Create an Automated Structured Mesh Generation Method for Rotor Blades using Exclusively Open Source Software

Bachelor Thesis
by
cand. aer. Michael Andreas Heider

Conducted at the
Institute for Aerodynamics and Gas Dynamics
at the University of Stuttgart

Stuttgart, September 2023

Universität Stuttgart

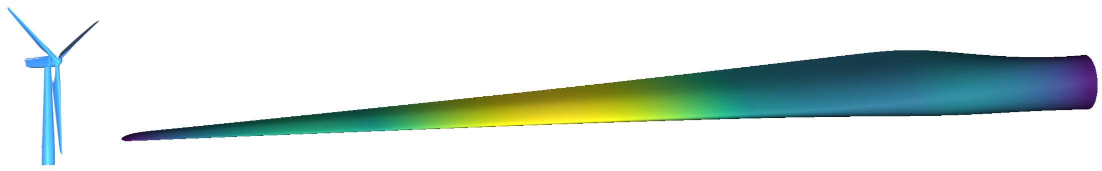
INSTITUT FÜR AERODYNAMIK UND GASDYNAMIK IAG

DIREKTOR: PROF. DR.-ING. EWALD KRÄMER

Pfaffenwaldring 21, 70550 Stuttgart, Tel (0711) 685-3401, Fax 3402, email:kraemer@iag.uni-stuttgart.de

Bachelor Thesis for Michael Heider

Create an Automated Structured Mesh Generation Method for Rotor Blades using Exclusively Open Source Software



To conduct high fidelity fluid dynamics siumulations, it is often convenient to rely on structured computational grids. To that end, the Wind Energy Research Group of the Institute of Aerodynamics and Gas Dynamics relies on commercial software that has been partly automated to generate high quality turbine blade meshes.

The topic of the thesis is thus to develop an alternative approach to structured mesh generation that relies solely on open source software. It will allow to complete the surface mesh part of the tool chain without the need for licensed software and also be able to extrude this mesh into a component of the Chimera-based distributed numerical fluid domain.

Milestones:

- create a method to build a structured surface mesh on the outer mantel of a rotor blade or wing surface
- expand the method to allow extending the surface mesh to include the tip of the blade as well
- verify the capability of the method to work with different blade profiles
- ensure the high quality of the generated meshes by automatically inspecting the skewness and growth ratio and/or other quality parameters of the surface mesh and by need implementing mitigation strategy using automatic blade inspection or, if necessary, simple user input requests
- implement an extrusion method to generate a high quality 3D mesh starting from the generated full surface mesh
- tune the extrusion method to allow setting mesh parameters such as the boundary condition at its root and the growth ratio, given a function or a list of values
- implement a verification of the resulting quality of the 3D mesh along with automatic mitigation strategies

Date Issued: May 15th, 2023

Date Submitted: September 15th, 2023

Student: Michael Heider

Advisor: Louis Gagnon

Examiner: Thorsten Lutz

Statement of Originality

Statement of Originality

This thesis has been performed independently with support of my supervisor. It contains no material that has been accepted for the award of a degree at this or any other university. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person except where due reference is made in the text. I further declare that I have performed this thesis according to the existing copyright policy and the rules of good scientific practice. In case this work contains contribution of someone else (eg. pictures, drawings, text passages etc.), I have clearly identified the source of these contributions, and, if neccesary, have obtained approval from the originator for making use of them in this thesis. I am aware that I have to bear the consequences in case I have contravened theses duties.

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig mit Unterstützung des Betreuers angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit oder wesentliche Bestandteile davon sind weder an dieser noch an einer anderen Bildungseinrichtung bereits zur Erlangung eines Abschlusses eingereicht worden. Ich erkläre weiterhin, bei der Erstellung der Arbeit die einschlägigen Bestimmungen zum Urheberschutz fremder Beiträge entsprechend den Regeln guter wissenschaftlicher Praxis eingehalten zu haben. Soweit meine Arbeit fremde Beiträge (z.B. Bilder, Zeichnungen, Textpassagen etc.) enthält, habe ich diese Beiträge als solche gekennzeichnet (Zitat, Quellenangabe) und eventuell erforderlich gewordene Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt. Mir ist bekannt, dass ich im Falle einer schuldhaften Verletzung dieser Pflichten die daraus entstehenden Konsequenzen zu tragen habe.

.... Ort, Datum, Unterschrift

Abstract

The generation of high-quality Computational Fluid Dynamics (CFD) meshed wings for wind energy turbines currently requires either a very laborious process of building each wing by hand or requires the purchase of very expensive software. With university budgets and the advantages of open-source in mind, an open-source solution is desirable. The following thesis tackles this problem via a grid generation code in Python with Gmsh and extrusion via Pyhyp. However, designing software that has all of the features of commercial software with open source software leads to situations in which the intended way of usage is not the best way for the given task. An example of another open-source approach is the MACH-AERO framework which is an aerodynamic shape optimization tool that creates a grid along the way and extrudes it. However, this does not cover the intended usage of openblademesh for purely structured grid generation for predefined wings. Here we show one way to achieve an open-source solution with Gmsh and Pyhyp. All the necessary software, the versions needed, and installation instructions are described. Also how the wing generation process is achieved with special concentration on the wing-tip is part of the following thesis. The thesis also describes how to use the presented software Openblademesh and how to achieve the best results. The tool uses various automatisms to take the workload off of the user and helps to achieve the best mesh. Finally, the advantages and limitations of the tool are assessed and future extensions are described.

Die Erstellung von qualitativ hochwertigen Computational Fluid Dynamics (CFD)-vernetzten Flügeln für Windenergieanlagen ist derzeit entweder ein sehr aufwändiger Prozess, bei dem jeder Flügel von Hand gebaut wird, oder durch sehr teure software ermöglicht wird. Mit Blick auf die Universitätsbudgets ist eine Open-Source-Lösung wünschenswert. In der folgenden Arbeit wird dieses Problem durch einen Gittergenerierungscode in Python mit Gmsh und Extrusion über Pyhyp angegangen. Die Entwicklung einer Software, die alle Eigenschaften kommerzieller Software besitzt, mit Open-Source-Software führt jedoch zu Situationen, in denen die vorhergesehene Code Struktur der Open-source-Software nicht die beste für die gegebene Aufgabe ist. Ein Beispiel für einen anderen Open-Source Ansatz ist das MACH-AERO-Framework, ein Werkzeug zur aerodynamischen Formoptimierung von Flügeln generell, das unter anderem auch ein Gitter erstellt und dieses extrudiert. Dies deckt jedoch nicht die beabsichtigte Nutzung von Openblademesh zur rein strukturierten Gittergenerierung für vordefinierte Tragflächen ab. Hier wird ein Weg gezeigt, wie man eine Open-Source-Lösung mit Gmsh und Pyhyp erreichen kann. Es werden die benötigte Software, die benötigten Versionen und die Installationsanweisungen beschrieben. Auch wie der Prozess der Flügelgenerierung mit besonderem Augenmerk auf die Flügelspitze erreicht wird, ist Teil der folgenden Arbeit. In der Thesis wird auch beschrieben, wie man die vorgestellte Software Openblademesh benutzt und wie man die besten Ergebnisse erzielt. Das Tool verwendet einige Automatismen, um den Benutzer zu entlasten und hilft, das beste Netz zu erzielen. Obwohl es gelungen ist, ein quelloffenes Werkzeug zur Gittergenerierung zu entwickeln, gibt es noch Probleme, die in Zukunft behoben werden müssen. Abschließend werden die Vorteile und Grenzen des Tools im Vergleich zu kommerzieller Software bewertet.

Contents

Acronyms

CAD	Computer Aided Design
CFD	Computational Fluid Dynamics
DLR	Deutsches Zentrum für Luft- und Raumfahrt
GUI	Graphical User Interface
IAG	Institute of Aerodynamics and Gas dynamics
NACA	National Advisory Committee for Aeronautics
TE	Trailing Edge

Symbols

a first step in geometric series

$a1$ first step on examined line

$a2$ first step on second section

k control variable

$l1$ length of first section

$l3$ length of last section

$n1$ entire nodes over wing

$n2$ number of nodes on entire wing, excapt last section

$n3$ amount of nodes at the end of section one

r coefficient of geometric series

$r1$ coefficient of geometric series for node distributuion

y marching direction normal to cell

Introduction

Climate Change Climate change affects us all and pushes the global community towards renewable energy sources. This is because most of the consumed energy over the last century was provided by fossil fuels. Therefore, the world needs to consider different solutions to the question of how the increasing energy demands can be met to preserve today´s standards while still protecting the earth from the man-made threat of climate change. To tackle one of the biggest problems of the twenty-first century, a lot of research and development needs to be dedicated to this topic, and to harvest the free available resources of the earth such as wind, water or tidal forces is a very convincing step towards a climate-neutral future. One of many potential solutions to emission-free energy production that is currently being worked on in parallel focuses on wind energy. However, since the use of this form of energy production on such a big scale is quite new it requires constant development.

Wind Energy Historically, wind energy was used in windmills, sailing boats, or windpumps. Today however, the majority of wind energy is used for converting kinetic energy from the wind to electric energy via a generator installed inside the wind energy plant.

Wind turbines have gained center stage in scientific research since wind is accessible almost everywhere. The high availability and near independence of day and night cycles make them an ideal field of research. They tend to supply a better energy conversion at night time in many regions and are therefore, in combination with solar power, an ideal alternative [1]. To further increase its efficiency without having to pay a lot of money for a lot of wind tunnel tests, simulations are the only solution. With these simulations, blade outlines, the choice of airfoil used at different parts of the blade and and many other parameters can be improved.
Another important factor to take into consideration is the constellation and place for the turbine positionings. Usually, more than one turbine is placed and therefore they influence each other due to the way they manipulate the wind. This can be used in advantageous situations to achieve the highest possible yield or at least to create the least amount of losses possible.
Also, another factor to take into consideration is the position of the turbines. For example, an already widespread constellation is offshore parks which harness the great amount of oceanic winds while still being almost invisible for the low settlement of the oceans. But how woods, hills, and nearby mountains influence the performance of the turbine are parameters that have to be taken into consideration when the decision for new wind turbines is made.

Simulation While the need for more efficient turbines increases, thankfully the computational power of computers increases as well. Therefore, better simulations can be achieved and more complex problems solved. Thereby, increasingly complex simulation tasks can be solved and therefore converge closer to reality with better results for the blade with each new computer generation. The problem with this is that the older data often cannot be reused because newer insiights showed that some of the results are not precise enough and need to be redone.

However, there still is the problem with very cost-intensive software which is necessary for efficient simulation work. With budgeting in mind, an open-source solution with the same functionality would be the ideal solution. To get a Computational Fluid Dynamics (CFD) simulation that provides processible results, a good grid generation is key. To achieve this, the grid generation and extrusion are the content of this thesis.

Open-Source One of the benefits of open-source software is the possibility to be able to achieve a good quality grid with software that can be developed for a possibly very specific application. Also, the possibility to freely access it will give a lot of interested peers the possibility to work with Computational Fluid Dynamics (CFD) software without needing to buy commercial software which often can be very expensive. Because the Computational Fluid Dynamics (CFD) software the Institute of Aerodynamics and Gas dynamics (IAG) uses needs to get a structured grid the Openblademesh software gives the resulting mesh in structured form.

Structured Grid The advantage of this approach is that a structured grid uses less memory compared to an unstructured grid and the distributed calculation points can be set by the user, while the downside is that a structured grid generation is more complex because the coding and generation needs to be precise and automated tools are not able to generate it by itself. Therefore, different methods are used to achieve structured and partly automated grids with open source software which will be described in the following thesis.

Goals of the Thesis The goal of this thesis is mainly to show that it is possible to achieve a mesh generator that creates a structured grid completely on open-source software. Furthermore, several waypoints are:

1. Enable the code to generate a wingtip from one pointcloud
2. Generate the wingtip with subsections for smooth transition
3. Create a wing with several airfoils
4. Automate wing generation for flexible pointclouds
5. Enable Pyhyp to read output from Gmsh
6. Improve node distribution in wing direction
7. Improve usability for user

Solutions from other Engineers Other approaches to this problem have been made but have their limitatians or only achieve some of the wanted targets. For example, the free version of Chimera Grid Tools which relies on the chimera overset grid approach, is a very promising tool, but as it works with the National Advisory Committee for Aeronautics (NACA) chimera grid it is not open source and most importantly only free for US-American citizens or people who qualify as such and is therefore out of reach for any official institute outside of the USA. Another mentionable software is MegaCads by the Deutsches Zentrum für Luft- und Raumfahrt (DLR) which creates hexahedral structured grids. But this software is still not open source and therefore not very customizable. Also, the documentation and support is not well continued.. The software used by the Institute of Aerodynamics and Gas dynamics (IAG) is the automesh software which is open source. It was first designed for wind energy blades and then extended to helicopter blades. However, it only creates unstructured meshes and therefore still does not apply to the given tasks.

1 Fundamentals

1.1 Mathematical Foundations

1.1.1 Spline

Splines are mathematical functions that are smooth lines that have boundary conditions that dictate the form of the resulting graph. Splines get used a lot in Computer Aided Design (CAD) for their smooth adaption between specified points. Two kinds of splines will be explained in the following from which one is used in this thesis. The B-spline or basis spline is a spline that is often used in numerical applications due to its numerical stability. However in this thesis, only cubic splines are used, therefore they will be explained in more detail. Cubic splines are specified by connection curves which share one or a severall points and by ensuring that their first and second derivation at this point are the same. This guarantees that the graph will be coherent and smooth because the same derivations translate to the same curvature at that point [2]. The Figure 1.1 demonstrates how parts one and two are connected at the blue highlighted point to generate a smooth curve via cubic spline.

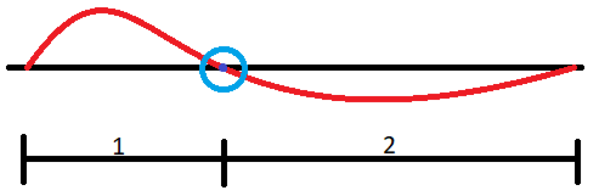


Figure 1.1: example spline with connection point highlighted in blue

In this thesis, cubic splines are used for their easy implementation in the code and adequate preciseness.

1.1.2 Progression

Because in Computational Fluid Dynamics (CFD) computation power and consequently time is the regulating factor, as explained later in the section of the fundamentals for Computational Fluid Dynamics (CFD), it is always important to save as many computations as possible. For Computational Fluid Dynamics (CFD)simulations this means a trade between resolution and time, and is achieved by regulating how many points (or in Computational Fluid Dynamics (CFD) often called nodes) are computed. To achieve this on the aerodynamically interesting sections, the number of nodes is increased.

Consequently, on the lesser interesting parts, the number of points is reduced. This is achieved through different progressions on the lines. In this thesis, two different kinds of progressions are used and therefore explained. First, the easiest one is a progression that distributes the points at a growing rate of one. Therefore, the distance between every point is constand. In Figure 1.2 pictured as the red points on the black line.

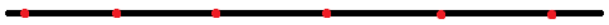


Figure 1.2: line with progression with a growing factor of one

The next progression is a geometrical series, which is simpy called progression by Gmsh.

$$\sum_{n=1}^{k-1} ar^n \tag{1.1}$$

This series converges for $|a \cdot r| < 1$ to

$$l = a \cdot \frac{1 - r^n}{1 - r} \tag{1.2}$$

If the series converges, each new summation step in the calculation is smaller than before and the series as a whole approaches the value which is calculated in the closed form of Equation 1.2. So in terms of point distribution, the series variable a can be understood as the distance from the first point to the second. The nvariable r is the coefficient that decides how fast or slow the series grows and the variable k can be understood as the number of points which are distributed on an imaginary line. In Gmsh it is only possible to change the value of r and k by adapting the progression coefficient and the number of points used. So, to know what first step size will be created to achieve a smooth transition from one line to the other, the series needs to be converted to

$$a = l \cdot \frac{1 - r}{1 - r^n} \tag{1.3}$$

which is converted from (1.2) and will result in progression that looks like Figure 1.3

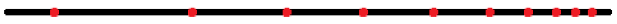


Figure 1.3: example line with point progression and a coefficient smaller than one

If $|a \cdot r|$ is > 1 the series diverges to infinity and every step is bigger than the one before. This can be used if it is useful if the distribution goes from a small precise area to an area that needs a coarser mesh again displayed in Figure 1.4

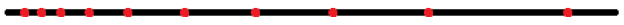


Figure 1.4: example line with point progression and a coefficient greater than one

1.1.3 Interpolation

In this thesis, some points are calculated through linear interpolation. This is a mathematical equation that leads to a point before, after, or in between two other points, which is on the connecting line of them with the following equation

$$y = y_0 + (x - x_0) \cdot \frac{y_1 - y_0}{x_1 - x_0} \tag{1.4}$$

and can be used to achieve a point on a linear curve for which only two coordinates are known.

1.1.4 Transfinite Algorithm

The transfinite algorithm is used in this thesis to dictate the position of points on a given curve and later on to dictate the position of nodes on planes. It is not necessary to fully understand the mathematical underlayings of this technique regarding this thesis as it is only used as a tool for point distribution in the code. If one wishes, the math can be inspected in [3].

The name derives from the fact that, in contrast to regular algorithms, the transfinite algorithm matches a nondenumerable number of points. The algorithm is described, in the words of William J. Gordon and Linda C. Thiel [3]: "Our purpose here is to describe how the techniques of bivariate and trivariate "blending function" interpolation, which was originally developed for and applied to geometric problems of computer-aided design of sculptured surfaces and 3-D solids, can be adapted and applied to the geometric problems of grid generation"

1.1.5 Hyperbolic Extrusion

To extrude from the generated code on the wing surface in the third dimension, to allow for a computation of the fluid surrounding it, there are several methods from which two are relevant for this thesis, and therefore are introduced. First, the hyperbolic extrusion, as described in [4] is the method Pyhyp uses to extrude the wing surface. It has many advantages such as creating almost orthogonal grids which result in a better solution for the simulation and can be generated in significantly less computational time than other methods. However, some drawbacks sometimes can be decisive. The mesh generation methods are less robust and this often leads to no generation at all because the input grid has to be very good in the first place. Furthermore, it tends to disperse input discontinuities which increases the aforementioned problem. So if the input grid is too uneven or if the sizing factor between two adjacent cells is too big, the software cannot compute the correct extrusion without overlaying or restraining one extruded cell to the other, and therefore gives back an error [5].

1.1.6 Algebraic Extrusion

Another approach that is used is algebraic extrusion. However, because in this thesis this is only demonstrated as replacement when the hyperbolic extrusion of Pyhyp was not able to achieve the desired result, it is only descibed superficially. The algebraic extrusion method is distinctly more stable than the hyperbolic extrusion and therefore better suited for not entirely refined grids. It is, however, less orthogonal [6].

1.2 CFD

1.2.1 Simulation

For many tests or applications, it is much more reasonable to run a simulation than to test it in reality. First, it is less expensive and time consuming, second, it is often more precise and controllable because every input and output variable can be adjusted. Additionally it would sometimes not be possible to run an experiment comparable to the simulation. For example, it is possible to simulate the erosion of a wind energy blade over its life span in a relatively reasonable time frame, while it would take the life span of the blade or some adjusted parameters in real life. To work with reliable and accurate fluid simulations, which are crucial for essentially every aspect of modern engineering, a solid mathematical foundation is needed. Computational Fluid Dynamics (CFD) simulations are the solution of the Navier-stokes equations for a given set of nodes.

1.2.2 Navier Stokes

The Navier-Stokes equations describe a flow field in its entirety by giving the solution for the impulse in three cartesian dimensions aswell as the pressure, the temperature, and the density. By calculating these values over the geometry of the model that is to be simulated, the state of the fluid is calculated to know how the fluid in this section is behaving.

The model is specified by the positions of the equations that are placed on the surface of the model and from there marching in the normal direction of the model. As shown in Figure 1.5 the black wing is coated with a regular grid which is the surface mesh. From there in red and blue the grid extrudes in the normal direction of every cell. In this picture, the extruded grid is separated into two dimensions for a better visualization but in reality, is connected and surrounds the whole surface. The extruded mesh around the wing is the area in which the equations are solved and therefore is important to be set according to the needed tasks.

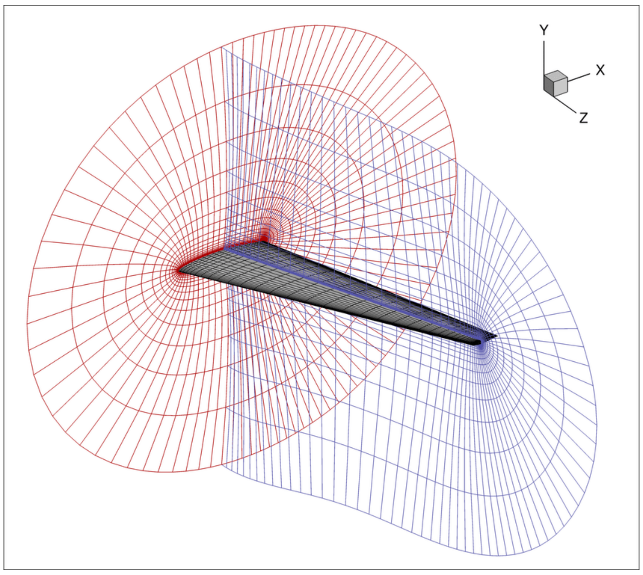


Figure 1.5: example wing with surface mesh and extrusion in Y and Z direction from [7]

1.2.3 Parameters

A very important parameter for Computational Fluid Dynamics (CFD) simulations is the y+ value. It describes the distance of the first step from the surface. Depending on the refinement one wants to simulate, a very small step can lead to very precise near-wall (as the surface is called) results. Because the fluid of the stream has a viscosity that interacts with the friction of the wall, the fluid gets more decelerated near the wall than further away. This effect can be very important or completely obsolete, depending on the desired results and the used calculation methods, if the friction is neglected for example. So it is important to know and calculate the correct y value for each simulation, but there are online applications that help with this [8].

In the current state of the art, the workflow to create a Computational Fluid Dynamics (CFD) simulation involves several steps which can be boiled down to three big steps. First, in the pre-processing, the model itself gets prepared to ensure that only the aerodynamically interesting parts are in the simulation. Also, the surface is represented by a grid on which the calculations are set. Furthermore, part of the pre-processing is the definition of the boundary values.
Next up is the solution which takes up the majority of the computation time of the simulation in which the equations get solved on the predetermined nodes. This can take from several hours up to days. Lastly, the results are validated to confirm that they represent the reality. This is achieved by comparison with table values or simplified algebraic calculations. If it is determined that the simulation was successful the solutions get visualized to achieve a faster and easier understanding of the results.

It is also very important for the mesh to be smooth. Smooth means that between two adjacent cells, there is no big difference in size as every cell should be as close to a square as possible. This is because the calculation for the extrusion takes one calculation for each node and if the distance, for example, has different sizes, the extrusions of the two adajcent cells will not fit into the next layer of cells in the y+ direction. Every cell needs to be in a way that the extrusion software can create another layer on top of it, and by keeping the cells as regular as possible, this is achieved. Also, big jumps in the cell size lead to very unclean calculations for the fluid at that part of the model, giving results in a very uneven form [6].

1.2.4 Unstructured Mesh

One way of meshing is that the program that is used sets the connecting points inside of defined boundaries randomly. This is a fast approach that can be calculated by most meshing tools automatically. This has the advantage that the mesh is created fast and is easy to automate. It can be achieved by reading a Computer Aided Design (CAD) file in a meshing software and a mesh can be generated. But with the drawback that the file is pretty large and the position of the mesh nodes are not precise and therefore more than would be necessary, if every node would be placed exactly where it is needed. Unstructured grids are usually triangles and randomly arranged in the mesh as shown in Figure 1.6.

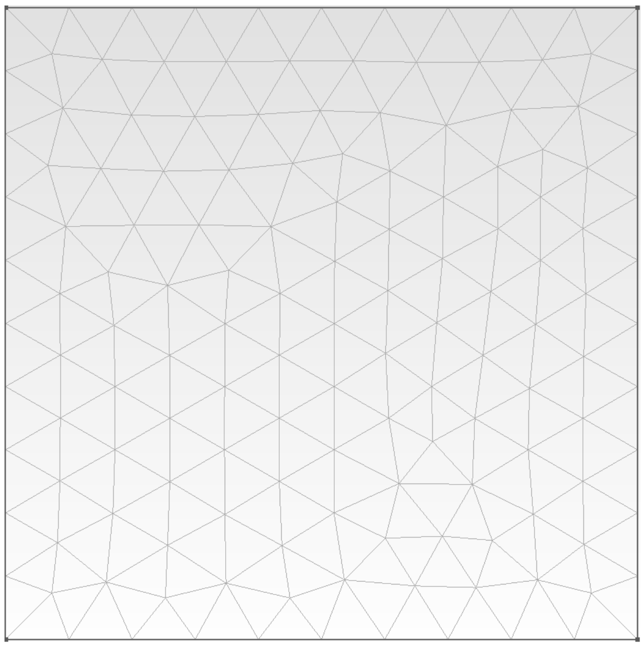


Figure 1.6: example picture of an unstructured mesh with trihedral meshing

1.2.5 Structured Mesh

Another way to assemble the mesh is that the points needed are set manually and the code simply connects them. By this approach, the mesh looks more regular which enables the user to control where refinements are and because of the regular look, the mesh reminds of a grid. Therefore a structured mesh is often called grid. More importantly, all the nodes are sorted in a cartesian way so that each point knows its position (by i and j), which are X and Y coordinates in a relative system (depending on each point) and the position of its neighboring points (i+-1 and j+-1). By deciding the position of each point manually they can be exactly where the user wishes them to be thus saving all the nodes that the computer would set. By this decrease of nodes and because each node knows the position of its neighboring cell relative to its own position a significant amount of memory capacity and computational power, or in Computational Fluid Dynamics (CFD) terms, computational time, can be saved. Structured grids are always quadrangular in shape and enable high efficiency as well as resolution. In Figure 1.7 an example of a structured grid can be seen with one node and its adjacent nodes with their relative coordinates.

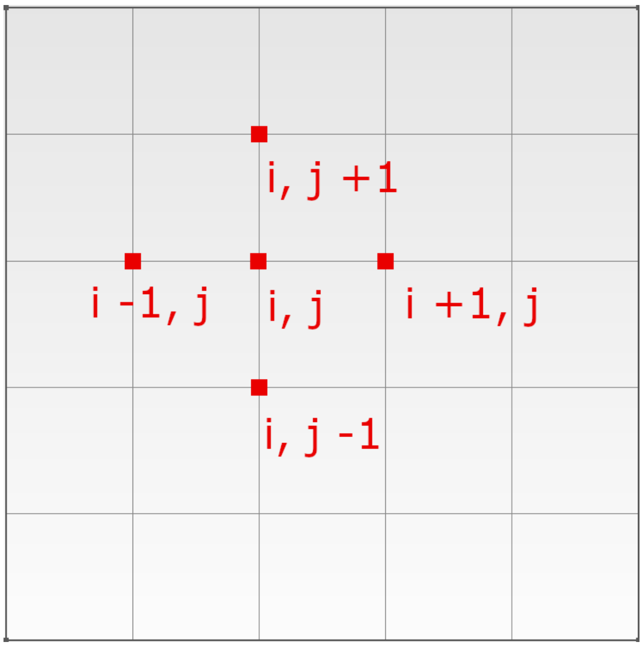


Figure 1.7: Structured mesh with quadrangular mesh

So it comes to the decision if the less computational extensive structured grid approach is the greater setup time of the meshing software worth. Or as in the case of the software used in the Institute of Aerodynamics and Gas dynamics (IAG), what kind of mesh can be read. The software of the Institute of Aerodynamics and Gas dynamics (IAG) in the wind energy group can only read structured meshes and therefore the meshing software needs to generate structured grids.

1.3 Aerodynamics

1.3.1 Airfoils

In aerodynamics, it is often necessary to achieve a force simply by letting an object be circulated by a fluid. Airfoils describe the shape, a wing or a similar aerodynamical component has that is in a fluid.
To get this, through the work of Otto Lilienthal [9], the airfoil which is based on the section of a bird's wing, has become the standard. By using different proportions, different effects can be achieved, from simply creating drag to resulting in lift, the airfoil shape is a very important design factor. Therefore a lot of research has gone into designing different airfoil forms and consequently, many standards have evolved for engineers to communicate without having to describe the complete form. As an example of one of the standards, the National Advisory Committee for Aeronautics (NACA) airfoil with four digits will be described in the following. The first digit always describes the maximal airfoil arching in percent and in relation to the airfoil chord. The second describes the arching position in tenths of the airfoil chord and the third and fourth digits stand for the maximal airfoil thickness in percent of the airfoil chord length. Because NACA four-digit airfoils always have the thickest point at 30% of the airfoil, these digits describe an airfoil exhaustive.

Figure 1.8 shows the National Advisory Committee for Aeronautics (NACA) four-digit standard described airfoil. The first digit describes the curvatures of the camber line, the second X_Cmax, and the third and fourth the maximal thickness of the airfoil.

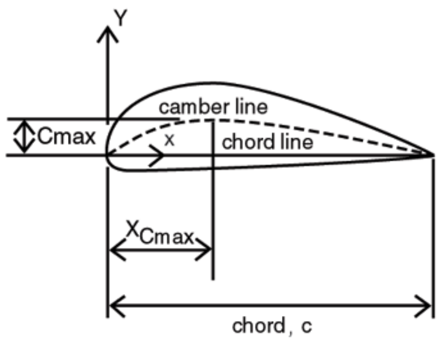


Figure 1.8: visualization of the parameters described by the NACA four-digit standard from [10]

With all of the digits known an engineer can search for the respective table and find all the data for this profile which have been identified beforehand in experimental trials in wind or water channels.

1.3.2 Selig Format

It is usefull to have a standard to know how to describe an airfoil but still, a computer program needs a simpler way of reading and saving data of an airfoil. To this end, the Selig format has evolved which describes any airfoil in a list of coordinates and can therefore easily be put in any file and quickly read by any code. The Selig format puts any airfoil between zero and one on the X-axis. Here, the airfoil nose is at the origin and the Trailing Edge (TE) is at the x coordinate one. Beginning from the Trailing Edge (TE) the airfoil is described by walking counterclockwise around it. Therefore, first the upper side is defined, then the nose, and finally the lower side. This is shown in Figure 1.9

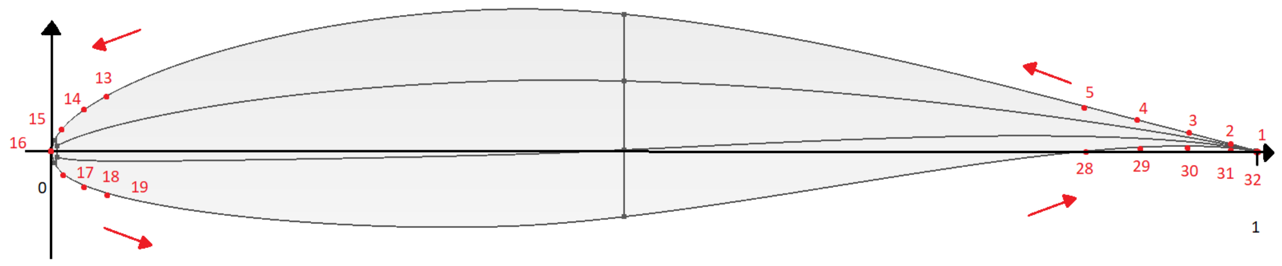


Figure 1.9: visualization on how the Selig format distributes the points

1.3.3 Induced Drag Wingtip

A wing has, generally speaking, one main function, which is to redirect a great portion of the generated drag force of the wing to another wished direction. This is the case for example in an airplane to generate lift and enable the airplane to overcome its force pulling towards the center of the earth induced by its weight, or in a wind energy turbine to set the rotor in a circular rotation to generate electricity. A great portion of the generated drag is simply due to the fact that something is hindering the flow of the surrounding air and this air, generating a resistance. For the wing to generate the desired force it creates a pressure difference between the upper and lower side of the wing and by this, the intended usage is enabled to generate a force towards the low-pressure field. But by this pressure difference, the fluid beneath the wing is forced towards the wingtip, for a finite wing. There the high pressure from the lower side and the lower pressure from the upper side mix together and together with the chordwise flow a vortex is created. So the drag which is due to the pressure differences on the upper and lower side of the wing is responsible for the induced vortex. These vortices can be very big, depending on the size and the weight of the wing and everything attached to it. But they begin at a very small scale and to fully understand and consequently reduce them a precise prediction is necessary to get meaningful data from the simulation. To achieve this it is important to have a small grid at the wingtip to allow the simulation to generate a fine grid and calculate the fluid movement at the necessary refinement. Refinements could be different airfoil shapes or wingtip attachments such as winglets, for example.[4] Such a vortex is shown in Figure 1.10.

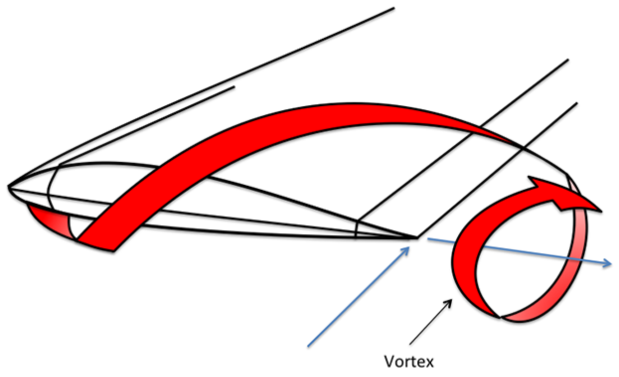


Figure 1.10: visualization of the vortex created at the wingtip of a wing from [11]

1.4 Software

1.4.1 Python

As the programming language for openblademesh, Python was chosen for its simple code structure, fast learnability, and its wide variety of libraries that can be implemented. It can be learned in a relatively fast way, for the language itself is straightforward and many decisions and definitions that other languages require (which are useful and allow them to be faster for example) are taken care of by Python.

For example, it automatically detects which kind of variable is used and declares it as integer or float, just naming an example. For the ease of usability, it is widespread in the scientific as well as other noninformatic communities. For this reason, many programmers have implemented many very useful tools that can be imported, such as the software that is used in this thesis.

1.4.2 Open-Source

The need to program openblademesh in open-source software is derived from many different reasons. Open-source software first and foremost is extremely flexible due to its nature that anybody who wants to implement or change a particular part of the code is free to do so. Therefore the code can grow in many different directions which is especially good for specific use cases.
Also, the speed at which the software can be added is much faster than in commercial software. In commercial software, the distributor needs to be convinced that some changes would be necessary and then the new version is distributed, open-source can just be changed on the run when it is needed. Further many scientific or professional software is very expensive. On that scale, it has to be taken into consideration if some software is really needed. In contrast open-source software is free and therefore and therefore worth it even if it is only needed once. Another key feature of open-source software is that through the large userbase a great amount of improvement is granted.

2 Software used

2.1 Numpy

Numpy enables Python users, besides a simple multidimensional array, to handle efficiently implemented functions for numerical calculations. In this thesis, Numpy is implemented for its array capabilities and its mathematical functions as described in [12] on the Numpy website.

2.2 Scipy

Scipy is a Python library that enables different mathematical features. It is implemented in this code for the Cubic Spline feature which is needed in the wingtip generation process. How Scipy is used can also be looked up in [13].

2.3 GMSH

As a grid generation tool, GMSH (in version 4.11.1) was chosen due to its wide accessibility. GMSH is an open-source three-dimensional finite element mesh generator with built-in Computer Aided Design (CAD) kernel and postprocessing facilities. It is programmable either in the Graphical User Interface (GUI), from the command line, using text files written in Gmsh's own scripting language, or with C, C++, Julia, Fortran, and Python. The latter (Python Version: 3.11.0) was selected for this thesis due to its simple way of operation. GMSH is used by either defining geometric entities with ascending dimensions beginning at points that connect to lines, etc., that work in a bottom-up way or by directly defining bodies such as a cuboid or sphere via implemented codes. For the bottom-up approach, one first has to define points with dimensional entities zero and then lines with dimension one, planes with dimension two, and, if wanted, bodies with dimension three. This is the way it is used in this thesis. It gives the user the option to create the most complex structures. Gmsh has two implemented geometry kernels that can be used, "*geo*" and "*opencascade*". "*Opencascade*" offers the possibility to more easily read through the input files which "*geo*" does not and is therefore used in this thesis. All the information and the Gmsh software are provided under an open-source license and further information as well as the source code and pre-compiled binaries can be found in [14].

Installation Gmsh is installable in a Pythonic way via the

```
pip install --upgrade gmsh
```

command. This way it is installed in the current stable version. All the necessary implementations and bindings are handled by Pip and need no further adjusting.

Gmsh is the foundation for the entire code. The whole building process for a new wing is based upon the bottom-up approach to enable it to read point clouds from airfoil lists online. Therefore the entire geometric building procedure uses GMSH functions to generate the airfoils through which later on the wing will be generated [14]

2.4 Pyhyp

Pyhyp is the open-source software that is used for the extrusions of the grid created in Gmsh. It can generate two or three-dimensional meshes around simple geometric configurations. It starts by creating an initial layer or mesh around the surface and extrudes this gradually until it reaches its desired distance. By this approach, the complete geometry is meshed and for complete documentation please visit the Pyhyp website [15], on which all the options will be explained.

2.4.1 Way of Computation

An overview of the hyperbolic mesh marching method implemented in Pyhyp can be found in Section II. A of Secco et al.,[16], Most of the theory for Pyhyp was taken from Chan and Steger, [4].

2.4.2 Installation

Pyhyp requires other modules installed to work, which are the CGNS library and PETSc. This can either be done by downloading and installing the mdolab/public docker container on docker, which is a container management tool, or by manually installing and linking the modules to Pyhyp. In this thesis, the installation process via docker is recommended for convenience purposes.

Docker is a container virtualization tool that allows the isolation of applications on one computer. With this, it is possible to work in a completely isolated environment. The other advantage and how it is used in this thesis, is for transporting complete program packages with already linked prerequisites. Thus, it takes away the work to download and connect the necessary implementations but the drawback is to work in the docker software in performance losings due to the docker software.

In docker, in the container from mdolab, the provider of Pyhyp, all the prerequisites are already connected, and Pyhyp works. The aforementioned can be downloaded and installed by the following instructions, as presented by the [17], framework.

2.4.3 Implementation

The tool works in two separate parts. First, it creates the geometric grid by extruding through the airfoils and generating the wingtip. Second, it takes the .p3d file generated by Gmsh and extrudes it relative to the starting point normal direction with Pyhyp. All of the Pyhyp-specific attributes are defined in the second part of the code and customization or refinement is possible in this section.

The options to change the way Pyhyp extrudes the grid are presented on the Pyhyp website, [15].

3 Openblademesh

Openblademesh, that is the software developed in this thesis, works by combining the way of operation of Gmsh and Pyhyp. It begins by generating the wing through the given airfoils and then extrudes the mesh around the generated wing. A flow diagram of the whole code is given in Figure 3.1. Here some steps are simplified and shall only gives a quick review of how the code works. Between the creation of the grid by Gmsh and the extrusion of Pyhyp there is the possibility to create a Graphical User Interface (GUI) to see what grid is created and to check if everything is in order.

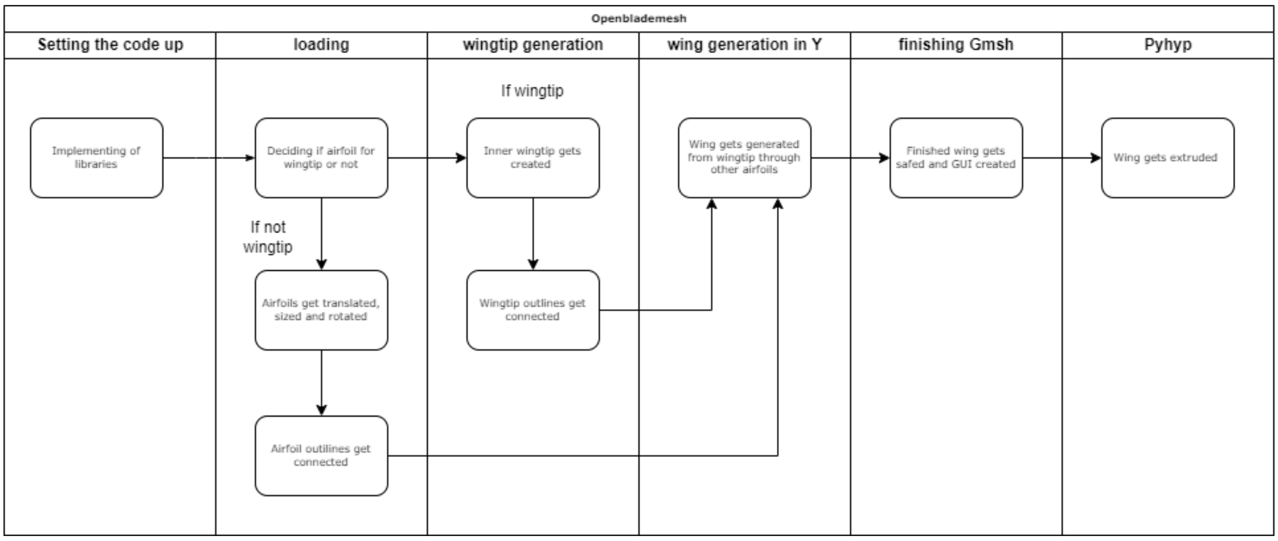

Openblademesh
Setting the code up
loading
wingtip generation
wing generation in Y
finishing Gmsh
Pyhyp
If wingtip
Implementing of libraries
Deciding if airfoil for wingtip or not
Inner wingtip gets created
Wing gets generated from wingtip through other airfoils
Finished wing gets safed and GUI created
Wing gets extruded
If not wingtip
Airfoils get translated, sized and rotated
Wingtip outlines get connected
Airfoil outilines get connected

Figure 3.1: Graph which shows how Openblademesh works

3.1 Folder Structure

The folder structure of Openblademesh contains different folders that contain the given airfoil dataset, the output files, and the main code itself. This way it is always a clean structure even if different data sets are used and therefore different output files are generated.

Data In the data folder, the point clouds need to be in order for the code to work properly, i.e.: beginning at the wingtip and moving towards the wing-root. The first airfoil folder begins with the, in the code defined, name of the airfoil folder, by default "Airfoil", and a number beginning with zero. Subfolders can be implemented for different projects but the code needs to be updated to contain the correct path to the airfoil files. Also, the airfoil files need to be in Selig format for this is the format with which the code is designed to run.

3.1.1 Parameter File

Furthermore, the parameter file is included in the data folder and enables the user to change different parameters. This way, it is possible to change the way the grid is generated. For the parameter file, the body-fixed coordinate system is used. Figure 3.2 demonstrates this system via an example airfoil.

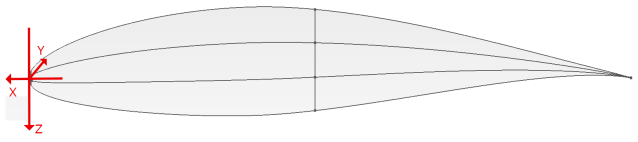


Figure 3.2: Airfoil with coordinate-system by which delta and rx/ry/rz is applied

The first parameters defined are associated with airfoil generation:

Delta The first parameter to define, while it does not matter in which order the parameters are given, is "delta". This gives the distance in the three-coordinate axis from the first airfoil to the next one. It is realized in openblademesh as a simple translation with its directions according to the noninertial body coordinate system as seen in Figure 3.2 and works just by adding the delta value to the according axis, as shown in Figure 3.3.

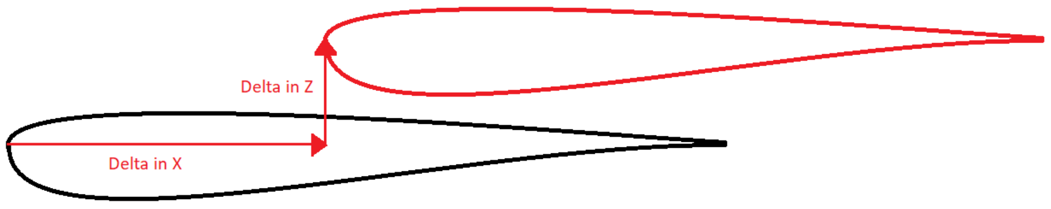


Figure 3.3: Sketch demonstrating the method of translation.

Angle The angle by which the airfoil is rotated is defined by the parameter "angle". It is important that the angle is given in radians. A positive angle results in a counterclockwise rotation.

Pivot Point The pivot point for the rotation is not fixed and is defined by the combination of the parameter "rx", "ry" and "rz". It does not matter if the parameters are given in the order of the airfoil or if first all of the "rx" then "ry" and then "rz" parameters are given. The points defining the pivot point are in Selig format for the airfoil. Figure 3.4 shows that for a rotation first the pivot point is reached and then the whole airfoil is rotated by the predefined angle.

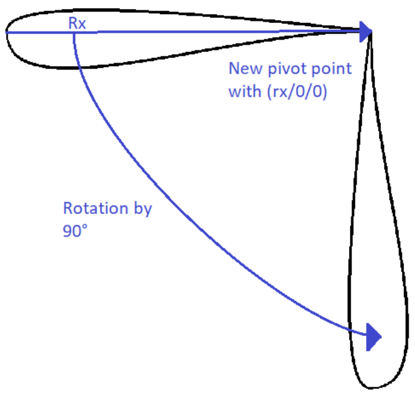


Figure 3.4: Sketch demonstrating the method of rotation.

Sizing Factor The aforementioned sizing factor allows the scaling of the airfoils from the standard chord length of one, from the Selig format, to the desired airfoil chord length by multiplying it.
The following parameters refer to grid generation and node distribution.

n_nodes_horizontal The variable "n_nodes_horizontal" is the number of points on each of the upper parts of the airfoil. For visualization see Figure 3.2. In the picture the lines from the middle of the airfoil to the nose and to the TE are where the here mentioned nodes are set.

By this parameter, the resolution of the grid generation is defined for the wingtip because the number of nodes on the vertical lines on the wingtip and the number of nodes which are on the airfoil nose result from the horizontal nodes and the used progression. How they are calculated is explained in the corresponding part of the code.

Progression1 The parameter defining the progression for n_nodes_horizontal is defined by "Progression1" and is used to calculate the distance taken between each node in the progression of the distribution.

Progression2 Progression2 is the progression for n_nodes_horizontal from the middle of the airfoil towards the Trailing Edge (TE).

n_nodes_vertical_wingtip The last parameter refers to the grid generation in the Z direction. The number of nodes in the Z direction is calculated for every new section between two airfoils. The starting value, which is important to enable a smooth transition between the wingtip and the wing in the Z direction, is the parameter "n_nodes_vertical_wingtip".

n_nodes_vertical_root The parameter for the number of nodes in the last section is "n_nodes_vertical_root". Between these two values, the number of nodes is calculated via an exponential decrease to enable a very precise mesh at the wingtip while still achieving an acceptable computational time by reducing the number of nodes drastically towards the less aerodynamically interesting end of the wing. n_nodes_vertical_root is important because even if the number of points on the wing decreases drastically from the very fine mesh size at the wingtip towards the root, it is very important that at the last section, there are still enough nodes to guarantee that the section gets meshed and can be extruded sufficiently. If it would reduce to one, for example, it would not be possible to create a sufficient mesh. To get a better idea of what the node progression is meant the following Figure 3.5 shows an example wing extruded. From every section beginning at the wingtip towards the root the number of nodes gets reduced and the grid gets thinner.

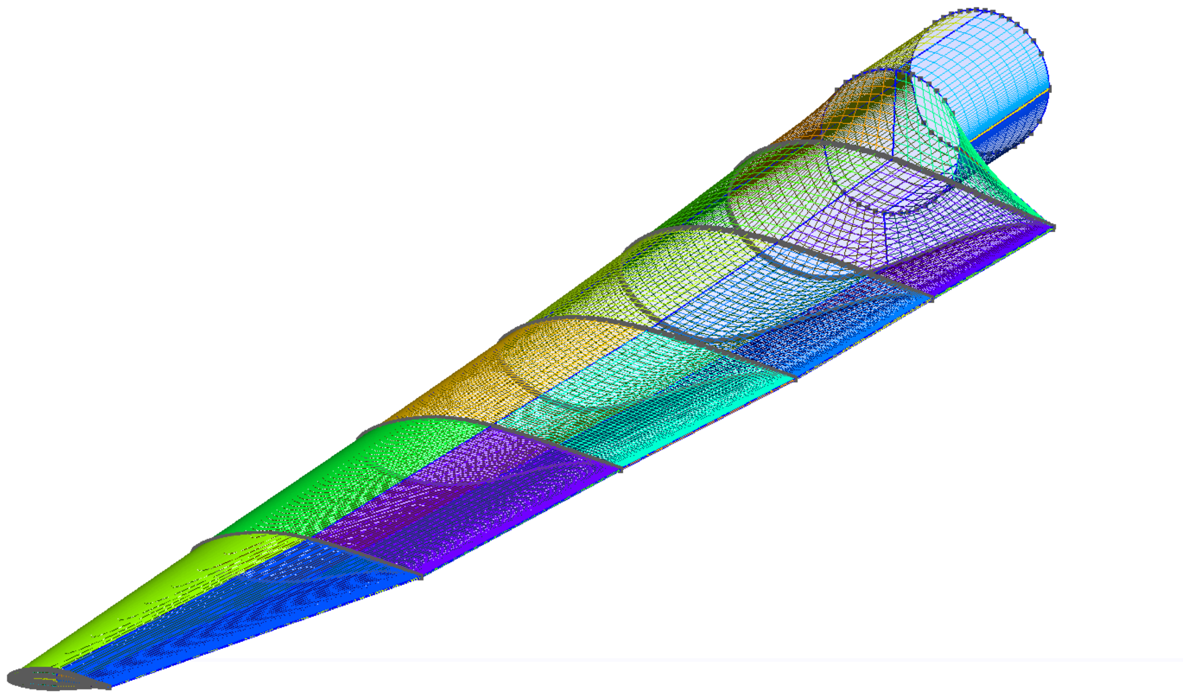


Figure 3.5: Airfoil which shows the point distribution

The concept for this calculation is, to take the length of the trailing edge and divide it by the number of points on this line to get the size of the smallest step on this line. Continuing with this length, the first and therefore smallest step of the exponential increase in the y direction is the just calculated by dividing the TE with the number of nodes, enable the smooth transition.

$$\frac{length\_of\_TE}{points\_on\_TE} = l * \frac{r^n - 1}{r - 1} \tag{3.1}$$

For the points distribution, the "*Progression*" mode is used. With this, first, the exponential decrease over the whole wing is calculated, and trying to mimic these curves on each subsection between two airfoils with extra "*Progression*" calculations the whole wing is set. However, the algebraic calculation takes too long to calculate and the numerical calculation was not implementable until now. So the current code calculates on each section one value for the number of points and sets it to this fixed value. This is equated with a negative exponential function with a fixed value for the number of nodes on the section closest to the wing root. This equation is given in the respective part of the code explanation.

Output The output folder contains the output files created by Gmsh. This output is the surface created by Gmsh and will be further used by Pyhyp to extrude the wing in the y direction.

Src The main code implementation is in the src folder and contains the actual Python code. In the following, the code is explained piece by piece and shall thereby help the user understand the best way to work with openblademesh, or how to implement or add new features if desired.

3.1.2 Section1: Setting up the code

To operate Gmsh, it first has to be initialized, which is done in the code beneath the comment section1 see appendix, and a new model has to be specified which is named "wing" in this example. This model is internal for Gmsh and does not appear in another point of Openblademesh.

Here all the needed parameters to create the grid as desired are imported from the parameter file and saved in the dictionary "Variables" by which they are called throughout the whole program. Also the three developer modes "*Only_wing_tip_generation*", "*create_GUI*" and "*extrude_wing*" exist to to decide if only parts of the code shall be computed to simplify and speed up the computation time. The code loops through the parameter file and searches for the line containing the name of the parameter, replaces the name and the following equal sign, and saves what is left in the parameter of interest. If there are several pieces of information in one line, see "delta" for example, the letters are separated and saved as lists in the parameter which can be a list containing information for different airfoils. Because some of the parameters contain more than one piece of information they are set to be lists while others need to be integers. The distinction between these two is realized by a simple if loop which checks for the type of the called variable.

3.1.3 Section2: Loading files

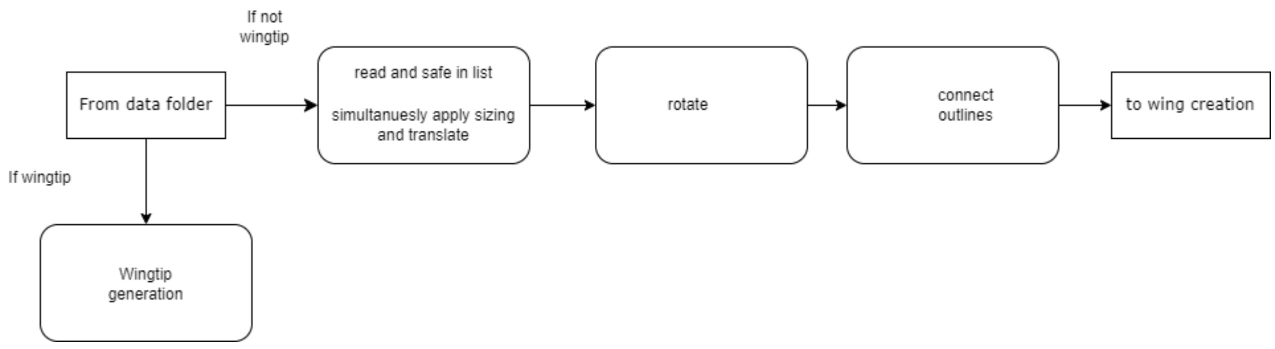


Figure 3.6: Graph depicting how the loading part of the code works.

The above graph Figure 3.6 gives an overview of how the loading part of the code works.

The following two paragraphs in this section are done separately for every airfoil. In this section, all the files are loaded into the code and the outlines of the airfoils, which are not the wingtip, are connected with splines.

The first lines in section 2 of the code define a function that counts from zero until the end of the first airfoil and then jumps to the number of the end of the second airfoil plus six. The six extra points are added to make space for the first and last points in the inner subsection of the airfoil which are the two generated points by the spline function in the inner airfoil at the trailing edge, the two points on the outer airfoil at the airfoil nose that are created by the spline to be more flexible from the points in the airfoils and finally the last two points that are on the inner airfoil at the airfoil nose. The loop continues to count up from there. The aforementioned points are shown in the following two pictures Figure 3.7 and Figure 3.8.

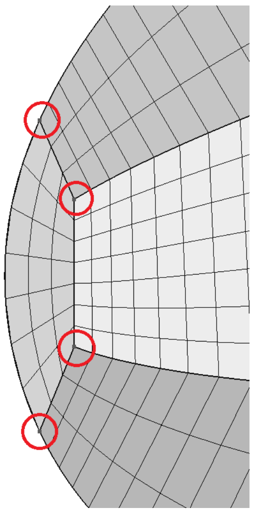


Figure 3.7: Airfoil with highlighted points created with spline

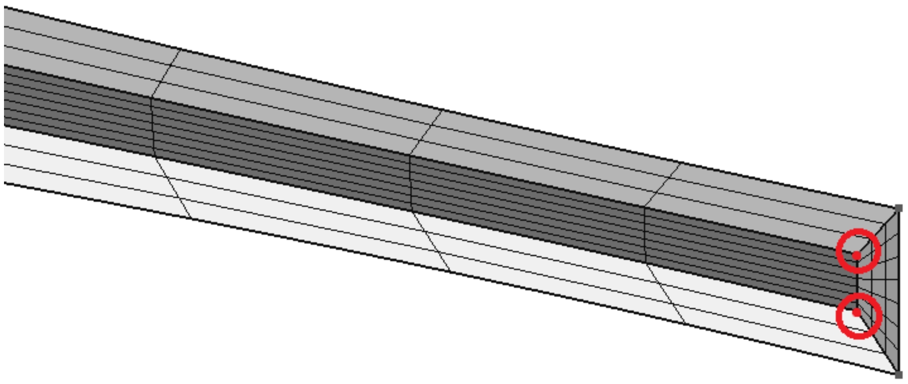


Figure 3.8: Airfoil with highlighted points created with spline

The just created list is now sorted in a list of lists of X and Y points to simplify the accessibility. This way, the code can work with all the points given and reach its X and Y values. Furthermore, the points are manipulated by delta and the rotation if needed in the parameter file that just has been defined.

It is important to note that the Gmsh function used to add points has downsides, such as, that the points are not callable later on. Therefore all the points for the airfoils will be added to the grid, but also a separate file will be added that includes the same points. This is important because while Gmsh adds to every added point a tag to identify the point later on, it is not possible to access point values from specific points for separate lists for the different airfoils for example. Therefore the points are saved twice.

Then the code continues to sort the airfoils that are not the first one, because the first airfoil is the wingtip, in a way that allows later on a transfinite algorithm. To this end, it counts the points that are created in a counterclockwise direction starting from the trailing edge. This is defined by the Selig format which is used.

Subsequently, the points in the file are rotated. This is done at this point and not earlier to achieve consistent handling. If, for example, the airfoil would be rotated so that the leading edge is no more the point with the smallest X value the code would create a wrong sorting.

Now all the points are connected with splines by the earlier defined sections. Also, they are given a tag and a line is created which connects the upper and lower trailing edge. Gmsh needs to create a surface through a closed wire of lines, a so-called curve loop, which is defined by all the just created splines and the connecting line.

3.1.4 Section3: Wingtip generation

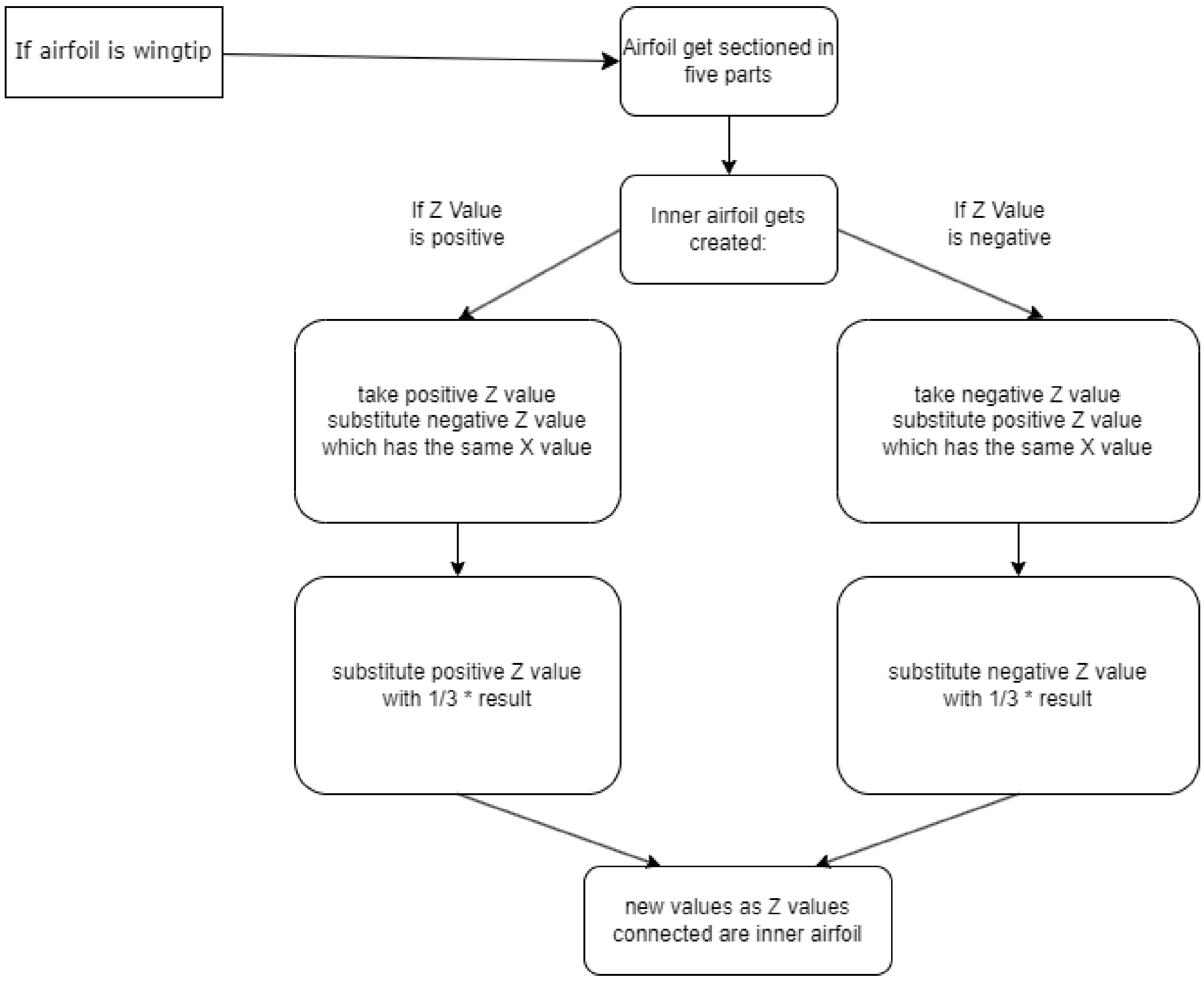

If airfoil is wingtip
Airfoil get sectioned in five parts
Inner airfoil gets created:
If Z Value is positive
If Z Value is negative
take positive Z value substitute negative Z value which has the same X value
take negative Z value substitute positive Z value which has the same X value
substitute positive Z value with 1/3 * result
substitute negative Z value with 1/3 * result
new values as Z values connected are inner airfoil

Figure 3.9: graph which shows how section 3 works

The flow diagram above Figure 3.9 shows the way the wingtip generation is realized.

The wingtip on a wing is responsible for a great amount of the induced drag. Because the importance of the wing tip in a Computational Fluid Dynamics (CFD) simulation of a wing is very high, the definition of the wingtip takes a great part of openblademesh and will be described in detail.

To generate a structured mesh it is necessary to have surfaces with four corners, given that the mesh should be quadlinear. To achieve that, the first airfoil is subsectioned into eight parts. As seen in the figures below *Figure* 3.12 are the results that are achieved in the following chapter.

The boundaries of the inner airfoil are generated by taking the difference between the upper and lower airfoil and either substituting or adding a fraction of this value to the original point depending on whether the original point is on the upper or lower airfoil. As a clarification, the code walks from the Trailing Edge (TE) towards the nose, takes every Z coordinate, and compares it to the Z coordinate with the same X coordinate. As seen in Figure 3.13 if the first value is positive, marked

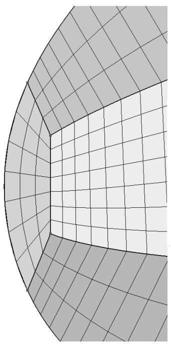


Figure 3.10: Airfoiltip

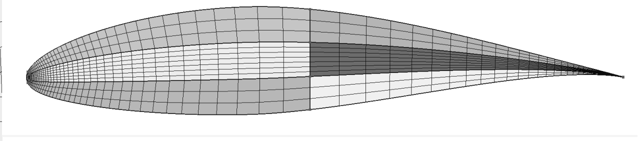


Figure 3.11: Airfoil

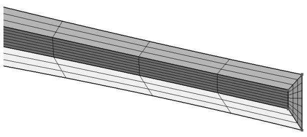


Figure 3.12: Airfoil trailing edge

in blue, then the negative value, here in yellow, is substituted. The result is then multiplied by 1/3 and substituted from the first value, shown in green in the figure. This results in an inner airfoil that is a third the size of the original airfoil and follows the shape of the first airfoil very precisely, even in concave areas.

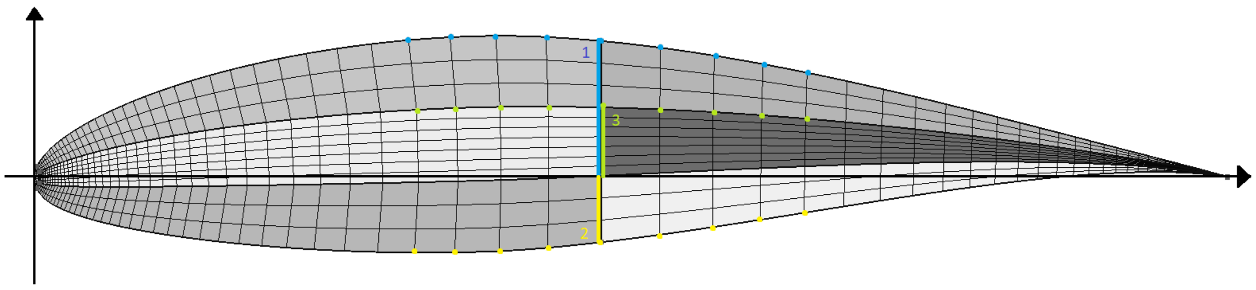


Figure 3.13: Airfoil with highlighted distances which are used to create the inner airfoil for the wingtip

By this approach, even very small and curved airfoils can be achieved and will result in the best grids possible by this big of a size difference. An example of such a difficult airfoil is given in Figure 3.14 below which shows the TE of the NREL 5MW wind energy blade which is fully described in the examples section.

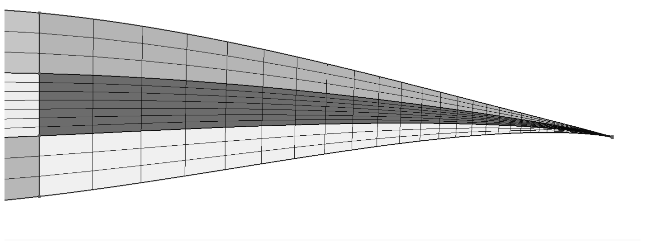


Figure 3.14: Trailing edge section of the NREL 5MW wingtip

The X values of the start and endpoints of the just-created splines are defined and saved in one list. The points for the end of the splines are created so that the connection lines from these to the outer airfoil are 45 degrees. The 45 degrees are important because they allow for the most regular translation of cells on each side of the connecting line.

Next up the spline is created which defines the position of the connection points of the inner and outer airfoils. This spline is purely to compute positions on the inner airfoil and is not seen in the model. It is calculated with the Scipy cubic spline function. Therefore at first, the corresponding X values are calculated to enable a 45-degree angle in the trailing edge to to yield cells as uniform as possible. The Trailing Edge (TE) of the example airfoil is given below to give a good example of how the Trailing Edge (TE) is created Figure 3.15.

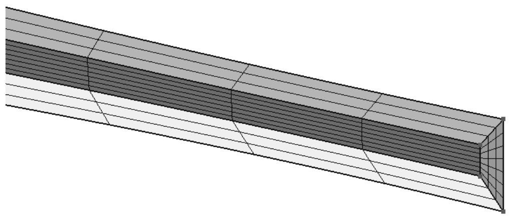


Figure 3.15: Airfoil TE

The airfoils are sub-sectioned to achieve planes with four sides to enable the transfinite algorithm.

To subsection the outer airfoil several points are placed on the airfoil. As seen in Figure 3.20 with with every color change a new plane occurs but also following the outline of the airfoil another part of the outline.

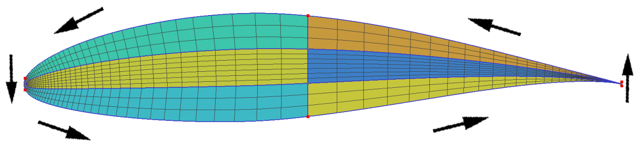


Figure 3.16: Airfoil depicting the marching direction of the points around the airfoil

Following the same idea of partitioning, the inner airfoil is sectioned. Starting from the upper point of the endpoint of the inner airfoil, towards the airfoil nose and then, following the arrows on Figure 3.21 back to the Trailing Edge (TE).

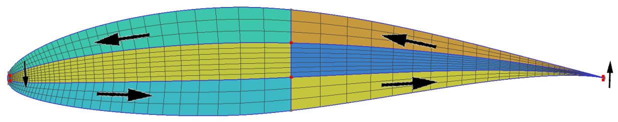


Figure 3.17: Airfoil depicting the marching direction of the points around the inner airfoil part

To achieve the desired planes with four edges the starting and end points as well as the middle points on the wingtip need to be connected with lines. To enable Gmsh to create a surface from the created splines and lines, curve loops are created in the way shown in the picture below Figure 3.22, and planes are installed in them.

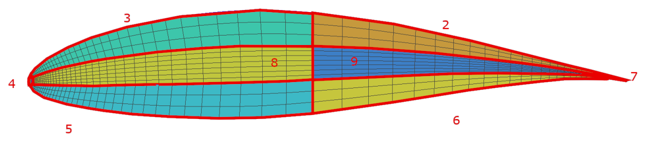


Figure 3.18: Airfoil depicting the curve loops and planes created with its respective numbers

Lastly in the wingtip generation process, to achieve a structured grid, the curves that connect the airfoil nose and the Trailing Edge (TE) and therefore represent the biggest part of the airfoil, are defined to be transfinite and it is defined how many points are set on each curve. The transfinite algorithm refers to the transfinite interpolation from numerical analysis which constructs functions over a planar surface so that these functions fit to boundary conditions on the boundary. Here the transfinite algorithm function is used to define a set number of points in a specific distribution. This distribution is realized with the so-called "*progression*" mode which is a geometrical series and distributes the points depending on the coefficient and number of points across the given line.

Because it is crucial to have smooth transitions between every cell, the number of nodes on the airfoil in the vertical direction and the number of nodes on the connection lines between the outer and inner airfoil are calculated by the following formula

$$\frac{length\_vertical\_line}{nodes\_on\_vertical\_line} = l \cdot \frac{r^n - 1}{r - 1} \tag{3.2}$$

The vertical line is the line of the inner airfoil at the nose section divided by the nodes on this line. l is the length of the curve on which the progression is set, r is the set coefficient and k is the number of nodes on this curve. This equation is converted for the number of nodes on the line and derives from

$$\sum_{k=0}^{n-1} a * r^k = a_0 * \frac{r - 1}{r^n - 1} \tag{3.3}$$

which is the geometrical series as described in the fundamentals section. Important to note is that a_0 is the smallest step taken.

$$\frac{length\_vertical}{number\_of\_nodes} = a_1 \tag{3.4}$$

In this equation $a1$ is the smallest step on the line which is examined. The number of nodes on the vertical line is calculated to achieve the needed smallest step.

Via the same approach, the amount of nodes on the connection lines between the inner and outer airfoil is calculated, with the difference that the line, for which the amount of nodes is calculated is now the connection line between the outer airfoil and the inner airfoil towards the nose section. For clarification see the next picture Figure 3.29. In this picture the crossing between the three lines is highlighted. The vertical line is for Figure 3.23 and the line that connects the inner and outer airfoil is for Figure 3.29. The last curve that follows the direction of the airfoil is the curve on which the progression is given with Progression1.

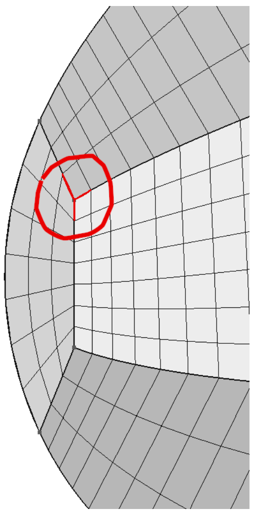


Figure 3.19: Airfoil with highlighted section by which the point distribution is calculated.

By this, the bordering cells on each side of the lines have the same size and enable regular cuboids to achieve a small sizing difference between each cell.

Because computation time is a major point to consider in Computational Fluid Dynamics (CFD) simulation it is important to save nodes where possible. So on these curves, the number of nodes is highest towards the nose and towards the Trailing Edge (TE) to achieve a good resolution of the important parts while simultaneously saving nodes in the less aerodynamically complicated middle. This distribution is a geometrical progression in two different directions which is calculated by the geometric series formula Figure 3.27 shown above.

3.1.5 Section4: Wing creation in Y direction

Here, the actual creation of the wing starts by creating a two-dimensional grid beginning at the first airfoil and connecting the other airfoils. To achieve a structured mesh on the newly generated planes again the transfinite algorithm is used. The corresponding curves on the newly generated airfoils to the curves on the outer wingtip airfoil are also set to be with transfinite algorithms with the same coefficients.

The newly created connection lines between the boundaries of the subsections between the airfoils are also set to be handled with the transfinite algorithm and here they are defined with computational time in mind. Because the adjacent cells need to be as similar as possible it is important to generate cells on the wing towards the wingtip which are as similar as possible to the cells on the wingtip on the edge. Because the wingtip will have very small cells, especially at the Trailing Edge (TE), the cells on the wing need to be very small. However, if this cell dimension would be consistent over the whole wing the computational load would be enormous without any major improvement to the simulation.

To achieve the transition between the refined amount of points towards the wingtip and fewer points towards the wing root, a formula is implemented which calculates the number of points in the Y direction for every section. But the algebraic solution approach takes too long and the numerical approach is yet to be finished so the formula is given here but is not used.

The idea behind this is to recreate one geometric series over the complete wing by geometric series on smaller subsections.
1. This is done beginning with the formulas for the geometric series over the whole wing with the set boundary conditions as the first step size $a1$, the length of the last section $l3$, the number of nodes on the entire wing $n1$, and number of nodes over the whole wing except for the nodes on the last section $n2$.

$$\sum_{k=0}^{n1-1} a1r1^k - \sum_{k=0}^{n2-1} a1r1^k - l3 = 0 \tag{3.5}$$

$$a1 = \frac{Te}{nodes - nose} \tag{3.6}$$

$$n2 = n1 - Nmin \tag{3.7}$$

by defining the first sum for the complete sum and the second one without the last section to enable the least number of nodes on the last section. This is realized via the factor that is substituted at the end. This equation should be solved for the coefficient of the sum $r1$

2. With the coefficient for the complete wing $r1$, the first step in the first section $a1$, which is the same as the whole wing and the length of the first section $l1$, with the equation

$$\sum_{k=0}^{n1-1} a1r1^k - l1 = 0 \tag{3.8}$$

the number of nodes for the first section can be calculated.
3. With this, the first step size in the first section and the coefficient for the whole wing via the equation

$$\sum_{k=0}^{n3} a1r1^k - \sum_{k=0}^{n3-1} a1r1^k = a2 \tag{3.9}$$

that takes as the first sum the geometric series until on node further than the end of the first section $n3$ and as the second sum the series until the second section $n3$ - 1, the first step size in the second section is equated which is $a2$.

4. Taking the number of nodes starting from the second section $n3$, the first step size $a2$, the coefficient for the whole wing (that is the same as for the first section) $r1$, the number of nodes of the whole wing $n1$ and the step size for the second section $a2$

$$\sum_{n3}^{n1-1} a2r1^k - \sum_{k=0}^{n1-1} a2r2^k = 0 \tag{3.10}$$

the coefficient for the second section is calculated.

repeating these steps from 2. to 4. until the last section of the wing is reached in which the number of nodes is already known.

With this approach, a very smooth transition between a high number of nodes at the wingtip to the less detailed section at the root can be achieved using the progression Gmsh provides.

But as this approach is not available at the current state of implementation, a second approach was chosen which is coarser. So, every new surface, created by interpolating between two airfoils, gets a smaller amount of points on the wing and therefore less to compute. This is realized by a negative exponential formula.

Here n is the number of nodes on the wing root and a is calculated by dividing the length of the Trailing Edge (TE) by the number of nodes on it. x represents the length of the wing.

$$n \cdot e^{ax} \tag{3.11}$$

The resulting number of nodes at the beginning of each new section is taken and therefore the quality of the grid refines drastically for more sections but is not sufficient for the example with the NREL 5MW. So every section gets a new distance between two points and on the section.

At last, the new surfaces get set to be structured as well by the transfinite approach.

3.1.6 Section5: Finishing Gmsh

With the recombine command the grid is set to be quadrangular and with the write command, it is specified where the output file is to be written.

The last two commands create a Graphical User Interface (GUI) so that the user can see and rate the resulting grid.

3.1.7 Section6: Pyhyp

Here, Pyhyp begins with its extrusion of the just-created wing. Here all the options from the Pyhyp website are implemented. Because of the problems with the node distribution calculations mentioned earlier, Pyhyp cannot extrude the whole wing for too big step size differences between the wingtip and the wing.

3.2 User Manual

To use openblademesh, the first step is to implement the researched airfoils into the data folder and label each file with airfoil and its respective number. Also, it is important to save each airfoil in Selig format to make sure that the program can work with it as it is supposed to.

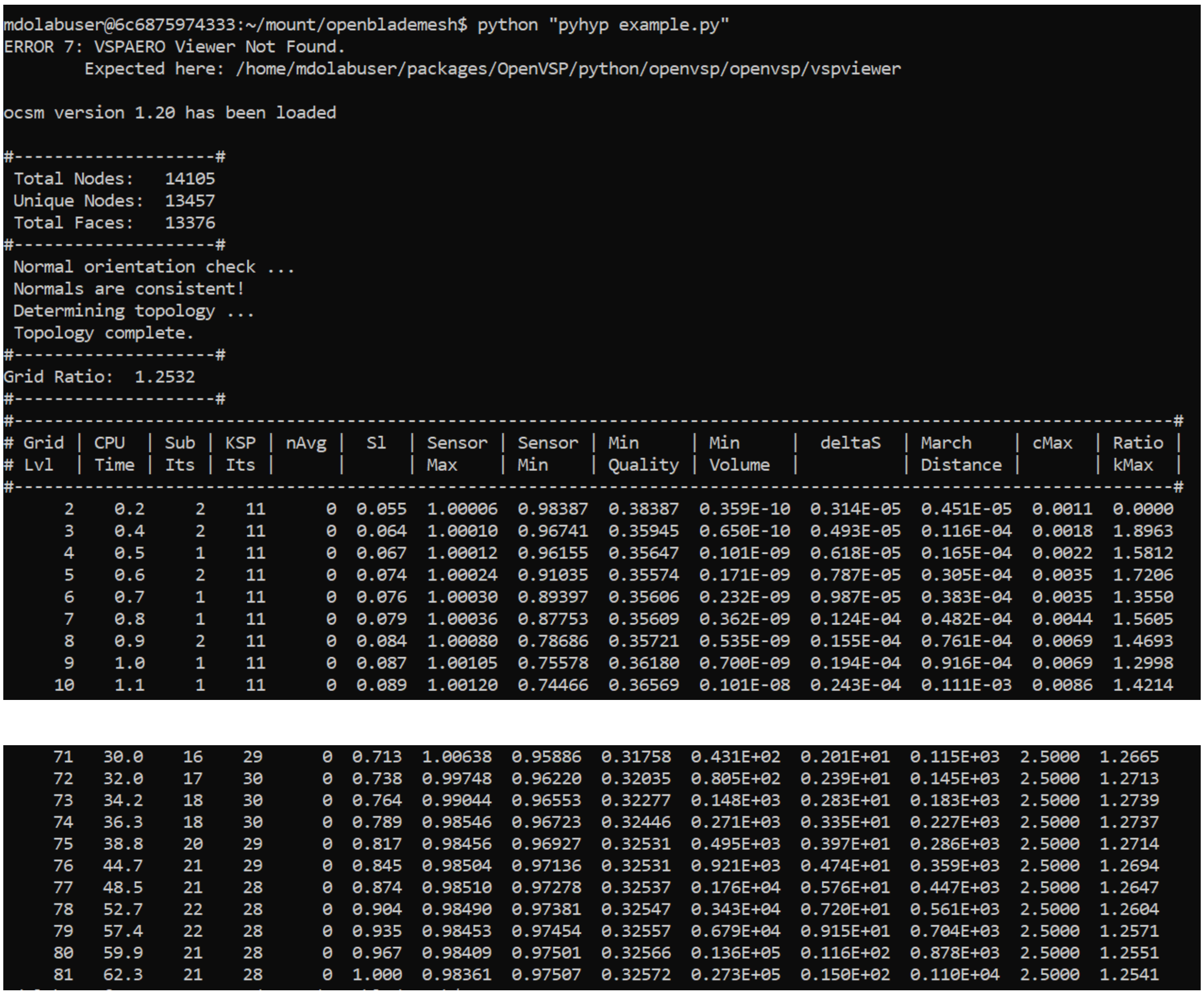
In the parameter file, the necessary parameters to build the grid need to be updated and the Pyhyp data needs to be set to enable openblademesh to generate a mesh of the grid. If it is wished to first just generate the wingtip without the rest of the wing to save time while checking for an even grid generation, the line which is at the beginning of the code before it reads the parameter file: only_wingtip_generation can be set to TRUE, to skip over the rest of the wing generation and only generate the wingtip.

3.3 Examples

3.3.1 Pyhyp Example

To demonstrate an example of how Pyhyp works, even if openblademesh has problems with extruding due to the big difference in steps in grid size, a Pyhyp example is shown in the following. First the necessary options are implemented to achieve a solid extrusion. In this case, 81 steps with an initial step size of $4 \cdot 10^{-6}$ are used, and a march distance in the normals direction of 1100. Pyhyp imports the file, in this case, a p3d file, and starts extruding it. It automatically tries to show the result in VSP AERO which is a visualization software and would show the extruded wing, but because the code is used in the docker container it is not able to use the graphic card and therefore the error arises, as seen in Figure 3.20.
Next, it shows the user the amount of nodes and faces found in the file and how many of the nodes are unique. Through the extrusion of every cell and with a small tollerance it is possible for cells to be generated double. To be able to create a mesh, the grid needs to have consistent normals for every face, which is checked by Pyhyp next. Finally, before it starts to extrude, a grid ratio is given which is an evaluation number for the grid and should be between 1 and 1.2 as the Pyhyp documentation recommends.

Now Pyhyp creates a table with all the relevant information for the user. Most important is the column that displays the Min Quality which indicates the evenness and regularity for each step and should always be greater than zero. As seen in Figure 3.20 for this extrusion the quality is sufficient and therefore the extrusion will be valid.



```
mdolabuser@6c6875974333:~/mount/openblademesh$ python "pyhyp example.py"
ERROR 7: VSPAERO Viewer Not Found.
        Expected here: /home/mdolabuser/packages/OpenVSP/python/openvsp/openvsp/vspviewer

ocsm version 1.20 has been loaded

#--------------------#
 Total Nodes:   14105
 Unique Nodes:  13457
 Total Faces:   13376
#--------------------#
 Normal orientation check ...
 Normals are consistent!
 Determining topology ...
 Topology complete.
#--------------------#
Grid Ratio:  1.2532
#--------------------#
#--------------------------------------------------------------------------------------------------------#
# Grid | CPU  | Sub | KSP | nAvg |   Sl   | Sensor | Sensor | Min     | Min       |   deltaS   | March    | cMax    | Ratio |
# Lvl  | Time | Its | Its |      |        | Max    | Min    | Quality | Volume    |            | Distance |         | kMax  |
#--------------------------------------------------------------------------------------------------------#
      2    0.2     2    11      0  0.055  1.00006  0.98387  0.38387  0.359E-10  0.314E-05  0.451E-05  0.0011  0.0000
      3    0.4     2    11      0  0.064  1.00010  0.96741  0.35945  0.650E-10  0.493E-05  0.116E-04  0.0018  1.8963
      4    0.5     1    11      0  0.067  1.00012  0.96155  0.35647  0.101E-09  0.618E-05  0.165E-04  0.0022  1.5812
      5    0.6     2    11      0  0.074  1.00024  0.91035  0.35574  0.171E-09  0.787E-05  0.305E-04  0.0035  1.7206
      6    0.7     1    11      0  0.076  1.00030  0.89397  0.35606  0.232E-09  0.987E-05  0.383E-04  0.0035  1.3550
      7    0.8     1    11      0  0.079  1.00036  0.87753  0.35609  0.362E-09  0.124E-04  0.482E-04  0.0044  1.5605
      8    0.9     2    11      0  0.084  1.00080  0.78686  0.35721  0.535E-09  0.155E-04  0.761E-04  0.0069  1.4693
      9    1.0     1    11      0  0.087  1.00105  0.75578  0.36180  0.700E-09  0.194E-04  0.916E-04  0.0069  1.2998
     10    1.1     1    11      0  0.089  1.00120  0.74466  0.36569  0.101E-08  0.243E-04  0.111E-03  0.0086  1.4214
```

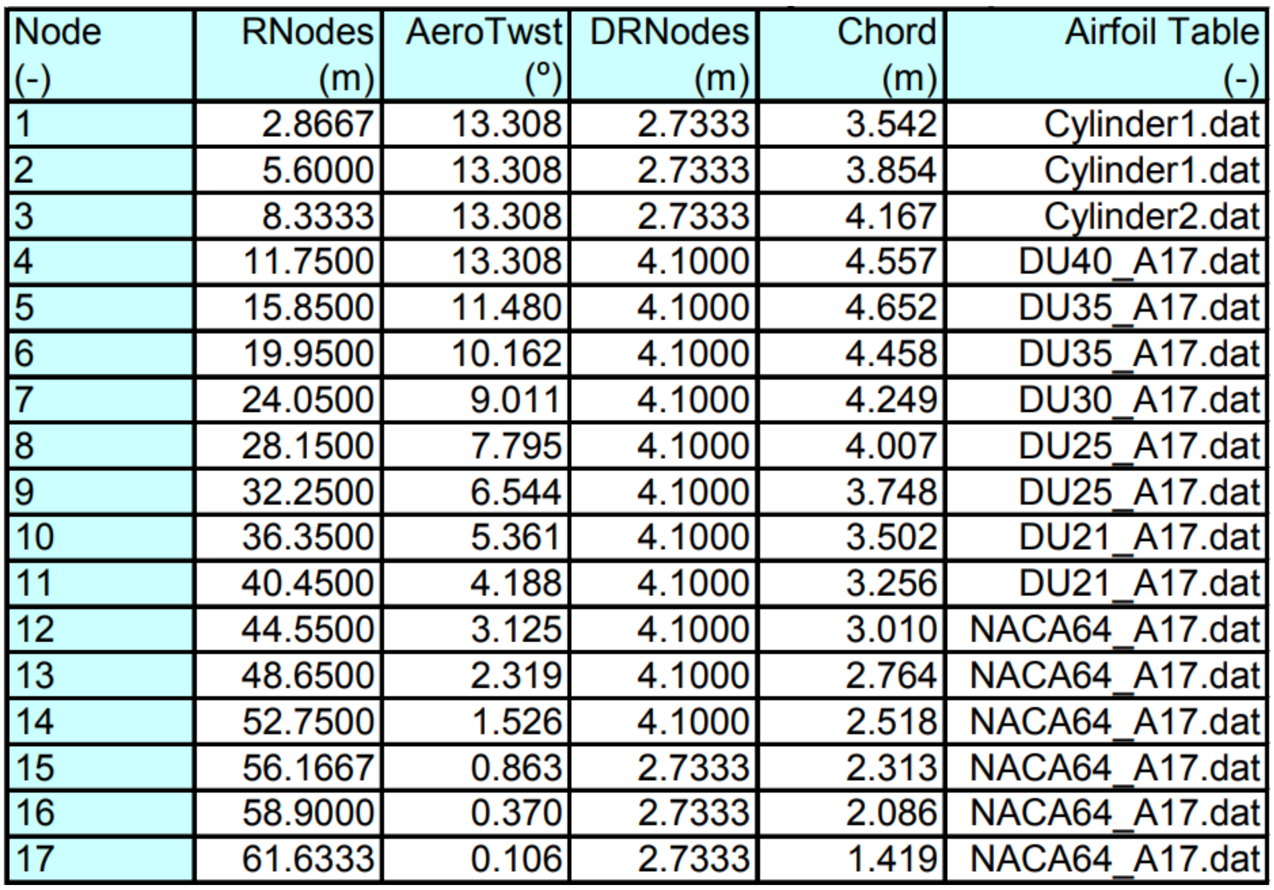
```
     71   30.0    16    29      0  0.713  1.00638  0.95886  0.31758  0.431E+02  0.201E+01  0.115E+03  2.5000  1.2665
     72   32.0    17    30      0  0.738  0.99748  0.96220  0.32035  0.805E+02  0.239E+01  0.145E+03  2.5000  1.2713
     73   34.2    18    30      0  0.764  0.99044  0.96553  0.32277  0.148E+03  0.283E+01  0.183E+03  2.5000  1.2739
     74   36.3    18    30      0  0.789  0.98546  0.96723  0.32446  0.271E+03  0.335E+01  0.227E+03  2.5000  1.2737
     75   38.8    20    29      0  0.817  0.98456  0.96927  0.32531  0.495E+03  0.397E+01  0.286E+03  2.5000  1.2714
     76   44.7    21    29      0  0.845  0.98504  0.97136  0.32531  0.921E+03  0.474E+01  0.359E+03  2.5000  1.2694
     77   48.5    21    28      0  0.874  0.98510  0.97278  0.32537  0.176E+04  0.576E+01  0.447E+03  2.5000  1.2647
     78   52.7    22    28      0  0.904  0.98490  0.97381  0.32547  0.343E+04  0.720E+01  0.561E+03  2.5000  1.2604
     79   57.4    22    28      0  0.935  0.98453  0.97454  0.32557  0.679E+04  0.915E+01  0.704E+03  2.5000  1.2571
     80   59.9    21    28      0  0.967  0.98409  0.97501  0.32566  0.136E+05  0.116E+02  0.878E+03  2.5000  1.2551
     81   62.3    21    28      0  1.000  0.98361  0.97507  0.32572  0.273E+05  0.150E+02  0.110E+04  2.5000  1.2541
```

Figure 3.20: Pyhyp output example file which was given by the Pyhyp documentation

3.3.2 NREL 5MW

As an example to demonstrate how openblademesh works, the NREL 5MW wind energy wing was chosen. Further information about this wind power plant can be seen here [18].

Therefore it is quite easy to get access to all of the necessary data [18], including which airfoils are used and the respective data, which is the distance from the wingtip and the chord length which is easily adapted to the sizing factor for every airfoil is in Selig format and therefore with a chord length of one, given. The used data can be seen in Figure 3.21



Node (-)	RNodes (m)	AeroTwst (º)	DRNodes (m)	Chord (m)	Airfoil Table (-)
1	2.8667	13.308	2.7333	3.542	Cylinder1.dat
2	5.6000	13.308	2.7333	3.854	Cylinder1.dat
3	8.3333	13.308	2.7333	4.167	Cylinder2.dat
4	11.7500	13.308	4.1000	4.557	DU40_A17.dat
5	15.8500	11.480	4.1000	4.652	DU35_A17.dat
6	19.9500	10.162	4.1000	4.458	DU35_A17.dat
7	24.0500	9.011	4.1000	4.249	DU30_A17.dat
8	28.1500	7.795	4.1000	4.007	DU25_A17.dat
9	32.2500	6.544	4.1000	3.748	DU25_A17.dat
10	36.3500	5.361	4.1000	3.502	DU21_A17.dat
11	40.4500	4.188	4.1000	3.256	DU21_A17.dat
12	44.5500	3.125	4.1000	3.010	NACA64_A17.dat
13	48.6500	2.319	4.1000	2.764	NACA64_A17.dat
14	52.7500	1.526	4.1000	2.518	NACA64_A17.dat
15	56.1667	0.863	2.7333	2.313	NACA64_A17.dat
16	58.9000	0.370	2.7333	2.086	NACA64_A17.dat
17	61.6333	0.106	2.7333	1.419	NACA64_A17.dat

Figure 3.21: NREL 5MW table from where the data was taken from [19]

With the adaptation of the parameter file to the needed data, all the user has to do to implement the wing data to openblademesh is done. How it is implemented can be seen in the first appendix by the parameter file.

The next and last step is to polish the resulting grid. The recommendation is to first only create the wingtip with the default values for the point distribution and progressions and see what the code presents. If problems result, such as that the grid is very uneven or does not fit the wanted fineness, a closer inspection of the points and progression is needed. But for this example, the wingtip looks sufficient because the progression is smooth, towards the nose and the wingtip, the nodes get refined, and no obvious difference in cell size can be seen as shown in Figure 3.22

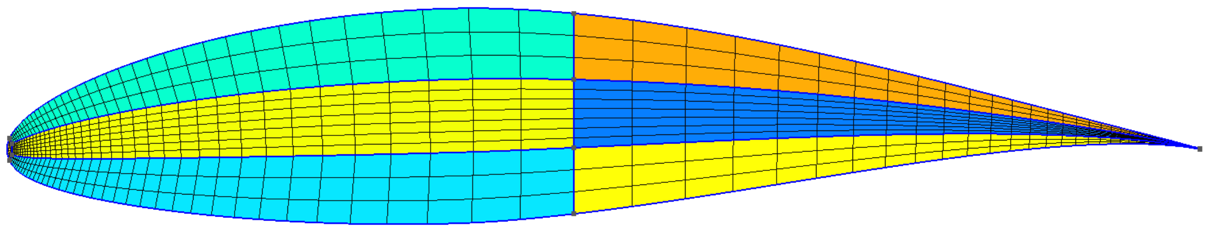


Figure 3.22: NREL 5MW wingtip grid

Now the code needs to generate the two-dimensional grid of the wing. To do so only_wingtip_generation is set to FALSE and with the default values for the point distribution over the wing the code is run again and the generated wing can be inspected as in Figure 3.23 shown.

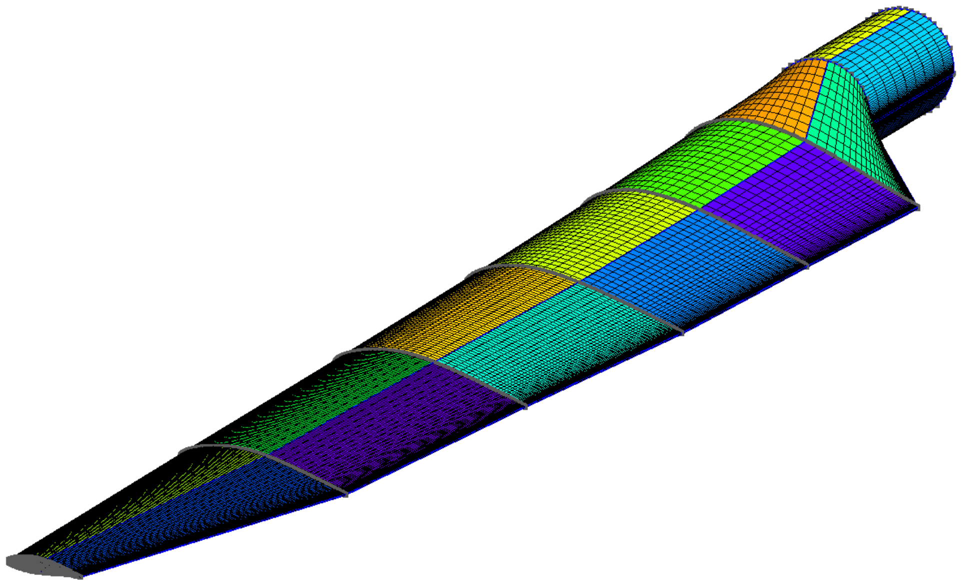


Figure 3.23: NREL 5MW Grid

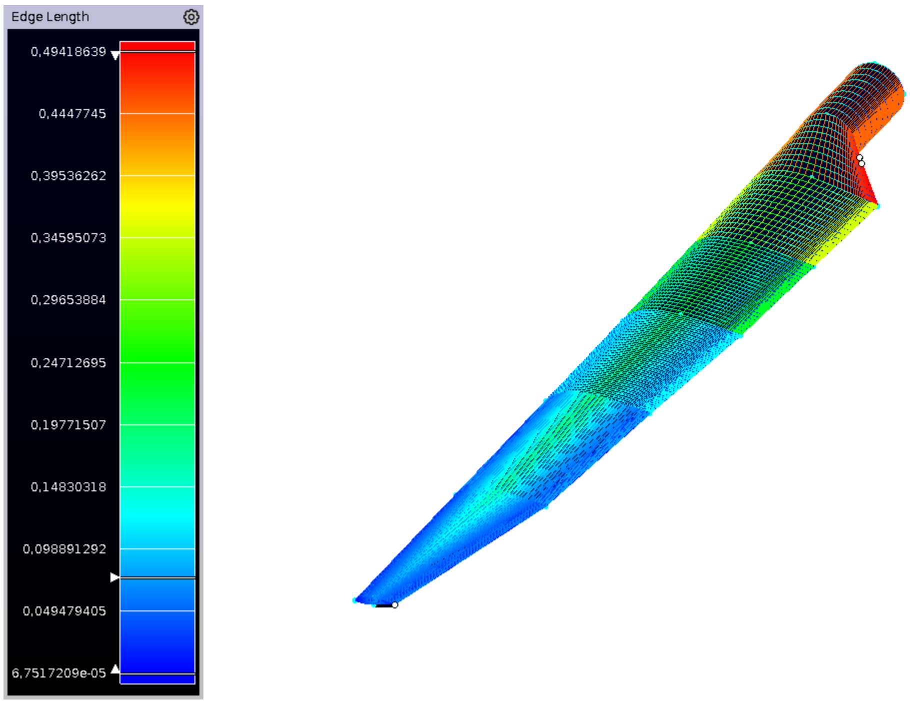


Figure 3.24: edge length

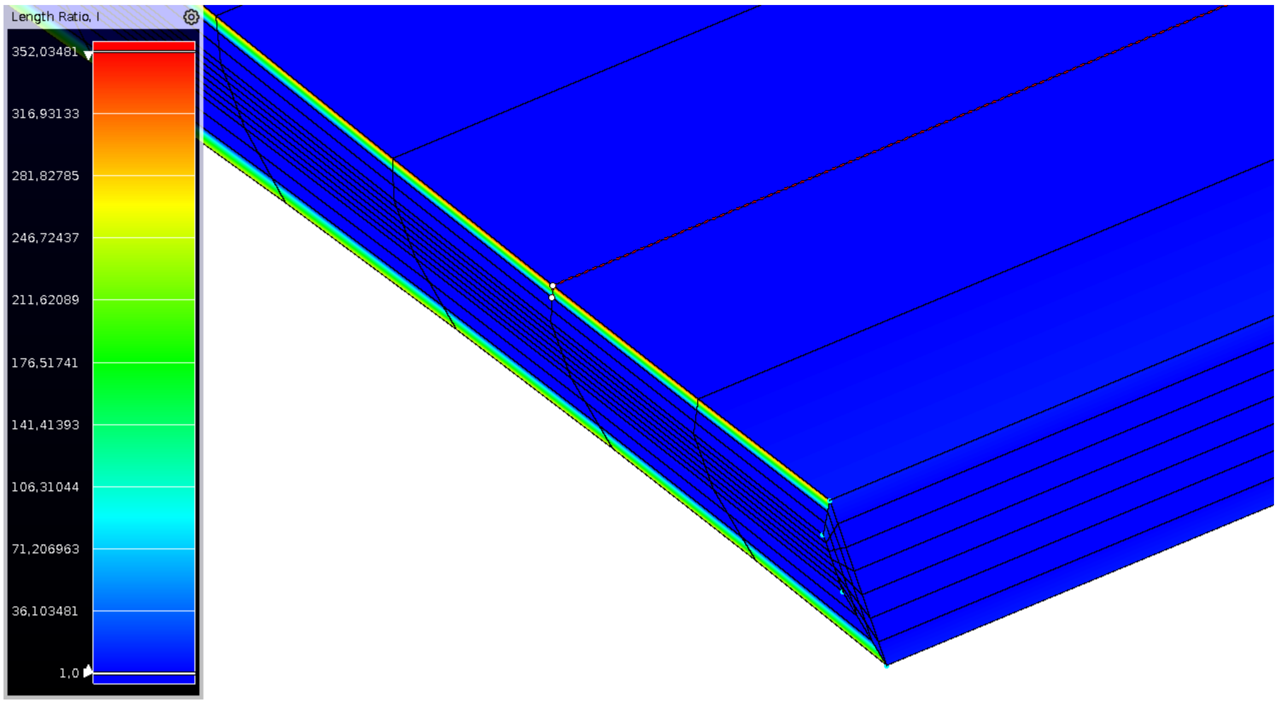


Figure 3.25: length ratio i

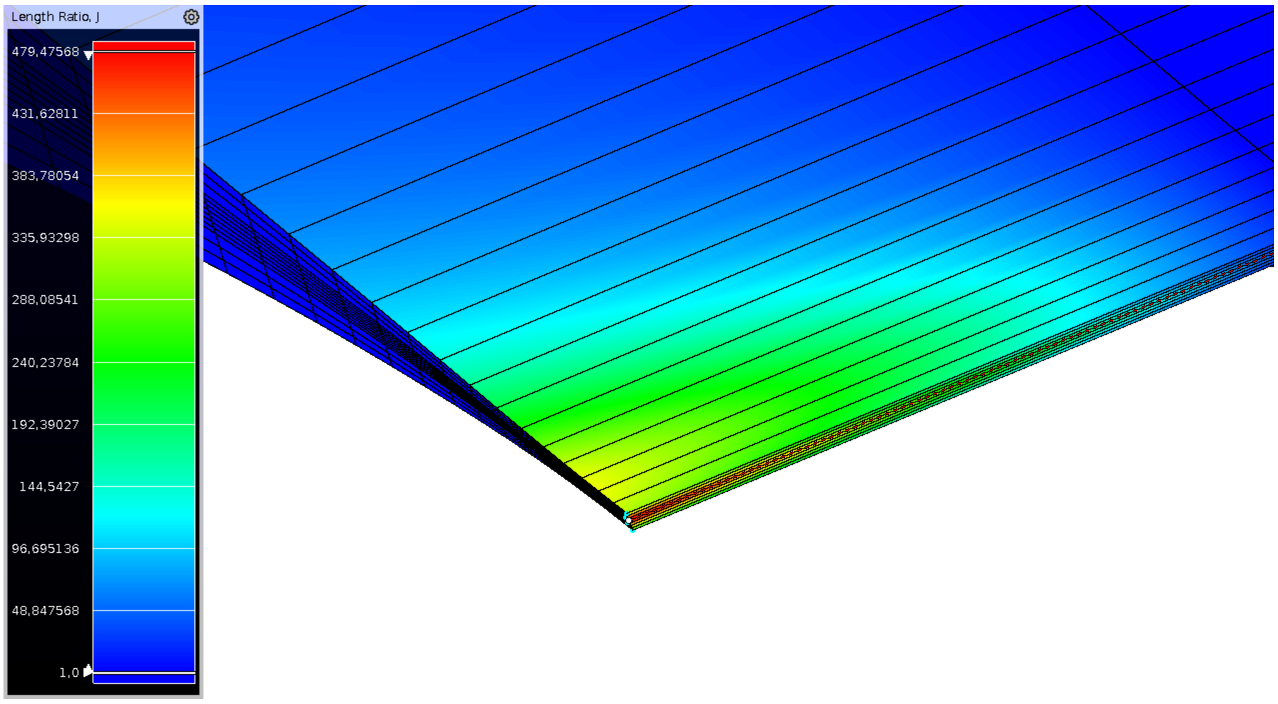


Figure 3.26: length ratio j

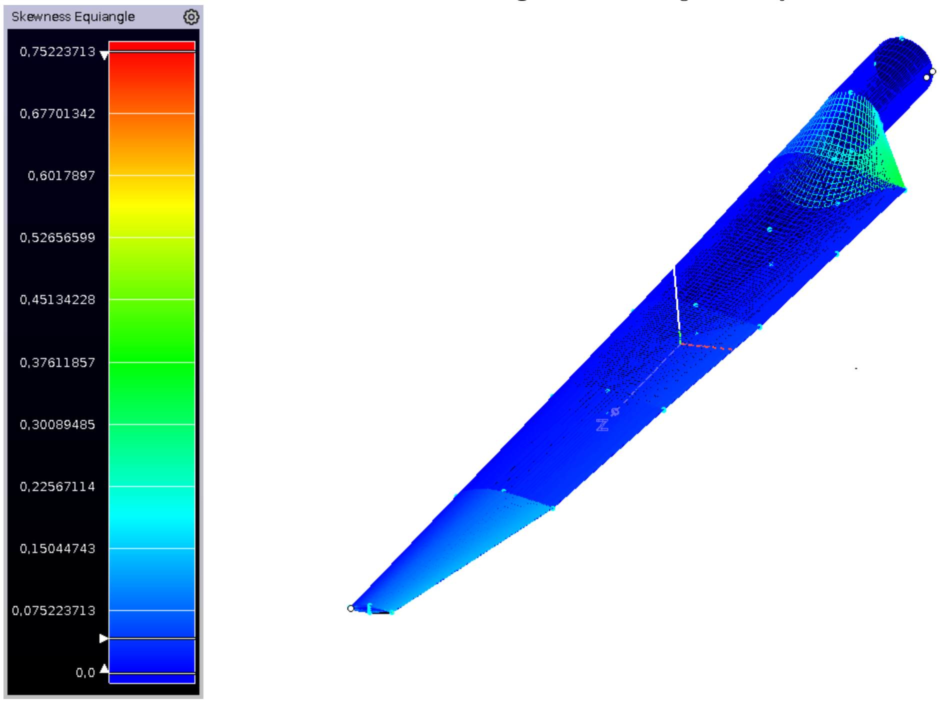
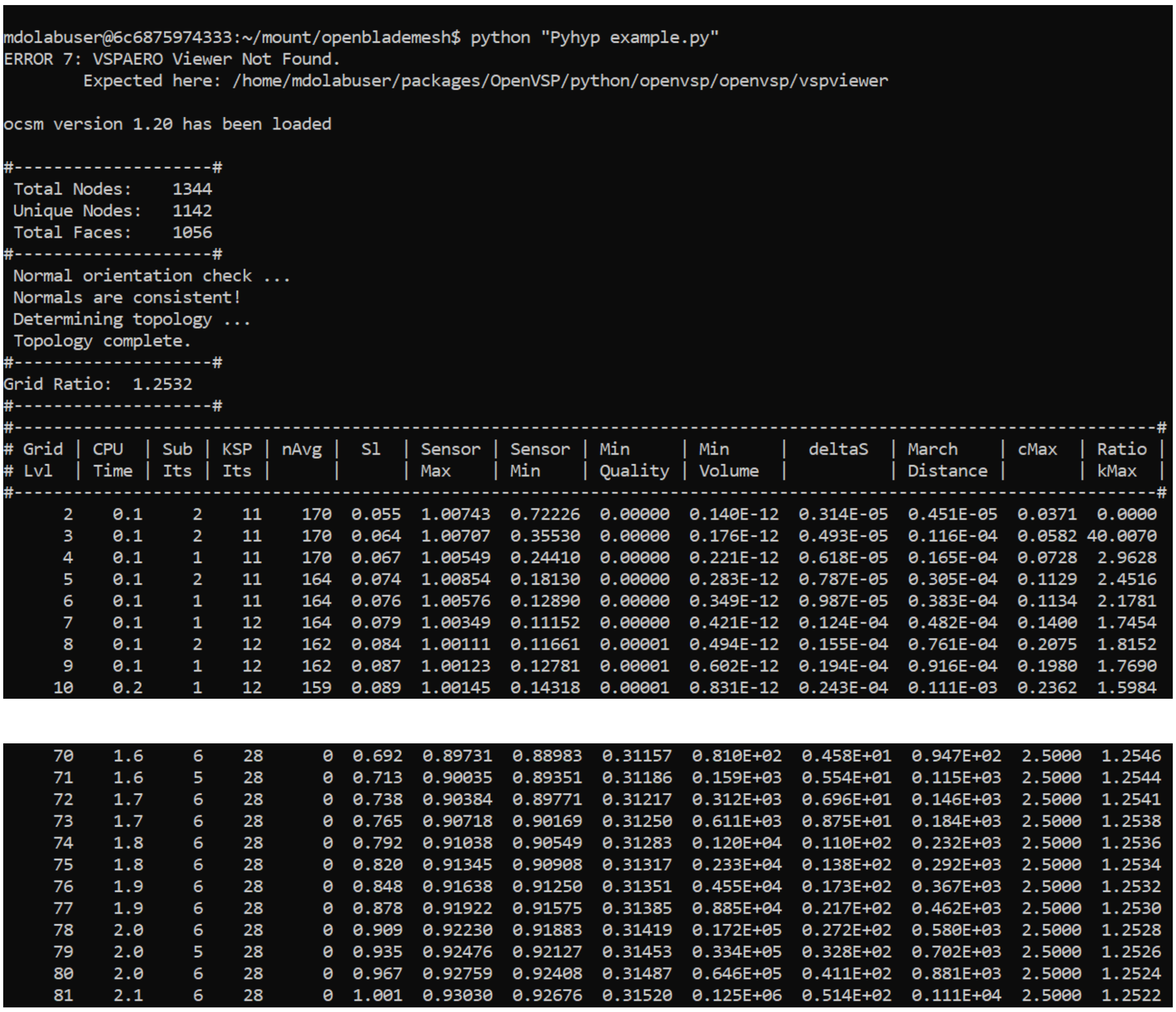


Figure 3.27: skewness

Figure 3.27 show the skewness and Figure 3.24 the edge length of the blade. Both diagnostic values are acceptable and only on the rotor root critical values are found for the edge length because of the coarse node distribution over the wing and the bid step towards the wing root. This is due to the immense difference in mesh size from the airfoils to the circle of the rotor root.

This would be tackled by the grid size reduction algorithm Equation 3.5 for this would create a smoother transition at the wingtip while still being able to achieve a wider grid at the root to enable openblademesh to generate a smoother transition. Figure 3.25 and Figure 3.26 visualize the length ratio at its most critial areas which is in both cases the transition from the wingtip to the wing. This would be bypassed aswell with a smoother transition on the wing in y direction.

Now the code would extrude the wing in the y direction. However, because of the implementation problems with the grid size reduction algorithm Equation 3.5, Pyhyp is not able to create a mesh from this file. So in Figure 3.28 is only the wingtip extruded with Pyhyp, with the Pyhyp output of Figure 3.28.



```
mdolabuser@6c6875974333:~/mount/openblademesh$ python "Pyhyp example.py"
ERROR 7: VSPAERO Viewer Not Found.
        Expected here: /home/mdolabuser/packages/OpenVSP/python/openvsp/openvsp/vspviewer

ocsm version 1.20 has been loaded

#--------------------#
 Total Nodes:    1344
 Unique Nodes:   1142
 Total Faces:    1056
#--------------------#
 Normal orientation check ...
 Normals are consistent!
 Determining topology ...
 Topology complete.
#--------------------#
Grid Ratio:  1.2532
#--------------------#
#-----------------------------------------------------------------------------------------------------------#
# Grid | CPU  | Sub | KSP | nAvg |   Sl   | Sensor | Sensor | Min     | Min       |   deltaS  | March    | cMax   | Ratio |
# Lvl  | Time | Its | Its |      |        | Max    | Min    | Quality | Volume    |           | Distance |        | kMax  |
#-----------------------------------------------------------------------------------------------------------#
      2    0.1     2    11    170  0.055  1.00743  0.72226  0.00000  0.140E-12  0.314E-05  0.451E-05  0.0371  0.0000
      3    0.1     2    11    170  0.064  1.00707  0.35530  0.00000  0.176E-12  0.493E-05  0.116E-04  0.0582 40.0070
      4    0.1     1    11    170  0.067  1.00549  0.24410  0.00000  0.221E-12  0.618E-05  0.165E-04  0.0728  2.9628
      5    0.1     2    11    164  0.074  1.00854  0.18130  0.00000  0.283E-12  0.787E-05  0.305E-04  0.1129  2.4516
      6    0.1     1    11    164  0.076  1.00576  0.12890  0.00000  0.349E-12  0.987E-05  0.383E-04  0.1134  2.1781
      7    0.1     1    12    164  0.079  1.00349  0.11152  0.00000  0.421E-12  0.124E-04  0.482E-04  0.1400  1.7454
      8    0.1     2    12    162  0.084  1.00111  0.11661  0.00001  0.494E-12  0.155E-04  0.761E-04  0.2075  1.8152
      9    0.1     1    12    162  0.087  1.00123  0.12781  0.00001  0.602E-12  0.194E-04  0.916E-04  0.1980  1.7690
     10    0.2     1    12    159  0.089  1.00145  0.14318  0.00001  0.831E-12  0.243E-04  0.111E-03  0.2362  1.5984
```

```
     70    1.6     6    28      0  0.692  0.89731  0.88983  0.31157  0.810E+02  0.458E+01  0.947E+02  2.5000  1.2546
     71    1.6     5    28      0  0.713  0.90035  0.89351  0.31186  0.159E+03  0.554E+01  0.115E+03  2.5000  1.2544
     72    1.7     6    28      0  0.738  0.90384  0.89771  0.31217  0.312E+03  0.696E+01  0.146E+03  2.5000  1.2541
     73    1.7     6    28      0  0.765  0.90718  0.90169  0.31250  0.611E+03  0.875E+01  0.184E+03  2.5000  1.2538
     74    1.8     6    28      0  0.792  0.91038  0.90549  0.31283  0.120E+04  0.110E+02  0.232E+03  2.5000  1.2536
     75    1.8     6    28      0  0.820  0.91345  0.90908  0.31317  0.233E+04  0.138E+02  0.292E+03  2.5000  1.2534
     76    1.9     6    28      0  0.848  0.91638  0.91250  0.31351  0.455E+04  0.173E+02  0.367E+03  2.5000  1.2532
     77    1.9     6    28      0  0.878  0.91922  0.91575  0.31385  0.885E+04  0.217E+02  0.462E+03  2.5000  1.2530
     78    2.0     6    28      0  0.909  0.92230  0.91883  0.31419  0.172E+05  0.272E+02  0.580E+03  2.5000  1.2528
     79    2.0     5    28      0  0.935  0.92476  0.92127  0.31453  0.334E+05  0.328E+02  0.702E+03  2.5000  1.2526
     80    2.0     6    28      0  0.967  0.92759  0.92408  0.31487  0.646E+05  0.411E+02  0.881E+03  2.5000  1.2524
     81    2.1     6    28      0  1.001  0.93030  0.92676  0.31520  0.125E+06  0.514E+02  0.111E+04  2.5000  1.2522
```

Figure 3.28: NREL 5MW Grid

For a better understanding, the whole blade was extruded in Figure 3.29 by pointwise which is a commercial mesh generation software.

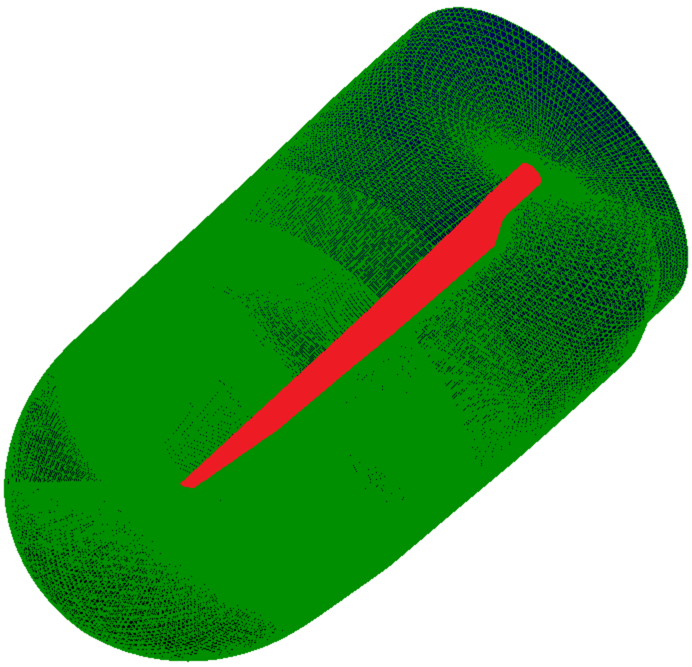


Figure 3.29: NREL 5MW Grid extruded by pointwise

4 Summary

4.1 Conclusion

The main goal of this thesis is to demonstrate the feasibility of the task to create a grid generation tool solely on open source software which is called openblademesh. To achieve this goal, as a basis, Gmsh was used as a grid generation tool to be programmed using Python. To extrude the mesh provided by Gmsh, Pyhyp was chosen and implemented. The reasoning behind the choice for Gmsh, as well as python and Pyhyp, was simply due to their easy usability and open-source easy-to-handle nature. To enable Gmsh to create some of the more complicated calculations, scipy and numpy were implemented because of their good documentation as well as their open source distribution.
To achieve a good quality grid, one of the first things that had to be managed was how the wingtip generation was handled due to its nature of needing a high-quality grid as well as having big and small cells. To this end, inside the wingtip, a second airfoil copying the form of the first airfoil was installed, and therefore cells with quadrangular outlines were enabled due to the creation of quadrangular planes. Furthermore, the distribution of the nodes inside the wingtip with progressions was implemented to allow for a fine grid towards the nose and the trailing edge as well as bigger distances in the middle of the wingtip to save computational time. This feature also extends to the wing and creates a progression over the whole wing.
To create from the wingtip wing, the code was expanded to be able to read several airfoils. By interpolating between adjacent airfoils the wing is created and by applying the transfinite algorithm on each surface a structured grid is created.
For achieving a smooth transition of the grid between the wingtip and the wing a formula was implemented that handled the distribution on the wing. This first formula was not sufficient to achieve the smooth transition needed, so a second more complex calculation was created which was not algebraically solvable, in a reasonable time, and therefore attempted to be solved numerically. This approach was not available at moment of publication and therefore the extrusion with Pyhyp was not enabled as a standalone. While the wingtip on its own can be meshed with Pyhyp the whole wing can only be meshed by software with different extrusion methods such as pointwise.
Always changing the parameters in the code is very tedious they were put in the parameter file so the user can change what is needed to be change in a separate file.
How Openblademesh works and with which concepts are described in this thesis, beginning with a broad overview of the fundamentals, continuing with the code structure and ending with examples of how openblademesh can be used and was used. Finally, suggestions on how the tool can be extended in the future and what the limitations are.

In conclusion, openblademesh is a success in proving that it is possible to create an open-source structured grid-generating software that is partly automated. Openblademesh realizes this and can generate adequate grids and extrude them in its given limitations. One should be able to generate and manipulate the grid generation as well as the extrusion process.

4.2 Limitations and Possible Improvements

In the future, possible supplements would be to implement the feature to give a range between which the cell growth can vary. This would lead to a guaranteed stable extrusion but would give the possibility for the code to run into impossible boundary conditions. For example, when too few cells are combined with a very short grid in the y direction, the step for each cell could exceed the given maximum cell growth. Also, an automated extrusion would be a possible reasonable addition following the example of the MACH-AERO software packages from which already Pyhyp was included, but works with unstructured meshes. In the following, the code limitations and corresponding improvements are proposed.

4.2.1 Wingtips

Because of the way the airfoils are added and then extruded it is not guaranteed that the implementation of winglets or similar wing shapes that are twisted around other axes than the Z-axis succeed. Even though the part of the code dedicated to turning each airfoil is able without big conversions to allow turning around any given axis and around any given point, this feature is not yet tested and may result in unforeseen problems.

4.2.2 Step Size in Y Direction

In the current state, the step size in the Y direction is calculated for each section between two consecutive airfoils separately and therefore does not supply the needed smooth transition between each step. However, a calculation approach is given for points distribution in the Y direction, Equation 3.5 that needs to be implemented in openblademesh using a computationally efficient way.

4.2.3 Growth Rate between Cells

To achieve a stable extrusion of the grid in the normal direction of the cells, it is important to have a smooth growth rate between each cell and not to exceed a factor between 1.3 and 1.6. There is, however, no option to set a maximum value or a range in the current version. Therefore, it is only adjustable passively by changing the amount of nodes and progression. So in a consecutive work, it may be one target to set a maximum cell growth rate to ensure a stable grid extrusion with any set of data.

4.2.4 Non-linear creating of Wing through Airfoils

In Openblademesh, the two-dimensional creation of the wing in the wing direction is only possible in a linear approach. So if a wing with curved outlines is needed, the only solution is to build it with many airfoil sections in a short succession so that the round edge can be reconstructed with linear edges. This might lead to longer computation time. This also could be extended in a complementing work.

4.2.5 Pyhyp

Pyhyp is in its current state purely manual and every change needs to be implemented by hand. Because there is already the automated MACH-AERO framework, [17], it might be possible to implement Openblademesh into it to achieve a more automated code that also generates a structured grid.

References

[1] Nyenah, Emmanuel, Sterl, Sebastian, and Thiery, Wim. “Pieces of a puzzle: solar-wind power synergies on seasonal and diurnal timescales tend to be excellent worldwide”. In: *Environmental Research Communications, Volume 4, Number 5* (2022). DOI: 10.1088/2515-7620/ac71fb.

[2] Boor, Carl de. *A Practical Guide to Spline*. Vol. Volume 27. Jan. 1978. DOI: 10.2307/2006241.

[3] Gordon, William J. and Thiel, Linda C. “Transfinite mappings and their application to grid generation”. In: *Applied Mathematics and Computation* 10-11 (1982), pp. 171–233. ISSN: 0096-3003. DOI: https://doi.org/10.1016/0096-3003(82)90191-6. URL: https://www.sciencedirect.com/science/article/pii/0096300382901916.

[4] William M. Chan, Joseph L. Steger. “Enhancements of a three-dimensional hyperbolic grid generation scheme”. In: *Applied Mathematics and Computation* 51 (1992), pp. 181–205.

[5] Chan, William M. and Steger, Joseph L. “Enhancements of a three-dimensional hyperbolic grid generation scheme”. In: *Applied Mathematics and Computation* 51.2 (1992), pp. 181–205. ISSN: 0096-3003. DOI: 10.1016/0096-3003(92)90073-A. URL: https://www.sciencedirect.com/science/article/pii/009630039290073A.

[6] Parthan, Veena. *Smooth Extrusion for Accurate Viscous Flow Simulation.* URL: https://community.cadence.com/cadence_blogs_8/b/cfd/posts/smooth-extrusion-for-accurate-viscous-flow-simulation (visited on 09/06/2023).

[7] Secco, Ney et al. “Efficient Mesh Generation and Deformation for Aerodynamic Shape Optimization”. In: *AIAA Journal* 59 (Oct. 2020), p. 2020. DOI: 10.2514/1.J059491.

[8] *Y+-Calculation.* URL: https://www.cadence.com/en_US/home/tools/system-analysis/computational-fluid-dynamics/y-plus.html (visited on 06/09/2023).

[9] Lilienthal, Otto. *Der Vogelflug als Grundlage der Fliegekunst.* 1943.

[10] Oliveira, Nícolas, Loureiro, Eric, and Hallak, Patrícia. “STUDY OF MESH REFINEMENT ON THE AERODYNAMIC COEFFICIENTS FOR NACA2412 PROFILE WITH DIFFERENT ANGLE OF ATTACK AND k - w TURBULENCE MODEL”. In: *Revista Mundi Engenharia, Tecnologia e Gestão (ISSN: 2525-4782)* 5 (May 2020). DOI: 10.21575/25254782rmetg2020vol5n21141.

[11] URL: https://code7700.com/index.htm (visited on 06/09/2023).

[12] URL: https://numpy.org/ (visited on 06/09/2023).

[13] URL: https://scipy.org/ (visited on 06/09/2023).

[14] Geuzaine, C. and Remacle., J.-F. *Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities.* 2009.

[15] URL: https://mdolab-pyhyp.readthedocs-hosted.com/en/latest/ (visited on 06/09/2023).

[16] al., Ney R. Secco et. URL: https://mdolab-pyhyp.readthedocs-hosted.com/en/latest/ (visited on 06/09/2023).

[17] URL: `https://mdolab-mach-aero.readthedocs-hosted.com/en/latest/` (visited on 06/09/2023).

[18] Feliciano, J et al. “Generalized Analytical Displacement Model for Wind Turbine Towers under Aerodynamic Loading”. In: *Journal of Wind Engineering and Industrial Aerodynamics* 176 (Mar. 2018). DOI: `10.1016/j.jweia.2018.03.018`.

[19] Jonkman, J et al. “Definition of a 5-MW Reference Wind Turbine for Offshore System Development”. In: (Feb. 2009). DOI: `10.2172/947422`. URL: `https://www.osti.gov/biblio/947422`.

Appendix

Parameter file

```
# define translation of wing sections as
# delta[which airfoi][delta x, delta y, delta z]
delta = 0 0 0
delta = 0 -8.25 0
delta = 0 -15.68 0
delta = 0 -21.98 0
delta = 0 -27.47 0
delta = 0 -32.32 0
delta = 0 -36.49 0
delta = 0 -40.49 0


# define rotation of wing sections as
# angle[which airfoil][angle of rotation in radian, counterclockwise]
angle = 0
angle = 0
angle = 0
angle = 0
angle = 0
angle = 0
angle = 0
angle = 0

# define point around which airfoil is rotated
rx = -1
rx = -1
rx = -1
rx = -1
rx = -1
rx = -1
rx = -1
rx = -1
```

```

ry = 0
ry = 0
ry = 0
ry = 0
ry = 0
ry = 0
ry = 0
ry = 0

rz = 0
rz = 0
rz = 0
rz = 0
rz = 0
rz = 0
rz = 0
rz = 0

# define sizing factor by which airfoil is adjusted
sf = 1.419
sf = 3.256
sf = 3.748
sf = 4.249
sf = 4.458
sf = 4.652
sf = 2.0
sf = 2.0

n_nodes_horizontal = 40
Progression1 = 0.9
Progression2 = 0.9
```

```

n_nodes_vertical_wingtip = 150
n_nodes_vertical_root = 10
```

Python Code from Openblademesh

```python
#Datafile to Gmsh airfoil converter, with structured result and possibilty
#to extrude wing in third dimension.
#V1.1: Datafile needs to be in Selig format and without closed trailing edge

# start of section 1:
# setting the code up
#-------------------------------------------------------------------------------------------------------------
# Import modules:
import gmsh
import sys
import math
import os
import numpy as np
from scipy.interpolate import CubicSpline
from scipy.optimize import fmin
from scipy.optimize import minimize
#from pyhyp import pyHyp

# Initialize gmsh:
gmsh.initialize()
gmsh.model.add("Wing")
gmsh.option.setNumber('Mesh.CgnsExportStructured', 1)

# Parameter to restrict what the code creates do safe time if only specific parts shell be revised
# only_wing_tip_generation disables the creation of the wing through the airfoil greater then 0
# create_GUI enables that gmsh prints what it creates after it finishes. If only the extruded Pyhyp file is needed,
# this can bedisabled to safe time
# extrude_wing enables Pyhyp. If only the Gmsh part is looked upon, pyhyp can be disabled to safe time and compational time
only_wing_tip_generation = False
create_GUI = True
extrude_wing = False

# Scan how many files are in "Data" folder
files = 0
for _, dirnames, filenames in os.walk(r"../data/NREL 5MW/"):
    files += len(filenames)

# read and safe parameters from parameter file
Variables = {
    "delta" : [],
    "angle" : [],
    "rx" : [],
    "ry" : [],
    "rz" : [],
    "sf" : [],
    "n_nodes_horizontal" : "",
    "n_nodes_vertical_wingtip" : "",
    "n_nodes_vertical_root" : "",
    #"Progression1" : "",
    #"Progression2" : ""
    }

# reading Variables from parameter file to variables in code in dictionary
for x in Variables:
    Platzhalter = []
    # string to search in file
    word = str(x) + " = "
    with open(r"../data/Parameter file.txt", "r") as fp:
        # read all lines in a list
        lines = fp.readlines()
        for line in lines:
            # check if string present on a current line
            if line.find(word) != -1:
                line = line.replace(str(x) + " = ", "").split()
                Platzhalter = []
                for entry in line:
                    if type(Variables[x]) == list:
                        entry = float(entry)
                        Platzhalter.append(entry)
                    else:
                        Platzhalter = int(entry)
                if type(Variables[x]) == list:
                    Variables[x].append(Platzhalter)
                else:
                    Variables[x] = int(Platzhalter)

# Start of section 2: loading files
#--------------------------------------------------------------------------
---
# here begins gmsh to work

airfoil_files = []
airfoil_list = []
file_counter = 00
points = []
while file_counter < files:

# get points from txt file and convert to List with strings
# path to load the airfoil file
    with open(r"../data/NREL 5MW/Airfoil"
              + str(file_counter) + ".txt", "r") as f:
        airfoil_file = f.read()
        airfoil_file = airfoil_file.replace(",", ".")
        airfoil_list.append(airfoil_file.replace("\n", " ").split())

# checking to see if List entry is convertible to float
    def is_float(coordinates, file_counter):
        if coordinates is None:
            return False
        try:
            float(coordinates)
            return True
        except:
            print(coordinates + " in file: " + str(file_counter) + " is not
a valid entry, please enter numbers")
            sys.exit(0)
            return False

    for coordinates in airfoil_list[file_counter]:
        is_float(coordinates, file_counter)

# transform points from List as gmsh points
# safe Points temporarely in airfoil_points
# and generate List of points for spline sorting
# containing lists of Coordinates of airfoils
# also: translate points by earlier defined delta
    lc = 0.1
    po = 0
    n = 1
    airfoil_points = []

    # Counts from 0 to end of length of first airfoil and then continues after
    # the lenght of two times the first airfoil to give place for the wingtip
    # generation and six extra points for the inner airfoil generation
    def point_number(counter, n):
        x = 0
        c = len(airfoil_list[0])/2
        if counter < 1:
            return n
        else:
            while x <= counter:
                 c = c + len(airfoil_list[counter -1])/2 + 6
                 x += 1
            return c + n
    # points get safed in airfoil_points and translated by delta from parameter file and adjusted with sf (sizing factor)
    while n <= len(airfoil_list[file_counter])/2:
        airfoil_points.append([
            float(airfoil_list[file_counter][po]) * float(Variables["sf"][file_counter][0]) - Variables["delta"][file_counter][0],
            float(airfoil_list[file_counter][po+1]) * float(Variables["sf"][file_counter][0]) - Variables["delta"][file_counter][2],
            0  * int(Variables["sf"][file_counter][0]) + Variables["delta"][file_counter][1]])
        gmsh.model.occ.add_point(
            float(airfoil_list[file_counter][po]) * float(Variables["sf"][file_counter][0]) - Variables["delta"][file_counter][0],
            float(airfoil_list[file_counter][po+1]) * float(Variables["sf"][file_counter][0]) - Variables["delta"][file_counter][2],
            0 * float(Variables["sf"][file_counter][0]) + Variables["delta"][file_counter][1],
            lc, int(point_number(file_counter, n)))
        n += 1
        po += 2

    points.append(airfoil_points)


    if file_counter > 0:
    # define subsections on airfoil[>0] to enable transfinite algorythm with s0x following the airfoil.
    # In counterclockwise direction, from TE (trailing edge) to the upper middle of the outer airfoil, then to the nose,
    # then around the nose, then to the lower middle and finally back to the TE
        s01 = []
        s02 = []
        s03 = []
        s04 = []
        s05 = []

        n = 0

        while float(points[file_counter][n][int(0)]) >= float(points[file_counter][0][0])/2:
            s01.append(point_number(file_counter, n+1))
            n += 1
        while float(points[file_counter][n][int(0)]) >=
float(1.4142*0.003*float(Variables["sf"][0][0])*0.5):
            s02.append(point_number(file_counter, n))
            n += 1
        s02.append(point_number(file_counter, n))
        X_Wert = n-1
        n += 1

        s03.append(point_number(file_counter, n-1))
        s03.append(point_number(file_counter, n))
        while float(points[file_counter][n][int(1)]) >=
float(points[file_counter][X_Wert][int(1)]):
            s03.append(point_number(file_counter, n+1))
            n += 1

        s04.append(point_number(file_counter, n))
        while float(points[file_counter][n][int(0)]) < (1 *
float(Variables["sf"][file_counter][0]) -
Variables["delta"][file_counter][0])/2:
            s04.append(point_number(file_counter, n+1))
            n += 1

        s05.append(point_number(file_counter, n))
        while float(points[file_counter][n][int(0)]) < 1 *
float(Variables["sf"][file_counter][0]) -
Variables["delta"][file_counter][0]:
            s05.append(point_number(file_counter, n+1))
            n += 1
        s05.append(point_number(file_counter, n+1))
        n = 1

        s02.reverse()
        s05.reverse()

    # rotate points from subsection on pivot point rx/ry/rz
        n = 1
        while point_number(file_counter, n) <= point_number(file_counter,
(len(airfoil_list[file_counter])/2-1)):
            gmsh.model.occ.rotate([(0, point_number(file_counter, n))],
                                  -float(Variables["rx"][file_counter][0] *
-Variables["sf"][file_counter][0]),
                                  -float(Variables["rz"][file_counter][0] *
-Variables["sf"][file_counter][0]),
                                  float(Variables["ry"][file_counter][0] *
-Variables["sf"][file_counter][0]),
                                  0, 0, 1,

float(Variables["angle"][file_counter][0]))
            n += 1

        # creating outer line for airfoils > 0 from earlier defined s0x
        gmsh.model.occ.add_spline(s01, 10001 + file_counter * 10000)
        gmsh.model.occ.add_spline(s02, 10002 + file_counter * 10000)
        gmsh.model.occ.add_spline(s03, 10003 + file_counter * 10000)
        gmsh.model.occ.add_spline(s04, 10004 + file_counter * 10000)
        gmsh.model.occ.add_spline(s05, 10005 + file_counter * 10000)
        # create line to connect TE
        gmsh.model.occ.add_line(int(max(s05)), int(min(s01)), 10000 +
file_counter * 10000)

        gmsh.model.occ.addCurveLoop([10000 + file_counter * 10000,
                                     10001 + file_counter * 10000,
                                     10002 + file_counter * 10000,
                                     10003 + file_counter * 10000,
                                     10004 + file_counter * 10000,
                                     10005 + file_counter * 10000],
                                    10006 + file_counter * 10000)
        gmsh.model.occ.synchronize()

    file_counter += 1

# Start of Section3:
#---------------------------------------------------------------------------
----

# Here starts Wingtip generation!

# copy and dilate original points[0]
# to generate a copy for inner domain generation
# with X defining the factor of the dilation and A being the Y coordinate
# of the upper airfoil and B the Y coordinate of the lower airfoil

X = 0.33

i = 0
k = 1+len(points[0])
points_inner = []
x_points = []
y1_points = []
y2_points = []
y3_points = []
y4_points = []
y_points = []

while i < len(airfoil_list[0]):
    n = 0
    while n < len(airfoil_list[0]):
        X1 = X
        if i == n:
            n = n + 2
        if float(airfoil_list[0][i]) == float(airfoil_list[0][n]):
            # if loop is for the part where the points are on the positive
Y value
            if float(airfoil_list[0][i+1]) > float(airfoil_list[0][n+1]):
                # here with A, B, C and E is the formula to calculate the
position of the inner airfoil
                A = float(airfoil_list[0][i+1])
                B = float(airfoil_list[0][n+1])
                C = A - B
                E = A - X1 * C
                # also y points needs to be adjusted by sf and delta to fit
into place of resulting airfoil
                y1_points.append(E * float(Variables["sf"][0][0]) -
Variables["delta"][0][2])
                y3_points.append(float(airfoil_list[0][i+1])*
float(Variables["sf"][0][0]) - Variables["delta"][0][2])

                points_inner.append([float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                     - Variables["delta"][0][0],E *
float(Variables["sf"][0][0]) - Variables["delta"][0][2]])
                gmsh.model.occ.add_point(float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                     - Variables["delta"][0][0],E *
float(Variables["sf"][0][0]) - Variables["delta"][0][2],
                    0, lc, k)

                break
            # the else section for the y section after passing through the
origin
            else:
                A = float(airfoil_list[0][n+1])
                B = float(airfoil_list[0][i+1])
                C = A - B
                D = B + X1 * C

                x_points.append(float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                        - Variables["delta"][0][0])
                y2_points.append(D * float(Variables["sf"][0][0]) -
Variables["delta"][0][2])
                y4_points.append(float(airfoil_list[0][i+1])*
float(Variables["sf"][0][0]) - Variables["delta"][0][2])

                points_inner.append([float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                        - Variables["delta"][0][0],D *
float(Variables["sf"][0][0]) - Variables["delta"][0][2]])
                gmsh.model.occ.add_point(float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                        - Variables["delta"][0][0],D *
float(Variables["sf"][0][0]) - Variables["delta"][0][2],
                    0, lc, k)


                break
        # lastly the airfoil tip, it is seperately added
        else:
            if float(airfoil_list[0][i]) == 0:
                    points_inner.append([float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                          -
Variables["delta"][0][0],float(airfoil_list[0][i+1]) *
float(Variables["sf"][0][0]) - Variables["delta"][0][2]])
                    gmsh.model.occ.add_point(float(airfoil_list[0][i]) *
float(Variables["sf"][0][0])
                                          -
Variables["delta"][0][0],float(airfoil_list[0][i+1]) *
float(Variables["sf"][0][0]) - Variables["delta"][0][2],
                        0, lc, k)
                    break
            n += 2
    k += 1
    i += 2


# adding points for trailing edge domain generation so that the ...
# ... TE domain boundaries are 45 degrees
# generation lists and points for the spline generation to be ay easy as
possible

y_points.append(y1_points)
y_points.append(y2_points)
y_points.append(y3_points)
y_points.append(y4_points)

x_points.insert(0,0 - Variables["delta"][0][0])
y_points[0].append(0 - Variables["delta"][0][2])
```

```
y_points[1].insert(0,0 - Variables["delta"][0][2])
y_points[2].append(0 - Variables["delta"][0][2])
y_points[3].insert(0,0 - Variables["delta"][0][2])
y_points[0].reverse()
y_points[2].reverse()
x_points = np.array(x_points)
y_points[0] = np.array(y_points[0])
y_points[1] = np.array(y_points[1])
y_points[2] = np.array(y_points[2])
y_points[3] = np.array(y_points[3])

# create spline for interpolatin of beginning and ending of the inner and
outer airfoil between the nose
# and the trailing section

cs = []
n = 0
while n < 4:
   cs.append(CubicSpline(x_points,y_points[n],bc_type='natural'))
   n += 1

Inter_X = []

# adding the X values from which the spline start and end to the list
Inter_X.append(float(points_inner[0][0]) - 1/4 * (float(points[0][0][1]) -
float(points[0][-1][1])))
Inter_X.append(float(points_inner[-1][0]) - 1/4 * (float(points[0][0][1]) -
float(points[0][-1][1])))
Inter_X.append(float(1.4142*0.003*float(Variables["sf"][0][0])))
Inter_X.append(float(0.5 * Inter_X[2]))

n = 0
i = 0
k = 0
# and adding points to the model from the created spline
while k < 6:
    if n == 2:
        i = 0
        points_inner.append([Inter_X[n],cs[i](Inter_X[n])])
        gmsh.model.occ.add_point(Inter_X[n],cs[i](Inter_X[n]),
                                 0, lc, len(airfoil_list[0])+k+1)
        k += 1
        i = 1

    if n == 3:
      points_inner.append([Inter_X[n],cs[i](Inter_X[n])])
      gmsh.model.occ.add_point(Inter_X[n],cs[i](Inter_X[n]),
                               0, lc, len(airfoil_list[0])+k+1)
      k += 1
      i += 1

    points_inner.append([Inter_X[n],cs[i](Inter_X[n])])
    gmsh.model.occ.add_point(Inter_X[n],cs[i](Inter_X[n]),
                             0, lc, len(airfoil_list[0])+k+1)
    n += 1
    i += 1
    k += 1

P = []
X_value = []
Y_value = []
n = 0

```

```
while n < 6:
    P.append(len(airfoil_list[0])/2+n)
    X_value.append(points_inner[int(P[n])][int(0)])
    Y_value.append(points_inner[int(P[n])][int(1)])
    n += 1


# generate Spline from points[0] for Airfoil domain with
# s5 beginning on the outer airfoil
# s1 to the nose section
# s15 connecting around the nose
# s4 from the lower nose section
# s8 from there to the trailing edge on the outer airfoil
# Then continuing for the inner airfoil with
# s6 and s2 for the upper half and then s3 and s7 for the lower half.
s1 = []
s2 = []
s3 = []
s4 = []
s5 = []
s6 = []
s7 = []
s8 = []
s15 = []
s = []
n = 0
i = 0

while float(points[0][n][int(0)]) >= float(points[0][0][0])/2:
    s5.append(n+1)
    n += 1
s5.append(n+1)
half_upper_top = n+1

while float(points[0][n][int(0)]) >= X_value[4]:
    s1.append(n+1)
    n += 1
s1.append(len(airfoil_list[0])+5)

s15.append(len(airfoil_list[0])+5)
while float(points[0][n][int(1)]) >= Y_value[5]:
    s15.append(n+1)
    n += 1
s15.append(len(airfoil_list[0])+6)

s4.append(len(airfoil_list[0])+6)
while float(points[0][n][int(0)]) < (1 * float(Variables["sf"][0][0]) -
Variables["delta"][0][0])/2:
    s4.append(n+1)
    n += 1
half_lower_low = n

s8.append(n)
while float(points[0][n][int(0)]) < 1 * float(Variables["sf"][0][0]) -
Variables["delta"][0][0]:
    s8.append(n+1)
    n += 1
s8.append(n+1)
n = 1

s6.append(len(airfoil_list[0])+1)
while float(points_inner[n][int(0)]) >= float(points_inner[0][0]/2):
    s6.append(n + len(points[0])+1)
    n += 1
s6.append(n + len(points[0])+1)
half_upper_low = (n + len(points[0])+1)

while float(points_inner[n][int(0)]) >= X_value[2]:
    s2.append(n + len(points[0])+1)
    n += 1
s2.append(len(airfoil_list[0])+3)

while  float(points_inner[n][int(1)]) >= Y_value[3]:
    n += 1

s3.append(len(airfoil_list[0])+4)
while float(points_inner[n][int(0)]) < float(points_inner[0][0]/2):
    s3.append(n + len(points[0])+1)
    n += 1
half_lower_top = n-1 + len(points[0])+1

s7.append(n + len(points[0]))
while float(points_inner[n][int(0)]) <= X_value[1]:
    s7.append(n + len(points[0])+1)
    n += 1
s7.append(len(airfoil_list[0])+2)

# to enable the correct arrangement of the normals, the first spline of
each plan needs to be in the correkt order
# so because the progression script is both ways with the same factor (for
convinience reasons for the user) some splines
# needs to be reversed
s1.reverse()
s2.reverse()
s7.reverse()
s8.reverse()

s.append(s1)
s.append(s2)
s.append(s3)
s.append(s4)
s.append(s5)
s.append(s6)
s.append(s7)
s.append(s8)
s.append(s15)

n = 0
while n < 8:
    gmsh.model.occ.add_spline(s[n], n + 1)
    n += 1
gmsh.model.occ.add_spline(s15, 15)


#generate lines for domain boundaries
gmsh.model.occ.add_line(
    int(len(airfoil_list[0])+3),
    int(len(airfoil_list[0])+5), 9)

gmsh.model.occ.add_line(
    int(len(airfoil_list[0])+1),
    int(points[0].index(max(points[0][:1]))+1), 12)

gmsh.model.occ.add_line(
    int(len(airfoil_list[0])+4),
    int(len(airfoil_list[0])+6), 10)
```

```
gmsh.model.occ.add_line(
    int(len(airfoil_list[0])+2),
    int(points_inner.index(points_inner[-1])-5), 11)

gmsh.model.occ.add_line(
    int(len(airfoil_list[0])+1),
    int(len(airfoil_list[0])+2), 18)

gmsh.model.occ.add_line(
    int(points_inner.index(points_inner[-1])-5),
    int(points[0].index(max(points[0][:1]))+1), 19)

gmsh.model.occ.add_line(
    int(len(airfoil_list[0])+3),
    int(len(airfoil_list[0])+4), 16)

gmsh.model.occ.add_line(
    int(half_upper_low),
    int(half_upper_top), 13)

gmsh.model.occ.add_line(
    int(half_lower_top),
    int(half_lower_low), 14)
gmsh.model.occ.synchronize()

gmsh.model.occ.add_line(
    int(half_upper_low),
    int(half_lower_top), 17)
gmsh.model.occ.synchronize()

# generate curved loops for domains 2, 3, 4, 5, 6, 7, 8, 9 and 10006 which
indicates that this is a curve loop
# that is used for the creation in y direction
gmsh.model.occ.add_curve_loop([19, 5, 1, 15, 4, 8], 10006)
gmsh.model.occ.add_curve_loop([12, 5, 13, 6], 2)
gmsh.model.occ.add_curve_loop([13, 1, 9, 2], 3)
gmsh.model.occ.add_curve_loop([9, 15, -10, -16], 4)
gmsh.model.occ.add_curve_loop([10, -4, -14, -3], 5)
gmsh.model.occ.add_curve_loop([-14, 8, -11, -7], 6)
gmsh.model.occ.add_curve_loop([18, 11, 19, 12], 7)
gmsh.model.occ.add_curve_loop([16, 3, 17, 2], 8)
gmsh.model.occ.add_curve_loop([17, 7, 18, 6], 9)

# add surfaces on wing_tip for structured mesh generation
n = 2
while n < 10 :
    gmsh.model.occ.add_plane_surface([n], n)
    n += 1

# progression for the distribution of points around the airfoil.
progression1 = 1.1
progression2 = 1.1


n_nodes_nose = int(((cs[0](Inter_X[2])-cs[1](Inter_X[2]))*
                (progression1**(Variables["n_nodes_horizontal"]-1)-1))/(
                    (float(points[0][s[0][-1]][0])-
float(Inter_X[3]))*(progression1-1)))

n_nodes_vertical_airfoil = int(((cs[2](Inter_X[3])-cs[0](Inter_X[2]))*
                (progression1**(Variables["n_nodes_horizontal"]-1)-1))/(
                    (float(points[0][s[0][-1]][0])-
float(Inter_X[3]))*(progression1-1)))

# creating a structured grid through a transfinite approach

n = 1
while n <= 19 :
    if n < 3 :
        gmsh.model.mesh.setTransfiniteCurve(
            n, Variables["n_nodes_horizontal"], meshType="Progression",
coef = progression2)
    elif n < 5 :
        gmsh.model.mesh.setTransfiniteCurve(
            n, Variables["n_nodes_horizontal"], meshType="Progression",
coef = progression2)
    elif n < 7:
        gmsh.model.mesh.setTransfiniteCurve(
            n, Variables["n_nodes_horizontal"], meshType="Progression",
coef = progression1)
    elif n < 9:
        gmsh.model.mesh.setTransfiniteCurve(
            n, Variables["n_nodes_horizontal"], meshType="Progression",
coef = progression1)
    elif n < 15:
        gmsh.model.mesh.setTransfiniteCurve(
            n, n_nodes_vertical_airfoil)
    elif n < 20:
        gmsh.model.mesh.setTransfiniteCurve(
            n, n_nodes_nose)
    n += 1

gmsh.model.occ.synchronize()

n = 2
while n < 10 :
    gmsh.model.mesh.setTransfiniteSurface(n)
    n += 1

gmsh.model.occ.synchronize()

# start of section4:
# start of wing generation in Y direction
#---------------------------------------------------------------------------
----
# end of wing tip generation

if only_wing_tip_generation == False:

    n = 0
    while n < files -1:
    # Creating 3d model of wing through interpolation between airfoils
        gmsh.model.occ.addThruSections([20006 + n * 10000,
                                        10006 + n * 10000], False, False,
True)
        gmsh.model.occ.synchronize()
        n += 1

    n = 1

    while n < files:
    # Setting Grid on Z-Axis to be structured...
        # ...for the surface edges
        gmsh.model.mesh.setTransfiniteCurve(10001 + n * 10000,
Variables["n_nodes_horizontal"], meshType="Progression", coef =
progression2)
        gmsh.model.mesh.setTransfiniteCurve(10002 + n * 10000,

Variables["n_nodes_horizontal"], meshType="Progression", coef =
progression2)
        gmsh.model.mesh.setTransfiniteCurve(10003 + n * 10000,
                                            n_nodes_nose)
        gmsh.model.mesh.setTransfiniteCurve(10004 + n * 10000,

Variables["n_nodes_horizontal"], meshType="Progression", coef =
progression1)
        gmsh.model.mesh.setTransfiniteCurve(10005 + n * 10000,

Variables["n_nodes_horizontal"], meshType="Progression", coef =
progression1)
        gmsh.model.mesh.setTransfiniteCurve(10000 + n * 10000,
                                            n_nodes_nose)
        n += 1

    #n_nodes_vertical_wingtip =
(abs(Variables["delta"][1][2])*n_nodes_vertical_airfoil*1.4142)/(2/3*(abs(p
oints[0][0][1]-points[0][-1][1])))
    #print(n_nodes_vertical_wingtip)

# here beginns the calculation of the distribution of the nodes in y
direction to achieve a very small space at the wingtip
# that fits the distance of the cells at the TE and continuesly growing
distances to save time
    n = 0
    i = files - 1
    while n < 6 * (files - 1):
        # ...for the edges in wing direction
        if n%6 == 0:
            Variables["n_nodes_vertical_wingtip"] =
Variables["n_nodes_vertical_root"]*math.exp(

(np.log(Variables["n_nodes_vertical_wingtip"]/Variables["n_nodes_vertical_r
oot"]))/(files-1)*i)
            i -= 1
        gmsh.model.mesh.setTransfiniteCurve(10000 * files + 6 + n,
int(Variables["n_nodes_vertical_wingtip"]))
        n += 1


    n = 0
    while n < files - 1:
        n1 = 1
        while n1 < 7:
            # ... for the surfaces
            gmsh.model.mesh.setTransfiniteSurface(9 + n1 + n * 6)
            gmsh.model.occ.synchronize()
            n1 += 1
        n += 1

# Start of section5:
# finishing gmsh
#---------------------------------------------------------------------------
--------
gmsh.model.occ.synchronize()

# smoothing grid to generate a more regular grid
gmsh.option.setNumber("Mesh.Smoothing", 0)
```

```python

# cleaning the grid if some nodes may be duplicate
gmsh.model.occ.synchronize()
gmsh.model.mesh.removeDuplicateNodes()
gmsh.model.occ.removeAllDuplicates()

# Generate a quadrangular grid:
gmsh.model.mesh.generate(2)
gmsh.model.mesh.recombine()

# Write mesh data:
gmsh.write("../output/NREL5MW.p3d")

if create_GUI == True:
    # Creates graphical user interface
    if 'close' not in sys.argv:
        gmsh.fltk.run()

# It finalizes the Gmsh API
gmsh.finalize()

# Start of section6:
# Pyhyp
#--------------------------------------------------------------------------------------------------------------------

# here beginns pyhyp with its parameter and where to search for the input file and how to safe it after its been created
baseDir = os.path.dirname(os.path.abspath(__file__))
surfaceFile = os.path.join(baseDir, "NREL5MW.p3d")
volumeFile = os.path.join(baseDir, "Wing_hyp.cgns")

options = {
    # ---------------------------
    #        Input Parameters
    # ---------------------------
    "inputFile": surfaceFile,
    "fileType": "PLOT3D",
    "unattachedEdgesAreSymmetry": False,
    "outerFaceBC": "farfield",
    "autoConnect": True,
    #"mode": "elliptic",
    #"BC": {1: {"jlow": "zSymm"}},
    # ---------------------------
    #        Grid Parameters
    # ---------------------------
    "N": 3,
    "s0": 1e-3,
    "marchDist": 500.0,
    #"nConstantStart": 1,
    #"coarsen": 2,
    #"nConstantEnd": 1,
    #"nodeTol": 1e-08,
    "splay": 1e-4,
    "splayEdgeOrthogonality": 0.5,
    "splayCornerOrthogonality": 0.5,
    "cornerAngle": 60.0,
    # ---------------------------
    #   Pseudo Grid Parameters
    # ---------------------------
    #"ps0": -1.0,
    #"pGridRatio": -1.0,
    "cMax": 1.0,
    # ---------------------------
    #   Smoothing parameters
    # ---------------------------
    #"epsE": 1.0,    #Explicit smoothing factor
    #"epsI": 2.0,    #Implicit smoothing factor
    #"theta": 3.0,
    #"volCoef": 0.25,
    #"volBlend": 0.0002,
    #"volSmoothIter": 150,

}

#decide if extrusion is done
if extrude_wing == True:
    # rst object
    hyp = pyHyp(options=options)
    # rst run
    hyp.run()
    hyp.writePlot3D(volumeFile)

```