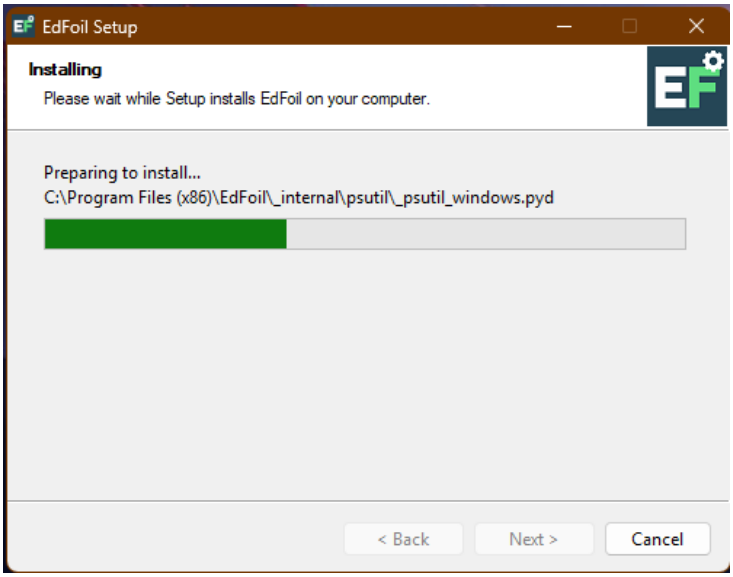


Figure 5: Installing (Step 5)

Once installation is complete, a final window appears. You can choose to launch **EdFoil** immediately. Click "**Finish**" to exit the installer.

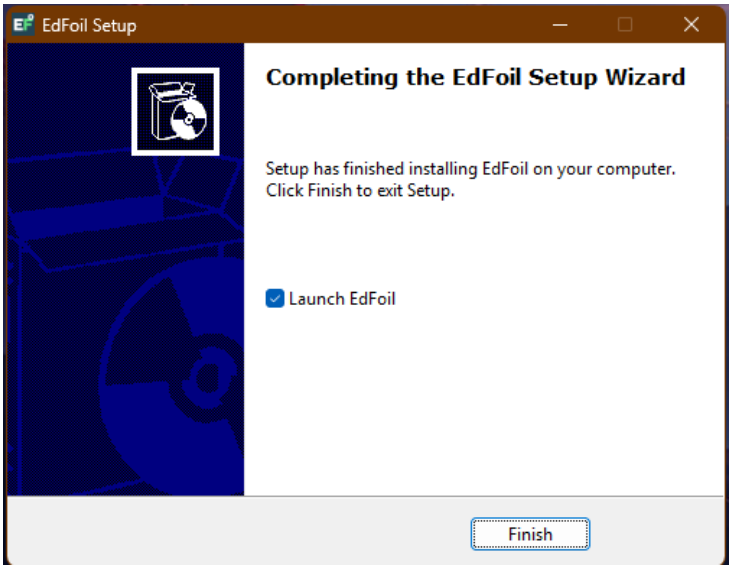


Figure 6: Completing setup (Step 6)

After installation, the **EdFoil** application opens to the home screen. Here, you can start a

new project or load an existing one. The application opens a terminal in the background which it is intended to show execution statements and errors whilst using the software.

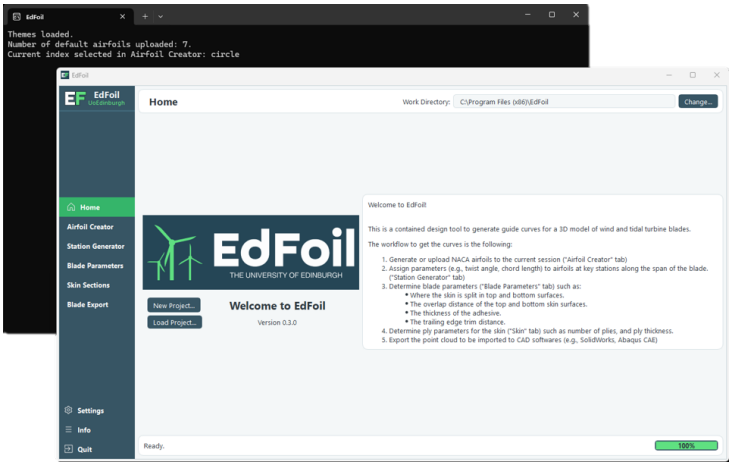


Figure 7: Launching EdFoil (Step 7)

2.2 Software Executable

A zip file is available for download that once extracted can run the software straight from the executable file (See Figure 8). It is important to note that the "_internal" folder must be present for the executable to load all the required packages.

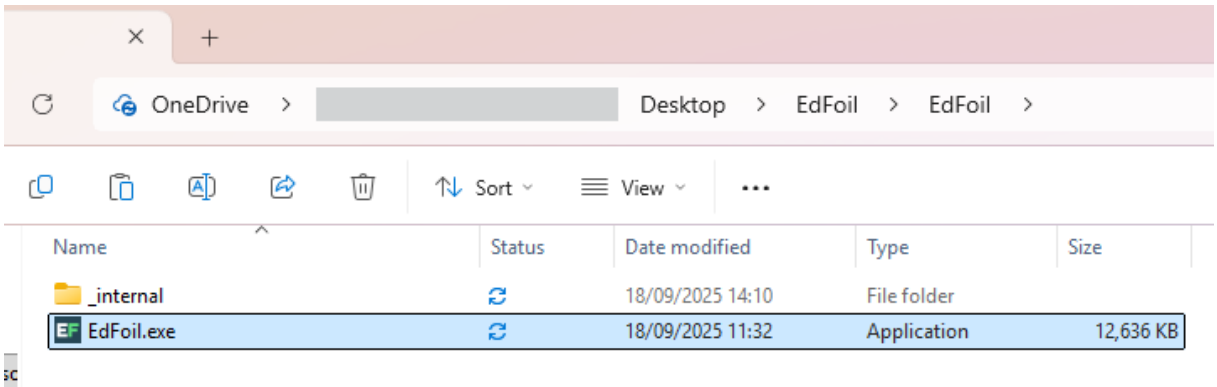


Figure 8: Files inside the zip file.

2.3 GitHub Clone

EdFoil is also available as a GitHub repository, which gives access to all the modules and packages. If the user intends to clone the repository, it is required to have git installed

in the machine. The official website of git[1] has an installer for all operating systems. However, **EdFoil** has only been tested in the Windows operating system. The following steps are needed to use **EdFoil**:

1. Clone the repository with the following command. (Open cmd in the desired root directory)

```
git clone https://github.com/MiguelAVC/TBlade-Designer.git
```

2. Install python version 3.12.2 (or 3.12) in your computer. This version is available in the official Python website.[2]

3. Create a python virtual environment inside the root directory. This can be done in the terminal with the following command:

```
python -m venv .venv
```

4. Install all the required dependencies with the following commands:

```
.venv\Scripts\activate
pip install -r requirements.txt
```

5. Run the main compiler in the terminal:

```
python main.py
```

3 Modules

Regardless of the installation method used, **EdFoil** greets you with a home page as seen in Figure 9. The work directory is set to the installation location by default, however, it can be changed to the desired location. This path is important for exporting tasks.

[1] https://git-scm.com/downloads
[2] https://www.python.org/downloads/

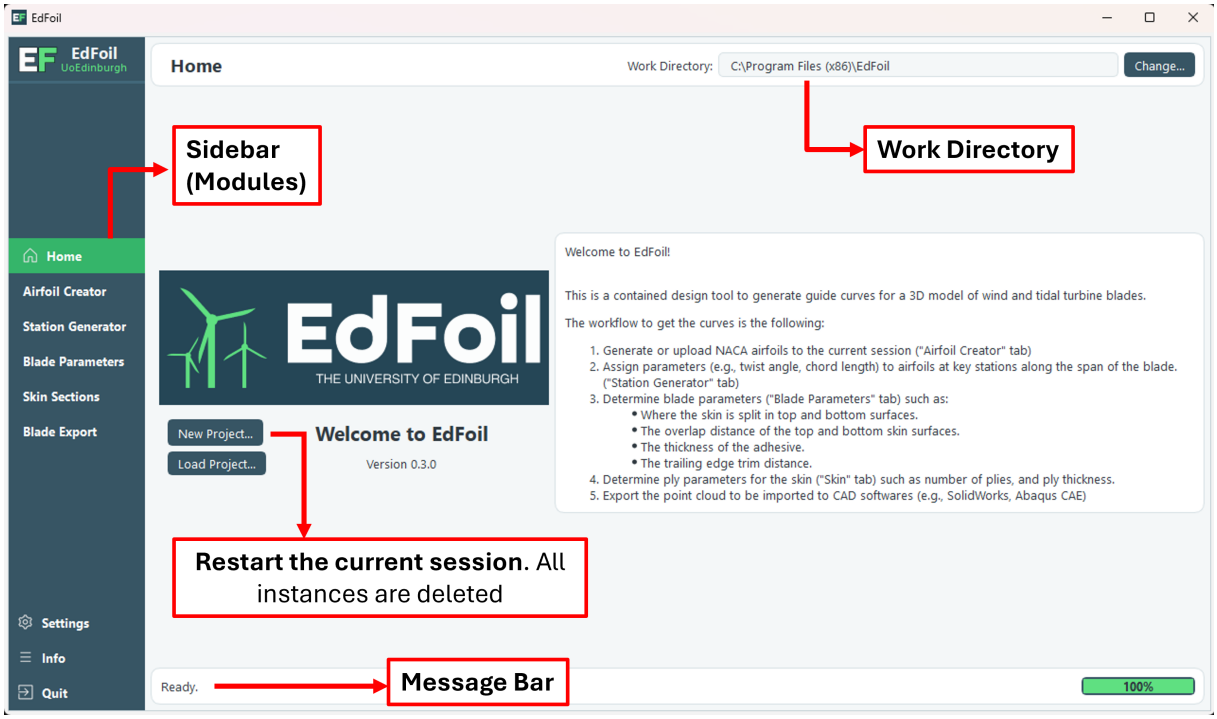


Figure 9: EdFoil home page.

A simplified workflow to use **EdFoil** is described in the home page. This workflow follows the order of the sidebar modules. All the modules are linked together through three main class objects:

- Airfoil class
- Station class
- Section class

The properties, methods, and signals of these classes are detailed in Section 5.

3.1 Airfoil Creator

The first step involves the "Airfoil Creator" tab shown in Figure 10. In this tab, the user can create different airfoils based on three available NACA families. The NACA families available in this version are: NACA 4-digit series, NACA 6-series, and NACA 6A-series. The digits in each family of airfoils are related to different aerodynamic factors. For instance, Figure 11 explains the meaning behind each digit and how it influences the

shape of the airfoil. More information about all the NACA families and their digits can be found in the Aerospace website.[3]

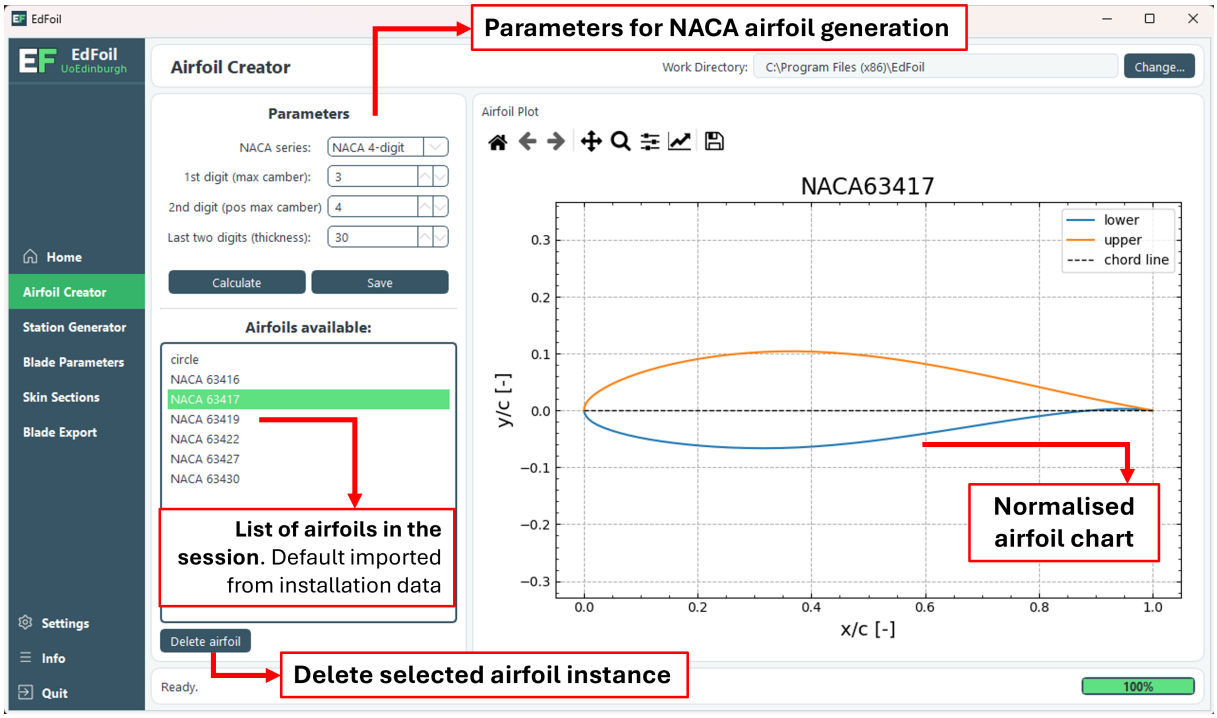


Figure 10: Airfoil creator tab.

The installation provides some examples of airfoils that are loaded when **EdFoil** is initiated. New airfoils can be generated by selecting the NACA family and describing the rest of the parameters. The generated airfoil is shown in the graph, which should effectively plot the correct upper and lower surface, along the chord length. Every airfoil is normalised, meaning it goes from 0 to 1 in the x axis, where the leading edge (LE) should be located in the coordinates (0,0) and the trailing edge (TE) should be located in (1,0). An airfoil will show an asterisk next to its name (e.g., NACA 63543*) if it has not been saved yet.

[3] https://aerospaceweb.org/question/airfoils/q0041.shtml

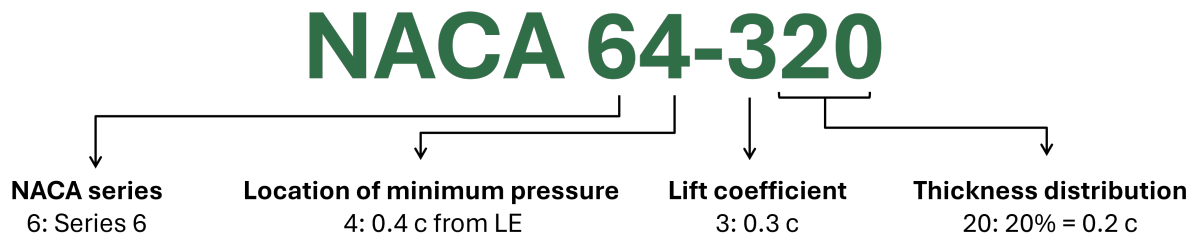


Figure 11: NACA 6-series digit nomenclature.

The generation of coordinates for some of the NACA families can be complex and no fundamental equations are given but rather tabulated data to use for interpolation. Figure 12 shows the difference in the coordinates for the same NACA airfoil (NACA 63-418 and NACA 63-430) between another open software GnacaLt[4] and tabulated data from Abbot[5]. **EdFoil** is based on the Public Domain Aeronautical Software (PDA) when creating airfoil coordinates and has been validated with the sample cases provided in its documentation[6].

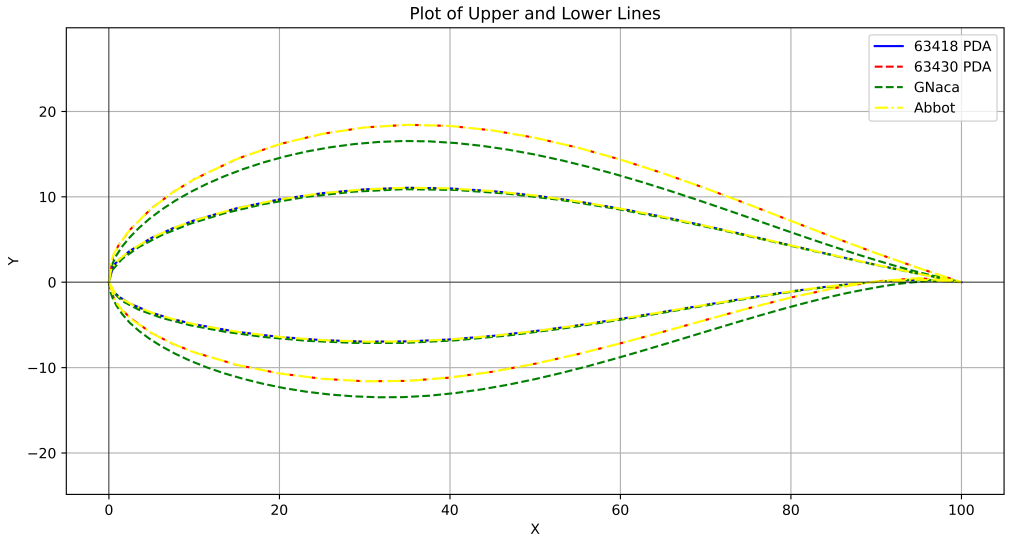


Figure 12: Coordinate accuracy for different methods of airfoil creation.

[4] https://tracfoil.com/airfoils/index.php?page=en_gnaca
[5] https://ntrs.nasa.gov/citations/19930090976
[6] https://www.pdas.com/index.html

3.2 Station Generation

The **"Station Generation"** module allows to turn the normalised airfoils into transformed stations across a blade design. The first tab in this module (shown in Figure 13) shows the interactive page to create a new station instance. Stations are saved in the current section with the convention "sta_z", where "z" stands for the distance along the span of the blade.

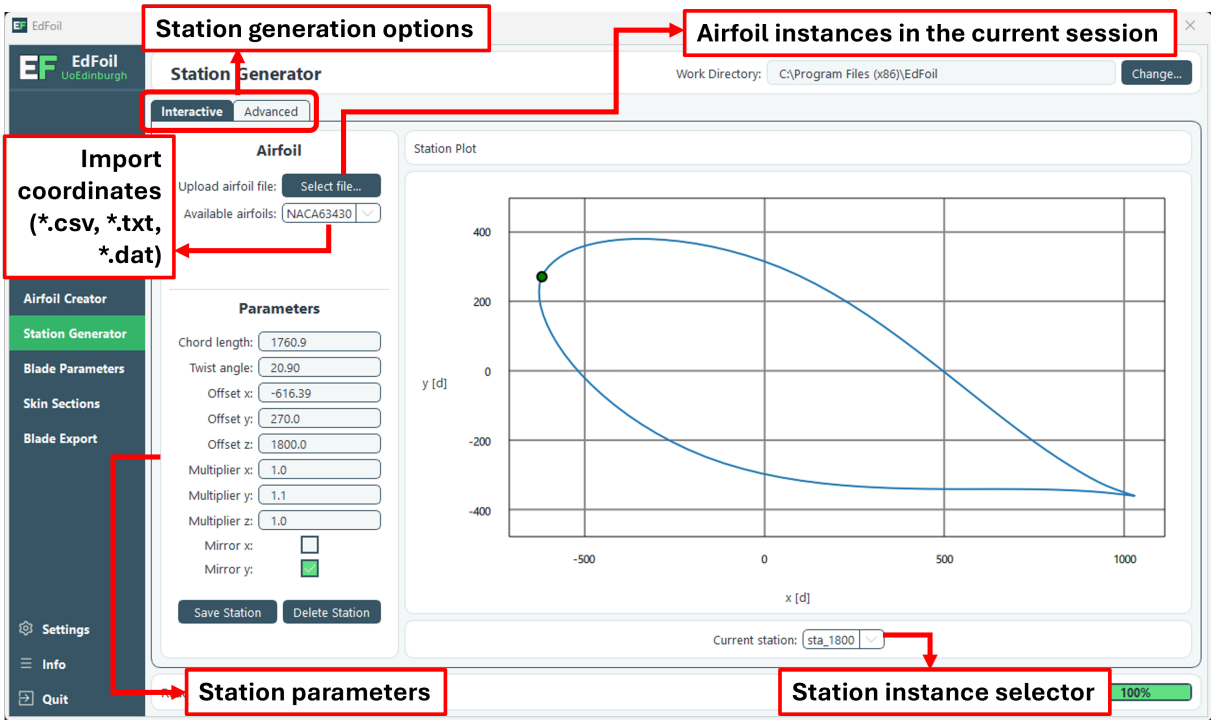


Figure 13: Interactive station generation tab.

3.2.1 Airfoil Selection

Airfoil data can be imported from different files and **EdFoil** will attempt to create new airfoil instances from them. The files should contain coordinates and the current accepted formats are: *.csv, *.txt, and *.dat. All the airfoil instances in the current session are shown in the dropdown menu. The airfoil graph in this tab updates any time the selection changes.

3.2.2 Airfoil Parameters

The station class has several parameters required to be created. This turns the airfoil instance into a station instance after rotating, scaling, and translating the normalised coordinates. The main parameters are the following:

- **Chord Length:** Distance from the leading edge to the trailing edge.
- **Twist angle:** Clockwise rotation in degrees from the horizontal axis.
- **Offsets:** Translation coordinates for the entire airfoil (x, y, z).
- **Multipliers:** Scaling factors for all 3 coordinates (x, y, z).
- **Mirroring:** Coordinate flipping for the XY-plane (x, y).

The airfoil plot updates itself every time one of these parameters is changed. The graph also includes a green square, highlighting the current location of leading edge (LE). The new station instance generated needs to be saved to be included in further steps of the workflow.

Note: Stations are saved based on their distance across the blade. Saving a station will override any existing station instance at same Z-value.

3.2.3 Advanced Station Generation

A more advanced and optimised tab (shown in Figure 14) is available to create multiple stations simultaneously. There is no limit in the number of stations that can be created, however, all must include the parameters mentioned in the previous section. The first column "Airfoil" includes a dropdown menu that shows all the available airfoil instances in the current section. The stations can be created in any order, and the **"Sort Stations"** button allows the user to sort all the stations based on the "z-offset" column.

Import from file...: This option takes a file (*.txt, *.csv) and populates all the columns in the creation table. It should contain the header and be separated by commas. The message bar prints out the file path if it has been imported successfully. As an example, a file with two stations would contain the following data:

```
airfoil,chord,twist_angle,offset_x,offset_y,offset_z,multiplier_x,multiplier_y,multiplier_z,mirror_x,mirror_y
circle,311.05,28.5,0,0,1200,1,1,1,False,True
NACA63416,739,4.8,-260,20,6250,1,1,1,False,True
```

Note: The "Airfoil" column defaults to the first airfoil instance if the data provided has not been generated in the current session.

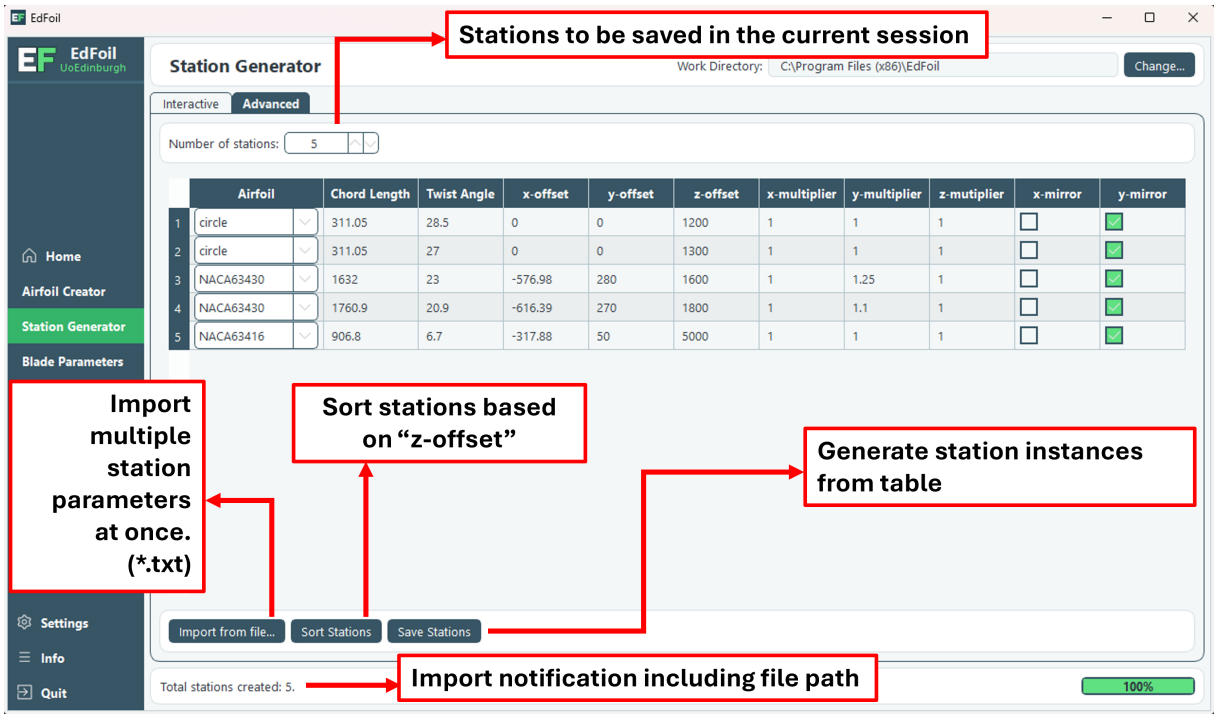


Figure 14: Advanced station generation tab.

3.3 Blade Parameters

The "Blade Parameters" tab involves variables that will impact all the station instances generated. In its current state, it calculates the location and length of the overlap between the top and bottom skin surfaces. However, this module is intended for further component implementation. The general view of this tan be observed in Figure 15.

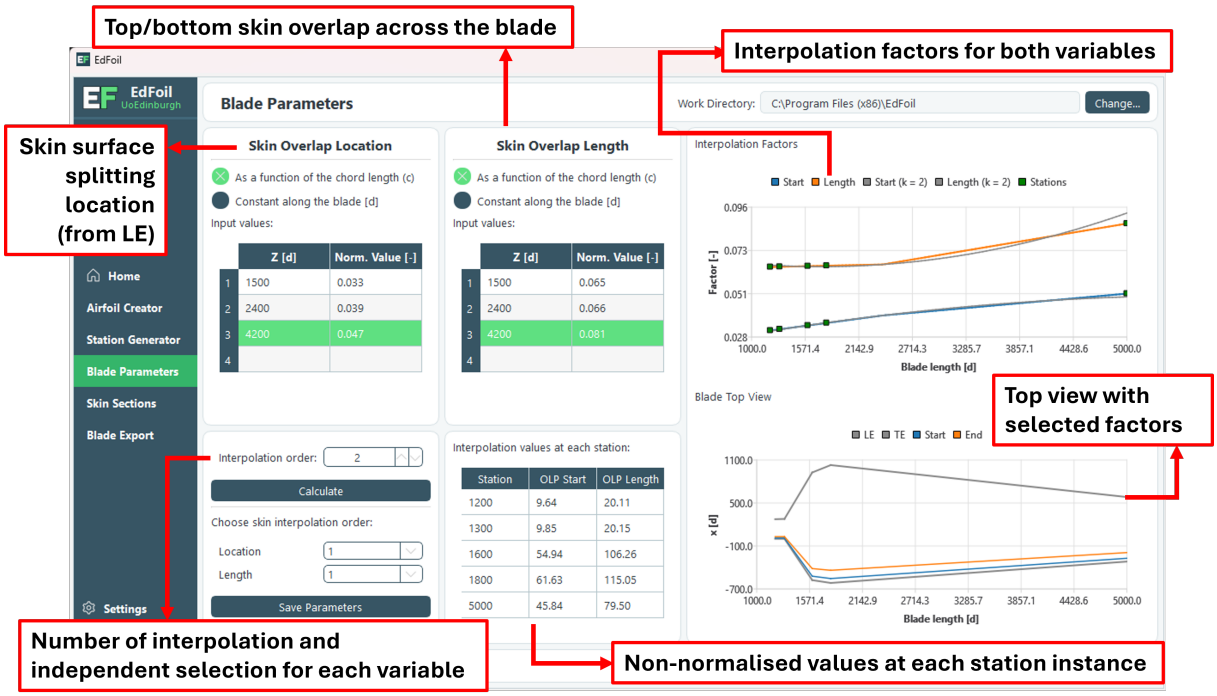


Figure 15: Blade parameters tab.

The location of this overlap is determined by its distance from the leading edge (LE) along the chord length. The diagram in Figure 16 shows the location of this overlap in relationship with the chord length of a station, where c is the chord length at the given station, c_1 is the distance from the leading edge to the start of the skin overlap, and c_2 is the length of the overlap along the chord length axis.

Note: The overlap can only be located in the top surface of the station. Future implementation will include the option to have it in the bottom surface.

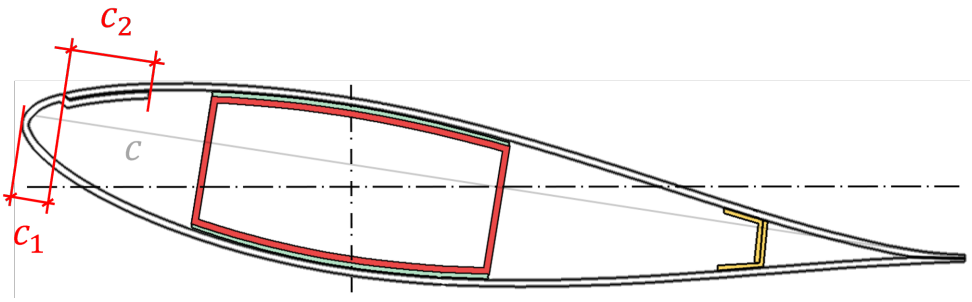


Figure 16: Skin overlap diagram

The interpolation depends on known values at different distances from the root of the blade. There are two options to calculate these distances based on the user's data availability:

- **As a function of the chord length (c):** The second column "Norm. Value [-]" is the distance normalised to the local chord length.
- **Constant along the blade [d]:** Only a scalar is needed and this will be replicated in all stations.

If the parameters are defined based on the chord length, the user can decide the number of interpolations to make and fill the tables accordingly. For an interpolation order of n, $n + 1$ values are required to run the interpolation before clicking on the **"Calculate"** button. The default is the linear interpolation, however a higher number can be chosen. For example, quadratic interpolations are also assessed in Figure 15. The graphs show the distance and length of this overlap and how the different interpolation orders have an effect on each station instance. This can be useful when optimising the volume of material needed. A table shows up with the preview of the non-normalised values at each station instance in the current session. These values are not saved until the user clicks on the **"Save Parameters"** button.

Note: When the second option (Constant along the blade [d]) is selected, the bottom left box regarding interpolation is not needed and hides from the user.

3.4 Section Generation

In the "Section Generation" tab, the station instance is grabbed and a section instance is generated based on the parameters chosen in this module and in the "Blade Parameters" module. Figure 17 shows an overview of a generated section instance with 8 plies and a bond layer. This is only a preview and section instances must be saved with the **"Save Section"** button for them to be exported.

Note: The section preview changes automatically every time a parameter is changed.

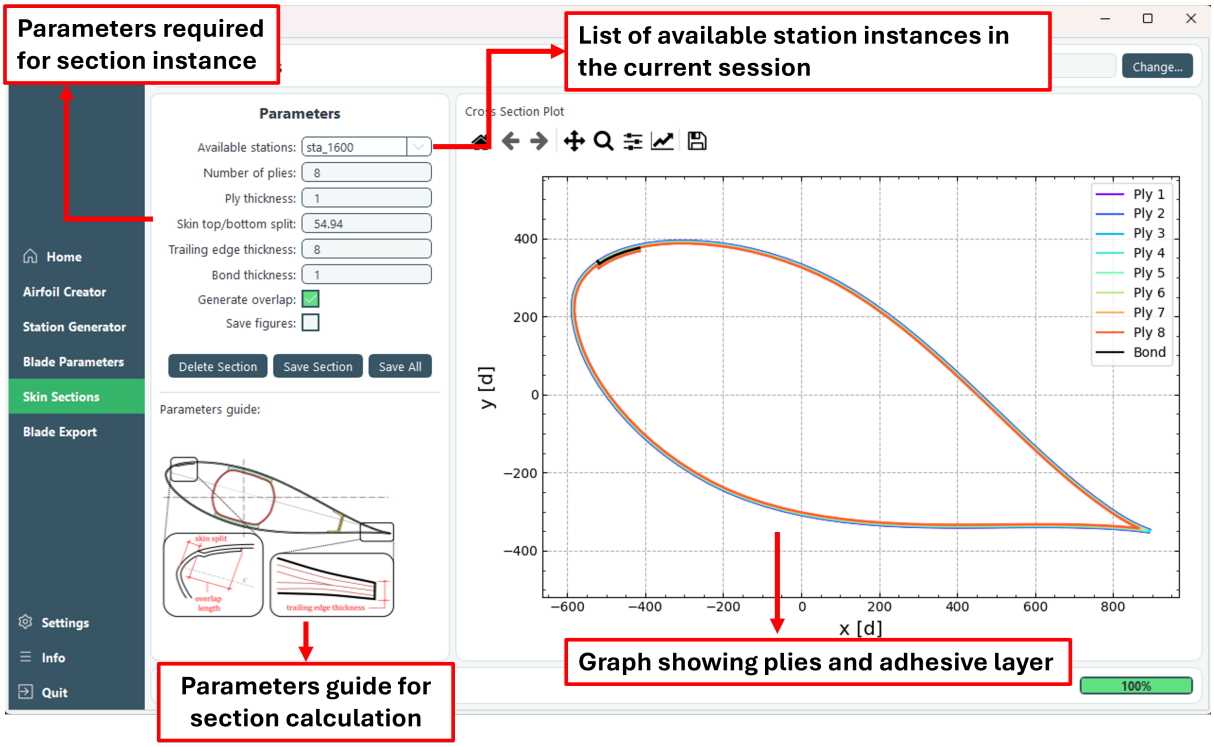


Figure 17: Section generation tab.

The section parameters are the following:

- **Available stations:** A dropdown menu with all the station instances saved in the current session.
- **Number of plies:** Constant across the blade and similar for bottom and top skin surfaces.
- **Ply thickness:** Constant and similar for all plies, default value is 1.
- **Skin top/bottom split:** Minimum distance from the leading edge (LE) to the skin overlap. This value cannot be changed as it is calculated from the previous module.
- **Trailing edge thickness:** Length of the trailing edge normal to the chord line. This avoids a singular point at the trailing edge.
- **Bond thickness:** Thickness of the bond layer between the top and bottom skin surface.
- **Generate overlap:** This option generates the entire overlap when checked, otherwise the method only generates the splitting location of the skin. (See Figure 18a)

- **Save figures:** Figures are saved in the current work directory when checked.

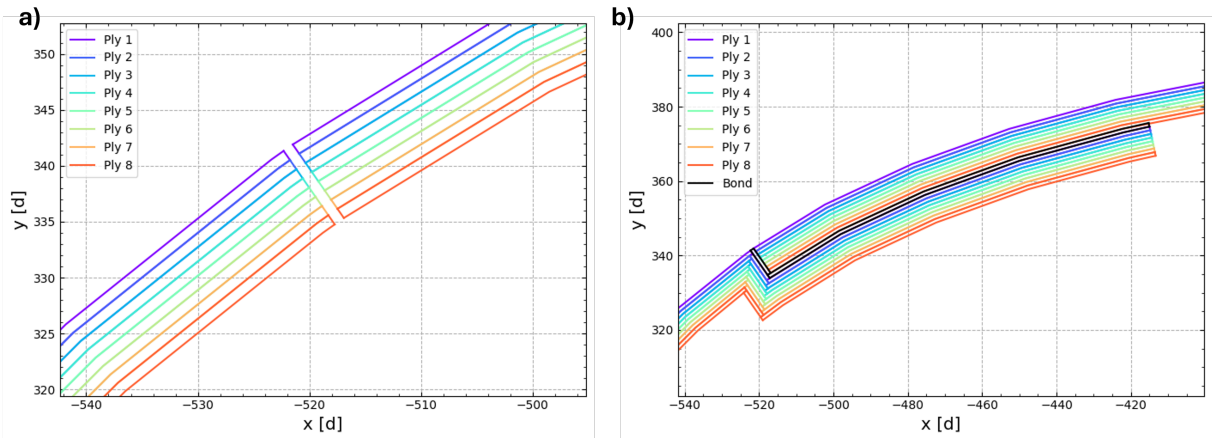


Figure 18: Overlap options: a) with **"Generate overlap"** unchecked, b) with **"Generate overlap"** checked.

The user can also click on the **"Save All"** button to skip the station-by-station process and generate a section for all available station instances. It is important to note that this option takes the same parameters for all stations and station customisation is not possible.

Note: The only parameter that changes for each station when using the **"Save All"** option is the "Skin top/bottom split" parameter if the blade parameters were interpolated in the previous module.

3.5 Export Module

The export module takes all the section instances in the current session and provides options to export their coordinates as cloud points. The overview of this module can be observed in Figure 19. In this tab, we can select one of the two output formats:

- **Single file (JSON):** Compact and structured *.json file that includes all sections, and their respective plies. It can be easily loaded for further analysis.
- **Multiple files (CSV):** Each curve is exported as a separate *.csv file. The curves are grouped based on the ply number.

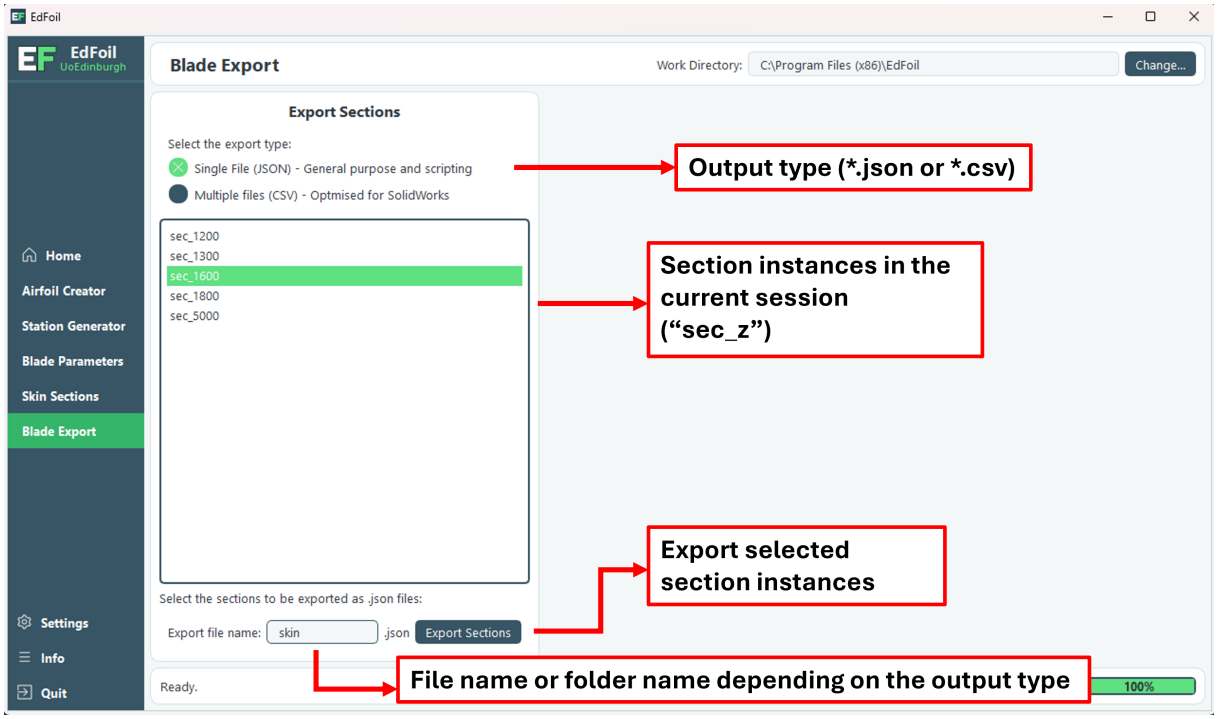


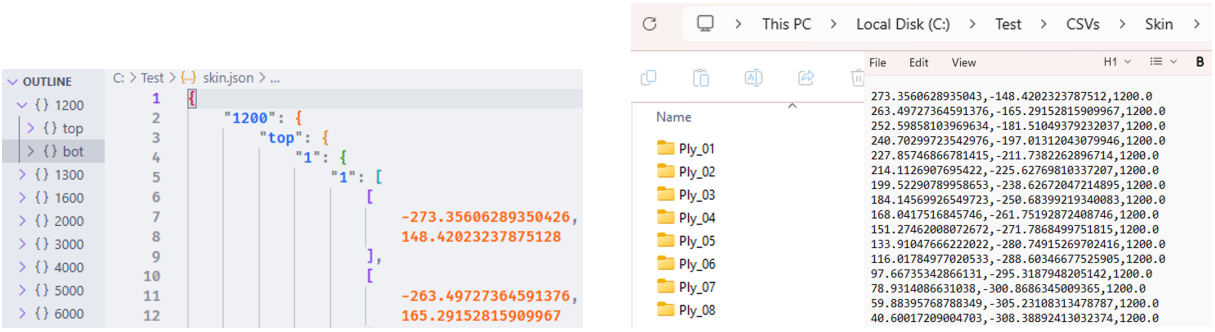
Figure 19: Section export tab.

Examples of the output types are shown in Figure 20. The single *.json file in Figure 20a follows the structure:

```
{Section} -> {Skin side} -> {Ply number} -> {Curve} -> {coordinates}
```

On the other hand, the *.csv output format in Figure 20b shows the following structure inside the export folder:

```
/{Ply number}/{Skin side}_{Section}_{Curve}.csv
```



(a) Single *.json file.

(b) Multiple *.csv files.

Figure 20: Export format examples.

Regardless of the output format selected, the user must select all the sections required to be exported. The file name or file folder can be changed in the bottom field of the tab before pressing the "**Export Sections**" button. The output will be created inside the current work directory.

4 Examples

4.1 Example A: Generating curves using the GUI

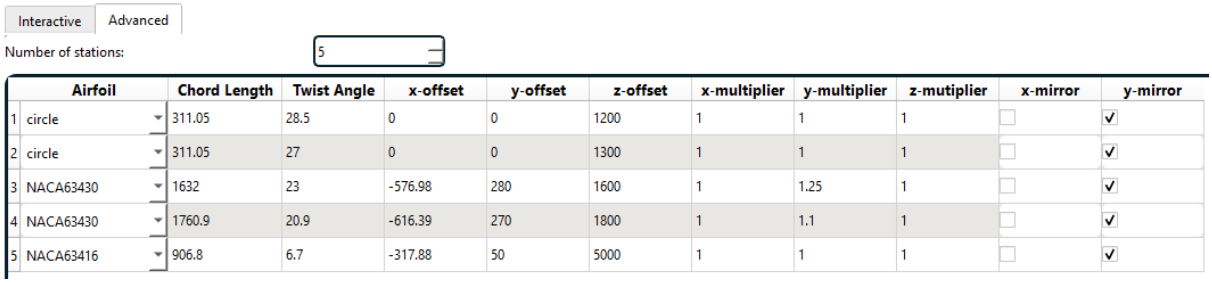
In this example, we generate the cloud of points for a 4-metre tidal blade using the GUI, exporting the skin surface to CSV files, and finally importing them to SolidWorks. The data for each station along the blade is saved in the file "**data.txt**", located by default in "C:\Program Files (x86)\EdFoil_internal\resources\". The station parameters for this 4 stations are shown in Table 1. We use the SI units for this current session (mm).

Table 1: Example A: station data.

Airfoil	Chord	Twist Angle	Offset X	Offset Y	Offset Z	Scale Y	Mirror Y
circle	622.10	28.5	0	0	1200	1	True
circle	622.10	27	0	0	1300	1	True
NACA63430	1632	23	-576.98	280	1600	1.25	True
NACA63430	1760.9	20.9	-616.39	270	1800	1.1	True
NACA63416	906.8	6.7	-317.88	50	5000	1	True

Once the file is imported in the advanced station generator tab, the table is populated

with our data as shown in Figure 21. We proceed to save these station by clicking **"Save Stations"**.



	Airfoil	Chord Length	Twist Angle	x-offset	y-offset	z-offset	x-multiplier	y-multiplier	z-mutiplier	x-mirror	y-mirror
1	circle	311.05	28.5	0	0	1200	1	1	1	☐	☑
2	circle	311.05	27	0	0	1300	1	1	1	☐	☑
3	NACA63430	1632	23	-576.98	280	1600	1	1.25	1	☐	☑
4	NACA63430	1760.9	20.9	-616.39	270	1800	1	1.1	1	☐	☑
5	NACA63416	906.8	6.7	-317.88	50	5000	1	1	1	☐	☑

Figure 21: Example A: Station import from *.txt file.

In the next module we define the location and the length of the skin overlap across the blade. We have measurements of its location at 3 different distances from the root. We increase the number of interpolation orders from 1 to 2 and fill the data in the table. For the length of the overlap we specify that we want a constant length across the blade of 200 mm.

After filling all the data, we press **"Calculate"** and the graphs are generated, as well as the table with the non-normalised values across the different stations. The top graph shows that with a quadratic interpolation the overlap would be located closer to the leading edge in the final station (Station 5000). With that in mind, we select the second interpolation order for the skin overlap and see values updated in the bottom table.

We press **"Save Parameters"** to save these parameters.

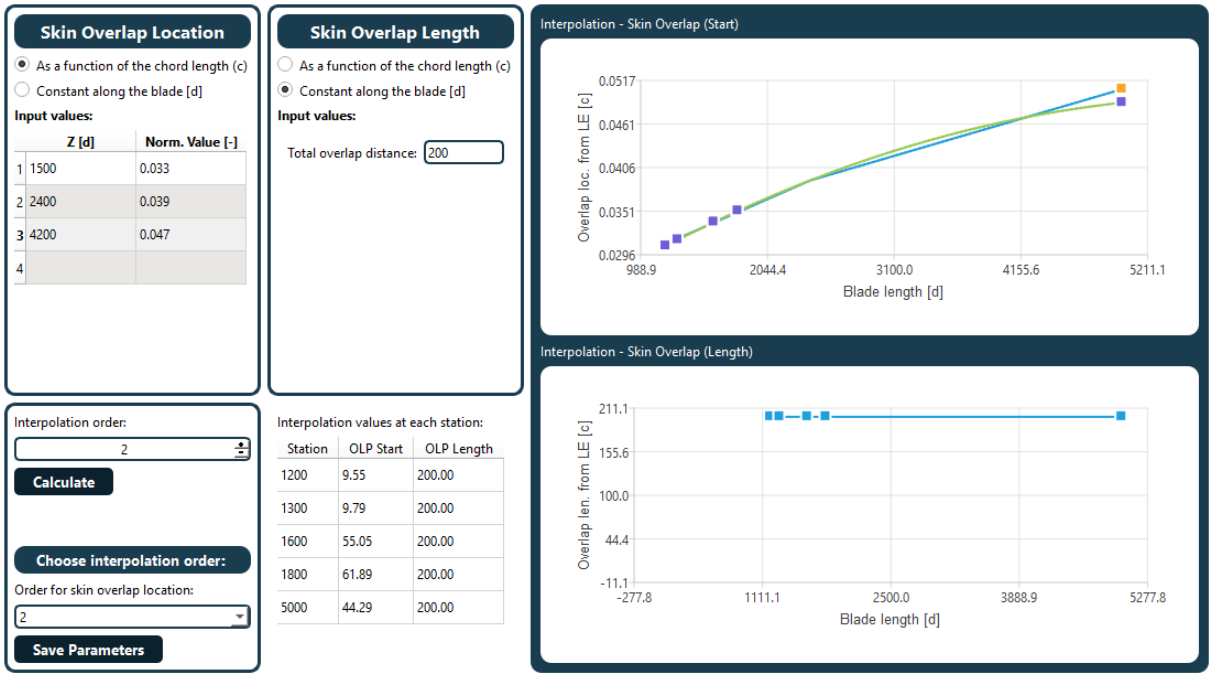


Figure 22: Example A: Skin overlap parameters.

We continue to the next module "Skin", where the first preview (shown in Figure 23) is calculated automatically for the first station (Station 1200). We maintain the default values as we want 8 1-mm plies in the section, a trailing edge thickness of 8 mm, and a bond thickness of 1 mm. We notice that **EdFoil** does not generate the overlap in circular section. This is by design as it is common for this section of the blade to be attached to the root connection. Hence the circular section is only split between the top and the bottom surface. Non-circular airfoils generate all the subsequent methods.

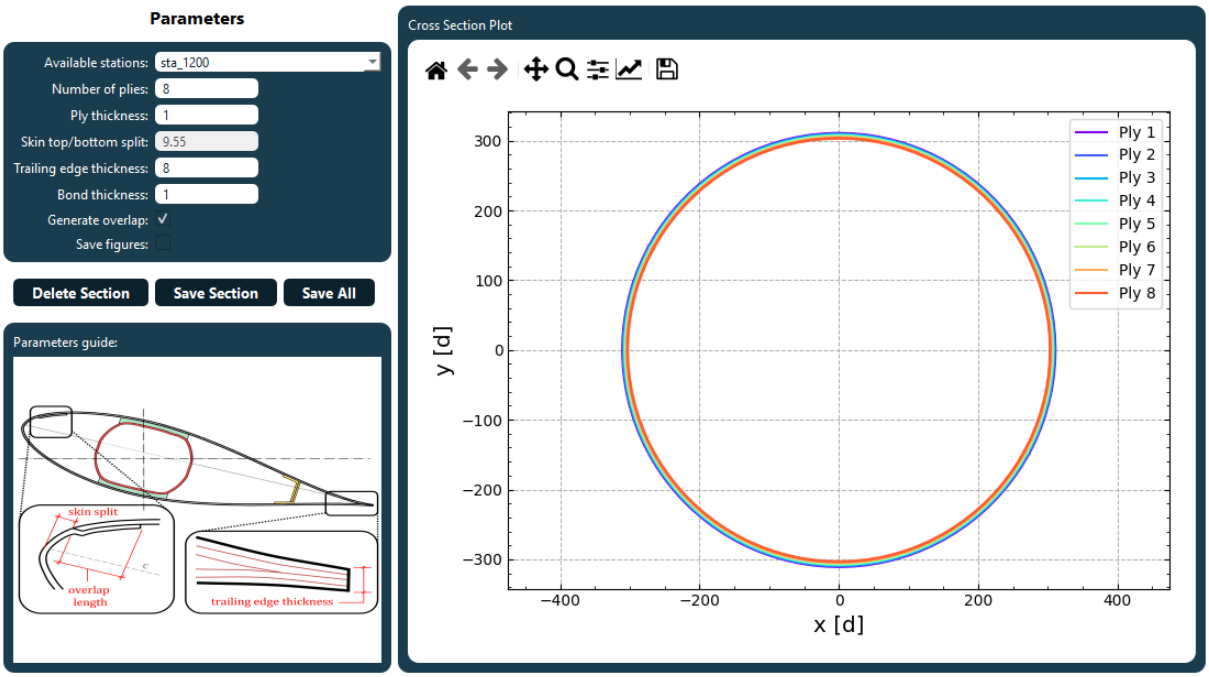


Figure 23: Example A: First section instance "circle".

We further inspect other sections by selecting a different station in the first parameter menu, such as the last section (Station 5000) in Figure 24. The overlap is generated and we can observe the trailing edge trimmed. A closer inspection to the skin overlap shows it was calculated correctly (see Figure 25. We can also observe the bond layer starting from the outermost layer all the way to the end of the overlap, effectively bonding the entire interface between these two surfaces.

We select **"Save All"** to apply the same parameters to all 5 stations, effectively generating 5 section instances for the next module.

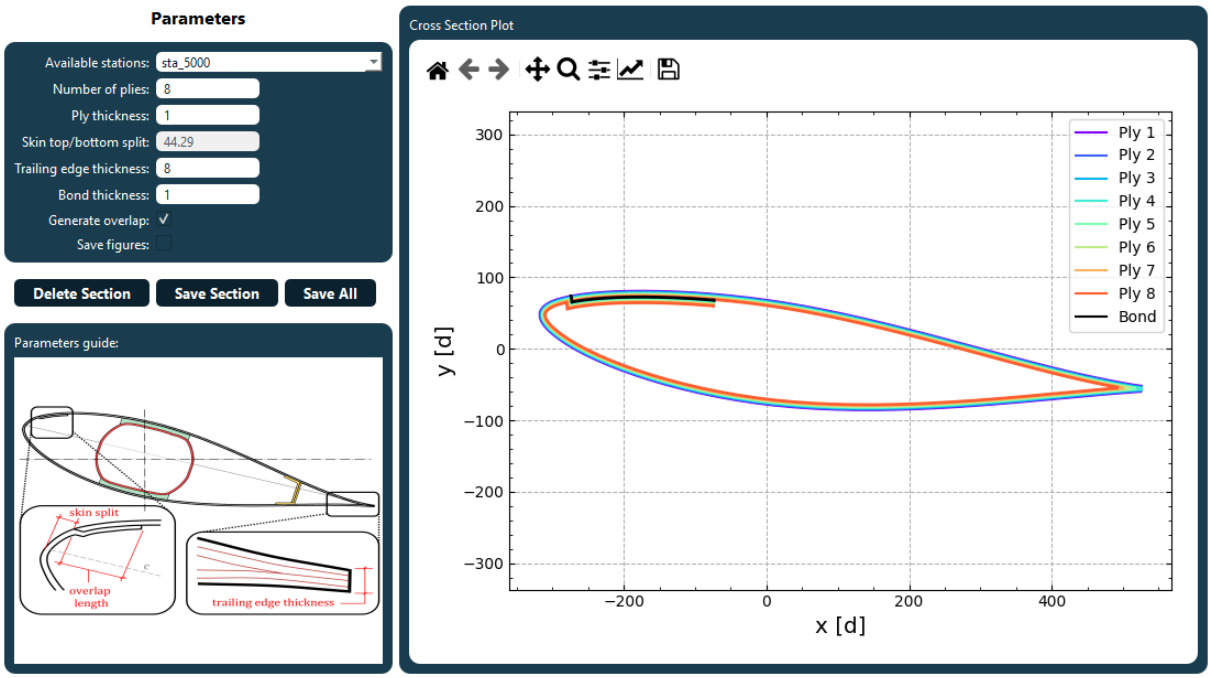


Figure 24: Example A: Last section instance "NACA 63416".

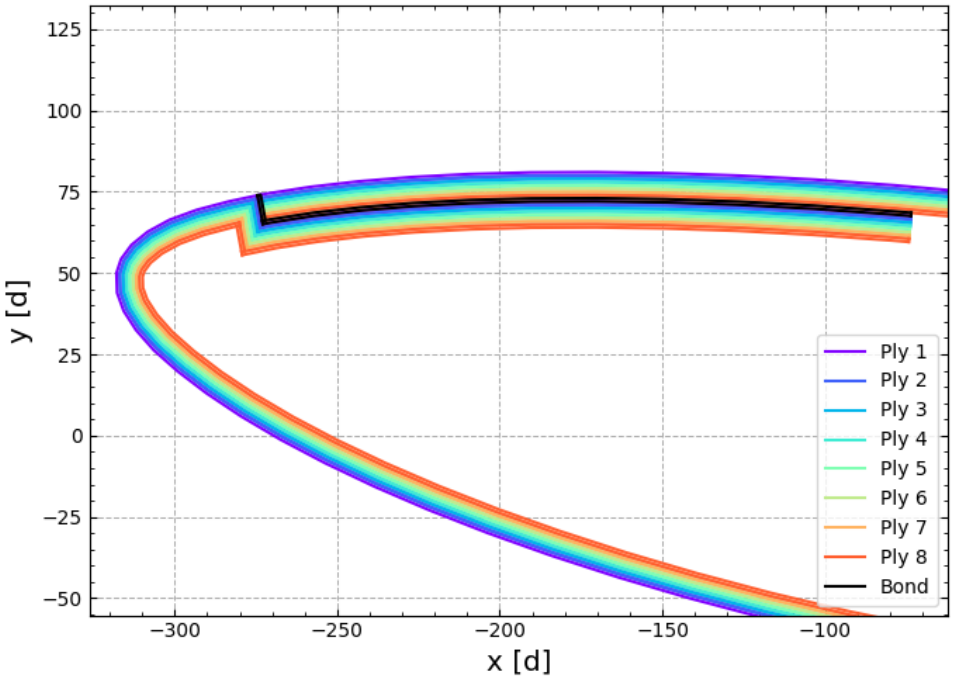


Figure 25: Example A: Closer look at the skin overlap in the last section instance.

In the last module (Export tab), we select **"Multiple files (CSV)"** to export our 5 section instances by selecting all of them in the menu (see Figure 26). We change the folder name to "Skin" and press **"Export Sections"** to generate the folders in the work directory.

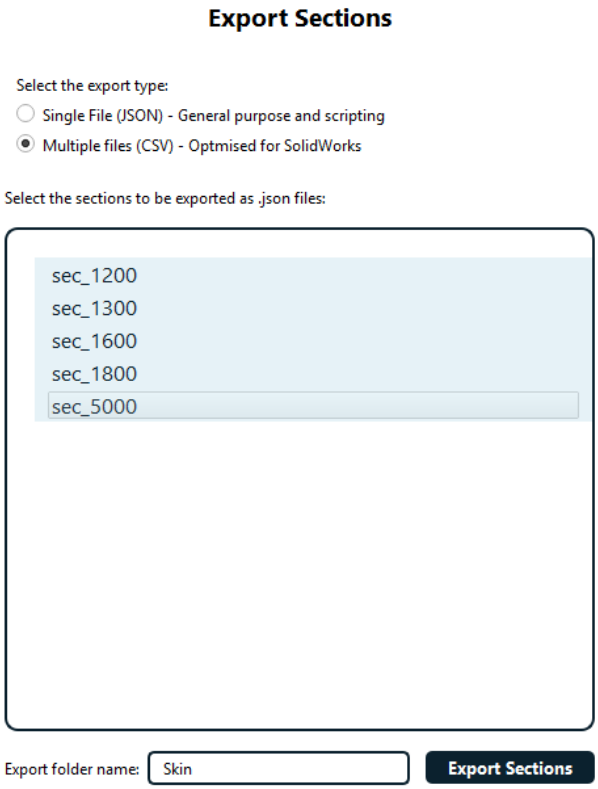


Figure 26: Example A: Export setting for the current session.

Lastly, we open SolidWorks and start a new part. We navigate to **Tools -> Macro -> Run...** and we find the plugin the comes with the **EdFoil** installation. This plugin is located by default in "C:\Program Files (x86)\EdFoil_internal\resources\plugins\imp_solidworks.swp". This will open a folder selection dialogue and we select one of the ply folders from our exported sections. Figure 27 shows the curves successfully imported to SolidWorks, which can now be used to generate lofted surfaces or solids. The final CAD for the first ply is shown in Figure 28. The procedure can be repeated for the remaining 7 plies.

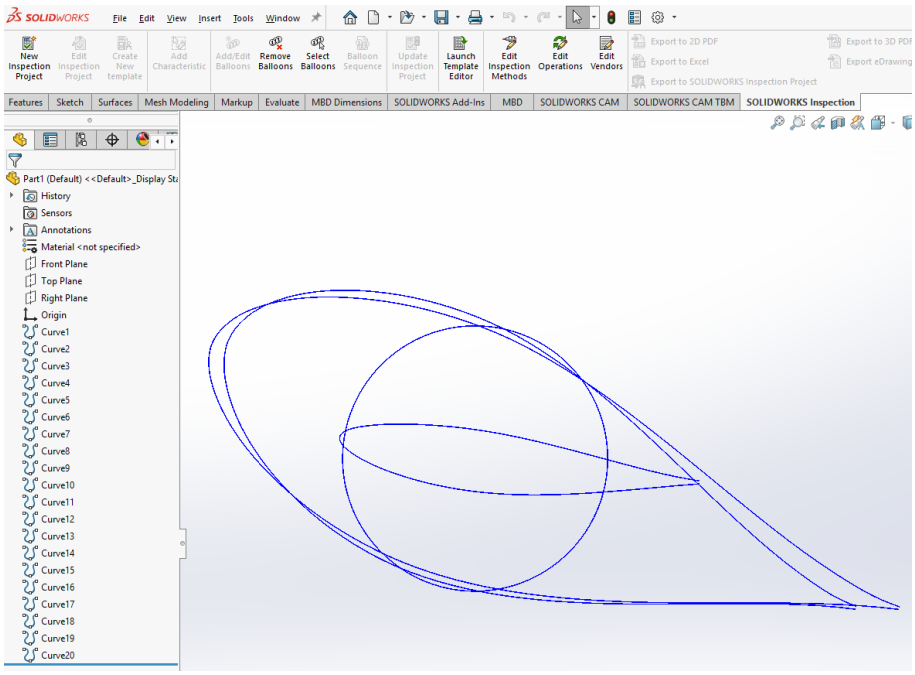


Figure 27: Example A: Ply curves imported to SolidWorks.

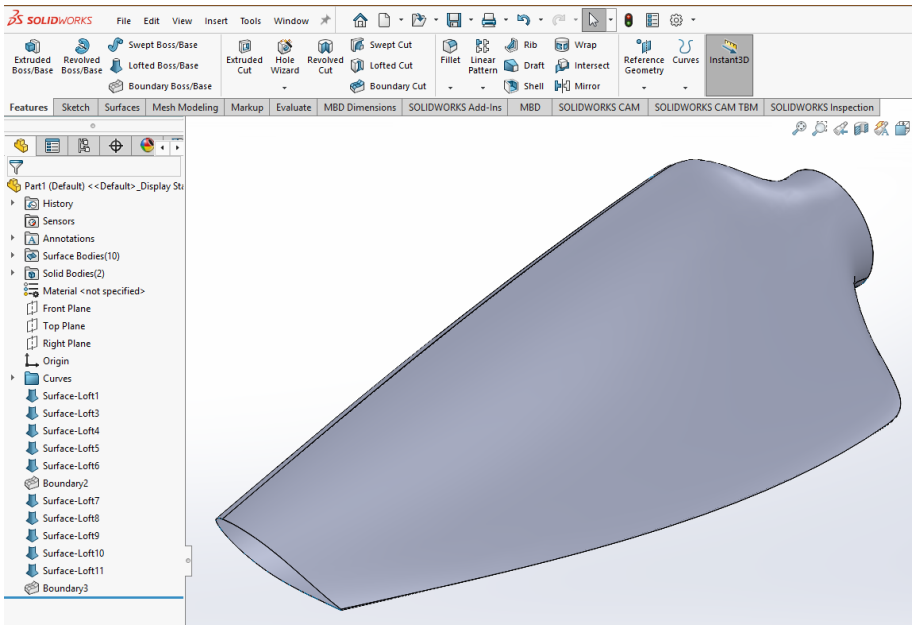


Figure 28: Example A: Separate solid bodies for the top ply and bottom ply.

4.2 Example B: Generating curves using the EdFoil framework

This example includes the same data as Example A, however, the steps will be done through the command line or by running a python code. This example shows more capabilities of the **EdFoil** framework that have not yet been implemented in the GUI. You can also follow this example directly in the Jupyter notebook provided in **"C:\Program Files (x86)\EdFoil_internal\resources\examples\exampleB\exambleB.ipynb"**.

The first step is to import the libraries. In this case we need the main three classes to generate a blade. It is important to note that paths are relative, thus if the current directory is not the root, the compiler might not be able to find the libraries.

```
import os
import pandas as pd
from edfoil.classes.airfoil import Airfoil
from edfoil.classes.section import Section
from edfoil.classes.station import Station
```

Two methods are defined to create the Station and the Section instances inside Pandas dataframes. Alternatively, a loop can be coded to iterate through each row and append the new instance to a list.

```
def get_sta(row):
    return Station(
        airfoil = airfoils[row['airfoil']],
        chord = row['chord'],
        twist_angle = row['twist_angle'],
        x_offset = row['offset_x'],
        y_offset = row['offset_y'],
        z_offset = row['offset_z'],
        x_multiplier = row['multiplier_x'],
        y_multiplier = row['multiplier_y'],
        z_multiplier = row['multiplier_z'],
        x_mirror = row['mirror_x'],
        y_mirror = row['mirror_y'],
    )
```

```
def get_sec(row):
    return Section(
        station = stations[row.name],
        n_plies = row['n_plies'],
        ply_thickness = row['ply_thickness'],
        overlap_target = row['overlap_target'],
        te_thickness = row['te_thickness'],
        bond_thickness = row['bond_thickness'],
        genFig = row['genFig'],
        saveFig = row['saveFig'],
        tolerance = row['tolerance'],
    )
```

All the data regarding the blade is imported into the current session, each on a separate dataframe to keep the column labels. The following information is stored in each file:

- **data.txt:** All the parameters needed to create a station instance (Airfoil, chord length, twist angle, offset, scaling, mirror).
- **data_sec.txt:** All the parameters needed to create a section instance (Number of plies, ply thickness, overlap location, trailing edge thickness, bond thickness, generate figures, save figures, tolerance).
- **data_jig.txt:** All the parameters needed to use the .jiggle method on a section instance (Overlap length, bond thickness)

```
data_sta = pd.read_csv(f'resources\\examples\\exampleB\\data.txt')
data_sec = pd.read_csv(f'resources\\examples\\exampleB\\data_sec.txt')
data_jig = pd.read_csv(f'resources\\examples\\exampleB\\data_jig.txt')
```

All the airfoils required are imported and an instance is created for each of the unique airfoils. In this example, the first two stations use the same airfoil (circle), hence only one instance is created for both.

```
airfoils = {}
for name in data_sta['airfoil'].unique():
    airfoil = Airfoil(name)
    airfoil.importCoords(f'edfoil/airfoils/{name}.txt')
    airfoils[name] = airfoil
```

After creating the airfoil instances, the method "get_sta" is called on each row. The stations are then turned into a list, sorted along the span of the blade.

```
stations = data_sta.apply(get_sta, axis=1).tolist()
```

As the data needed for section creation is stored in a Pandas dataframe, some of the parameters can be made constant for all stations. In this example, we set the bond thickness constant to 3 mm, the ply thickness set to 1 mm, and the trailing edge thickness set to 8 mm.

The data inside "data_sec" for the overlap location and length are normalised to the chord length, hence, we make sure these values are multiplied by the chord length at each station.

Finally, we generate the section instances with the method we defined earlier called "get_sec". This gives us a list with all the section instances in the same order as the stations.

```
# Changing the bond thickness in all stations to 3mm
data_sec['bond_thickness'] = 3

# Changing the ply thickness in all stations to 1mm
data_sec['ply_thickness'] = 1

# Changing the trailing edge thickness to 8mm
data_sec['te_thickness'] = 8

# Calculating the non-normalised values for the location of the overlap
# based on the local chord length.
data_sec['overlap_target'] = data_sec['overlap_target'] *
↪  data_sta['chord']

# Calculating the non-normalised values for the length of the overlap
# based on the local chord length.
data_jig['overlap_dist'] = data_jig['overlap_dist'] * data_sta['chord']

# Generating the sections
sections = data_sec.apply(get_sec,axis=1).tolist()
```

We perform the .jiggle method on each section to create the point cloud of the bond layer and the sunken section of the bottom skin. We assign a constant bond thickness of 3 mm.

```
# Iterating through each section created
for x in range(len(sections)):
    if data_jig['bond_thickness'][x] != 0:
        sections[x].jiggle(
            overlap_dist = data_jig['overlap_dist'][x],
            bond_thickness = 3,
        )
```

We also want to generate the trailig edge spar along the blade. We follow the same procedure as the previous method. We assign constant values for the .teSpar method:

- **te_distance:** Distance from the leading edge to the rear spar web along the chord-line. For this we set a distance of 220 mm.
- **thickness:** Web thickness, set as 1 mm for this example.
- **flange_distance:** Length of the flanges along the chordline, set to 50 mm.
- **n_tePlies:** Number of plies for the rear spar. Set to 2 plies.

```
# Iterating through each section created
for x in range(len(sections)):
    if data_jig['bond_thickness'][x] != 0:
        sections[x].teSpar(
            te_distance = 220,      # Trailing edge distance
            thickness = 1,          # Ply thickness
            flange_distance = 50,   # Flange distance away from TE
            n_tePlies = 2,          # Number of plies
        )
```

We can monitor each section by calling the attribute ".figs" and any of the figures created along the session. In this case (Figure 29) we see the figure created after generating the trailing edge spar. This rear spar seems appropriately placed 220 m from the trailing edge.

Note: The trailing edge is the vertex from the initial airfoil instance, not the furthermost point in the x-axis in the section.

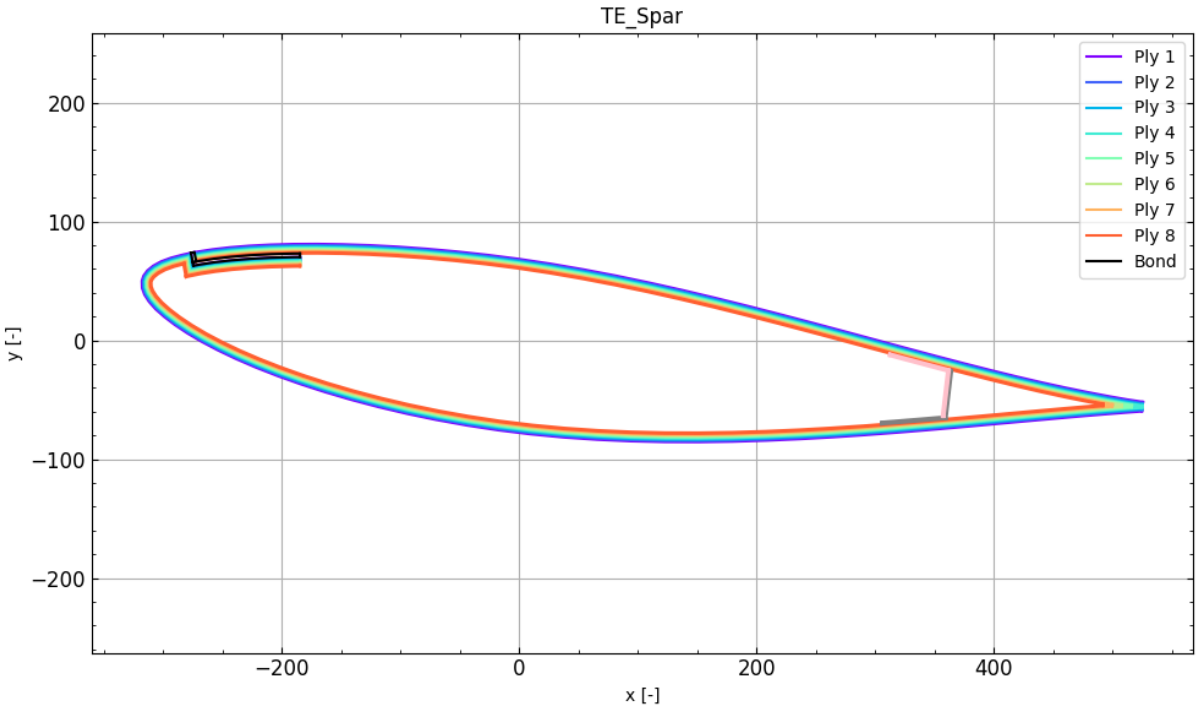


Figure 29: Resulting airfoil with 8 plies and a trailing edge.

Finally, we can export the point clouds for each component into different *.csv files that can be used in any CAD software for further assembly. We first create different folders for each component. In this example we are only showing the skin and the rear spar being exported, however, the Jupyter notebook available in the example folder contains code to export all components.

```
# Define the export location
output_path = f'resources\\examples\\exampleB\\output'

# Create folders
## Skin
path_skin = f'{output_path}\\Skin'
os.makedirs(path_skin, exist_ok=True)

## Bond
path_bond = f'{output_path}\\Bond'
os.makedirs(path_bond, exist_ok=True)

## Overlap
```

```
path_olp  = f'{output_path}\\Overlap'
os.makedirs(path_olp, exist_ok=True)

## Trailing edge spar
path_tesp = f'{output_path}\\TESpar'
os.makedirs(path_tesp, exist_ok=True)
```

The skin and the rear spar points are exported from the section instances. Each instance has the attribute ".points", where the coordinates (x, y) are located. The convention for this attribute is the following:

section.['component'][ply number][curve]['x']

These *.csv files can then be imported exactly like Example A to generate a solid body.

```
z_all = data_sta['offset_z'].tolist()

# Skin
for side in ['top','bot']:
    for i in range(len(z_all)):
        for ply in range(1,sections[-1].parameters['n_plies']+1):
            for spline in [0,1]:
                x = sections[i].points[f'{side}_1'][ply][spline]['x']
                y = sections[i].points[f'{side}_1'][ply][spline]['y']
                z = [z_all[i]] * len(x)

                df = pd.DataFrame({'x':x,'y':y,'z':z})

                df.to_csv(
                    f'{path_skin}\\{side}_{z_all[i]}_{ply}_{spline}.csv',
                     index = False,
                     header = False,
                )

# Rear Spar
for i in range(2,len(z_all)):
    for spline in [0,1]:
```

```
        x = sections[i].points['te_spar'][1][1][1][spline]['x']
        y = sections[i].points['te_spar'][1][1][1][spline]['y']
        z = [z_all[i]] * len(x)

        df = pd.DataFrame({'x':x,'y':y,'z':z})

        df.to_csv(
            f'{path_tesp}\\tesp_{z_all[i]}_{spline}.csv',
            index = False,
            header = False,
        )
```

5 EdFoil Framework Class Documentation

The **EdFoil** framework defines three central classes that describe the geometry and construction of composite blade sections: Airfoil, Station, and Section. The design is hierarchical:

- An **Airfoil** holds the base two-dimensional profile.
- A **Station** is a transformed copy of the airfoil located at a spanwise position with chord, twist, scaling, and offsets.
- A **Section** constructs the laminate stack, offsets, overlaps, and guides from a given station.

5.1 Class Airfoil

Purpose: Represents a two-dimensional airfoil profile.

Attributes:

- `path` (str): Path if imported.
- `name` (str): profile name (e.g. `NACA63416`).
- `xy` (list): full ordered coordinates (lower then upper).
- `upper`, `lower` (list): separated surfaces.
- `n_points` (int): number of coordinate points.

- `family`, `profile` (str): e.g. `NACA`, `63-206`.
- `imported` (bool): Import status.
- `report` (str): Contains log reports from airfoil creation.

Main Methods:

__init__(name):
Description: Instance initialisation.

- `name` (str): Airfoil instance name.

importCoords(path):
Description: Read `.dat/.txt` coordinate files.

- `path` (str): Location of the file containing the coordinates.

update(coords):
Description: Update with new point list.

- `coords` (list): Nx2 list of x and y coordinates.

changeName(name):
Description: Changes name in airfoil instance.

- `name` (str): Name of the new airfoil.

exportAirfoil(path):
Description: Write coordinates to file.

- `path` (str): Export path including the name of the file.

plotAirfoil(display):
Description: Quick Matplotlib plot.

- `display` (bool): If true displays the plot, if false it can be stored in a variable.

naca6(profile, path, n_points):
Description: Load NACA 6-series JSON data (Deprecated).

- `profile` (str): Name of NACA profile.
- `path` (str): Location of the tabulated data in a json format.
- `n_points` (int): Number of points to interpolate.

naca456(namelist_text, keep_files):
Description: Run bundled Fortran backend.

- `namelist_text` (str): Input parameters for NACA profile (more details in "nameinput" method).
- `keep_files` (bool): Keep creation log.

nameinput(profile_code, name, chord, dencode):
Description: Build a Fortran input namelist.

- `profile_code` (str): Airfoil NACA code name.
- `name` (str): Title desired on output. It must be enclosed in quotes.
- `chord` (float): Model chord used for listing ordinates in dimensional units.
- `dencode` (int): Spacing of the x-array where the points are computed

5.2 Class Station

Purpose: Represents a spanwise blade station by transforming an airfoil.

Attributes:

- `parameters`: dictionary with chord, twist angle, offsets, multipliers, mirrors, circle flag.
- `airfoil` (str): reference airfoil name.
- `xy` (list): transformed station coordinates.

Transformations:

1. Mirror (optional, x/y axes).
2. Scaling by chord and multipliers.
3. Rotation by twist angle.
4. Offset in (x, y, z) space.

Main Methods:

__init__(airfoil, chord, twist_angle, x_offset, y_offset, z_offset, x_multiplier, y_multiplier, z_multiplier, x_mirror, y_mirror):
Description: Instance initialisation.

- `airfoil` (Airfoil): Airfoil instance.
- `chord` (float): Desired chord length.
- `twist_angle` (float): Twist angle in degrees.
- `x_offset` (float): Offset in the x-axis for all coordinates.
- `y_offset` (float): Offset in the y-axis for all coordinates.
- `z_offset` (float): Spanwise distance from the root.
- `x_multiplier` (float): Scaling factor in the x-direction.
- `y_multiplier` (float): Scaling factor in the y-direction.
- `z_multiplier` (float): Scaling factor in the z-direction.
- `x_mirror` (bool): Mirror x-coordinates.
- `y_mirror` (bool): Mirror y-coordinates.

plot(name):
Description: Plot transformed station geometry.

- `name` (str): Graph title.

xyRange():
Description: Return bounding box of coordinates.

5.3 Class Section

Purpose: Builds the laminate skin section from a station, including ply offsets, overlap, trailing edge cut, and bonding region.

Attributes:

- `parameters`: ply thickness, number of plies, overlap, twist, thicknesses.
- `splines`: cubic spline fits for each offset curve.
- `indexes`: leading edge, overlap, and trailing edge indices.
- `t`: parameter values along splines (bot/top/plies).
- `points`: dictionaries of coordinate groups for plies and bonds.
- `guides`: chord, overlap, trailing edge, spar lines.

- `figs`: stored Matplotlib figures.

Main Methods:

__init__(station, n_plies, ply_thickness, overlap_target, te_thickness, bond_thickness, genFig, saveFig, tolerance, ini_u0):
Description: Instance initialisation.

- `station` (Station): Station instance.
- `n_plies` (int): Number of skin plies.
- `ply_thickness` (float): Thickness of each ply.
- `overlap_target` (float): Overlap location from the LE.
- `te_thickness` (float): Trailing edge thickness normal to chordline.
- `bond_thickness` (float): Bond thickness at the skin joint.
- `genFig` (bool): Generate figures inside instance.
- `saveFig` (bool): Save figures in current work directory.
- `tolerance` (int): Tolerance for decimal places ACIS in LE.
- `ini_u0` (int): Initial guess for splineIntersection method starting from the trailing edge.

info():
Description: Print summary of keys and properties.

plotSection():
Description: Plot all ply boundaries.

jiggle(overlap_dist, bond_thickness):
Description: Compute overlap geometry with “jiggle”.

- `overlap_dist` (float): Overlap length from overlap location towards TE.
- `bond_thickness` (float): Bond thickness at the skin overlap.

teSpar(te_distance, thickness, flange_distance, n_tePlies):
Description: Add trailing-edge spar region.

- `te_distance` (float): Distance between the rear spar and the TE vertex.
- `thickness` (float): Channel thickness.
- `flange_distance` (float): Length of the flanges in the LE direction.
- `n_tePlies` (int): Number of plies in the rear spar.