Using URANS CFD to Optimize the Pitching Motion and Path of the Cycloidal Rotor Blades

Master Thesis
by
Dipl.-Ing. (FH) cand. aer. Korbinian Kasper

conducted at the
Institute of Aerodynamics and Gas Dynamics
at the University of Stuttgart

Stuttgart, August 2022

Universität Stuttgart

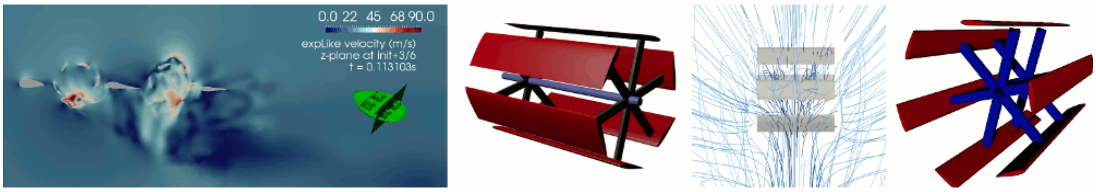
INSTITUT FÜR AERODYNAMIK UND GASDYNAMIK IAG

DIREKTOR: PROF. DR.-ING. EWALD KRÄMER

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

Master Thesis for Korbinian Kasper

Using URANS CFD to Optimize the Pitching Motion and Path of the Cycloidal Rotor Blades



Cycloidal rotors have the advantages of providing 360° thrust forces and having constant flow velocities on their blades. However, they generate and operate in a downflow that makes it impossible to avoid considerable parasitic drag generation while maintaining a circular blade path. An overset mesh method for cyclorotor simulations which allows any motion function was developed at IAG.

The theme of the thesis is thus to **investigate novel blade pitch and motion paths** and then study their effect on rotor **energy efficiency**. The objective is to obtain a better **qualitative evaluation** of the influence of the cyclorotor patch on the **energy efficiency** of these rotors.

Milestones:

- use the current OpenFOAM-based cyclorotor methodology and build an easily adaptable 2D rotor mesh by merging a structured blade to an unstructured cartesian background
- implement a two-dimensional spline-based blade motion control for both path and pitch where spline parameters are adjusted to ensure continuity of the second derivative
- rely on numerical eucharistic algorithms to optimize both path and pitch leading to the maximal figure of merit (FoM)
- evaluate the impact of the number of blades on the optimal path and pitch functions

Date Issued: Januar 25th, 2022

Date Submitted: August 12th, 2022

Student: Korbinian Kasper

Advisor: Louis Gagnon

Examiner 1: Manuel Keßler

Examiner 2: Ewald Krämer

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 contributions of someone else (eg. pictures, drawings, text passages etc.), I have clearly identified the source of these contributions, and, if necessary, 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 these duties.

Date, Signature

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.

Abstract

This master thesis describes the procedure for optimisation of the pitching and the trajectory for cyclorotor blades to increase the efficiency based on 2D CFD calculations. The open-source software OpenFOAM with URANS is used for these CFD analyses. Considering various numbers of blades (one to four), the use of the chimera technique is necessary using the built-in OpenFOAM solver *overPimpleDyMFoam*. B-splines describe the arbitrary pitching and trajectory implemented in separate OpenFOAM motion classes. Two possible modes of drive are investigated for the cycloidal system; 'constant velocity' and 'constant angular velocity'. The Dakota toolkit performs the parametric optimisation with an evolutionary algorithm. A Python script initialises, monitors and evaluates each CFD case.

Fourteen optimisation setups are carried out. An increase in the efficiency for each run is achieved. The main reason for the improvement is the better alignment of the blade forces to the global thrust. Another reason is that the optimised motion induces force peaks, which leads to an increase in thrust. The best result is captured for a four-blade case with a circular motion and a pitching path optimisation. The figure of merit is 0.758. Two further optimisation runs with higher Reynolds numbers are carried out for the two-blade case with a circular motion. Despite the pitching paths' similarity, the figure of merit can be significantly increased (+8.8% for double Reynolds number and +17.7% for fourfold Reynolds number). Due to a false precalculation of the trajectory, the optimisation results for the 'constant angular velocity' drive are invalid.

Abstract

Die vorliegende Masterarbeit beschreibt das Vorgehen zur Optimierung des Anstellwinkels sowie der Bahnbewegung von Rotorblättern von Cyclorotoren in Hinblick auf eine Effizienzsteigerung auf Basis von 2D numerischen Strömungsberechnungen. Für diese CFD Analysen wird das quellenoffene Softwareprogramm OpenFOAM verwendet. Aufgrund der Anzahl der untersuchten Blätter (1 bis 4) wurde die Chimera-Technik eingesetzt, welche durch den OpenFOAM Solver *overPimpleDyMFoam* zur Verfügung steht. Zur mathematischen Beschreibung der Anstellwinkels und der Trajektorie werden B-Splines verwendet, welche in separate OpenFOAM Klassen implementiert werden. Zwei unterschiedliche Antriebe des Cyclorotors werden untersucht; 'konstante Geschwindigkeit' und 'konstante Drehzahl'. Die Parameteroptimierung wird mit der Software Dakota durchgeführt, wobei ein evolutionärer Algorithmus zur Anwendung kommt. Mit Hilfe einens Python Skripts werden die einzelnen CFD Berechnung intialisiert, überwacht und ausgewertet.

Insgesamt werden 14 unterschiedliche Optimierungläufe durchgeführt, wobei für alle eine Steigerung der Effizienz erreicht wird. Der Hauptgrund für die Verbesserung liegt zum einen an der günstigeren Ausrichtung der einzelnen Blattkräfte in Richtung des resultierenden Schubs. Des weiteren führen die optimierten Blattbewegungen zu Kraftspitzen, wodurch der Schub deutlich gesteigert wird. Die größte Steigerung wird bei einem 4-Blatt Rotor mit Kreisbahn und einer Anstellwinkeloptimierung erreicht. Die entsprechend Leistungsgütezahl beträgt 0.758. Für den 2-Blatt Zyklorotor werden zustätzliche Optimierung mit höheren Reynoldszahlen durchgeführt. Trotz der Ähnlichkeit der optimierten Anstellwinkelverläufe zueinander ist eine nennenswerte Verbesserung der Leistungsgüte zu verzeichnen (+8.8% für doppelte Reynoldszahl and +17.7% bei vierfacher Reynoldszahl). Aufgrund einer fehlerhaften Vorabberechnung der Trajektorie sind die Ergebnisse für die Optimierung für den Antrieb 'konstante Drehzahl' nicht verwertbar.

Contents

Nomenclature

Symbol	Unit	Definition
Latin Symbols		
c	m	chord length
$C_{Power,rot}$	-	coefficient for rotation power
$C_{Power,tra}$	-	coefficient for translation power
d_z	m	depth of mesh domain
F	N	force
k_e	$\frac{\mathrm{m}^2}{\mathrm{s}^2}$	turbulence kinetic energy
K	-	element of knot vector
n_T	-	control variable for splines
N		basis function of spline
n_{Blade}	-	number of blades
R	m	radius
Re	-	Reynolds number
T_P	s	period
t_{sim}	s	simulation time
U_∞	$\frac{\mathrm{m}}{\mathrm{s}}$	free stream velocity
v	$\frac{\mathrm{m}}{\mathrm{s}}$	velocity
w_R	m	rotor width
X, Y	m	coordinates of spline
Greek Symbols		
α_0	degree, °	Angle between the tangent of trajectory and mean line of the airfoil
β	degree, °	Angle between the horizontal and the global thrus vector
γ	m	coordinate vector of spline
θ	degree, °	Angle between the tangent of trajectory and the vertical
κ		knot vector
ν	$\frac{\mathrm{m}^2}{\mathrm{s}}$	kinematic viscosity
ρ	$\frac{\mathrm{kg}}{\mathrm{m}^3}$	fluid density
σ	$\frac{1}{\mathrm{m}}$	curvature of trajectory
φ	degree, °	Counter angle, between the mean line of the airfoil and the vertical
Ψ	degree, °	Azimuth angle, in x-y plane
ω_{diss}	$\frac{1}{\mathrm{s}}$	specific turbulence dissipation
ω_{rot}	$\frac{1}{\mathrm{s}}$	angular velocity
Superscript		
^{p}		corresponds to pitching spline
^{t}		corresponds to trajectory spline
$'$		derivative of function
Subscript		
$_{len}$		corresponds to lenght spline
$_{pit}$		corresponds to pitching spline
$_{tra}$		corresponds to trajectory spline

Abbreviations

Symbol	Definition
CFD	Computational Fluid Dynamics
CFL	Courant-Friedrichs-Lewy
CS	Coordinate System
DP	Design Point
EA	Evolutionary Algorithm
FOM	Figure of Merit
FVM	Finite Volume Mesh
IAG	Institute of Aerodynamics and Gas Dynamics
$NACA$	National Authority Comitee for Aerodynamic
$NURBS$	Non-Uniform Rational B-Splines
OF	Objective Function
$PISO$	Pressure-Implicit with Splitting of Operators
$SIMPLE$	Semi-Implicit Method for Pressure Linked Equations
SST	Shear Stress Transport

1 Introduction

In contrast to common helicopters, a cyclorotor produces the thrust by rotating the blades parallel to the rotation axis. To achieve a resulting force acting in one direction, the angle of attack of each blade must be adjusted during the rotation. This is mainly done with a push/pull rod system supported excentric to the rotation axis (see Figure 1.1).

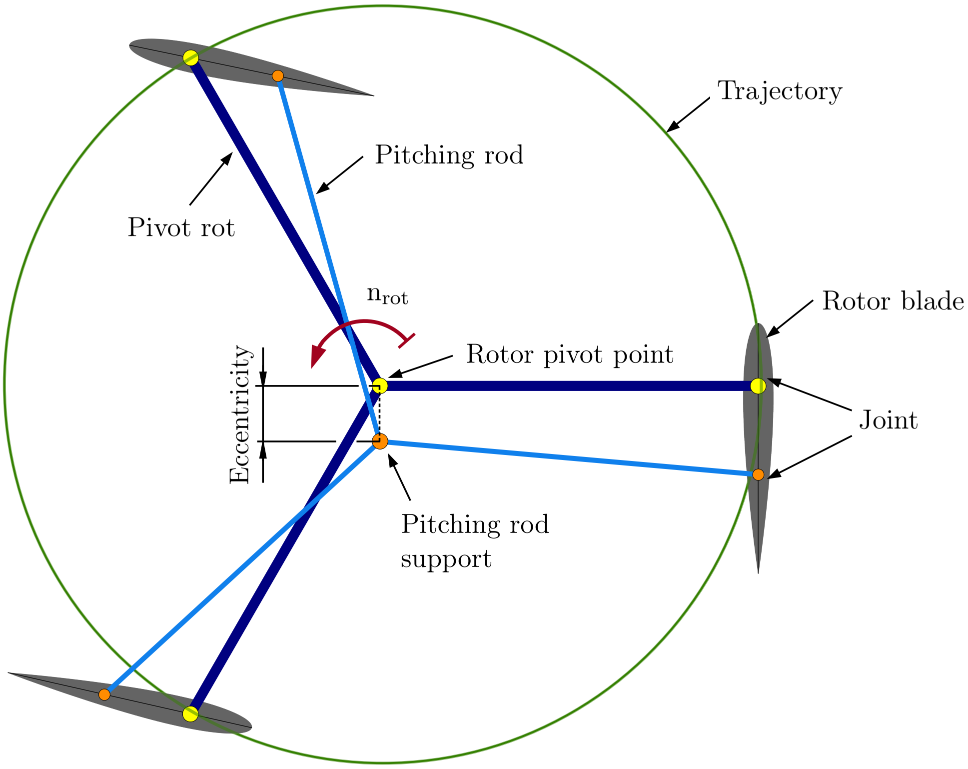


Figure 1.1: Functional principle of a cyclogyro.

A suitable thrust direction is needed for different manoeuvres like ascending and forward flight. By changing the position of the rod support, the pitching progression can be modified and the thrust direction, all without tilting the aircraft, which is one advantage of cyclorgyros over helicopters. Another benefit of the cyclorotor system is that the dynamic pressure remains constant along the whole blade length. This leads to better utilisation of the blade compared to a helicopter blade, where aerodynamic force varies.

However, there are also disadvantages. First and foremost, the vertical movement of the blades mainly produces parasitic drag. This movement occurs twice during a rotation and is an intrinsic property of cyclorotors

In the past, different approaches have been taken to determine the characteristics and behaviour of cyclogyros. So did the National Advisory Committee for Aeronautics (NACA) in the early thirties of the last century. They published two reports dealing with the cyclogyro, where the first document TN 467 [1], contains an analytical attempt to calculate the characteristic parameters like thrust and required power. A prototype of a cyclogyro and the concerned measurements in a wind tunnel are the content of the report TN 528 [2]. A comparison to the former analytical calculation was carried out for different Reynolds numbers and free stream velocities.

Due to technological progress, the computational fluid dynamics and enhanced materials like fibre-reinforced plastics gave the cyclorotor a possibility for a comeback. Whereas enthusiastic amateurs build radio-controlled toys, serious companies are designing prototypes for commercial usage like CycloTech GmbH [3].

Over the past few years, scientists at the Institute of Aerodynamics and Gas Dynamics have researched the cyclorotor in hover flight. The objective of the investigations is to obtain a better knowledge of the cyglogyro's flow conditions and to reduce the power consumption or increase the efficiency, respectively. The primary tool for the research is computational fluid dynamic (CFD), but a prototype of a cyclogyro also came into use. One idea to improve the efficiency is to use an alternative pitching path or even a non-circular movement of the blades. The advantage of this new movement could lead to a reduction of the parasitic drag and avoid a stall. The pitching and trajectory for maximum efficiency shall be found by optimisation.

Based on this intent, this thesis came up, where a procedure shall be designed to optimise the pitching path as well as the trajectory of the blades. Adapted from existing calculations of cyclogyros, a CFD setup has to be built, which enables an arbitrary pitching path and trajectory. The movement is to be implemented using splines, where the corresponding parameters shall guarantee a continuity of the paths. The computational fluid dynamic is carried out by the open-source software OpenFOAM. The tool kit Dakota is used for optimisation.

2 Theory

This chapter gives a short overview of the basic theory.

2.1 Computational Fluid Dynamics

There are many numerical approaches to solving differential equations (DE), like the Finite Difference Method for ordinary DEs. The Finite Element Method is used to solve a structural problem, which is the means of choice for partial DEs. For the computational investigation of flow processes of fluids and gasses, the Finite Volume Method (FVM) has been established, as this method can handle (hyperbolic) conservation equations well, see Munz [4].

To obtain the physical quantity of a flow field, density ρ, velocity vector $\mathbf{v}$, pressure p, and energy e, the following equations need to be solved in each volume cell.

Mass conservation:

$$\rho_t + \nabla \cdot (\rho \mathbf{v}) = 0 \ .$$

Momentum conservation:

$$(\rho \mathbf{v})_t + \nabla((\rho \mathbf{v}) \circ \mathbf{v}) + \nabla p = \nabla \cdot \boldsymbol{\tau} + \mathbf{g} \ .$$

Energy conservation:

$$e_t + \nabla \cdot (\mathbf{v} \cdot (e + p)) = \nabla \cdot (\boldsymbol{\tau} v) - \nabla \cdot \mathbf{q} + Q \ .$$

The subscript t marks the derivative with respect to time, i.e. $\frac{\mathrm{d}\rho}{\mathrm{d}t} \equiv \rho_t$. The symbol "$\circ$" represents the dyadic product; the $\boldsymbol{\tau}$ is the friction tensor, the $\mathbf{g}$ vector contains the external forces, $\mathbf{q}$ is the heat flux through the boundaries, and Q is the amount of heat supplied from outside [4].

These three conservation equations together form the Navier-Stokes equations for Newtonian fluids. Depending on the state of the problem, steady or transient flow, compressible or incompressible, the equations can be simplified. Many numerical solvers for different flow problems are mostly embedded in commercial software products such as ANSYS Fluent or STAR-CCM+. Nevertheless, there is also free, open-source software like OpenFOAM, released in 2004, which provides only the numerical solvers and simple mesh generators but no post-processing. This tool is used for this thesis with the release OpenFOAM-v2112.

2.2 OpenFOAM solver

To obtain a stable CFD calculation, the most explicit solvers must meet particular criteria, the so-called CFL condition, invented by Courant-Friedrichs-Lewy. This stability criterion depends on the velocity with which information is transported through the mesh, the time step Δt and the minimum cell length Δx, [4].

$$\text{CFL} = |a| \cdot \frac{\Delta t}{\Delta x} \quad < 1$$

If the CFL value is less than one, it is ensured that the information is transported from one cell to another without skipping a cell. With this equation, the time step can be estimated.

$$\Delta t = \frac{\Delta x}{|a|} \cdot \sigma$$

A safety factor σ was introduced to counteract numerical inaccuracies. With this value, the *Courant number* σ shall be less than one, [4].

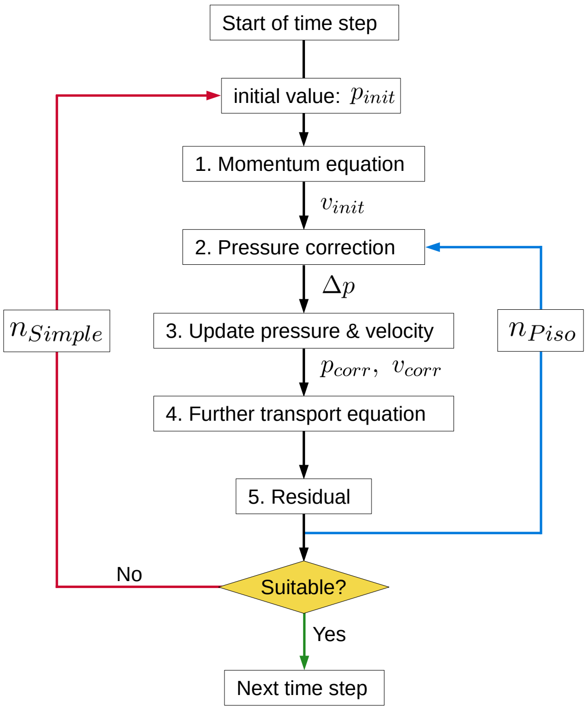
One of the used OpenFOAM solvers is the `pimpleFfoam`, which is applicable for Newtonian fluid's incompressible, turbulent flow. This algorithm combines two further solvers, a steady-state solver `simpleFoam` and the transient solver `pisoFoam`.

The general steps of the SIMPLE algorithm are as follows:

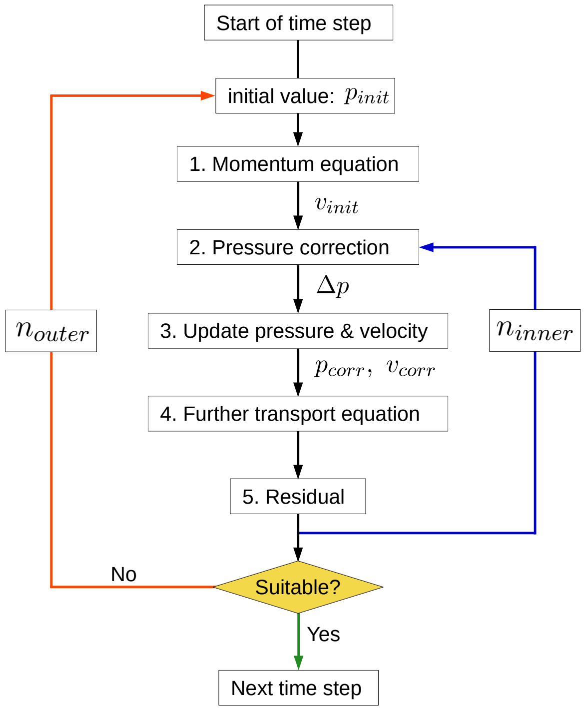
1. Based on an initial guess of the pressure field p_{init}, the momentum equation provides an initial velocity field v_{init}.
2. The Poisson equation calculates a pressure correction Δp, which represents the difference between the correct pressure field p_{corr} and the initial pressure field p_{init}.
   $p_{corr} = p_{init} + \Delta p$
3. The pressure field and the velocity are updated to obtain p_{corr} and v_{corr}.
4. Further transport equations like turbulence and temperature can be solved.
5. A residual checks the quality of the results. If the quality is suitable, the calculation will proceed with the next time step. If not, the algorithm starts the loop over again, where the current results are used as the initial parameters.

In contrast, the PISO algorithm has an inner loop, which calculates the steps 2, 3 and 4 multiple times until the solution converges. However, the computational expensive and slow momentum equation (step 1) is calculated only once. This approach requires a CFL number smaller than one, which entails a very small time step.

The PIMPLE algorithm consists of two loops. The inner loop, which is the PISO algorithm, and the outer loop. The number of performed loops n_{inner} and n_{outer} can be defined. This adjustment enables the user to get a stable calculation with a CFL value or a Courant number much higher than one, see Holzmann [5].



(a) `simpleFoam` / `pisoFoam`



(b) `pimpleFoam`

Figure 2.1: The procedure of different OpenFOAM solver algorithm in accordance with Jiyuan, [6].

Another OpenFOAM solver is the `overPimpleDyMFoam`, which is based on the pimple algorithm with the extension for chimera interpolation between multiple meshes [7].

The user can control the solvers with different parameters and flags to get reasonable results. Extensive knowledge is necessary to correctly adjust the solver, particularly for overset use. The CFD cases for this thesis are based on customised input templates, which are applicable for investigating cyclorotors (see section 4.4).

2.3 Turbulence

Flow processes are almost subject to turbulence, which can be seen in the form of eddies in the water or smoke. The flickering over the asphalt on a hot day is also the result of turbulent air. The Navier-Stokes equation implemented in CFD simulations can determine this stochastic movement of fluids. This equation is implemented in *direct numerical simulation* (DNS) and requires a very fine grid with tiny time steps. However, the tremendous computational effort required for these analyses is disproportionate mainly to the results, which are also rarely needed at this resolution. The introduction of specific turbulence models circumvented this problem.

One of the models is the unsteady Renolds averaged Navier Stokes (URANS) equation, adding a stress tensor to the momentum equation [8]. The $k - \omega$ SST-Model, is a sub-group of RANS and is used for this thesis.

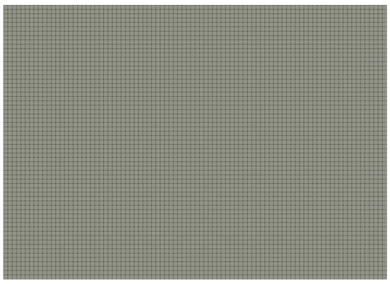
2.4 Chimera method

The chimera method, or overset method, as it is also called in OpenFOAM, is a method to interpolate between two or more finite volume meshes (FVM). In this approach, non-matching meshes can be used, necessary for complex geometries, i.e. landing gears, and relative body motion can be implemented, i.e. to consider flaps deflection and propeller rotation. The procedure will be explained on the basis of the current task, a moving blade.

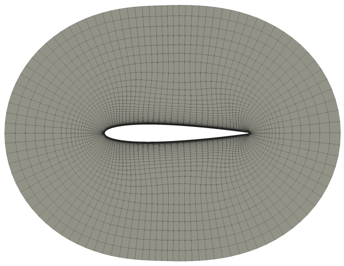
A background mesh covers the whole domain flow field and the component mesh containing the airfoil. Figure 2.2 shows an example of an overset setup. For the calculation, OpenFOAM adjusts the behaviour of specific cells [9]. The *calculated* cells are normal FVM cells, marked blue in Figure 2.3. The field information will be transferred between the meshes at the *interpolation* cells, marked grey. Furthermore, there are the *blocked* cells in which the moving body lays. These cells will be neglected for one specific time step, marked red in 2.3(a). Based on the pimple algorithm, the solver handling the overset method is the `overPimpleDyMFoam`.

With all the benefits that the Overset offers, numerical difficulties are possible. The most apparent problem is the non-physical pressure fluctuation, a well-known phenomenon [10]. Many runs with various solver options were tested, and even a different mesh approach, according to [11], was carried out (see Subsection 3.2.3), but the pressure oscillation remained.

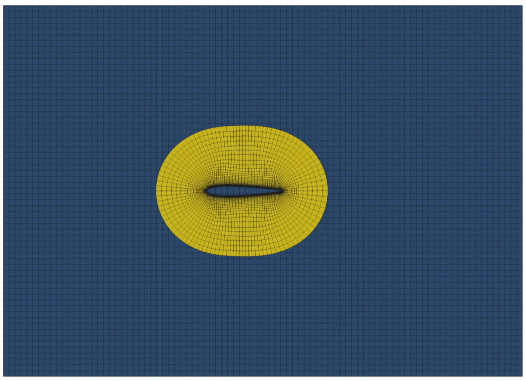
This phenomenon also has an impact on the resulting forces and moments. A comparison of the thrust for one blade cyclogyro is shown in Figure 2.4, once with an entire mesh motion and once with overset. There is a considerable deviation between these thrust curves. However, the deviation between the mean thrust of the single case $\overline{T}_{single} = 0.537$ N and the overset case $\overline{T}_{oveset} = 0.543$ N is about 1%, which is acceptable. The noise of the overset curve is a result of the chosen relaxation factor.



(a) Background mesh for flow field.

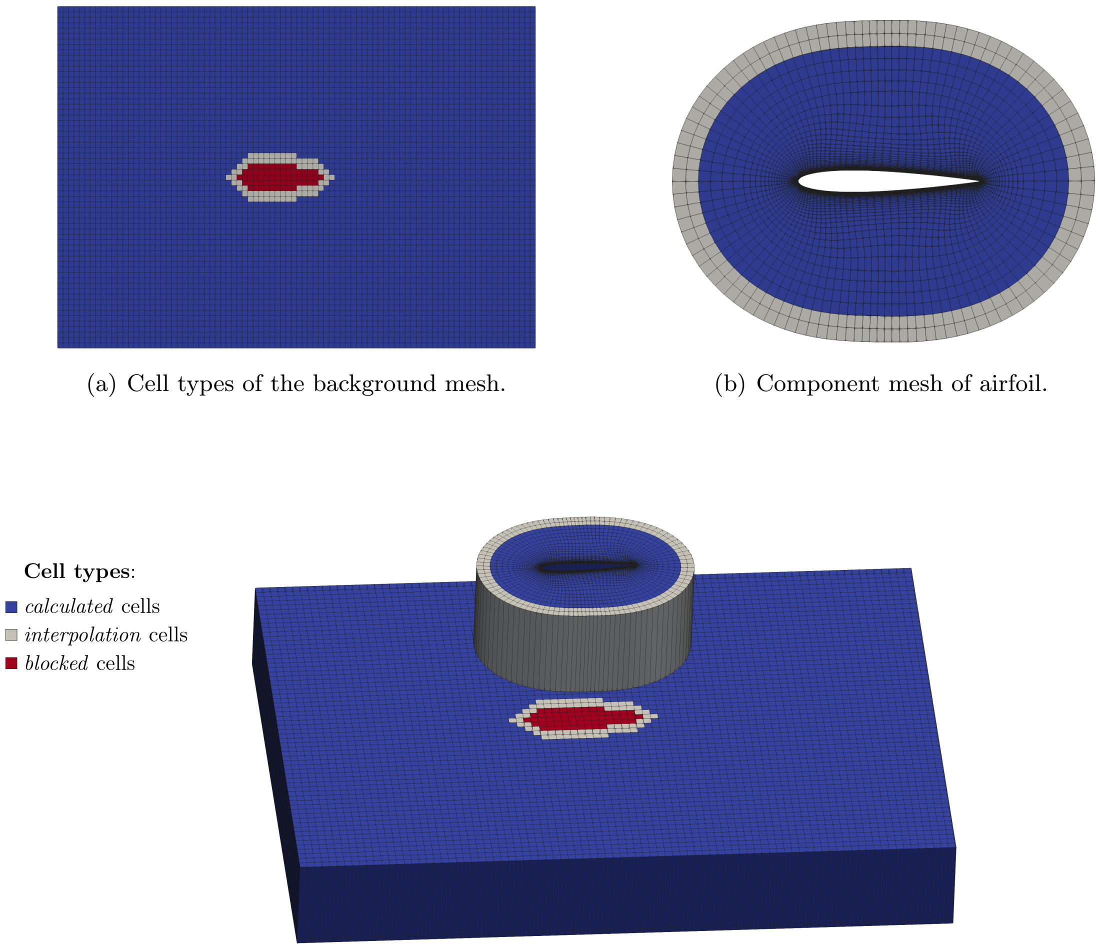


(b) Component mesh of airfoil.



(c) Merged mesh.

Figure 2.2: Example for a overset mesh setup.



(a) Cell types of the background mesh.

(b) Component mesh of airfoil.

(c) Merged mesh.

Figure 2.3: Various cell types for overset calculation.

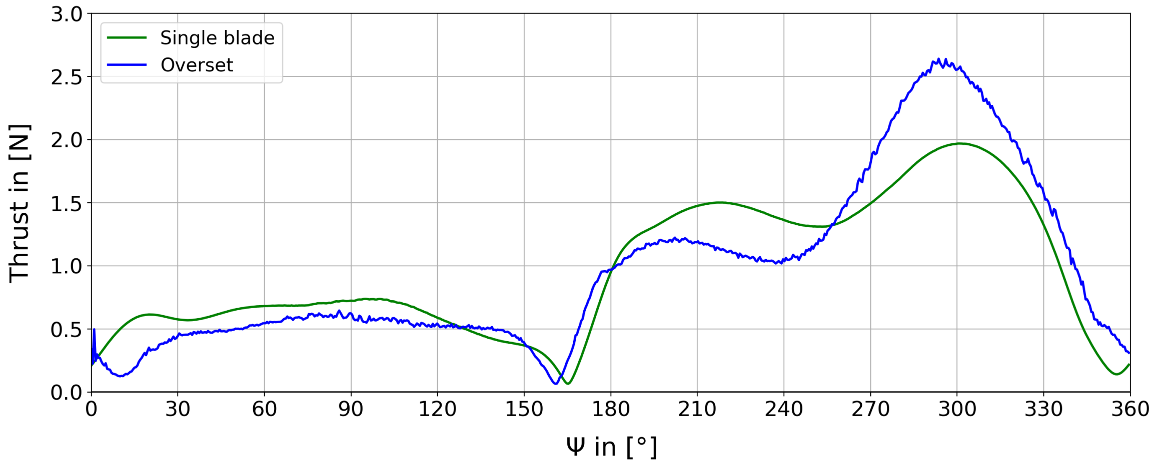


Figure 2.4: Comparison of the thrust over azimuth angle for a single blade mesh and an overset mesh.

2.5 Optimisation

The tool kit used for the optimisation is DAKOTA, which was initiated in 1994. The software offers different algorithms for optimisation and uncertainty investigations, parameter estimation, design of experiments and sensitivity analysis. Independent of the research subject, the main procedure is that DAKOTA sends a set of variable values to a user's simulation code. On the basis of the received objective functions (OF), DAKOTA determines a new set of values, see Figure 2.5. DAKOTA determines a new set of values based on the received objective functions (OF) (see Figure 2.5).

This loop is performed until the minimum of the objective functions has been found for a given convergence tolerance. Different constraints concerning the variable and the method can be defined.

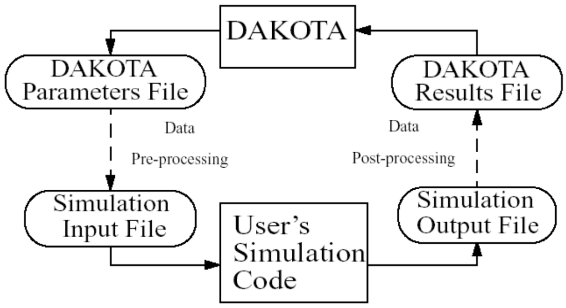


Figure 2.5: Flow chart of the DAKOTA procedure according to the User's Manual, [**Dakota_1**]. The dotted lines represent the data transfer, which has to be implemented by the user.

DAKOTA provides many methods for local as well as global optimisation. For optimising the pitching path and trajectory, a global method with EVOLUTIONARY ALGORITHMS (EA) is chosen. As the name suggests, this algorithm is inspired by biological evolution with the 'survival of the fittest'. The procedure is as follows:

- The algorithm generates design points (DP), which consist of randomly chosen values within the boundaries. This set of unique design points forms the first population, a string like a DNA.
- The resulting objective functions, provided by the simulation code, are assessed, and the 'fittest' or best OFs are selected.
- A crossover technique exchanges the variables of the best OFs (parents) and generates new design points (children).
- The 'mutation' is implemented by changing some variables of a design point with new random values.
- Parents and children form the second population, which are sent to the simulation code.

Figure 2.6 shows an example of an evolutionary algorithm for two populations.

The user can choose how the selection, crossover and mutation are performed. autoref shows the principle of a genetic optimisation.

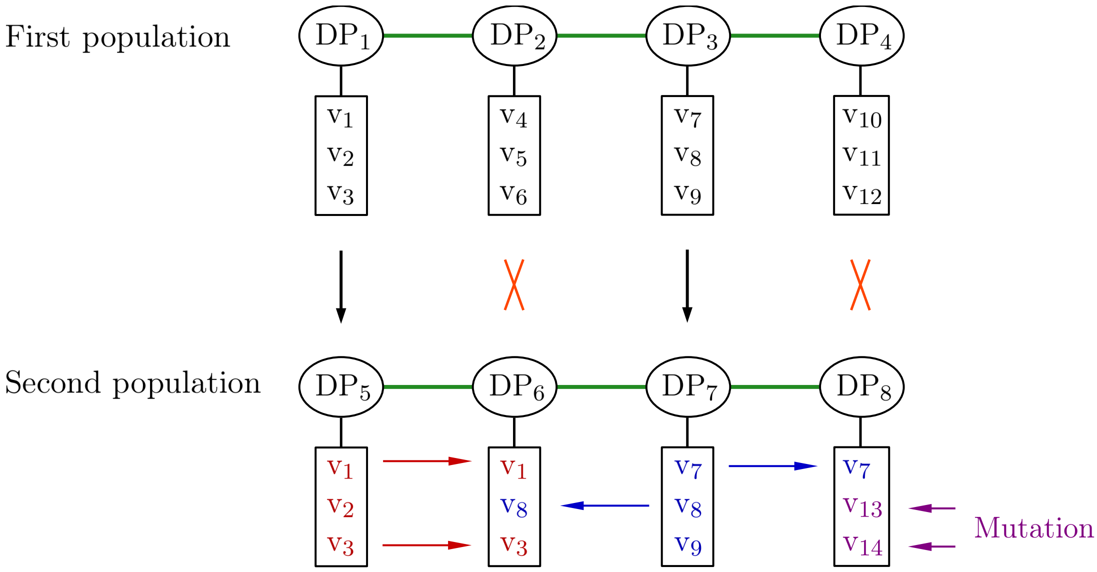


Figure 2.6: Example for the evolutionary algorithm.

3 Finite Volume Model

This section describes the basic geometry and the necessary mesh derived from them.

3.1 Geometry

The airfoil of the rotor blades is always the symmetric NACA0012 [12] with a chord length of 1 m. For the multi-blade calculation, the airfoils' trailing edge is shortened by about two percent of the total length to obtain a better mesh in this region. Both airfoil trailing edges are closed with an arc, which enables a good boundary layer extrusion growth. The ratio of chord length to the radius of rotation is given to be $R_C = 1.5$ as this value offers good efficiency for cycloidal rotors.

$$R_C = \frac{\text{chord length}}{\text{radius}} = \frac{c}{R} \overset{!}{=} \frac{2}{3} \tag{3.1}$$

Therefore, the resulting radius is 1.5 m. The depth of the 2D domain is set to 1 meter. The arc length of the circular trajectory is

$$L_{arc} = 2\pi \cdot R = 3\pi \tag{3.2}$$

This length is maintained constant for all optimisation runs, independent of the trajectory's shape.

The rotor is turning in the counter-clockwise direction, starting at the right-hand side with an azimuth angle of $\Psi = 0°$, see Figure 3.1. The aerodynamic centre of the blade lays at 25% of the chord length, which is also the pivot point for pitching. As a result of the rotation, there exists no upper or lower side of the blade. Therefore the blades are divided into an inner and an outer side.

Figure 3.2 shows the used coordinate systems necessary for the calculation and the postprocessing. There is a global Cartesian x-y system and three local systems with their origin in the blades' aerodynamic centre:

- a local Cartesian x-y system CS_{loc}, remains parallel to the global system.
- a tangential coordinate system CS_{tan}, where the x-axis stays tangential to the trajectory and points against the movement; the y-axis points outwards, see Figure 3.2(a)).
- an aerodynamic coordinate system CS_{aero}, where the x-axis lies on the blades' chord line pointing toward the trailing edge - direction of drag. The y-axis is pointing outwards, as shown in Figure 3.2(b).

Relevant angles are defined as follows:

Ψ	azimuth angle
α_0	pitching angle, between the tangent of trajectory and chord line
ϕ	counter angle, between the tangent of trajectory and the vertical
$\theta = \phi - \alpha_0$	difference angle

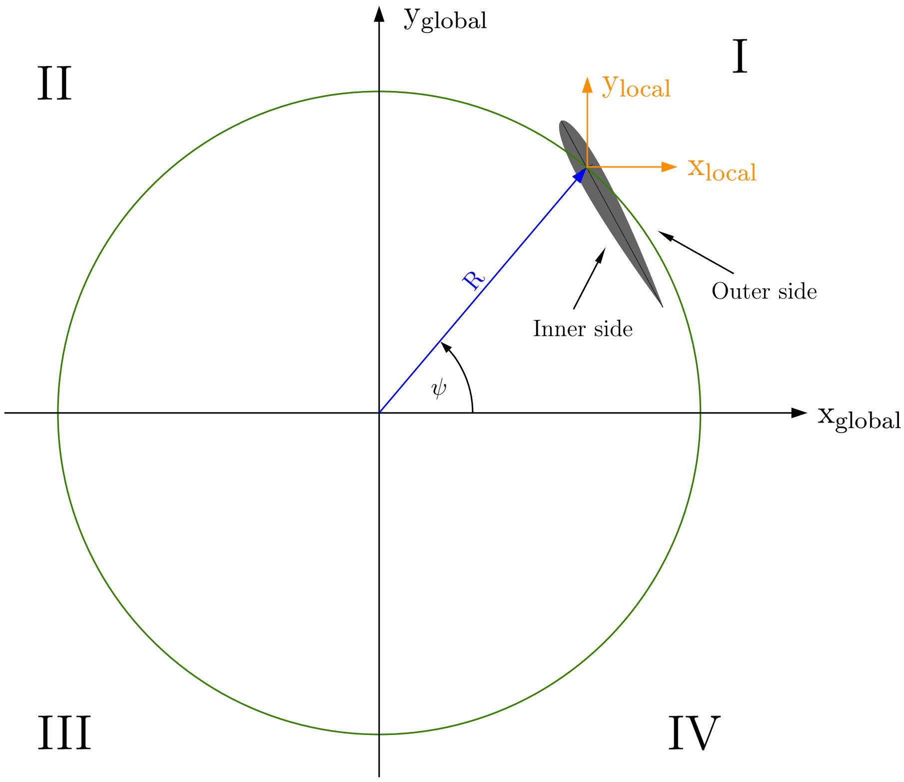
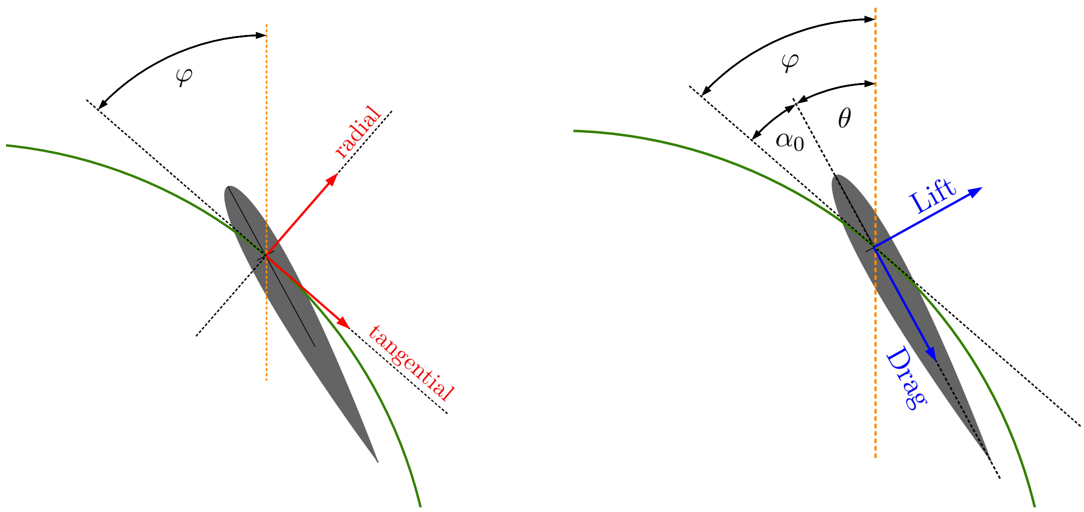


Figure 3.1: Basic geometrical definitions.



(a) Coordinate system tangential at the trajectory, CS_{tan}.

(b) Aerodynamic coordinate system, CS_{aero}.

Figure 3.2: Geometrical definition and coordinate systems.

During the optimisation, the number of blades varies from one to a maximum of four, $n_{Blade} = [1 \ldots 4]$. The initial position of the blades is on the circular trajectory with no pitching angle (see Figure 3.3). All geometrical dimensions are summarised in Table 3.1.

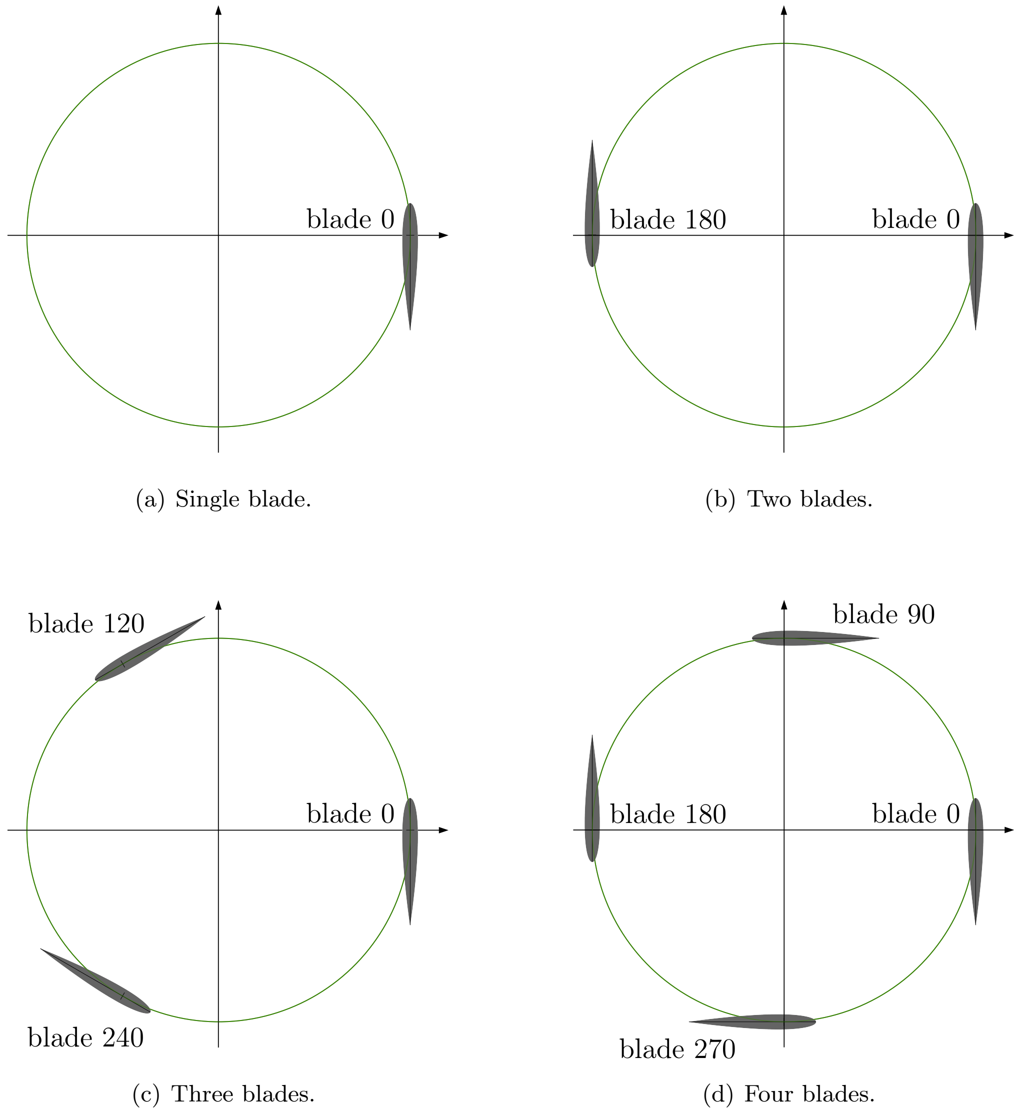


Figure 3.3: Initial positions of the blades for all considered combinations.

Table 3.1: Listing of all geometric dimensions used over all cases.

Name	Symbol	Value	Unit	Source
Type of airfoil		NACA0012	[−]	project definition
Chord lenth, single blade	c_{SB}	1.0011	[m]	calculated
Chord lenth, multi blade	c_{MB}	0.9922	[m]	calculated
Chord-Radius ratio	R_c	$\frac{2}{3}$	[−]	project definition
Radius of rotation	R	1.5	[m]	calculated du to given ratio
Number of blades	n_{blade}	1, 2, 3, 4	[−]	project definition
Depth of domain	d_z	1.0	[m]	unit value

3.2 Finite Volume Mesh

Depending on the number of blades, different approaches for generating finite volume meshes (FVM) are performed.

Due to the expected high number of single CFD cases necessary within the optimisation run, the main focus relies on low computational intensive meshes with a small number of volume cells. This approach is especially true for the overset method, where a substantial amount of field data must be interpolated between the blade and the background mesh. However, such coarse FVMs could neglect significant effects like separation or stall. The y^+ parameter, which should be equal to or less than one for a correct boundary layer calculation, reached values about two and highly depends on the blades' motion: the pitching angle and position at the arbitrary trajectory. The value exceeds only in a narrow range, which is acceptable. Figure 3.4 and Figure 3.5 shows the y^+ parameter for two optimisations. A detailed mesh convergence study was not carried out because satisfying accuracy was already determined for such parameters by Huang, [13], and Gagnon/Zimmer [14].

The distance between the farfield boundary to the blade remains constant and was defined to be 70 times the chord length.

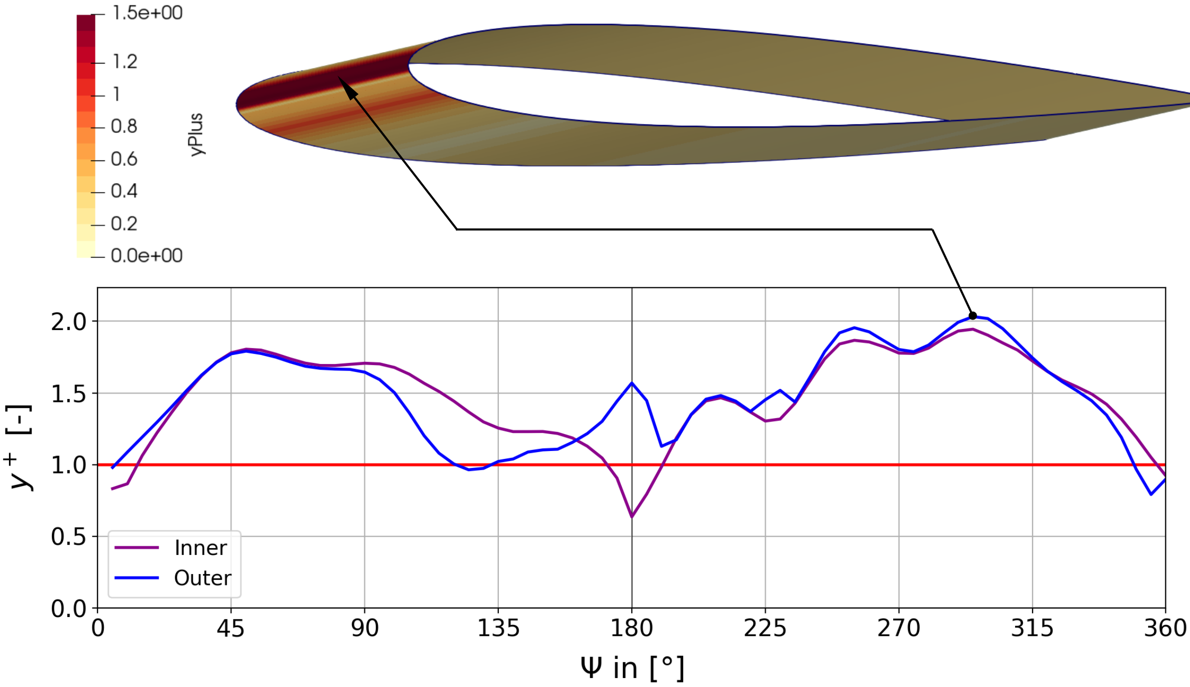


Figure 3.4: y^+ parameter for one blade optimisation (Opt-1BV).

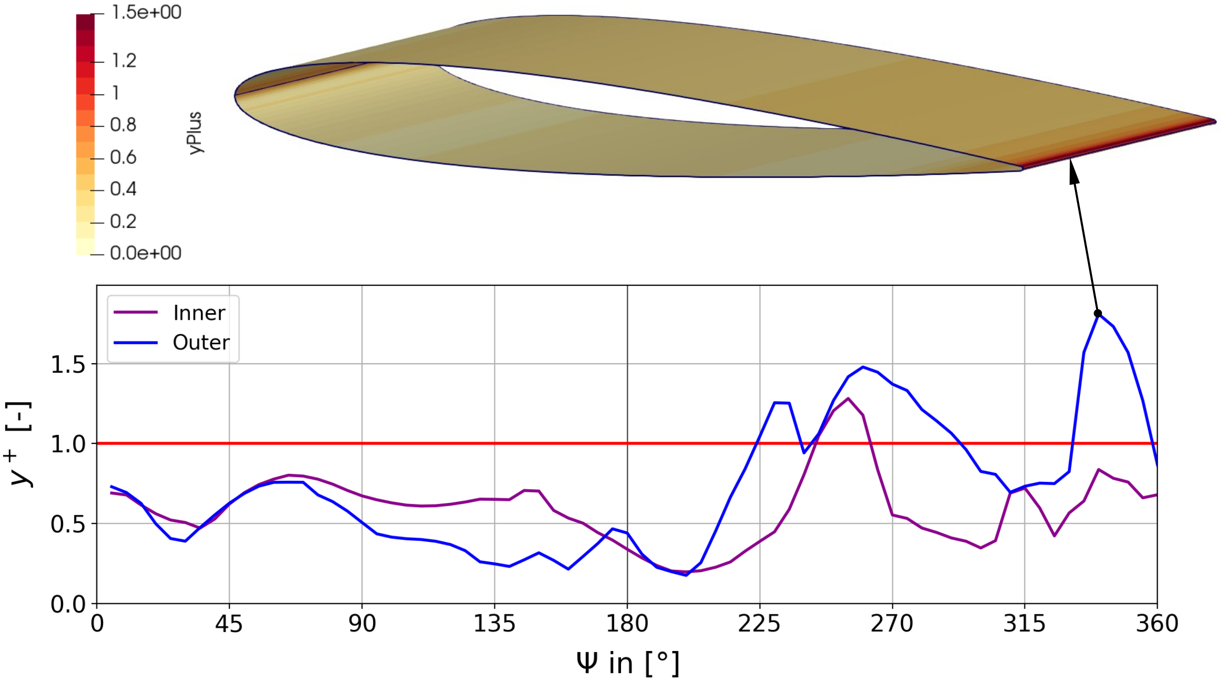


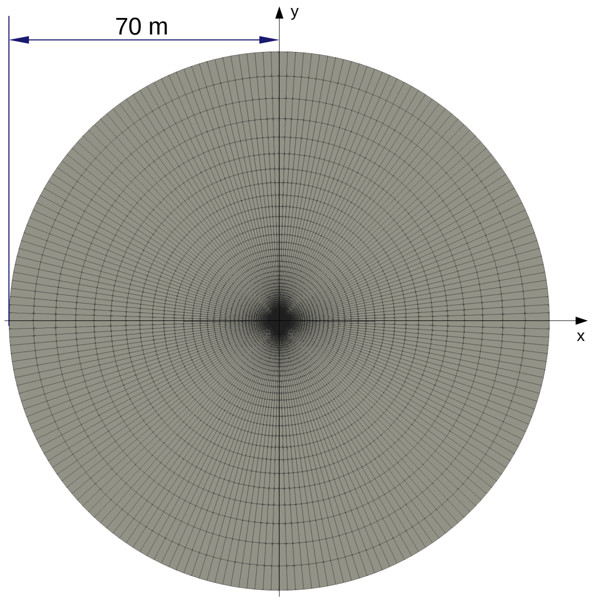
Figure 3.5: y^+ parameter for two blade optimisation (Opt-2BV).

3.2.1 Single blade mesh

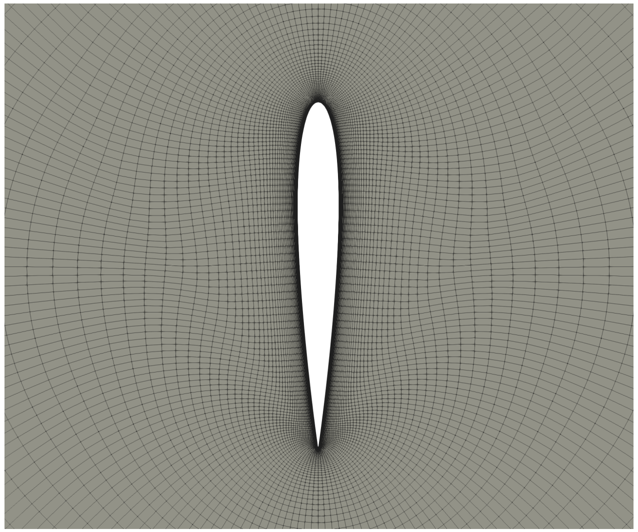
For the single blade case, the whole mesh is moved in translation and in rotation, and so a simple mesh extrusion from the blade contour to the farfield is done with the software Pointwise (see Figure 3.6 and Figure 3.7). Table 3.2 lists the mesh generation properties for the single mesh. The single blade mesh consists of 23 920 volume cells.

Table 3.2: Listing of the mesh generation properties.

Initial Δs	Growth rate	Number of steps	Method
$2 \cdot 10^{-4}$ m	1.05	10	hyperbolic
-	1.1	105	hyperbolic

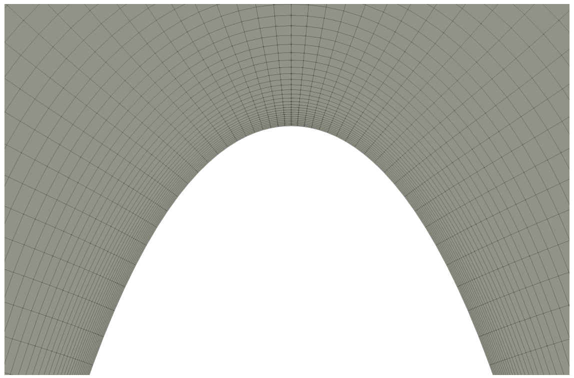


(a) Farfield of single blade mesh.

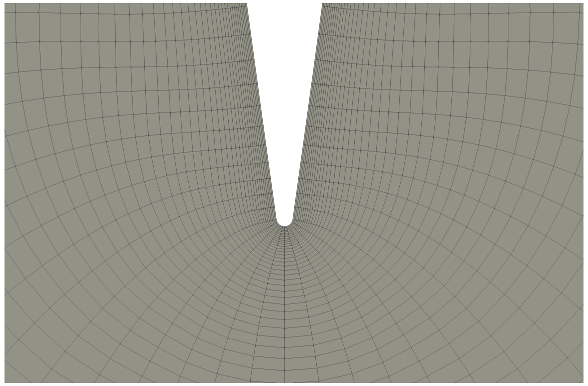


(b) Closer look at NACA0012 blade mesh.

Figure 3.6: Mesh for single blade CFD.



(a) Mesh at the leading edge.



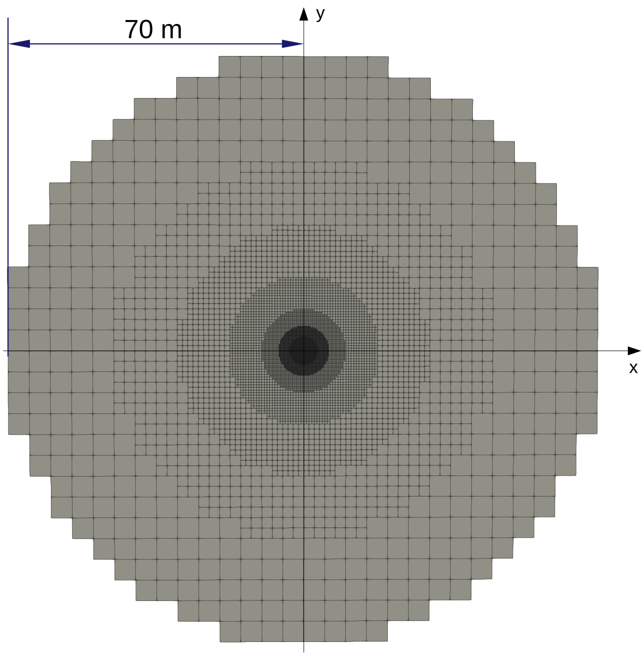
(b) Mesh at the trailing edge.

Figure 3.7: Closeup view of the *single blade* mesh.

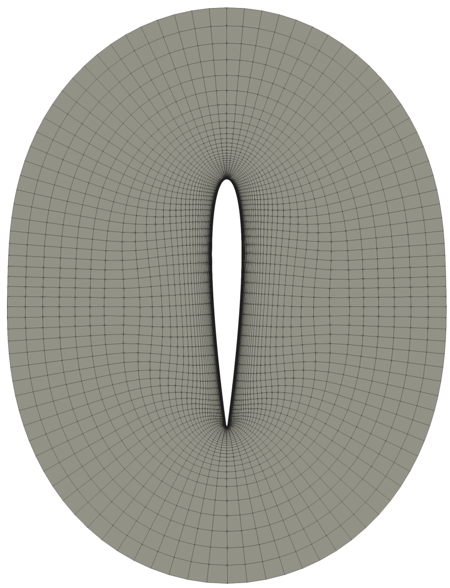
3.2.2 Overset mesh

The overset method enables an arbitrary movement for more than one blade (for theory description, see section 2.4). The local blade mesh is generated by Pointwise with 6 344 volume cells (see 3.8(b)). The blade for the overset has a finer boundary layer mesh than the single blade mesh. This approach is necessary to reduce the influence of pressure oscillation. The mesh generation properties are in Table 3.3.

The background mesh is generated within two steps. At first, a quadratic background mesh is built with an edge length of 140 m and a cell length of 2.4 m in x- and y-direction (one cell in depth). The split hex, mesh generator SnappyHexMesh came is used for the refinement of the mesh, controlled by an OpenFOAM dictionary. Five concentric regions ensure an adequate mesh propagation starting from a coarse block mesh (see Table 3.4 for properties). For the CFD analysis with a non-circular trajectory, the last refinement step is defined by an individual geometry with a refinement level of 6. A python script generates an STL geometry representing the outline created by the blades' movement over one rotation. A closer look into the algorithm is given in subsection 8.3.2. As a result of the unique meshing for each case, the number of volume cells varies. One can estimate the extent of the meshes based on the values given in Table 3.5.



(a) Standard background mesh of flow domain, 19 500 cells.



(b) Component mesh of blade, 6 344 cells.

Figure 3.8: Mesh parts for overset method.

Table 3.3: Listing of the mesh generation properties.

Initial Δ s	Growth rate	Number of steps	Method
$1 \cdot 10^{-4}$	1.05	10	hyperbolic
-	1.1	12	hyperbolic
-	1.15	28	hyperbolic
-	1.175	5	hyperbolic
-	1.05	6	algebraic

Table 3.4: Listing of the mesh generation properties.

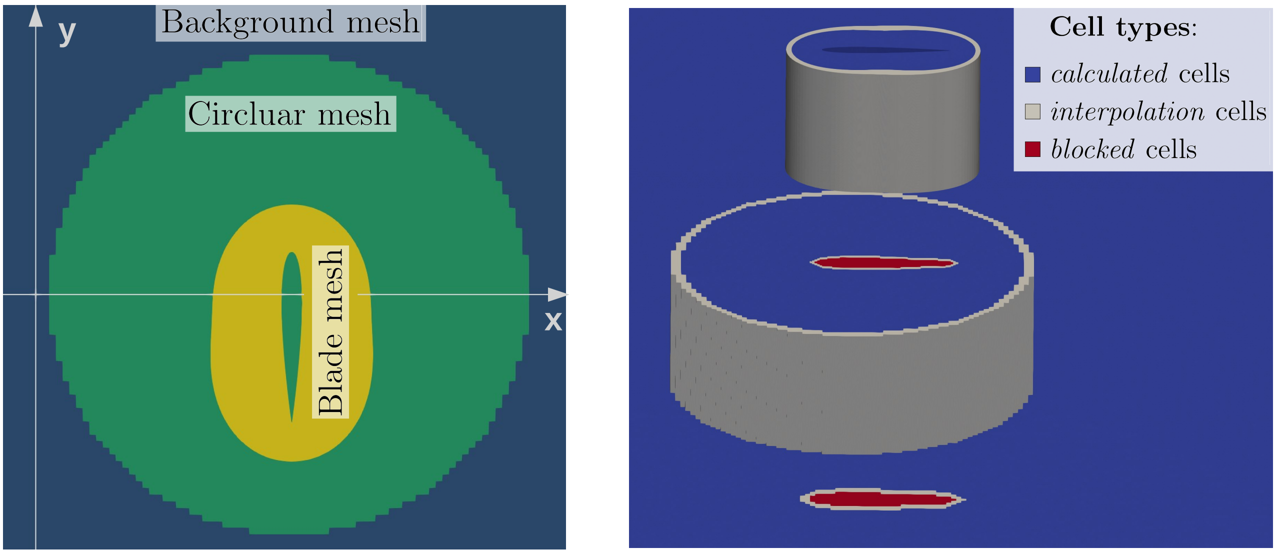
#	Radius	Refinement level
1	6 m	5
2	10 m	4
3	18 m	3
4	30 m	2
5	45 m	1

Table 3.5: Number of volume cells for different number of blades; for the pitching optimisation.

Number of blades	1	2	3	4
Number of volume cells:	23 920	32 188	38 532	44 876

3.2.3 Dual overset mesh

As mentioned in Section 2.4, there are small pressure fluctuations all over the flow field, see Figure 3.10. These oscillations are most substantial near the rotor blades but extend into the farfield. During an online training, held by wolf dynamics, [11], they presented a dual overset to eliminate pressure problems and increase accuracy. The approach is to add another mesh between the blade mesh and the background. The advantage for the cyclogyro is seen in the uncoupling of the rotation and the translation. A circular mesh is generated in which the blade is embedded (see Figure 3.9). Contrary to expectations, the fluctuation still occurrs, with the execution time increasing simultaneously due to the higher mesh interpolation effort. Since it was not possible to suppress the pressure fluctuations with this approach, the idea of the dual overset mesh was discarded.



(a) Merged mesh.

(b) Cell types.

Figure 3.9: Dual mesh.

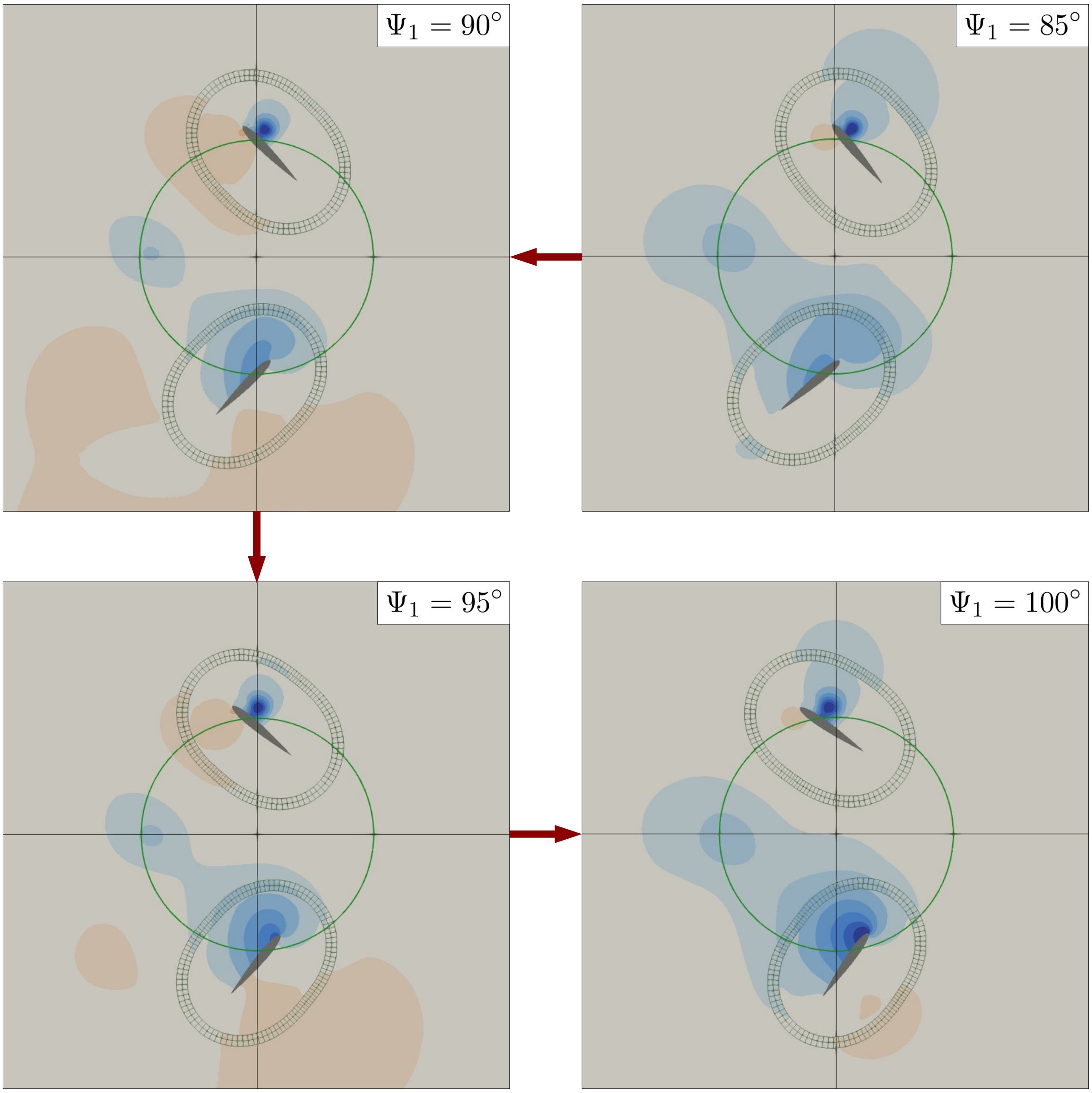


Figure 3.10: Pressure fluctuations as a result of the overset method. The black wireframes represent the interpolation cells.

4 Input data

All essential input values and properties for the CFD solvers are described in this section.

4.1 General properties

With few exceptions, the Reynolds number is set to 50 000 and remains constant for most investigations. The upstream flow for the blades is then

$$v_{Blade} = \frac{\mathrm{Re} \cdot \nu}{c} \; = 0.775 \ \mathrm{\tfrac{m}{s}} \ , \tag{4.1}$$

with the kinematic viscosity for 22°C of $\nu = 15.5 \cdot 10^{-6} \ \mathrm{\frac{m^2}{s}}$ [15].

The angular velocity for a circular movement with a radius of 1.5 m is

$$\omega_{rot} = v_{Blade} \cdot R \; = 0.516\overline{6} \ \mathrm{\tfrac{1}{s}} \ . \tag{4.2}$$

The duration of one revolution is the period

$$T_P = \frac{2\pi}{\omega_{rot}} \; = 12.1608 \ \mathrm{s} \ . \tag{4.3}$$

The shape of the trajectory is adjusted during the optimisation. However, the arc length and thus period is constrained to remain constant.

For the air density, a standard value of $\rho = 1.225 \frac{\mathrm{kg}}{\mathrm{m}^3}$ is chosen.

For a few number of optimisation runs, higher Reynolds numbers are carried out, see Table 4.1.

The optimisation is carried out for the hovering flight. Therefore the free stream velocity U_∞ of the farfield is zero.

Table 4.1: Considered Reynolds numbers, rotational speed and period.

Reynolds number	Rotational speed	Period
50 000	0.5167 $\frac{1}{\mathrm{s}}$	12.1608 s
100 000	1.0334 $\frac{1}{\mathrm{s}}$	6.0804 s
200 000	2.0667 $\frac{1}{\mathrm{s}}$	3.0402 s

Table 4.2: Listing of all constant boundary conditions used over all cases.

		Value	Unit
Reynolds number	Re	50 000	$[-]$
Air density	ρ	1.225	$\left[\frac{\mathrm{kg}}{\mathrm{m}^3}\right]$
Kinematic viscosity	ν	$15.5 \cdot 10^{-6}$	$\left[\frac{\mathrm{m}^2}{\mathrm{s}}\right]$
Chord lenth	c	1.0	[m]
Free stream velocity	U_∞	0	$\left[\frac{\mathrm{m}}{\mathrm{s}}\right]$

4.2 Mode of drive

To obtain an arbitrary trajectory for a cyclogyro, the whole design and the mode of drive must to be different from a usual one. This thesis considers two possible ways to realise a non-circular trajectory, which influences the blades' velocity. That is why the designs are subdivided into a 'constant velocity' drive and a 'constant angular velocity' drive.

4.2.1 Constant velocity drive

A rail in the shape of the desired trajectory can be used to achieve a constant velocity of the blades. A slider with a joint connects the blade with the rail. Separate actuators can realise the pitching of each blade. A chain connects the slider and ensures a constant distance between them. The chain can be driven by any reasonable power unit (electric motor, combustion engine or jet turbine). Figure 4.1 shows a sketch of the constant velocity drive. According to the three different Reynolds numbers considered for the optimisation, there are three different velocities.

$$v_{Re=50\ 000} = 0.775\ \frac{\mathrm{m}}{\mathrm{s}}\ , \qquad v_{Re=100\ 000} = 1.55\ \frac{\mathrm{m}}{\mathrm{s}}\ , \qquad v_{Re=200\ 000} = 3.1\ \frac{\mathrm{m}}{\mathrm{s}}\ .$$

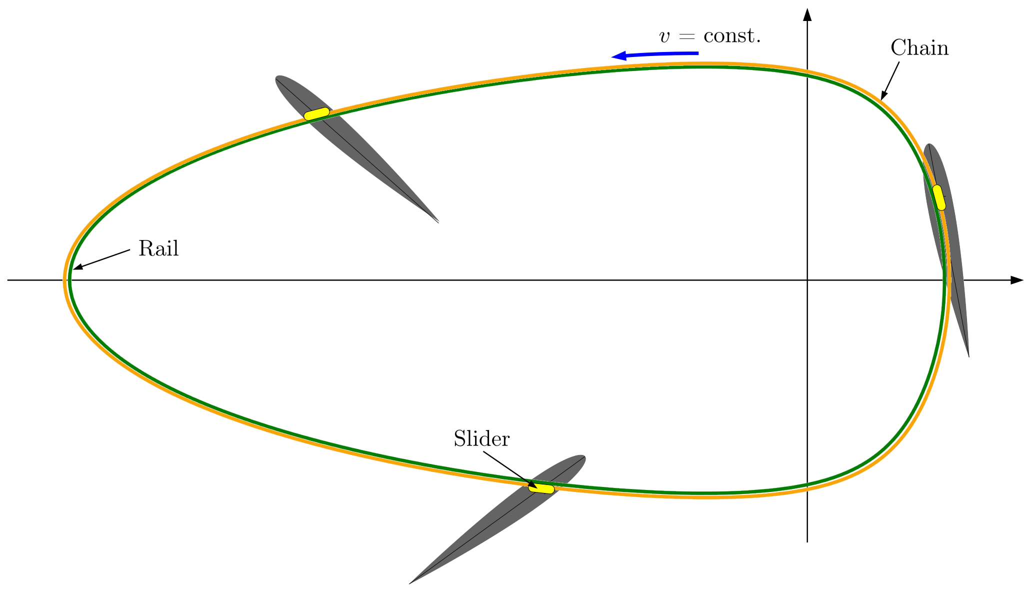


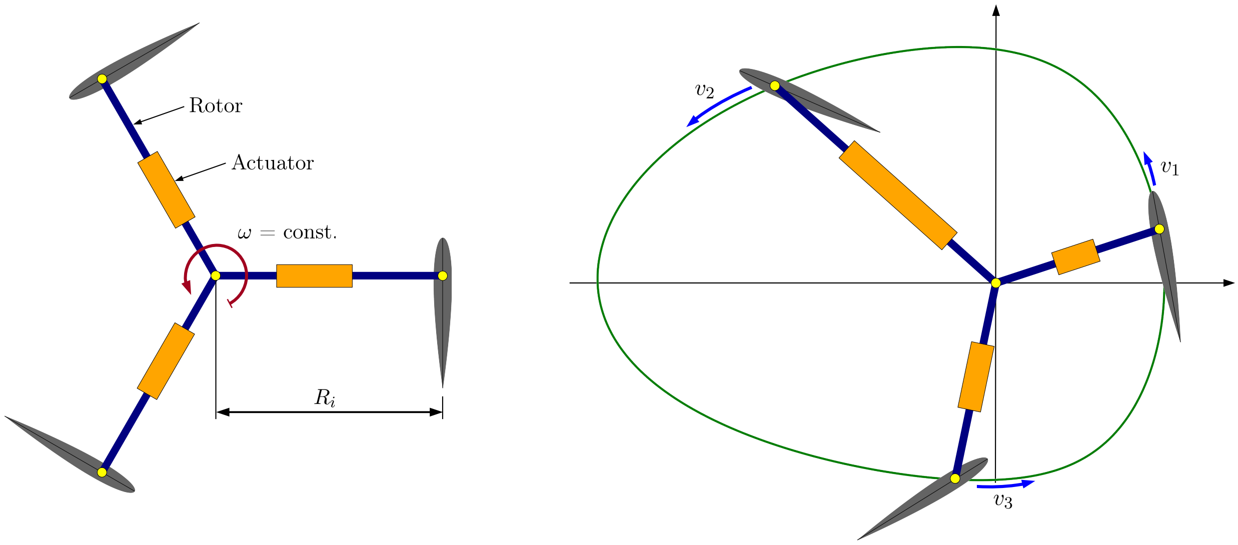
Figure 4.1: Sketch for a contant velocity drive.

4.2.2 Constant rotation speed drive

This drive mode has a separate rotor arm with a mounted blade, which is similar to the ideas of Bogrash, [16]. Each rotor arm has an actuator to change the arms' length or the blades' radius, respectively (see Figure 4.2). With a constant angular velocity, each blade has a different speed depending on the current radius.

$$v_i = R_i \cdot \omega_{rot}$$

The advantage of this drive mode is that the actuators can adapt the trajectory during the operation. So, the most efficient trajectory can be applied for different rotor RPMs.



(a) Basic drive design.

(b) Adjusted actuators to obtain trajectory.

Figure 4.2: Sketch for a constant rotation speed drive.

4.3 Turbulence properties

The turbulence model $k - \omega - SST$ is used for the CFD, which requires the turbulence kinetic energy k_e and specific turbulence dissipation ω_{diss} as input values. In general, the farfield and consequently the inflow is not disturbed. However, a turbulence intensity for the input values is needed, estimated to be 1% of the upstream flow. According to the OpenFOAM user guide, [17], the turbulence kinetic energy is

$$k_e = \frac{3}{2} \cdot (Tu \cdot v_{Blade})^2 \quad = 9.01 \cdot 10^{-7} \ \frac{\mathrm{m}^2}{\mathrm{s}^2} \tag{4.4}$$

Then, the specific turbulence dissipation rate can be calculated [17].

$$\omega_{diss} = \frac{k_e^{0.5}}{C_\mu^{0.25} \cdot c} \quad = 1.73 \cdot 10^{-3} \ \frac{1}{\mathrm{s}} \tag{4.5}$$

with $C_\mu = 0.09$ and c = 1 m.

4.4 CFD input

The incompressible, transient OpenFOAM solver `pimpleFoam` and `overPimpleDyMFoam` are used for the CFD calculations. A short overview of the main properties and input values for both solvers are given in the following. The generic dictionaries for initial values, solver input and numerical schemes can be found in Appendix D.

One essential input value is the time step Δt, which significantly impacts the accuracy of the results and the execution time of the CFD runs. Different test cases are carried out with varying time steps for the single blade and overset mesh. A suitable time step is $\Delta t = 16.89$ ms, for which the figure of merit remains approximately constant with a reasonable execution time (see section 8.1 for figure of merit). This time step corresponds to an azimuth angle of $\Delta\Psi = 0.5°$ and is set for all optimisation cases.

To determine the necessary number of rotations and thus the end time, four cases with 100 rotations each are carried out. These differ in the mesh setup (single blade $\leftrightarrow$ overset) and the implementation of rotation (rotating moation $\leftrightarrow$ spline motion); see the following paragraph.

Figure Figure 4.3 shows the figure of merit over the rotations for the single blade case, where a constant FoM is reached after 14 revolutions. The increase of the FoM for revolutions greater than 22 is presumably due to the farfield size. No further investigations are conducted on this issue. For the single-blade optimisation, 14 revolutions are carried out.

The FoM curve for the overset case shows different behaviour, see Figure Figure 4.4. Even for a high number of rotations (>50), there are small discontinuities of the FoM. A reasonable value is reached at about 90 rotations. The execution time for an overset case is three times higher than a single mesh case. Thus a high number of rotations for the calculation is unacceptable for the optimisations.

The following values are defined, which represent a suitable compromise between he accuracy of the results and the execution time.

Case	Revolution	End time
Single blade	14	170.2512 s
Overset	10	121.608 s

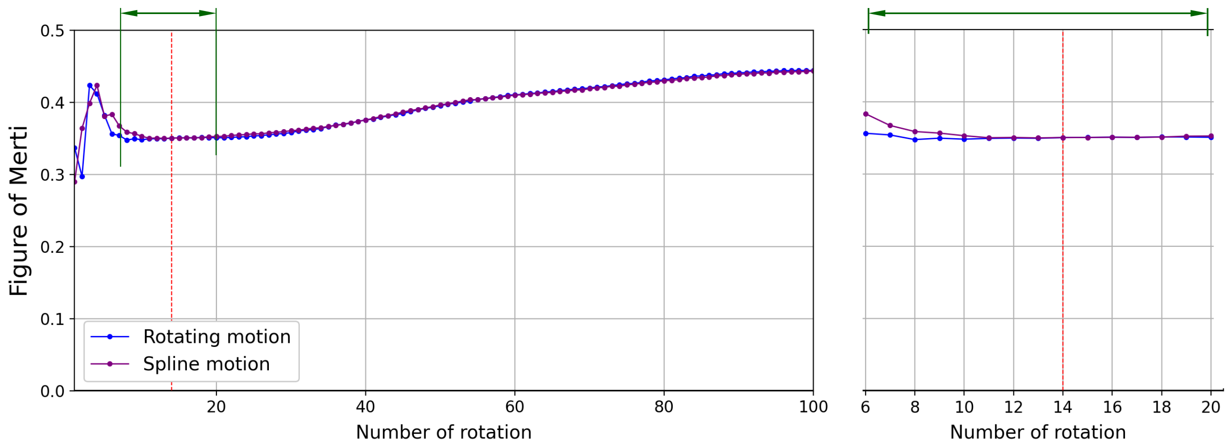


Figure 4.3: Figure of merit over 100 rotations for single-blade case.

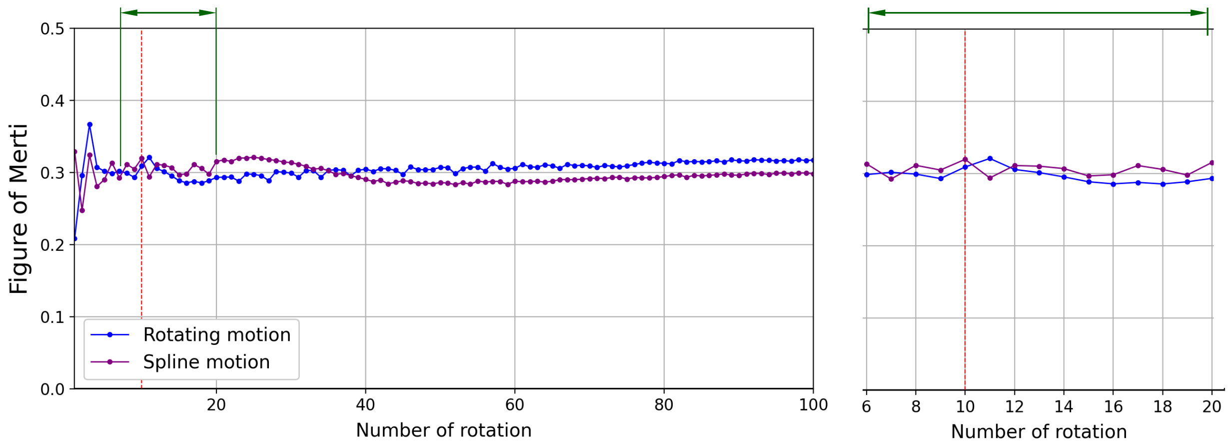


Figure 4.4: Figure of merit over 100 rotations for two-blade case.

Mesh motion

According to the number of blades, the following motion class are defined in the `dynamiMeshDict`.

Case	Motion class	
Single mesh	`dynamicFvMesh`	`dynamicMotionSolverFvMesh;`
Overset	`dynamicFvMesh`	`dynamicOversetFvMesh;`

For a circular motion, the OpenFOAM class `rotatingMotion` is used. For the arbitrary trajectory, the customised motion class `bSplineMotion` is used, see section 7.3.

The customised class `bSplinePitching` defines the pitching path of the blades, see section 7.2.

Solver input and numerical schemes

The solver inputs and numerical schemes are taken from previous CFD calculations of cycloidal rotor performed by Gagnon/Zimmer [14].

For overset cases, the following code is added to the `fvSchemes`. A higher value for `nPushFront` than one enables less disturbance of the blades' boundary layer due to the overset interpolation. However, a higher push front can lead to large CFL numbers and crash the CFD case. Therefore the push front of one is set.

```
oversetInterpolation
{
        method          inverseDistancePushFront;
        searchBox       (-3.4 -3.4 -1)(3.4 3.4 1);
        voxelSize       0.008;
        nPushFront      1;
        layerRelax      0.5;
}

oversetInterpolationSuppressed
{}
```

The chosen relaxation factors for the pressure fields depend on the mesh motion, single mesh or overset. The factors for the overset lead to noisy blade forces, see Figure 2.4. However, the execution time is reduced by about 10%.

Case	Relaxation	
	`p`	`pfinal`
Single blade	0.3	1.0
Overset	0.3	0.7

5 NURBS

The task is to optimise the pitching of the airfoils and their trajectory with the DAKOTA toolkit. As the optimiser only provides discrete values, the pitching and translation path shall be defined by only a few parameters. Additionally, the paths should not contain peaks or discontinuities to obtain good CFD results.

The first idea to generate the trajectory by assembling segments of circles with different radii was discarded. The reason is that discontinuities appear at the connection between each circle. Therefore both pitching and trajectory are defined as NON-UNIFORM RATIONAL B-SPLINES (NURBS).

These NURBS are a mathematically exact representation of curves, surfaces or even volumes. The NURBS and their derivatives are continuous depening on the NURBS's order. With a few sets of parameters, NURBS can be highly adjusted at any will, making them valuable for many topics in computer-aided design. They can also be used for interpolation for a given set of data.

According to [18], NURBS are *"the projection of a nonrational (polynomial) B-spline curve defined in four-dimensional (4D) homogenous coordinate space back into three-dimensional (3D) physical space."*. The equation for NURBS is

$$P(t) = \sum_{i=1}^{p} B_i R_{i,k}(t).$$

With B_i as the control vertices, k is the order of the spline and p is the number of vertices. The possible maximum order k_{max} of the b-spline is equal to the number of the control vertices.

$$k_{max} = p$$

One can choose an order less than the maximum. The terms $R_{i,k}(t)$ are the rational basis function, which can be determined as follows.

$$R_{i,k}(t) = \frac{h_i \cdot N_{i,k}(t)}{\sum\limits_{i=1}^{p} h_i \cdot N_{i,k}(t)}$$

The resulting polynomial of the recursive equation has the degree $m = k - 1$. The parameter t is a control variable that defines the spline's position.

5.1 B-Splines

B-splines are a specific case of NURBS, where the parameter h_i is set to one.

$$h_i = 1, \quad \text{for all i} \quad \Rightarrow \quad \sum_{i=1}^{p} h_i \cdot N_{i,k}(n_T) = 1$$

As a result of this simplification, the b-spline offers less adjustments, which are still enough and lead to shorter equations.

Deviating from the definitions made in [18], the 2D b-spline is defined as

$$\gamma(n_T) = \left(\begin{array}{c} X_S(n_T) \\ \\ Y_S(n_T) \end{array} \right) = \sum_{i=1}^{p} N_{i,k}(n_T) \cdot V_i \tag{5.1}$$

where $\gamma(n_T)$ represents the splines coordinates, V_i contains the control vertices. The index i represents the number of the control vertex and ranges from one to the maximum vertex number p.

The parameter n_T is the control variable of the spline and defines the current position on the spline curve. The spline starts at $n_T = n_{T,min}$ and ends at $n_T = n_{T,max}$. The range of the control variable depends on the knot vector; see below.

The $N_{i,k}(n_T)$ are the basic functions, which can be calculated by the following recursive equation.

$$N_{i,k}(n_T) = \frac{n_T - K_i}{K_{i+k-1} - K_i} \cdot N_{i,k-1}(n_T) + \frac{K_{i+k} - n_T}{K_{i+k} - K_{i+1}} \cdot N_{i+1,k-1}(n_T) \tag{5.2}$$

$$N_{i,1}(n_T) = \begin{cases} 1 & \text{if } K_i \leq t < K_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

The equation results are polynomials with the degree $m = k - 1$, where k is the splines' order. The polynomials can be interpreted as weighting coefficients and determine the influence of each control vertex depending on n_T. The K_i are elements of the knot vector κ and has a great influence on the shape of the spline. They can assume any values with only one restriction: each element has to be equal to or greater than the previous element.

$$K_i \leq K_{i+1} \tag{5.4}$$

With k as the order of the b-spline and p the number of control vertices, the knots' elements can be determined.

$$\begin{aligned} K_i &= 0 \qquad \text{for} \quad 1 \leq i \leq k && (5.5) \\ K_i &= i - k \qquad \text{for} \quad k+1 \leq i \leq p && (5.6) \\ K_i &= p - k + 1 \qquad \text{for} \quad p+1 \leq i \leq p+k && (5.7) \\ & && (5.8) \end{aligned}$$

The knot vector also defines the range of the control variable n_T.

$$K_1 \leq n_T \leq K_p \tag{5.9}$$

5.2 Derivatives

The derivatives of the b-spline are essential to calculating the tangent angle or the curvature of the b-spline. The equations are taken from [18].

The first derivative of b-spline is

$$\frac{\mathrm{d}\gamma}{\mathrm{d}n_T} = \gamma'(n_T) = \begin{pmatrix} X'_S(n_T) \\ Y'_S(n_T) \end{pmatrix} = \sum_{i=1}^{p} N'_{i,k}(n_T) \cdot V_i \tag{5.10}$$

The first derivative of basis function is

$$\begin{aligned} N'_{i,k}(n_T) \;=\; & \frac{N_{i,k}(n_T) + (n_T - K_i) \cdot N'_{i,k}(n_T)}{K_{i+k-1} - K_i} + \\ & + \frac{(K_{i+k} - n_T) \cdot N'_{i+1,k-1}(n_T) - N_{i+1,k-1}(n_T)}{K_{i+k} - K_{i+1}} \;. \end{aligned} \tag{5.11}$$

The second derivative of b-spline is

$$\frac{\mathrm{d}^2\gamma}{\mathrm{d}n_T^2} = \gamma''(n_T) = \begin{pmatrix} X''_S(n_T) \\ Y''_S(n_T) \end{pmatrix} = \sum_{i=1}^{p} N''_{i,k}(n_T) \cdot V_i \;, \tag{5.12}$$

with its second derivative of basis function

$$\begin{aligned} N''_{i,k}(n_T) = \; & \frac{2 \cdot N_{i,k-1}(n_T) + (n_T - K_i) \cdot N''_{i,k-1}(n_T)}{K_{i+k-1} - K_i} + \\ & + \frac{(K_{i+k} - n_T) \cdot N''_{i+1,k-1}(n_T) - 2 \cdot N_{i+1,k-1}(n_T)}{K_{i+k} - K_{i+1}} \;. \end{aligned} \tag{5.13}$$

5.3 Arc length

To determine the arc length of a function analytically, the equation

$$L(\gamma) = \int ||\dot{\gamma}(n_T)|| \; \mathrm{d}n_T \tag{5.14}$$

is given by Papula [19]. The term gets too complex for an analytical integration. So the b-spline was integrated numerically. For small steps of the control variable $\Delta n_T = n_{T,i+1} - n_{T,i} = \mathit{const.}$, the distances between two neighbouring points on the trajectory are determined and summed.

$$L^*(\gamma) = \sum_{i=0}^{m} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{5.15}$$

5.4 Curve Fitting

Instead of determining a spline by given control vertices, the reverse procedure can be performed. For a given set of points, which lay on a curve, the control vertices shall be calculated so that the resulting spline fits the curve best. For this purpose, the equation for a b-spline can alternatively be written in a matrix formulation,

$$\gamma(n_T) = \underline{D} = \sum_{i=1}^{p} N_{i,k}(n_T) \cdot V_i = \mathbf{N} \cdot \underline{V} \tag{5.16}$$

with the vector $\underline{D}$ for the points on the arc length curve, the matrix $\mathbf{N}$ containing the basis values and the vector $\underline{V}$ for the control vertices. After calculating the inverse basis value matrix $\mathbf{N}^{-1}$, the control vertices are calculated via ordinary matrix multiplication.

$$\underline{V} = \mathbf{N}^{-1} \cdot \underline{D} \tag{5.17}$$

Figure 5.1 shows an example spline with its points and control vertices.

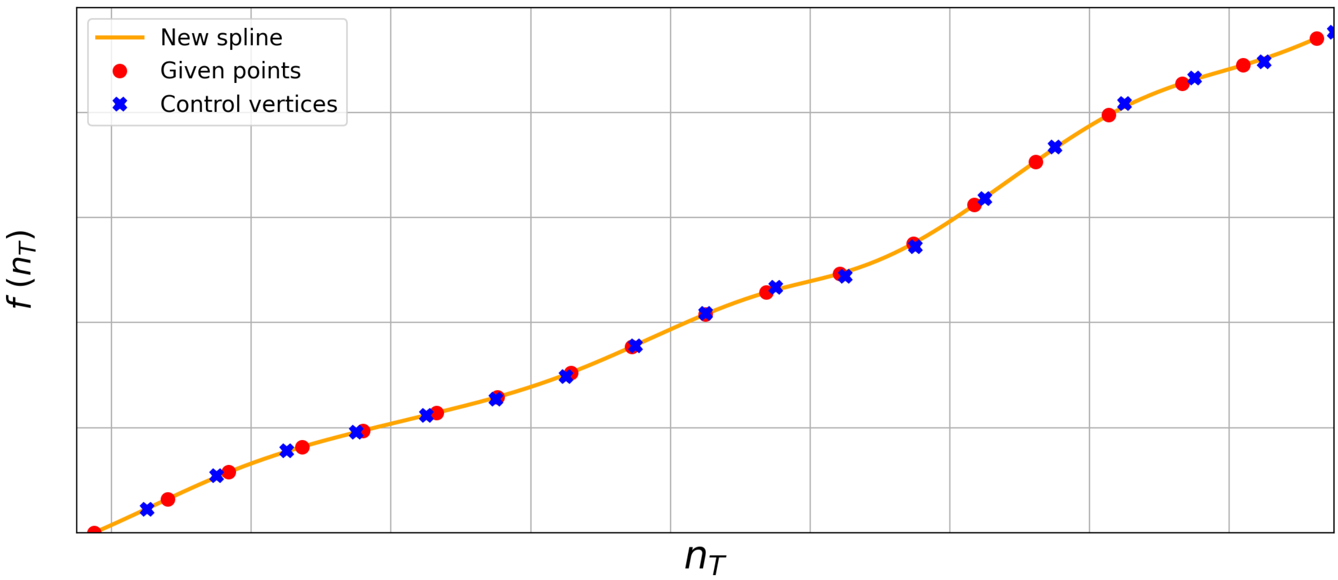


Figure 5.1: Example for a spline matching the given points.

5.5 Curvature

The curvature σ will be used to evaluate the splines' shape. It is the inverse value of the local spline radius. The following equation gives the curvature of a function (Papula [19]).

$$\sigma = \frac{\left|\dot{X}_S(n_T) \cdot \ddot{Y}_S(n_T) - \dot{Y}_S(n_T) \cdot \ddot{X}_S(n_T)\right|}{\left(\dot{X}_S^2(n_T) + \dot{Y}_S^2(n_T)\right)^{\frac{3}{2}}} = \frac{1}{R_{local}} \tag{5.18}$$

6 Arbitrary movement

This chapter describes the concrete application of the spline equations to obtain a pitching path and trajectory.

6.1 Pitching

Sixteen control vertices define the pitching spline with a degree of three (degree $m_{pit} = 3$, order $k_{pit} = 4$), enabling reasonable path control. To obtain a periodic spline without any discontinuities, two more vertices are inserted into the control vector $\underline{V}_{Pit}$, one at each spline end. The concerning equation for the pitching spline is

$$\gamma_{pit}(n_{T,pit}) = \sum_{i=1}^{19} N_{i,4}(n_{T,pit}) \cdot V_{i,pit} \tag{6.1}$$

with its control vector

$$\begin{aligned} \underline{V}_{pit} = [\ & P_0,\ P_1,\ P_2,\ P_3,\ P_3,\ P_4,\ P_4,\ P_5,\ P_6,\ P_7,\ P_8,\ P_9, \\ & P_{10},\ P_{11},\ P_{12},\ P_{13},\ P_{14},\ P_{15},\ P_{16}\ ,P_{17},\ P_{18}]\ , \end{aligned} \tag{6.2}$$

and the multiple control vertices

$$P_0 = P_{16}, \quad P_{17} = P_1, \quad P_{18} = P_2\ .$$

The vertices contain only one scalar value. The distance between the vertices depends on the knot vector. This vector with its evenly spaced elements K_i is

$$\kappa_{pit} = [\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 19,\ 20,\ 21,\ 22\ ]\ .$$

Figure 6.1 shows a resulting pitching spline. The control vertices are adjusted to get a sinusoidal form with an amplitude of 45°, see Figure 6.2. For the comparison, a sinus curve is also plotted in the figure.

Only a part of the spline will be used, marked with vertical lines. This is obtained by an adapted range of the control variable $n_{T,pit}$

$$3\ \leq\ n_{T,pit}\ \leq\ 19\ .$$

Within this range, the spline is subdivided into 16 sections.

The difference between both curves is shown in Figure 6.3. The highest deviation occurs at the end of the spline, which is -0.16°.

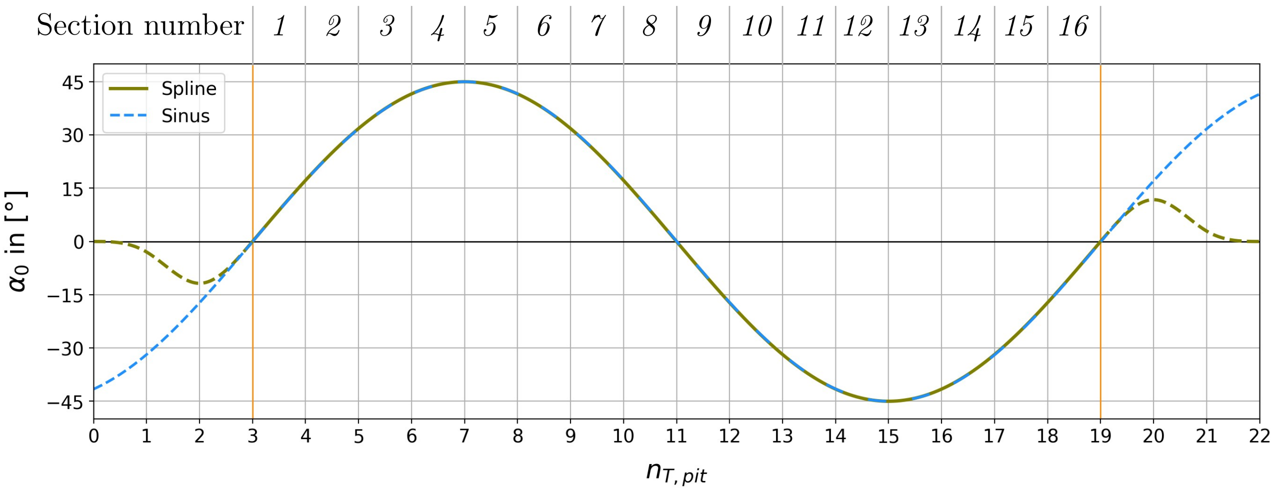


Figure 6.1: Complete curve of the pitching spline.

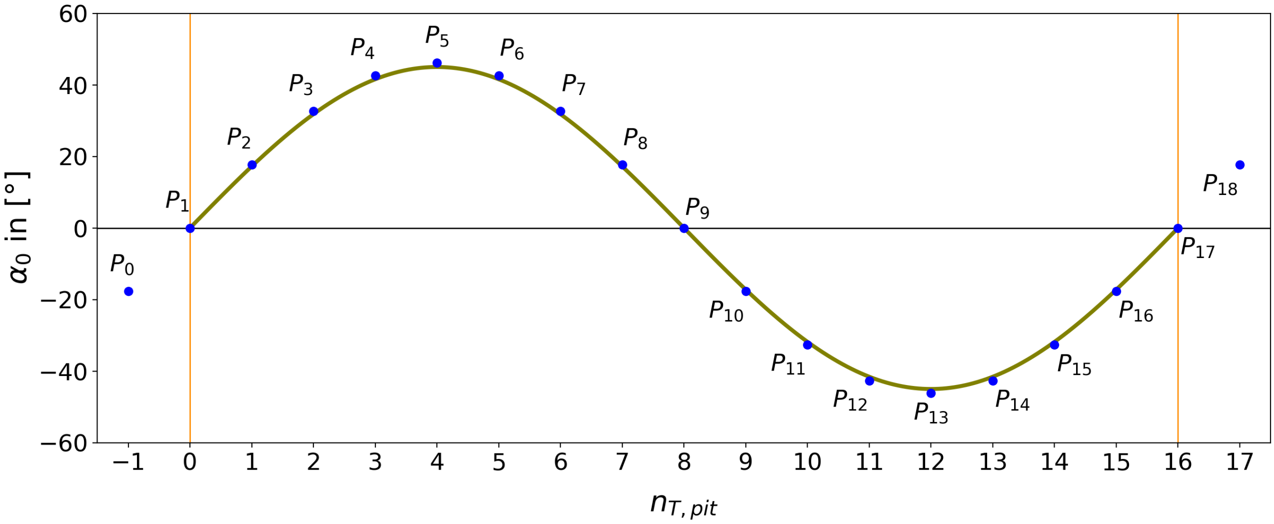


Figure 6.2: Complete curve of the pitching spline.

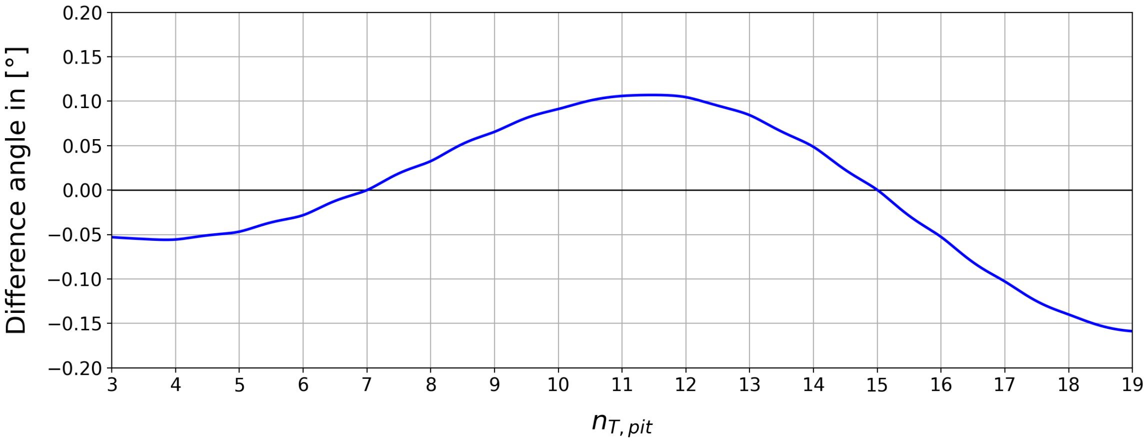


Figure 6.3: Difference between the sinus curve and the spline.

Figure 6.4 shows all basis functions $N_{1,4}(n_{T,pit})$ to $N_{19,4}(n_{T,pit})$ necessary for the pitching spline. They all have the same shape and are only shifted in relation to each other.

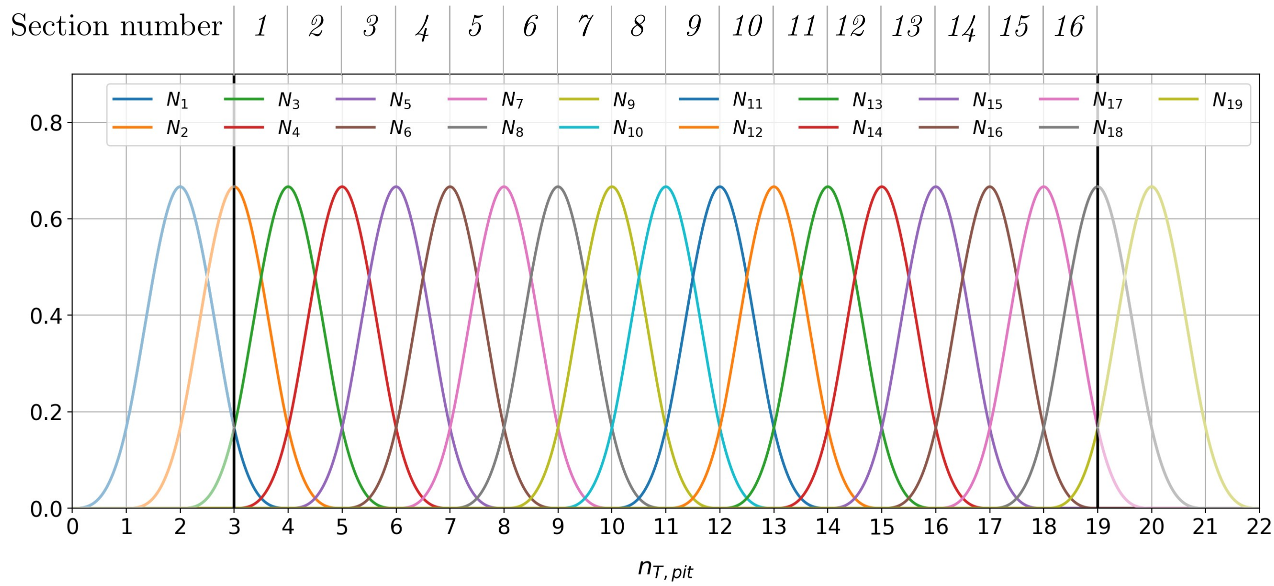


Figure 6.4: Basis functions for the pitching spline.

The following equation shows the carried out recurrence scheme of the first basis function $N_{1,4}(n_T)$, based on Equation (5.2). The subscript $_{pit}$ has been omitted for better readability.

For the first spline section *1*, between P_1 and P_2 and with $3 \leq n_{T,pit} \leq 4$, Equation (5.3) gives the last basis coefficients (the rightmost N of the scheme).

$$N_{4,1}(n_T) = 1, \qquad N_{i,1}(n_T) = 0 \quad \text{for} \quad i \neq 4$$

The resulting polynomial of $N_{1,4}(n_T)$ is

$$N_{1,4} = \frac{K_5 - n_T}{K_5 - K_2} \cdot \frac{K_5 - n_T}{K_5 - K_3} \cdot \frac{K_5 - n_T}{K_5 - K_4},$$

$$N_{1,4} = \frac{4 - n_T}{3} \cdot \frac{4 - n_T}{2} \cdot (4 - n_T),$$

$$N_{1,4} = -\frac{1}{6} \cdot n_T^3 + 2 \cdot n_T^2 - 8 \cdot n_T + \frac{32}{3}.$$

Analog the polynomial of $N_{2,4}(n_T)$ is

$$N_{2,4} = \frac{n_T - K_2}{K_5 - K_2} \cdot \frac{K_5 - n_T}{K_5 - K_3} \cdot \frac{K_5 - n_T}{K_5 - K_4} + \frac{K_6 - n_T}{K_6 - K_3} \cdot \left(\frac{n_T - K_3}{K_5 - K_3} \cdot \frac{K_5 - n_T}{K_5 - K_4} + \frac{K_6 - n_T}{K_6 - K_4} \cdot \frac{n_T - K_4}{K_5 - K_4} \right)$$

$$N_{2,4} = \frac{n_T - 1}{3} \cdot \frac{4 - n_T}{2} \cdot (4 - n_T) + \frac{5 - n_T}{3} \cdot \left(\frac{n_T - 2}{2} \cdot (4 - n_T) + \frac{5 - n_T}{2} \cdot (n_T - 3) \right)$$

$$N_{2,4} = \frac{1}{2} \cdot n_T^3 - \frac{11}{2} \cdot n_T^2 + \frac{39}{2} \cdot n_T - \frac{131}{6}$$

These polynomials are valid for the first spline section ($3 \leq n_{T,pit} \leq 4$).

$$
N_{1,4}(n_T) = \begin{cases}
\dfrac{n_T - K_1}{K_4 - K_1} \cdot N_{1,3}(n_T),\ N_{1,3}(n_T) = \begin{cases}
\dfrac{n_T - K_1}{K_3 - K_1} \cdot N_{1,2}(n_T),\ N_{1,2}(n_T) = \begin{cases}
\dfrac{n_T - K_1}{K_2 - K_1} \cdot N_{1,1}(n_T),\ N_{1,1}(n_T) = \begin{cases} 1 & \text{if } K_1 \leq t < K_2 \\ 0 & \text{otherwise} \end{cases} \\
+ \\
\dfrac{K_3 - t}{K_3 - K_2} \cdot N_{2,1}(n_T),\ N_{2,1}(n_T) = \begin{cases} 1 & \text{if } K_2 \leq t < K_3 \\ 0 & \text{otherwise} \end{cases}
\end{cases} \\
+ \\
\dfrac{K_4 - t}{K_4 - K_2} \cdot N_{2,2}(n_T),\ N_{2,2}(n_T) = \begin{cases}
\dfrac{n_T - K_2}{K_3 - K_2} \cdot N_{2,1}(n_T),\ N_{2,1}(n_T) = \begin{cases} 1 & \text{if } K_2 \leq t < K_3 \\ 0 & \text{otherwise} \end{cases} \\
+ \\
\dfrac{K_4 - t}{K_4 - K_3} \cdot N_{3,1}(n_T),\ N_{3,1}(n_T) = \begin{cases} 1 & \text{if } K_3 \leq t < K_4 \\ 0 & \text{otherwise} \end{cases}
\end{cases}
\end{cases} \\
+ \\
\dfrac{K_5 - t}{K_5 - K_2} \cdot N_{2,3}(n_T),\ N_{2,3}(n_T) = \begin{cases}
\dfrac{n_T - K_2}{K_4 - K_2} \cdot N_{2,2}(n_T),\ N_{2,2}(n_T) = \begin{cases}
\dfrac{n_T - K_2}{K_3 - K_2} \cdot N_{2,1}(n_T),\ N_{2,1}(n_T) = \begin{cases} 1 & \text{if } K_2 \leq t < K_3 \\ 0 & \text{otherwise} \end{cases} \\
+ \\
\dfrac{K_4 - t}{K_4 - K_3} \cdot N_{3,1}(n_T),\ N_{3,1}(n_T) = \begin{cases} 1 & \text{if } K_3 \leq t < K_4 \\ 0 & \text{otherwise} \end{cases}
\end{cases} \\
+ \\
\dfrac{K_5 - t}{K_5 - K_3} \cdot N_{3,2}(n_T),\ N_{3,2}(n_T) = \begin{cases}
\dfrac{n_T - K_3}{K_4 - K_3} \cdot N_{3,1}(n_T),\ N_{3,1}(n_T) = \begin{cases} 1 & \text{if } K_3 \leq t < K_4 \\ 0 & \text{otherwise} \end{cases} \\
+ \\
\dfrac{K_5 - t}{K_5 - K_4} \cdot N_{4,1}(n_T),\ N_{4,1}(n_T) = \begin{cases} 1 & \text{if } K_4 \leq t < K_5 \\ 0 & \text{otherwise} \end{cases}
\end{cases}
\end{cases}
\end{cases}
$$

The polynomials for the other spline sections have the form

$$N_{i,4} = a_{p,i} \cdot n_T^3 + b_{p,i} \cdot n_T^2 + c_{p,i} \cdot n_T + d_{p,i} \ ,$$

with i as the counter for the corresponding control vertex. The coefficients of the polynomials $a_{p,i}$, $b_{p,i}$, $c_{p,i}$ and $d_{p,i}$ change with every spline section, although the shape of the basis curves remains the same.

A closer look at the basis function curves in Figure 6.4 shows that each spline section between $3 \leq n_{T,pit} \leq 19$ consist of four different but repeating curves. Instead of calculating the basis functions for each spline section *1* to *16*, the pitching curve can be determined with only four polynomials N_1^p to N_4^p. These four curves are extracted and transformed into the range $0 \leq n_{T,pit} \leq 1$, see Figure 6.5.

The corresponding polynomials for these basis functions are

$$N_1^p = -\frac{1}{6} \cdot n_T^3 + \frac{1}{2} \cdot n_T^2 - \frac{1}{2} \cdot n_T + \frac{1}{6} \ , \tag{6.3}$$

$$N_2^p = \frac{1}{2} \cdot n_T^3 - 1 \cdot n_T^2 + \frac{2}{3} \ , \tag{6.4}$$

$$N_3^p = -\frac{1}{2} \cdot n_T^3 + \frac{1}{2} \cdot n_T^2 + \frac{1}{2} \cdot n_T + \frac{1}{6} \ , \tag{6.5}$$

$$N_4^p = \frac{1}{6} \cdot n_T^3 \ . \tag{6.6}$$

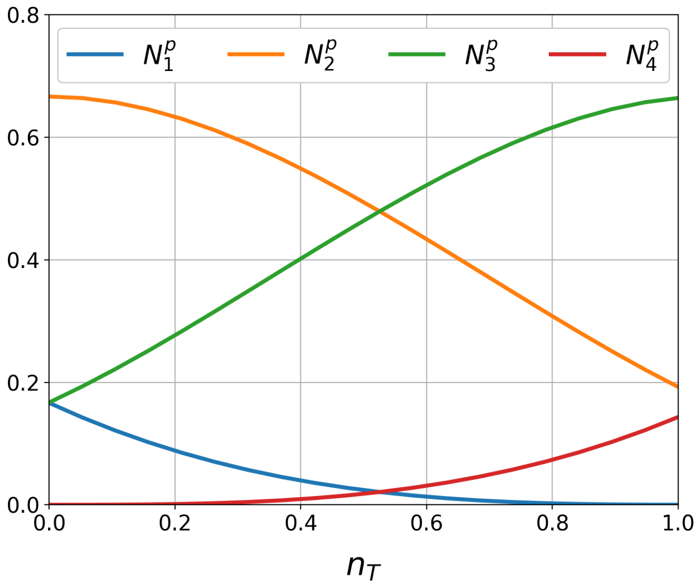


Figure 6.5: Transferred basis function curve.

The new equation for the pitching spline is now

$$\gamma_{pit}(n_T) = N_1^p(n_T) \cdot P_j + N_2^p(n_T) \cdot P_{j+1} + N_3^p(n_T) \cdot P_{j+2} + N_4^p(n_T) \cdot P_{j+3} \tag{6.7}$$

with the section counter j between *1* and *16*. Only the vertices have to be changed according to the section.

Section	P_i	P_{j+1}	P_{j+2}	P_{j+3}
1	P_0	P_1	P_2	P_3
2	P_1	P_2	P_3	P_4
3	P_2	P_3	P_4	P_5
$\vdots$				
16	P_{15}	P_{16}	P_{17}	P_{18}

The scalar values of the basis functions in Equation (6.7) can be interpreted as a weighting factor for each control vertex.

The advantage of the adjusted Equation (6.7) is that only a matrix of the coefficients of the four polynomials has to be saved in the code. The basic polynomial is

$$N_k^p = a_{p,k} \cdot n_T^3 + b_{p,k} \cdot n_T^2 + c_{p,k} \cdot n_T + d_{p,k} \ , \tag{6.8}$$

where k for the four basis fuctions, $k = [1{,}2{,}3{,}4]$. The corresponding coefficients for the pitching spline are

$$\mathbf{N}_{Pit} = \begin{bmatrix} -\frac{1}{6} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{6} \\ \frac{1}{2} & -1 & 0 & \frac{2}{3} \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} a_{p,1} & b_{p,1} & c_{p,1} & d_{p,1} \\ a_{p,2} & b_{p,2} & c_{p,2} & d_{p,2} \\ a_{p,3} & b_{p,3} & c_{p,3} & d_{p,3} \\ a_{p,4} & b_{p,4} & c_{p,4} & d_{p,4} \end{bmatrix} . \tag{6.9}$$

Please note, that there are two different control variables:

- The *outer* control variable $n_{T,pit}$, which ranges between 0 and 16 to define the position on the pitching spline.
- The *inner* control variable n_T, which ranges between 0 and 1 to calculate the current basis functions for each spline section.

Figure 6.6 shows the evolution for both control variables over two rotaion.

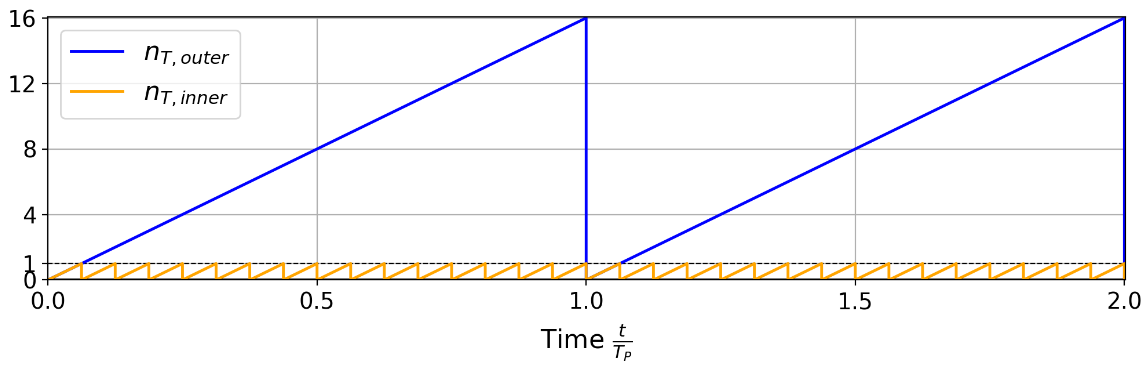


Figure 6.6: Inner and outer control variables for the pitching spline.

Figure 6.7 shows an example of an arbitrary pitching spline.

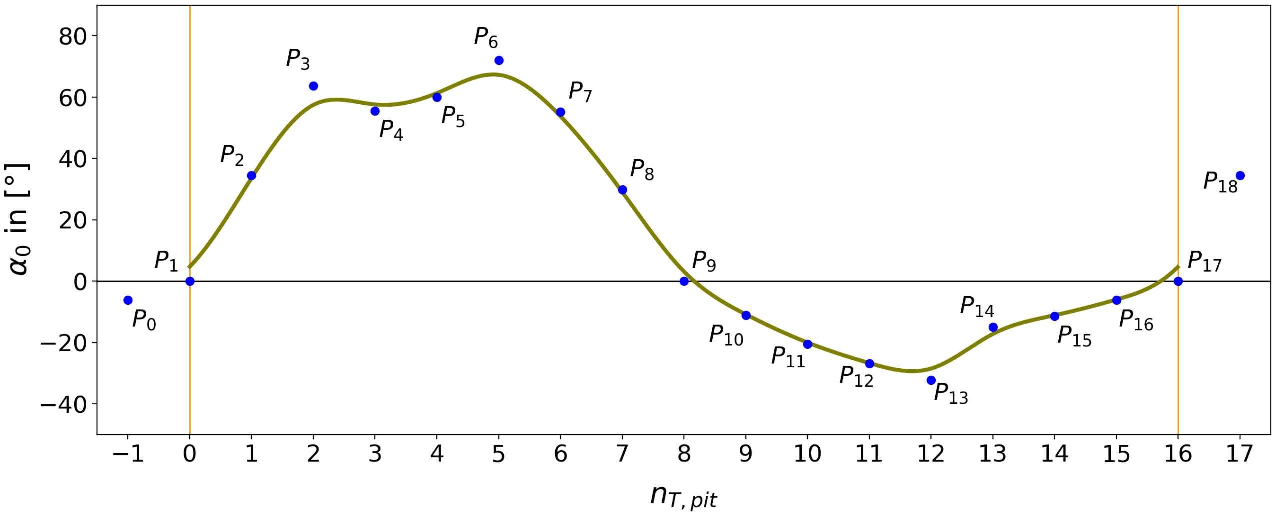


Figure 6.7: Example for an arbitrary pitching spline.

6.2 Trajectory

6.2.1 Path

The spline for the trajectory path consists of eight control vertices. In contrast to the pitching, the degree of the trajectory spline is four (degree $m_{tra} = 4$, order $k_{tra} = 5$). This high order is necessary to obtain continuous first and second derivatives of the spline and thus a continuous movement of the blade. By all means, the trajectory must be a closed spline. Therefore, five more multiple control vertices were added. The following equation gives the trajectory path.

$$\gamma_{tra}(n_{T,tra}) = \sum_{i=1}^{13} N_{i,5}(n_{T,tra}) \cdot V_{i,tra} \tag{6.10}$$

with its control vector

$$\underline{V}_{tra} = \ [\, T_7,\, T_8,\, T_1,\, T_2,\, T_3,\, T_4,\, T_5,\, T_6,\, T_7,\, T_8,\, T_1,\, T_2,\, T_3] \ . \tag{6.11}$$

The elements T_1 to T_8 of the vector $\underline{V}_{tra}$ contain the cartesian coordinates of the control vertices.

The knot vector is periodic with evenly spaced elements.

$$\kappa_{tra} = [\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ \ 10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 16,\ 17\ ]$$

To obtain a closed trajectory, the range of the spline's control variable is adjusted to

$$4.5 \ \leq \ n_{T,tra} \ \leq \ 12.5 \ .$$

The spline would end as a spiral at the origin for a control variable less than 4.5 or greater than 12.5.

Figure 6.8 shows an example of a circular spline.

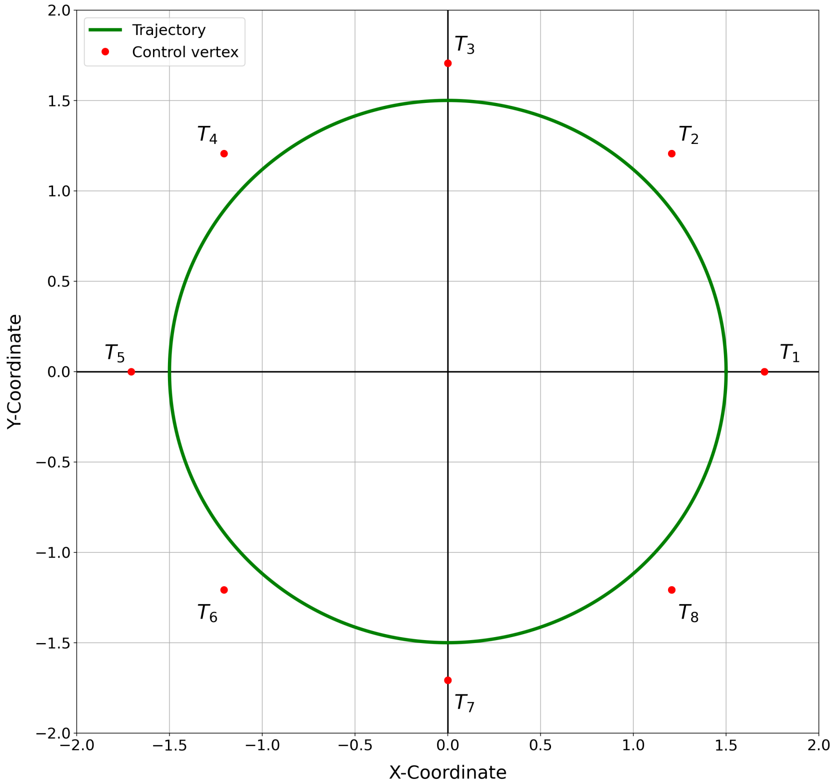


Figure 6.8: Closed circular b-spline.

A brief comment on the procedure for the CFD cases. Before a CFD case starts, there is a previous determination of the trajectory to get characteristic values (i.e. the length) and perform quality checks (i.e. intersection, curvature). For this procedure, a number of the trajectory coordinates are calculated. The number of these points influences the quality of the generated trajectory. The spline in Figure 6.8 consists of 1 000 points and looks like a circle. However, the resulting radius is slightly undulated, as shown in Figure 6.9. Therefore, the precalculation for the later optimisation run uses 100 000 points for high accuracy.

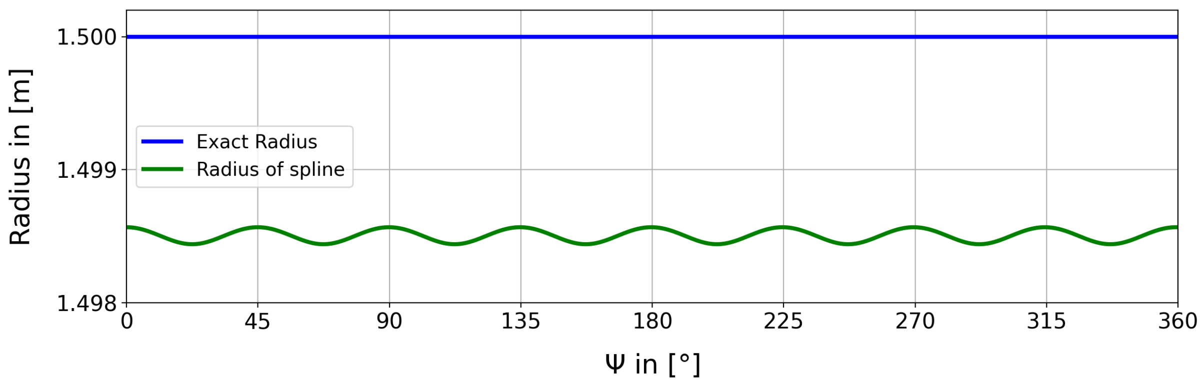


Figure 6.9: Radius of the spline trajectory, which consists of 1 000 points.

Figure 6.10 shows the curves of the basis function. Analog to the pitching spline, they all have the same shape and are only shifted in relation to each other.

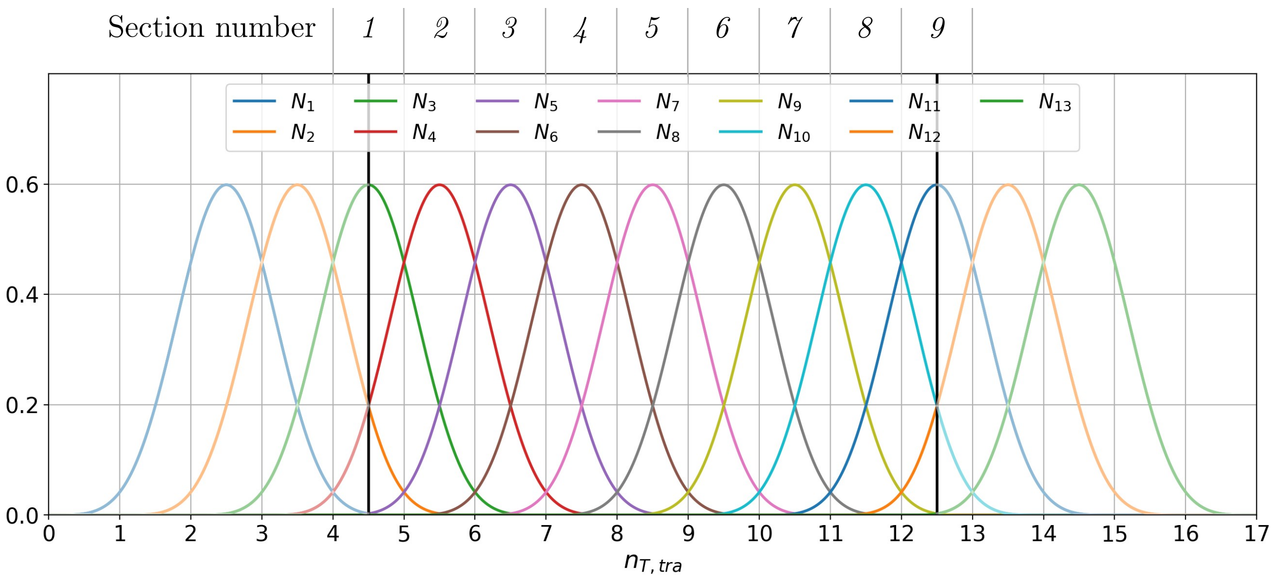


Figure 6.10: Basis functions for the trajectory spline.

Each spline section consits of five different basis curves, which are shown in Figure 6.11. The polynomials of these curves, now with a degree of five, are

$$N_1^t = \frac{1}{24} \cdot n_T^4 - \frac{1}{6} \cdot n_T^3 + \frac{1}{4} \cdot n_T^2 - \frac{1}{6} \cdot n_T + \frac{1}{24} \ , \tag{6.12}$$

$$N_2^t = -\frac{1}{6} \cdot n_T^4 + \frac{1}{2} \cdot n_T^3 - \frac{1}{4} \cdot n_T^2 - \frac{1}{2} \cdot n_T + \frac{11}{24} \ , \tag{6.13}$$

$$N_3^t = \frac{1}{4} \cdot n_T^4 - \frac{1}{2} \cdot n_T^3 - \frac{1}{4} \cdot n_T^2 + \frac{1}{2} \cdot n_T + \frac{11}{24} \ , \tag{6.14}$$

$$N_4^t = -\frac{1}{6} \cdot n_T^4 + \frac{1}{6} \cdot n_T^3 + \frac{1}{4} \cdot n_T^2 + \frac{1}{6} \cdot n_T + \frac{1}{24} \ , \tag{6.15}$$

$$N_5^t = \frac{1}{24} \cdot n_T^4 \ . \tag{6.16}$$

The *inner* control variable for the trajectory basis function ranges from 0 to 1. The coefficients of the polynomials are stored in the matrix

$$\mathbf{N}_{tra} = \begin{bmatrix} \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \\ -\frac{1}{6} & \frac{1}{2} & -\frac{1}{4} & -\frac{1}{2} & \frac{11}{24} \\ \frac{1}{4} & -\frac{1}{2} & -\frac{1}{4} & \frac{1}{2} & \frac{11}{24} \\ -\frac{1}{6} & \frac{1}{6} & \frac{1}{4} & \frac{1}{6} & \frac{1}{24} \\ \frac{1}{24} & 0 & 0 & 0 & 0 \end{bmatrix} . \tag{6.17}$$

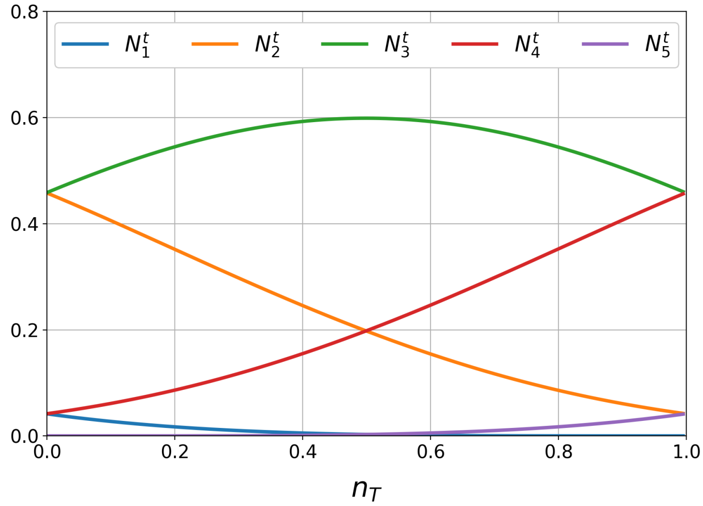


Figure 6.11: Transferred basis functions for trajectory spline.

With the five essential basis functions, the equation for the trajectory spline is simplified.

$$\gamma_{tra}(n_T) = N_1^t(n_T) \cdot T_j + N_2^t(n_T) \cdot T_{j+1} + N_3^t(n_T) \cdot T_{j+2} + N_4^t(n_T) \cdot T_{j+3} + N_5^t(n_T) \cdot T_{j+4} \quad (6.18)$$

with the section counter j between *1* and *9*.

Only the control vertices have to be changed according to the section.

Section	T_i	T_{j+1}	T_{j+2}	T_{j+3}	T_{j+4}
1	T_7	T_8	T_1	T_2	T_3
2	T_8	T_1	T_2	T_3	T_4
3	T_1	T_2	T_3	T_4	T_5
$\vdots$					
9	T_7	T_8	T_1	T_2	T_3

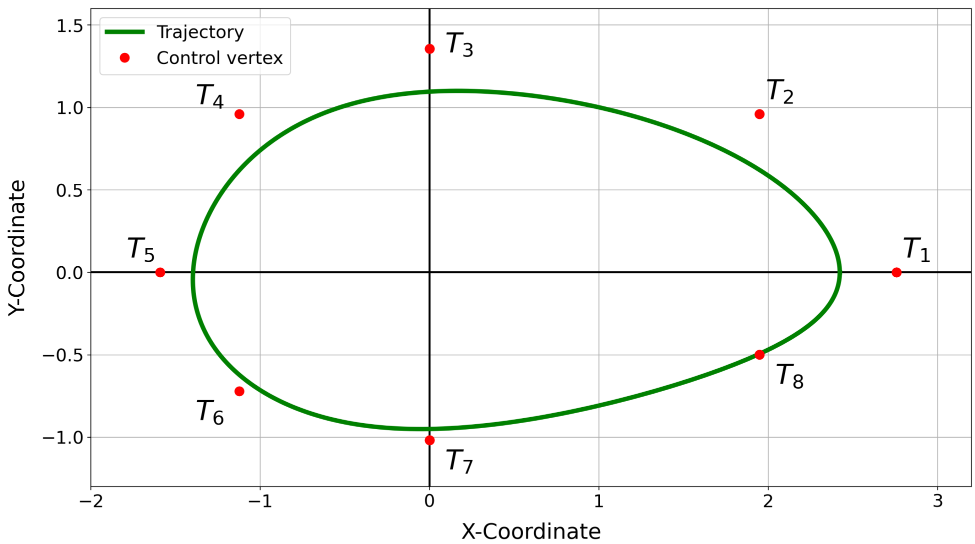


Figure 6.12: Example for an arbitrary trajectory.

6.2.2 Counter angle

In the simulation, the blades' motion is a superposition of the trajectory path and the pitching. The coordinates from Equation (6.18) induce the blade to perform a translation but would remain in its initial alignment, i.e. vertically for `airfoil0` (see Figure 3.3). To force the blade into a tangential trajectory, the slope of the trajectory must be determined. The first derivative of the spline gives the tangent vector.

$$\begin{aligned}\gamma'_{tra}(n_T) &= \begin{pmatrix} X'_{tra}(n_T) \\ Y'_{tra}(n_T) \end{pmatrix} \\ &= N'_{1,5}(n_T)\cdot T_i + N'_{2,5}(n_T)\cdot T_{i+1} + N'_{3,5}(n_T)\cdot T_{i+2} + N'_{4,5}(n_T)\cdot T_{i+3} + N'_{5,5}(n_T)\cdot T_{i+4}\end{aligned} \tag{6.19}$$

The basic derivation of the function polynomials is

$$N'_{*,5} = 4\cdot n_{N,*1}\cdot n_T^3 + 3\cdot n_{N,*2}\cdot n_T^2 + 2\cdot n_{N,*3}\cdot n_T + n_{N,*4}\ . \tag{6.20}$$

The resulting basis functions are:

$$N'_{1,5} = -\frac{1}{6}\cdot n_T^3 + \frac{1}{4}\cdot n_T^2 - \frac{1}{6}\cdot n_T + \frac{1}{24}\ , \tag{6.21}$$

$$N'_{2,5} = +\frac{1}{2}\cdot n_T^3 - \frac{1}{4}\cdot n_T^2 - \frac{1}{2}\cdot n_T + \frac{11}{24}\ , \tag{6.22}$$

$$N'_{3,5} = -\frac{1}{2}\cdot n_T^3 - \frac{1}{4}\cdot n_T^2 + \frac{1}{2}\cdot n_T + \frac{11}{24}\ , \tag{6.23}$$

$$N'_{4,5} = +\frac{1}{6}\cdot n_T^3 + \frac{1}{4}\cdot n_T^2 + \frac{1}{6}\cdot n_T + \frac{1}{24}\ , \tag{6.24}$$

$$N'_{5,5} = \frac{1}{24}\cdot n_T^3\ . \tag{6.25}$$

Figure 6.13 shows the first derivatives of the basis functions.

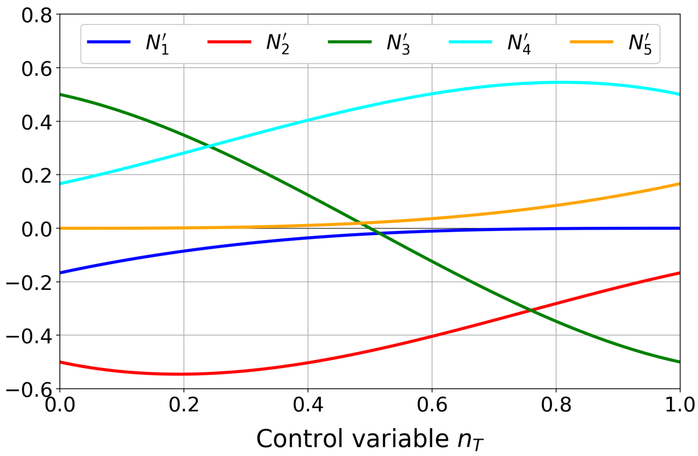


Figure 6.13: First derivatives of basis functions.

The matrix $\mathbf{N'}_{tra}$ contains the coefficient of the derivative functions.

$$\mathbf{N'}_{tra} = \begin{bmatrix} \frac{1}{6} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{6} \\ -\frac{4}{6} & \frac{3}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 1 & -\frac{3}{2} & -\frac{1}{2} & \frac{1}{2} \\ -\frac{4}{6} & \frac{1}{2} & \frac{1}{2} & \frac{1}{6} \\ \frac{1}{6} & 0 & 0 & 0 \end{bmatrix} . \tag{6.26}$$

The necessary angle of the slope or the counter angle φ is

$$\varphi = \tan\left(\frac{Y'_{tra}(n_T)}{X'_{tra}(n_T)}\right) . \tag{6.27}$$

6.2.3 Velocity

Depending on the shape of the spline, the distance between two neighbouring points is not constant despite a linearly increasing control variable n_T. Figure 6.14 shows an example trajectory with unequal distances, depending on the splines' curvature.

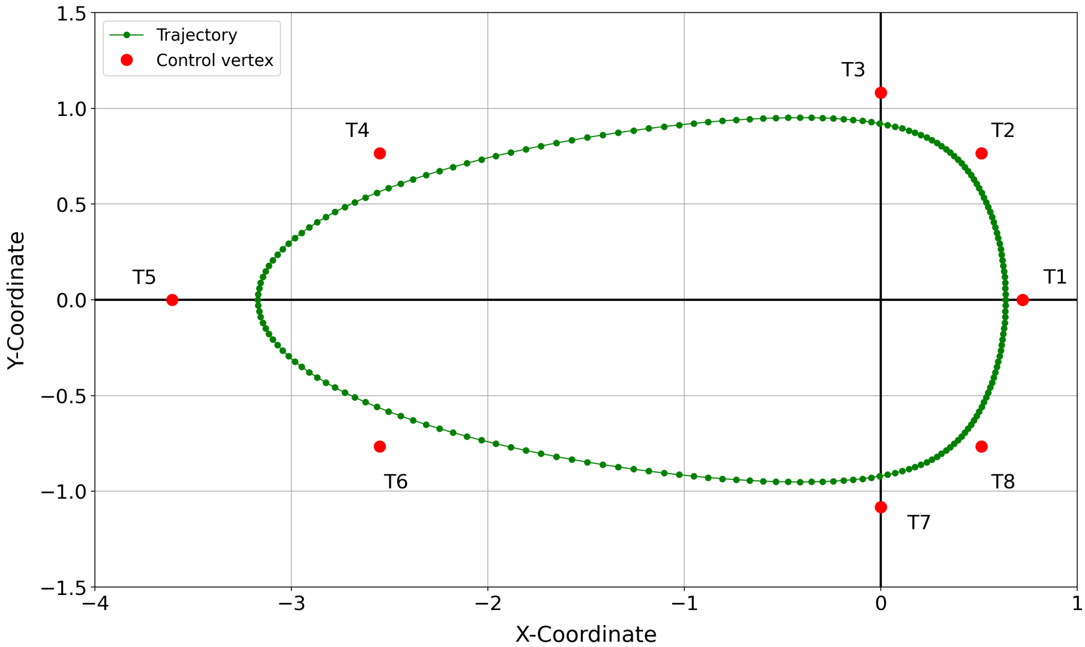


Figure 6.14: Example for a possible trajectory with unequal distance between the trajectory points.

Linear coupling between the simulation time t_{sim} and the control variable n_T would result in a varying velocity during the rotation. For the given example, this coupling means a higher velocity between the vertices T_3 and T_4 compared to the velocity between T_8 and T_2. Figure 6.15 shows the velocity over one rotation, which is not acceptable.

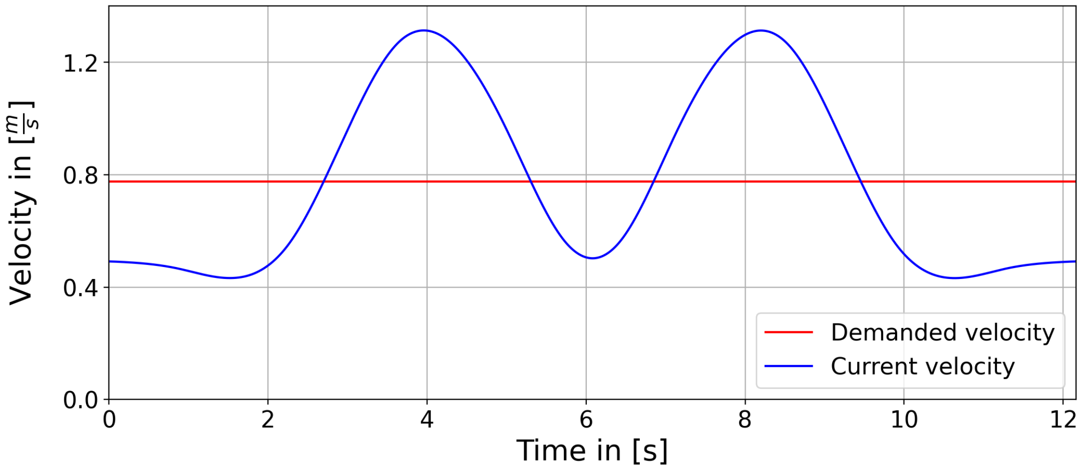


Figure 6.15: Velocity of the blade over time.

Therefore, the control variable must be calculated so that a constant velocity results. A constant velocity requires a linear increase of the trajectory's length. Figure 6.16 shows the length over time for the example trajectory above, which is visibly not linear.

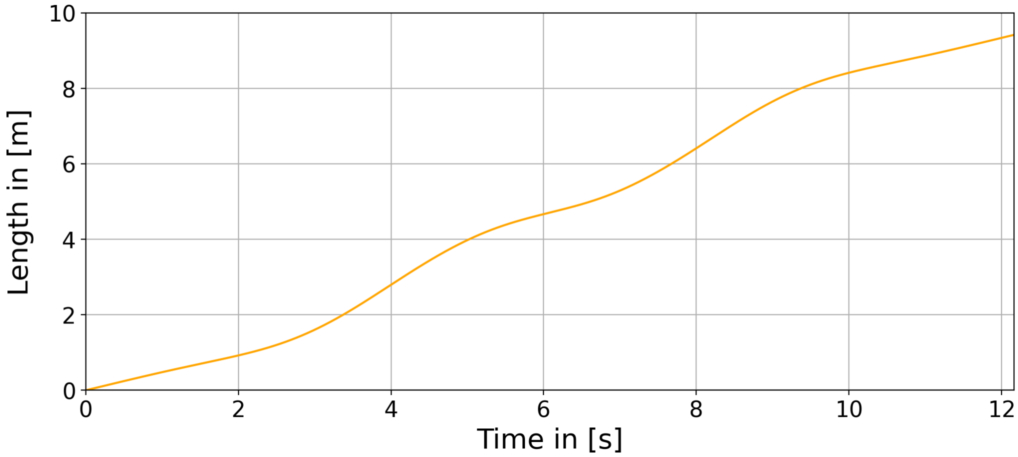


Figure 6.16: Length of the trajectory over time.

A blade moving with a constant velocity covers a certain length within a specific time.
The required length is given by

$$L_{req}(t_{sim}) = L_{tra} \cdot \frac{t_{sim}}{T_P} = 3\pi \cdot \frac{t_{sim}}{T_P} \ , \tag{6.28}$$

with L_{tra} as the total length of the trajectory, which must always be 3π and period time T_P, which depends on the Reynolds number (see section 4.1). For a given length L_{req}, the corresponding control variable can be taken from the plot in Figure 6.17.

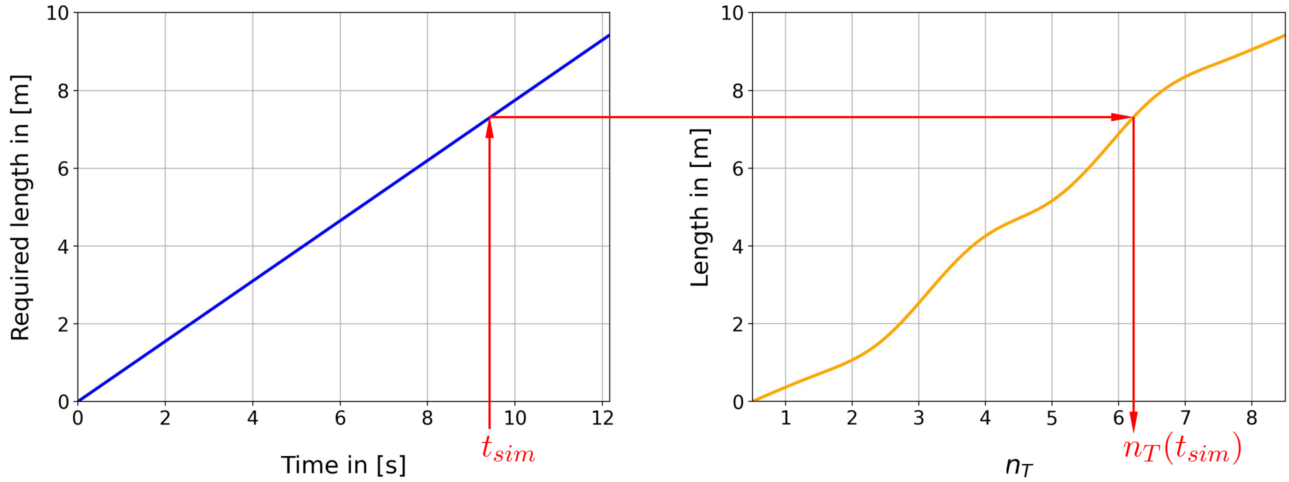
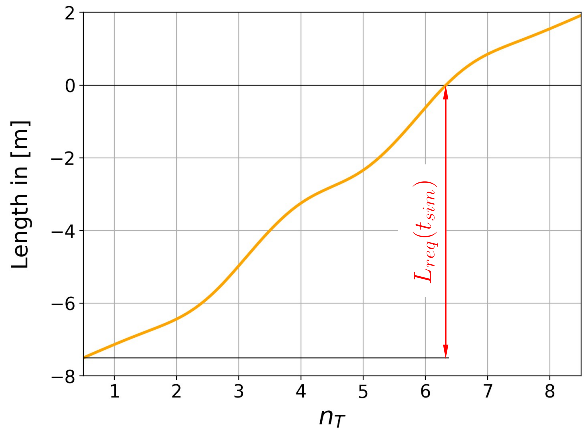
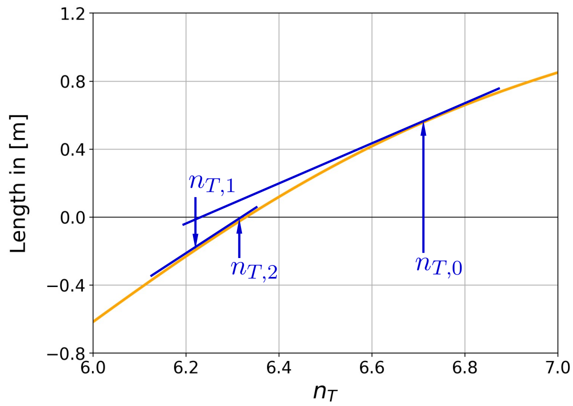


Figure 6.17: Procedure to determine $n_{T,tra}$.

However, a mathematical expression for the length like $L_{tra}(n_T)$ is unavailable (see Section 5.3). Therefore, yet another spline shall represent the length curve. To obtain the control variable n_T, the Newton–Raphson method is used to find the solution Figure 6.17. This method will converge as the length is strictly monotonic increasing. Because the Newton–Raphson method is a search for zero, the length curve needs to be shifted about the required length L_{req}, see 6.18(a). The zero of the length then corresponds to the sought variable n_T, as shown in 6.18(b).



(a) Shift of the length.



(b) Newton–Raphson method.

Figure 6.18: G.

The numerical procedure is as follows (Papula [19]).

$$n_{T,j+1} = n_{T,j} - \frac{L_{tar}(n_{T,j})}{L'_{tar}(n_{T,j})} \,. \tag{6.29}$$

The $L_{tar}(n_{T,j})$ is the length curve within a specific section, where $L'_{tar}(n_{T,j})$ is the corresponding first derivative. A tolerance criterion checks the control variables' accuracy. If the criterion reaches a limit, the approach aborts.

$$\Delta n_T = |n_{T,j+1} - n_{T,j}| \ < 10^{-5} \tag{6.30}$$

The Newton–Raphson method needs a starting point $n_{T,0}$. Linear coupling between the simulation time t_{sim} and the control variable n_T is chosen for the first guess.

According to Section 5.4, the curve fitting procedure provides the essential length spline γ_{len}. The following equation is carried out with the selected set of data points and the basis function of the curve fitting.

$$\underline{V}_{len} = \mathbf{N}_{len}^{-1} \cdot \underline{D}_{len} \tag{6.31}$$

Where $\underline{V}_{len}$ is the control vector, $\mathbf{N}_{len}^{-1}$ the inverse coefficient matrix and $\underline{D}_{len}$ the data vector. The control vector will be calculated in a preliminary step before the CFD calculation and is a further input for the OpenFOAM function (see next chapter).

The degree of the length spline γ_{len} is set to four ($m_{len} = 4$, order $k_{len} = 5$) to ensure an adequate representation of the length curve. The number of data points shall be at least sixteen (twice as much as the control vertices for the trajectory). Additional data points are needed to ensure a periodic spline. An open knot vector is chosen for the curve fitting.

As a result of these requirements, the range of the control variable $n_{T,len}$ lays no longer between 0.5 and 8.5. The range depends on the knot vector κ_{len} of the new length spline. An alternative approach could have been to transform the knot vector to map the trajectory's knot vector κ_{tra}. However, this would lead to knot elements with fractions, and the range of the spline sections would not be integers. Therefore, a knot vector, which consists only of integers according to the Equation (5.5), is used to avoid these numerical uncertainties.

Due to the changing range of the control variable, a transformation from $n_{T,len}$ into $n_{T,tra}$ is required.

$$n_{T,tra} = 0.5 + (n_{T,len} - n_{min}) \cdot \frac{8}{n_{max}} \; , \tag{6.32}$$

with n_{min} as the starting point and n_{max} as the ending point of the corresponding length spline, see Figure 6.19.

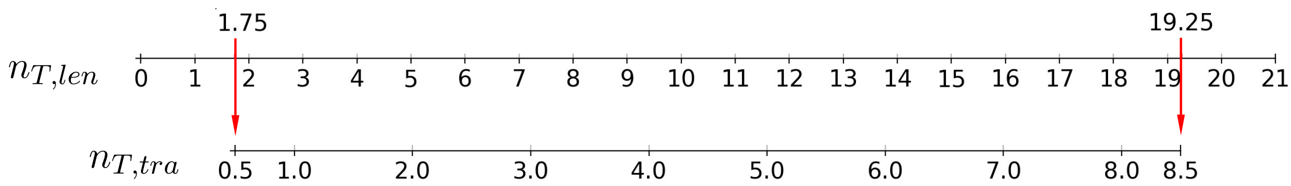


Figure 6.19: Transformation of the control variable.

The preliminary calculation of the trajectory is carried out with 100 000 steps, which corresponds to $\Delta n_{T,tra} = 8 \cdot 10^{-5}$. It follows that the data points for the curve fitting are only available for a finite control variable. It must be ensured that n_{min}, as well as n_{max}, are rational numbers. Indeed, this simple problem depends on the number of data points influencing the knot vector. Nevertheless, the distribution of the data points must also be taken into account to avoid oscillation of the length spline. A sufficient setup is 25 data points with $n_{min} = 1.75$ and $n_{max} = 19.25$. This leads to the following knot vector :

$$\kappa_{len} = [\, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 21, 21, 21, 21 \,] \, .$$

The range of the control variable is

$$1.75 \; \leq \; n_{T,len} \; \leq 19.25 \; . \tag{6.33}$$

The transformation between these two variables is

$$n_{T,tra} = 0.5 + (n_{T,len} - 1.75) \cdot \frac{8}{17.5} \ . \tag{6.34}$$

As a result of the open knot vector, the basis functions have a different shape compared to the previous splines (trajectory or pitching). But the functions can be reduced to five basic forms, coloured in Figure 6.18. The basic functions $N_{len,22}$ to $N_{len,25}$ are symmetrical with respect to the origin. The basic functions $N_{len,5}$ to $N_{len,21}$ are mirror-symmetrical.

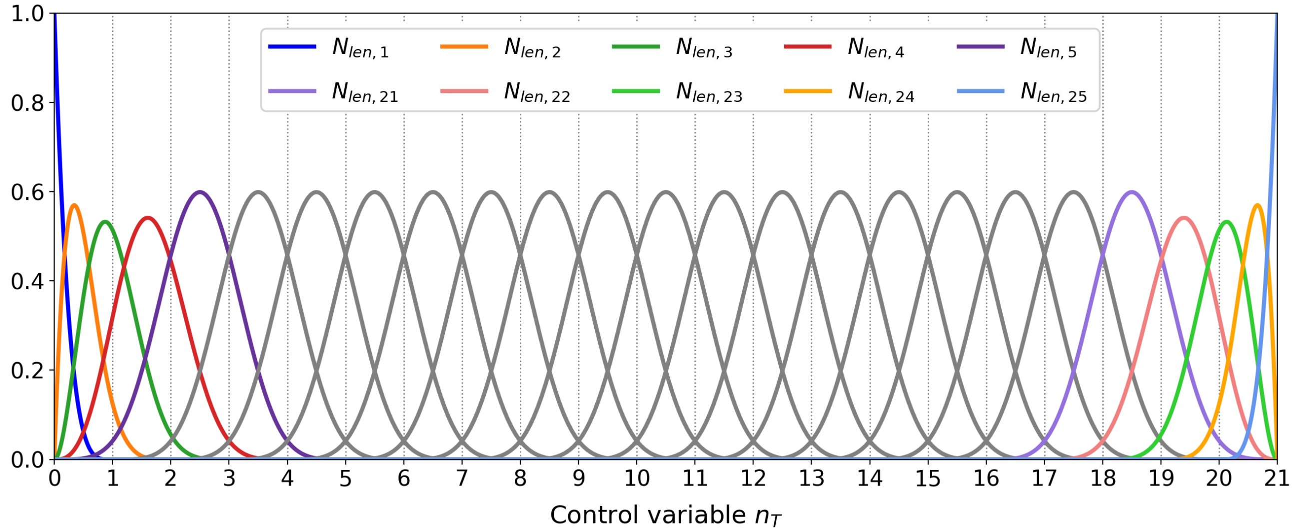


Figure 6.20: Basis functions for the length spline γ_{len}.

Analog to the previous approach, the basis functions are polynomial and valid in the range $0 \leq n_{T,len} \leq 1$. The basic form of the function polynomial is

$$N_{bf,*} = n_{bf,**1} \cdot n_{T,len}^4 + n_{bf,**2} \cdot n_{T,len}^3 + n_{bf,**3} \cdot n_{T,len}^2 + n_{bf,**4} \cdot n_{T,len} + n_{bf,**5} \ , \tag{6.35}$$

with $n_{bf,**1}$ as the element of the corresponding coefficient matrix $\mathbf{N}_{bf,1}$. The first star '*' indicates the number of the following matrices, and the second star indicates the corresponding row.

$$\mathbf{N}_{bf,1} = \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \end{bmatrix} , \tag{6.36}$$

$$\mathbf{N}_{bf,2} = \begin{bmatrix} -\frac{15}{8} & 7 & -9 & 4 & 0 \\ \frac{1}{8} & -\frac{1}{2} & \frac{3}{4} & -\frac{1}{2} & \frac{1}{8} \end{bmatrix} , \tag{6.37}$$

$$\mathbf{N}_{bf,3} = \begin{bmatrix} \frac{85}{72} & -\frac{11}{3} & 3 & 0 & 0 \\ -\frac{23}{72} & \frac{19}{18} & -\frac{11}{12} & -\frac{5}{18} & \frac{37}{72} \\ \frac{1}{18} & -\frac{2}{9} & \frac{1}{3} & -\frac{2}{9} & \frac{1}{18} \end{bmatrix} , \tag{6.38}$$

$$\mathbf{N}_{bf,4} = \begin{bmatrix} -\frac{25}{72} & \frac{2}{3} & 0 & 0 & 0 \\ \frac{23}{72} & -\frac{13}{18} & -\frac{1}{12} & \frac{11}{18} & \frac{23}{72} \\ -\frac{13}{72} & \frac{5}{9} & -\frac{1}{3} & -\frac{4}{9} & \frac{4}{9} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{4} \end{bmatrix}, \tag{6.39}$$

$$\mathbf{N}_{bf,5} = \begin{bmatrix} \frac{1}{24} & 0 & 0 & 0 & 0 \\ -\frac{1}{6} & \frac{1}{6} & \frac{1}{4} & \frac{1}{6} & \frac{1}{24} \\ \frac{1}{4} & -\frac{1}{2} & -\frac{1}{4} & \frac{1}{2} & \frac{11}{24} \\ -\frac{1}{6} & \frac{1}{2} & -\frac{1}{4} & -\frac{1}{2} & \frac{11}{24} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{4} \end{bmatrix}. \tag{6.40}$$

It must be noted that the coefficient matrix for the whole length spline is a combination of the matrices $\mathbf{N}_{bf,1}$ to $\mathbf{N}_{bf,5}$. The final matrix for the length spline $\mathbf{N}_{len}$ has the dimension 25x25.

From now on, the Newton Raphson method can be applied. For each section, only a part of the length spline is used.

$$\gamma_{len,sec}(n_T) = \mathbf{N}_{sec,1}(n_T)\cdot L_i + \mathbf{N}_{sec,2}(n_T)\cdot L_{i+1} + \mathbf{N}_{sec,3}(n_T)\cdot L_{i+2} + \mathbf{N}_{sec,41}(n_T)\cdot L_{i+3} + \mathbf{N}_{sec,5}*(n_T)\cdot L_{i+4}\ , \tag{6.41}$$

The vector $\underline{V}_{len}$ stores the control vertices L_i.

The coefficient matrix $\mathbf{N}_{sec}$ for each section is a combination of the matrices $\mathbf{N}_{bf,1}$ to $\mathbf{N}_{bf,5}$. For example, the coefficient matrix $\mathbf{N}_{len,1}$ for the first spline section, $0 \leq n_T \leq 1$, is

$$\mathbf{N}_{len,1} = \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ -\frac{15}{8} & 7 & -9 & 4 & 0 \\ \frac{85}{72} & -\frac{11}{3} & 3 & 0 & 0 \\ -\frac{25}{72} & \frac{2}{3} & 0 & 0 & 0 \\ \frac{1}{24} & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} N_{bf,1\ r=1} \\ N_{bf,2\ r=2} \\ N_{bf,3\ r=3} \\ N_{bf,4\ r=4} \\ N_{bf,5\ r=5} \end{bmatrix},$$

with r as the column counter of the matrix.

The coefficient matrix for the spline section *3* is as follows

$$\mathbf{N}_{len,3} = \begin{bmatrix} \frac{1}{18} & -\frac{2}{9} & \frac{1}{3} & -\frac{2}{9} & \frac{1}{18} \\ -\frac{13}{72} & \frac{5}{9} & -\frac{1}{3} & -\frac{4}{9} & \frac{4}{9} \\ \frac{1}{4} & -\frac{1}{2} & -\frac{1}{4} & \frac{1}{2} & \frac{11}{24} \\ -\frac{1}{6} & \frac{1}{6} & \frac{1}{4} & \frac{1}{6} & \frac{1}{24} \\ \frac{1}{24} & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} N_{bf,3\ r=3} \\ N_{bf,4\ r=3} \\ N_{bf,5\ r=3} \\ N_{bf,5\ r=2} \\ N_{bf,5\ r=1} \end{bmatrix}.$$

The coefficient matrix for spline sections *5* to *17* consists only of the $N_{bf,5}$. From section *18* onwards, it is necessary to mirror the basis function $N_{len,1} \ldots N_{len,3}$ to get $N_{len,23} \ldots N_{len,25}$.

A separate algorithm determines the essential sub matrix $\mathbf{N}_{sec,*}$. For that, a tensor $\mathbf{M}_{len}$ summarises the matrices $\mathbf{N}_{bf,1}$ to $\mathbf{N}_{bf,5}$ (see Figure 6.21). The tensors's dimension is 5x5x5. Rows that do not exist in the $\mathbf{N}_{bf,*}$ are filled with a zero line.

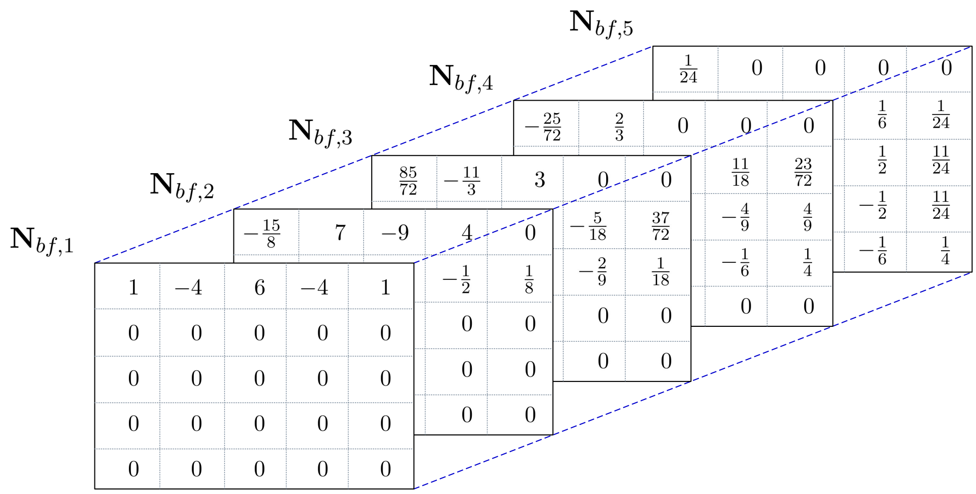


Figure 6.21: Tensor $\mathbf{M}_{len}$.

An algorithm compounds the submatrix $\mathbf{N}_{sec}$ from the tensor $\mathbf{M}_{len}$ according to a predefined specification. The specification defines the order of the coefficient matrices $\mathbf{N}_{bf,*}$ listed in the order matrix $\mathbf{I}_O$ and the corresponding row number listed in the row matrix $\mathbf{I}_R$. Each row of the two specification matrices $\mathbf{I}_O$ and $\mathbf{I}_R$ corresponds to a specific section of the length spline γ_{len}.

$$\mathbf{I}_O = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 4 \\ 2 & 3 & 4 & 4 & 4 \\ 3 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 3 \\ 4 & 4 & 4 & 3 & 2 \\ 4 & 4 & 3 & 2 & 1 \\ 4 & 3 & 2 & 1 & 0 \end{bmatrix} \qquad \mathbf{I}_R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 2 & 2 & 2 & 1 & 0 \\ 3 & 3 & 2 & 1 & 0 \\ 4 & 3 & 2 & 1 & 0 \\ 4 & 3 & 2 & 1 & 3 \\ 4 & 3 & 2 & 2 & 2 \\ 4 & 3 & 1 & 1 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad (6.42\quad \text{a, b})$$

For example, the coefficient matrix for the spline section *3* can be written as follows

$$\mathbf{N}_{len,3} = \begin{bmatrix} \frac{1}{18} & -\frac{2}{9} & \frac{1}{3} & -\frac{2}{9} & \frac{1}{18} \\ -\frac{13}{72} & \frac{5}{9} & -\frac{1}{3} & -\frac{4}{9} & \frac{4}{9} \\ \frac{1}{4} & -\frac{1}{2} & -\frac{1}{4} & \frac{1}{2} & \frac{11}{24} \\ -\frac{1}{6} & \frac{1}{6} & \frac{1}{4} & \frac{1}{6} & \frac{1}{24} \\ \frac{1}{24} & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{M}_{len,2,2} \\ \mathbf{M}_{len,3,2} \\ \mathbf{M}_{len,4,2} \\ \mathbf{M}_{len,4,1} \\ \mathbf{M}_{len,4,0} \end{bmatrix} = \begin{bmatrix} \mathbf{M}_{len}[\mathbf{I}_{O,sec,1}][\mathbf{I}_{Row,sec,1}] \\ \mathbf{M}_{len}[\mathbf{I}_{O,sec,2}][\mathbf{I}_{Row,sec,2}] \\ \mathbf{M}_{len}[\mathbf{I}_{O,sec,3}][\mathbf{I}_{Row,sec,3}] \\ \mathbf{M}_{len}[\mathbf{I}_{O,sec,4}][\mathbf{I}_{Row,sec,4}] \\ \mathbf{M}_{len}[\mathbf{I}_{O,sec,5}][\mathbf{I}_{Row,sec,5}] \end{bmatrix}.$$

The sub-matricessubmatrix $\mathbf{N}_{sec}$ for the spline sections *5* to *17* are identical, represented by the fifth row of $\mathbf{I}_O$ and $\mathbf{I}_R$.

The mirroring of the basis function would lead to new coefficients of the polynomials. To avoid this issue, the control variable $n_{T,len}$ was transformed for polynomials $N_{len,17}$ to $N_{len,25}$.

$$n_{T,mirrow} = 1 - n_{T,len} \tag{6.43}$$

At last, the basis functions for the first derivative are needed for the Newton-Raphson method. The basic equation with the already known matrix elements $n_{bf,**}$ is

$$N'_{bf,*} = 4 \cdot n_{bf,**1} \cdot n_T^3 + 3 \cdot n_{bf,**2} \cdot n_T^2 + 2 \cdot n_{bf,**3} \cdot n_T + n_{bf,**4} \ . \tag{6.44}$$

The polynomials are mirrored by multiplying them by minus one.

$$N'_{bf,17} = -1 \cdot N'_{bf,4}$$

To perform the Newton-Raphson method, it is necessary to know in which section of the length spline the current blade is located (see Figure 6.12). Thus, a different vector is defined, containing each section's length.

$$\underline{S}_{len} = [\ L_{tra}|_{n_T=1}, \quad L_{tra}|_{n_T=2}, \quad L_{tra}|_{n_T=3}, \quad \ldots \quad L_{tra}|_{n_T=20}\ ] \tag{6.45}$$

In summary, the Newton-Raphson method needs the following input data.

Seven matrices, which remain constant:

- the basis function coefficient matrix $\mathbf{N}_{bf,1}\ \ldots\ \mathbf{N}_{bf,5}$
- the order matrix $\mathbf{I}_O$,
- the row matrix $\mathbf{I}_R$.

And for each unique trajectory

- the control vector $\underline{V}_{len}$,
- the section vector $\underline{S}_{len}$.

As mentioned in the previous section 4.2, there are two modes of drive. The blades' velocity is no longer constant for a constant rotational speed drive. The previous procedure is still applicable with a few changes to obtain a constant angular velocity. Instead of the length, the azimuth angle Ψ over time is used and shall increase linearly.

The required azimuth angle according to the simulation time is

$$\Psi_{req}(t_{sim}) = 2\pi \cdot \frac{t_{sim}}{T_P} \ . \tag{6.46}$$

The azimuth angle over time is calculated with the values of the trajectory,

$$\Psi_{tra}(n_T) = \tan\left(\frac{Y_{tra}(n_T)}{X_{tra}(n_T)}\right) \ . \tag{6.47}$$

7 Implementation

The spline functions described in the previous chapter are implemented in the OpenFOAM functions. The basis for the implementation is the `rotatingMotion` class provided by the software.

Generally, these functions are written in `C++` and consist of two files. The header file with the ending `*.H` contains the declaration. The other file has ending `*.C` and contains the actual function codes. All files can be found in the Appendix E.

First, a few code snippets will be explained, which are used by both motion functions. After that, the implementation of the splines is explained in detail. Finially The naming of the variables or parameters in the code differs from those used for the derivation in Chapter 6. To keep the context, both names are mentioned in the text, where the code names are written in `typewriter font`.

7.1 Code snippets

The initial position of the blades does not necessarily have to match the specified pitching angle or trajectory position in Figure 7.4. The software moves the blade in the first step to its calculated position. However, this significant movement within a single time step would lead to a disturbed flow field and numerical artefacts, which can influence the results. A factor is applied to the motion to avoid such jumps, ensuring a smooth transition from the initial position to the final movement. The user sets a delay, a fraction of a rotation, to determine how long the lag lasts.

$$\text{Duration} = \text{Delay} \cdot \text{Period} \tag{7.1}$$

The factor, called lag, increases from zero to one within the duration (see Figure 7.2)

$$\text{Lag} = \frac{1}{2} \cdot \left(1 - \cos\left(\frac{\text{Time}}{\text{Duration}} \cdot \pi\right)\right) \quad . \tag{7.2}$$

```
    scalar lag_ = 1;
    scalar duration_ = delay_ * period_;
    if (time_.value()  < duration_ )
        {
            lag_ = (1 - cos(time_.value() / duration_ * pi ) ) / 2;
        }
```

This function is implemented in the standard OpenFOAM class `rotatingMotion` to achieve delayed circular motion. The new class `delayRotatingMotion.C` can be found in Chapter 6.

The motion of the blade, pitching as well as trajectory, is determined by the simulation time t_{sim}. For multi-blade cases, an internal time t_{int} is defined to consider the varying blade positions (see Figure 3.3). The simulation time receives an offset, which is the fraction of lead azimuth angle, $\Delta_{\text{offset}} = 0.25$ for 90° blade lead, $\Delta_{\text{offset}} = \frac{1}{3}$ for 120° blade lead and so on.

$$t_{int} = t_{sim} + \Delta_{\text{offset}} \cdot T_P \ . \tag{7.3}$$

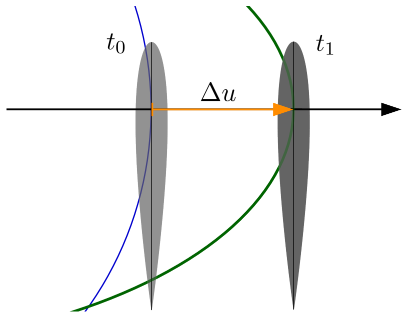


Figure 7.1: Movement between initial position t_1 and final position on the trajectory t_2.

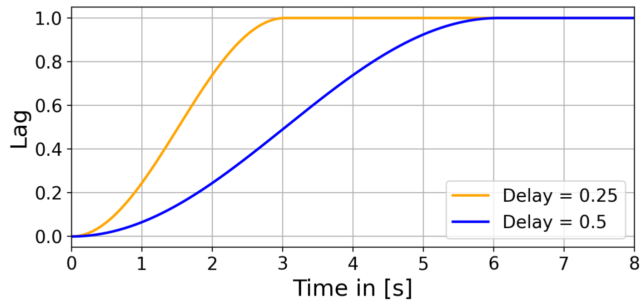


Figure 7.2: Two lags with different delay values.

This time then determines the control variable, which, in turn, defines the motion.

```
scalar t_ = time_.value() + offset_ * period_;
```

The simulations will be carried out with plenty of rotations. However, the range of the spline is limited to the range of the control variable. A counter for the number of rotations n_{Rot} is defined to enable multiple revolutions.

$$n_{Rot} = floor\left(\frac{t_{int}}{T_P}\right) \tag{7.4}$$

```
scalar nRot_ = floor(t_ / period_);
```

7.2 Pitching

The code `bSplinePitching.C` enables arbitrary pitching and will be explained in the following.

The pitching spline consists of 16 sections. The counter for the current section is

$$k = floor\left(\frac{t_{int} - T_P \cdot n_{Rot}}{T_P} \cdot 16\right) \ . \tag{7.5}$$

```
int k = floor(( t_ - period_ * nRot_ ) / period_ * 16);
```

The control variable $n_{T,pit}$ is directed by the simulation time; however, it must range from zero to one in each section.

$$n_{T,pit} = 16 \cdot \frac{t_{int}}{T_P} - k - n_{Rot} \cdot 16 \tag{7.6}$$

```
    scalar nT_=  16 * t_ / period_ - k - nRot_ * 16;
```

Figure 7.3 shows the behaviour of the variables over one and a half rotation.

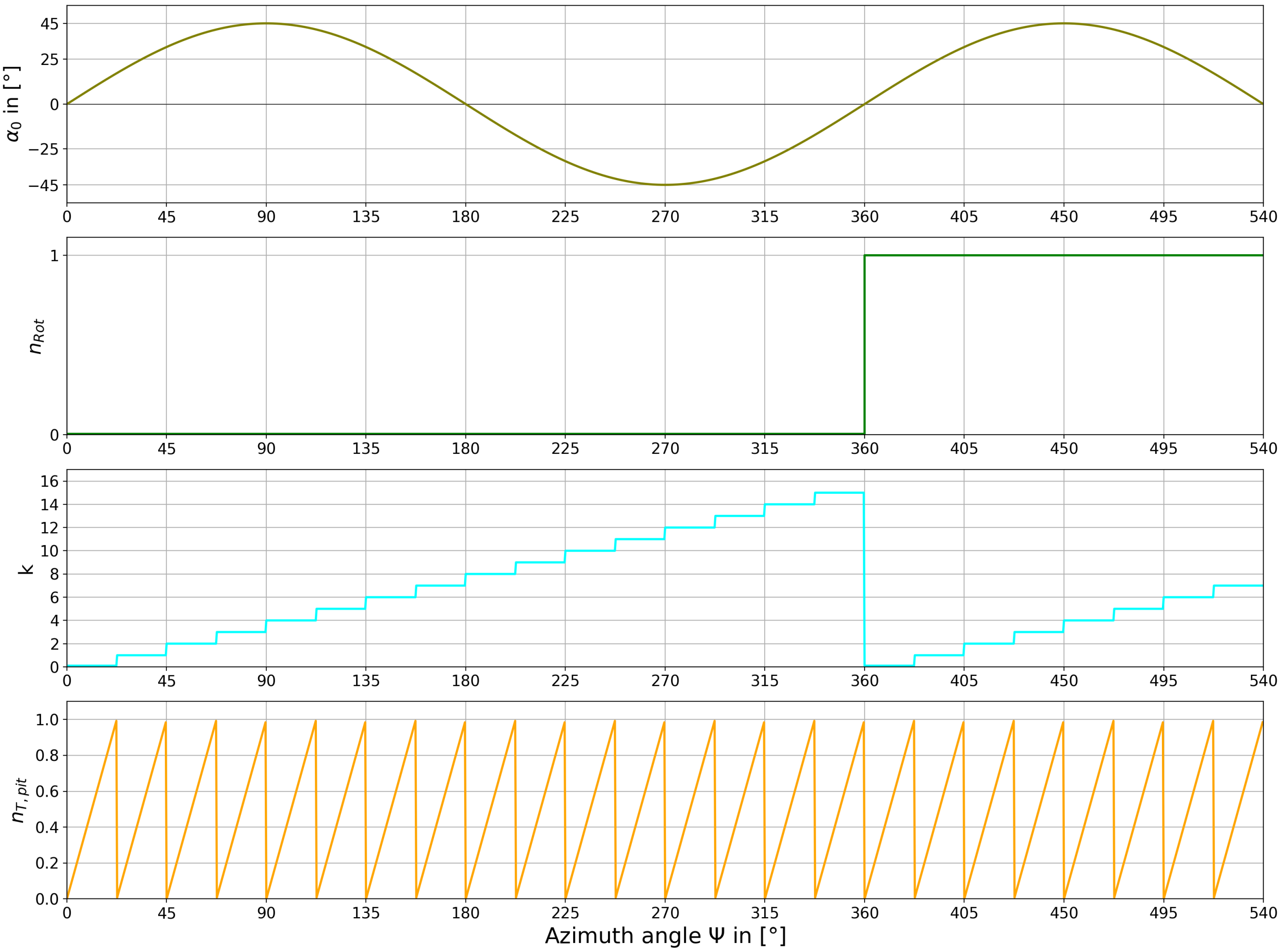


Figure 7.3: Path for different variable over one and a half rotation.

The next step is to calculate the basis functions. The matrix of Equation (6.9) with its coefficients is predefined in the `bSplinePitching.H` as the array `bFactor_`.

```
        double bFactor_[4][4]
        {
            {-1.0/6, 1.0/2, -1.0/2, 1.0/6}, // N1
            {1.0/2, -1, 0, 2.0/3}, // N2
            {-1.0/2, 1.0/2, 1.0/2, 1.0/6}, // N3
            {1.0/6, 0, 0, 0} // N4
        };
```

Equation (6.3) to Equation (6.6) are then implemented, where `R1_` corresponds to N_1^p.

```
scalar R1_ =  (bFactor_[0][0]*pow(nT_,3)+bFactor_[0][1]*pow(nT_,2)+bFactor_[0][2]*nT_+bFactor_[0][3]);
scalar R2_ =  (bFactor_[1][0]*pow(nT_,3)+bFactor_[1][1]*pow(nT_,2)+bFactor_[1][2]*nT_+bFactor_[1][3]);
scalar R3_ =  (bFactor_[2][0]*pow(nT_,3)+bFactor_[2][1]*pow(nT_,2)+bFactor_[2][2]*nT_+bFactor_[2][3]);
scalar R4_ =  (bFactor_[3][0]*pow(nT_,3)+bFactor_[3][1]*pow(nT_,2)+bFactor_[3][2]*nT_+bFactor_[3][3]);
```

The last calculation is the pitching angle with the lag and conversion from degree to radian.

```
scalar angle_ = - ( R1_*Polygon_[k]+R2_*Polygon_[k+1]+R3_*Polygon_[k+2]+R4_*Polygon_[k+3] ) * pi / 180.0 * lag_;
```

The `Polygon_[]` contains the control variables according to Equation (6.2).

This value is given to the two OpenFOAM classes `quaternion()` and `septernion()`, which perform translations and rotations in 3D space. Finally, the code passes the results to the solver.

7.3 Trajectory

The code `bSplineMotion.C` enables arbitrary trajectory and will be explained in the following.

The required length, according to Equation (6.28) is

$$L_{req}(t_{sim}) = \frac{t_{int} - T_P \cdot n_{Rot}}{T_P} \cdot L_{end}. \tag{7.7}$$

The value L_{end} is an input value, which depends on the drive mode; $L_{end} = 3\pi$ for the constant velocity and $L_{end} = 2\pi$ for constant angular velocity.

```
scalar Lreq = (t_ - period_ * nRot_ ) / period_ * endValue_;
```

Newton–Raphson method

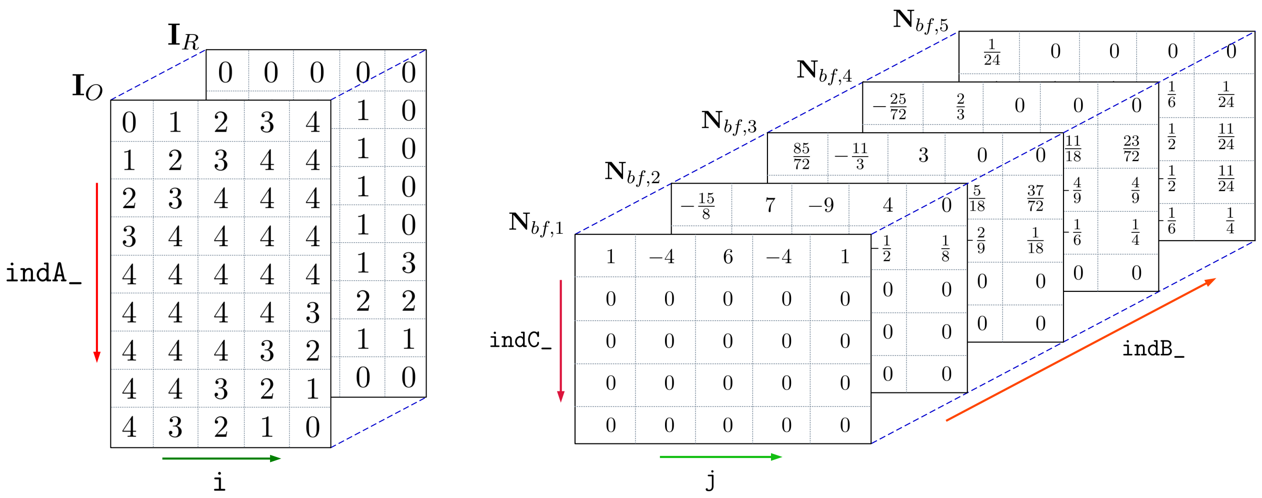
For the Newton–Raphson method, the corresponding section number *sec* (`m`) of the length spline γ_{len} has to be determined by comparing the current length with the length for each section.

```
int m = 1;
bool flagSec = false;
while (flagSec == false)
{
    if (Lreq < sectionL_[m])
    {
        flagSec = true;
    }
    else
    {
        m++;
    }
}
```

The `for` loop in row `120 - 123` combines the basic coefficients $\mathbf{N}_{bf,*}$, which are stored in the tensor $\mathbf{M}_{len}$ (`Length_[][][]`) to obtain the sub-matrix $\mathbf{N}_{sec,*}$ (`matR_`).

The index `indB_` defines the page of the multidimensional array, which corresponds to the predefined order of the $\mathbf{N}_{bf,*}$ given by the matrix $\mathbf{I}_O$. The index `indC_` defines the row of the multidimensional array, which corresponds to the matrix $\mathbf{I}_R$. Both specification matrices $\mathbf{I}_O$ and $\mathbf{I}_R$ are stored in the array `Index_[][][]`. The correct row of the specification matrices is determined by the index `indA_`, which corresponds to the section. The basis function for the section between 5 and 17 are equal. Therefore section counter `indA_` remains 4.

Figure 7.4 shows the assignment of arrays and their indices.



(a) `Index_[][][]` (b) `Length_[][][]`

Figure 7.4: Assignment for the matrices.

```
97      double matR_[5][5];
98      int indA_ = 0;
99      int indB_ = 0;
100     int indC_ = 0;
101
102     if (m < 4)
103     {
104         indA_ = m;
105     }
106     else if (m < 17)
107     {
108         indA_ = 4;
109     }
110     else
111     {
112         indA_ = m - 12;
113     }
114
115     for (int i = 0; i < 5; i++)
116     {
117         indB_ = Index_[0][indA_][i];
118         indC_ = Index_[1][indA_][i];
119
```

```
        for (int j = 0; j < 5; j++)
        {
            matR_[i][j]=Length_[indB_][indC_][j];
        }
    }
```

The starting value $n_{T,0}$ (`nL_`) for the Newton-Raphson method is derived from the internal time, transferred into the length spline range and then truncated into the final range of $0 \leq n_{T,len} \leq 1$.

```
    scalar nL_ = 1.75 + (t_ - period_ * nRot_ ) / period_ * 17.5;
    nL_ = nL_ - floor(nL_);
```

A `while` loop carries out the numerical solution for $n_{T,len}$ until the difference `diffT` between the two steps is equal o or less than 10^{-5}.

Within the loop, the basis functions `LR`* and their first derivative `LdR`* of the length spline are calculated for each of the five control vertices. The control variable is transformed depending on the spline section `m`, and a negative sign is set to consider the mirrored basis functions and their derivatives, respectively, according to Equation (6.43) (`code line 142, 146, 150`). A reverse transformation for the control variable `nL_` has to be done before the actual numerical scheme is carried out (`code line 157`).

```
134     while (diffT > 1e-5)
135     {
136         scalar LR1 = matR_[0][0]*pow(nL_,4) + matR_[0][1]*pow(nL_,3) + matR_[0][2]*pow(nL_,2) + matR_[0][3]*nL_
            ↪  + matR_[0][4];
137         scalar LdR1 = 4* matR_[0][0]*pow(nL_,3) + 3* matR_[0][1]*pow(nL_,2) + 2* matR_[0][2]*nL_ + matR_[0][3];
138 
139         scalar LR2 = matR_[1][0]*pow(nL_,4) + matR_[1][1]*pow(nL_,3) + matR_[1][2]*pow(nL_,2) + matR_[1][3]*nL_
            ↪  + matR_[1][4];
140         scalar LdR2 = 4* matR_[1][0]*pow(nL_,3) + 3* matR_[1][1]*pow(nL_,2) + 2* matR_[1][2]*nL_ + matR_[1][3];
141 
142         if (m >= 19) { nL_ = 1 - tNew; sign = -1; }
143         scalar LR3 = matR_[2][0]*pow(nL_,4) + matR_[2][1]*pow(nL_,3) + matR_[2][2]*pow(nL_,2) + matR_[2][3]*nL_
            ↪  + matR_[2][4];
144         scalar LdR3 = sign * ( 4* matR_[2][0]*pow(nL_,3) + 3* matR_[2][1]*pow(nL_,2) + 2* matR_[2][2]*nL_ +
            ↪  matR_[2][3] );
145 
146         if (m >= 18) { nL_ = 1 - tNew; sign = -1; }
147         scalar LR4 = matR_[3][0]*pow(nL_,4) + matR_[3][1]*pow(nL_,3) + matR_[3][2]*pow(nL_,2) + matR_[3][3]*nL_
            ↪  + matR_[3][4];
148         scalar LdR4 = sign * ( 4* matR_[3][0]*pow(nL_,3) + 3* matR_[3][1]*pow(nL_,2) + 2* matR_[3][2]*nL_ +
            ↪  matR_[3][3] );
149 
150         if (m >= 17) { nL_ = 1 - tNew; sign = -1; }
151         scalar LR5 = matR_[4][0]*pow(nL_,4) + matR_[4][1]*pow(nL_,3) + matR_[4][2]*pow(nL_,2) + matR_[4][3]*nL_
            ↪  + matR_[4][4];
152         scalar LdR5 = sign * ( 4* matR_[4][0]*pow(nL_,3) + 3* matR_[4][1]*pow(nL_,2) + 2* matR_[4][2]*nL_ +
            ↪  matR_[4][3] );
153 
154         L = LR1*vectorL_[m]+LR2*vectorL_[m+1]+LR3*vectorL_[m+2]+LR4*vectorL_[m+3]+LR5*vectorL_[m+4] - Lreq; //
            ↪  function for Length-over-nT minus Lreq, vertical transformation
155         dL = LdR1*vectorL_[m]+LdR2*vectorL_[m+1]+LdR3*vectorL_[m+2]+LdR4*vectorL_[m+3]+LdR5*vectorL_[m+4];
```

```
        if (m >= 17) { nL_ =  tNew; sign = 1; }

        tNew = nL_ - L / dL;
        diffT = fabs(tNew - nL_);
        nL_ = tNew;
    }
```

Coordinates

According to the Equation (6.32), the resulting n_T (`nL_`) must be transformed into the valid range of the trajectory ($0.5 \leq n_T \leq 8.5$).

```
    nL_ = 0.5 + (m + nL_ - 1.75)  * 8.0 / 17.5;
```

As a result of numerical uncertainties, the control variable could lay neglectable below 0.5 or above 8.5. This aberration would lead to a false calculation of the blades' position. The n_T (`nL_`) is limited to avoid this issue.

```
    if (nL_ < 0.5)
    {
        nL_ = 0.5;
    }
    else if (nL_ > 8.5)
    {
        nL_ = 8.5;
    }
```

With the section counter `k` for the trajectory, the n_T (`nL_`) is transformed in the range between zero and one.

```
    int k = floor(nL_);
    scalar nT_ = nL_  - k;
```

Then, the five basis functions $N_{1,5}$ ... $N_{5,5}$ (`R1_ ... R5_`) of the trajectory are determined. The vector `displacement` stores the resulting coordinates.

Due to the implementation of the septernion in OpenFOAM, the coordinates have to be given relative to the blades' initial position. Thus, the corresponding blade's initial coordinates `origin_.x()` and `origin_.y()` have to be subtracted.

```
    scalar R1_ = bFactor_[0][0]*pow(nT_,4) + bFactor_[0][1]*pow(nT_,3) + bFactor_[0][2]*pow(nT_,2) +
    ↪   bFactor_[0][3]*nT_ + bFactor_[0][4];  // N_1.5
    scalar R2_ = bFactor_[1][0]*pow(nT_,4) + bFactor_[1][1]*pow(nT_,3) + bFactor_[1][2]*pow(nT_,2) +
    ↪   bFactor_[1][3]*nT_ + bFactor_[1][4];  // N_2.5
```

```
    scalar R3_ = bFactor_[2][0]*pow(nT_,4) + bFactor_[2][1]*pow(nT_,3) + bFactor_[2][2]*pow(nT_,2) + bFactor_[2][3]*nT_ + bFactor_[2][4];  // N_3.5
    scalar R4_ = bFactor_[3][0]*pow(nT_,4) + bFactor_[3][1]*pow(nT_,3) + bFactor_[3][2]*pow(nT_,2) + bFactor_[3][3]*nT_ + bFactor_[3][4];  // N_4.5
    scalar R5_ = bFactor_[4][0]*pow(nT_,4) + bFactor_[4][1]*pow(nT_,3) + bFactor_[4][2]*pow(nT_,2) + bFactor_[4][3]*nT_ + bFactor_[4][4];  // N_5.5

    displacement.x() = ( ( R1_*Polygon_[k][0] + R2_*Polygon_[k+1][0] + R3_*Polygon_[k+2][0] + R4_*Polygon_[k+3][0] + R5_*Polygon_[k+4][0] ) - origin_.x() ) * lag_;
    displacement.y() = ( ( R1_*Polygon_[k][1] + R2_*Polygon_[k+1][1] + R3_*Polygon_[k+2][1] + R4_*Polygon_[k+3][1] + R5_*Polygon_[k+4][1] ) - origin_.y() ) * lag_;
```

Counter angle

During the simulation, the blades' motion is a superposition of the trajectory path and the pitching.

The first derivatives of the basis functions $N'_{1,1}$ to $N'_{1,5}$ (`dR1_ ... dR5_`) are calculated along with the components of the slope vector (`tangentX_` and `tangentY_`). Based on the slope vector, the counter angle φ (`phi`) is determined.

```
    scalar dR1_ = dFactor_[0][0]*pow(nT_,3) + dFactor_[0][1]*pow(nT_,2) + dFactor_[0][2]*nT_ + dFactor_[0][3];  // N'_1.5
    scalar dR2_ = dFactor_[1][0]*pow(nT_,3) + dFactor_[1][1]*pow(nT_,2) + dFactor_[1][2]*nT_ + dFactor_[1][3];  // N'_2.5
    scalar dR3_ = dFactor_[2][0]*pow(nT_,3) + dFactor_[2][1]*pow(nT_,2) + dFactor_[2][2]*nT_ + dFactor_[2][3];  // N'_3.5
    scalar dR4_ = dFactor_[3][0]*pow(nT_,3) + dFactor_[3][1]*pow(nT_,2) + dFactor_[3][2]*nT_ + dFactor_[3][3];  // N'_4.5
    scalar dR5_ = dFactor_[4][0]*pow(nT_,3) + dFactor_[4][1]*pow(nT_,2) + dFactor_[4][2]*nT_ + dFactor_[4][3];  // N'_5.5

    scalar tangentX_ = (dR1_*Polygon_[k][0] + dR2_*Polygon_[k+1][0] + dR3_*Polygon_[k+2][0] + dR4_*Polygon_[k+3][0] + dR5_*Polygon_[k+4][0]);
    scalar tangentY_ = (dR1_*Polygon_[k][1] + dR2_*Polygon_[k+1][1] + dR3_*Polygon_[k+2][1] + dR4_*Polygon_[k+3][1] + dR5_*Polygon_[k+4][1]);

    scalar phi = - atan2(tangentX_, tangentY_);
```

The output range of the arc tangent 2 lies between -π and π. For a blade transition from the second to the third quadrant, there is a jump of 2π for the counter angle. In addition, after each completed rotation, the counter angle starts from 0° all over again. These discontinuities lead to a blade flip and generate numerical artefacts or crash the case.

There are two functions to ensure a steady increase of the counter angle. The first one shifts the negative part of φ up to the positive section by adding a value of 2π. The range of the counter angle is now: $0 \leq \varphi \leq 2\pi$. A particular condition occurs for the blade `blade0`. Due to the slope of the trajectory, the counter angle may be negative at the initial position, which shall not be interpreted as a discontinuity. The 2π shift is skipped for the first half rotation and concerns only the `blade0`.

```
    scalar multiplier_ = 1;
    if (time_.value() < 0.25*period_ && t_ < 0.25*period_)
    {
        multiplier_ = 0;
    }

    if (phi < 0)
    {
        phi = phi + 2.0*pi *multiplier_ ;
    }
```

The second function also adds a value of 2π at a particular time to obtain a continuous increase of the counter angle. The necessary offset is 2π multiplied by the number of rotation n_{Rot} (nRot_). Nevertheless, there are two different conditions which need to be considered.

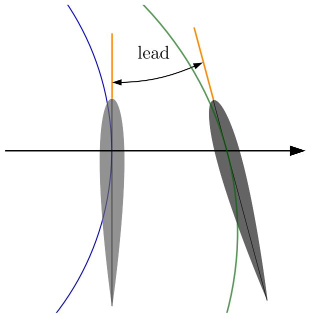
The initial position of every blade mesh is tangential to the circle (see Figure 3.3). However, the slope of the actual trajectory may differ from the circle's slope, which could result in a lead or a lag angle.

If there is a lead, the counter angle reaches 2π, although a complete rotation has not yet been accomplished. This circumstance occurs when the counter angle is positive (phi > 0) and the control variable $n_{T,len}$ (nL_) is greater than 7.0 (as a reminder: one full rotation results for $n_{T,len} = 8.5$). Therefore, an extra shift of 2π has to be added before the rotation counter n_{Rot} (nRot_) gets incremented by one.

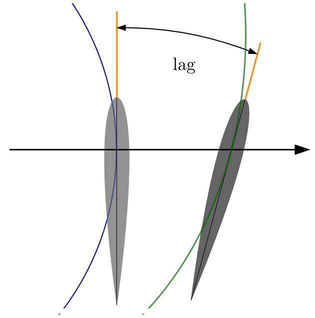
In contrast, the counter angle does not reach 2π in the case of a lag, even though the rotation is not yet completed. As a result, a shift must not be added. This circumstance occurs when the counter angle is negative (phi < 0) and the control variable $n_{T,len}$ (nL_) is smaller than 2.0 (as a reminder: every rotation starts from $n_{T,len} = 0.5$).

Figure 7.5 shows the behaviour of the counter angle for a lead and a lag angle.

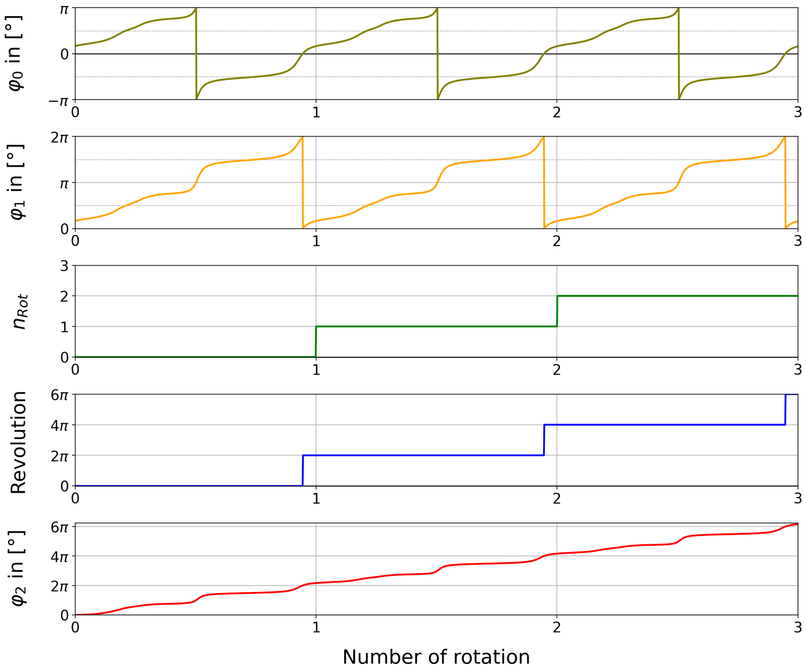
```
    scalar Revolution = 0;
    if (phi > 0 && nL_ > 7.0)
    {
        Revolution = (nRot_ + 1) * 2*pi;
    }

    else if (nRot_ > 0)
    {
        if (phi < 0 && nL_ < 2.0)
        {
            Revolution = (nRot_ -1) * 2*pi;
        }
        else
        {
            Revolution = nRot_ * 2*pi;
        }
    }
```



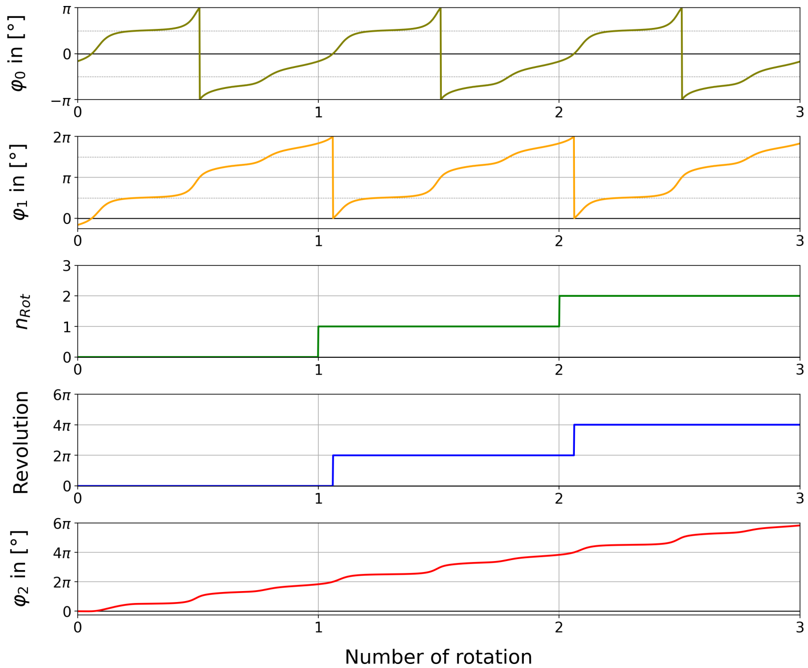
(a) Lead angle.



(b) Lag angle.



(c) Counter angle with lead blade.



(d) Counter angle with lag blade.

Figure 7.5: φ_0: primary counter angle calculated by the arc tangent 2 function. φ_1: couner angle shifted about 2π to get only positive angle.
n_{Rot}: number of completed rotation. `Revolution` $= 2\pi \cdot n_{Rot}$: offset, to obtain a continuous counter angle.
φ_2: final counter angle, including the delay.

A further offset angle has to be considered for blades, which do not have a vertical inial alignment. This concerns all but `blade0`.

```
    scalar AngleOffset_ =  2.0*pi *offset_;
```

The final counter angle is the summation multiplied by the lag.

```
    rotation.z() =  ( phi - AngleOffset_ + Revolution )  *lag_;
```

Final procedure

The two vectors `displacement` and `rotation` are merged in the final motion vector `TR`. This vector is given to the two OpenFOAM classes `quaternion` and `septernion`, which perform translations and rotations in 3D space.

```
    Vector2D<vector> TRV(displacement,rotation);
    quaternion R(quaternion::XYZ, TRV[1]);
    septernion TR(septernion(-origin_ + -TRV[0])*R*septernion(origin_));
```

Finally, the code passes the results to the solver.

8 Optimisation

Before starting the optimisation, a characteristic value or an objective function has to be defined for which the minimum or maximum shall be reached. This value will be derived in the Section 8.1. The following sections describe the chosen setup and the process loop for the optimisation cases. The last Section 8.4 contains the input data for the DAKOTA optimisation.

8.1 Evaluation

For conventional helicopters, the efficiency of a rotor is given by the figure of merit. It's defined as the ratio of the ideal induced power to the real power needed for hovering.

$$\xi_l = FoM \ = \frac{P_{id}}{P_{real}} \tag{8.1}$$

The ideal rotor has a figure of merit equal to one. Typical values for FoM lay between 0.6 and 0.7, modern rotors reach FoM ≈ 0.8 according to van der Wall, [20].

This figure of merit is the characteristic value for optimisation, which DAKOTA maximises. The mean values of the last rotation are used for the FoM calculation. Only the aerodynamic forces and moments are taken into account; inertial loads are not.

8.1.1 Ideal power

The ideal induced power as a result of the momentum theory is given by

$$P_{ideal} = v_i \cdot T \ = \sqrt{\frac{\overline{T}^3}{2 \cdot \rho \cdot S}} \ . \tag{8.2}$$

This equation was derived from actuator disk theory and is technically speaking only valid for rotor blades that lie in the disk's plane with a constant induced velocity v_i. In contrast to helicopters, the blades of cycloidal rotors are subjected to different flow velocities due to their movement. The blades also cross their downwash, which influences flow velocity. However, Equation (8.2) is used to determine the ideal power.

As defined in the Section 4.1, the air density is $\rho = 1.225 \frac{\mathrm{kg}}{\mathrm{m}^3}$.

The reference surface S represents the disc surface through which the flow passes. For the cycloidal rotor, the cross-section is used, as shown in Figure 8.1. It is assumed that the fluid mainly flows in the negative y-direction.

$$S = w_R \cdot d_R \ , \tag{8.3}$$

with the rotor width w_R and the rotor depth d_R, see Figure 8.1. The width can vary depending on the optimisation subject; the depth is always 1m.

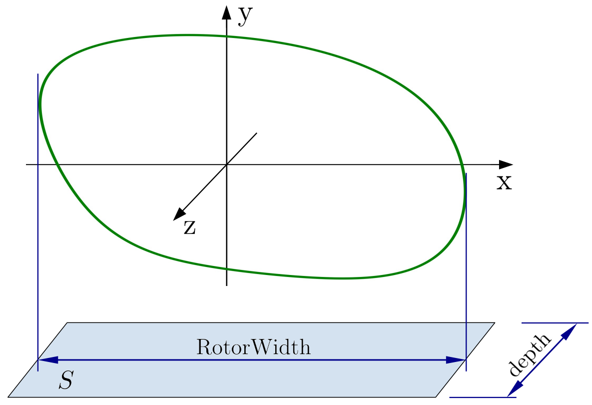


Figure 8.1: Reference surface for cycloidal rotors.

The thrust can be calculated by the force components given for each blade

$$T = \sqrt{F_x^2 + F_y^2}\ .$$

Later on, a function calculates the mean values of the components for one rotation ($\overline{F}_x$, $\overline{F}_y$). This leads to the mean thrust $\overline{T}$ as well as the mean ideal power $\overline{P}_{id}$.

8.1.2 Real power

The blades' motion was linearised between two time steps t_i and t_{i+1}, and all values were interpolated between the steps. This approach is necessary to calculate the blades' angular velocity.

There is a translation and a rotation of the blade, which can be considered separately for one revolution, see Figure 8.3. Therefore, the real power is the sum of translation and rotation power. Again, the mean values over one rotation are calculated.

$$\overline{P}_{real} = \overline{P}_{tra} + \overline{P}_{rot} \tag{8.4}$$

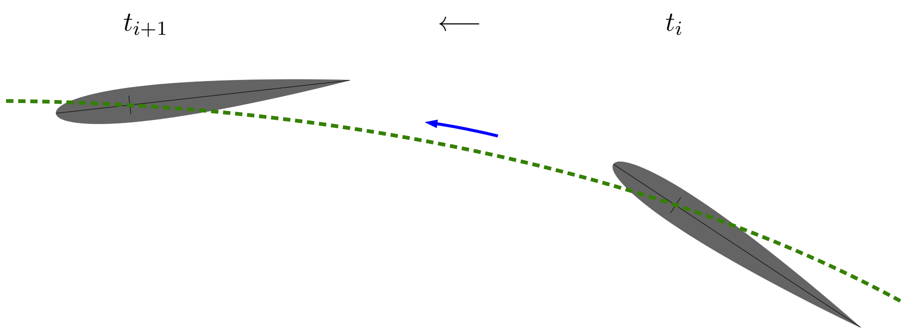
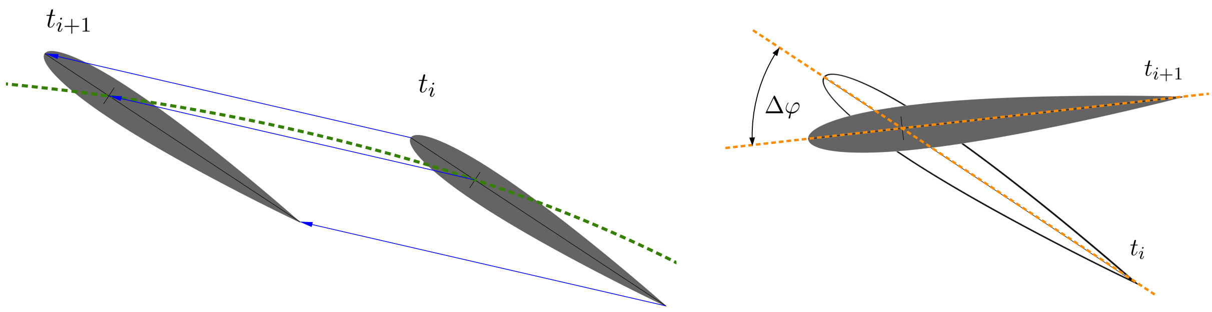


Figure 8.2: Blade movement from time step t_i to t_{i+1}.



(a) Linearised translation of blade.　　(b) Rotation of blade.

Figure 8.3: Splitting of the blade movement.

8.1.3 Translation power

The translation power is the tangential force F_t multiplied by the current velocity

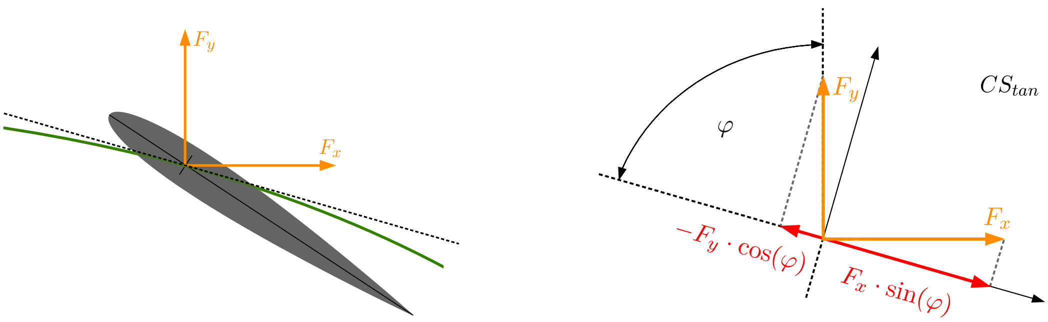
$$P_{tra} = F_t \cdot v \ . \tag{8.5}$$

The global forces F_x and F_y are translated into the second local coordinate system CS_{tan} and summed up to obtain the tangential force (see Figure 8.4).

$$F_t = F_x \cdot \sin(\varphi) - F_y \cdot \cos(\varphi) \tag{8.6}$$

The fact that the tangential forces for two time steps t_i and t_{i+1} are not colinear is neglected. This is valid because of the small time steps. The interpolated tangential force between two steps is

$$\overline{F}_t = \frac{1}{2} \cdot (F_{t,i} + F_{t,i+1}) \ . \tag{8.7}$$



(a) Forces in the local cartesian system.　　(b) Transferred forces in the tangential coordinate system CS_{tan}.

Figure 8.4: Tangential force for the translation power calculation.

Due to the two different modes of drive (see Section 4.2), there are two different velocities.

1. Constant velocity: $v_{Blade} = \text{const.}$ depending on the Reynolds number.
2. Constant rotation velocity: $\omega_{Drive} = \text{const.}$ with $v_{Blade} = \omega_{Drive} \cdot R_i$.

For the constant rotation velocity, the blade's velocity is interpolated.

$$\overline{v}_i = \frac{1}{2} \cdot (R_i + R_{i+1}) \cdot \omega_{Drive} \ , \tag{8.8}$$

where the current radius is given by

$$R_i = \sqrt{X_i^2 + Y_i^2} \ . \tag{8.9}$$

8.1.4 Rotation power

The rotation power is the local moment M_{Blade} as a result of the aerodynamic pressure on the airfoil surface multiplied by the angular velocity ω_{rot}.

$$P_{rot} = M_{Blade} \cdot \omega_{rot}$$

In OpenFOAM, the rotation centre for the output moments is fixed and cannot be adjusted during the calculation. The necessary local moment of the blade, which refers to the moving pivot point, is unavailable.

Therefore, the output moment with its rotation centre in the global origin is taken. This moment is the sum of the local moment and a correction moment (see Figure 8.5).

$$M_{global} = M_{Blade} + M_{corr} \tag{8.10}$$

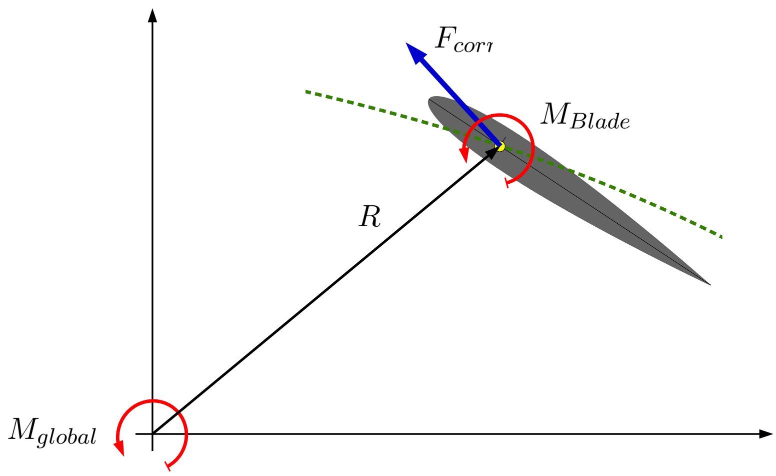


Figure 8.5: Compounding the global moment.

The correction moment is the corrector force F_{corr} multiplied by the current lever to the blades' pivot point.

$$M_{corr} = F_{corr} \cdot R$$

The correction force is perpendicular to the lever, which leads to

$$F_{corr} = -F_x \cdot \sin(\Psi) + F_y \cdot \cos(\Psi) \ . \tag{8.11}$$

The local blade moment is then

$$M_{Blade} = M_{global} - F_{corr} \cdot R \ . \tag{8.12}$$

Again, the values were interpolated

$$\overline{M}_{Blade} = \frac{1}{2} \cdot (M_{Blade,i} + M_{Blade,i+1}) \ . \tag{8.13}$$

The rotating motion is a superposition of two different rotations. The first part of the rotation is the tilting angle φ due to the blades' movement. The change of the pitching angle α_0 is the second part. The resulting rotation angle of the blade is

$$\theta = \varphi + \alpha_0 \ . \tag{8.14}$$

The angular velocity is the change of the rotating angle from one time step to another.

$$\omega_{rot} = \frac{\Delta\theta}{\Delta t} = \frac{\theta_{i+1} - \theta_i}{t_{i+1} - t_i}$$

The pitching and tilting angles (α_0, θ) are determined by OpenFOAM functions and are written in the log file.

8.2 Setup

There are different possible combinations to generate runs for optimisation. The following parameters can be varied.

- The number of blades: 1, 2, 3, 4.
- The subject of optimisation, which is subdivided into three parts "Pitching", "Trajectory", and "Both".
  - "Pitching": for a circular blade motion, the blade pitching is optimised.
  - "Trajectory": for sinusoidal pitching with a fixed amplitude $\hat{\alpha}_0$, the blades' trajectory is optimised.
  - "Both": the combination of the pitching and the trajectory is optimised.
- Type of drive mode: constant velocity and constant rotational speed.
- For the latter mode drive, three different values were considered: ω_{rot}, $2 \cdot \omega_{rot}$, $4 \cdot \omega_{rot}$. These values were only considered for the two-blade case.

Table 8.1 shows all optimisation runs which are carried out for this thesis.

Table 8.1: Considered optimisation runs and corresponding parameters.
$v_{Drive} = 0.775\,\frac{\mathrm{m}}{\mathrm{s}}, \;\; \omega_{Drive} = 0.517\,\frac{1}{\mathrm{s}}$

Name	Blades	Variable	Mode of drive	Value
Opti_1PV	1	Pitching	const. velocity	v_{Drive}
Opti_1TV	1	Trajectory	const. velocity	v_{Drive}
Opti_1BV	1	Both	const. velocity	v_{Drive}
Opti_1TO	1	Pitching	const. angular velocity	v_{Drive}
Opti_1BO	1	Both	const. angular velocity	v_{Drive}
Opti_2PV	2	Pitching	const. velocity	v_{Drive}
Opti_2PVx2	2	Pitching	const. velocity	$2 \cdot v_{Drive}$
Opti_2PVx4	2	Pitching	const. velocity	$4 \cdot v_{Drive}$
Opti_2TO	2	Trajectory	const. angular velocity	ω_{Drive}
Opti_2TOx2	2	Trajectory	const. angular velocity	$2 \cdot \omega_{Drive}$
Opti_2TOx4	2	Trajectory	const. angular velocity	$4 \cdot \omega_{Drive}$
Opti_2BO	2	Both	const. angular velocity	ω_{Drive}
Opti_2BOx2	2	Both	const. angular velocity	$2 \cdot \omega_{Drive}$
Opti_2BOx4	2	Both	const. angular velocity	$4 \cdot \omega_{Drive}$
Opti_3PV	3	Pitching	const. velocity	v_{Drive}
Opti_3TV	3	Trajectory	const. velocity	v_{Drive}
Opti_3BV	3	Both	const. velocity	v_{Drive}
Opti_3TO	3	Pitching	const. angular velocity	v_{Drive}
Opti_3BO	3	Both	const. angular velocity	v_{Drive}
Opti_4PV	4	Pitching	const. velocity	v_{Drive}
Opti_4TV	4	Trajectory	const. velocity	v_{Drive}
Opti_4BV	4	Both	const. velocity	v_{Drive}
Opti_4TO	4	Pitching	const. angular velocity	v_{Drive}
Opti_4BO	4	Both	const. angular velocity	v_{Drive}

8.3 Procedures

The following two sections describe the procedure of the optimisation runs. The general workflow is that the user initialises an optimisation run, which automatically starts a series of CFD cases. Due to the different optimisation subjects ("Pitching" or "Trajectory"), there are two separate procedures, which will be explained in the following sections. The procedure for pitching and trajectory optimisation ("Both") is a combination of them.

Please note that a *run* is one DAKOTA optimisation with a unique setup, listed in Table 8.1, whereas a *case* is a single CFD calculation within a run.

In Appendix C is the procedure scripts `operateDict.py` and the code for its functions `Control.py`. Theses scripts are used for the pithcing and trajectory optimisation with constant velocity.

8.3.1 Pitching

Figure 8.6 shows the flowchart for the pitching procedure.

1. DAKOTA sends a set of 16 variables and an ID to the PYTHON script `operateDict.py`. These variables are the control vertices for the pitching spline.
2. A new case is generated by copying and renaming the initial case (`Run_ + ID`).
3. The script updates the `dynamicMeshDict` by overwriting the control vertices for the spline pitching with the new one.
4. The OPENFOAM case is started via the `Run.sh` file.
5. During the calculation, the log file is read to check the state. There are four possible states:
   - Still running: The function will sleep for a while, and after that, it will reread the log file.
   - TimeOut error: The case will be aborted if the log file's content does not change over 20 minutes due to an internal failure.
   - OpenFOAM error: As a result of a high CFL number or other numerical issues, the case will be aborted by OpenFOAM itself.
   - End: The case has ended successfully; the next step can be executed.

   If one of the errors occurs, a penalty value is assigned by setting the figure of merit to 10^{-6}, and the case dictionary is deleted.
6. If the case ended successfully, a function calculates the figure of merit assessed on the output forces and moment.
7. A `RunLog.txt` stores the essential input and output values. It also notes errors during the case.
8. The case dictionary is deleted.
9. The PYTHON script sends the inverse value of the FoM back to DAKOTA.

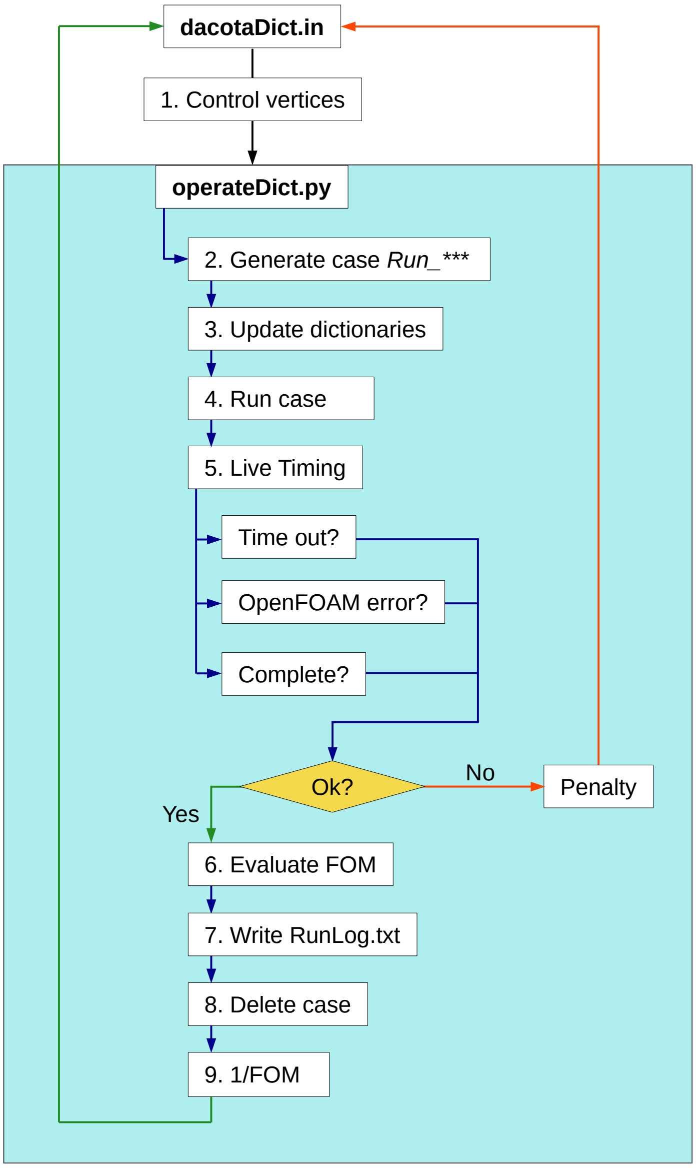


Figure 8.6: Flowchart of the pitching optimisation procedure.

8.3.2 Trajectory

A different procedure is shown in Figure 8.7 for the trajectory procedure.

1. DAKOTA sends a set of 14 variables and an ID to the PYTHON script `operateDict.py`. These variables are the control vertices for the trajectory spline.
2. At first, the control vertices are assessed if they would generate a trajectory with an intersection, which would lead to a distorted or senseless path, as shown in Figure 8.8
   If yes, a penalty value is assigned by setting the figure of merit to 10^{-6}. The case then is aborted.
3. If the assessment is successful, the trajectory is calculated based on the given control vertices. The coordinates of the vertices are transformed to ensure that the spline's arc length is exactly 3π.

$$T_i^* = T_i \cdot \frac{3\pi}{L_{tra}(\gamma)} \tag{8.15}$$

4. The resulting trajectory is assessed in terms of three criteria.
   - Maximum curvature: the trajectory's curvature is determined according to Chapter 5.5 and may not exceed the limit of $\sigma_{lim} = 4\ \frac{1}{\mathrm{m}}$. Figure 8.9(a) shows a trajectory with a just acceptable curvature, whereas the curvature of Figure 8.9(b) is too high.
   - Minimum distance, only necessary for the mode drive with a constant angular velocity. For this drive, a minimum distance between the trajectory and the rotation origin must be met, as the actuators have a minimum retraction length. Therefore, the distance of each control vertices to the origin has to be greater than 1 meter, see Figure Figure 8.10. If not, a penalty value is assigned by setting the figure of merit to 10^{-6}. The case then is aborted.
   - Maximum ΔR, only necessary for the mode drive with a constant angular velocity. The change of the radius or the radial velocity has to be limited. A high motion would lead to numerical issues (high CLF number), which could interrupt the case. From a technical point of view, actuators have a limited positioning speed, which should be considered. The radial velocity may not exceed $\pm 0.65\ \frac{\mathrm{m}}{\mathrm{s}}$, see Figure Figure 8.11.

$$v_{rad} = \frac{\Delta R}{\Delta t} = \frac{R_{i+1} - R_i}{t_{i+1} - t_i} < 0.65\ \frac{\mathrm{m}}{\mathrm{s}} \tag{8.16}$$

5. If the assessment is successful, a new case is generated by copying and renaming the initial case (`Run_+ID`).
6. To obtain a low computationally intensive mesh, a customised STL volume gives the shape for the last refinement level of the background mesh. The pitching path and the trajectory are used to calculate the border of this geometry.
   - Figure 8.12(a) shows the bounding box, representing the outside of the blade mesh, which is placed multiple times along the trajectory, considering the associated pitching angle, see Figure 8.12(b). The resulting coordinates of the bounding boxes are stored in an array. The next step is to collect all points which form the envelope.
   - A triangle with an interior angle of 4.5° and an edge length of 10 m is constructed. It rotates around the origin in 4.5° steps until a complete rotation is reached. For every triangle's position, a function collects the points, which lay inside the shape, and determines its distance to the origin. The point with the farthest distance defines this section's geometry border, see Figure 8.12(c) and 8.12(d).

- Based on the remaining points, a function generates a STL volume, see Figure 8.12(e). 8.12(f) shows the resulting refinment mesh.

7. TThe script updates the `dynamicMeshDict` by overwriting the input values. This concerns the control vertices for the spline pitching as well as the corresponding length and section vectors. The `fvSchemes` is also updated. The coordinates for the `searchBox` are adjusted, where the dimension of the previous STL volume are taken.
8. A script builds the necessary mesh within the following steps:
   - Generating a block mesh.
   - Refinement with `snappyHexMesh` and STL volume.
   - Merging of the required blade meshes with the background mesh.
   - Extrude mehs to obtain a mesh with one single cell in the z-direction.
9. The OpenFOAM case is started via the `Run.sh` file.
10. During the calculation, the log file is read to check the state. There are four possible states:
    - Still running: The function will sleep for a while, and after that, it will reread the log file.
    - TimeOut error: The case will be aborted if the log file's content does not change over 20 minutes due to an internal failure.
    - OpenFOAM error: As a result of a high CFL number or other numerical issues, the case will be aborted by OpenFOAM itself.
    - End: The case has ended successfully; the next step can be executed.

    If one of the errors occurs, a penalty value is assigned by setting the figure of merit to 10^{-6}, and the case dictionary is deleted.
11. If the case ended successfully, a function calculates the figure of merit based on the output forces and moment.
12. The essential input and output values are stored in a `RunLog.txt`.
13. The case folder is deleted.
14. The Python script sends the inverse value of the FoM back to Dakota.

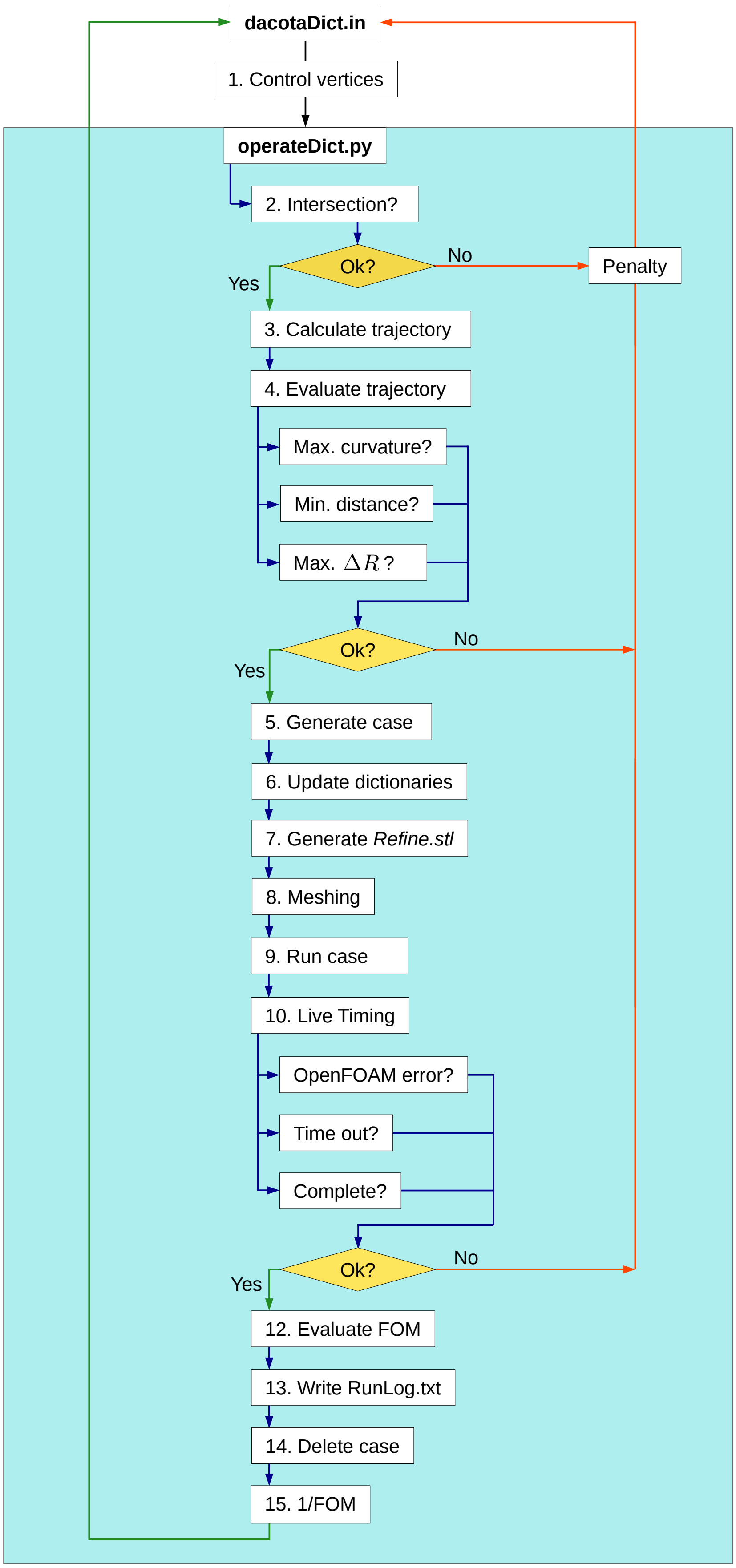
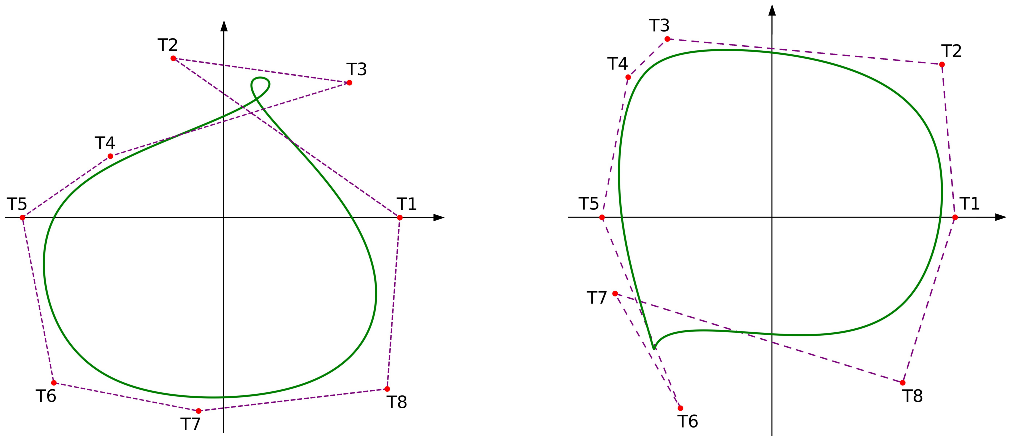
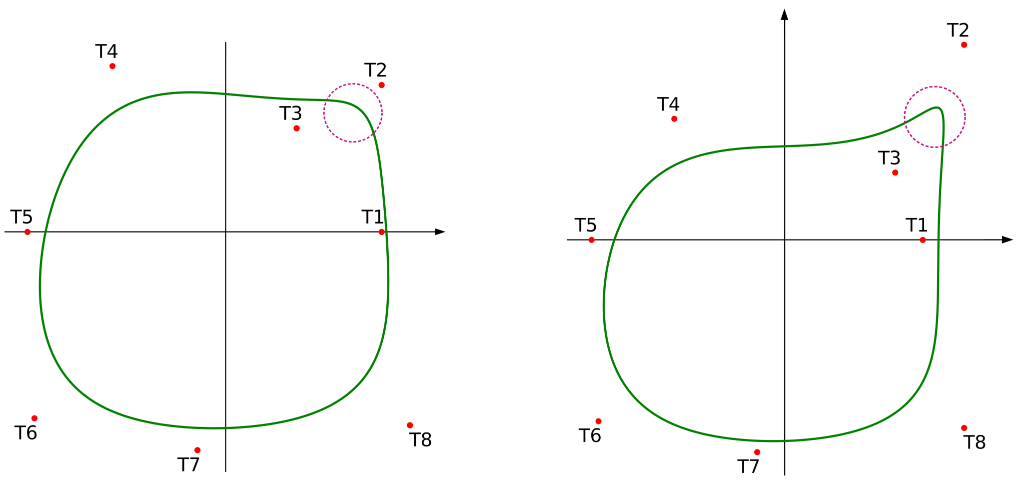


Figure 8.7: Flowchart of the trajectory optimisation procedure.



(a) Intersection inducing in a loop.
(b) Intersection causing a peak.

Figure 8.8: Example of trajectory with an intersection.



(a) Curvature $\sigma = 4$, just acceptable.
(b) Curvature $\sigma = 26.6$, too high.

Figure 8.9: Example of trajectory with different curvatures.

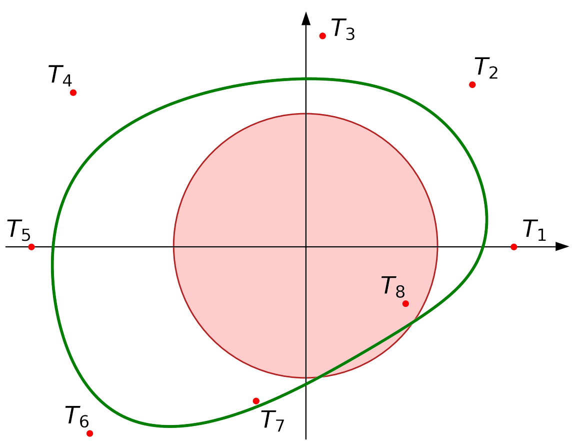


Figure 8.10: Trajectory within the forbidden area, radius = 1 m.

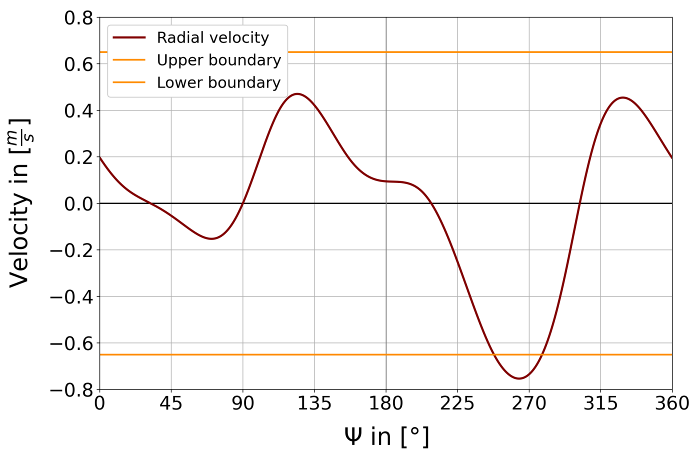
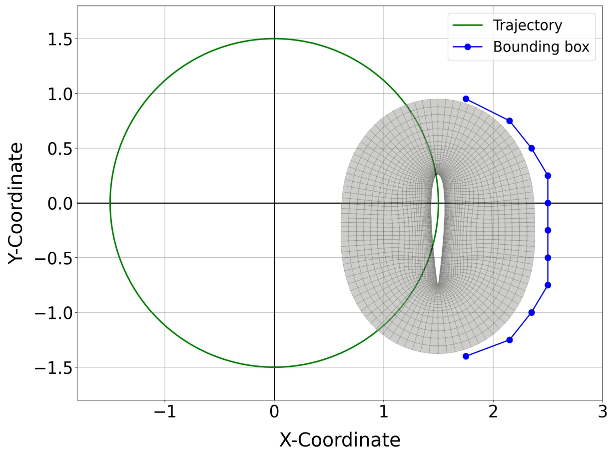
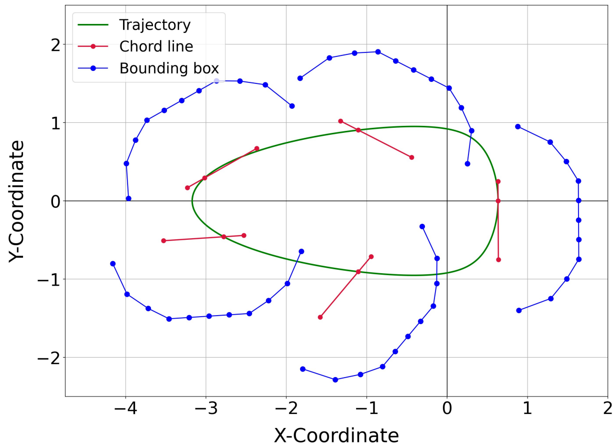


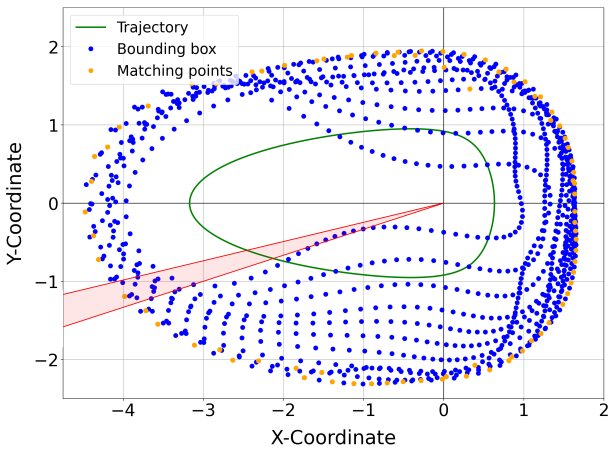
Figure 8.11: Radial speed over azimuth angle.



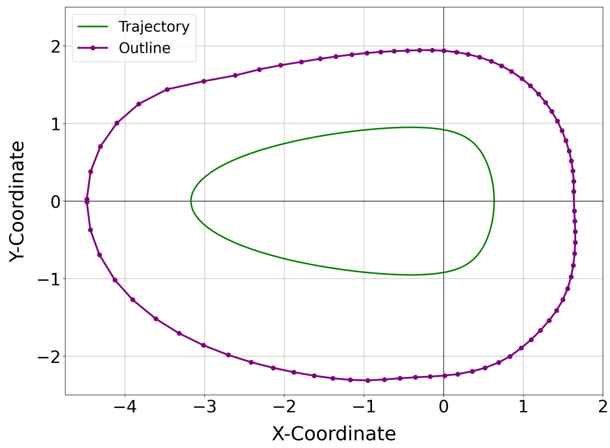
(a) Bounding box of the blade.



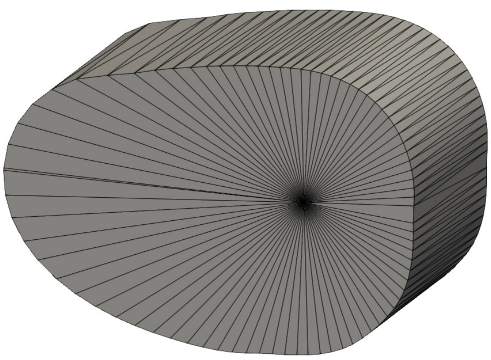
(b) Bounding box placed along the trajectory.



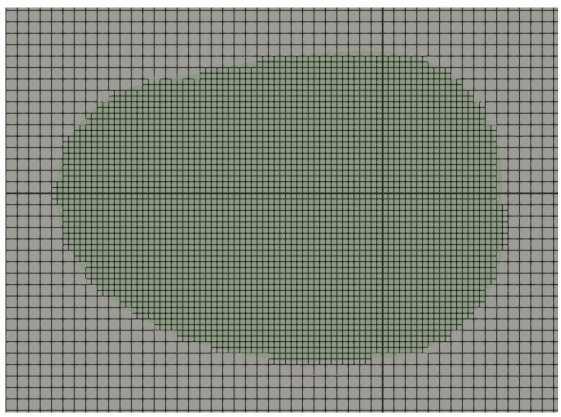
(c) Selecting the farthest points for every triangle.



(d) Oultine of all bounding boxes.



(e) Resulting STL geometry.



(f) Resulting refinement mesh.

Figure 8.12: Approach for individual refinement mesh.

8.4 Dakota

A standard DAKOTA input file with a genetic solver method is adapted for the current optimisation. The Hawk of the HIGH-PERFORMANCE COMPUTING CENTER STUTTGART is available for the optimisation calculation. As one node provides 128 cores, the population size for each run is set to 128.

DAKOTA stops the optimisation run if it reaches one of the three criteria, 'maximum iteration', 'maximum evaluations' or 'convergence tolerance'. The maximum iteration is eightfold the population size; the maximum evaluation is 256 times the maximum iteration. This value guaranteed a sufficient number of cases to obtain an optimum. The convergence tolerance is changed during the runs between 10^{-9} and 10^{-3}. It turned out that these criteria were still too strict. Therefore, two additional criteria are introduced. For this, the figure of merit is sorted in ascending order. First, the difference between the two best FoMs should be less than 0.5 percent. Secondly, the difference between the best and the five hundredth value should be less than one percent. An example of a suitable convergence is shown in Figure 8.13(a).

$$\mathrm{Crit}_1 = \frac{FoM_{2nd}}{FoM_{1st}} - 1 \ \leq \ 0.005 \ \hat{=} 0.5\% \ , \quad \mathrm{Crit}_2 = \frac{FoM_{500th}}{FoM_{1st}} - 1 \ \leq \ 0.01 \ \hat{=} 1\% \ . \tag{8.17}$$

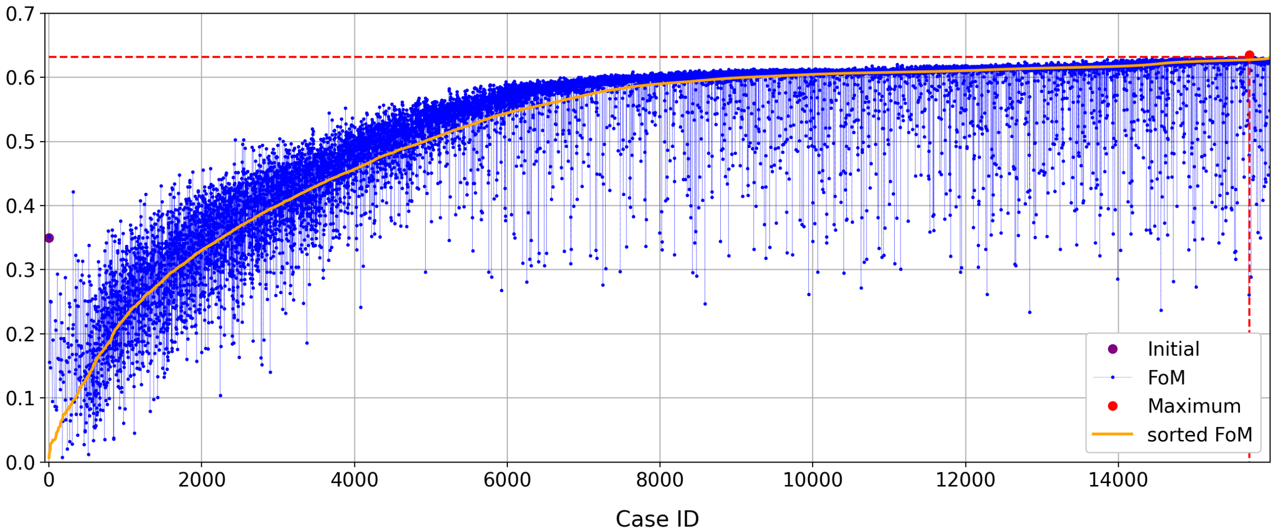
In some circumstances, the optimisation run is aborted due to a lacking convergence, see Figure 8.13(b). As a result of the high execution time of CFD cases some runs are aborted due to a lack of time. The following rable summarises the stop criteria.

Table 8.2: Considered optimisation cases and corresponding parameters.

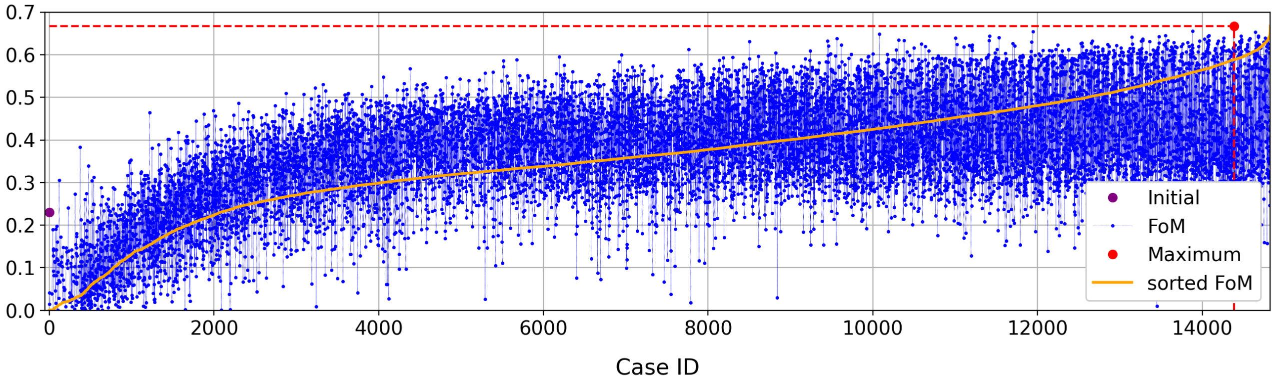
Name	Maximum iteration	Maximum evaluations	Convergence tolerance	FoM criteria 1	FoM criteria 2	Lack of convergence	Lack of time
	maxIter	maxEval	convTol	Crit_1	Crit_2	lackConv	lackTime
Value	1 024	262 144	10^{-3}	0.5%	1 %	-	-

Further essential inputs are variables and their boundaries, which have to be declared in the DAKOTA file. The variables generated by the optimiser correspond to the control vertices of the pitching as well as the trajectory path.

For the boundaries of the pitching vertices, two sinusoidal envelopes limit the values (see Figure 8.14).



(a) Run: Opti-1BV, max. FoM = 0.632, at Index 15 714, Crit_1 = -0.14%, Crit_2=-0.97%.



(b) Run: Opti-4TV, max. FoM = 0.667, at Index 14 384, Crit_1 = -1.76%, Crit_2=-13.0%.

Figure 8.13: Examples for the development of the figure of merit.

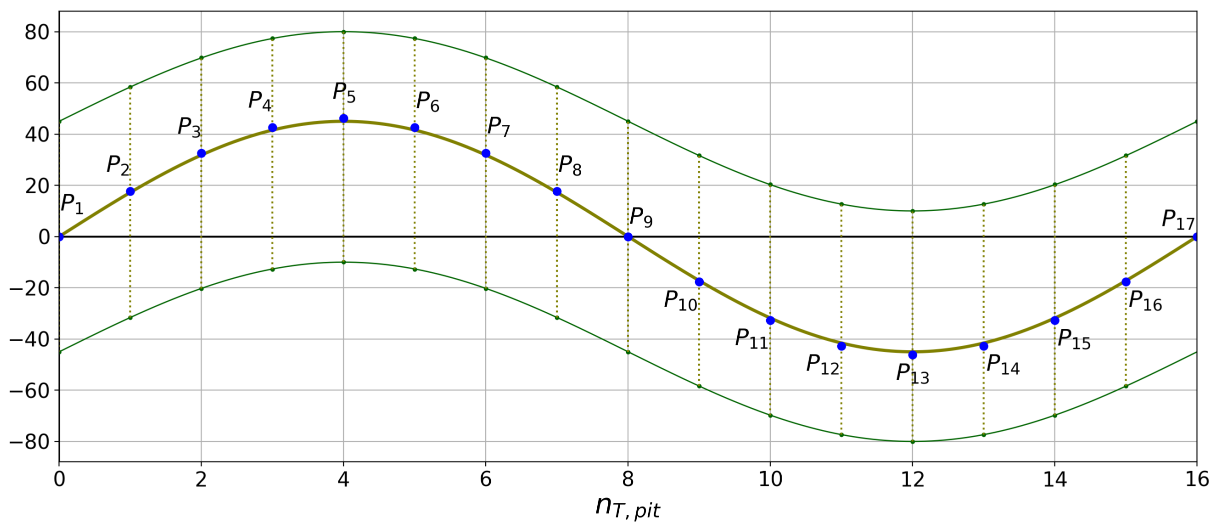


Figure 8.14: Boundary for the pitching control vertices.

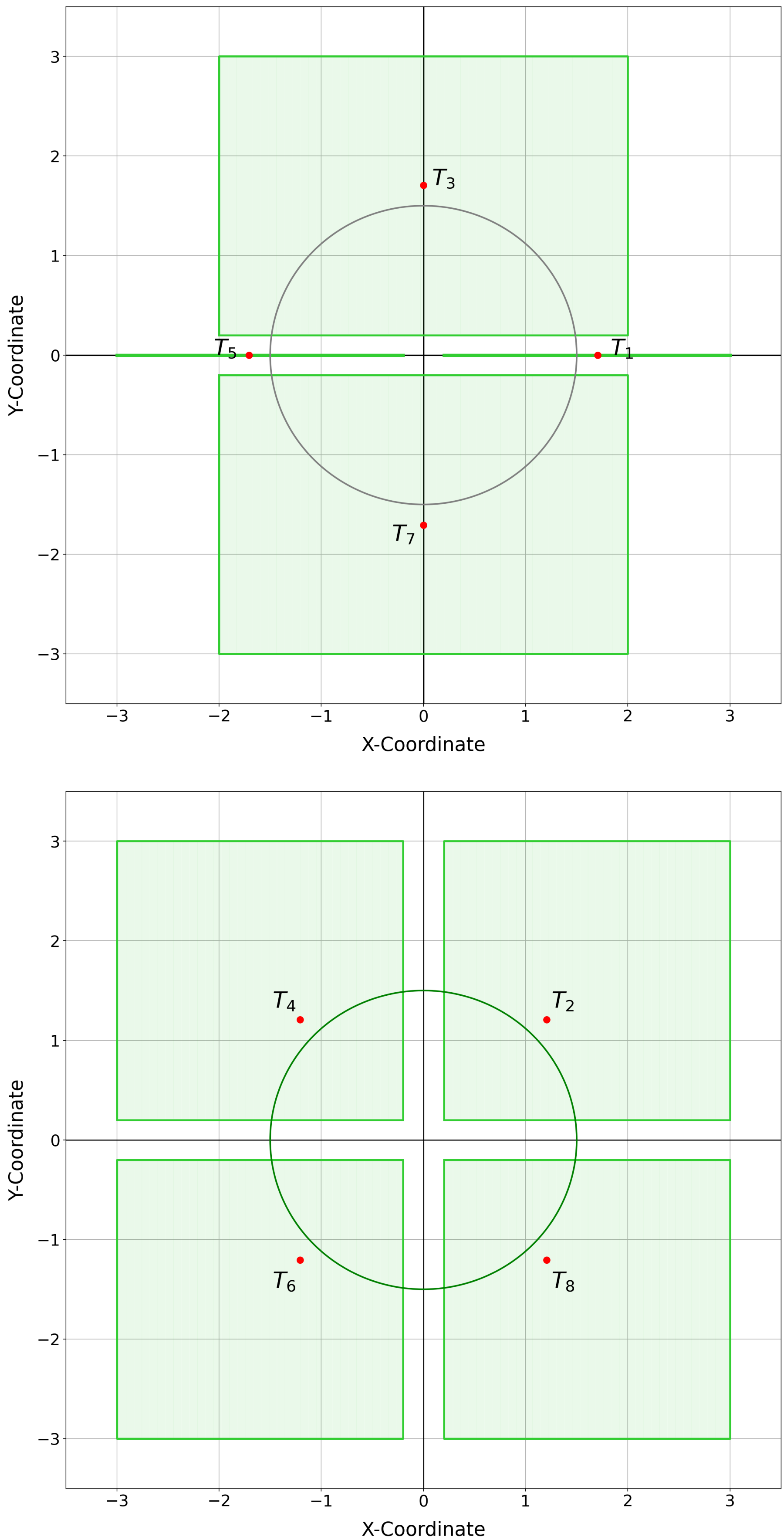


Figure 8.15: Boundary for the trajectory control vertices.

Listing 1 shows the adjusted input file, where the symbol '***' marks run dependent entities. It is essential to mention that the optimiser searches for a minimum. Therefore, the inverse of the figure of merit has been sent to the optimisation solver.

```
environment
     top_method_pointer = 'SOGA'

method
  id_method = 'SOGA'
  model_pointer = 'M1'
  soga
    fitness_type merit_function
    population_size = 128
    max_iterations = 1024
    max_function_evaluations = 262144
    convergence_tolerance = ***

    scaling
    seed = 123456

model
  id_model = 'M1'
  single
    variables_pointer = 'V1'
    interface_pointer = 'I1'
    responses_pointer = 'R1'

variables
  id_variables = 'V1'
  continuous_design = ***
    initial_point     ***
    lower_bounds      ***
    upper_bounds      ***
    descriptors       ***
    scale_types      'auto'

interface
  id_interface = 'I1'
  analysis_driver = ***
    fork asynchronous evaluation_concurrency = 128
    parameters_file = 'parameters.in'
    results_file    = 'results.out'
    file_tag

responses
  id_responses = 'R1'
  objective_functions = 1
  no_gradients
  no_hessians
```

Listing 1: Dakota input file, '***' marks run dependent entities.

9 Amendment

As defined in section 3.1, the rotation starts at an azimuth angle of $\Psi = 0°$. This does not necessarily apply to an arbitrary trajectory, where the starting point could lay above the x-axis. The trajectory's resulting vertical shift does not influence the result for the drive mode 'constant velocity'.

However, for the constant angular velocity the trajectory has to be adjusted so that the starting point lays on the x-axis. Thus the rotating origin lies in the global origin, see Figure 9.1. The adjustment is necessary to carry out the correct evaluation. Unfortunately, this essential shift was not entirely implemented in the `operateDict.py`. The lack of shift leads to a false calculation of the splines' radius due to the false origin, see Figure 9.1 (radius is the distance between the global origin and the trajectory). The maximum deviation of the radius for this example is -24.7%, which is too large to be accepted. This results in a varying angular velocity over the rotation, as shown in Figure 9.3. Also, the calculation of the blade moment is false (see Section 8.1.4 for definition). As a result the determined, translation and rotation power and thus the figure of merit is not correct. Therefore, the captured optimation results are not valid.

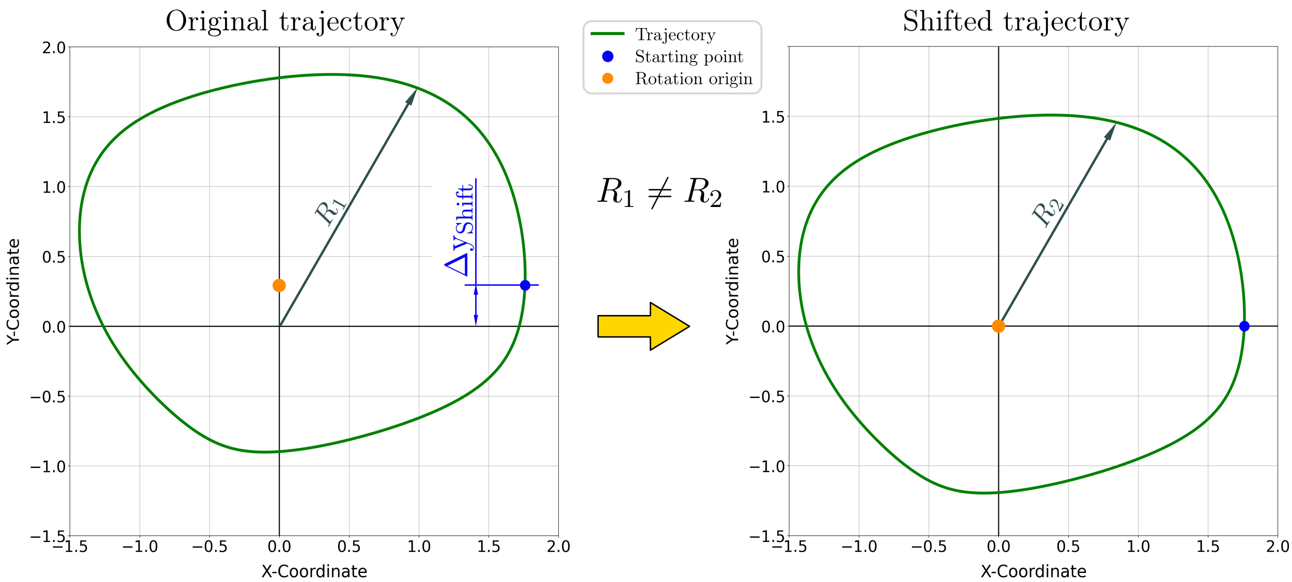


Figure 9.1: Comparison of original and shifted trajectory, Opti-2TO.

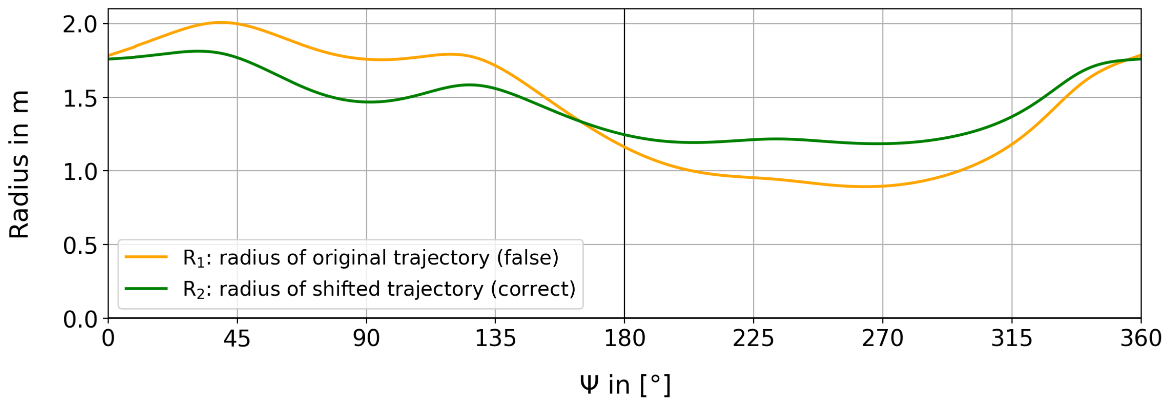


Figure 9.2: Comparison of the two radii R_1 and R_2, Opti-2TO. The maximum deviation is -24.7%.

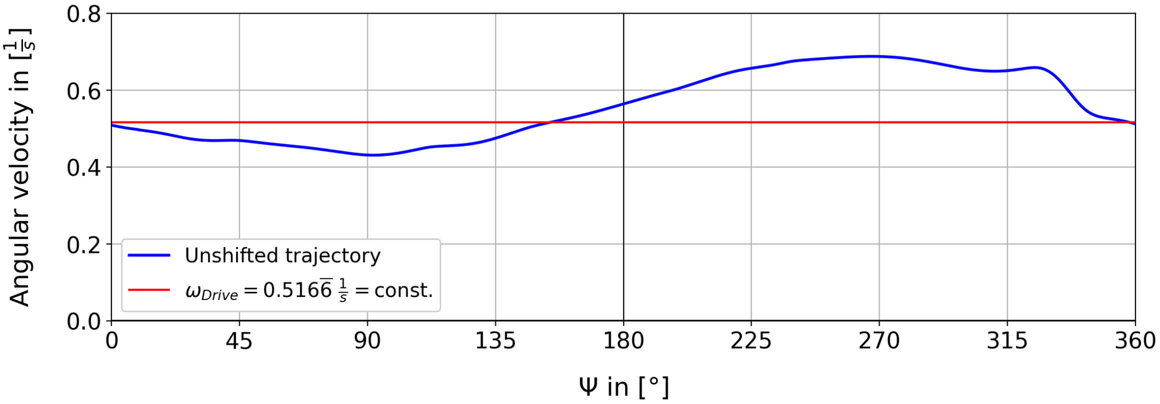


Figure 9.3: Angular velocity for a trajectory without vertical shift, Opti-2TO. The maximum deviation is 33.2%.

The error can be corrected during the precalculation by shifting the control variables in a vertical direction about the value $\Delta y_{\mathrm{Shift}}$, as shown in the following code.

```
yShift = vecTY[0] # get the y value of the starting point

for i in range(0,13):
        Polygon[i][1] = Polygon[i][1] - yShift
```

10 Results

This chapter first presents the initial cases to get the necessary reference values. An overview presents essential results of the optimisation runs. The conclusion gives a résumé about the optimisation runs and their results, which are discussed in detail in the following. Please not, that the pressure field given by OpenFOAM is in relation to the density ρ.

First, some definitions are made, which will be used to evaluate the results.

Dimensionless coefficients for the lift, drag and power are defined to enable a reasonable comparison of the results with each other. The direction of the lift is defined to be perpendicular to the flow direction of the air; the drag is again perpendicular to the lift. As a result of the blade's movement and the downwash, the free stream direction is hard to determine. Thus the lift and drag coefficients are neglected. A distinction is made between translation and rotation power for the corresponding coefficient. In contrast to the calculation of the figure of merit, the sign of the power is taken into account in this coefficient.

$$C_{Power,tra} = \frac{P_{tra}}{\frac{\rho}{2} \cdot v_{blade}^3 \cdot S_{Blade}} \tag{10.1}$$

$$C_{Power,rot} = \frac{P_{rot}}{\frac{\rho}{2} \cdot v_{blade}^3 \cdot S_{Blade}} \tag{10.2}$$

To better understand the pitching and the trajectory, a specific type of plot is used. In addition to plotting the pitch angle α_0 over the azimuth angle Ψ there is a plot of the pitch angle along the trajectory. This helps for a better comprehension of the blade's motion. The colour of the shaded area gives information about the pitching angle's sign:

- red for potitive angle α_0 (leading edge 'outside' the trajectory),
- blue for a negtiv angle α_0 (leading edge 'inside' the trajectory).

Figure 10.1, 10.2 shows the two different plots for a sinusoidal pitching path.

Another different type of plot will be used to assess the direction of the blade thrust. For this visualization, the forces acting on the blade are plotted along the trajectory, as shown in Figure Figure 10.4 exemplary. This qualitative illustration helps to understand the force distribution, especially for comparison. The length of the vectors indicates the normalized force magnitude. Figure Figure 10.5 shows the definition of the colour scheme for the vectors. The classification into the three colour parts always corresponds to the resulting global thrust vector, which is represented by the black arrow:

- red: $\pm 85°$,
- gray: between $85°$ and $95°$,
- blue: greater than $95°$.

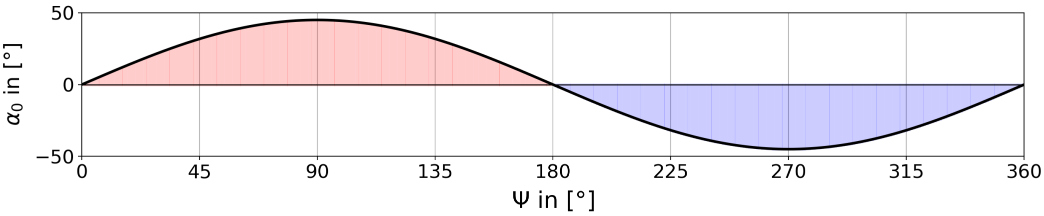


Figure 10.1: Conventional plot of pitching angle over azimuth angle.

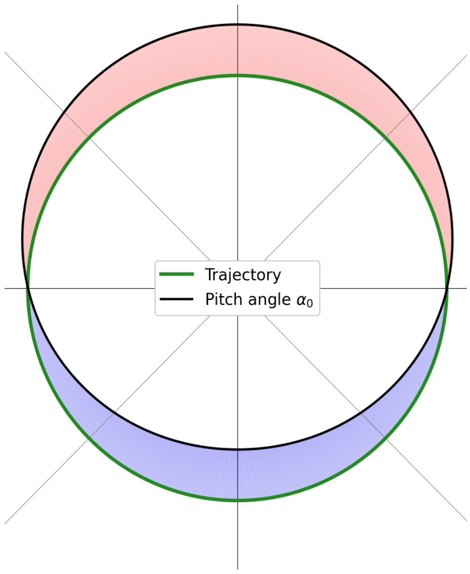


Figure 10.2: Pitching angle over trajectory.

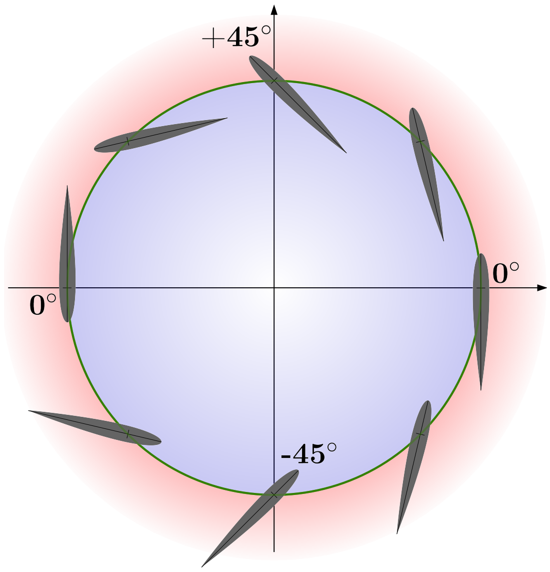


Figure 10.3: Corresponding alignment of the blade.

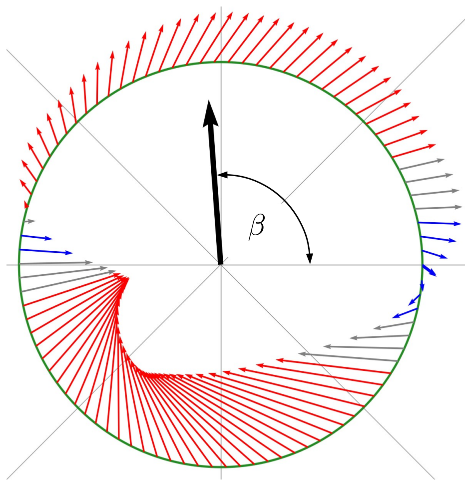


Figure 10.4: Force vectors over trajectory, case 1-Rot.

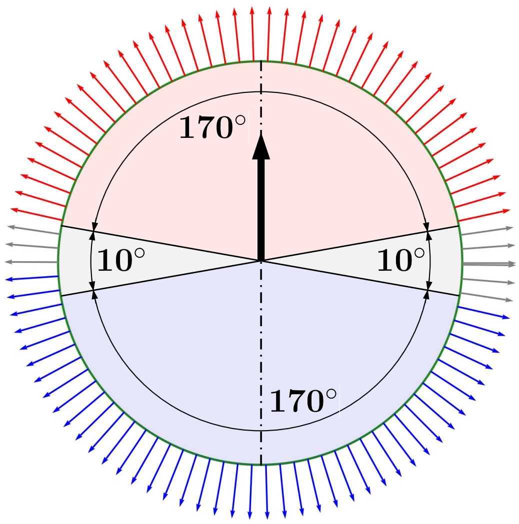


Figure 10.5: Definition for the color scheme.

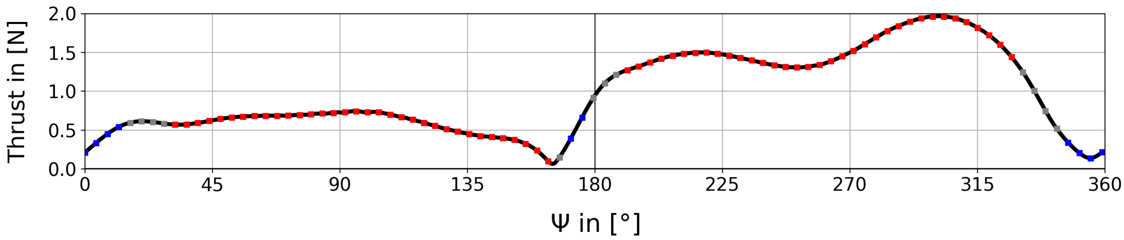


Figure 10.6: Thrust over azimuth angle Ψ.

10.1 Reference cases

An initial case with a sinusoidal pitching ($\pm 45°$) and circular trajectory is carried out for each optimisation case. These cases are the basis for the upcoming evaluation of the optimisation results.

There are four cases with a different number of blades. As a result of the different rotating motions, there are two further cases for each blade. Additionally, two different Reynolds numbers are considered for the two-blade case.

Table 10.1 gives an overview of the reference cases and their resulting figure of merit. The execution time is valid for a single-core calculation on the IAG's Prandtl cluster.

There is a deviation between the two motion classes for the multi-blade cases. The highest FoM deviation of 15% occurs for the three-blade cases. For the two- and four-blade cases, the FoM deviation is 4%. The reason for the discrepancy could be the uncertainties due to the overset or discontinuities of the splines (see Section 10.3).

Table 10.1: Considered optimisation cases and corresponding parameters.

Name	Blades	Motion	FoM [-]	Thrust [N]	Real power [W]	Re [-]	Execution [s]
1-Rot	1	`solidBody`	0.350	0.537	0.414	50 000	3 433
1-Spline	1	`bSpline`	0.350	0.542	0.420	50 000	3 446
2-Rot	2	`solidBody`	0.309	0.615	0.576	50 000	7 275
2-Rotx2	2	`solidBody`	0.336	2.41	4.10	100 000	7 292
2-Rotx4	2	`solidBody`	0.378	11.4	37.3	200 000	7 313
2-Spline	2	`bSpline`	0.320	0.703	0.680	50 000	7 283
3-Rot	3	`solidBody`	0.284	0.733	0.816	50 000	9 968
3-Spline	3	`bSpline`	0.327	0.751	0.734	50 000	10 003
4-Rot	4	`solidBody`	0.222	0.857	1.32	50 000	13 884
4-Spline	4	`bSpline`	0.230	0.928	1.43	50 000	14 103

10.2 Overview

The absolute maximum figure of merit of 0.758 is reached with four blades and optimisation of pitching only, Opti-4P. This case is also the one with the best improvement of +241%.

The overall results are listed in Table 10.2 and shown in Figure 10.7.

The control vertices for the optimal cases are listed in Section F.

Table 10.2: Overall results of optimisation and corresponding parameters. For criterion see Table 8.2.

Name	FoM [-]	Improve-ment	Thrust [N]	Real power [W]	Valid runs	Criterion
Opti-1P	0.646	+85%	0.312	0.119	14 638	$Crit_2$
Opti-1TV	0.490	+40%	0.776	0.471	13 953	$Crit_2$
Opti-1BV	0.632	+81%	0.368	0.130	13 576	$Crit_2$
Opti-2P	0.690	+123%	0.802	0.383	9 330	$Crit_2$
Opti-2Px2	0.731	+118%	3.18	2.86	11 339	$Crit_2$
Opti-2Px4	0.753	+99%	13.8	25.2	11 293	$Crit_2$
Opti-2TV	0.421	+32%	1.08	1.00	15 298	$Crit_2$
Opti-2BV	0.754	+169%	1.65	1.04	14 802	convTol
Opti-3P	0.708	+149%	1.06	0.568	9 458	convTol
Opti-3TV	0.602	+84%	1.69	1.45	15 296	lackConv
Opti-3BV	0.743	+127%	1.72	1.15	15 877	lackTime
Opti-4P	**0.758**	+241%	2.78	2.25	11 201	lackTime
Opti-4TV	0.667	+190%	1.58	1.34	11 900	lackConv
Opti-4BV	0.400	+48%	1.73	2.45	10 639	lackConv

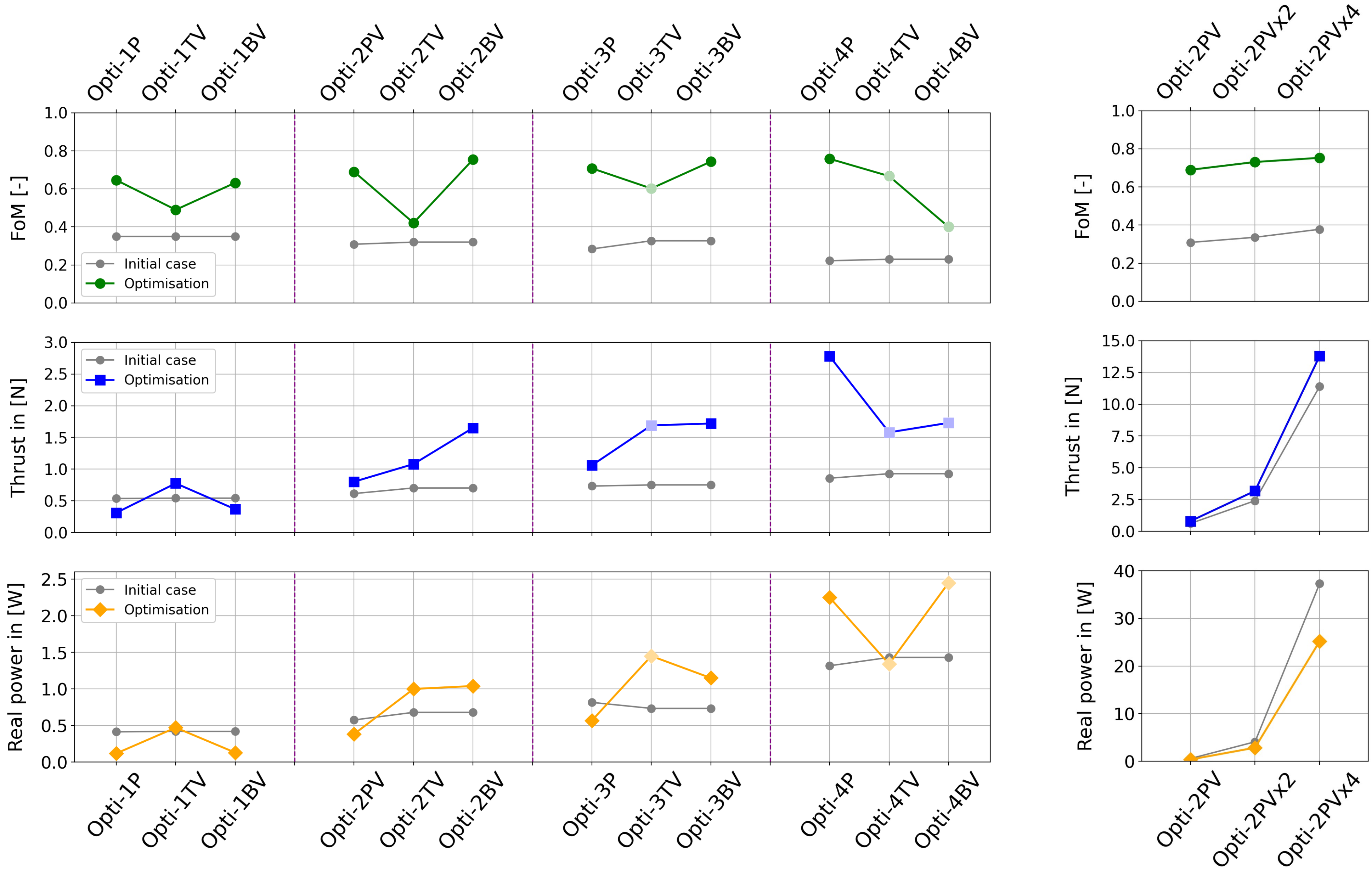


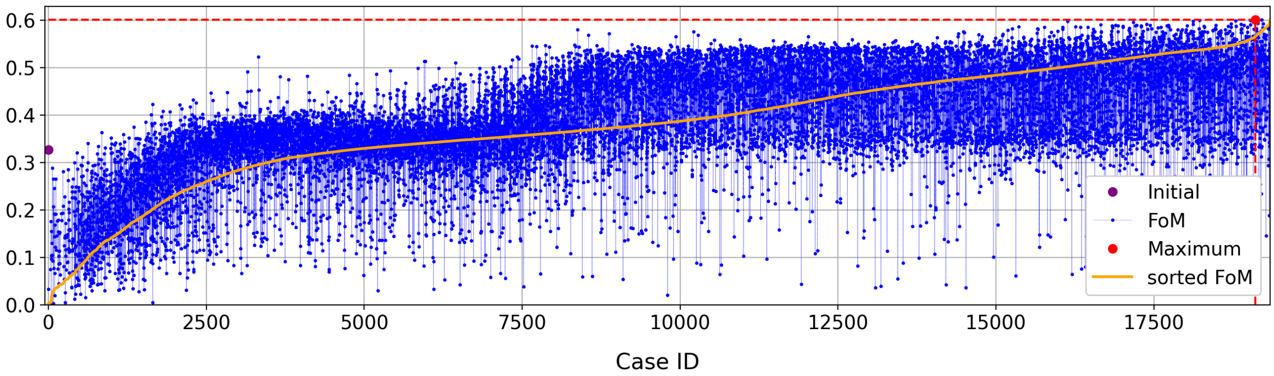
Figure 10.7: Overall results of optimisation. The results of Opti-3TV, Opti-4TV and Opti-4BV are shaded due to a lack of convergence.

10.3 Conclusion

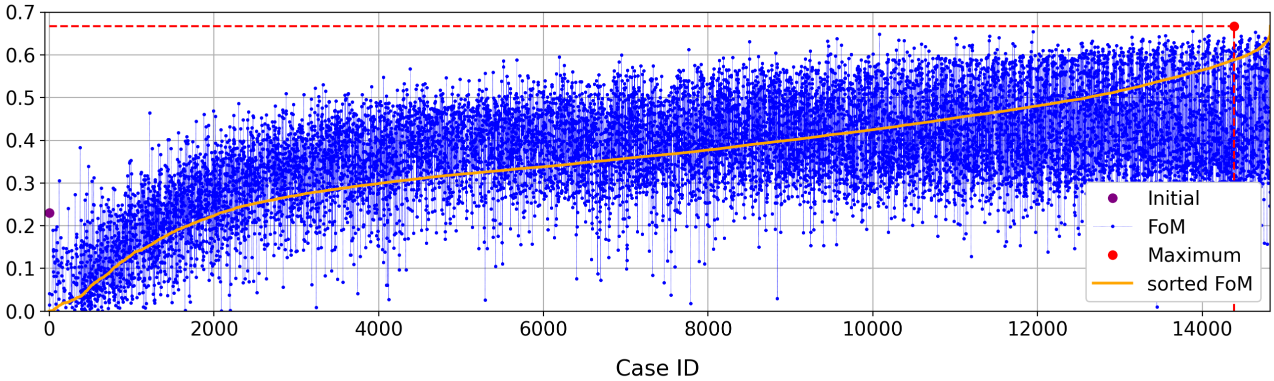
The following conclusions can be drawn from the optimisation runs and their results.

- For the three runs, Opti-3TV, Opti-4TV, and Opti-4BV, no apparent convergence has been reached even for many cases (see Figure 10.8). The optimiser could not consider the previous cases with a suitable FoM.
- At the beginning of each trajectory optimisation run, many variable sets lead to ill trajectories (intersection, curvature etc.), which results in inefficient utilisation of the Hawk node. This could be solved by redefining the boundaries for DAKOTA. Instead of a square boundary space, as previously shown in Figure 8.15, each control variable could lay in non-overlapping segments of a circle, see Figure 10.11. In this manner, intersections can be avoided, and the minimum radius can always be met.
- The boundaries for the pitching path are never reached ($\alpha_{0,bound} = \pm 80°$). The maximum pitch is $\alpha_{0,max} = 51.6°$ (Opti-3BV), and the minimum pitch is $\alpha_{0,min} = -70.7°$ (Opti4P).
- For each blade category [1 ... 4], the 'only trajectory' optimisation run captures the least figure of merit (Opti-4BV has not converged and therefore is not considered).
- It was expected that the optimiser would prefer a horizontally stretched trajectory to generate lift. The opposite is true, as shown for Opti-2BV, Opti-3TV and Opti-4TV, where the trajectories are stretched in the vertical direction.
- There is a considerable increase in the thrust for the multi-blade cases. The optimiser avoids generating a low-pressure area inside the cyclogyro, as shown in Figure 10.9. This reduces the lift of the upper blades. See Figure 10.10 for the comparison of the force distribution for a single-blade and a three-blade case.
- There are two main reasons for the increase of cyclogyros' effectiveness, which are valid for almost all optimal cases.
  1. The blade forces are better aligned in direction to the global thrust.
  2. A rapid change of the pitch and/or a narrow trajectory induce a force peak.
- It is also noticeable that the main blade forces are predominantly generated in the lower half of the trajectory. The optimiser tries to reduce the required power for the remaining trajectory by decreasing the blade forces.
- Two optimisation runs with different Reynolds numbers are carried out for the two-blade case with 'only pitching' ($\mathrm{Re}_{x2} = 100\ 000$ and Re_{x4} 200 000). Although the pitching paths of the three cases are similar, there is an optimum for each Reynolds number. While the figure of merit decreases for the initial case and higher Reynolds numbers, it increases for the two optimisation runs by 8.8% (Re_{x2}) and 17.7% (Re_{x4}). For a commercial cyclogyro in hover flight, the pitching trajectory can be adapted to achieve the best efficiency for the current cargo.
- The reference surface to calculate the figure of merit is not valid for every case (see definiton in Section 8.1.1). The movement of the blades inclines the direction of the downwash, and thus the surface changes through which the flow passes. The correct surface is approximated based on the velocity field. The FoM is recalculated by $\mathrm{FoM}_{\mathrm{corr}} = FoM_{\mathrm{Case}} \cdot \sqrt{\frac{S_{\mathrm{old}}}{S_{\mathrm{new}}}}$. This approach is imprecise as the velocity field is transient, and an exact threshold is hard to define.
- There are discontinuities in the force path, as shown in Figure 10.12(a). The kinks seem to occur

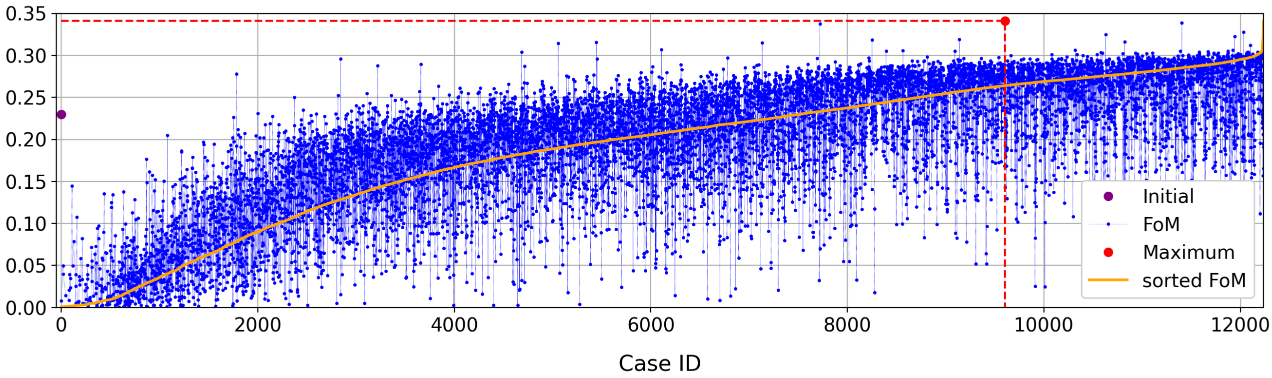
at each pitching spline section. The jump at the beginning of each rotation, which lasts only for a short time, seems to be caused by the trajectory spline. This artefacts are accepted and may cause the differen results of initial cases (see Table 10.1). A reason for the discontinuities could not be found by the end of the thesis.



(a) Run: Opti-3TV, max. FoM = 0.602, at Index 19 107, Crit_1 = 0%, Crit_2=-9.5%.



(b) Run: Opti-4TV, max. FoM = 0.667, at Index 14 384, Crit_1 = -1.76%, Crit_2=-13.0%.



(c) Run: Opti-4BV, max. FoM = 0.341, at Index 9 604, Crit_1 = -0.76%, Crit_2=-15.2%.

Figure 10.8: Figure of merit over cases.

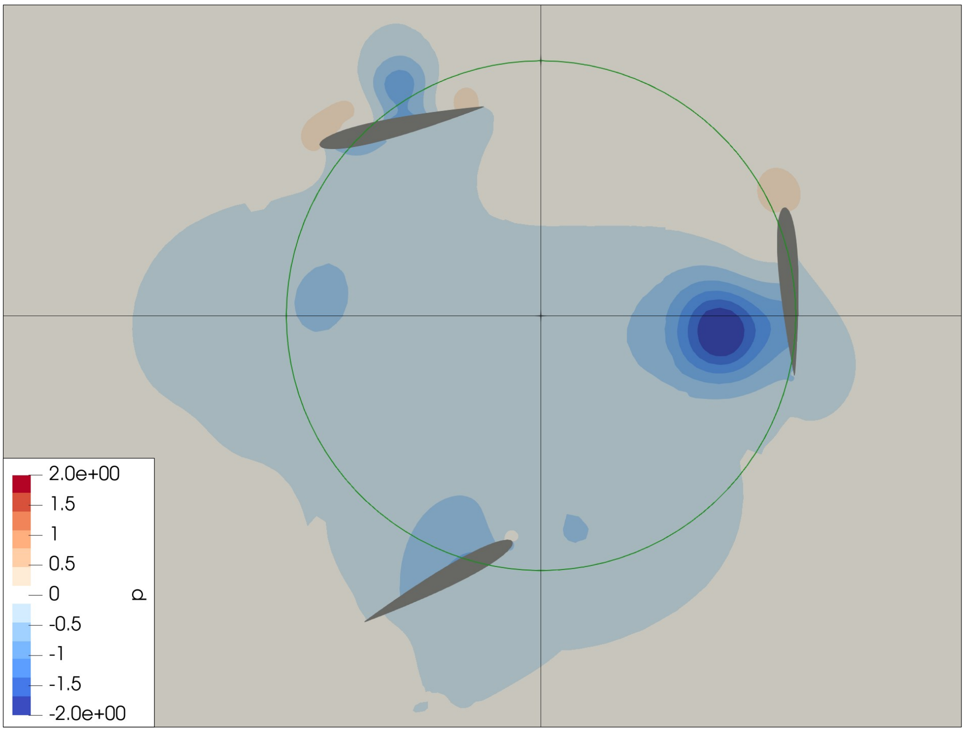
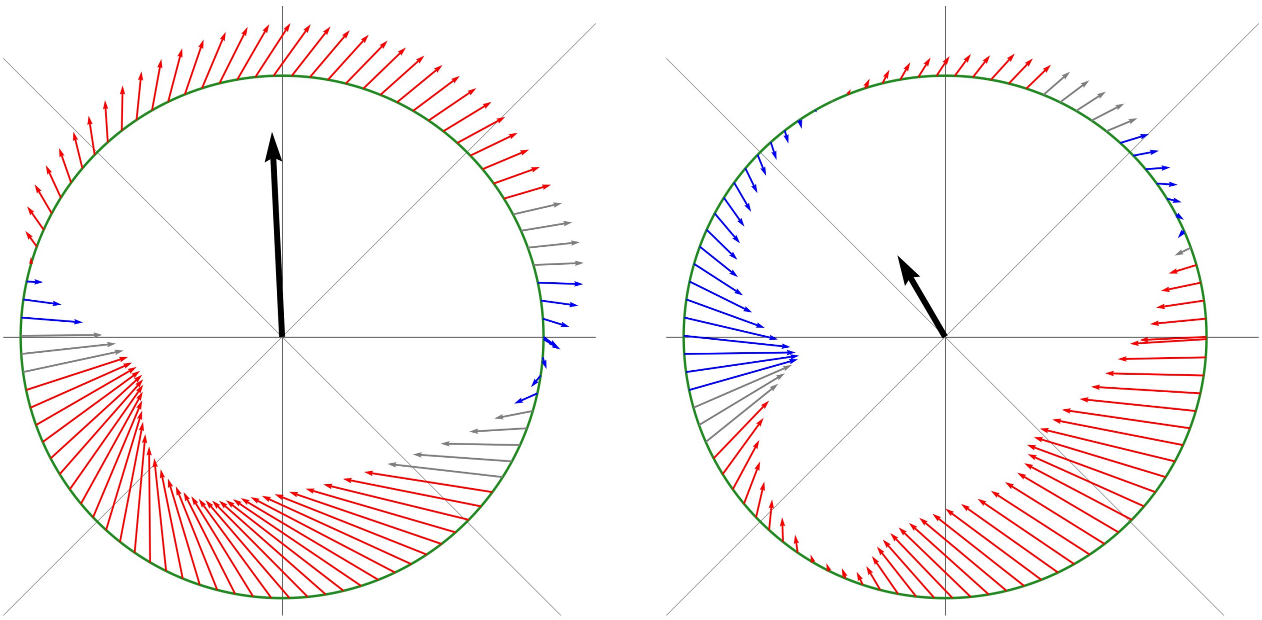


Figure 10.9: Pressure field of three-blade initial case.



(a) Single-blade case

(b) Three-blade case

Figure 10.10: Comparison of the blade forces.

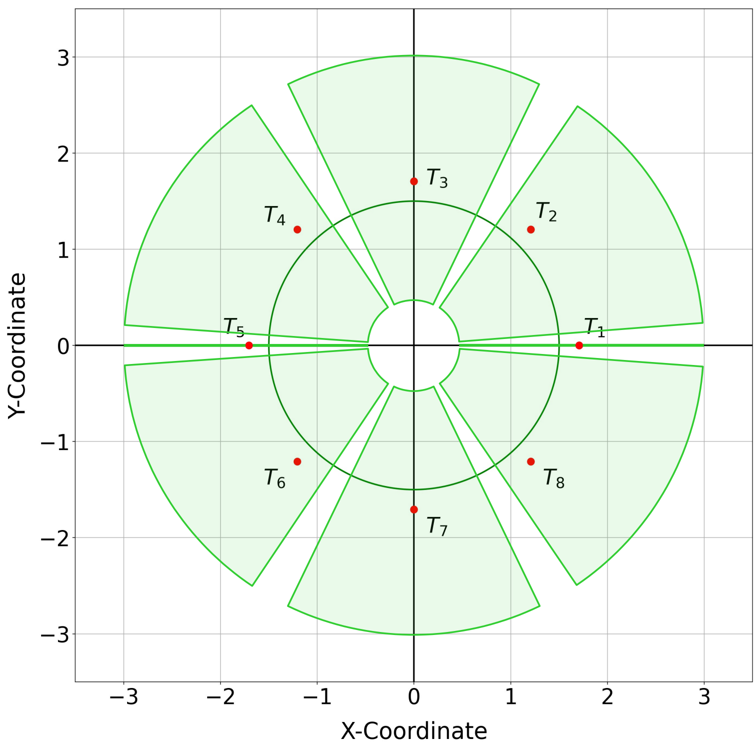
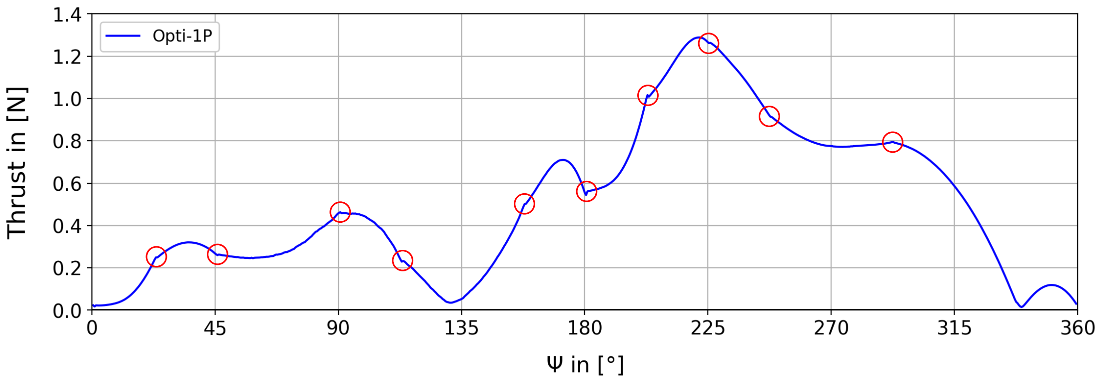
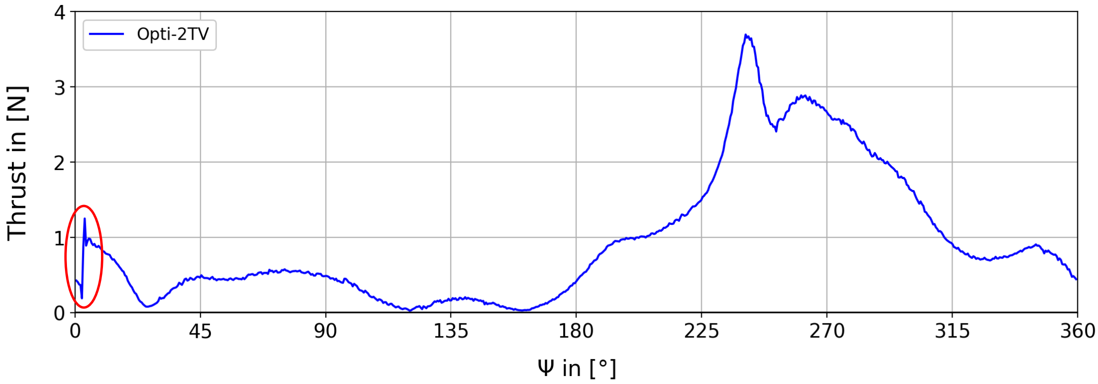


Figure 10.11: Example for a convenient boundary definition for the trajectory control vertices.



(a) Kinks at almost each end of the pitching spline section.



(b) Jump at the beginning of the trajectory spline.

Figure 10.12: Dicontinuities in the force path as a result of the spline motion.

10.4 Single blade

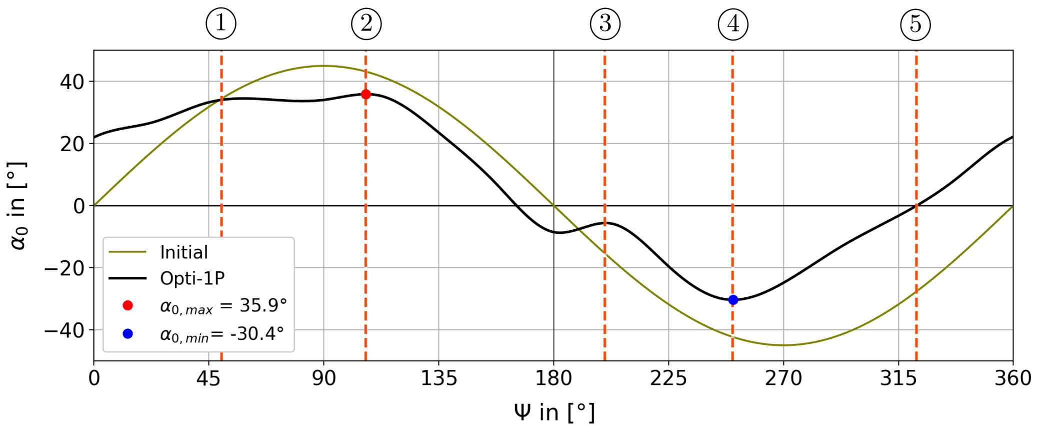
10.4.1 Opti-1P

The rotation begins with a substantial lead angle of $\alpha_{lead} = 22°$, see Figure 10.13. Between ① and ②, the pitch remains nearly constant and reaches its maximum. Further on, the pitch plateau leads to a closer vortex at the leading edge, and the required rotation power is reduced significantly, see $C_{p,rot}$ in Figure 10.17. However, the blade forces are reduced in the revolution's first half, as shown Figure 10.14.

The second half rotation begins with a short-lasting lowering of the pitch angle ($\Delta 5°$) at ③. After the pitch reaches its minimum at ④, it increases almost linearly. At ⑤, the pitch already turns positive, whereas the initial case has a pitch of $\alpha_{0,init} = -30°$. As a result, the shedding of the leading edge vortex is avoided, which occurs in the initial case and requires a lot of power, see $C_{p,tra}$ in Figure 10.17. In addition, the pitching path during the second half rotation straightens the blade forces compared to the initial case, see Figure 10.14.

The blade force vectors of the initial case in the lower half contain a sizeable horizontal component and point in the opposite direction. This circumstance cancels a part of the produced lift, which does not contribute to the resulting thrust and requires power. The pitching path of this optimisation leads to a considerably smaller thrust compared to the initial case (-42%), but the figure of merit increase is twice as much, $\mathrm{FoM}_{1P} = 0.65$.

Table 10.3 lists characteristic values of the optimisation compared to its initial case.



Pos.	Ψ
①	50°
②	106°
③	200°
④	250°
⑤	322°

Figure 10.13: Pitching angle α_0 over azimuth angle Ψ.

Table 10.3: Mean values for Opti-1P.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.537 N	0.145 W	0.343 W	0.071 W	0.414 W	0.350	94°
1P	0.312 N	0.077 W	0.085 W	0.034 W	0.119 W	0.646	64°
Δ	-42%	-47%	-75%	-52%	-71%	+85%	

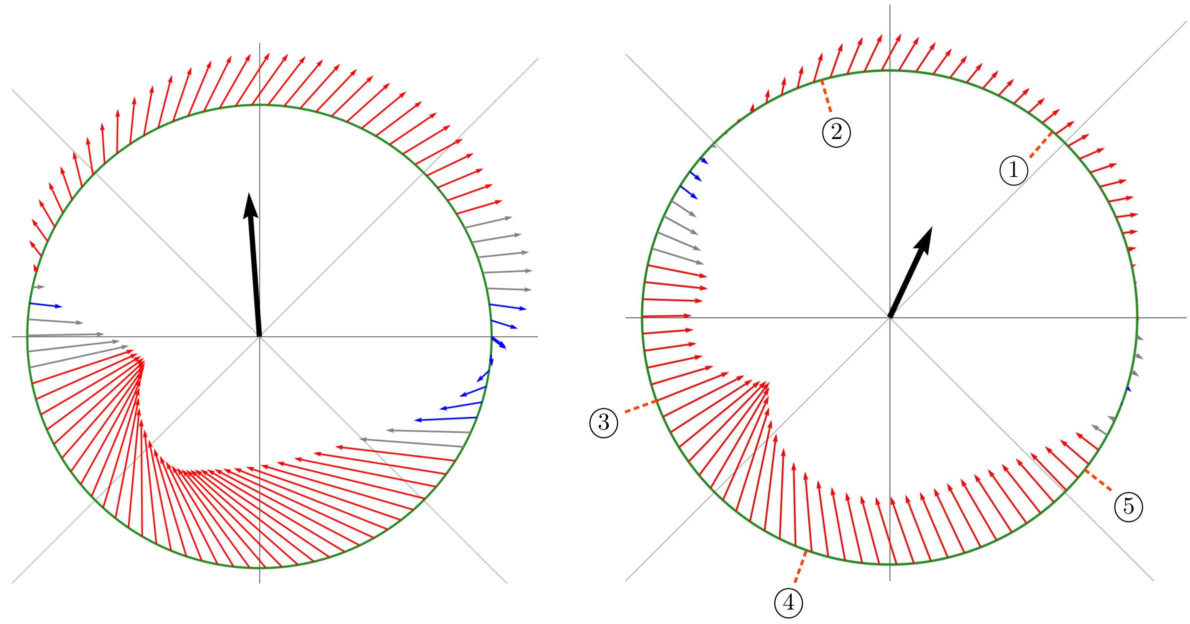


Figure 10.14: Resulting blade force over trajectory, black arrow represents the thrust.

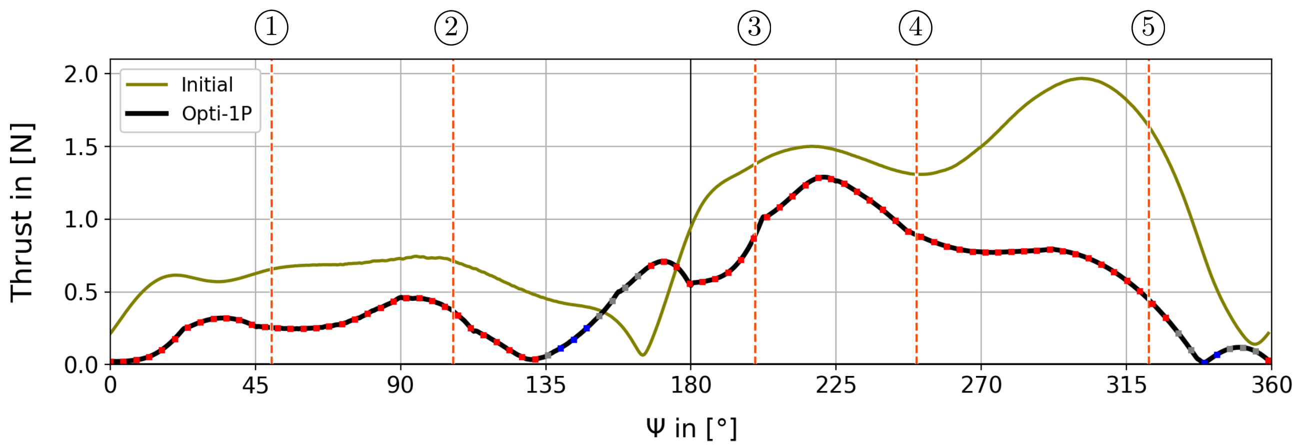


Figure 10.15: Thrust over azimuth angle Ψ.

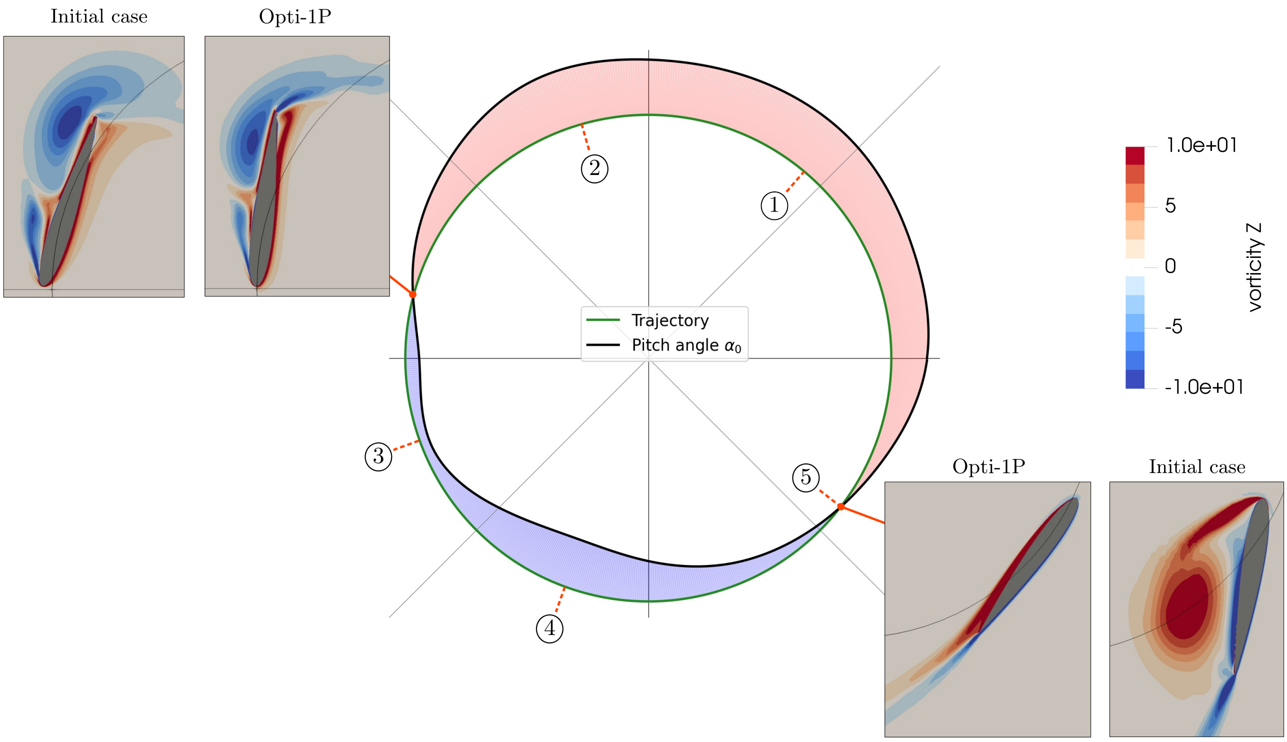
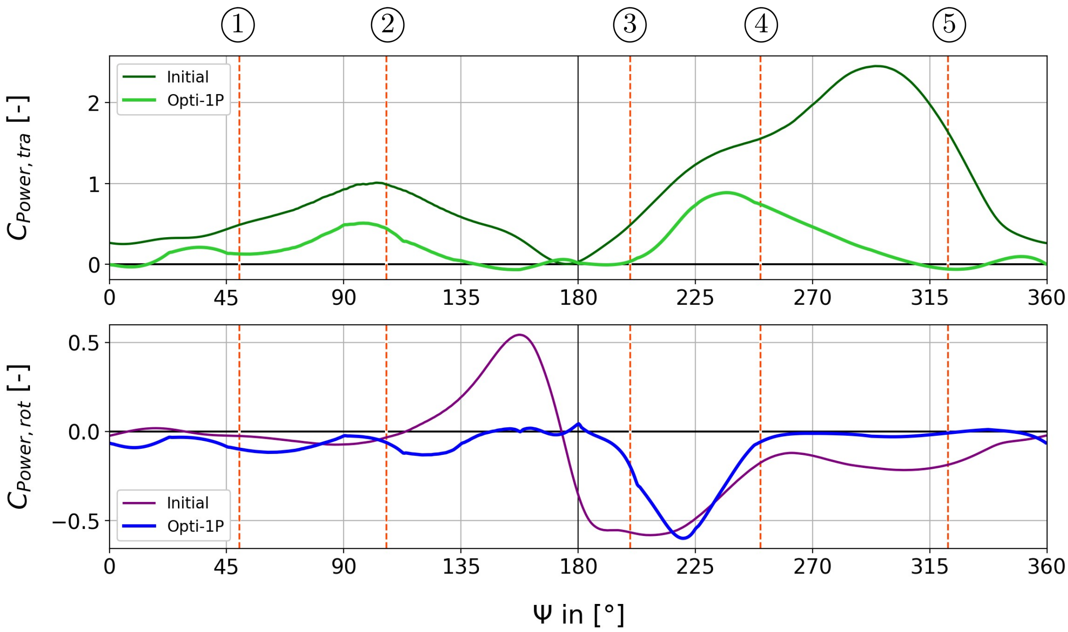


Figure 10.16: Pitching path over trajectory and vorticity, Opti-1P.



Pos.	Ψ
1	50°
2	110°
3	200°
4	250°
5	322°

Figure 10.17: Translation and rotation power coefficients, Opti-1P.

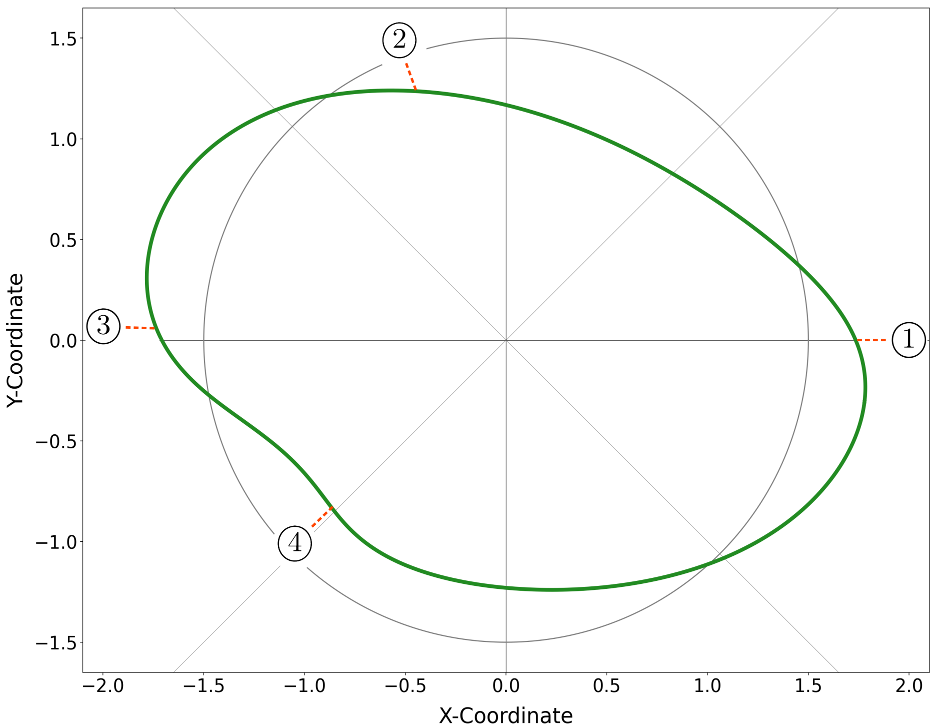
10.4.2 Opti-1TV

The trajectory for the 1-blade case is shown in Figure 10.18. The rotation begins at 1 with an almost flat ascending to 2, where the blade reaches its maximum pitch, and a large vortex occurs at the trailing edge. However, this vortex remains at the outer blade side and vanishes at 3. The coefficients are similar to the initial case during the first half of rotation, see Figure 10.22. The benefit of the first half trajectory is a better alignment of the blade forces, see Figure 10.19.

Due to the small trajectory radius and the pitching at 3 and 4, the blade performs a rapid rotation resulting in a force peak, which requires a lot of rotation power. The last part of the rotation is a circular movement, generating an almost constant blade force.

Although the force peaks require a lot of power (more than the initial case), the figure of merit is increased to $FoM_{1TV} = 0.49$ (+40%).

Table 10.4 lists characteristic values of the optimisation compared to its initial case.

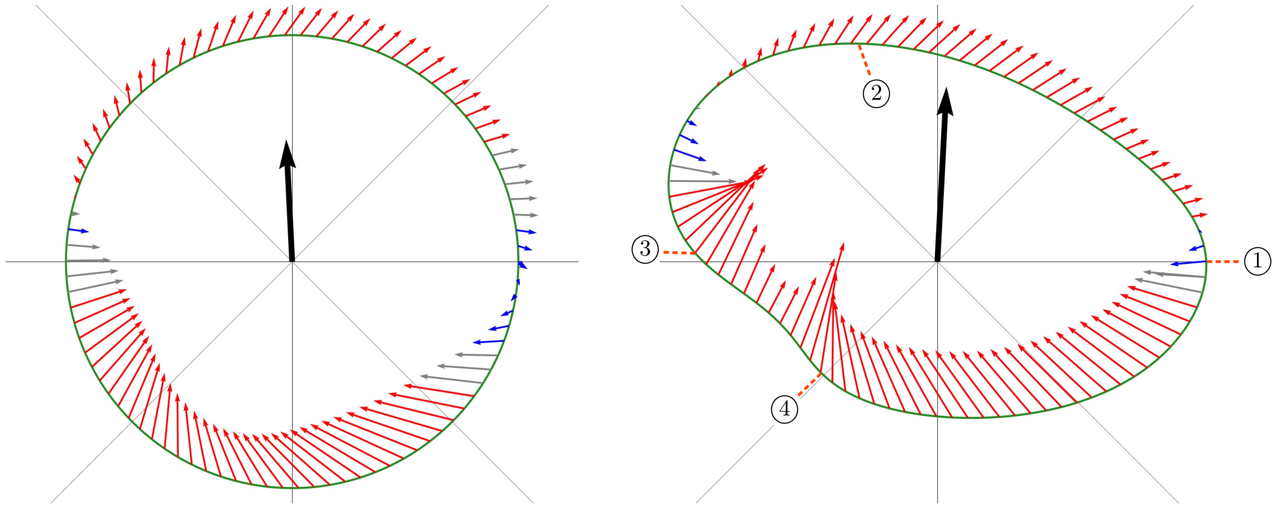


Pos.	Ψ
1	0°
2	110°
3	178°
4	224°

Figure 10.18: Trajectory of Opti-1TV, rotor width $w_R = 3.57$ m.

Table 10.4: Mean values for Opti-1TV.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.542 N	0.147 W	0.346 W	0.074 W	0.420 W	0.350	93 °
Opit-1TV	0.776 N	0.231 W	0.374 W	0.098 W	0.471 W	0.490	87 °
Δ	+43 %	+57 %	+8 %	+32 %	+12 %	+40 %	



(a) Initial case (b) Opti-1TV

Figure 10.19: Resulting blade force over trajectory, black arrow represents the thrust.

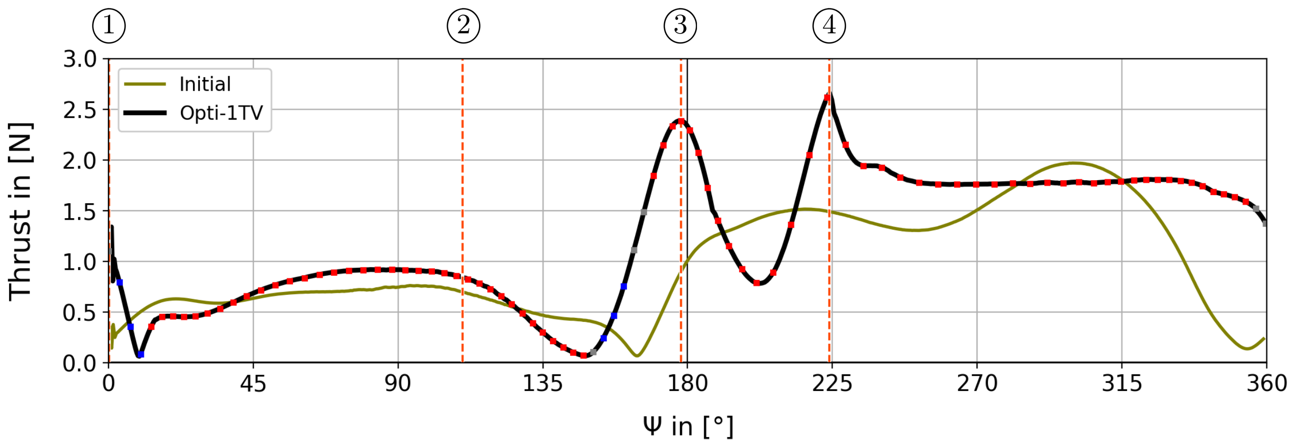


Figure 10.20: Thrust over azimuth angle Ψ.

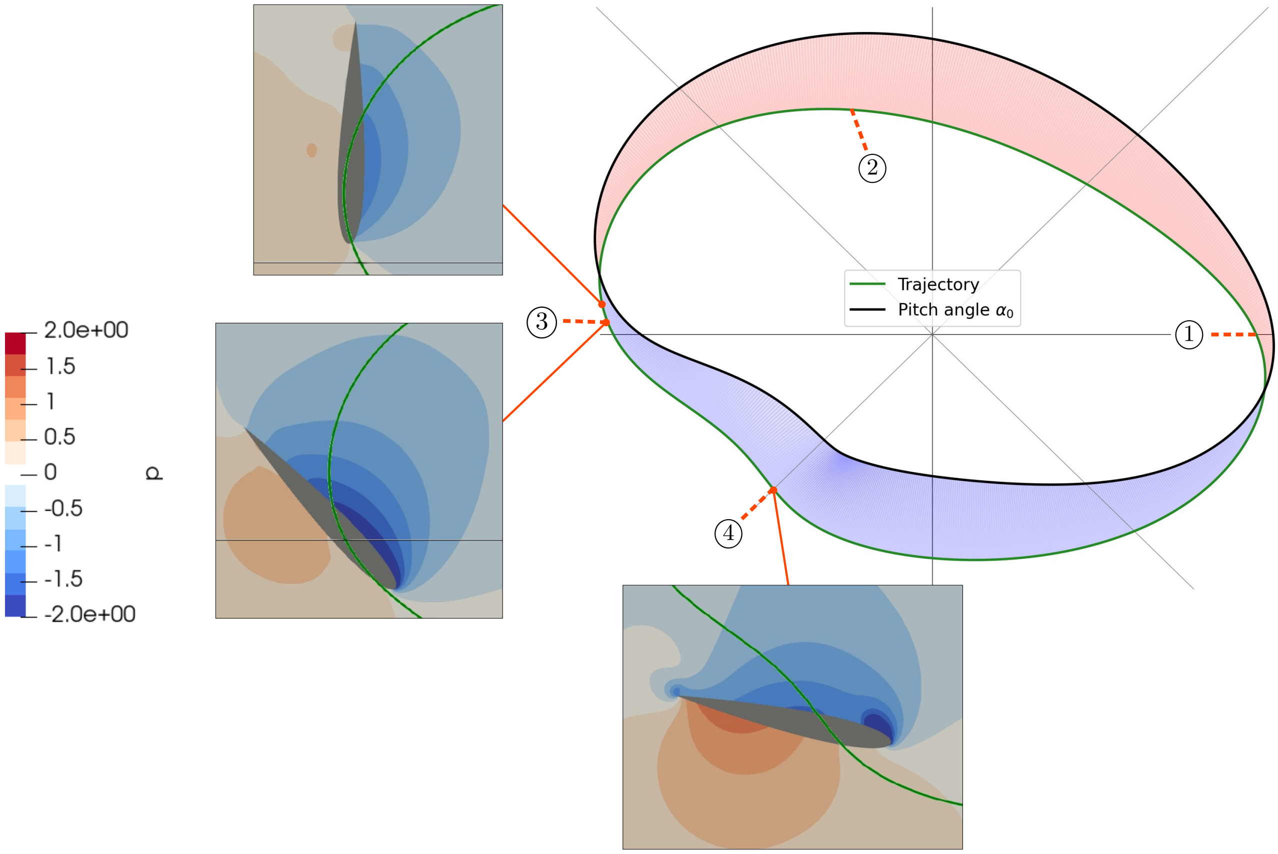
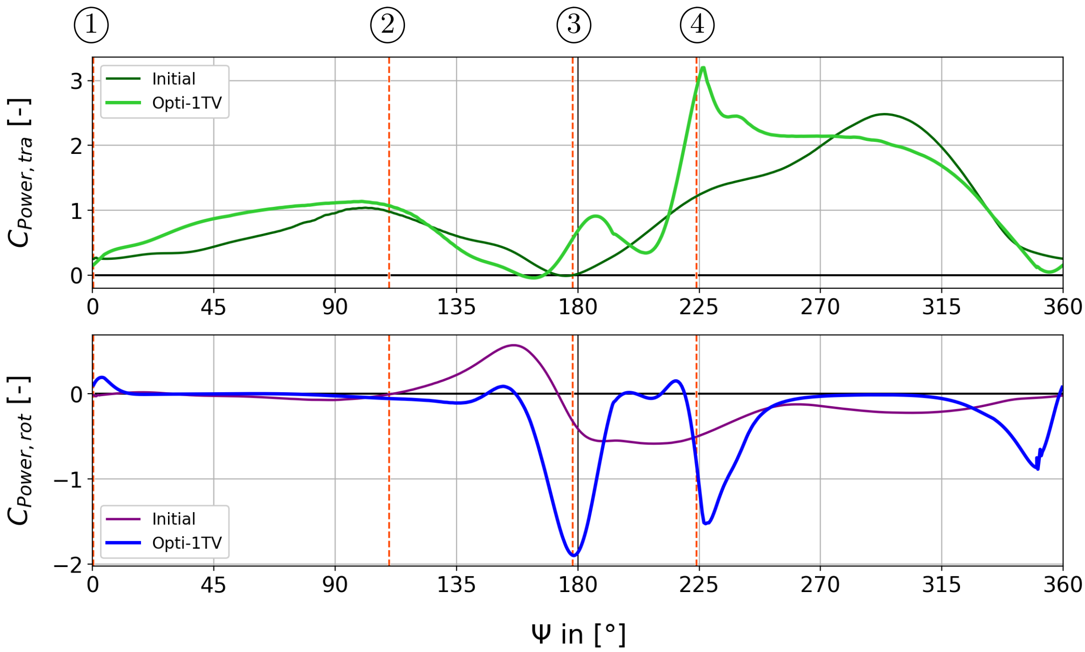


Figure 10.21: Pitching path over trajectory, Opti-1TV.



Pos.	Ψ
1	0°
2	110°
3	178°
4	224°

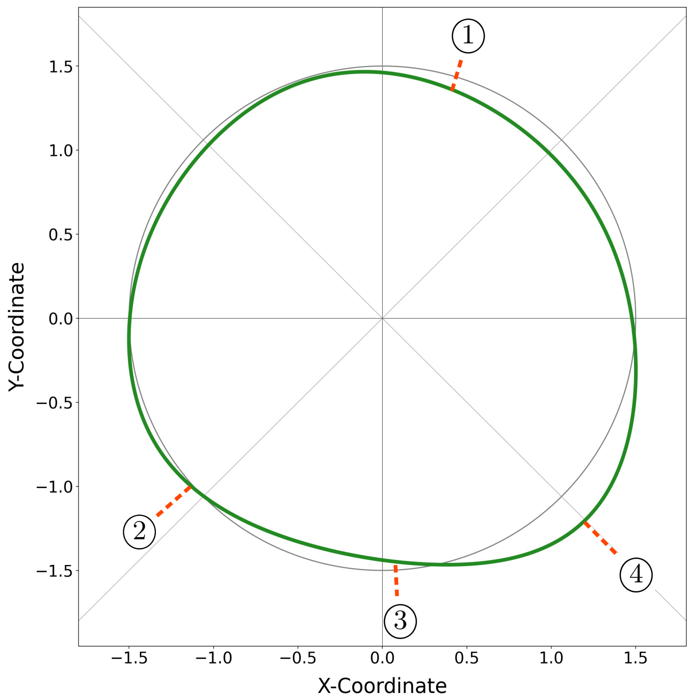
Figure 10.22: Translation and rotation power coefficients, Opti-1TV.

10.4.3 Opti-1BV

The trajectory's shape for this optimisation is quite close to a circle and has a small protrusion at ④, as shown in Figure 10.23. There is a clear waviness in the optimised pitching path, see Figure 10.24. As with Opti-1P, temporary lowering of the pitch also appears this time at ①, ② and ③. The result is that the leading edge vortex during the first half rotation is generated later. The vortex during the second half rotation does not occur at all. This leads, on the one hand, to a lower power consumption, see Figure 10.28. On the other hand, the lift is considerably reduced. After the second pitch reduction at ②, a force peak in the vertical direction occurs. As a result of the protrusion at ④, the blade forces are upturned compared to the initial case (see Figure 10.25).

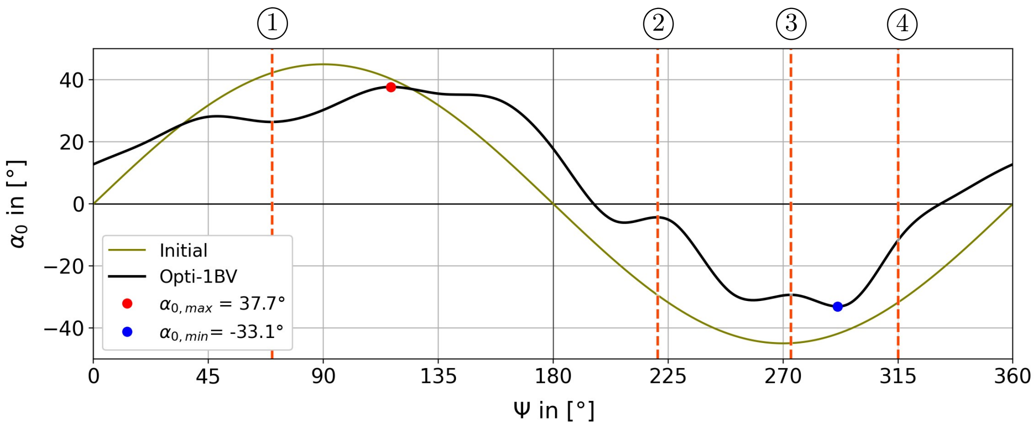
The figure of merit for this optimisation is $\mathrm{FoM}_{1BV} = 0.63$ (+81%).

Table 10.5 lists characteristic values of the optimisation compared to its initial case.



Pos.	Ψ
①	70°
②	221°
③	273°
④	315°

Figure 10.23: Trajectory of Opti-1BV, rotor width $w_R = 3.00$ m.

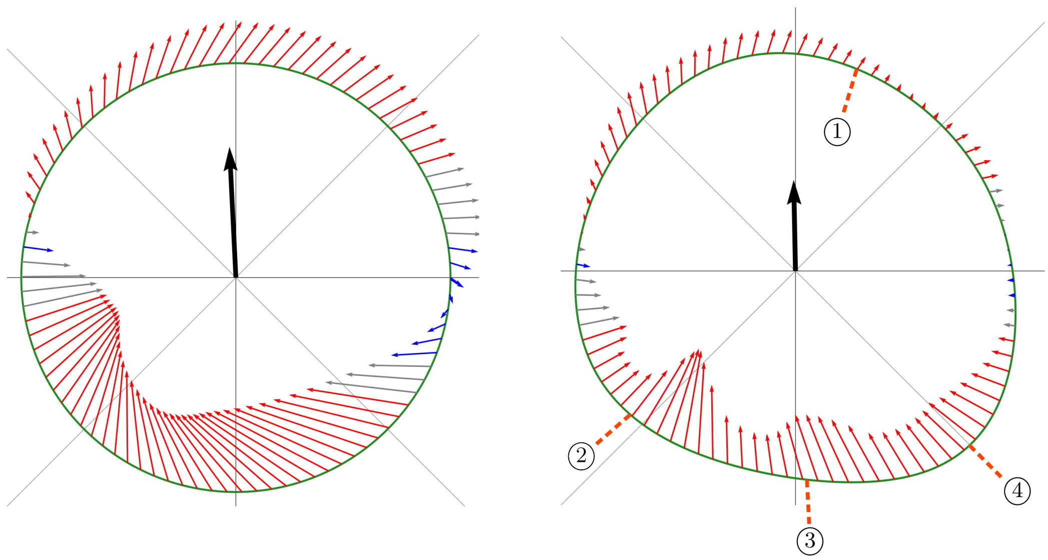


Pos.	Ψ
①	70°
②	221°
③	273°
④	315°

Figure 10.24: Pitching angle α_0 over azimuth angle Ψ.

Table 10.5: Mean values for Opti-1BV.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.542 N	0.147 W	0.346 W	0.074 W	0.420 W	0.350	93 °
Opit-1BV	0.368 N	0.082 W	0.097 W	0.033 W	0.130 W	0.632	91 °
Δ	-32 %	-44 %	-72 %	-55 %	-69 %	+81 %	



(a) Initial case (b) Opti-1BV

Figure 10.25: Resulting blade force over trajectory, black arrow represents the thrust.

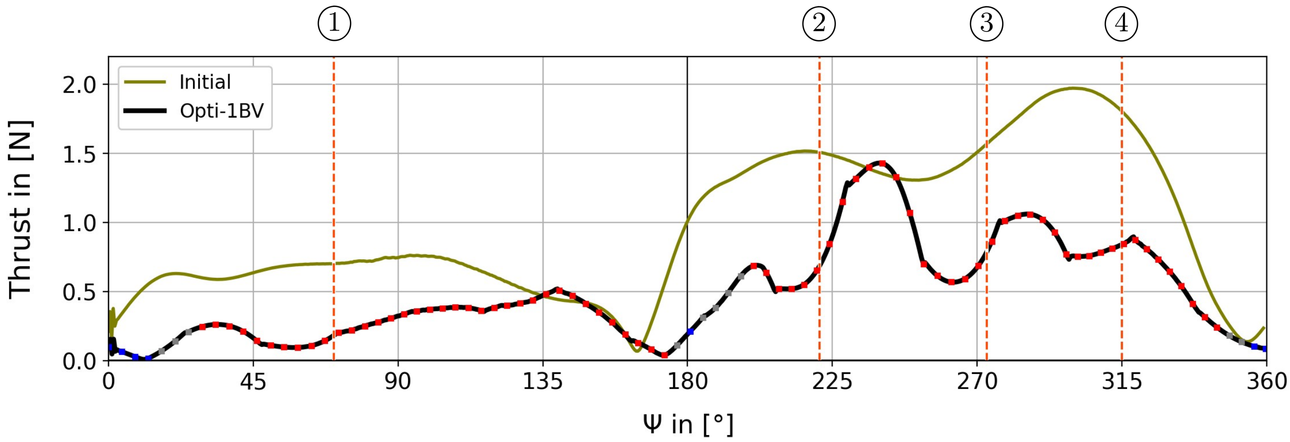


Figure 10.26: Thrust over azimuth angle Ψ.

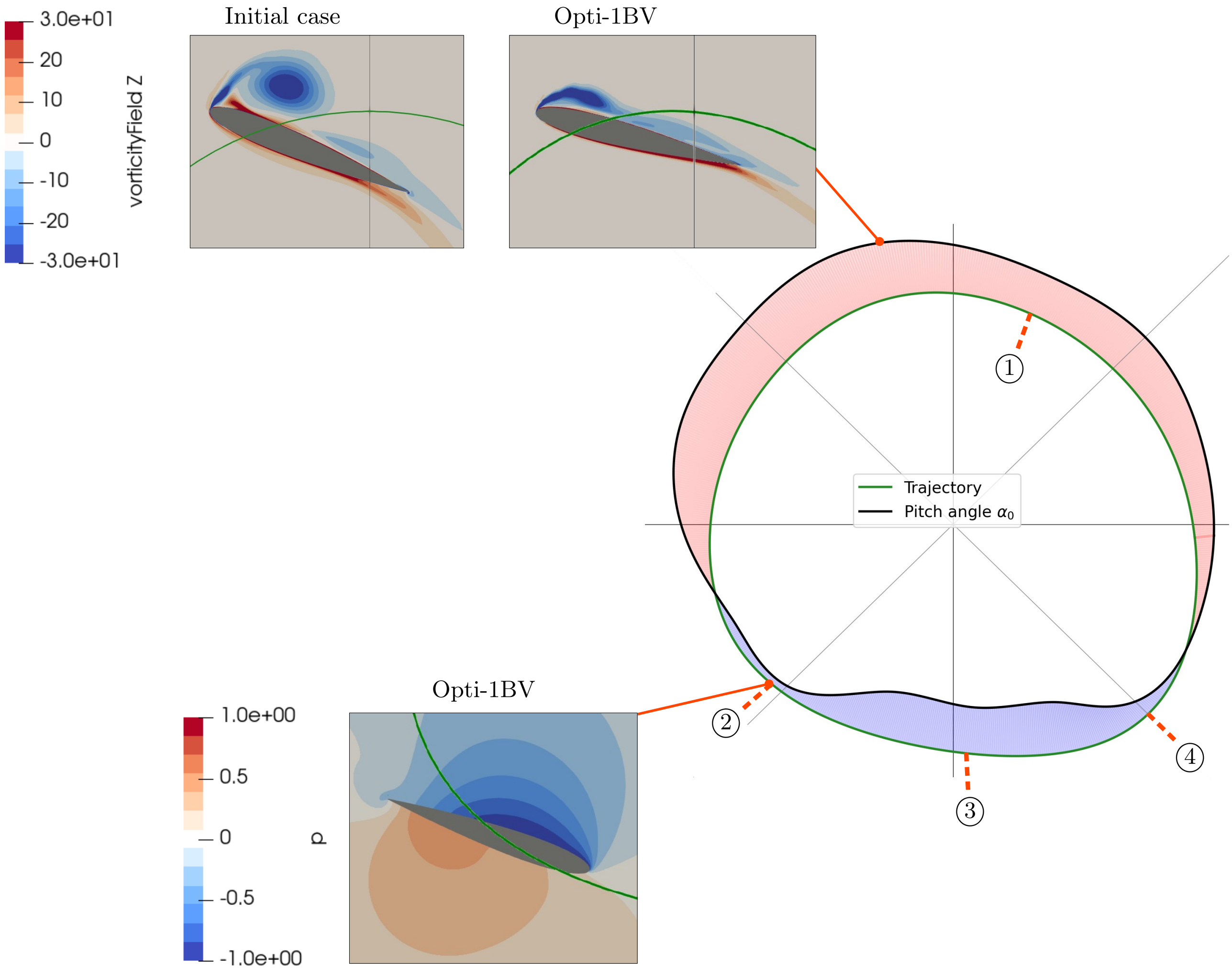
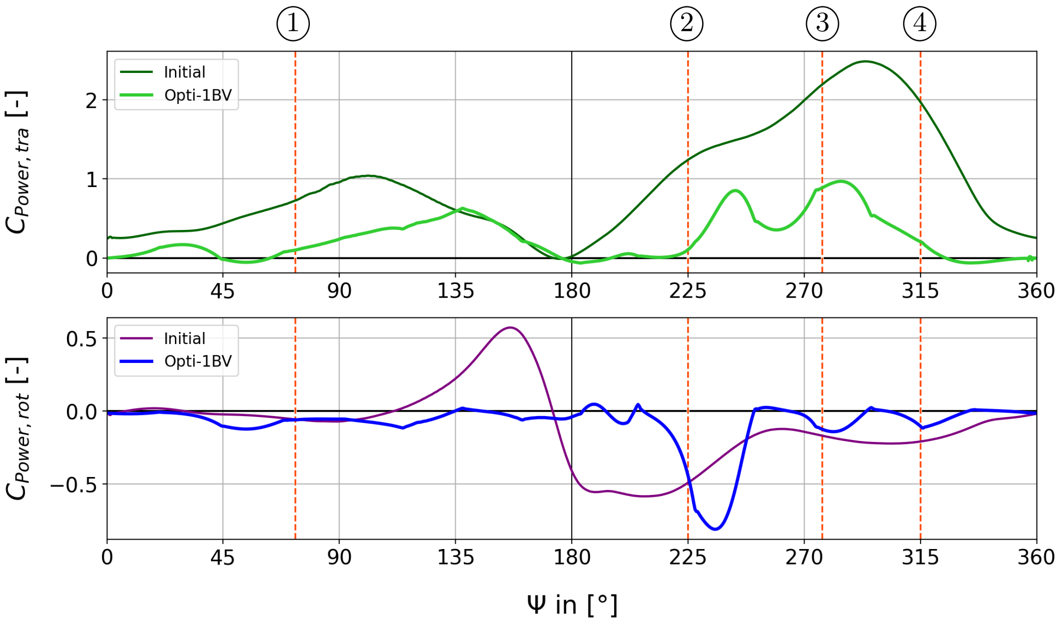


Figure 10.27: Pitching path over trajectory, Opti-1BV.



Pos.	Ψ
1	73°
2	221°
3	273°
4	315°

Figure 10.28: Translation and rotation power coefficients, Opti-1BV.

10.5 Two blades

10.5.1 Opti-2P

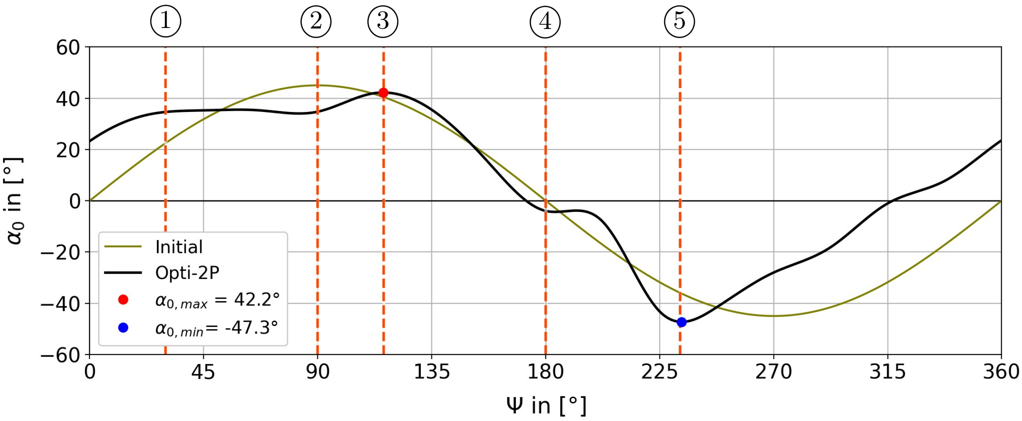
The pitching path for this two-blade optimisation is similar to the one-blade case, both shown in Figure 10.29. The curve now has a distinct plateau between ① and ②. This movement avoids a leading edge vortex but with a loss of lift. But more essential, this movement avoids the detachment of a vortex, which in turn induces a vortex on the following blade in the initial case, see Figure 10.33. The small pitch up to its maximum produces a small lift, ③. The following section, between ③ and ④, is characterised by a steep reduction of the pitching angle leading to an adverse force directed aginst the global thrust.

The second half rotation starts with a short constant pitch followed by a second steep pitching to its minimum at ⑤. During this movement, the blade induces a great force peak, see Figure 10.31. In contrast, the force distribution of the initial case shows a loss of lift near position ⑤. The pitch after ⑤ increases nearly linear like the one-blade case. As a result, the leading edge vortex is smaller, and thus there is less influence on the following blade, which leads to a considerably less required power, see Figure 10.34. The linear increasing pitching path also leads to an alignment of the blade forces closer to the global thrust.

Two further optimisations for the two-blade pitching case are carried out; one with a Reynolds number Re = 100 000 and one with Re = 200 000. Figure 10.35 shows the resulting pitching paths, which are pretty similar. A slight shift of the maximum pitch to higher azimuth angles can be identified.

Figure 10.35 shows the figure of merit for the optimisation with three different Reynolds numbers, where each run has a distinct optimum for a particular flow velocity. The increase of the maximum FoM is moderate; 6% and 3%. However, the figure of merit decreases substantially if another Reynolds number is defined than the assigned one.

Table 10.6 lists characteristic values of the optimisation compared to its initial case.



Pos.	Ψ
①	30°
②	90°
③	116°
④	180°
⑤	233°

Figure 10.29: Pitching angle α_0 over azimuth angle Ψ.

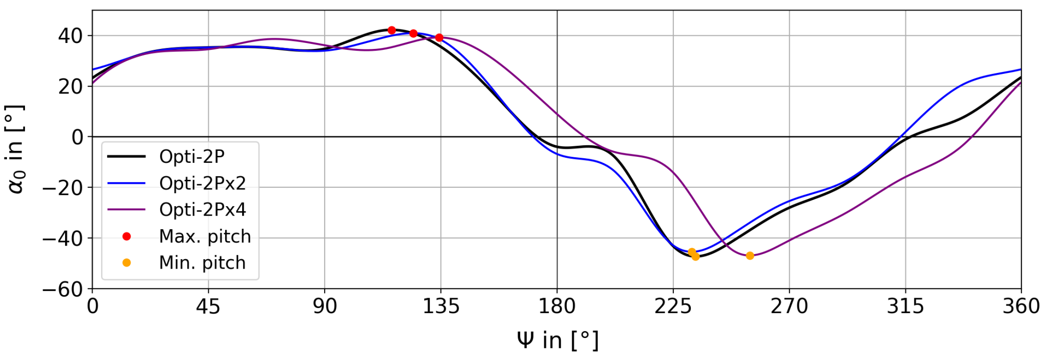
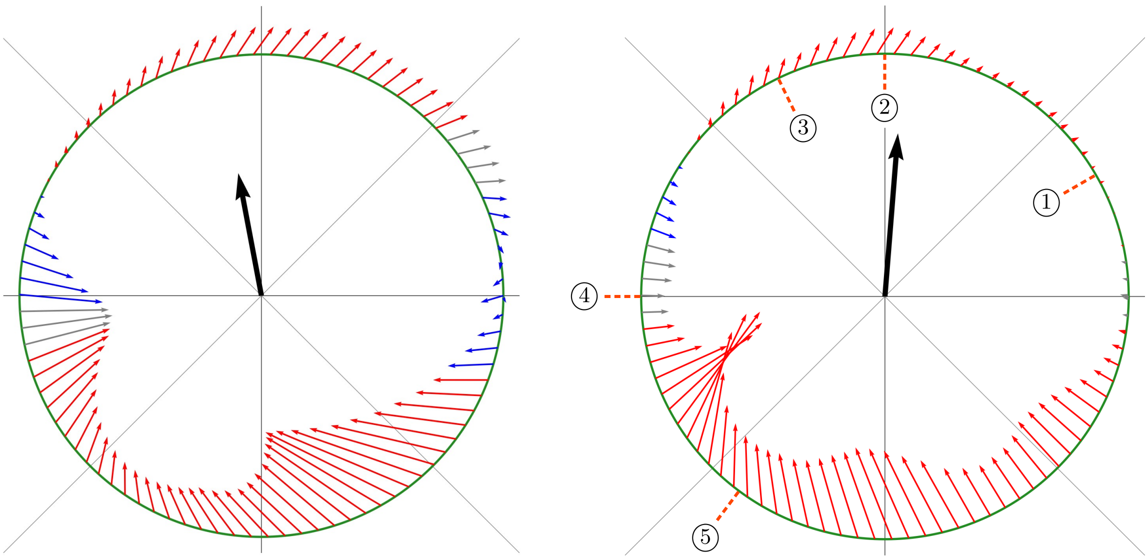


Figure 10.30: Pitching angle α_0 over azimuth angle Ψ for multiple Reynolds numbers. Opti-2P: Re = 50 000, Opti-2Px2: Re = 100 000, Opti-2Px4: Re = 200 000



(a) Initial case

(b) Opti-2P

Figure 10.31: Resulting blade force over trajectory, black arrow represents the thrust.

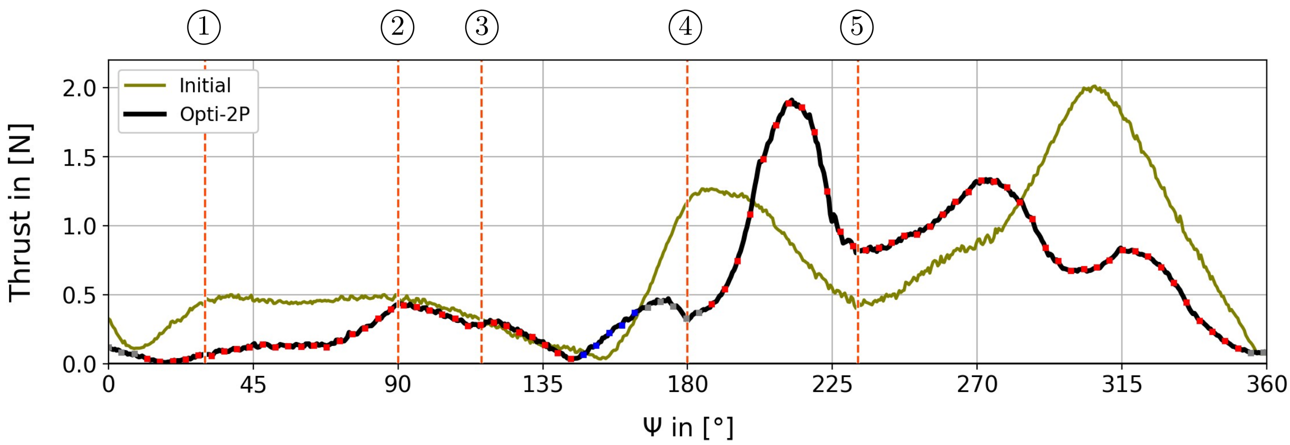


Figure 10.32: Thrust over azimuth angle Ψ.

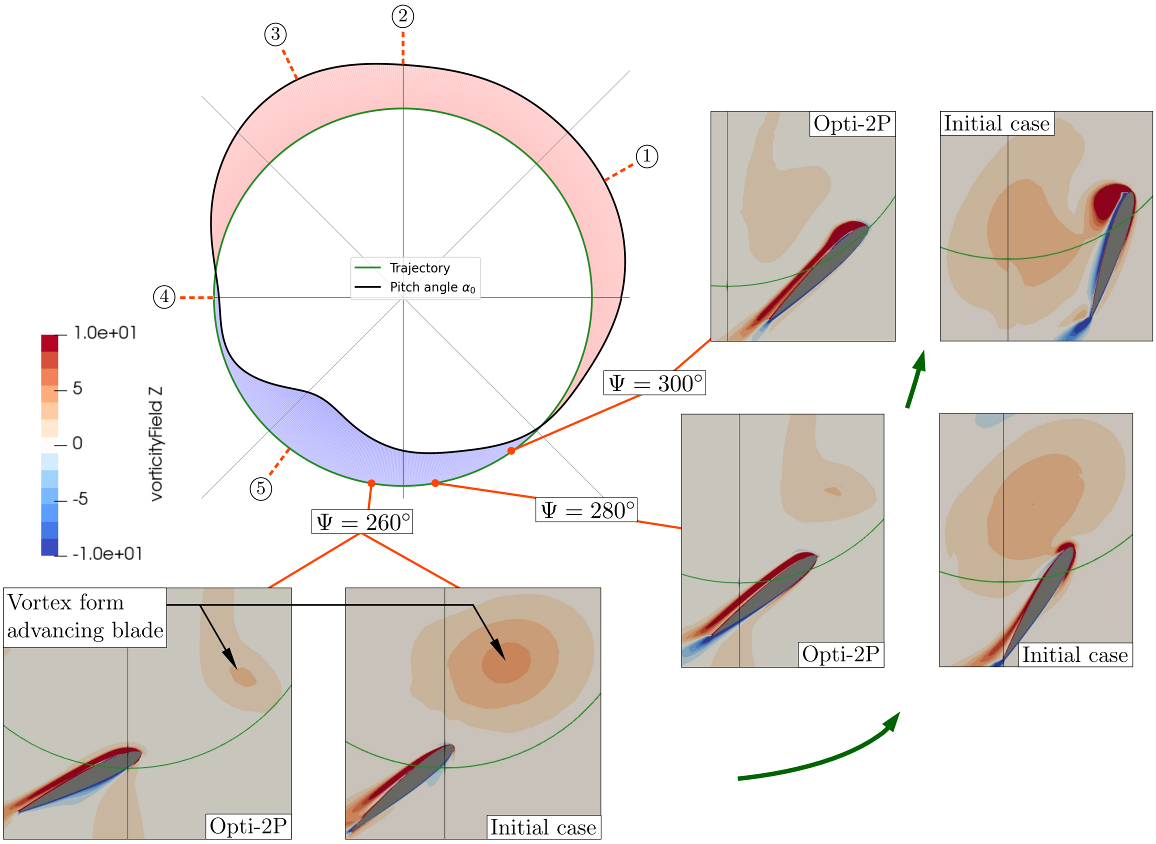
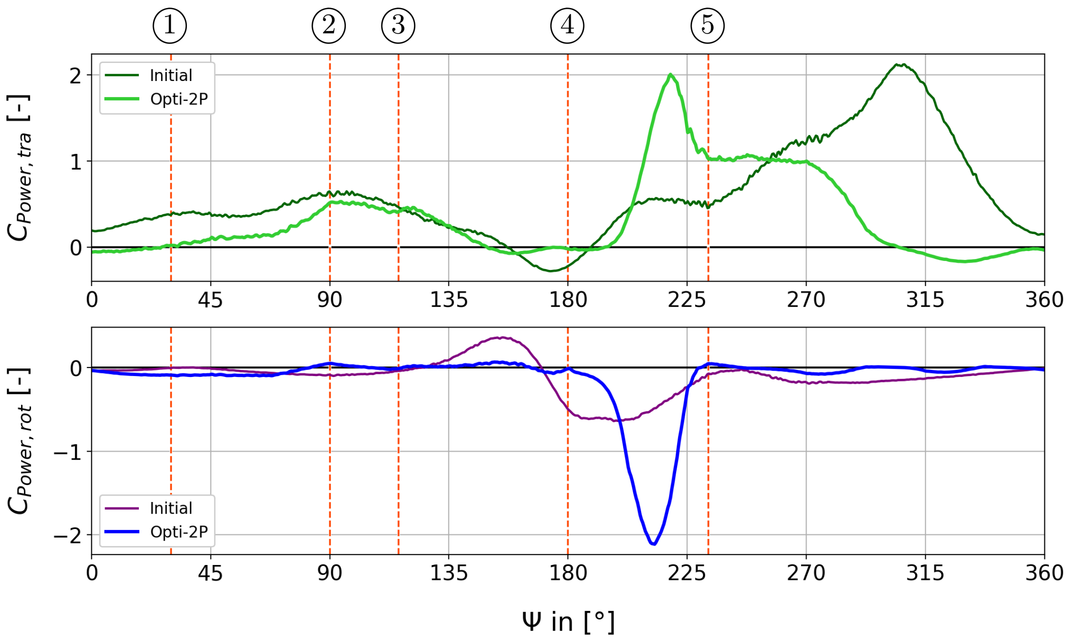


Figure 10.33: Pitching path α_0 over trajectory and vorticity in z-direction.



Pos.	Ψ
1	30°
2	90°
3	116°
4	180°
5	233°

Figure 10.34: Translation and rotation power coefficients, Opti-2P.

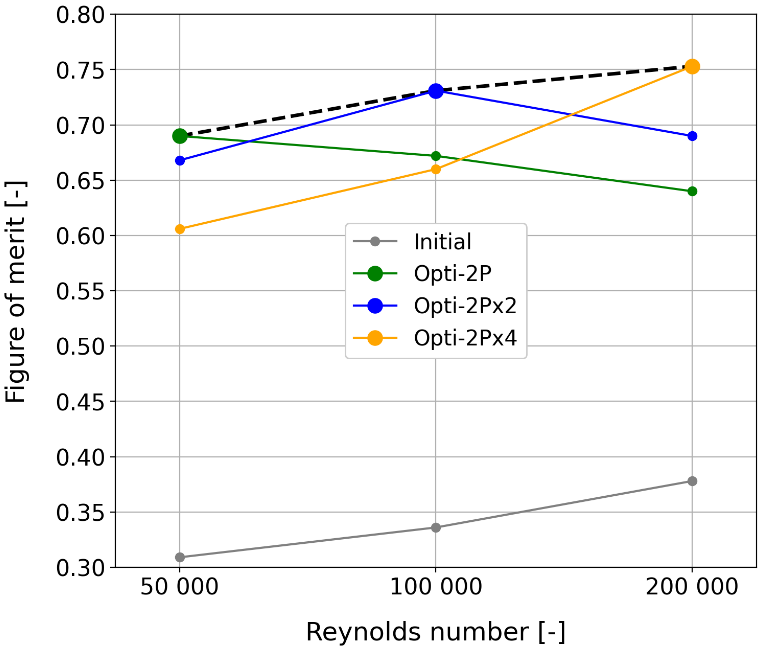


Figure 10.35: Figure of merit over Reynolds number.

Figure 10.36: Figure of merit for various Reynolds number.

Run	Reynolds number $\cdot 10^3$ 50	100	200
Initial	0.309	0.336	0.378
Opti-2P	**0.690**	0.672	0.640
Opti-2Px2	0.668	**0.731**	0.690
Opti-2Px4	0.606	0.660	**0.753**

Table 10.6: Mean values for the pitching optimisation with two blades. The deviations refer in each case to the initial case.

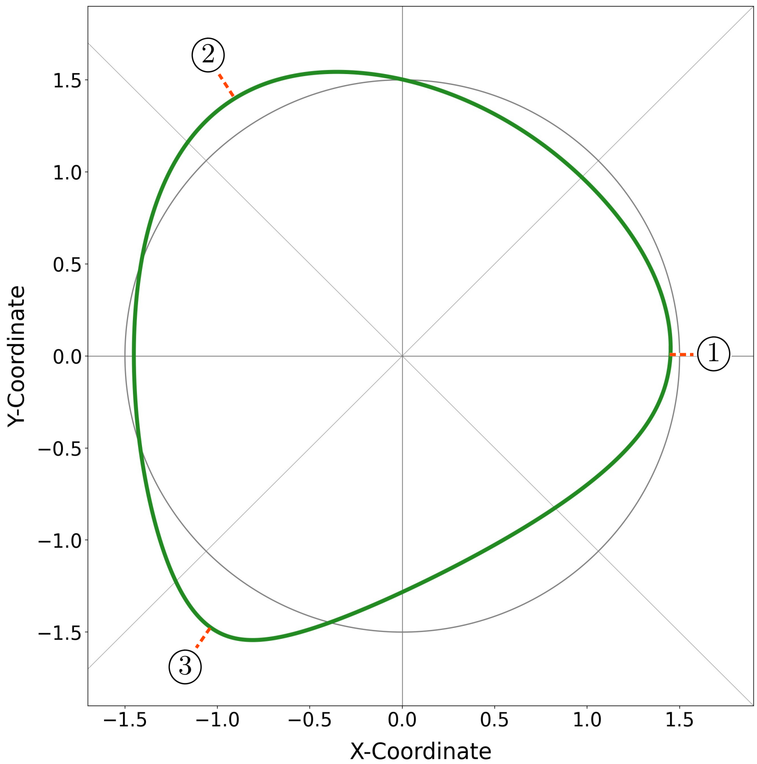
Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.615 N	0.178 W	0.462 W	0.114 W	0.576 W	0.309	101 °
Opit-2P	0.802 N	0.265 W	0.273 W	0.109 W	0.383 W	0.690	85 °
Δ	+30 %	+49 %	-41 %	-4 %	-34 %	+123 %	
Initialx2	2.41 N	1.38 W	3.32 W	0.779 W	4.10 W	0.336	105 °
Opit-2PVx2	3.18 N	2.09 W	2.13 W	0.727 W	2.86 W	0.731	77 °
Δ	+32 %	+51 %	-36 %	-7 %	-30 %	+118 %	
Initial	11.4 N	14.1 W	30.7 W	6.67 W	37.3 W	0.378	99 °
Opit-2PVx4	13.4 N	18.9 W	18.5 W	6.70 W	25.2 W	0.753	98 °
Δ	+66 %	+34 %	-40 %	+0 %	-33 %	+99 %	

10.5.2 Opti-2TV

The shape of the trajectory is similar to a triangle with its vertices at 1, 2 and 3, see Figure 10.37. The blade produces mostly adverse lift during the first edge, as shown in Figure 10.38. Towards position 3, the blade performs a nearly vertical movement, and again only adverse lift is generated. The interaction of a small trajectory radius and the minimum pitch angle at 3 results in a fast blade rotation, and a huge force peak occurs. The last section of the rotation is an approximately linear movement, where the blade forces are aligned in the direction of the thrust.

The figure of merit is increased to $FoM_{2TV} = 0.42$ (+32%).

Table 10.7 lists characteristic values of the optimisation compared to its initial case.

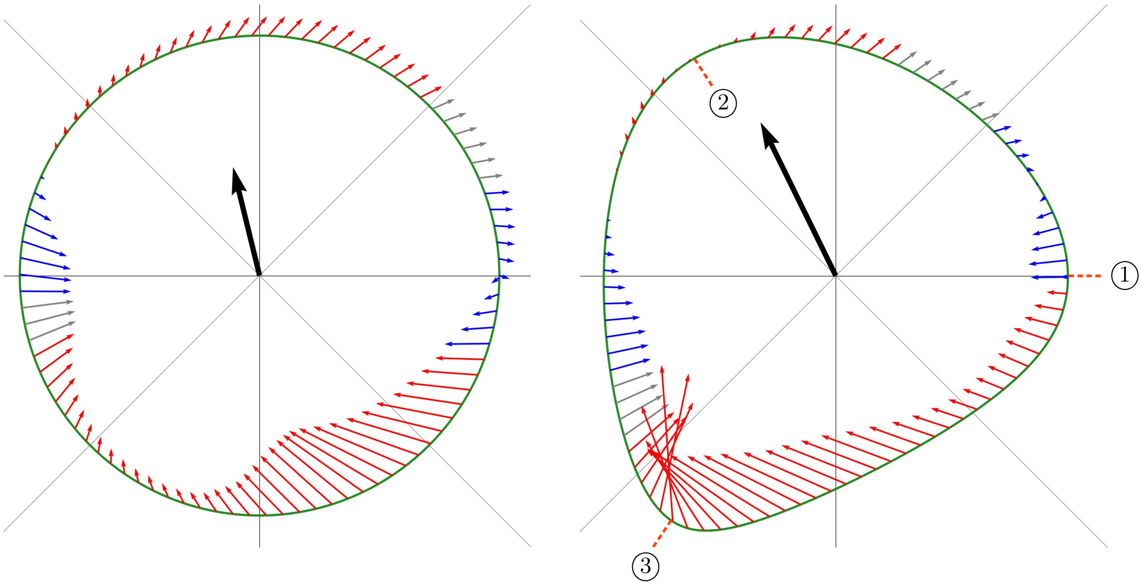


Pos.	Ψ
1	0°
2	123°
3	236°

Figure 10.37: Trajectory of Opti-2TV, rotor width $w_R = 2.90$ m.

Table 10.7: Mean values for Opti-2TV.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.703 N	0.217 W	0.555 W	0.125 W	0.680 W	0.320	104 °
Opit-2TV	1.08 N	0.421 W	0.740 W	0.260 W	1.00 W	0.421	117 °
Δ	+54 %	+94 %	+33 %	+108 %	+47 %	+32 %	



(a) Initial case (b) Opti-2TV

Figure 10.38: Resulting blade force over trajectory, black arrow represents the thrust.

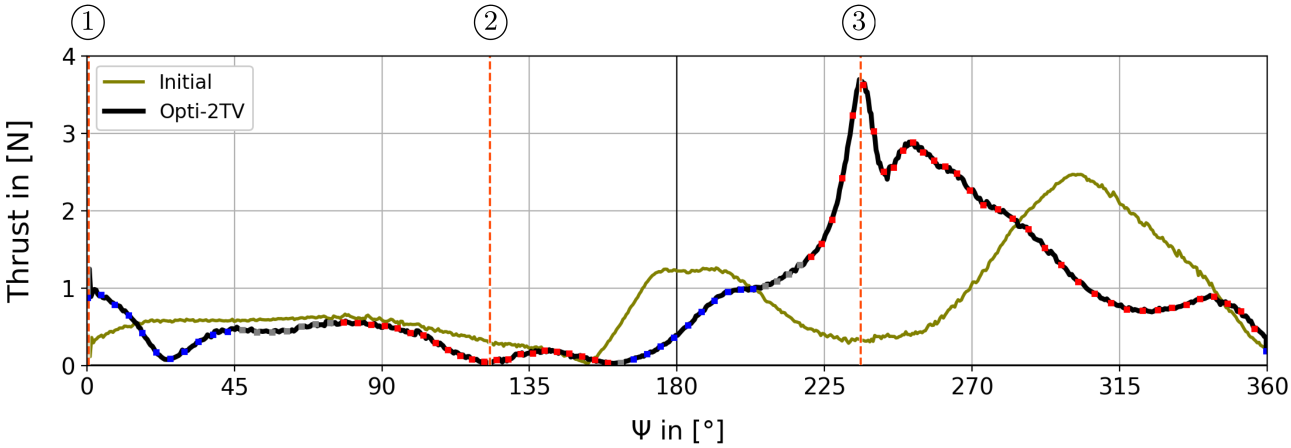


Figure 10.39: Thrust over azimuth angle Ψ.

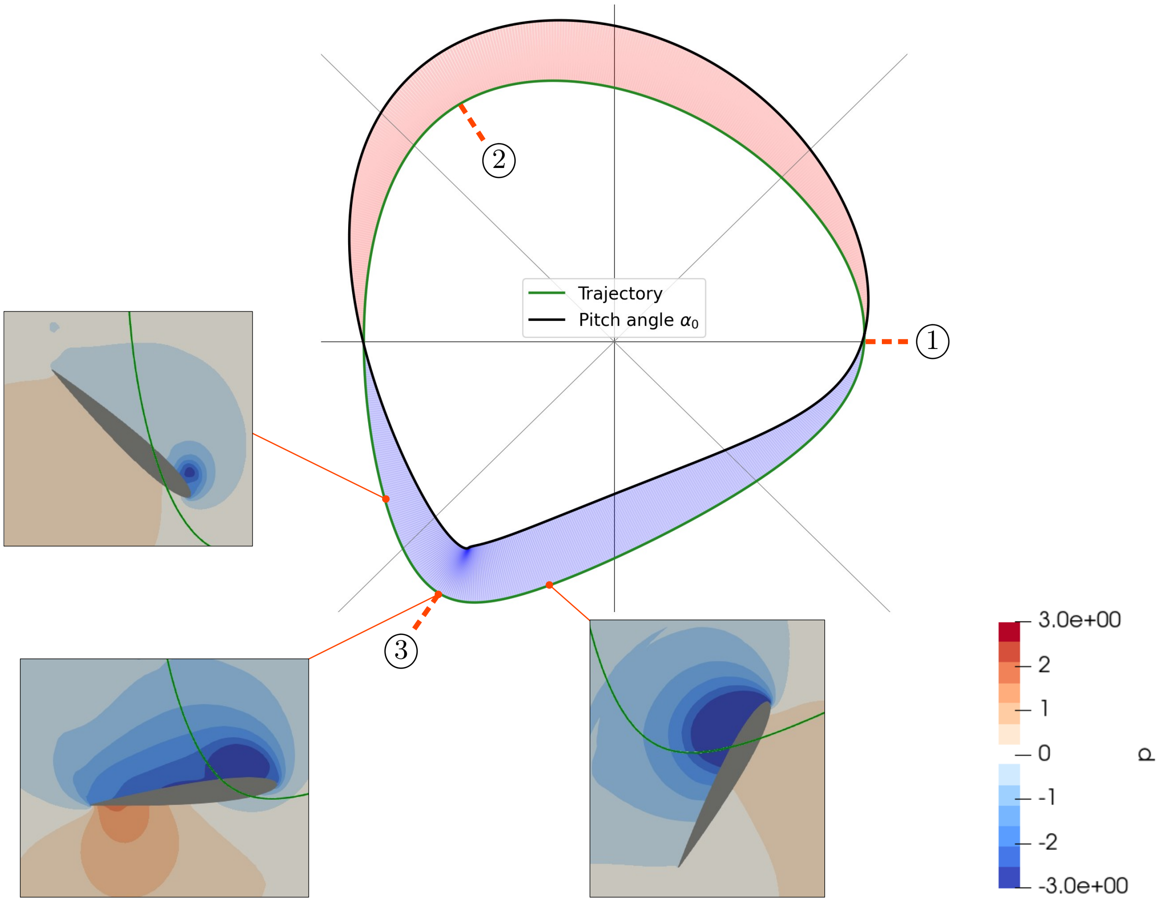
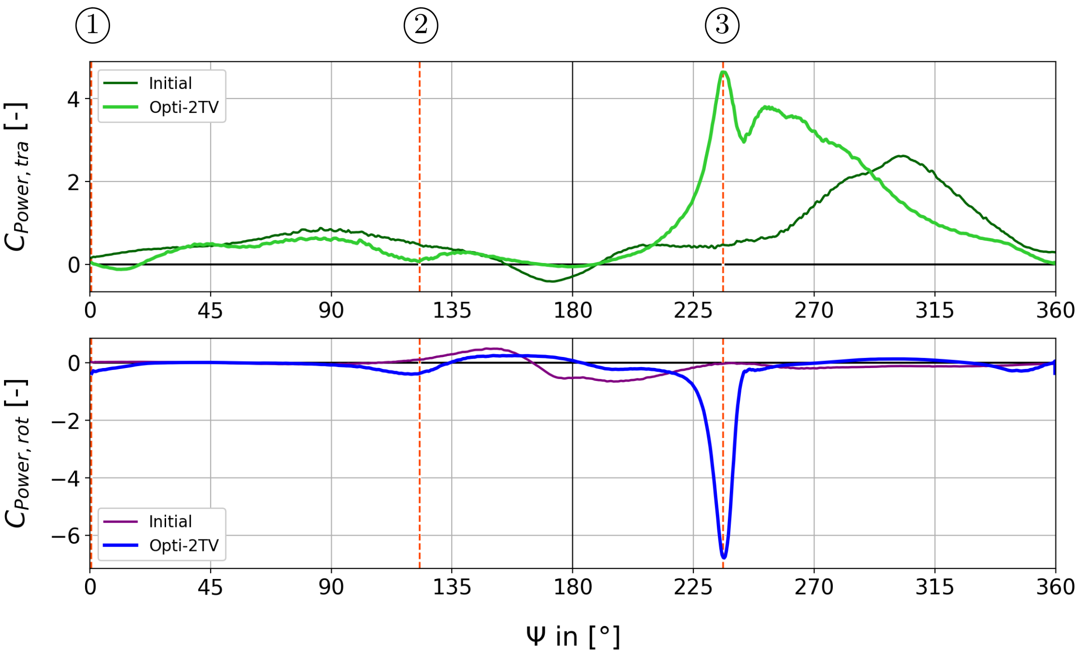


Figure 10.40: Pitching path over trajectory, Opti-2TV.



Pos.	Ψ
1	0°
2	123°
3	236°

Figure 10.41: Translation and rotation power coefficients, Opti-2TV.

10.5.3 Opti-2BV

Figure 10.42 shows the optimisation trajector, which tends to be a quad shape. Although the maximum pitch angle is reached during the first quarter rotation, from ① to ②, the blade generates nearly no lift, see Figure 10.46. Between ② and ③, the blade generates a small amount of lift. After that, the blade performs an almost vertical movement downwards with no lift. At ④, a force peak is generated due to a narrow trajectory and an immediate increase of the pitching. The blade forces remain at a high level until a second force peak occurs at ⑤. The pitching path increases rapidly, ending in an overshoot at ⑥, see Figure 10.44. This movement also generates an elemental blade force, mostly aligned with the global thrust.

The blade motion inclines the downwash by about 20° to the y-axis. As a result, the width changes, and so does the reference surface through which the flow passes changes. The correct width is estimated based on the downwash as shown in the velocity field in Figure 10.43. A value of $w_{R,corr} = 3.0$ m is assumed. The corrected figure of merit is $\text{FoM}_{2BV,corr} = 0.75$ (+136%).

Table 10.8 lists characteristic values of the optimisation compared to its initial case.

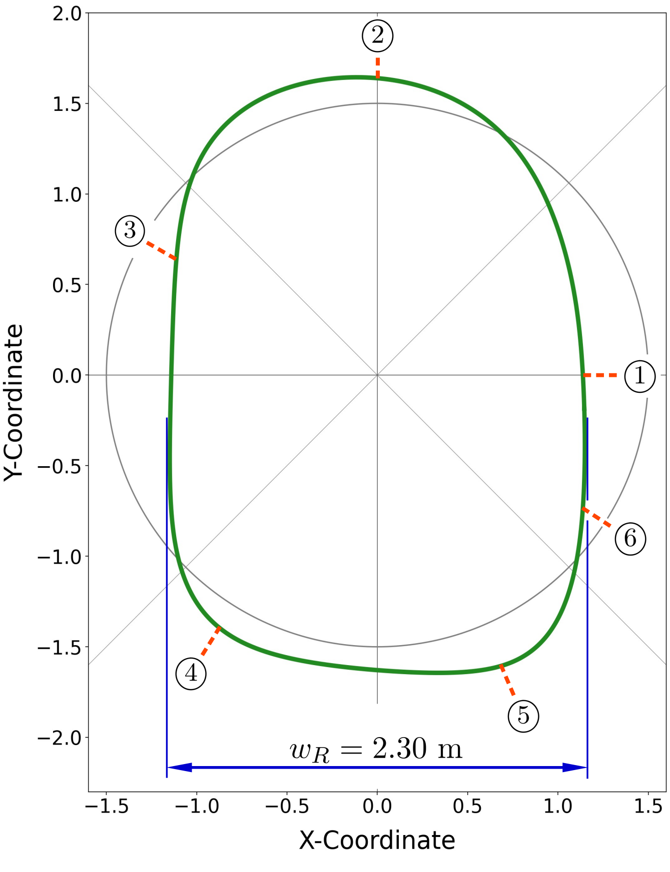


Figure 10.42: Trajectory of Opti-2BV; rotor width $w_R = 2.30$ m.

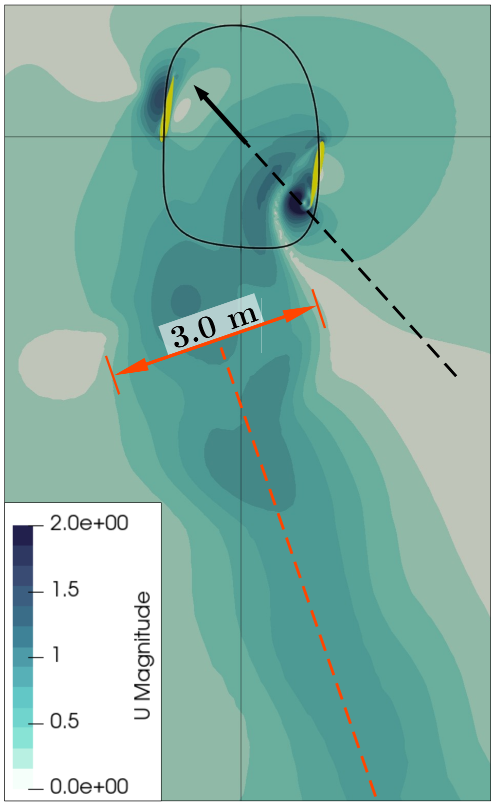
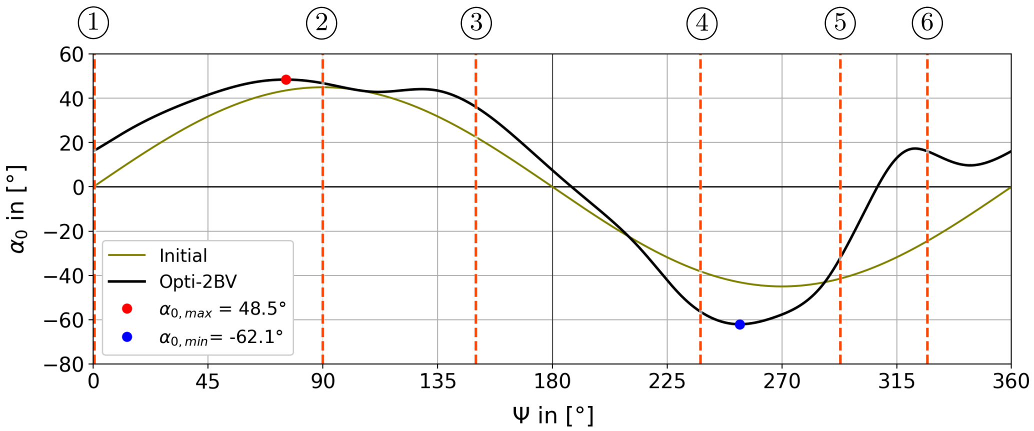


Figure 10.43: Magnitude of velocity.



Pos.	Ψ
①	0°
②	90°
③	150°
④	238°
⑤	293°
⑥	327°

Figure 10.44: Pitching angle α_0 over azimuth angle Ψ, Opti-2BV.

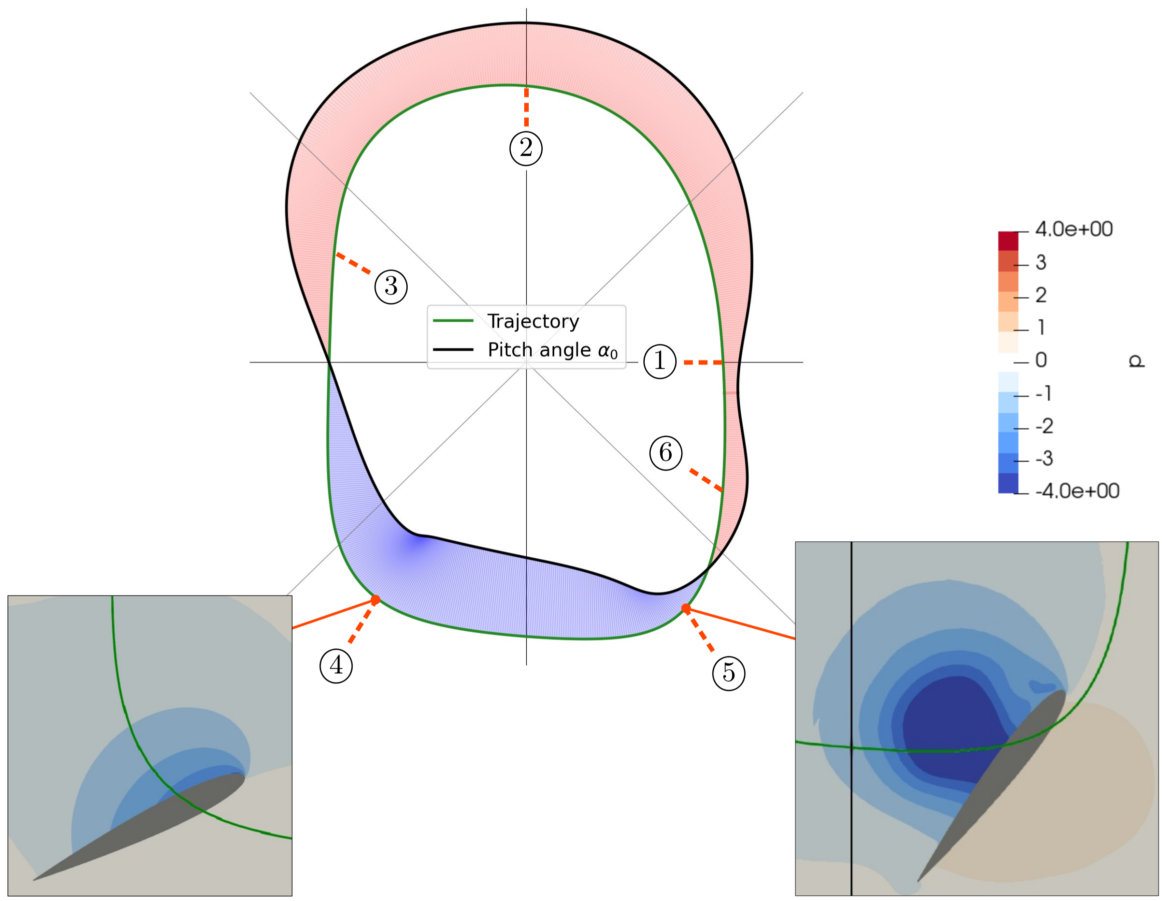
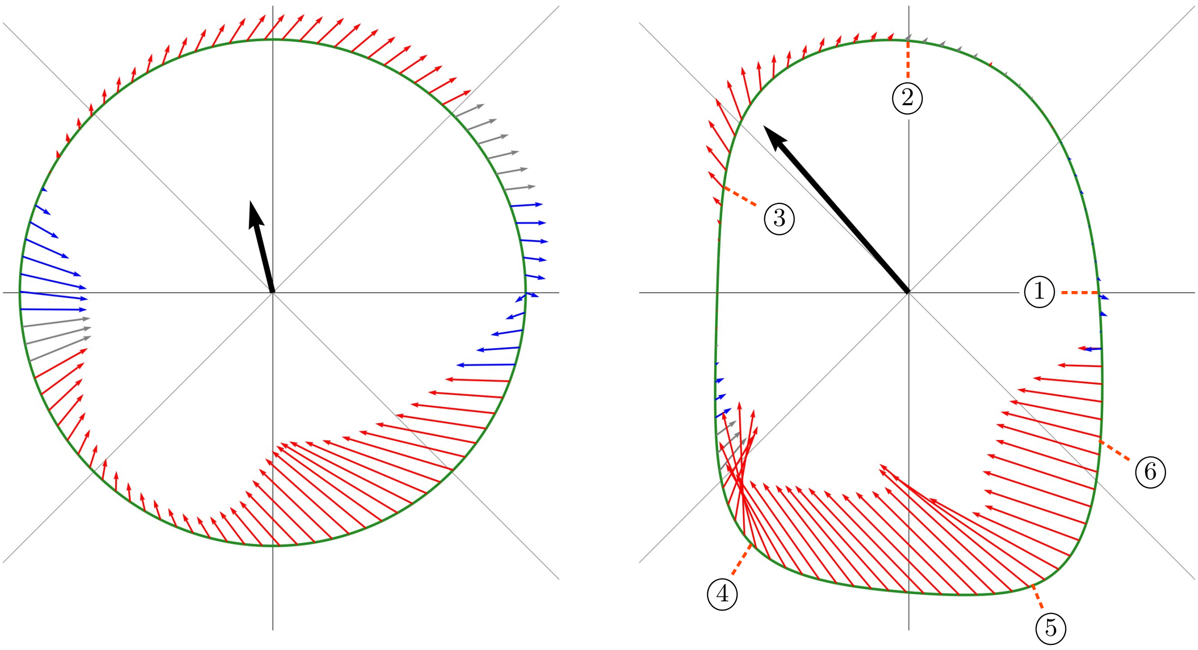


Figure 10.45: Pitching path over trajectory, Opti-2BV.



(a) Initial case

(b) Opti-2BV

Figure 10.46: Resulting blade force over trajectory, black arrow represents the thrust.

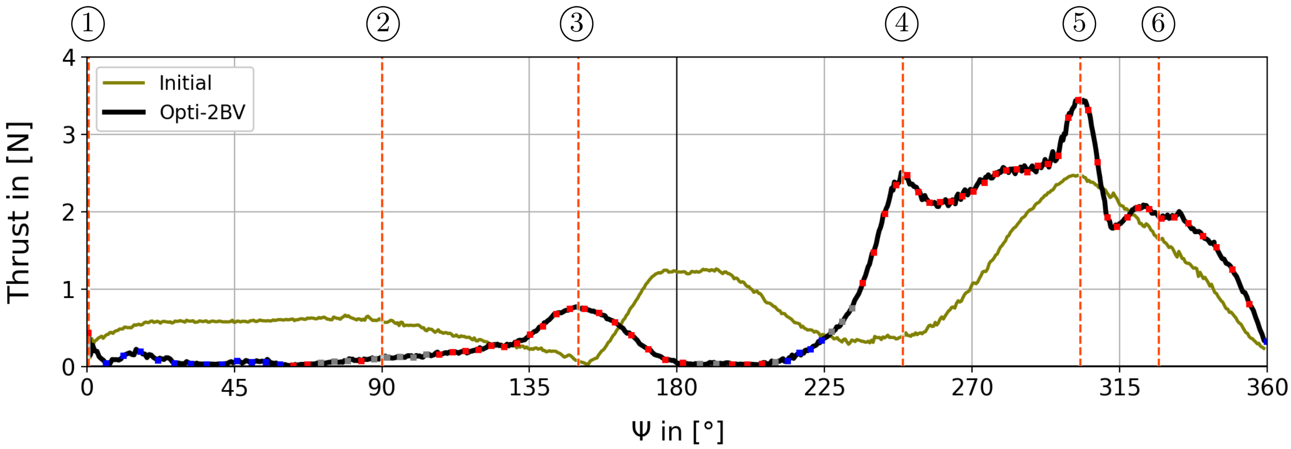
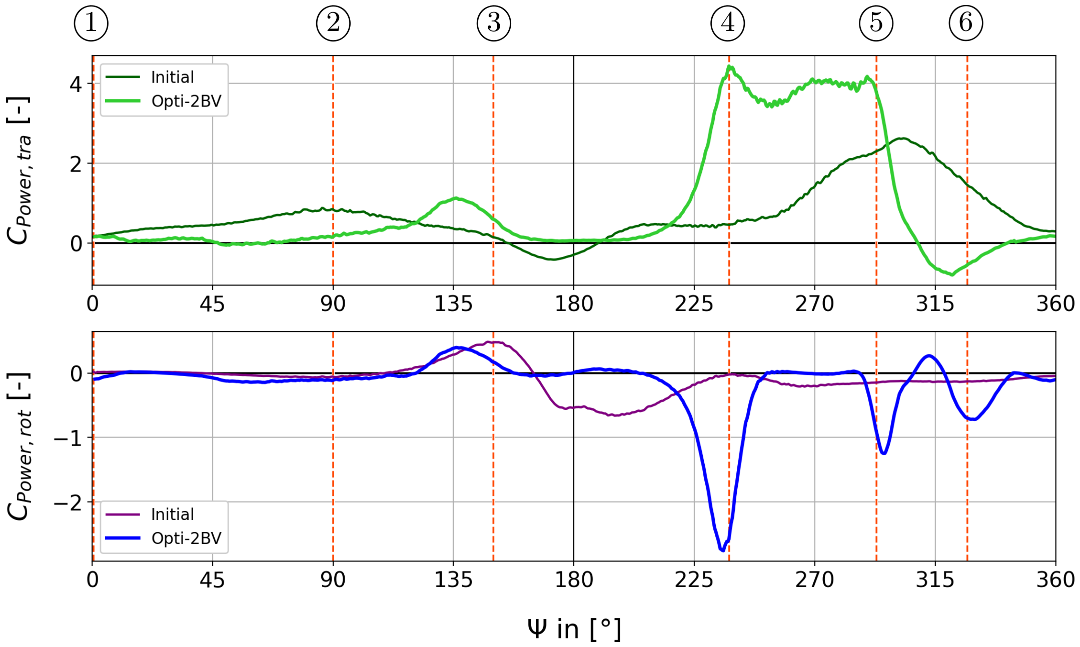


Figure 10.47: Thrust over azimuth angle Ψ.

Table 10.8: Mean values for Opti-2BV.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.703 N	0.217 W	0.555 W	0.125 W	0.680 W	0.320	104 °
Opit-2BV	1.65 N	0.895 W	0.836 W	0.204 W	1.04 W	0.754	132 °
Δ	+135 %	+312 %	+51 %	+63 %	+53 %	+136 %	



Pos.	Ψ
①	0°
②	90°
③	150°
④	238°
⑤	293°
⑥	327°

Figure 10.48: Translation and rotation power coefficients, Opti-2BV.

10.6 Three blades

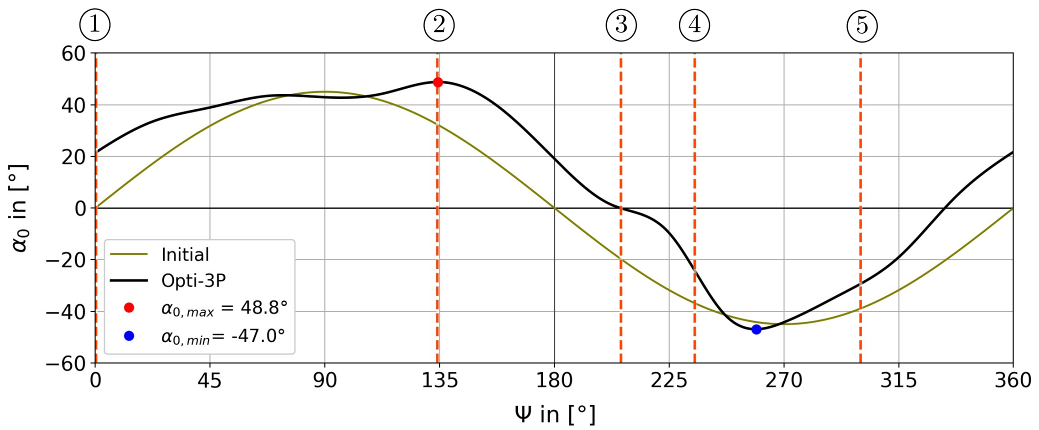
10.6.1 Opti-3P

Figure 10.49 shows the pitching path for the three-blade optimisation. A comparison of the pitching paths for 1P, 2P and 3P is shown in Figure 10.50. The maximum pitch is shifted to higher azimuth angles for the three-blade optimisation.

The optimised pitching path considerably improves the blade force distribution compared to the initial three-blade case, as shown in Figure 10.51. The lead angle avoids the adverse blade forces at the beginning of the rotation at ①. Like the previous optimisations 1P and 2P, the pitch plateau and the maximum pitch at ② generate small blade forces. But more importantly, the maximum pitch, in combination with its delayed decrease, prevents the generation of another adverse blade force. Figure 10.53 shows the pressure distribution for the initial case and the optimisation at ③. During the second half rotation, the fast pitch decrease produces a force peak at ④. The minimum pitch angle and the following linearly increase lead to a second force peak at ⑤. In contrast to the optimisation Opti-2P, the advancing vortex has no distinct influence on the following blade in the last quadrant.

As a result of the better force alignment, the figure of merit is 0.71 (+149%).

Table 10.9 lists characteristic values of the optimisation compared to its initial case.



Pos.	Ψ
①	0°
②	134°
③	206°
④	235°
⑤	299°

Figure 10.49: Pitching angle α_0 over azimuth angle Ψ.

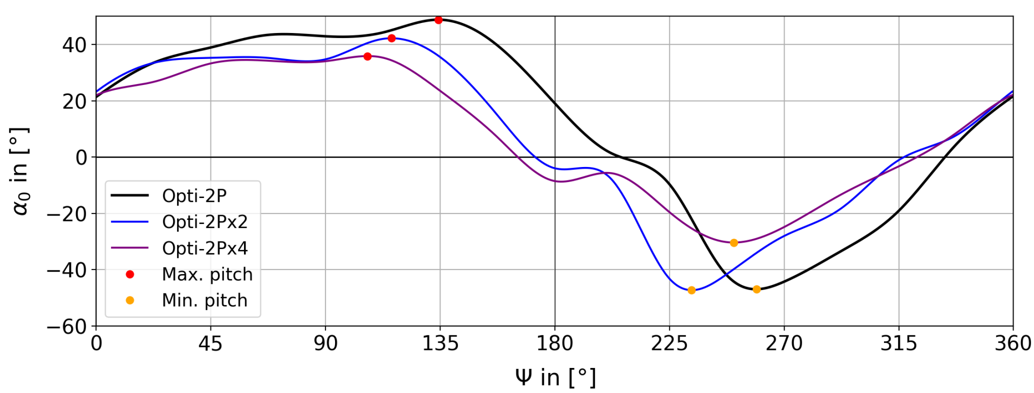


Figure 10.50: Comparison of the pitching path for different blade numbers.

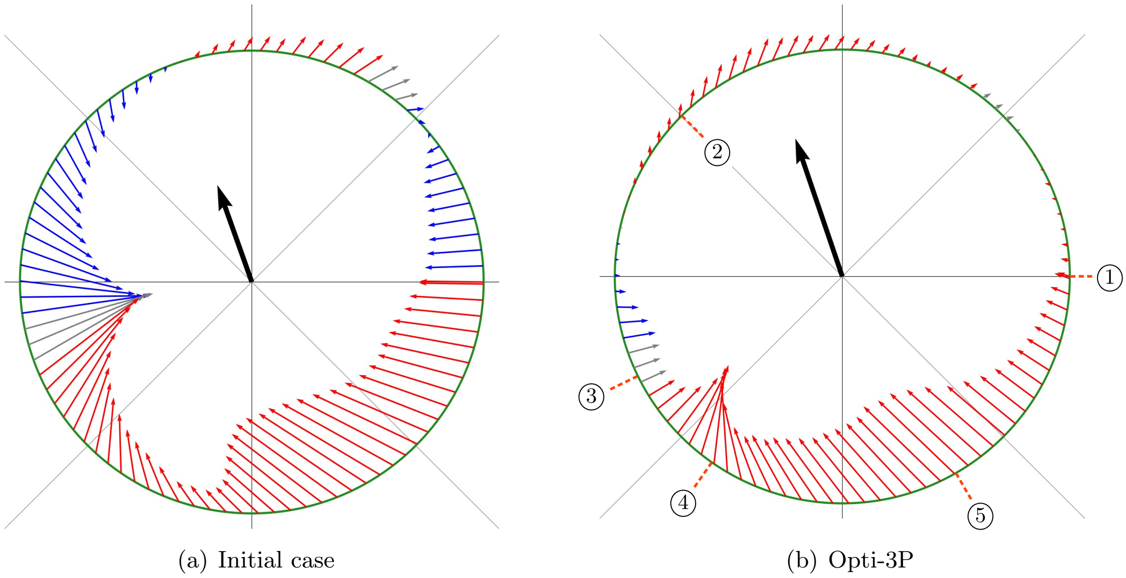


Figure 10.51: Resulting blade force over trajectory, black arrow represents the thrust.

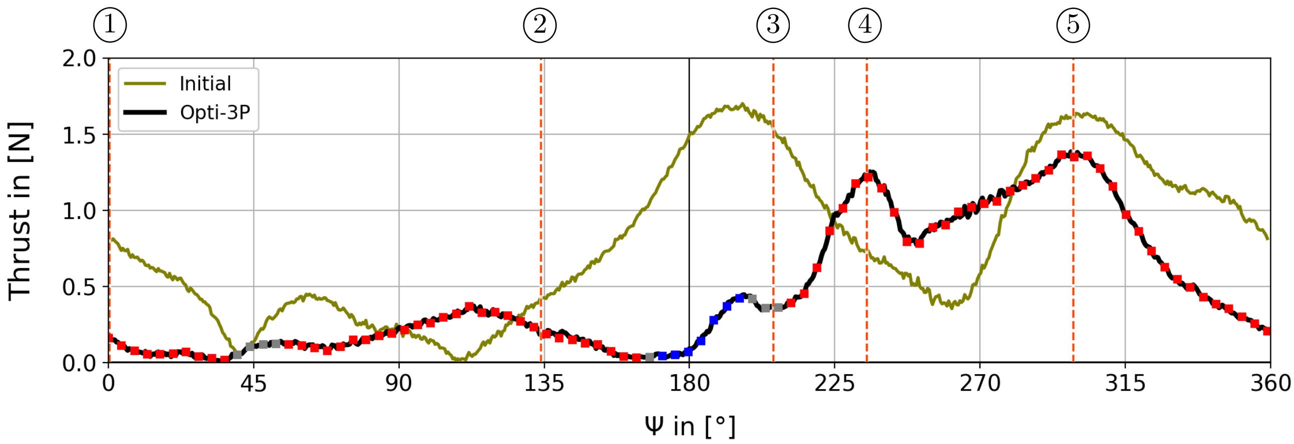
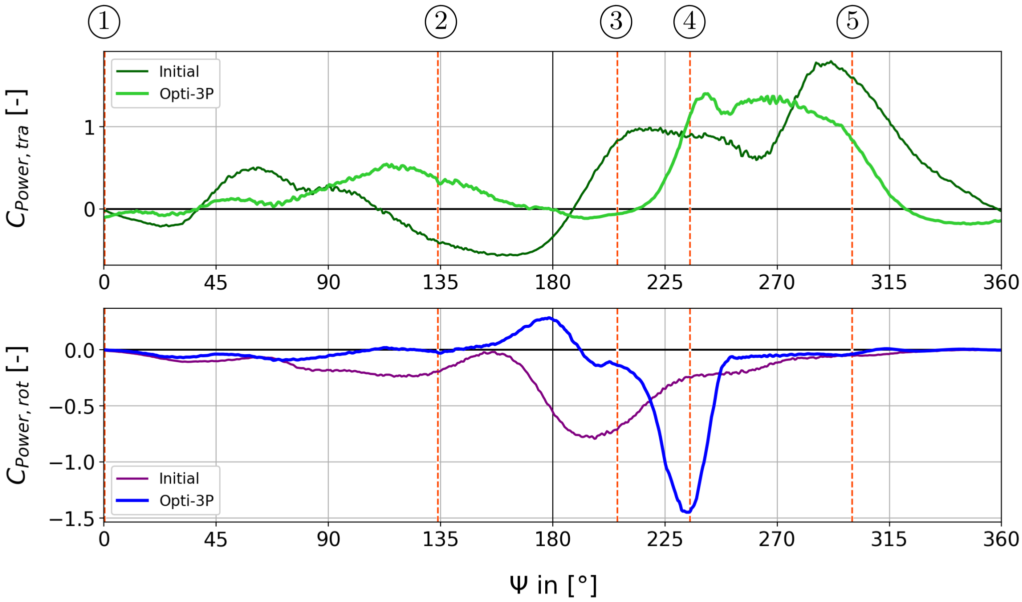


Figure 10.52: Thrust over azimuth angle Ψ.

Table 10.9: Mean values for Opti-3P.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.733 N	0.232 W	0.614 W	0.202 W	0.816 W	0.284	110 °
Opit-3PV	1.06 N	0.402 W	0.421 W	0.147 W	0.568 W	0.708	109 °
Δ	+44 %	+73 %	-31 %	-27 %	-30 %	+149 %	



Pos.	Ψ
①	0°
②	134°
③	206°
④	235°
⑤	299°

Figure 10.53: Translation and rotation power coefficients, Opti-3P.

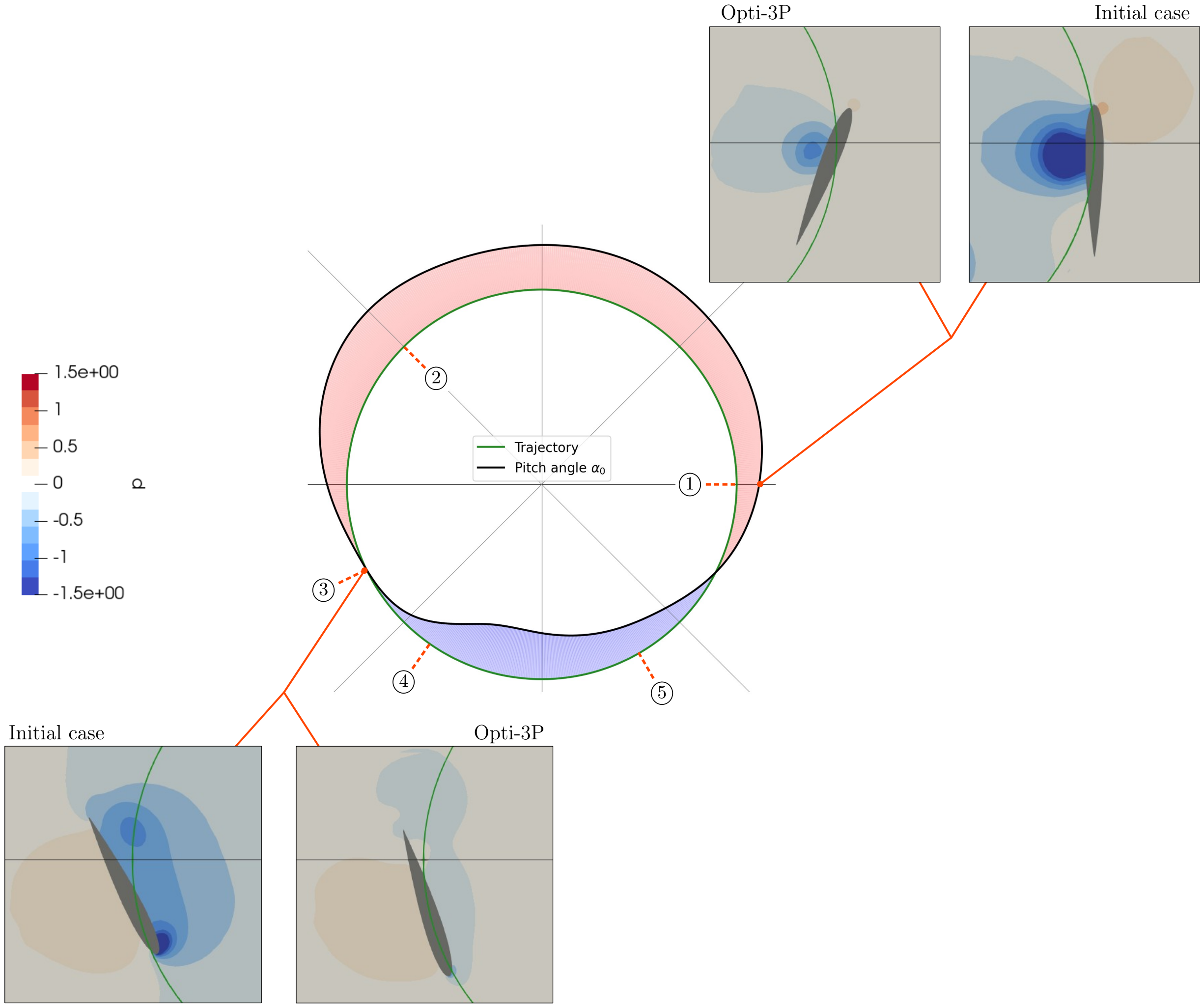


Figure 10.54: Pitching path over trajectory, Opti-3P.

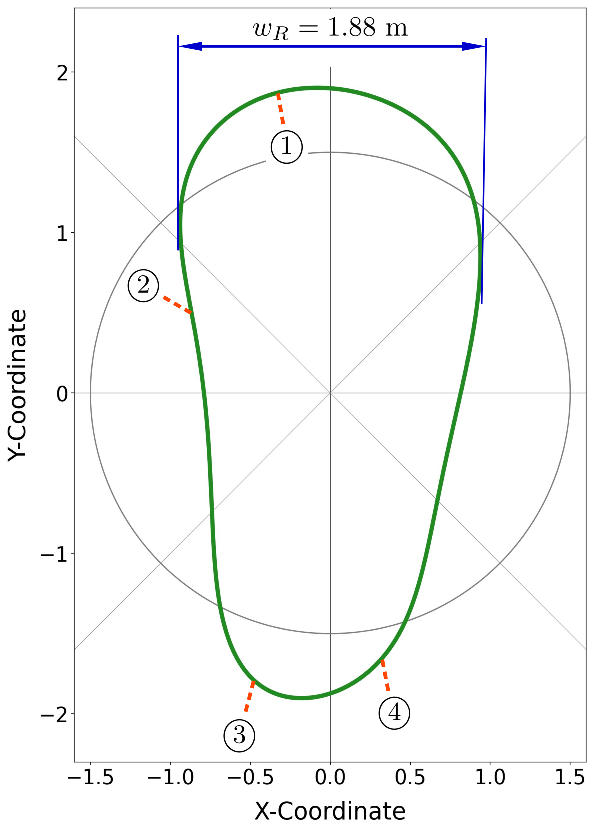
10.6.2 Opti-3TV

The trajectory of this optimisation is considerably stretched in the vertical direction, as shown in Figure 10.55, where the upper radius is slightly larger than the lower radius.

During the upper circular motion, beneficial blade forces are only produced between ① and ②. After an almost vertical decent, the blade performs a narrow circular movement. Combined with the minimum pitch angle, a force peak occurs at ③ and a second at ④. The blade forces during the movement from ④ to ① are mostly aligned horizontally.

As a result of the stretched trajectory, the downwash is inclined to the y axis by about 56°. As with the Opti-2BV, the same approach is chosen and a correct width is estimated, as shown in the velocity field in Figure 10.58. With a value of $w_{R,corr} = 2.6$ m, the figure of merit is $\mathrm{FoM}_{3TV,corr} = 0.60$ (+84%).

Table 10.10 lists characteristic values of the optimisation compared to its initial case.

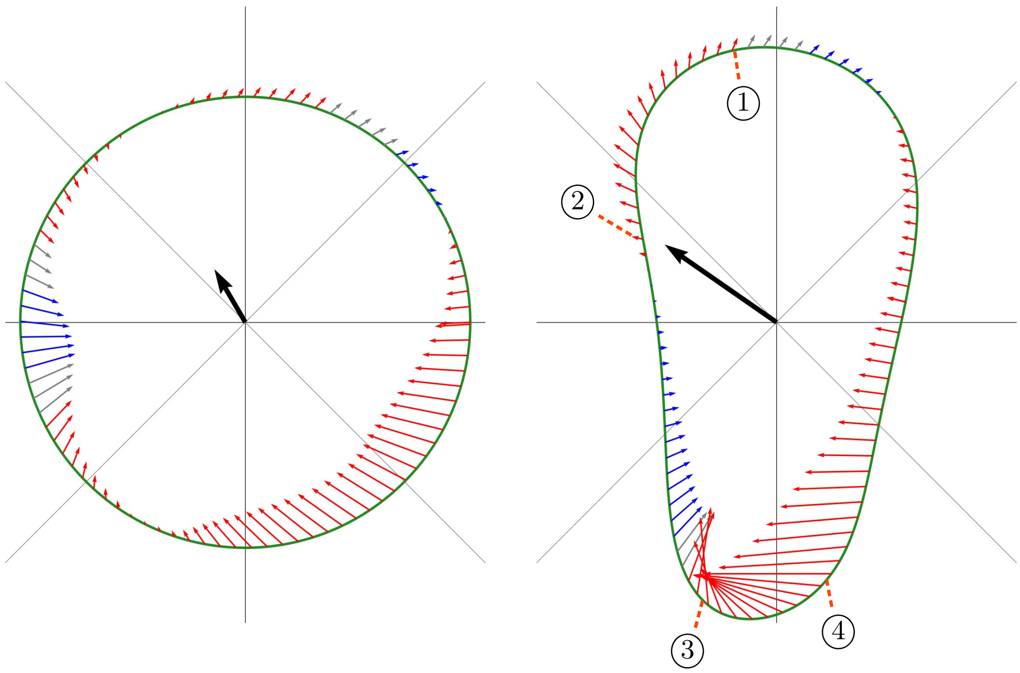


Pos.	Ψ
①	100°
②	150°
③	255°
④	281°

Figure 10.55: Trajectory of Opti-3TV, rotor width $w_R = 1.88$ m.

Table 10.10: Mean values for Opti-3TV, figure of merit with corrected width.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.751 N	0.240 W	0.568 W	0.166 W	0.734 W	0.327	121 °
Opit-3TV	1.69 N	1.02 W	0.993 W	0.455 W	1.45 W	0.602	146 °
Δ	+125 %	+327 %	+75 %	+174 %	+97 %	+84 %	



(a) Initial case (b) Opti-3TV

Figure 10.56: Resulting blade force over trajectory, black arrow represents the thrust.

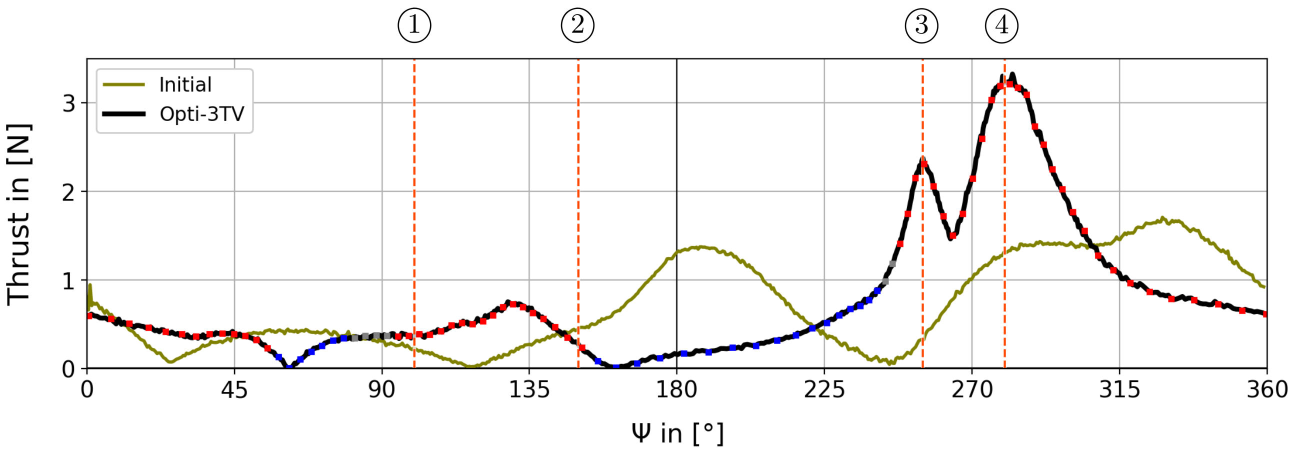


Figure 10.57: Thrust over azimuth angle Ψ.

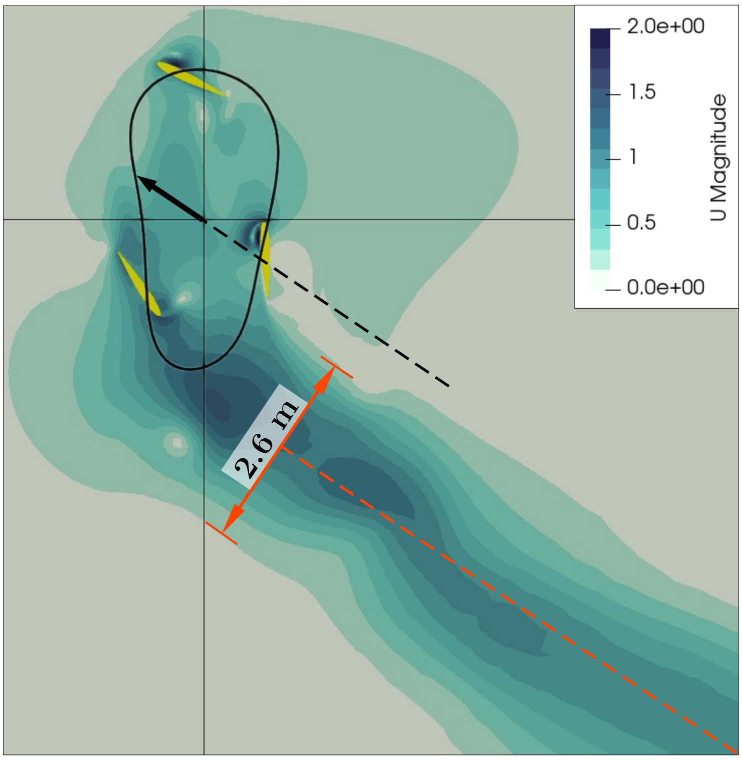


Figure 10.58: Magnitude of the velocity. The flow direction (orange line) is parallel to the global thrust (black line).

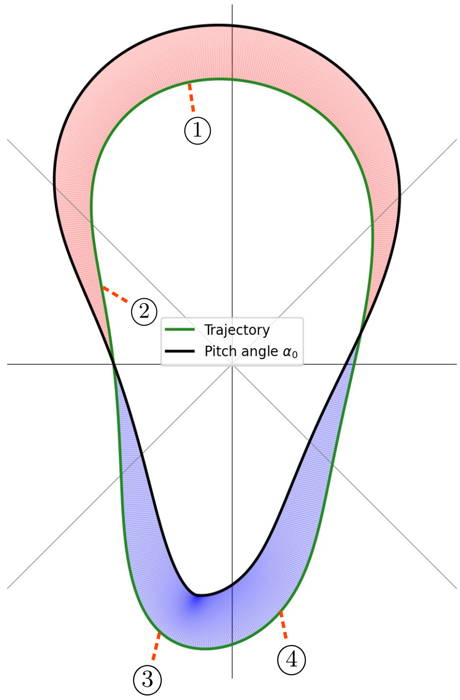
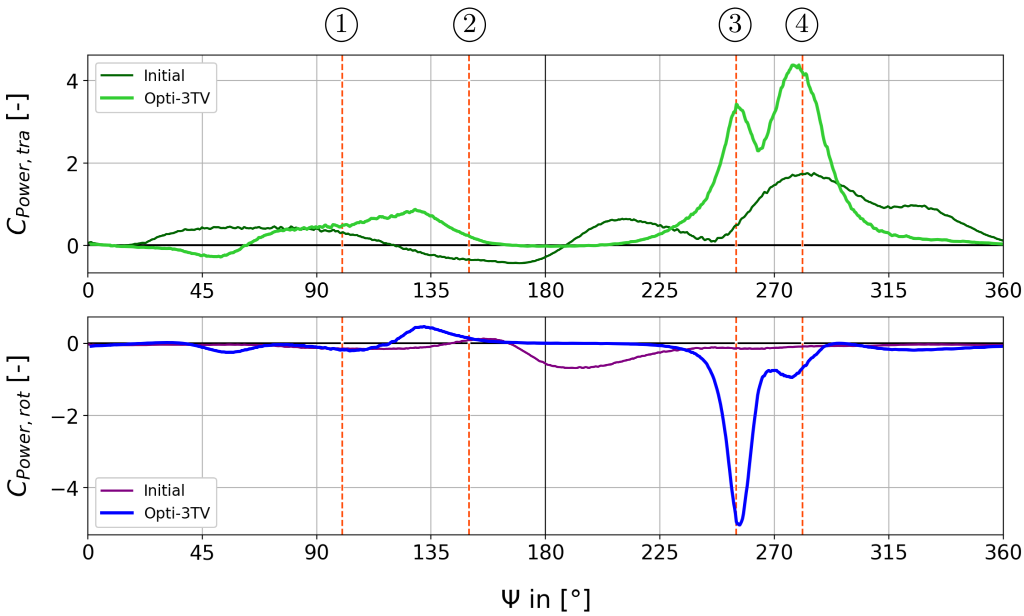


Figure 10.59: Pitching path over trajectory, Opti-3TV.



Pos.	Ψ
1	100°
2	150°
3	255°
4	281°

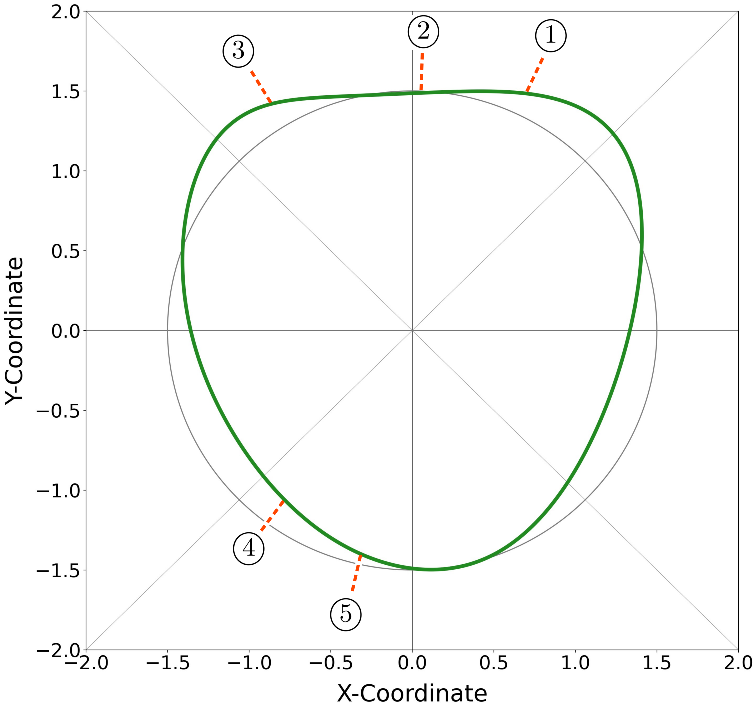
Figure 10.60: Translation and rotation power coefficients, Opti-3TV.

10.6.3 Opti-3BV

The trajectory has a triangle shape with an almost horizontal edge between ① and ③, see Figure 10.61. Within this straight movement, the pitch decreases about $\Delta\alpha_0 = 16°$ at ②, leading to a slight increase in the blade forces, Figure 10.63. The maximum pitch angle is reached at the second 'corner' of the trajectory, ③. However, this pitch also produces small blade forces. As shown in Figure 10.62, the pitch decreases rapidly after the maximum. In combination with the trajectory shape, the adverse blade forces are reduced compared to the initial case. A force peak occurs at ④ before the minimum pitch angle is reached at ⑤. The following trajectory with a large radius generates blade forces, which are aligned reasonably to the global thrust, and thus no adverse forces occur, see Figure 10.65.

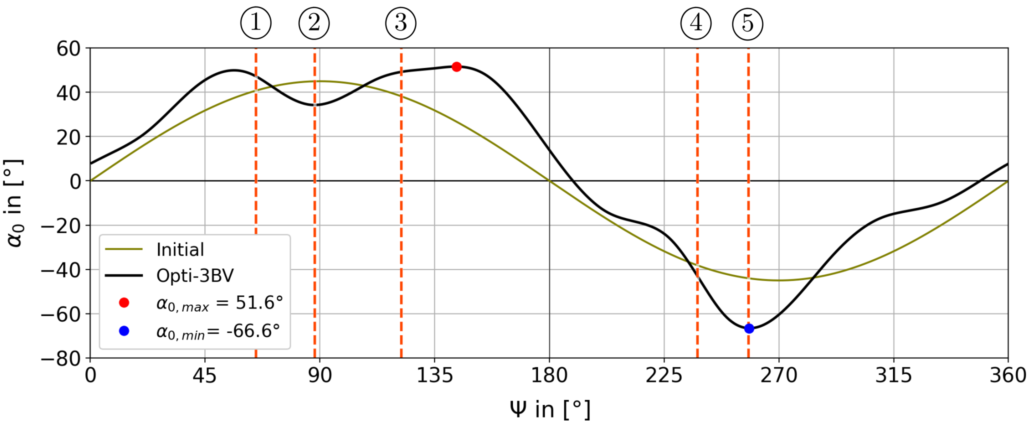
The main advantage of this optimisation is the reduction of adverse blade forces. And as a result, the figure of merit is increased to $\mathrm{FoM}_{3BV} = 0.743$ (+127%).

Table 10.11 lists characteristic values of the optimisation compared to its initial case.



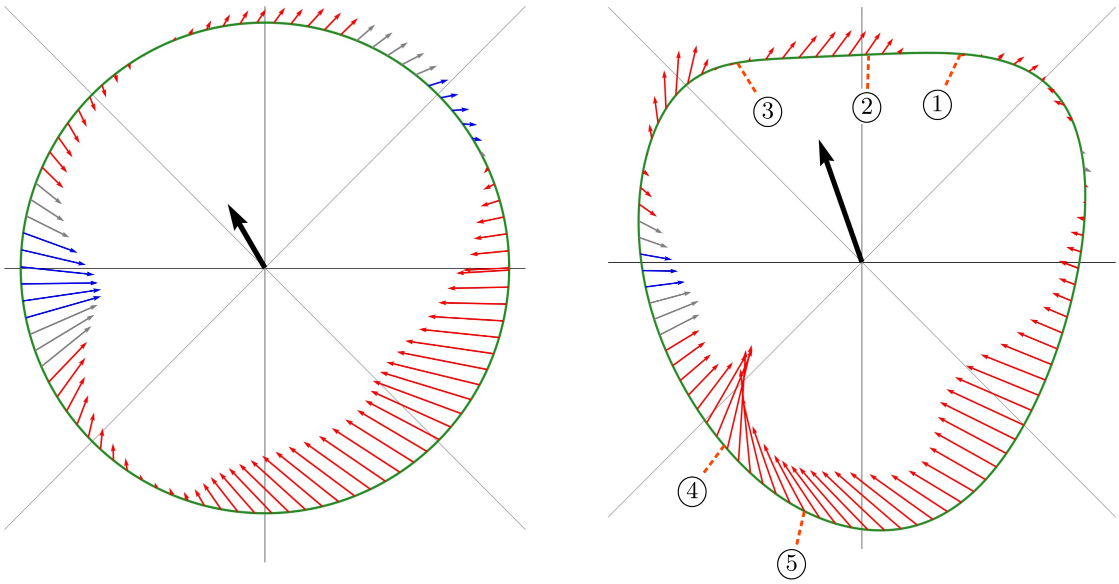
Pos.	Ψ
①	65°
②	88°
③	122°
④	233°
⑤	258°

Figure 10.61: Trajectory of Opti-3BV; rotor width $w_R = 2.82$ m.



Pos.	Ψ
①	65°
②	88°
③	122°
④	238°
⑤	258°

Figure 10.62: Pitching angle α_0 over azimuth angle Ψ, Opti-3BV.



(a) Initial case (b) Opti-3BV

Figure 10.63: Resulting blade force over trajectory, black arrow represents the thrust.

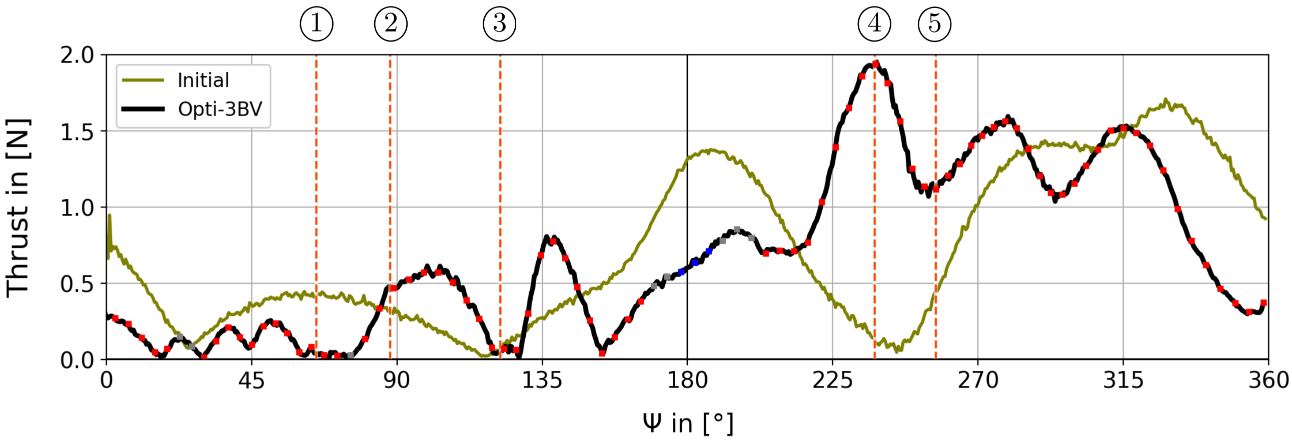


Figure 10.64: Thrust over azimuth angle Ψ.

Table 10.11: Mean values for Opti-3BV.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.751 N	0.240 W	0.568 W	0.166 W	0.734 W	0.327	121 °
Opit-3BV	1.72 N	0.856 W	0.895 W	0.258 W	1.15 W	0.743	112 °
Δ	+129 %	+257 %	+58 %	+55 %	+57 %	+127 %	

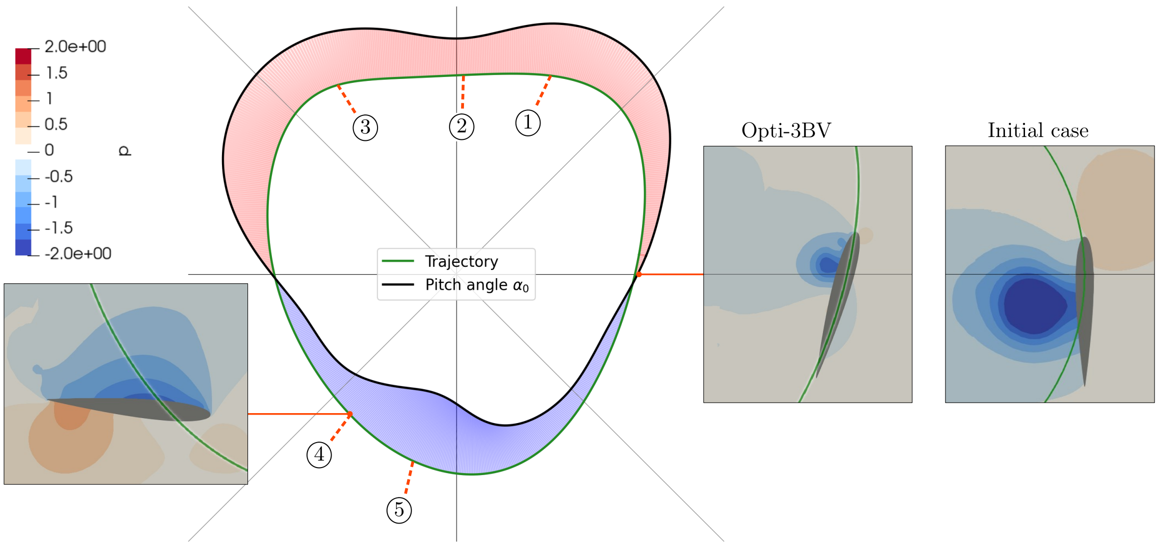
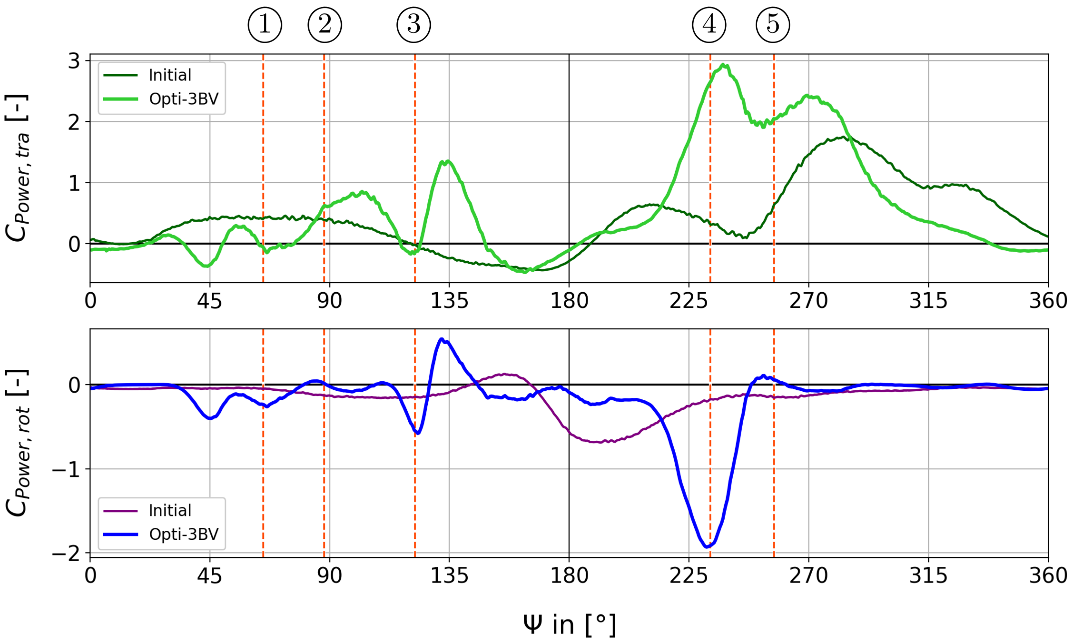


Figure 10.65: Pitching path over trajectory, Opti-3BV.



Pos.	Ψ
1	65°
2	88°
3	122°
4	238°
5	257°

Figure 10.66: Translation and rotation power coefficients, Opti-3BV.

10.7 Four blades

10.7.1 Opti-4P

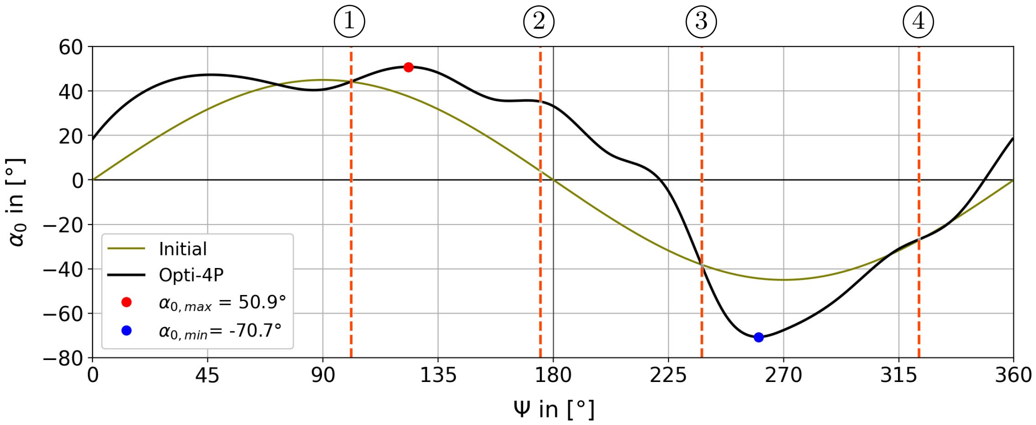
Despite the interaction between the four blades, the pitching path is again similar to the previous one as shown in Figure 10.68.

The pitching path starts with a lead angle of about $\alpha_0 = 20°$ and increases rapidly up to $\alpha_0 = 47°$. However, until ① no blade forces are generated. The movement between ① and ② generates small blade forces, although the maximum pitch occurs. During the steep decrease of the pitch towards the minimum, a considerable force peak occurs at ③. After that, the forces reduce and reach a second maximum at ④.

During the initial case, the blade motion in the last quadrant leads to a shedding vortex. The following blade hits this vortex and, in turn, induces a vortex, which is also shed, see Figure 10.71. The optimisation avoids the vortex at the advancing blade by rapidly increasing the pitch from ③ to the end of the rotation. Although the required power remains similar between the two cases, the pressure distribution is improved to generate more and better-aligned blade forces towards the end.

The figure of merit is increased to $\text{FoM}_{4P} = 0.76$ (+241%), which is the absolute maximum over all optimisation runs.

Table 10.12 lists characteristic values of the optimisation compared to its initial case.



Pos.	Ψ
①	100°
②	175°
③	238°
④	323°

Figure 10.67: Pitching angle α_0 over azimuth angle Ψ.

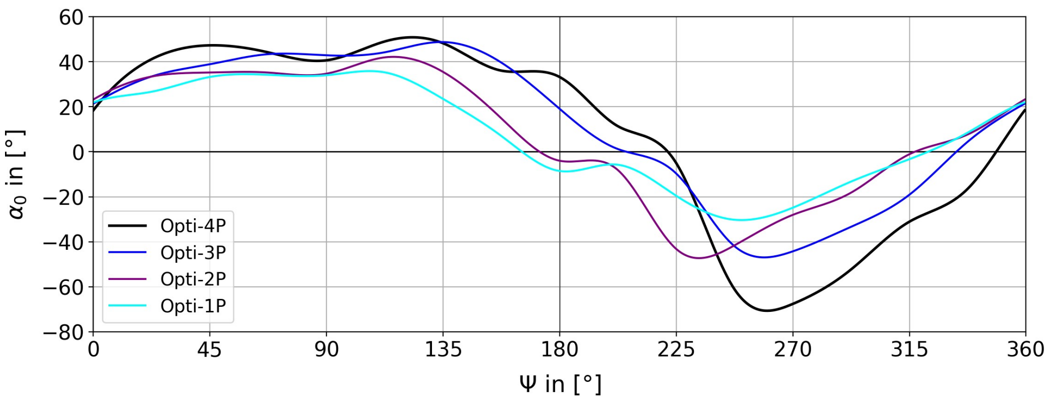
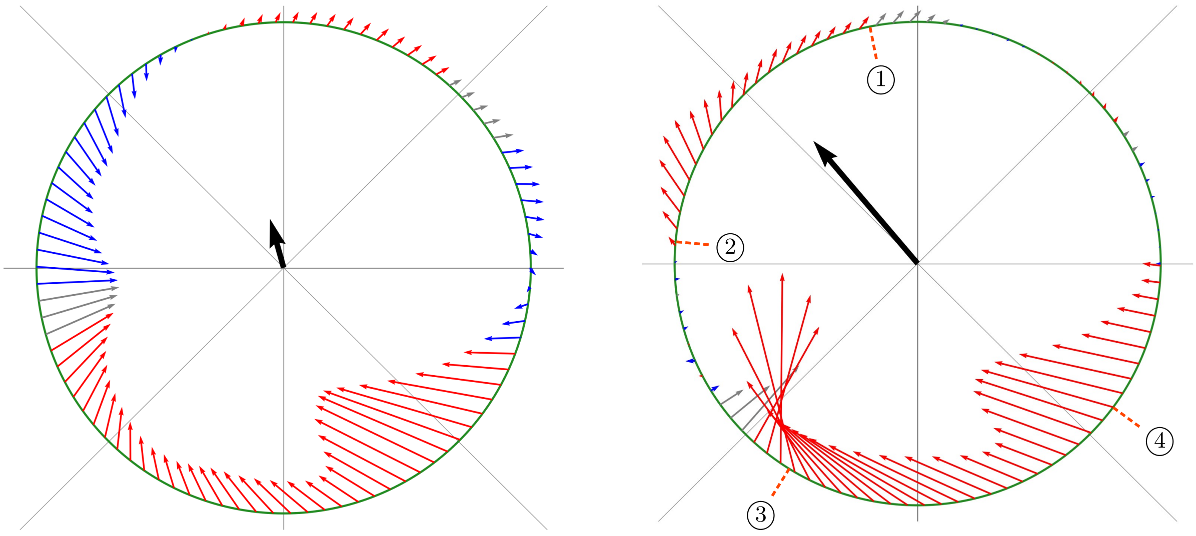


Figure 10.68: Comparison of the pitching path for pitch optimisations with different blade numbers.



(a) Initial case (b) Opti-4P

Figure 10.69: Resulting blade force over trajectory, black arrow represents the thrust.

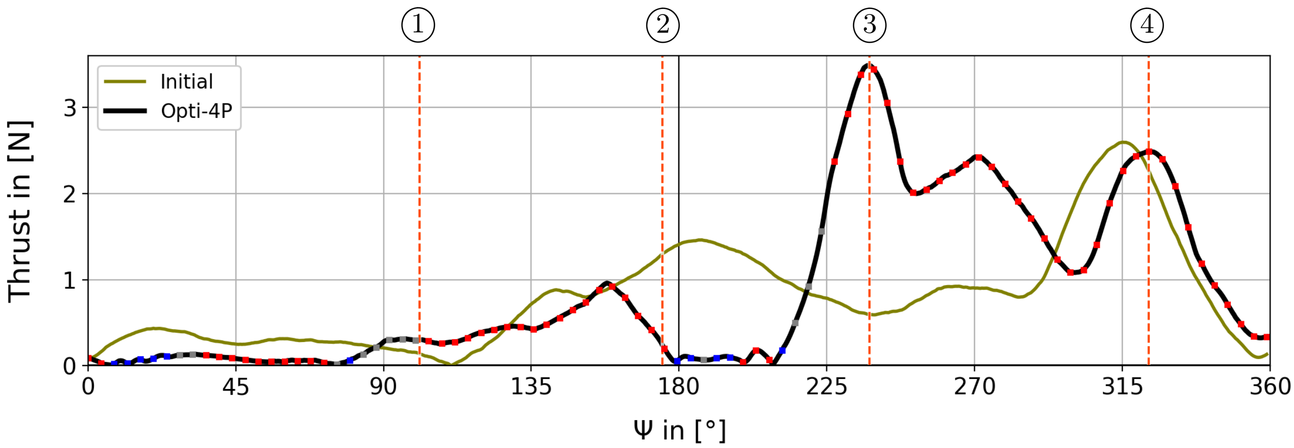


Figure 10.70: Thrust over azimuth angle Ψ.

Table 10.12: Mean values for Opti-4P.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.857 N	0.293 W	1.02 W	0.300 W	1.32 W	0.222	107 °
Opit-4PV	2.78 N	1.71 W	1.61 W	0.642 W	2.25 W	0.758	131 °
Δ	+224 %	+483 %	+59 %	+114 %	+71 %	+241 %	

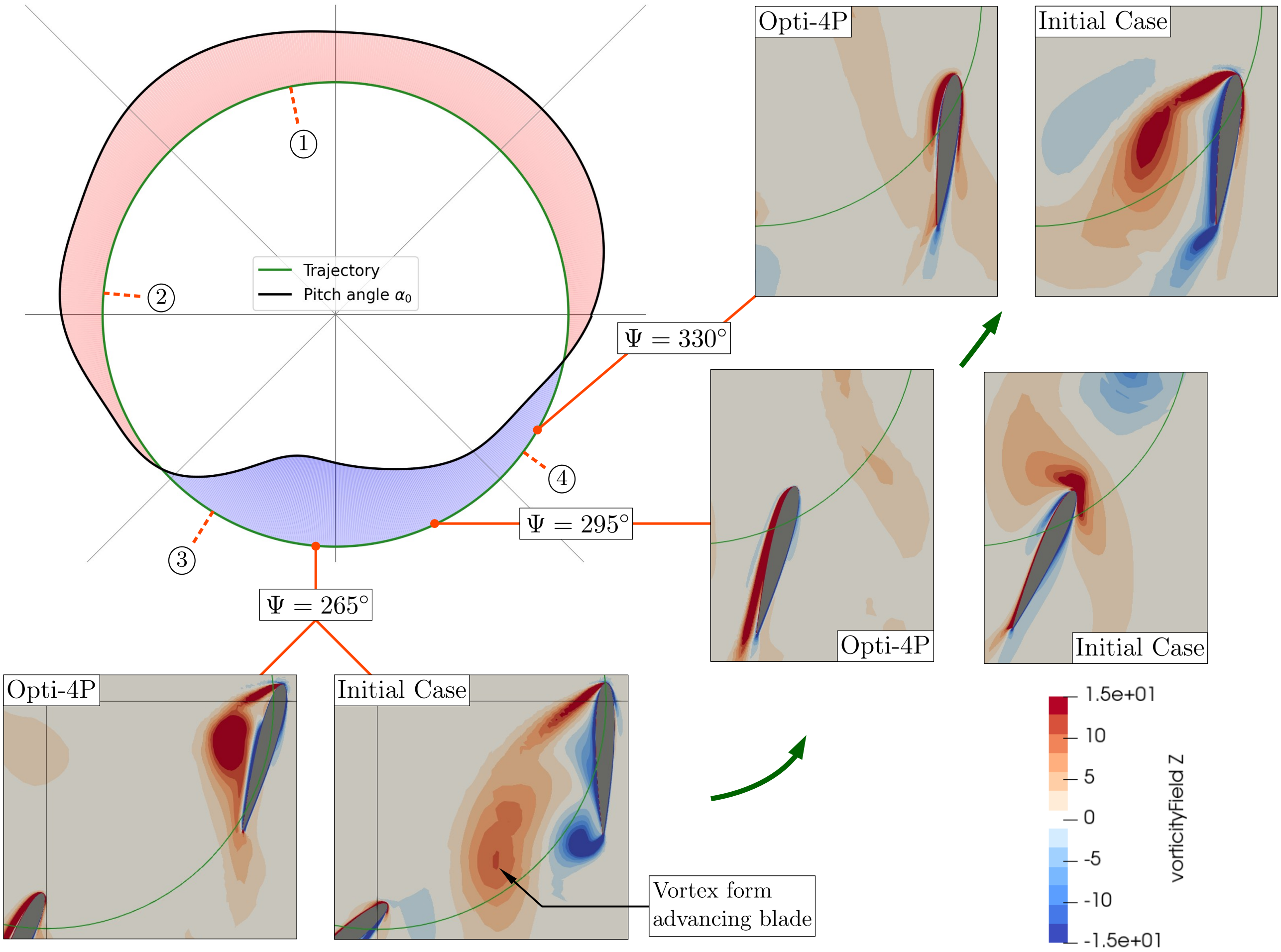
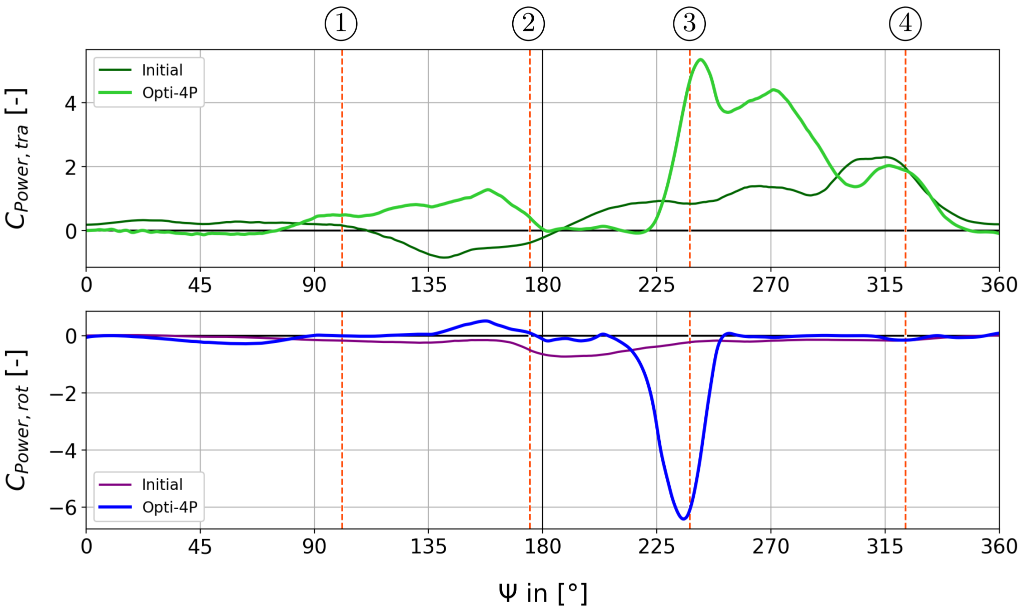


Figure 10.71: Pitching path over trajectory, Opti-4P.



Pos.	Ψ
1	100°
2	175°
3	238°
4	323°

Figure 10.72: Translation and rotation power coefficients, Opti-4P.

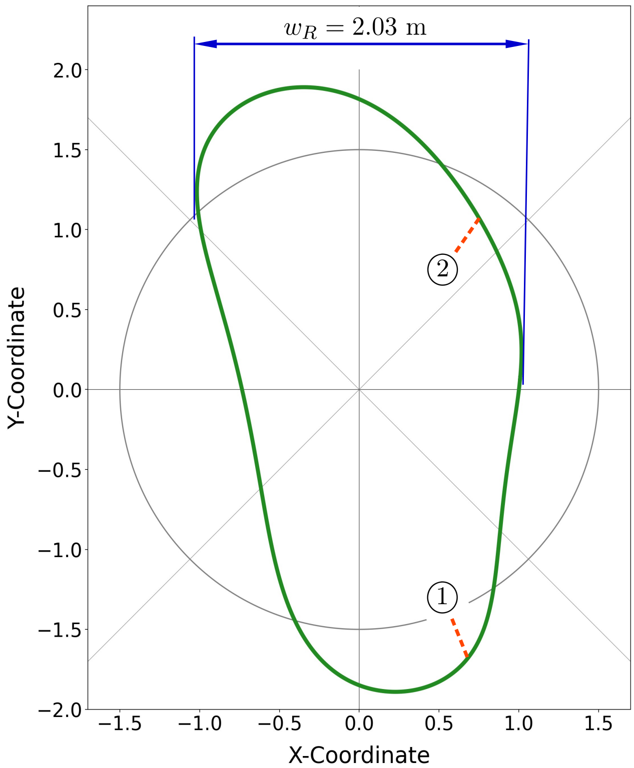
10.7.2 Opti-4TV

This trajectory is quite similar to the three-blade optimisation, which is also highly stretched in the vertical direction, see Figure 10.73.

Only one section produces beneficial blade forces, starting with a peak force at 1. After that, the forces decrease and remain constant for a while and vanish at 2. The blade performs a nearly vertical movement in this section, which leads to a global thrust with a high inclination of $\beta = 69°$. As mentioned at the Opti-2BV, he downwash is estimated based on the velocity field, see Figure 10.76. With a value of $w_{R,corr} = 2.2\text{m}$, the corrected figure of merit $FoM_{4TV,corr} = 0.641$ (+177%).

Although this optimisation run has not converged, the result is quite good.

Table 10.13 lists characteristic values of the optimisation compared to its initial case.

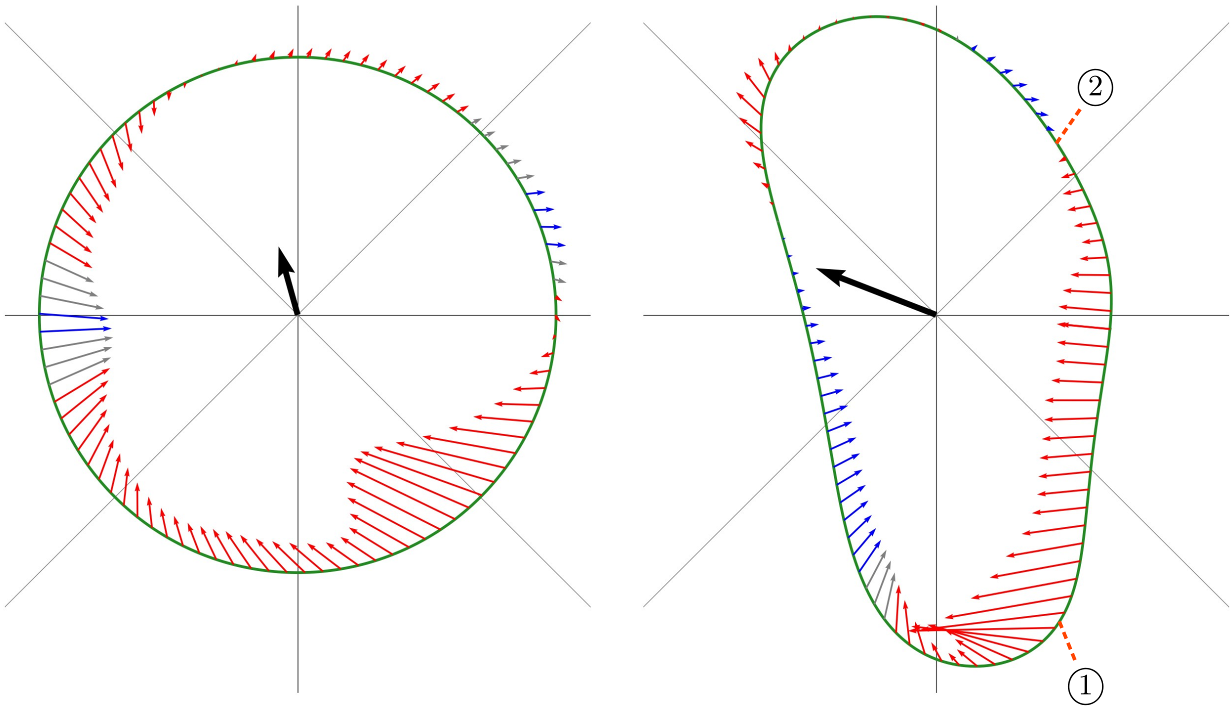


Pos.	Ψ
1	292 °
2	55 °

Figure 10.73: Trajectory of Opti-4TV, rotor width $w_R = 2.03$ m.

Table 10.13: Mean values for Opti-4TV, figure of merit with corrected width.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.928 N	0.330 W	1.11 W	0.317 W	1.43 W	0.230	116 °
Opit-4TV	1.58 N	0.890 W	0.872 W	0.463 W	1.34 W	0.641	159 °
Δ	+70 %	+170 %	-22 %	+46 %	-7 %	+177 %	



(a) Initial case (b) Opti-4TV

Figure 10.74: Resulting blade force over trajectory, black arrow represents the thrust.

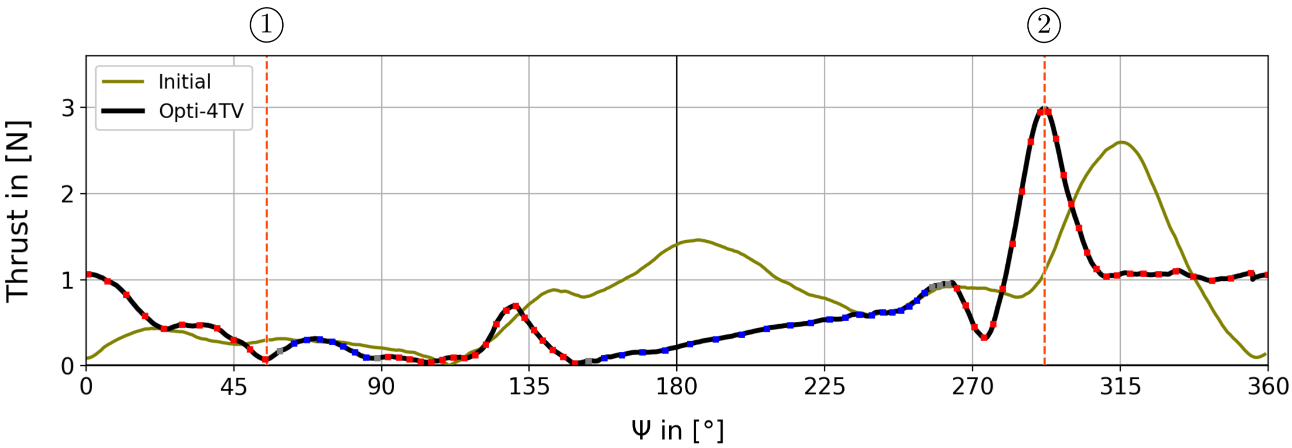


Figure 10.75: Thrust over azimuth angle Ψ.

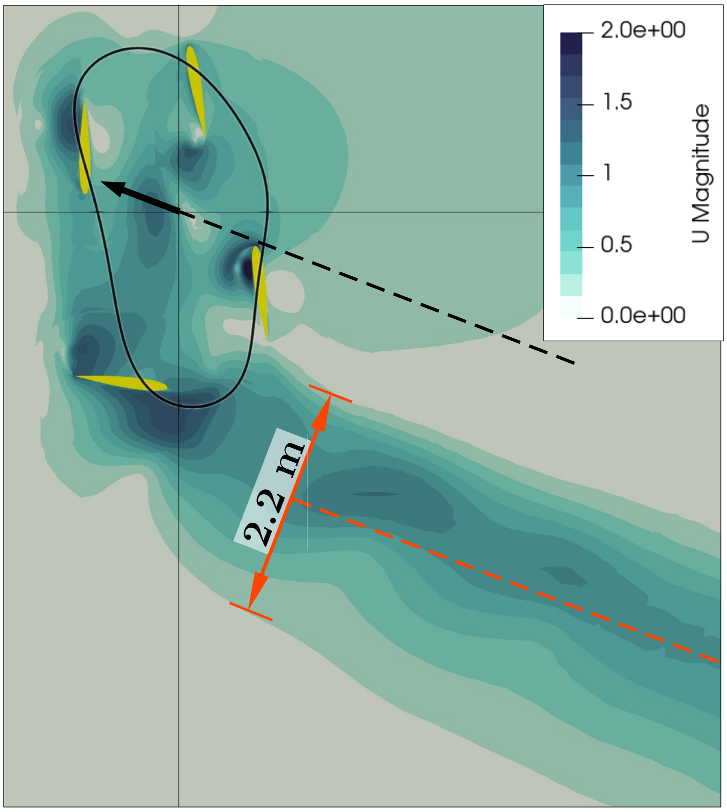


Figure 10.76: Magnitude of the velocity. The flow direction (orange line) is parallel to the global thrust (black line).

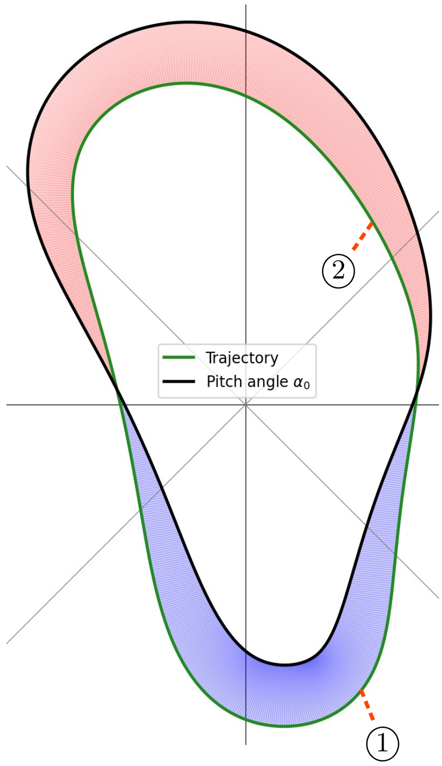
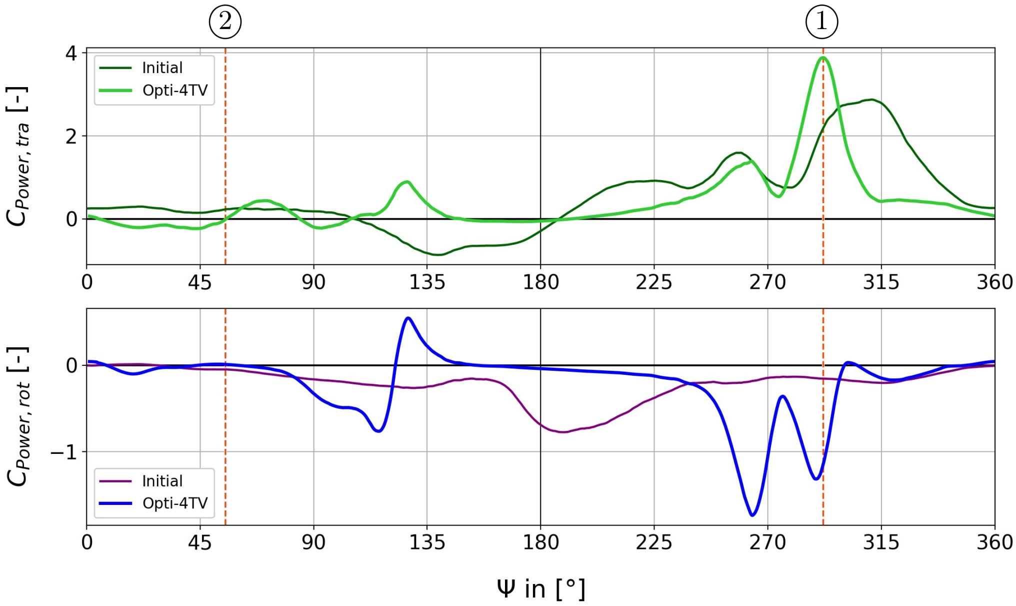


Figure 10.77: Pitching path over trajectory, Opti-4TV.



Pos.	Ψ
①	292 °
②	55 °

Figure 10.78: Translation and rotation power coefficients, Opti-4TV.

10.7.3 Opti-4BV

This trajectory is shaped like a triangle, see Figure 10.79. The trajectory can be divided into four sections, with conducive and adverse blade forces alternating, as shown in Figure 10.81.

①→② The section begins with a fast pitch increase, remaining nearly constant. With the stretched trajectory, the blade generates adverse forces.

②→③ In this section, the pitch reaches its maximum and decreases linearly. The blade performs an almost circular movement and generates conducive forces.

③→④ The linear pitch decrease continues in this section, leading to adverse force.

④→① The last section begins with a sharp pitch drop to its minimum. Shortly after the minimum, the pitch increases rapidly. During this section, the blade performs a short-lasting circular movement followed by an almost vertical ascent. As a result, a vast force peak is generated, and a second occurs just before the sections' end.

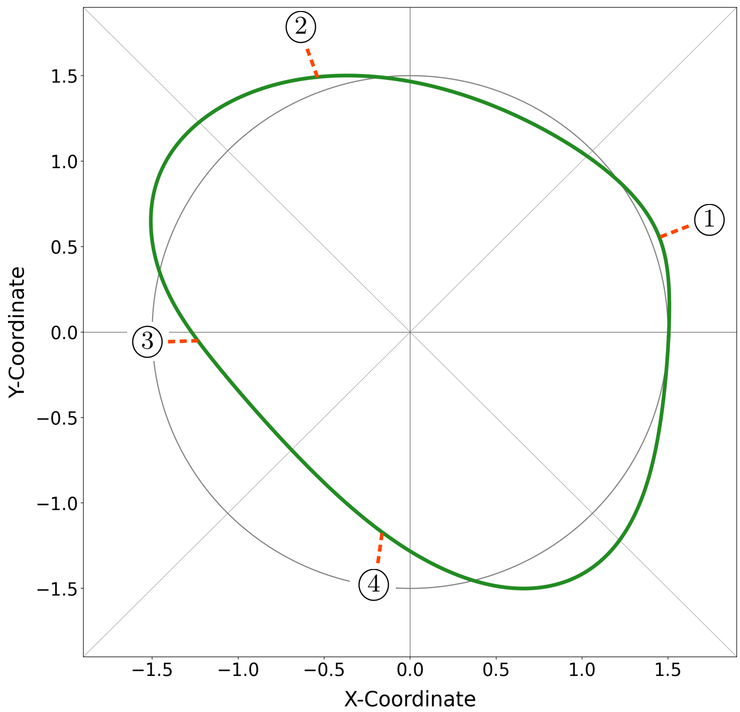
However, the direction of the first peak has a considerable inclination of about 67° compared to the global thrust, and thus the peak is less effective.

Therefore the figure of merit $FoM_{4BV} = 0.34$ is moderat, which may is a result of the poor convergence, see Figrue 10.8(c). In contrast to previous optimisations the width of this downwash is smaller than the width of the trajectory as shown in Figure 10.83. With the corrected value the figure of merit is now $FoM_{4BVcorr} = 0.4$ (+74%).

Table 10.14 lists characteristic values of the optimisation compared to its initial case.

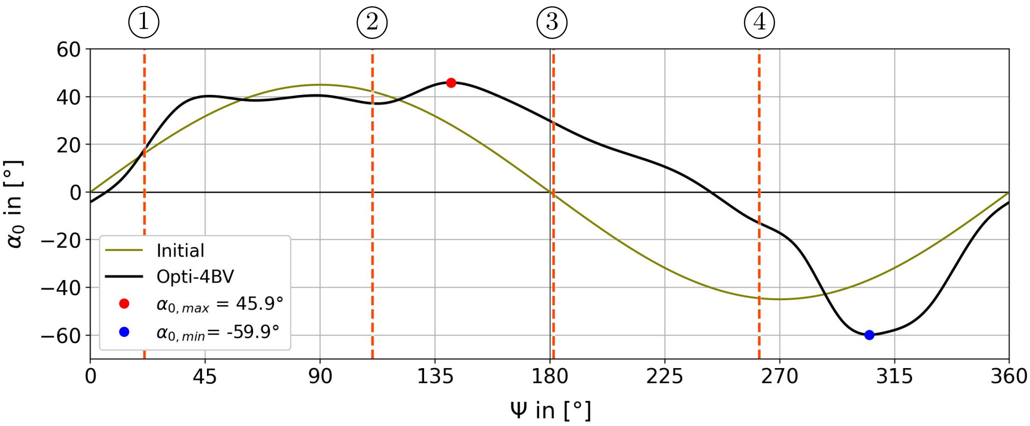
Table 10.14: Mean values for Opti-4BV.

Type	Thrust	P_{id}	P_{tra}	P_{rot}	P_{real}	FoM	β
Initial	0.928 N	0.330 W	1.11 W	0.317 W	1.43 W	0.230	116 °
Opit-4BV	1.723 N	0.834 W	1.81 W	0.637 W	2.45 W	0.400	139 °
Δ	+86 %	+153 %	+63 %	+101 %	+71 %	+74 %	



Pos.	Ψ
1	21°
2	110°
3	182°
4	262°

Figure 10.79: Trajectory of Opti-4BV; rotor width $w_R = 3.02$ m.



Pos.	Ψ
1	21°
2	110°
3	182°
4	262°

Figure 10.80: Pitching angle α_0 over azimuth angle Ψ, Opti-4BV.

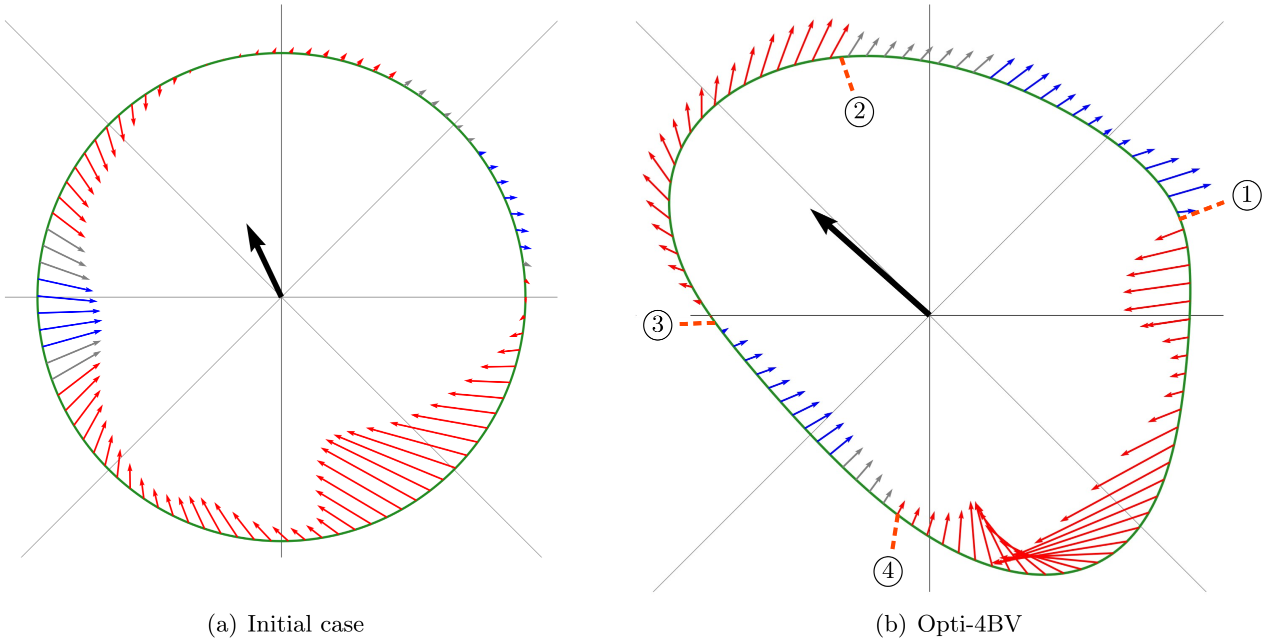


Figure 10.81: Resulting blade force over trajectory, black arrow represents the thrust.

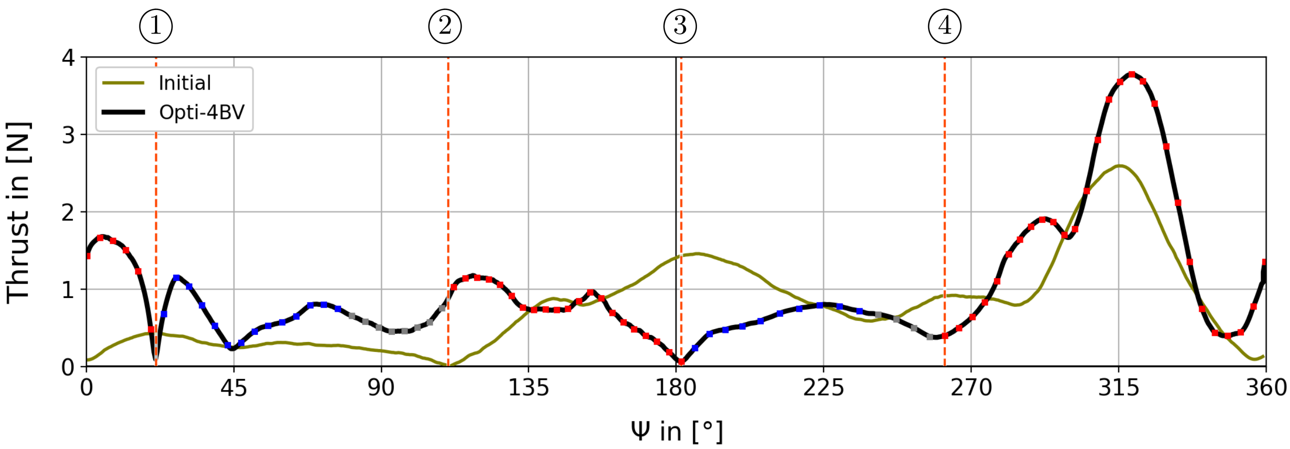


Figure 10.82: Thrust over azimuth angle Ψ.

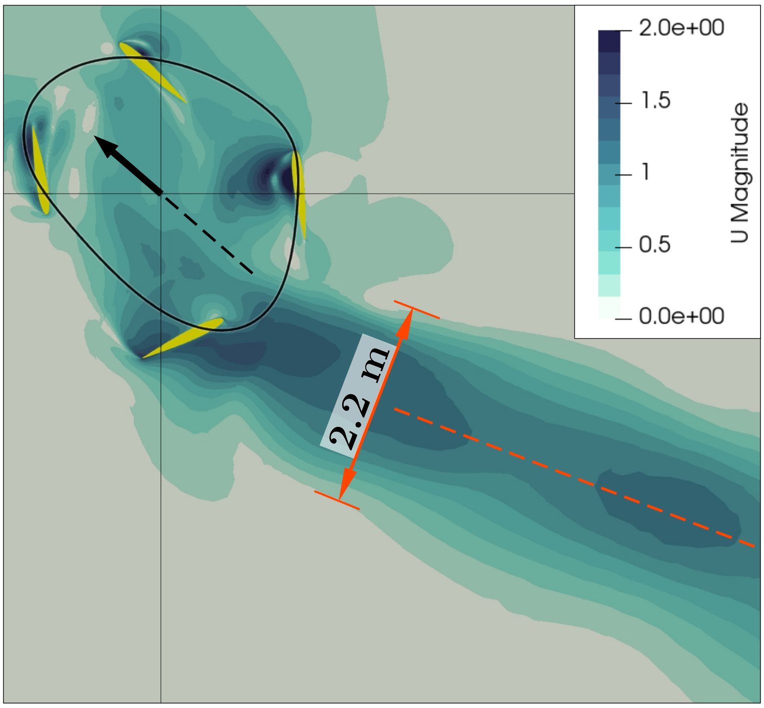


Figure 10.83: Magnitude of the velocity. Direction of flow (orange line) **not** parallel to the global thrust (black line).

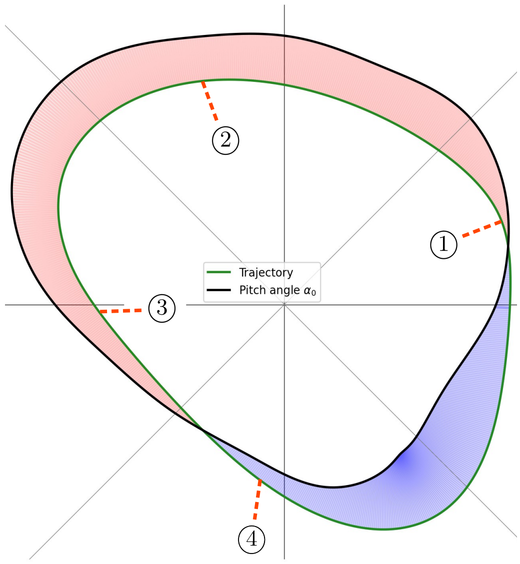
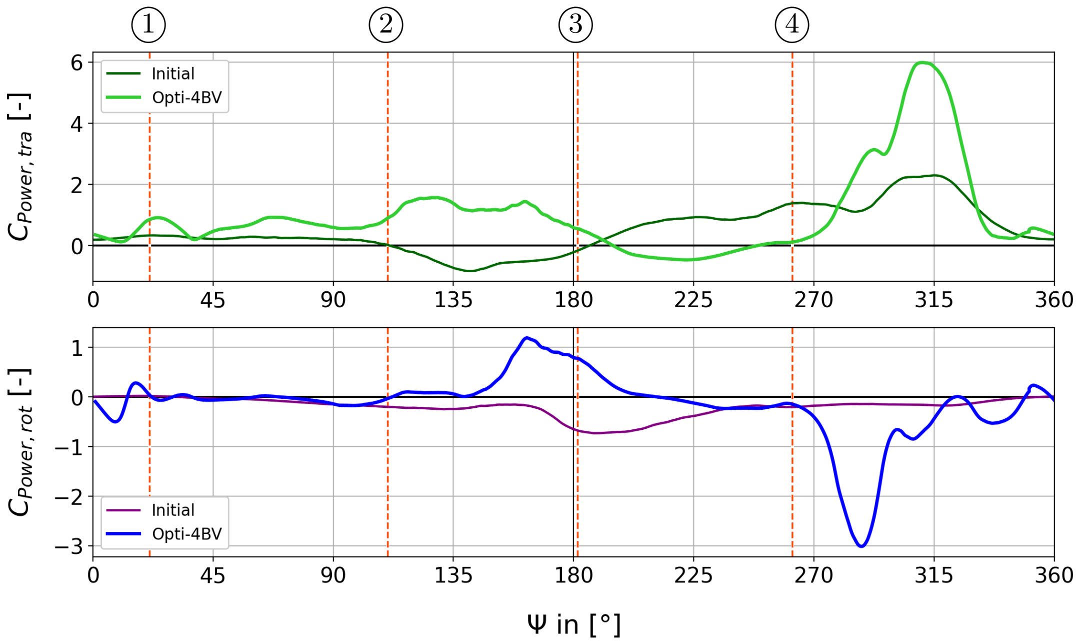


Figure 10.84: Pitching path over trajectory of Opti-4BV.



Pos.	Ψ
1	21°
2	110°
3	182°
4	262°

Figure 10.85: Translation and rotation power coefficients, Opti-4BV.

10.8 Constant angular velocity

As mentioned in Chapter 9, the optimisation results with a constant angular velocity are invalid due to a code error. Nonetheless, the resulting trajectories are shown the following figures.

The optimiser again tries to induce a certain force peak by forming a narrow trajectory. As in previous cases, the blade generates small forces during the upper movement.

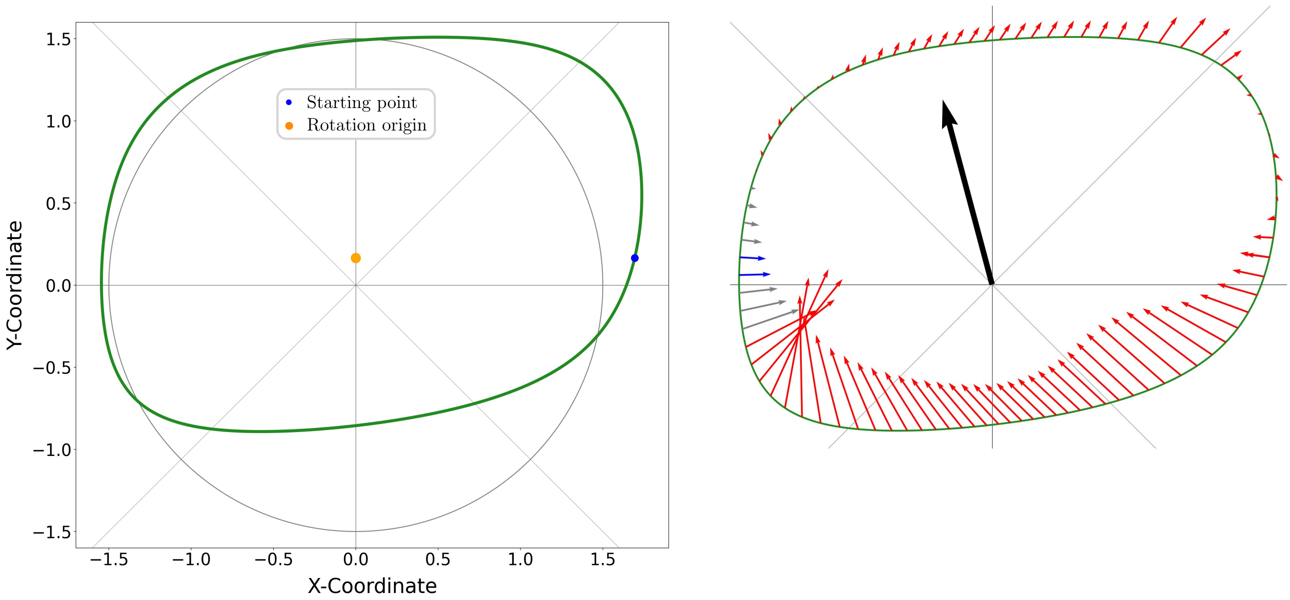


Figure 10.86: Trajectory and resulting forces for Opti-1TO.

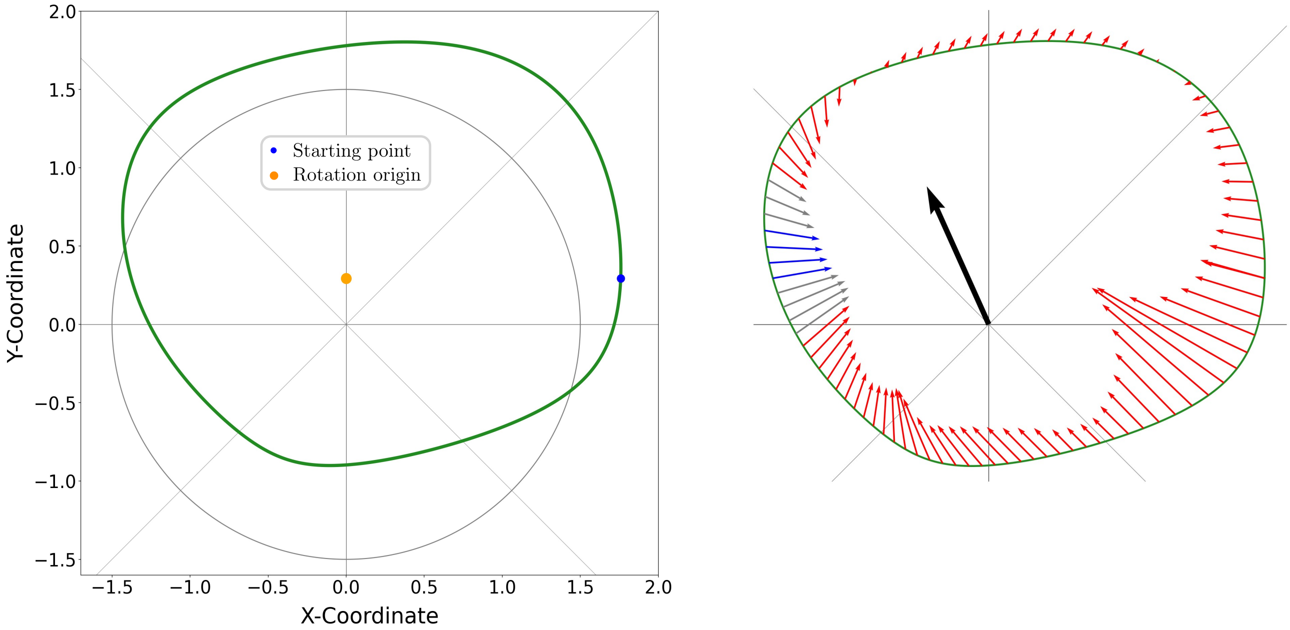
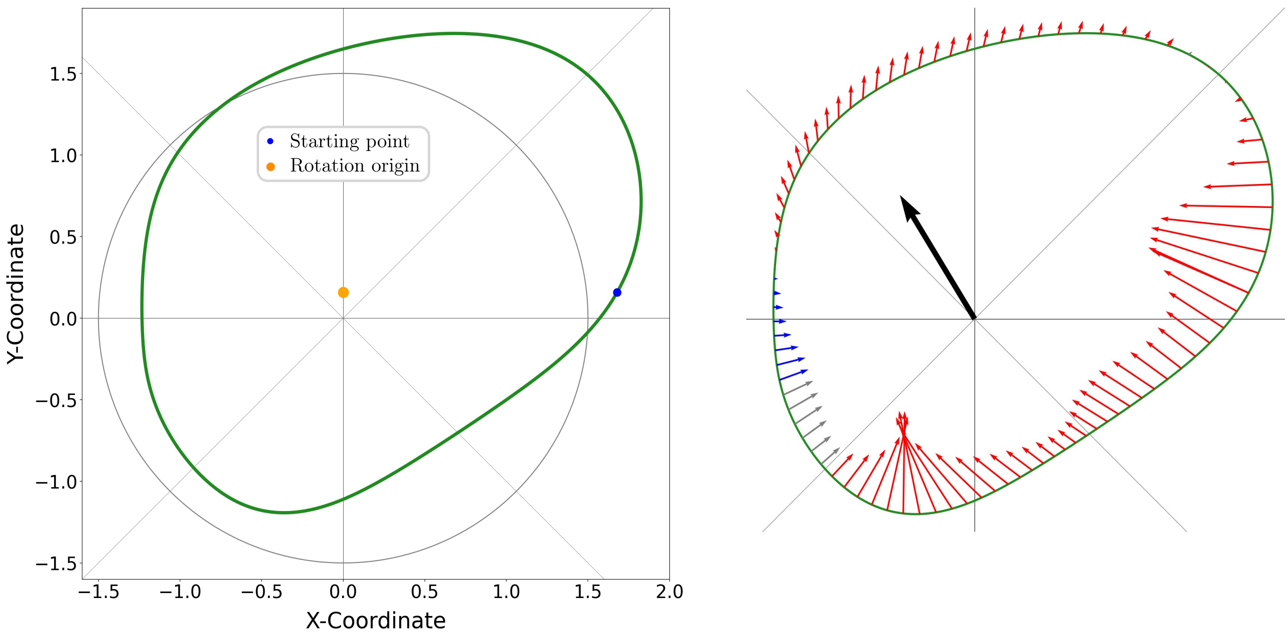
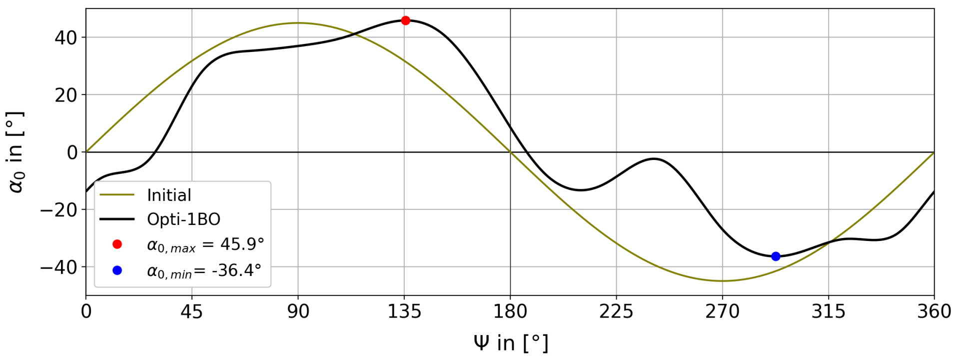


Figure 10.87: Trajectory and resulting forces for Opti-2TO.



(a) Trajectory and resulting forces.



(b) Pitching angle α_0 over azimuth angle Ψ.

Figure 10.88: Results for Opti-1BO.

11 Summary

The subject of this master thesis is to develop a procedure to optimise the pitching path and the trajectory for the blades of cycloidal rotors on the basis of CFD calculations. These 2D unsteady, incompressible URANS analyses are carried out with OpenFOAM. The number of investigated blades ranges from one to four Thereofore, the overset method is used to interpolate between the finite volume meshes. The overset method causes fluctuation of the pressure field and discontinuities at the interpolating cells, which leads to noisy force, which can be accepted. Although the y^+ parameter exceeds one over a small range, the coarse blade meshes enable a short execution time, which is essential due to the high number of CFD calculations.

For the implementation of arbitrary pitching and trajectory, B-splines are used. A four-order spline with 16 control vertices is used for the pitching path, which enables a suitable adjustment. For the trajectory, a fith-order spline is used to ensure continuity of the motion. Eight control vertices form the shape of the trajectory. Another spline is invented, representing the length over the control variable. With this spline and the Newton-Raphson method, the control variable is estimated to achieve a constant velocity along the trajectory.

Two motion classes are written in `C++`, which enables an arbitrary mesh motion in OpenFOAM. The `bSplinePitching.C` determines the blade's pitching angle based on the given control vertices, where as the class `bSplineMotion.C` determines the position of the trajectory. The latter contains the Newton-Raphson procedure and several handling exceptions, enabling a continuous blade motion.

The tool kit for the optimisation is Dakota with an evolutionary algorithm and a population size of 128 due to the Hawk node. The number of cases within an optimisation run lies between 10 000 and 15 000. Some cases are aborted due to a lack of convergence or time. For the interface between Dakota and OpenFOAM, the Python script `operateDict.py` is written. It asses the trajectory, generates the cases with its essential input files and builds a unique mesh for every CFD calculation. The script also watches the running case and evaluates the results. Due to a failure in the procedure, the optimisation results with a constant angular velocity are invalid.

The optimiser captured unexpected and unconventional pitching paths and trajectories. The absolute maximum figure of merit of 0.758 is reached with four blades and with only pitching optimisation, Opti-4P.

Two types of plots are invented for the detailed evaluation of the optimal cases. The pitch-over-trajectory plot helps to understand the blade motion. And the more essential plot of the blade forces over the trajectory to figure out the optimiser's attempt. With these plots, two main reasons are investigated, which lead to a better figure of merit although, the optimisation runs differ from each other (number of blades, subject of optimisation).

- The optimiser adjusted the blade motion to better align the resulting blade forces to the global thrust.
- In almost every optimal case, there is a more or less distinct force peak generated by a fast blade motion. This movement is often a combination of rapid pitching and a narrow trajectory curvature.

However, this motion and the force peaks can lead to high structural loads and vibration, which in turn can cause fatigue fracture.

It is also noticeable that the main blade forces are predominantly generated in the lower half of the trajectory. The optimiser tries to reduce the required power for the remaining trajectory by decreasing the blade forces.

For the two-blade run with the 'pitching only' optimisation, two more Reynolds numbers are investigated; 100 000 and 200 000. The figure of merit is increased compared to the pitching path with a Re = 50 000 (+8.8% for Re=100 000, and +17.7% or Re=200 000). For a commercial cyclogyro, the efficiency of the hover flight can be adapted for different loads, which results in different RPMs.

It turned out that the reference surface for the FoM calculation does not fit well for the trajectory optimisation. The movement of the blades inclines the direction of the downwash, and thus the surface changes through which the flow passes. The correct rotor width is assumed by downwash, which is just a defective estimation.

A closer look at the thrust plots shows discontinuities of the force paths. There are kinks at each end of a pitching spline section. The corresponding spline motion causes jumps at the beginning of a trajectory. Despite great effort, this circumstance could not be solved until the end of this thesis.

For further research, some suggestions can be made.

- The current value of the chord-radius ratio is $\frac{2}{3}$, with which the optimiser captures motions, inducing force peaks. Lowering this ratio could lead to more uniform force distribution over the trajectory.
- More efficient use of computing capacity is possible by a convenient definition of the boundary space for the control vertices.
- The order of the pitching spline could be increased to five, which might help to reduce the kinks in the force plots.
- Using the figure of merit for the evaluation is weak due to the surface calculation. However, the ratio thrust to real power seems unfavourable, as this method neglects the shape of the trajectory. An algorithm could asses the velocity field close to the trajectory to determine the reference surface.
- Secondary evaluation factors like the uniformity of the force distribution, the aerodynamic efficiency $\frac{c_l}{c_d}$ or the vorticity could be implemented and sent to the optimiser to achieve more even results.
- The blade's inertia is neglected for optimisation. The resulting figure of merit could be too high in the cases, where the blades perform a rapid motion. Therefore the evaluation should contain the calculation of the blade's inertia and thus the required power.
- The number of calculated revolutions shall be increased as the current number of 14 (single case) and 10 (overset mesh) might not be sufficient to achieve a converged flow field. In turn, the boundary space of the control variable could be chosen more narrowly.
- Averaging the results over more than one revolution for the evaluation could help to get rid of numerical uncertainties.

12 Acknowledgements

First and foremost, I would like to thank my advisor Louis Gagnon for this interesting theme and his lasting support during the thesis. There were some intense debates, which gave me helpful input and helped. Also, his comprehensive knowledge not only about OpenFOAM and great readiness was constructive for this work.

I also would like to thank my colleagues for their lively exchange and help.

A special thank goes to the IT administration of the IAG and the High-Performance-Computing-Cente Stuttgart, who ensured a reliable run of the Clusters Prandtl and Hawk.

Lastly, I would like to thank my family for their encouragement during stressful periods and for everlasting support.

Bibliography

[1] John B Wheatley. Simplified aerodynamic analysis of the cyclogiro rotating wing system.
[2] John B Wheatley. Wind-tunnel test of a cyclogrio rotor.
[3] CycloTech GmbH. Accessed: 04.08.2022. URL: https://www.cyclotech.at/.
[4] Thomas Westermann Claus-Dieter Munz. Numerische Differentialgleichungen. Springer Vieweg Berlin, 2019. ISBN: 978-3-662-55886-7. DOI: https://doi.org/10.1007/978-3-662-55886-7.
[5] Tobias Holzmann. Mathematics, Numerics, Derivations and OpenFOAM(R). DOI: 10.13140/RG.2.2.27193.36960. URL: https://Holzmann-cfd.de.
[6] Jiyuan Tu, Guan Heng Yeoh, and Chaoqun Liu. Computational Fluid Dynamics: A Practical Approach. Third Edition. USA: Butterworth-Heinemann, 2018. ISBN: 978-0-08-098243-4. DOI: https://doi.org/10.1016/C2015-0-06135-4.
[7] OpenFOAM wiki. OverPimpleDyMFoam. Accessed: 13.06.2022. URL: https://openfoamwiki.net/index.php/OverPimpleDyMFoam.
[8] IdealSimulations. Turbulence Models In CFD. Accessed: 15.02.2022. URL: https://www.idealsimulations.com/resources/turbulence-models-in-cfd/.
[9] OpenCFD Ltd. OpenFOAM: User Guide v2112: Overset. Accessed: 04.03.2022. URL: https://www.openfoam.com/documentation/guides/latest/doc/guide-overset.html.
[10] Dominic D.J. Chandar and Jayanarayanan Sitaraman. "A flux correction approach for the pressure equation in incompressible flows on overset meshes in OpenFOAM". In: Computer Physics Communications 273 (2022), p. 108279. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2021.108279. URL: https://www.sciencedirect.com/science/article/pii/S001046552100391X.
[11] wolf dynamics. Dynamic meshes in OpenFOAM. Accessed: 19.05.2022. URL: http://www.wolfdynamics.com/training/movingbodies/OF2021/dynamicmeshes_2021_OF8.pdf.
[12] UIUC Applied Aerodynamic Group. UIUC Airfoil Coordinates Database. 2022. URL: https://m-selig.ae.illinois.edu/ads/coord_database.html#N.
[13] Doudou Huang and Louis Gagnon. "Relying on Dynamically Morphing Blades to Increase the Efficiency of a Cycloidal Rotor". In: IOP Conference Series: Materials Science and Engineering 1226.1 (2022), p. 012014. DOI: 10.1088/1757-899x/1226/1/012014. URL: https://doi.org/10.1088/1757-899x/1226/1/012014.
[14] Florian Zimmer and Louis Gagnon. "Investigation of the Reynolds Number on the Performance of a Cycloidal Rotor". In: under review in J. Fluids Eng. (2022).
[15] VDI e. V., ed. VDI-Wärmeatlas.
[16] Philip Borgash. "Cycloidal rotor with non-circular blade orbit". Pat. US 2009/0226314 A1. 2009.
[17] OpenCFD Ltd. OpenFOAM: User Guide v2112: k-omega Shear Stress Transport (SST). Accessed: 15.02.2022. URL: https://www.openfoam.com/documentation/guides/latest/doc/guide-turbulence-ras-k-omega-sst.html.
[18] David F. Rogers. "An Introduction to NURBS: With Historical Perspective". In: (2000).
[19] Lothar Papula. Mathematische Formelsammlung. 10. Auflage. Vieweg+Teubner, 2009. ISBN: 978-3-8348-0757-1. DOI: https://doi.org/10.1007/978-3-8348-9598-1.
[20] Berend Gerdes van der Wall. Grundlagen der Hubschrauber-Aerodynamik. Braunschweig: Springer-Verlag Berlin Heidelberg, 2015. ISBN: 978-3-662-44399-6. DOI: 10.1007/978-3-662-44400-9.

List of Figures

List of Tables

Appendix

A Input Data

Table A.1: Listing of all geometric dimensions used over all cases.

Name	Symbol	Value	Unit	Source
Type of airfoil		NACA0012	[−]	project definition
Chord lenth, single blade	c_{SB}	1.0011	[m]	calculated
Chord lenth, multi blade	c_{MB}	0.9922	[m]	calculated
Chord-Radius ratio	R_c	$\frac{2}{3}$	[−]	project definition
Radius of rotation	R	1.5	[m]	calculated du to given ratio
Number of blades	n_{blade}	1, 2, 3, 4	[−]	project definition
Depth of domain	d_z	1.0	[m]	unit value

Table A.2: Considered Reynolds numbers, rotational speed and period.

Reynolds number	Factor	Rotational speed	Period
50 000	1	0.5167	12.1608
100 000	2	1.0334	6.0804
200 000	4	2.0667	3.0402

Table A.3: Listing of all constant boundary conditions used over all cases.

		Value	Unit
Reynolds number	Re	50 000	[−]
Air density	ρ	1.225	$\left[\frac{\mathrm{kg}}{\mathrm{m}^3}\right]$
Kinematic viscosity	ν	$15.5 \cdot 10^{-6}$	$\left[\frac{\mathrm{m}^2}{\mathrm{s}}\right]$
Chord lenth	c	1.0	[m]
Free stream velocity	U_∞	0	$\left[\frac{\mathrm{m}}{\mathrm{s}}\right]$

B Dakota

The basic Dacota input file is shown below, where the code passages marked with *** were adjusted according to the optimisation, see following comments.

```
environment
     top_method_pointer = 'SOGA'

method
  id_method = 'SOGA'
  model_pointer = 'M1'
  soga
    fitness_type merit_function
    population_size = 128
    max_iterations = 1024
    max_function_evaluations = 262144
    convergence_tolerance = ***

    scaling
    seed = 123456

model
  id_model = 'M1'
  single
    variables_pointer = 'V1'
    interface_pointer = 'I1'
    responses_pointer = 'R1'

variables
  id_variables = 'V1'
  continuous_design = ***
    initial_point     ***
    lower_bounds      ***
    upper_bounds      ***
    descriptors       ***
    scale_types      'auto'

interface
  id_interface = 'I1'
  analysis_driver = ***
    fork asynchronous evaluation_concurrency = 128
    parameters_file = 'parameters.in'
    results_file    = 'results.out'
    file_tag

responses
  id_responses = 'R1'
  objective_functions = 1
  no_gradients
  no_hessians
```

Code passage for 'pitching only' optimisation.

```
continuous_design = 16
  initial_point      0     17.7    32.7     42.7    46.2    42.7    32.7    17.7    0      -17.7   -32.7   -42.7   -46.2   -42.7    -32.7   -17.7
  lower_bounds      -45    -31.6   -20.3    -12.7   -10     -12.7   -20.3   -31.6   -45     -58.4   -69.8   -77.3   -80   -77.3   -69.8   -58.4
  upper_bounds       45     58.4    69.8     77.4    80      77.4    69.8    58.4    45      31.6    20.3    12.7    10   12.7    20.3    31.6
  descriptors       'P10' 'P11'   'P20'  'P21'  'P30'  'P31'  'P40'  'P41'  'P50'  'P51'  'P60'  'P61'  'P70'  'P71'  'P80'   'P81'
```

Code passage for 'trajectory only' optimisation.

```
continuous_design = 14
  initial_point     1.7      1.2      1.2      0.0     1.7      -1.2      1.2     -1.7     -1.2     -1.2      0.0    -1.7   1.2     -1.2
  lower_bounds      0.2      0.2      0.2     -2.0     0.2      -3.0      0.2     -3.0     -3.0     -3.0     -2.0    -3.0   0.2     -3.0
  upper_bounds      3.0      3.0      3.0      2.0     3.0      -0.2      3.0     -0.2     -0.2     -0.2      2.0    -0.2   3.0     -0.2
  descriptors      'T1X'    'T2X'    'T2Y'    'T3X'    'T3Y'    'T4X'    'T4Y'    'T5X'    'T6X'    'T6Y'    'T7X'   'T7Y'    'T8X'    'T8Y'
```

Code passage for 'both', pitching and trajectory optimisation.

```
continuous_design = 30
  initial_point     0     17.7    32.7     42.7    46.2    42.7    32.7    17.7    0      -17.7   -32.7   -42.7   -46.2   -42.7   -32.7   -17.7        1.7      1.2      1.2      0.0     1.7      -1.2      1.2     -1.7     -1.2     -1.2   0.0     -1.7      1.2     -1.2
  lower_bounds     -45    -31.6   -20.3    -12.7   -10     -12.7   -20.3   -31.6   -45     -58.4   -69.8   -77.3   -80   -77.3   -69.8   -58.4        1.0      0.5      0.5     -2.0     0.5      -3.0      0.5     -3.0     -3.0     -3.0   -2.0     -3.0      0.5     -3.0
  upper_bounds      45     58.4    69.8     77.4    80      77.4    69.8    58.4    45      31.6    20.3    12.7    10   12.7    20.3    31.6         3.0      3.0      3.0      2.0     3.0      -0.5      3.0     -1.0     -0.5     -0.5   2.0     -0.5      3.0     -0.5
  descriptors      'P10' 'P11'   'P20'  'P21'  'P30'  'P31'  'P40'  'P41'  'P50'  'P51'  'P60'  'P61'  'P70'  'P71'  'P80'  'P81'       'T1X'    'T2X'    'T2Y'    'T3X'    'T3Y'    'T4X'    'T4Y'    'T5X'    'T6X'    'T6Y'   'T7X'    'T7Y'    'T8X'    'T8Y'
```

C Python Script

The scripts depend on the number of the blade and the optimisation subject. The shown code is valid for pitching, trajectory optimisation, and constant velocity.

C.1 operateDict.py

```
#!/usr/bin/env python

import sys
import os
from shapely.geometry.polygon import LinearRing
import shutil
from subprocess import Popen
import time
import Control_***
from dakota import interfacing as di

sys.path.append('/zhome/academic/HLRS/iag/iagdonne/dakota-6.15.0-public-rhel7.Linux.x86_64-cli/
↪ share/dakota/Python')

inputFileFromDak = sys.argv[1]
outputFileForDak = sys.argv[2]

parameters, results = di.read_parameters_file(inputFileFromDak, outputFileForDak)

P10 = round(parameters['P10'],4)
P11 = round(parameters['P11'],4)
P20 = round(parameters['P20'],4)
P21 = round(parameters['P21'],4)
P30 = round(parameters['P30'],4)
P31 = round(parameters['P31'],4)
P40 = round(parameters['P40'],4)
P41 = round(parameters['P41'],4)
P50 = round(parameters['P50'],4)
P51 = round(parameters['P51'],4)
P60 = round(parameters['P60'],4)
P61 = round(parameters['P61'],4)
P70 = round(parameters['P70'],4)
P71 = round(parameters['P71'],4)
P80 = round(parameters['P80'],4)
P81 = round(parameters['P81'],4)

T1X = round(parameters['T1X'],6)
T2X = round(parameters['T2X'],6)
T2Y = round(parameters['T2Y'],6)
T3X = round(parameters['T3X'],6)
T3Y = round(parameters['T3Y'],6)
T4X = round(parameters['T4X'],6)
T4Y = round(parameters['T4Y'],6)
T5X = round(parameters['T5X'],6)
T6X = round(parameters['T6X'],6)
T6Y = round(parameters['T6Y'],6)
T7X = round(parameters['T7X'],6)
T7Y = round(parameters['T7Y'],6)
T8X = round(parameters['T8X'],6)
```

```python
T8Y = round(parameters['T8Y'],6)

#--- Definition ---------------------------------------------------------------
eval_id = int(parameters.eval_id)
cwd = os.getcwd().replace('/OverHead', '')
src = cwd + '/IdleRun***'
dst = cwd + '/Run_' + str(eval_id).zfill(5)
WriteLineNew = []

#--- Check Trajectory for intersections ------------------------------------------------------------
Ring = LinearRing([ (T1X, 0), (T2X, T2Y), (T3X, T3Y), (T4X, T4Y), (T5X, 0), (T6X, T6Y), (T7X, T7Y), (T8X, T8Y), (T1X, 0),  ])
flagInter = Ring.is_valid

if flagInter == True:
    #--- Generate Case ------------------------------------------------------------
    shutil.copytree(src, dst)

    #--- Manipulate dynamicMeshDict AND initialConditions ---------------------------------------
    Control_***.Pitching(dst, P10, P11, P20, P21, P30, P31, P40, P41, P50, P51, P60, P61, P70, P71, P80, P81)
    OutPut = Control_***.Trajectory(dst, T1X, T2X, T2Y, T3X, T3Y, T4X, T4Y, T5X, T6X, T6Y, T7X, T7Y, T8X, T8Y, P10, P11, P20, P21, P30, P31, P40, P41, P50, P51, P60, P61, P70, P71, P80, P81)
    Polygon = OutPut[0]
    maxKappa = OutPut[1]

    if maxKappa < 4:
        #--- Run Case ----------------------------------------------------------------
        os.chdir(dst)
        Popen(['sh', dst +  '/Run.sh'])
        time.sleep(240)

        #--- LiveTicker for Case ------------------------------------------------------
        flagRun = True
        flagState = False
        TimeOut = False
        RunTime = 0
        counter = 0

        while flagRun == True:
            time.sleep(300)
            LiveTiming = Control_***.LiveTiming(dst)
            flagRun = LiveTiming[0]
            flagState = LiveTiming[1]

            if RunTime == LiveTiming[3]:
                counter +=1
            else:
                counter = 0

            if counter == 40:
                flagRun = False
                flagState = False
                TimeOut = True

            RunTime = LiveTiming[3]
        #--- get FOM if Case succsessful -----------------------------------------------
        if flagState == True:
            Eval = Control_***.EvaluateCase(dst)
            if Eval[7] > 0 and Eval[7] < 1:
```

```
                FOM = Eval[7]
                WriteLineNew.append('\n'+str(eval_id).zfill(5)+'%3d' %LiveTiming[2]+'%+.4E' %Eval[0] +'%+.4E' %Eval[1] +'%.4E' %Eval[2] +'%.4E' %Eval[3] +'%.4E' %Eval[4] +'%.4E' %Eval[5] +'%.4E' %Eval[6] +'%.4E' %Eval[7] +'%.3E' %Eval[8] +'%.4f' %Eval[9] +'%+.6E' %P10 +'%+.6E' %P11 +'%+.6E' %P20 +'%+.6E' %P21 +'%+.6E' %P30 +'%+.6E' %P31 +'%+.6E' %P40 +'%+.6E' %P41 +'%+.6E' %P50 +'%+.6E' %P51 +'%+.6E' %P60 +'%+.6E' %P61 +'%+.6E' %P70 +'%+.6E' %P71 +'%+.6E' %P80 +'%+.6E' %P81 +'%+.6E' %Polygon[2][0] +'%+.6E' %Polygon[3][0] +'%+.6E' %Polygon[3][1] +'%+.6E' %Polygon[4][0] +'%+.6E' %Polygon[4][1] +'%+.6E' %Polygon[5][0] +'%+.6E' %Polygon[5][1] +'%+.6E' %Polygon[6][0] +'%+.6E' %Polygon[7][0] +'%+.6E' %Polygon[7][1] +'%+.6E' %Polygon[8][0] +'%+.6E' %Polygon[8][1] +'%+.6E' %Polygon[9][0] +'%+.6E' %Polygon[9][1])
            else:
                FOM = 1e-4
                WriteLineNew.append('\n'+str(eval_id).zfill(5)+ '--> FalseFOM %+.4E' %Eval[0] +...)
        elif TimeOut == True:
            FOM = 1e-4
            WriteLineNew.append('\n'+str(eval_id).zfill(5)+'--> TimeOut!  %5d' %RunTime+ '%+.6E' %P10 +...)
        else:
            FOM = 1e-4
            WriteLineNew.append('\n'+str(eval_id).zfill(5)+'--> OpenFOAM error! %+.6E' %P10 +...)
    else:
        FOM = 1e-5
        WriteLineNew.append('\n'+str(eval_id).zfill(5)+'--> max. Kappa =%.3E'%maxKappa+'too high!%+.6E'%P10+...)
    #--- Clean Case ---------------------------
    Control_***.CleanCase(cwd, eval_id)

else:
    FOM = 1e-6
    WriteLineNew.append('\n' + str(eval_id).zfill(5) + '--> Intersection! %+.6E' %P10 +...+ '%+.6E' %T8Y )

with open(cwd + '/OverHead/RunLog.txt', 'at') as meanValue:
    WriteLine= ''.join(WriteLineNew)
    meanValue.write(WriteLine)

results['obj_fn'].function =  1 / FOM

os.chdir(cwd + '/OverHead')
results.write()
```

C.2 Control.py

The control script contains the essential function, is shown separately, and is valid for optimisation subject 'both'.

```
import numpy as np
import re
import os
import math
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon
from stl import mesh
import shutil
```

```python
def Pitching(dst, P10, P11, P20, P21, P30, P31, P40, P41, P50, P51, P60, P61, P70, P71, P80, P81):
    with open(dst + '/constant/dynamicMeshDict', 'rt') as DMC:
        DMClines = DMC.readlines()
        for iLine in range(len(DMClines)):
            if re.search(r'vertexP_10', DMClines[iLine]):
                DMClines[iLine]='vertexP_10  ' + str(P10) +';\n'
                DMClines[iLine+1]='vertexP_11  ' + str(P11) +';\n'
                DMClines[iLine+2]='vertexP_20  ' + str(P20) +';\n'
                DMClines[iLine+3]='vertexP_21  ' + str(P21) +';\n'
                DMClines[iLine+4]='vertexP_30  ' + str(P30) +';\n'
                DMClines[iLine+5]='vertexP_31  ' + str(P31) +';\n'
                DMClines[iLine+6]='vertexP_40  ' + str(P40) +';\n'
                DMClines[iLine+7]='vertexP_41  ' + str(P41) +';\n'
                DMClines[iLine+8]='vertexP_50  ' + str(P50) +';\n'
                DMClines[iLine+9]='vertexP_51  ' + str(P51) +';\n'
                DMClines[iLine+10]='vertexP_60  ' + str(P60) +';\n'
                DMClines[iLine+11]='vertexP_61  ' + str(P61) +';\n'
                DMClines[iLine+12]='vertexP_70  ' + str(P70) +';\n'
                DMClines[iLine+13]='vertexP_71  ' + str(P71) +';\n'
                DMClines[iLine+14]='vertexP_80  ' + str(P80) +';\n'
                DMClines[iLine+15]='vertexP_81  ' + str(P81) +';\n'
    with open(dst +'/constant/dynamicMeshDict', 'wt') as DMC:
        DMClinesNew = ''.join(DMClines)
        DMC.write(DMClinesNew)
```

```python
def Trajectory(dst, T1X, T2X, T2Y, T3X, T3Y, T4X, T4Y, T5X, T6X, T6Y, T7X, T7Y, T8X, T8Y, P10, P11, P20, P21, P30, P31, P40, P41, P50, P51, P60, P61, P70, P71, P80, P81):
    #--- Input Data ----------------------------------------------------------
    flagLen = False
    Ratio = 1
    TPolygon = np.array([[T7X, T7Y], [T8X, T8Y],[T1X, 0], [T2X, T2Y], [T3X, T3Y], [T4X, T4Y], [T5X, 0], [T6X, T6Y], [T7X, T7Y], [T8X, T8Y], [T1X, 0], [T2X, T2Y], [T3X, T3Y],   ])
    deltaT = 1e-4
    Period = 10
    counter = 0
    lengthR = 3*math.pi

    fTra = np.array( [
         [1/24, -1/6, 1/4, -1/6, 1/24],
         [-1/6, 1/2, -1/4, -1/2, 11/24],
         [1/4, -1/2, -1/4, 1/2, 11/24],
         [-1/6, 1/6, 1/4, 1/6, 1/24],
         [1/24, 0, 0, 0, 0] ],)
    dfTra = np.array( [
        [1/6, -1/2, 1/2, -1/6],
        [-4/6, 3/2, -1/2, -1/2],
        [1, -3/2, -1/2, 1/2],
        [-4/6, 1/2, 1/2, 1/6],
        [1/6, 0, 0, 0],],)
    ddfTra = np.array( [
        [1/2, -1, 1/2],
        [-2, 3, -1/2],
        [3, -3, -1/2],
        [-2, 1, 1/2],
        [1/2, 0, 0],],)
    fLen = np.array( [
        [   [1, -4, 6, -4, 1],],
        [   [-15/8, 7, -9, 4, 0],
            [1/8, -1/2, 3/4, -1/2, 1/8],],
        [   [85/72, -11/3, 3, 0, 0],
            [-23/72, 19/18, -11/12, -5/18, 37/72],
            [1/18, -2/9, 1/3, -2/9, 1/18],],
        [   [-25/72, 2/3, 0, 0, 0],
            [ 23/72, -13/18,  -1/12, 11/18, 23/72],
            [-13/72, 5/9, -1/3, -4/9, 4/9],
            [1/24, -1/6, 1/4, -1/6, 1/24],],
        [   [1/24, 0, 0, 0, 0,],
            [-1/6, 1/6, 1/4, 1/6, 1/24],
            [1/4, -1/2, -1/4, 1/2, 11/24],
            [-1/6, 1/2, -1/4, -1/2, 11/24],
            [1/24, -1/6, 1/4, -1/6, 1/24],],    ], dtype = 'object')

    fIndex = np.array( [
        [   [0, 1, 2, 3, 4],
            [1, 2, 3, 4, 4],
            [2, 3, 4, 4, 4],
            [3, 4, 4, 4, 4],
            [4, 4, 4, 4, 4],
            [4, 4, 4, 4, 3],
            [4, 4, 4, 3, 2],
            [4, 4, 3, 2, 1],
            [4, 3, 2, 1, 0], ],


```

```
        [   [0, 0, 0, 0, 0],
            [1, 1, 1, 1, 0],
            [2, 2, 2, 1, 0],
            [3, 3, 2, 1, 0],
            [4, 3, 2, 1, 0],
            [4, 3, 2, 1, 3],
            [4, 3, 2, 2, 2],
            [4, 3, 1, 1, 1],
            [4, 0, 0, 0, 0], ], ], dtype = 'object')
    fPit = np.array( [
        [-1/6, 1/2, -1/2, 1/6],
        [1/2, -1, 0, 2/3],
        [-1/2, 1/2, 1/2, 1/6],
        [1/6, 0, 0, 0] ],)

    P00 = P81
    P90 = P10
    P91 = P11

    Pitching = np.array([P00, P10, P11, P20, P21, P30, P31, P40, P41, P50, P51, P60, P61, P70, P71, P80, P81, P90, P91])
    #--- end: Input Data ---------------------------------------------------------

    #--- Calculate Trajectory ---------------------------------------------------------
    while flagLen == False:
        time = 0
        vecTX, vecTY, vecTTime, vecTLen = [], [], [], []
        for i in range(0,13):
            TPolygon[i,0] = TPolygon[i,0] / Ratio
            TPolygon[i,1] = TPolygon[i,1] / Ratio
        while time < Period + deltaT:
            k = math.floor( time / Period * 8 + 0.5)
            TnT = 0.5 + 8 * time / Period - k
            vecTTime.append(round(TnT+k,6))
            TR1 = fTra[0][0]*TnT**4+fTra[0][1]*TnT**3+fTra[0][2]*TnT**2+fTra[0][3]*TnT+fTra[0][4]
            TR2 = fTra[1][0]*TnT**4+fTra[1][1]*TnT**3+fTra[1][2]*TnT**2+fTra[1][3]*TnT+fTra[1][4]
            TR3 = fTra[2][0]*TnT**4+fTra[2][1]*TnT**3+fTra[2][2]*TnT**2+fTra[2][3]*TnT+fTra[2][4]
            TR4 = fTra[3][0]*TnT**4+fTra[3][1]*TnT**3+fTra[3][2]*TnT**2+fTra[3][3]*TnT+fTra[3][4]
            TR5 = fTra[4][0]*TnT**4+fTra[4][1]*TnT**3+fTra[4][2]*TnT**2+fTra[4][3]*TnT+fTra[4][4]
            vecTX.append(TR1*TPolygon[k,0]+TR2*TPolygon[k+1,0]+TR3*TPolygon[k+2,0]+TR4*TPolygon[k+3,0]+TR5*TPolygon[k+4,0])
            vecTY.append(TR1*TPolygon[k,1]+TR2*TPolygon[k+1,1]+TR3*TPolygon[k+2,1]+TR4*TPolygon[k+3,1]+TR5*TPolygon[k+4,1])
            time = time + deltaT
        lengthT = 0
        for k in range(0, len(vecTX)-1):
            deltaTX = vecTX[k+1] - vecTX[k]
            deltaTY = vecTY[k+1] - vecTY[k]
            deltaL = math.sqrt(deltaTX**2+deltaTY**2)
            lengthT = lengthT + deltaL
            vecTLen.append(lengthT)
        Ratio = (lengthT/lengthR)
        counter +=1
        if abs(1-Ratio) < 1e-6:
            flagLen = True
        if counter > 20:
            flagLen = True
    #--- end: Calculate Trajectory ---------------------------------------------------------
```

```
    #--- Calculate Curvature ------------------------------------------------------------------
    time = 0
    vecTCA, vecTdX, vecTdY, vecTddX, vecTddY, kappa = [], [], [], [], [], []

    while time < Period + deltaT:
        k = math.floor( time / Period * 8 + 0.5)
        TnT = 0.5 + 8 * time / Period - k

        TdR1 = dfTra[0][0]*TnT**3+dfTra[0][1]*TnT**2+dfTra[0][2]*TnT+dfTra[0][3]
        TdR2 = dfTra[1][0]*TnT**3+dfTra[1][1]*TnT**2+dfTra[1][2]*TnT+dfTra[1][3]
        TdR3 = dfTra[2][0]*TnT**3+dfTra[2][1]*TnT**2+dfTra[2][2]*TnT+dfTra[2][3]
        TdR4 = dfTra[3][0]*TnT**3+dfTra[3][1]*TnT**2+dfTra[3][2]*TnT+dfTra[3][3]
        TdR5 = dfTra[4][0]*TnT**3+dfTra[4][1]*TnT**2+dfTra[4][2]*TnT+dfTra[4][3]

        vecTdX.append(TdR1*TPolygon[k,0]+TdR2*TPolygon[k+1,0]+TdR3*TPolygon[k+2,0]+TdR4*TPolygon[k+3,0] +TdR5*TPolygon[k+4,0])
        vecTdY.append(TdR1*TPolygon[k,1]+TdR2*TPolygon[k+1,1]+TdR3*TPolygon[k+2,1]+TdR4*TPolygon[k+3,1] +TdR5*TPolygon[k+4,1])
        vecTCA.append(-math.atan2(vecTdX[-1],vecTdY[-1]))

        TddR1 = ddfTra[0][0]*TnT**2+ddfTra[0][1]*TnT+ddfTra[0][2]
        TddR2 = ddfTra[1][0]*TnT**2+ddfTra[1][1]*TnT+ddfTra[1][2]
        TddR3 = ddfTra[2][0]*TnT**2+ddfTra[2][1]*TnT+ddfTra[2][2]
        TddR4 = ddfTra[3][0]*TnT**2+ddfTra[3][1]*TnT+ddfTra[3][2]
        TddR5 = ddfTra[4][0]*TnT**2+ddfTra[4][1]*TnT+ddfTra[4][2]

        vecTddX.append(TddR1*TPolygon[k,0]+TddR2*TPolygon[k+1,0]+TddR3*TPolygon[k+2,0]+TddR4*TPolygon[k+3,0] +TddR5*TPolygon[k+4,0])
        vecTddY.append(TddR1*TPolygon[k,1]+TddR2*TPolygon[k+1,1]+TddR3*TPolygon[k+2,1]+TddR4*TPolygon[k+3,1] +TddR5*TPolygon[k+4,1])
        kappa.append( abs( vecTdX[-1] * vecTddY[-1] - vecTdY[-1] * vecTddX[-1] ) / ( vecTdX[-1]**2 + vecTdY[-1]**2 )**(3/2) )

        time = time + deltaT

    maxKappa = max(kappa)
    #--- end: Calculate Curvature -------------------------------------------------------------

    if maxKappa < 4:
        #--- Calculate vectorl & sectionL ---------------------------------------------------------
        vectorL, sectionL = [], []

        vecLnT = [0, 0.525, 1.225, 1.75, 2.8, 3.675, 4.725, 5.6, 6.65, 7.525, 8.575, 9.45, 10.5, 11.375, 12.425, 13.475, 14.35, 15.225, 16.275, 17.325, 18.2, 19.25, 19.775, 20.475, 21]
        vecPD = [vecTLen[vecTTime.index(7.7)]-lengthR, vecTLen[vecTTime.index(7.94)]-lengthR, vecTLen[vecTTime.index(8.26)]-lengthR, vecTLen[vecTTime.index(0.5)], vecTLen[vecTTime.index(0.98)], vecTLen[vecTTime.index(1.38)], vecTLen[vecTTime.index(1.86)], vecTLen[vecTTime.index(2.26)], vecTLen[vecTTime.index(2.74)], vecTLen[vecTTime.index(3.14)], vecTLen[vecTTime.index(3.62)], vecTLen[vecTTime.index(4.02)], vecTLen[vecTTime.index(4.5)], vecTLen[vecTTime.index(4.9)], vecTLen[vecTTime.index(5.38)], vecTLen[vecTTime.index(5.86)], vecTLen[vecTTime.index(6.26)], vecTLen[vecTTime.index(6.66)], vecTLen[vecTTime.index(7.14)], vecTLen[vecTTime.index(7.62)], vecTLen[vecTTime.index(8.02)], vecTLen[vecTTime.index(8.5)], vecTLen[vecTTime.index(0.74)]+lengthR, vecTLen[vecTTime.index(1.06)]+lengthR, vecTLen[vecTTime.index(1.3)]+lengthR]

        matN = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        sec = 0
```

```
        for j in range(1,24):
            vecN = []
            k = math.floor(vecLnT[j])
            matR = []
            if k < 4:
                sec = k
            elif k < 17:
                sec = 4
            else:
                sec = k - 12

            for i in range(0,5):
                matR.append(fLen[fIndex[0][sec][i]][fIndex[1][sec][i]])
            for m in range(0,k):
                vecN.append(0)

            LnT = vecLnT[j]-k
            for i in range(0,5):
                if vecLnT[j] > 17:
                    if i == 21- k:
                        LnT = 1 - LnT
                vecN.append(round(matR[i][0]*LnT**4+matR[i][1]*LnT**3+matR[i][2]*LnT**2
                ↪  +matR[i][3]*LnT+matR[i][4],8))

            for m in range(k+4,24):
                vecN.append(0)

            matN  = np.vstack([matN,vecN])

        vecN = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
        matN = np.vstack([matN,vecN])
        matInvN = np.linalg.inv(matN)
        vectorL=np.matmul(matInvN, vecPD)

        #--- Calculate: sectionL
        matR=[]
        for j in range(0,21):
            if j < 4:
                sec = j
            elif j < 17:
                sec = 4
            else:
                sec = j - 12

            for i in range(0,5):
                matR.append(  fLen[  fIndex[0][sec][i]  ] [  fIndex[1][sec][i]  ]  )

        for i in np.arange(1,21,1):
            k = math.floor(i)
            LnT = i - k

            LR1 = matR[0+k*5][0]*LnT**4+matR[0+k*5][1]*LnT**3+matR[0+k*5][2]*LnT**2+matR[0+k*5][3]*LnT
            ↪  +matR[0+k*5][4]

            if k >= 20:
                LnT = 1 - (i - k)
            LR2 = matR[1+k*5][0]*LnT**4+matR[1+k*5][1]*LnT**3+matR[1+k*5][2]*LnT**2+matR[1+k*5][3]*LnT
            ↪  +matR[1+k*5][4]

```

```
            if k >= 19:
                LnT = 1 - (i - k)
            LR3 = matR[2+k*5][0]*LnT**4+matR[2+k*5][1]*LnT**3+matR[2+k*5][2]*LnT**2+matR[2+k*5][3]*LnT +matR[2+k*5][4]

            if k >= 18:
                LnT = 1 - (i - k)
            LR4 = matR[3+k*5][0]*LnT**4+matR[3+k*5][1]*LnT**3+matR[3+k*5][2]*LnT**2+matR[3+k*5][3]*LnT +matR[3+k*5][4]

            if k >= 17:
                LnT = 1 - (i - k)
            LR5 = matR[4+k*5][0]*LnT**4+matR[4+k*5][1]*LnT**3+matR[4+k*5][2]*LnT**2+matR[4+k*5][3]*LnT +matR[4+k*5][4]

            sectionL.append(LR1*vectorL[k]+LR2*vectorL[k+1]+LR3*vectorL[k+2]+LR4*vectorL[k+3] +LR5*vectorL[k+4])
        #--- end: Calculate vectorl & sectionL ----------------------------------------------------------

        #--- Calculate Pitching --------------------------------------------------
        time = 0
        vecPnT, vecPit = [], []
        while time <  Period:
            k = math.floor( time / Period * 16)
            PnT = 16 * time / Period - k
            vecPnT.append(round(16 * time / Period,4))

            PR1 = fPit[0][0]*PnT**3+fPit[0][1]*PnT**2+fPit[0][2]*PnT+fPit[0][3]
            PR2 = fPit[1][0]*PnT**3+fPit[1][1]*PnT**2+fPit[1][2]*PnT+fPit[1][3]
            PR3 = fPit[2][0]*PnT**3+fPit[2][1]*PnT**2+fPit[2][2]*PnT+fPit[2][3]
            PR4 = fPit[3][0]*PnT**3+fPit[3][1]*PnT**2+fPit[3][2]*PnT+fPit[3][3]

            vecPit.append(PR1*Pitching[k]+PR2*Pitching[k+1]+PR3*Pitching[k+2]+PR4*Pitching[k+3])
            time = round(time + deltaT,10)
        #--- end: Calculation Pitching --------------------------------------------------

        #--- Genearte STL ------------------------------------------------------------
        BBox = np.array([ [0.25, 0.95], [0.65, 0.75], [0.85, 0.5], [1, 0.25], [1, 0], [1, -0.25], [1, -0.5], [1, -0.75], [0.85, -1], [0.65, -1.25], [0.25, -1.4] ])
        Q1X, Q1Y, Q2X, Q2Y, Q3X, Q3Y, Q4X, Q4Y,   = [], [], [], [], [], [], [], [] # list for each Quadrant

        for i in range(0, len(vecTX)-2,10):  # calculate scatter points
            for j in range(0,len(BBox)):
                scatterX = ( vecTX[i] + BBox[j][0]*math.cos(-vecTCA[i]+math.radians(vecPit[i])) + BBox[j][1]*math.sin(-vecTCA[i]+math.radians(vecPit[i])) )
                scatterY = ( vecTY[i] - BBox[j][0]*math.sin(-vecTCA[i]+math.radians(vecPit[i])) + BBox[j][1]*math.cos(-vecTCA[i]+math.radians(vecPit[i])) )

                if scatterX > 0 and scatterY > 0:
                    Q1X.append(scatterX)
                    Q1Y.append(scatterY)
                elif scatterX < 0 and scatterY > 0:
                    Q2X.append(scatterX)
                    Q2Y.append(scatterY)
                elif scatterX < 0 and scatterY < 0:
                    Q3X.append(scatterX)
                    Q3Y.append(scatterY)
                else:
                    Q4X.append(scatterX)
                    Q4Y.append(scatterY)
```

```
        allX, allY = [], []
        allX.append(Q1X)
        allX.append(Q2X)
        allX.append(Q3X)
        allX.append(Q4X)
        allY.append(Q1Y)
        allY.append(Q2Y)
        allY.append(Q3Y)
        allY.append(Q4Y)

        triLen = 10
        Dividor = 20
        triAngle = math.pi / (2* Dividor)
        refineX, refineY = [], []

        for i in range(0,4):
            for j in range(0, Dividor):
                matchX, matchY, matchR = [], [], []
                rotAngle = - (j * triAngle + i * math.pi/2)
                addAngle = rotAngle - triAngle
                triPoly = Polygon([(0, 0), (triLen*math.cos(rotAngle), -triLen*math.sin(rotAngle)), (triLen*math.cos(addAngle), -triLen*math.sin(addAngle))])

                for k in range(0, len(allX[i])):
                    point = Point(allX[i][k], allY[i][k])
                    flagIN = triPoly.contains(point)

                    if flagIN == True:
                        matchX.append(allX[i][k])
                        matchY.append(allY[i][k])
                        matchR.append(math.sqrt( allX[i][k]**2 + allY[i][k]**2 ))
                refineX.append(matchX[matchR.index(max(matchR))])
                refineY.append(matchY[matchR.index(max(matchR))])

        lenR = len(refineX)
        nVer =  len(refineX) *2 +2
        vertMesh1 = np.zeros((nVer,3))

        for i in range(0,lenR):
            vertMesh1[i,:] = [refineX[i], refineY[i], 2]
            vertMesh1[i+lenR,:] = [refineX[i], refineY[i], -2]

        vertMesh1[-2,:] = [0, 0, 2]
        vertMesh1[-1,:] = [0, 0, -2]
        nFace = lenR *2  *2
        facMesh = np.zeros((nFace,3))
        lenV = len(vertMesh1)

        for i in range(0,lenR):
            sub = 0
            if i == lenR -1:
                sub = lenR

            facMesh[i*2,:] = [i, i+lenR, i+1 - sub]
            facMesh[i*2+1,:] = [i+1 -sub, i+lenR, i+1+lenR - sub]
            facMesh[i+2*lenR,:] = [lenV-2, i , i+1 -sub]
            facMesh[i+3*lenR,:] = [lenV-1, i+lenR, i+1+lenR -sub]

        Refine = mesh.Mesh(np.zeros(facMesh.shape[0], dtype=mesh.Mesh.dtype))
```

```python
        for i, f in enumerate(facMesh):
            for j in range(3):
                Refine.vectors[i][j] = vertMesh1[int(f[j]),:]
        Refine.save(dst +'/constant/triSurface/Refine.stl')
        #--- end: Genearte STL -----------------------------------------------------------

        #--- Update: fvSolution -----------------------------------------------------------
        BoxXmin = str( round( min(refineX) - 0.1,2) )
        BoxXmax = str( round( max(refineX) + 0.1,2) )
        BoxYmin = str( round( min(refineY) - 0.1,2) )
        BoxYmax = str( round( max(refineY) + 0.1,2) )

        with open(dst + '/system/fvSchemes', 'rt') as fvSchemes:
            fvLines = fvSchemes.readlines()

            for iLine in range(len(fvLines)):
                if re.search(r'searchBox', fvLines[iLine]):
                    fvLines[iLine]='searchBox (' + BoxXmin + ' ' + BoxYmin + ' -1)(' + BoxXmax + ' ' + BoxYmax + ' 1);\n'

        with open(dst + '/system/fvSchemes', 'wt') as FVS:
            fvLinesNew = ''.join(fvLines)
            FVS.write(fvLinesNew)

        #--- Update: initialConditions -----------------------------------------------------------
        RotorWidth = max(vecTX) - min(vecTX)
        with open(dst + '/system/initialConditions', 'rt') as initCond:
            ICLines = initCond.readlines()
            for iLine in range(len(ICLines)):
                if re.search(r'RotorWidth', ICLines[iLine]):
                    ICLines[iLine]='RotorWidth          ' + str(round(RotorWidth,6)) +';\n'

        with open(dst + '/system/initialConditions', 'wt') as DMC:
            ICLinesNew = ''.join(ICLines)
            DMC.write(ICLinesNew)

        #--- Write dynamicMeshDict ----------------------------------------------------------------------
        wrVec, wrSec = '', ''
        for i in range(0, len(vectorL)):
            wrVec= wrVec + (str(round(vectorL[i],6))) + '  '

        for i in range(0, len(sectionL)):
            wrSec= wrSec + (str(round(sectionL[i],6))) + '  '

        with open(dst + '/constant/dynamicMeshDict', 'rt') as DMD:
            DMDlines = DMD.readlines()
            for iLine in range(len(DMDlines)):
                if re.search(r'vertexT_1', DMDlines[iLine]):
                    DMDlines[iLine]='vertexT_1  (' + str(round(TPolygon[2][0],6)) +'  0  0);\n'
                    DMDlines[iLine+1]='vertexT_2  (' + str(round(TPolygon[3][0],6)) + '  ' + str(round(TPolygon[3][1],6)) + '  0);\n'
                    DMDlines[iLine+2]='vertexT_3  (' + str(round(TPolygon[4][0],6)) + '  ' + str(round(TPolygon[4][1],6)) + '  0);\n'
                    DMDlines[iLine+3]='vertexT_4  (' + str(round(TPolygon[5][0],6)) + '  ' + str(round(TPolygon[5][1],6)) + '  0);\n'
                    DMDlines[iLine+4]='vertexT_5  (' + str(round(TPolygon[6][0],6)) +'  0  0);\n'
                    DMDlines[iLine+5]='vertexT_6  (' + str(round(TPolygon[7][0],6)) + '  ' + str(round(TPolygon[7][1],6)) + '  0);\n'

```

```
                    DMDlines[iLine+6]='vertexT_7  (' + str(round(TPolygon[8][0],6)) + '  ' + str(round(TPolygon[8][1],6)) + '  0);\n'
                    DMDlines[iLine+7]='vertexT_8  (' + str(round(TPolygon[9][0],6)) + '  ' + str(round(TPolygon[9][1],6)) + '  0);\n'

                if re.search(r'vectorL', DMDlines[iLine]):
                    DMDlines[iLine]='vectorL (' + wrVec + ');\n'

                if re.search(r'sectionL', DMDlines[iLine]):
                    DMDlines[iLine]='sectionL (' + wrSec + ');\n'

        with open(dst +'/constant/dynamicMeshDict', 'wt') as DMC:
            DMDlinesNew = ''.join(DMDlines)
            DMC.write(DMDlinesNew)
    return TPolygon, maxKappa
```

```
def LiveTiming(dst):
    flagRun = True
    flagState = False
    execTime = 0
    RunTime = [0]
    if os.path.exists(dst + '/log.overPimpleDyMFoam'):
        with open(dst + '/log.overPimpleDyMFoam', 'rt') as reconFile:
            reconLines = reconFile.readlines()[-300:]
        for lines in reconLines:
            if re.search(r'End', lines):
                for lines in reconLines:
                    if re.search(r'^ExecutionTime', lines):
                        execTime = (float(re.split(r'\s', lines)[2]))
                flagRun = False
                flagState = True
            elif re.search(r'Foam::error', lines) or re.search(r'\?:\?', lines) or re.search(r'exiting', lines):
                flagRun = False
                flagState = False
                execTime = 0
            elif re.search(r'^Time =', lines):
                RunTime.append(float(re.split(r'\s', lines)[-2]))
    return flagRun, flagState, execTime, RunTime[-1]
```

```python
def EvaluateCase(dst):
    #--- Read initialConditions -----------------------------------------------------------
    with open(dst + "/system/initialConditions", "r") as initialCond:
        initLines = initialCond.readlines()[8:]

    for iLine in range(0,len(initLines)):
        if re.search(r'^Period', initLines[iLine]):
            Period = float(re.split(r'\s', initLines[iLine])[-2].replace(';', ''))
        if re.search(r'^DeltaDeg', initLines[iLine]):
            DeltaDeg = float(re.split(r'\s', initLines[iLine])[-2].replace(';', ''))
        if re.search(r'^Origin000', initLines[iLine]):
            xShift = float(re.split(r'\s', initLines[iLine])[-4].replace('(', ''))
        if re.search(r'^Rho', initLines[iLine]):
            rho = float(re.split(r'\s', initLines[iLine])[-2].replace(';', ''))
        if re.search(r'^Blades', initLines[iLine]):
            nBlades = int(re.split(r'\s', initLines[iLine])[-2].replace(';', ''))
        if re.search(r'^RotorWidth', initLines[iLine]):
            RotorWidth = float(re.split(r'\s', initLines[iLine])[-2].replace(';', ''))
    DeltaT = Period / 360 * DeltaDeg
    Depth = 1
    Velocity = 3*math.pi / Period
    Surface = RotorWidth * Depth

    #--- Read Log-File -----------------------------------------------------------
    with open(dst + '/log.overPimpleDyMFoam', 'rt') as logFile:
        LogLines = logFile.readlines()[:]

    Time, TX, TY, phi, Alpha0 = [], [], [], [], []
    for iLine in range(0,len(LogLines)):
        if re.search(r'^Time =', LogLines[iLine]):
            Time.append(float(re.split(r'\s', LogLines[iLine])[-2]))
        if re.search(r'^Trajectory_0', LogLines[iLine]):
            TX.append(float(re.split(r'\s', LogLines[iLine])[1].replace('((', '')))
            TY.append(float(re.split(r'\s', LogLines[iLine])[2]))
            phi.append(float(re.split(r'\s', LogLines[iLine])[-2].replace('))', '')))
        if re.search(r'^Alpha0_0', LogLines[iLine]):
            Alpha0.append(float(re.split(r'\s', LogLines[iLine])[1]))

    #--- Extract Data from postProcessing -----------------------------------------------------------
    with open(dst + '/postProcessing/airfoilInner0/0/force.dat', 'rt') as forceFile:
        FLines = forceFile.readlines()[4:]

    time = [float(line.split()[0]) for line in FLines]
    LastTime = round(time[-1],8)
    nRot = math.floor(LastTime/Period)
    StartI = time.index(round(Period * (nRot - 1),8)) +4
    EndI = time.index(round(Period * nRot,8)) +5

    Fx, Fy, Mglobal = [], [], []
    for sides in ['Outer', 'Inner']:
        with open(dst + '/postProcessing/airfoil'+sides+'0/0/force.dat', 'rt') as forceFile:
            lines = forceFile.readlines()[StartI:EndI]
        Fx_tmp = [float(line.split()[1].replace('(','')) for line in lines]
        Fy_tmp = [float(line.split()[2]) for line in lines]
        if not Fy:
            Fx = Fx_tmp
            Fy = Fy_tmp


```

```
        else:
                        Fx = [x+y for x,y in zip(Fx_tmp,Fx)]
                        Fy = [x+y for x,y in zip(Fy_tmp,Fy)]
        with open(dst+'/postProcessing/airfoil'+sides+'0/0/moment.dat', 'rt') as momentFile:
            lines = momentFile.readlines()[StartI:EndI]
        Mglobal_tmp = [float(line.split()[3].replace(')','')) for line in lines]
        if not Mglobal:
            Mglobal = Mglobal_tmp
        else:
              Mglobal = [x+y for x,y in zip(Mglobal_tmp,Mglobal)]

    #--- Truncate: Log-Data   ----------------------------------------------------------
    Time = Time[StartI-4:EndI-4]
    TX = TX[StartI-4:EndI-4]
    TY = TY[StartI-4:EndI-4]
    phi = phi[StartI-4:EndI-4]
    Alpha0 = Alpha0[StartI-4:EndI-4]

    #--- Calculation ----------------------------------------------------------
    Ft, Psi, Fcorr, R, Mcorr, MAirfoil, theta = [], [], [], [], [], [], []
    for i in range(0,len(TX)):
        Ft.append(Fx[i]*math.sin(phi[i])-Fy[i]*math.cos(phi[i]))
        Psi.append(math.atan2(TY[i],xShift+TX[i]))
        Fcorr.append(-Fx[i]*math.sin(Psi[-1])+Fy[i]*math.cos(Psi[-1]))
        R.append(math.sqrt((TX[i]+xShift)**2+TY[i]**2))
        Mcorr.append(Fcorr[-1]*R[-1])
        MAirfoil.append(Mglobal[i]-Mcorr[-1])
        theta.append(phi[i] + Alpha0[i])

    OmegaAirfoil, PowerRot, PowerTra = [], [], []
    for i in range(1,len(Fx)):
        PowerTra.append(abs(0.5*(Ft[i]+Ft[i-1])*Velocity))
        DeltaTheta = theta[i]-theta[i-1]
        if DeltaTheta < - math.pi:
            DeltaTheta = DeltaTheta + 2*math.pi
        OmegaAirfoil.append(DeltaTheta/DeltaT)
        PowerRot.append(abs(0.5*(MAirfoil[i]+MAirfoil[i-1])*OmegaAirfoil[-1]))

    #--- Mean Values ----------------------------------------------------------
    tFixed  = np.linspace(Time[0], Time[-1], num=3600, endpoint=False)
    MeanFx = (nBlades*np.mean(np.interp(tFixed, Time, Fx)))
    MeanFy = (nBlades*np.mean(np.interp(tFixed, Time, Fy)))
    MeanThrust = math.sqrt(MeanFx**2+MeanFy**2)
    MeanPIdeal = math.sqrt(MeanThrust**3/(2*rho*Surface))
    tFixed  = np.linspace(Time[0], Time[-2], num=3600, endpoint=False)
    MeanPTra= (nBlades*np.mean(np.interp(tFixed, Time[:-1], PowerTra)))
    MeanProt = (nBlades*np.mean(np.interp(tFixed, Time[:-1], PowerRot)))
    MeanPReal = MeanPTra + MeanProt
    MeanFOM = MeanPIdeal / MeanPReal
    MeanA = math.degrees(math.atan2(MeanFy, MeanFx))

    return MeanFx, MeanFy, MeanThrust, MeanPIdeal, MeanPTra, MeanProt, MeanPReal, MeanFOM, MeanA, RotorWidth
```

```
def CleanCase(cwd, eval_id):
    dst = cwd + '/Run_' + str(eval_id).zfill(5)
    shutil.rmtree(dst, ignore_errors=True)
```

D OpenFOAM dicionaries

D.1 initialConditions

Global parameters and values for the OpenFOAM calculation are stored in an `initialConditions` file. There are different versions depending on the number of blade and the optimisation subject. The code shows an idle version, where ther stars '***' marks the run dependent entities. Examples for these entities are given for Opti-3TV and Opti-4P.

```
// ./system/controlDict
Period              12.1608;
DeltaDeg            0.5;
DeltaT              #eval{ $Period / 360.0 * $DeltaDeg };
EndTime             #eval{ *** * $Period + 4 * $DeltaT };
WriteInterval       ***;
PurgeWrite          1;

// ./constant/dynamicMeshDict
Origin***           ***
Axis                (0 0 1);
EndValue            #eval{ *** *pi() };
rotOmega            #eval{ 2*pi() / $Period };
Delay               ***;

// ./constant/transportProperties
Nu                  1.55e-05;

// Function Objects
RhoInf              1.225;
freeVelocity        #eval{ 1.5 * $rotOmega };
lengthAF            ***;

// Geometric Data
Blades             ***;
Offset***          ***;
RotorWidth         ***;

// ./0
flowVelocity         (0 0 0);
pressure             0;
turbulentKE          #eval{ 3.0/2* pow((0.001 * $freeVelocity),2) };
turbulentOmega       #eval{ pow($turbulentKE,0.5) / (pow(0.09,0.25) * $lengthAF ) };
turbulentVisco       0;
```

Opti-3TV

```
Origin000       (1.5 0 0);

X120            #eval{ 1.5*cos(120.0/360 *2*3.141592653589793) };
Y120            #eval{ 1.5*sin(120.0/360 *2*3.141592653589793) };
Origin120       ($X120 $Y120  0);

X240            #eval{ 1.5*cos(240.0/360 *2*3.141592653589793) };
Y240            #eval{ 1.5*sin(240.0/360 *2*3.141592653589793) };
Origin240       ($X240 $Y240  0);

Blades          3;
Offset0         0;
Offset120       #eval{ 1/3.0 };
Offset240       #eval{ 2/3.0 };
```

Opti-4P

```
Opti-4PV:
rotOrigin       (0 0 0);
pitOrigin0      ( 1.5     0  0);
pitOrigin090    (   0   1.5  0);
pitOrigin180    (-1.5     0  0);
pitOrigin270    (   0  -1.5  0);

Blades          4;
Offset0         0;
Offset090       0.25;
Offset180       0.5;
Offset270       0.75;
```

D.2 fvSchemes

Single-blade case

```
ddtSchemes
{
    default         backward;
}

gradSchemes
{
    default         cellLimited leastSquares 1;
    grad(U)         cellLimited leastSquares 1;
    grad(yPsi)      cellLimited Gauss linear 1;
    limitedGrad     cellLimited leastSquares 1;
}

divSchemes
{
    default             none;
    div(phi,U)          Gauss limitedLinearV 1;
    div(phi,k)          Gauss upwind;
    div(phi,omega)      Gauss upwind;

    div((nuEff*dev2(T(grad(U)))))   Gauss linear;
}

laplacianSchemes
{
      default           Gauss linear limited corrected 0.5;
      laplacian(yPsi)   Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default     limited corrected 0.5;
}

fluxRequired
{
    default         no;
    pcorr           ;
    p               ;
    yPsi            ;
}

wallDist
{
    method Poisson;
    correctWalls    true;
}
```

Overset method

The `fvSchemes` for the overset method is similar to the single blade case. Only the following code snipped has to be added.

```
oversetInterpolation
{
    method          inverseDistancePushFront;
    searchBox       (*** *** -1)(*** *** 1);
    voxelSize       0.008;
    nPushFront      1;
    layerRelax      0.5;
}

oversetInterpolationSuppressed
{

}
```

D.3 fvSchefvSolutionmes

Single-blade case

```
solvers
{

    cellDisplacement
    {
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
        maxIter             100;
    }

    p
    {
        solver              PBiCGStab;
            preconditioner      DIC;
            tolerance           1e-4;
            relTol              0.001;
            minIter             1;
        maxIter             2000;
    }

        pFinal
    {
            $p;
            tolerance           1e-5;
    }

```

```
    pcorr
    {
        $p
        preconditioner      FDIC;
        minIter             1;
        relTol              0.1;
        maxIter             100;
    }

    pcorrFinal
    {
        $pcorr
        tolerance           1e-4;
        minIter             1;
        maxIter             100;
    }

    yPsi
    {
        solver              PBiCGStab;
        preconditioner      DIC;
        tolerance           1e-10;
        relTol              0.0;
    }

    "(U|omega|k)"
    {
        solver             smoothSolver;
        smoother           symGaussSeidel;
        tolerance          1e-12;
            relTol             0.1;
            minIter            1;
    }

    "(U|omega|k)Final"
    {
        $U;
        tolerance           1e-12;
        relTol              0;
    }
}

PIMPLE
{
    ddtCorr                     true;
    correctPhi                  false;

    checkMeshCourantNo          yes;
    momentumPredictor           false;
    nOuterCorrectors            50;
    nCorrectors                 2;
    nNonOrthogonalCorrectors    2;
    turbOnFinalIterOnly         true;

    residualControl
    {
        U
        {
            relTol          0;
```

```
            tolerance       1e-7;
        }

        p
        {
            relTol          0;
            tolerance       1e-3;
        }
    }
}

relaxationFactors
{
    fields
    {
        p               0.3;
        pFinal          1.0;
    }
    equations
    {
        U                           0.95;
        pcorr                       0.95;
        "yWall|yPsi"                1.0;
        "(k|omega)"                 0.95;
        "(k|omega|U|pcorr)Final"    0.95;
        "(yWall|yPsi)Final"         1.0;
    }
}
```

Overset method

```
solvers
{

    cellDisplacement
    {
        solver              PCG;
        preconditioner      DIC;
        tolerance           1e-06;
        relTol              0;
        maxIter             100;
    }

    p
    {
        solver              PBiCGStab;
        preconditioner      DILU;
        tolerance           1e-4;
        relTol              0.001;
        minIter             1;
        maxIter             2000;
    }

    pFinal
    {
```

```
        $p;
        tolerance           1e-5;
    }

    pcorr
    {
        $p
        preconditioner      FDIC;
        minIter             1;
        relTol              0.1;
        maxIter             100;
    }

    pcorrFinal
    {
        $pcorr
        tolerance           1e-4;
        minIter             1;
        maxIter             100;
    }

    yPsi
    {
        solver              PBiCGStab;
        preconditioner      DILU;
        tolerance           1e-10;
        relTol              0.0;
    }

    "(U|omega|k)"
    {
        solver             smoothSolver;
        smoother           symGaussSeidel;
        tolerance          1e-12;
    relTol             0.1;
    minIter            1;
    }

    "(U|omega|k)Final"
    {
        $U;
        tolerance           1e-12;
        relTol              0;
    }
}

PIMPLE
{
    ddtCorr                     true;
    correctPhi                  false;
    oversetAdjustPhi            true;

    checkMeshCourantNo          yes;
    momentumPredictor           false;
    nOuterCorrectors            50;
    nCorrectors                 2;
    nNonOrthogonalCorrectors    2;
    turbOnFinalIterOnly         true;

```

```
    residualControl
    {
        U
        {
            relTol          0;
            tolerance       1e-7;
        }

        p
        {
            relTol          0;
            tolerance       1e-3;
        }
    }
}

relaxationFactors
{
    fields
    {
        p               0.05;
        pFinal          0.05;
    }
    equations
    {
        U                           0.95;
        pcorr                       0.95;
        "yWall|yPsi"                1.0;
        "(k|omega)"                 0.95;
        "(k|omega|U|pcorr)Final"    0.95;
        "(yWall|yPsi)Final"         1.0;
    }
}
```

D.4 dynamicMeshDict

Code passage for *only pitching* optimisation.

```
#include        "../system/initialConditions"

dynamicFvMesh       dynamicOversetFvMesh;
motionSolverLibs    ("libdynamicMesh.so");
solver              multiSolidBodyMotionSolver;

movingZone***
{
        solidBodyMotionFunction                 multiMotion;
        rotatingMotion
        {
                solidBodyMotionFunction          delayRotatingMotion;
                delayRotatingMotionCoeffs
                {
                        origin      $rotOrigin;
                        axis        $Axis;
```

```
                        omega       $rotOmega;
                        delay       $Delay;
                }
        }

        pitchingMotion
        {
                solidBodyMotionFunction          bSplinePitching;
                bSplinePitchingCoeffs
                {
                        origin      $pitOrigin***;
                        axis        $Axis;
                        period      $Period;
                        offset      $Offset***;
                        delay       $Delay;

                        vertexP_***   ***;
                }
        }
}
```

Code passage for *trajectory* and *both* optimisation (`bSplineMotion`).

```
#include        "../system/initialConditions"

dynamicFvMesh       dynamicOversetFvMesh;
motionSolverLibs     ("libdynamicMesh.so");
solver              multiSolidBodyMotionSolver;

movingZone***
{
    solidBodyMotionFunction         multiMotion;
    rotatingMotion
    {
        solidBodyMotionFunction      bSplineMotion;
        bSplineMotionCoeffs
        {
            origin       $Origin***;
            period       $Period;
            endValue     $EndValue;
            offset       $Offset***;
            delay        $Delay;

            vertexT_***  (***);

            vectorL      (***);
            sectionL     (***);
        }
    }

    pitchingMotion
    {
        solidBodyMotionFunction      bSplinePitching;
        bSplinePitchingCoeffs
        {
            origin        $Origin000;
            axis          $Axis;
```

```
            period        $Period;
            offset        $Offset0;
            delay         $Delay;

            vertexP_***   ***;
        }
    }
}
```

E OpenFOAM motion classes

E.1 bSplinePitching

bSplinePitching.H

```
#ifndef bSplinePitching_H
#define bSplinePitching_H

#include "solidBodyMotionFunction.H"
#include "primitiveFields.H"
#include "point.H"
#include "Function1.H"
#include "autoPtr.H"

namespace Foam
{
namespace solidBodyMotionFunctions
{

//  Class bSplinePitching Declaration

class bSplinePitching
:
    public solidBodyMotionFunction
{
        const vector origin_;
        const vector axis_;
        scalar period_;
        scalar offset_;
        scalar delay_;

        scalar vertexP_10_;
        scalar vertexP_11_;
        scalar vertexP_20_;
        scalar vertexP_21_;
        scalar vertexP_30_;
        scalar vertexP_31_;
        scalar vertexP_40_;
        scalar vertexP_41_;
        scalar vertexP_50_;
        scalar vertexP_51_;
        scalar vertexP_60_;
        scalar vertexP_61_;
        scalar vertexP_70_;
        scalar vertexP_71_;
        scalar vertexP_80_;
        scalar vertexP_81_;

        double bFactor_[4][4] {
            {-1.0/6, 1.0/2, -1.0/2, 1.0/6},
            {1.0/2, -1, 0, 2.0/3},
            {-1.0/2, 1.0/2, 1.0/2, 1.0/6},
            {1.0/6, 0, 0, 0}  };

        bSplinePitching(const bSplinePitching&) = delete;
        void operator=(const bSplinePitching&) = delete;

```

```
public:
    TypeName("bSplinePitching");
        bSplinePitching
        (   const dictionary& SBMFCoeffs,
            const Time& runTime
        );
        virtual autoPtr<solidBodyMotionFunction> clone() const
        {   return autoPtr<solidBodyMotionFunction>
            (   new bSplinePitching
                (   SBMFCoeffs_,
                    time_
                )
            );
        }
    virtual ~bSplinePitching() = default;
        virtual septernion transformation() const;
        virtual bool read(const dictionary& SBMFCoeffs);
};   }   }
#endif
```

bSplinePitching.C

```
#include "bSplinePitching.H"
#include "addToRunTimeSelectionTable.H"
#include "unitConversion.H"
#include "mathematicalConstants.H"

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
{
namespace solidBodyMotionFunctions
{   defineTypeNameAndDebug(bSplinePitching, 0);
    addToRunTimeSelectionTable
    (   solidBodyMotionFunction,
        bSplinePitching,
        dictionary    );   }   }
// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
Foam::solidBodyMotionFunctions::bSplinePitching::bSplinePitching
(
    const dictionary& SBMFCoeffs,
    const Time& runTime
)
:
    solidBodyMotionFunction(SBMFCoeffs, runTime),
    origin_(SBMFCoeffs_.get<vector>("origin")),
    axis_(SBMFCoeffs_.get<vector>("axis")),
    period_(SBMFCoeffs_.get<scalar>("period")),
    offset_(SBMFCoeffs_.get<scalar>("offset")),
    delay_(SBMFCoeffs_.get<scalar>("delay")),

    vertexP_10_(SBMFCoeffs_.get<scalar>("vertexP_10")),
    vertexP_11_(SBMFCoeffs_.get<scalar>("vertexP_11")),
    vertexP_20_(SBMFCoeffs_.get<scalar>("vertexP_20")),
    vertexP_21_(SBMFCoeffs_.get<scalar>("vertexP_21")),
    vertexP_30_(SBMFCoeffs_.get<scalar>("vertexP_30")),
    vertexP_31_(SBMFCoeffs_.get<scalar>("vertexP_31")),
    vertexP_40_(SBMFCoeffs_.get<scalar>("vertexP_40")),
```

```
    vertexP_41_(SBMFCoeffs_.get<scalar>("vertexP_41")),
    vertexP_50_(SBMFCoeffs_.get<scalar>("vertexP_50")),
    vertexP_51_(SBMFCoeffs_.get<scalar>("vertexP_51")),
    vertexP_60_(SBMFCoeffs_.get<scalar>("vertexP_60")),
    vertexP_61_(SBMFCoeffs_.get<scalar>("vertexP_61")),
    vertexP_70_(SBMFCoeffs_.get<scalar>("vertexP_70")),
    vertexP_71_(SBMFCoeffs_.get<scalar>("vertexP_71")),
    vertexP_80_(SBMFCoeffs_.get<scalar>("vertexP_80")),
    vertexP_81_(SBMFCoeffs_.get<scalar>("vertexP_81"))
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //
Foam::septernion
Foam::solidBodyMotionFunctions::bSplinePitching::transformation() const
{
    scalar vertexP_00_={vertexP_81_};
    scalar vertexP_90_={vertexP_10_};
    scalar vertexP_91_={vertexP_11_};

    scalar Polygon_[19]
    {   vertexP_00_,
        vertexP_10_,
        vertexP_11_,
        vertexP_20_,
        vertexP_21_,
        vertexP_30_,
        vertexP_31_,
        vertexP_40_,
        vertexP_41_,
        vertexP_50_,
        vertexP_51_,
        vertexP_60_,
        vertexP_61_,
        vertexP_70_,
        vertexP_71_,
        vertexP_80_,
        vertexP_81_,
        vertexP_90_,
        vertexP_91_,   };

    scalar pi = Foam::constant::mathematical::pi;
    scalar lag_ = 1;
    scalar duration_ = delay_ * period_;
    if (time_.value()  < duration_ ) {  lag_ = (1 - cos(time_.value() / duration_ * pi ) ) / 2;  }

    scalar t_ = time_.value() + offset_ * period_;
    scalar nRot_ = floor(t_ / period_);
    int k = floor(( t_ - period_ * nRot_ ) / period_ * 16);
    scalar nT_=  16 * t_ / period_ - k - nRot_ * 16;

    scalar R1_ =  (bFactor_[0][0]*pow(nT_,3)+bFactor_[0][1]*pow(nT_,2)+bFactor_[0][2]*nT_+bFactor_[0][3]);
    scalar R2_ =  (bFactor_[1][0]*pow(nT_,3)+bFactor_[1][1]*pow(nT_,2)+bFactor_[1][2]*nT_+bFactor_[1][3]);
    scalar R3_ =  (bFactor_[2][0]*pow(nT_,3)+bFactor_[2][1]*pow(nT_,2)+bFactor_[2][2]*nT_+bFactor_[2][3]);
    scalar R4_ =  (bFactor_[3][0]*pow(nT_,3)+bFactor_[3][1]*pow(nT_,2)+bFactor_[3][2]*nT_+bFactor_[3][3]);

    scalar angle_ = - ( R1_*Polygon_[k]+R2_*Polygon_[k+1]+R3_*Polygon_[k+2]+R4_*Polygon_[k+3] ) * pi / 180.0 *
    ↪  lag_;

    quaternion R(axis_, angle_);
    scalar ADD = 360.0 * offset_;
```

```
    septernion TR(septernion(-origin_)*R*septernion(origin_));
    DebugInFunction << "Time = " << t_ << " transformation: " << TR << endl;

    return TR;
}

bool Foam::solidBodyMotionFunctions::bSplinePitching::read
(     const dictionary& SBMFCoeffs     )
{     solidBodyMotionFunction::read(SBMFCoeffs);
      return true;     }
```

E.2 bSplineMotion

bSplineMotion.H

```
#ifndef bSplineMotion_H
#define bSplineMotion_H

#include "solidBodyMotionFunction.H"
#include "primitiveFields.H"
#include "point.H"
#include "Function1.H"
#include "autoPtr.H"
#include "List.H"

namespace Foam
{
namespace solidBodyMotionFunctions
{
//  Class bSplineMotion Declaration
class bSplineMotion
:
    public solidBodyMotionFunction
{
        const vector origin_;
        scalar period_;
        scalar endValue_;
        scalar offset_;
        scalar delay_;

        vector vertexT_1_;
        vector vertexT_2_;
        vector vertexT_3_;
        vector vertexT_4_;
        vector vertexT_5_;
        vector vertexT_6_;
        vector vertexT_7_;
        vector vertexT_8_;

        scalarList vectorL_;
        scalarList sectionL_;

        double bFactor_[5][5] {
            {1.0/24, -1.0/6, 1.0/4, -1.0/6, 1.0/24},
```

```
            {-1.0/6, 1.0/2, -1.0/4, -1.0/2, 11.0/24},
            {1.0/4, -1.0/2, -1.0/4, 1.0/2, 11.0/24},
            {-1.0/6, 1.0/6, 1.0/4, 1.0/6, 1.0/24},
            {1.0/24, 0, 0, 0, 0}  };

        double dFactor_[5][4] {
            {1.0/6, -1.0/2, 1.0/2, -1.0/6},
            {-4.0/6, 3.0/2, -1.0/2, -1.0/2},
            {1, -3.0/2, -1.0/2, 1.0/2},
            {-4.0/6, 1.0/2, 1.0/2, 1.0/6},
            {1.0/6, 0, 0, 0}  };

        double Length_[5][5][5]{
            {   {1, -4, 6, -4, 1},
                {0, 0, 0, 0, 0,},
                {0, 0, 0, 0, 0,},
                {0, 0, 0, 0, 0,},
                {0, 0, 0, 0, 0,},  },

            {   {-15.0/8, 7, -9, 4, 0},
                {1.0/8, -1.0/2, 3.0/4, -1.0/2, 1.0/8},
                {0, 0, 0, 0, 0,},
                {0, 0, 0, 0, 0,},
                {0, 0, 0, 0, 0,},  },

            {   {85.0/72, -11.0/3, 3, 0, 0},
                {-23.0/72, 19.0/18, -11.0/12, -5.0/18, 37.0/72},
                {1.0/18, -2.0/9, 1.0/3, -2.0/9, 1.0/18},
                {0, 0, 0, 0, 0,},
                {0, 0, 0, 0, 0,},  },

            {   {-25.0/72, 2.0/3, 0, 0, 0},
                { 23.0/72, -13.0/18,  -1.0/12, 11.0/18, 23.0/72},
                {-13.0/72, 5.0/9, -1.0/3, -4.0/9, 4.0/9},
                {1.0/24, -1.0/6, 1.0/4, -1.0/6, 1.0/24},
                {0, 0, 0, 0, 0,},  },

            {   {1.0/24, 0, 0, 0, 0,},
                {-1.0/6, 1.0/6, 1.0/4, 1.0/6, 1.0/24},
                {1.0/4, -1.0/2, -1.0/4, 1.0/2, 11.0/24},
                {-1.0/6, 1.0/2, -1.0/4, -1.0/2, 11.0/24},
                {1.0/24, -1.0/6, 1.0/4, -1.0/6, 1.0/24},  },  };

        double Index_[2][9][5]{
            {   {0, 1, 2, 3, 4},
                {1, 2, 3, 4, 4},
                {2, 3, 4, 4, 4},
                {3, 4, 4, 4, 4},
                {4, 4, 4, 4, 4},
                {4, 4, 4, 4, 3},
                {4, 4, 4, 3, 2},
                {4, 4, 3, 2, 1},
                {4, 3, 2, 1, 0},  },

            {   {0, 0, 0, 0, 0},
                {1, 1, 1, 1, 0},
                {2, 2, 2, 1, 0},
                {3, 3, 2, 1, 0},
                {4, 3, 2, 1, 0},
```

```
                {4, 3, 2, 1, 3},
                {4, 3, 2, 2, 2},
                {4, 3, 1, 1, 1},
                {4, 0, 0, 0, 0},  }, };

        bSplineMotion(const bSplineMotion&) = delete;
        void operator=(const bSplineMotion&) = delete;
public:
    TypeName("bSplineMotion");
        bSplineMotion
        (   const dictionary& SBMFCoeffs,
            const Time& runTime
        );
        virtual autoPtr<solidBodyMotionFunction> clone() const
        {   return autoPtr<solidBodyMotionFunction>
            (   new bSplineMotion
                (   SBMFCoeffs_,
                    time_
                )
            );
        }
    virtual ~bSplineMotion() = default;
        virtual septernion transformation() const;
        virtual bool read(const dictionary& SBMFCoeffs);
};   }   }
#endif
```

bSplineMotion.C

```
#include "bSplineMotion.H"
#include "addToRunTimeSelectionTable.H"
#include "unitConversion.H"
#include "mathematicalConstants.H"

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
{
namespace solidBodyMotionFunctions
{   defineTypeNameAndDebug(bSplineMotion, 0);
    addToRunTimeSelectionTable
    (   solidBodyMotionFunction,
        bSplineMotion,
        dictionary  );  }  }

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
Foam::solidBodyMotionFunctions::bSplineMotion::bSplineMotion
(   const dictionary& SBMFCoeffs,
    const Time& runTime   )
:
    solidBodyMotionFunction(SBMFCoeffs, runTime),
    origin_(SBMFCoeffs_.get<vector>("origin")),
    period_(SBMFCoeffs_.get<scalar>("period")),
    endValue_(SBMFCoeffs_.get<scalar>("endValue")),
    offset_(SBMFCoeffs_.get<scalar>("offset")),
    delay_(SBMFCoeffs_.get<scalar>("delay")),

    vertexT_1_(SBMFCoeffs_.get<vector>("vertexT_1")),
```

```
    vertexT_2_(SBMFCoeffs_.get<vector>("vertexT_2")),
    vertexT_3_(SBMFCoeffs_.get<vector>("vertexT_3")),
    vertexT_4_(SBMFCoeffs_.get<vector>("vertexT_4")),
    vertexT_5_(SBMFCoeffs_.get<vector>("vertexT_5")),
    vertexT_6_(SBMFCoeffs_.get<vector>("vertexT_6")),
    vertexT_7_(SBMFCoeffs_.get<vector>("vertexT_7")),
    vertexT_8_(SBMFCoeffs_.get<vector>("vertexT_8")),

    vectorL_(SBMFCoeffs_.get<scalarList>("vectorL")),
    sectionL_(SBMFCoeffs_.get<scalarList>("sectionL"))
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //
Foam::septernion
Foam::solidBodyMotionFunctions::bSplineMotion::transformation() const
{
    scalar t_ = time_.value() + offset_ * period_;
    scalar pi = Foam::constant::mathematical::pi;
    scalar lag_ = 1;
    scalar duration_ = delay_ * period_;
    if (time_.value()  < duration_ ){
            lag_ = (1 - cos(time_.value() / duration_ * pi ) ) / 2;    }

    vector Polygon_[13]{
            vertexT_7_,
            vertexT_8_,
            vertexT_1_,
            vertexT_2_,
            vertexT_3_,
            vertexT_4_,
            vertexT_5_,
            vertexT_6_,
            vertexT_7_,
            vertexT_8_,
            vertexT_1_,
            vertexT_2_,
            vertexT_3_    };

    scalar nRot_ = floor(t_ / period_);
    scalar Lreq = (t_ - period_ * nRot_ ) / period_ * endValue_;

    int m = 1;
    bool flagSec = false;
    while (flagSec == false) {
        if (Lreq < sectionL_[m])  {  flagSec = true;  }
        else{  m++;  }   }

    double matR_[5][5];
    int indA_ = 0;
    int indB_ = 0;
    int indC_ = 0;

    if (m < 4) {  indA_ = m;  }
    else if (m < 17) {  ndA_ = 4;  }
    else{  indA_ = m - 12;  }

    for (int i = 0; i < 5; i++) {
        indB_ = Index_[0][indA_][i];
        indC_ = Index_[1][indA_][i];

```

```
        for (int j = 0; j < 5; j++) {  matR_[i][j]=Length_[indB_][indC_][j];  }   }

    scalar nL_ = 1.75 + (t_ - period_ * nRot_ ) / period_ * 17.5;
    nL_ = nL_ - floor(nL_);
    scalar tNew = nL_;
    scalar L = 0;
    scalar dL = 0;
    scalar diffT = 1;
    int sign = 1;

    while (diffT > 1e-5) {
        scalar LR1 = matR_[0][0]*pow(nL_,4) + matR_[0][1]*pow(nL_,3) + matR_[0][2]*pow(nL_,2) + matR_[0][3]*nL_ + matR_[0][4];
        scalar LdR1 = 4* matR_[0][0]*pow(nL_,3) + 3* matR_[0][1]*pow(nL_,2) + 2* matR_[0][2]*nL_ + matR_[0][3];

        scalar LR2 = matR_[1][0]*pow(nL_,4) + matR_[1][1]*pow(nL_,3) + matR_[1][2]*pow(nL_,2) + matR_[1][3]*nL_ + matR_[1][4];
        scalar LdR2 = 4* matR_[1][0]*pow(nL_,3) + 3* matR_[1][1]*pow(nL_,2) + 2* matR_[1][2]*nL_ + matR_[1][3];

        if (m >= 19) { nL_ = 1 - tNew; sign = -1; }
        scalar LR3 = matR_[2][0]*pow(nL_,4) + matR_[2][1]*pow(nL_,3) + matR_[2][2]*pow(nL_,2) + matR_[2][3]*nL_ + matR_[2][4];
        scalar LdR3 = sign * ( 4* matR_[2][0]*pow(nL_,3) + 3* matR_[2][1]*pow(nL_,2) + 2* matR_[2][2]*nL_ + matR_[2][3] );

        if (m >= 18) { nL_ = 1 - tNew; sign = -1; }
        scalar LR4 = matR_[3][0]*pow(nL_,4) + matR_[3][1]*pow(nL_,3) + matR_[3][2]*pow(nL_,2) + matR_[3][3]*nL_ + matR_[3][4];
        scalar LdR4 = sign * ( 4* matR_[3][0]*pow(nL_,3) + 3* matR_[3][1]*pow(nL_,2) + 2* matR_[3][2]*nL_ + matR_[3][3] );

        if (m >= 17) { nL_ = 1 - tNew; sign = -1; }
        scalar LR5 = matR_[4][0]*pow(nL_,4) + matR_[4][1]*pow(nL_,3) + matR_[4][2]*pow(nL_,2) + matR_[4][3]*nL_ + matR_[4][4];
        scalar LdR5 = sign * ( 4* matR_[4][0]*pow(nL_,3) + 3* matR_[4][1]*pow(nL_,2) + 2* matR_[4][2]*nL_ + matR_[4][3] );

        L = LR1*vectorL_[m]+LR2*vectorL_[m+1]+LR3*vectorL_[m+2]+LR4*vectorL_[m+3]+LR5*vectorL_[m+4] - Lreq; // function for Length-over-nT minus Lreq, vertical transformation
        dL = LdR1*vectorL_[m]+LdR2*vectorL_[m+1]+LdR3*vectorL_[m+2]+LdR4*vectorL_[m+3]+LdR5*vectorL_[m+4];

        if (m >= 17) { nL_ =  tNew; sign = 1; }

        tNew = nL_ - L / dL;
        diffT = fabs(tNew - nL_);
        nL_ = tNew;  }

    nL_ = 0.5 + (m + nL_ - 1.75)  * 8.0 / 17.5;

    if (nL_ < 0.5) {  nL_ = 0.5;  }
    else if (nL_ > 8.5) {  nL_ = 8.5;  }

    int k = floor(nL_);
    scalar nT_ = nL_  - k;

    vector displacement;
    displacement.x() = 0;
    displacement.y() = 0;
    displacement.z() = 0;
```

```
    vector rotation;
    rotation.x() = 0;
    rotation.y() = 0;
    rotation.z() = 0;

    scalar R1_ = bFactor_[0][0]*pow(nT_,4) + bFactor_[0][1]*pow(nT_,3) + bFactor_[0][2]*pow(nT_,2) + bFactor_[0][3]*nT_ + bFactor_[0][4];  // N_1.5
    scalar R2_ = bFactor_[1][0]*pow(nT_,4) + bFactor_[1][1]*pow(nT_,3) + bFactor_[1][2]*pow(nT_,2) + bFactor_[1][3]*nT_ + bFactor_[1][4];  // N_2.5
    scalar R3_ = bFactor_[2][0]*pow(nT_,4) + bFactor_[2][1]*pow(nT_,3) + bFactor_[2][2]*pow(nT_,2) + bFactor_[2][3]*nT_ + bFactor_[2][4];  // N_3.5
    scalar R4_ = bFactor_[3][0]*pow(nT_,4) + bFactor_[3][1]*pow(nT_,3) + bFactor_[3][2]*pow(nT_,2) + bFactor_[3][3]*nT_ + bFactor_[3][4];  // N_4.5
    scalar R5_ = bFactor_[4][0]*pow(nT_,4) + bFactor_[4][1]*pow(nT_,3) + bFactor_[4][2]*pow(nT_,2) + bFactor_[4][3]*nT_ + bFactor_[4][4];  // N_5.5

    displacement.x() = ( ( R1_*Polygon_[k][0] + R2_*Polygon_[k+1][0] + R3_*Polygon_[k+2][0] + R4_*Polygon_[k+3][0] + R5_*Polygon_[k+4][0] ) - origin_.x() ) * lag_;
    displacement.y() = ( ( R1_*Polygon_[k][1] + R2_*Polygon_[k+1][1] + R3_*Polygon_[k+2][1] + R4_*Polygon_[k+3][1] + R5_*Polygon_[k+4][1] ) - origin_.y() ) * lag_;

    scalar dR1_ = dFactor_[0][0]*pow(nT_,3) + dFactor_[0][1]*pow(nT_,2) + dFactor_[0][2]*nT_ + dFactor_[0][3]; // N'_1.5
    scalar dR2_ = dFactor_[1][0]*pow(nT_,3) + dFactor_[1][1]*pow(nT_,2) + dFactor_[1][2]*nT_ + dFactor_[1][3]; // N'_2.5
    scalar dR3_ = dFactor_[2][0]*pow(nT_,3) + dFactor_[2][1]*pow(nT_,2) + dFactor_[2][2]*nT_ + dFactor_[2][3]; // N'_3.5
    scalar dR4_ = dFactor_[3][0]*pow(nT_,3) + dFactor_[3][1]*pow(nT_,2) + dFactor_[3][2]*nT_ + dFactor_[3][3]; // N'_4.5
    scalar dR5_ = dFactor_[4][0]*pow(nT_,3) + dFactor_[4][1]*pow(nT_,2) + dFactor_[4][2]*nT_ + dFactor_[4][3]; // N'_5.5

    scalar tangentX_ = (dR1_*Polygon_[k][0] + dR2_*Polygon_[k+1][0] + dR3_*Polygon_[k+2][0] + dR4_*Polygon_[k+3][0] + dR5_*Polygon_[k+4][0]);
    scalar tangentY_ = (dR1_*Polygon_[k][1] + dR2_*Polygon_[k+1][1] + dR3_*Polygon_[k+2][1] + dR4_*Polygon_[k+3][1] + dR5_*Polygon_[k+4][1]);

    scalar phi = - atan2(tangentX_, tangentY_);

    scalar multiplier_ = 1;
    if (time_.value() < 0.25*period_ && t_ < 0.25*period_) {  multiplier_ = 0;  }

    if (phi < 0) {  phi = phi + 2.0*pi *multiplier_ ;  }

    scalar Revolution = 0;
    if (phi > 0 && nL_ > 7.0) {  Revolution = (nRot_ + 1) * 2*pi;  }

    else if (nRot_ > 0) {
        if (phi < 0 && nL_ < 2.0) {  Revolution = (nRot_ -1) * 2*pi;  }
        else{  Revolution = nRot_ * 2*pi;  }  }

    scalar AngleOffset_ =  2.0*pi *offset_;
    rotation.z() =  ( phi - AngleOffset_ + Revolution )  *lag_;

    Vector2D<vector> TRV(displacement,rotation);
    quaternion R(quaternion::XYZ, TRV[1]);
    septernion TR(septernion(-origin_ + -TRV[0])*R*septernion(origin_));
```

```
    scalar ADD = 360.0 * offset_;
    Info << "\n\nTrajectory_" << ADD;
    Info << ": " << TRV;
    Info << "\n";

    DebugInFunction << "Time = " << t_ << " transformation: " << TR << endl;

    return TR;
}
bool Foam::solidBodyMotionFunctions::bSplineMotion::read
(    const dictionary& SBMFCoeffs    )
{    solidBodyMotionFunction::read(SBMFCoeffs);
     return true;    }
```

E.3 delayRotatingMotion

delayRotatingMotion.C

```
#include "delayRotatingMotion.H"
#include "addToRunTimeSelectionTable.H"
#include "mathematicalConstants.H"
using namespace Foam::constant::mathematical;
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
{
namespace solidBodyMotionFunctions
{   defineTypeNameAndDebug(delayRotatingMotion, 0);
    addToRunTimeSelectionTable
    (   solidBodyMotionFunction,
        delayRotatingMotion,
        dictionary
    );
}
}
// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //
Foam::solidBodyMotionFunctions::delayRotatingMotion::delayRotatingMotion
(   const dictionary& SBMFCoeffs,
    const Time& runTime
)
:
    solidBodyMotionFunction(SBMFCoeffs, runTime),
    origin_(SBMFCoeffs_.lookup("origin")),
    axis_(SBMFCoeffs_.lookup("axis")),
    omega_(SBMFCoeffs_.get<scalar>("omega")),
    delay_(SBMFCoeffs_.get<scalar>("delay"))
{}
// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //
Foam::solidBodyMotionFunctions::delayRotatingMotion::~delayRotatingMotion()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //
Foam::septernion
Foam::solidBodyMotionFunctions::delayRotatingMotion::transformation() const
{
    scalar t = time_.value();
```

```
    scalar pi = Foam::constant::mathematical::pi;
    scalar period_ = abs(2.0*pi / omega_);
    scalar lag_ = 1;
    scalar duration_ = delay_ * period_;

    if (time_.value()  < duration_ ) {  lag_ = (1 - cos(time_.value() / duration_ * pi ) ) / 2;  }

    scalar angle = omega_ * t * lag_;
    quaternion R(axis_, angle);
    septernion TR(septernion(-origin_)*R*septernion(origin_));
    DebugInFunction << "Time = " << t << " transformation: " << TR << endl;
    return TR;
}
bool Foam::solidBodyMotionFunctions::delayRotatingMotion::read
(   const dictionary& SBMFCoeffs   )
{   solidBodyMotionFunction::read(SBMFCoeffs);
    return true;
}
```

F Control variables

Opti-1P

```
vertexP_10    24.9701;
vertexP_11    25.0398;
vertexP_20    35.0201;
vertexP_21    34.4016;
vertexP_30    32.6997;
vertexP_31    38.9346;
vertexP_40    23.5338;
vertexP_41     8.6503;
vertexP_50   -15.0033;
vertexP_51     0.3251;
vertexP_60   -20.7334;
vertexP_61   -33.6056;
vertexP_70   -25.9749;
vertexP_71   -12.7083;
vertexP_80    -3.8079;
vertexP_81     7.0438;
```

Opti-1TV

```
vertexT_1   (2.04113  0 0);
vertexT_2   (1.411143  0.662868  0);
vertexT_3   (0.078492  1.564281  0);
vertexT_4   (-1.725601  1.509977  0);
vertexT_5   (-2.097415  0  0);
vertexT_6   (-0.827778  -0.29126  0);
vertexT_7   (-1.00433  -1.04355  0);
vertexT_8   (1.220882  -1.188577  0);

vectorL     ( -0.909913  -0.748294  -0.455527  -0.0785  0.287445  0.720053  1.299275  1.979206  2.705847
↪  3.42104  4.021876  4.564918  5.017585  5.449909  5.881845  6.197045  6.519524  7.039939  7.829829  8.545227
↪  9.113692  9.539089  9.81293  10.049961  10.174909  );
sectionL    ( -0.340727  0.101552  0.512635  1.019969  1.645383  2.343996  3.058202  3.714285  4.287223
↪  4.786638  5.232883  5.660996  6.034884  6.366835  6.799207  7.443008  8.178302  8.817376  9.317918  9.725947
↪  );
```

Opti-1BV

```
vertexT_1   (1.64108  0  0);
vertexT_2   (1.164783  1.128253  0);
vertexT_3   (-0.258111  1.793746  0);
vertexT_4   (-1.267824  0.948777  0);
vertexT_5   (-1.607378  0  0);
vertexT_6   (-1.388869  -1.123224  0);
vertexT_7   (0.147749  -1.533886  0);
```

```
vertexT_8   (1.44739  -1.446416  0);

vectorL     ( -0.96455  -0.839488  -0.578656  -0.142281  0.431718  0.979933  1.566632  2.17377  2.711307
↪  3.233466  3.751844  4.225202  4.685476  5.156814  5.625421  6.17141  6.821837  7.459305  7.991869  8.440703
↪  8.985855  9.5748  9.991583  10.27026  10.41251  );
sectionL    ( -0.429763  0.143318  0.706355  1.275738  1.868152  2.438998  2.971588  3.490622  3.986102
↪  4.455254  4.921492  5.394228  5.905992  6.500435  7.13566  7.717727  8.216811  8.719117  9.280768  9.851373
↪  );

vertexP_10    12.7028;
vertexP_11    18.8694;
vertexP_20    31.2397;
vertexP_21    24.3609;
vertexP_30    29.9015;
vertexP_31    40.4845;
vertexP_40    33.7019;
vertexP_41    37.1604;
vertexP_50    17.0702;
vertexP_51   -13.738;
vertexP_60     4.2969;
vertexP_61   -38.403;
vertexP_70   -24.2948;
vertexP_71   -40.6542;
vertexP_80    -6.3308;
vertexP_81     1.5062;
```

Opti-2P

```
vertexP_10    24.9701;
vertexP_11    35.1794;
vertexP_20    35.0201;
vertexP_21    36.1708;
vertexP_30    31.6316;
vertexP_31    45.493;
vertexP_40    38.1008;
vertexP_41    15.8147;
vertexP_50   -11.0191;
vertexP_51     4.6285;
vertexP_60   -54.9305;
vertexP_61   -42.408;
vertexP_70   -26.3058;
vertexP_71   -21.6285;
vertexP_80     2.342;
vertexP_81     4.3762;
```

Opti-2Px2

```
vertexP_10   24.9701;
vertexP_11   35.1921;
vertexP_20   35.0201;
```

```
vertexP_21   36.1708;
vertexP_30   31.9729;
vertexP_31   39.7395;
vertexP_40   43.5055;
vertexP_41   16.343;
vertexP_50  -13.2354;
vertexP_51   -3.8166;
vertexP_60  -53.2726;
vertexP_61  -39.2979;
vertexP_70  -23.405;
vertexP_71  -20.5116;
vertexP_80    1.2234;
vertexP_81   24.2237;
```

Opti-2Px4

```
vertexP_10   24.9701;
vertexP_11   35.2529;
vertexP_20   32.88;
vertexP_21   40.5244;
vertexP_30   36.0121;
vertexP_31   32.1839;
vertexP_40   43.0415;
vertexP_41   30.8545;
vertexP_50    8.5162;
vertexP_51  -10.0721;
vertexP_60   -4.4668;
vertexP_61  -55.1432;
vertexP_70  -40.0833;
vertexP_71  -31.4176;
vertexP_80  -14.3473;
vertexP_81   -7.9696;
```

Opti-2TV

```
vertexT_1   (1.379668  0  0);
vertexT_2   (1.149268  0.671372  0);
vertexT_3   (0.1951  1.542619  0);
vertexT_4   (-1.399039  1.648877  0);
vertexT_5   (-1.774082  0  0);
vertexT_6   (-1.382681  -1.725867  0);
vertexT_7   (-0.814265  -1.681543  0);
vertexT_8   (1.194757  -0.705831  0);

vectorL     ( -0.669399  -0.539374  -0.323204  -0.07584  0.233689  0.583943  1.030593  1.572218  2.177153
↪  2.790469  3.31758  3.917298  4.656122  5.405536  6.020344  6.347853  6.685704  7.363254  8.165009  8.809018
↪  9.192846  9.500941  9.745299  9.927955  10.028649  );
sectionL    ( -0.248002  0.079776  0.414529  0.815241  1.308001  1.877673  2.480568  3.053458  3.626261
↪  4.292947  5.025661  5.69536  6.172558  6.531363  7.043808  7.762734  8.4696  8.986936  9.344476  9.667053
↪  );
```

Opti-2BV

```
vertexT_1   (1.420654  0  0);
vertexT_2   (1.400768  0.291499  0);
vertexT_3   (1.044336  1.698616  0);
vertexT_4   (-1.012331  1.527856  0);
vertexT_5   (-0.740926  0  0);
vertexT_6   (-1.086042  -1.781508  0);
vertexT_7   (0.42471  -1.732908  0);
vertexT_8   (1.499438  -1.966419  0);

vectorL     ( -1.006005  -0.875581  -0.580656  -0.090649  0.361451  0.660459  1.070181  1.593903  2.153323
↪  2.838927  3.384183  3.923846  4.644766  5.397427  5.984452  6.391218  6.978169  7.556226  7.983592  8.359428
↪  9.024225  9.587715  9.856189  10.014751  10.10762  );
sectionL    ( -0.421737  0.120637  0.509189  0.874683  1.33828  1.880358  2.495535  3.105474  3.661334  4.293181
↪  5.015518  5.676527  6.187832  6.69183  7.260548  7.761483  8.181403  8.699645  9.293185  9.755582  );

vertexP_10    0.9254;
vertexP_11   27.3619;
vertexP_20   40.8156;
vertexP_21   49.3131;
vertexP_30   48.2269;
vertexP_31   40.4845;
vertexP_40   47.1376;
vertexP_41   34.0007;
vertexP_50    0.2285;
vertexP_51  -23.4633;
vertexP_60  -51.3855;
vertexP_61  -64.6857;
vertexP_70  -59.4925;
vertexP_71  -49.0692;
vertexP_80   -6.3308;
vertexP_81   27.1419;
```

Opti-3P

```
vertexP_10   21.7546;
vertexP_11   35.1921;
vertexP_20   38.3487;
vertexP_21   44.8839;
vertexP_30   42.4487;
vertexP_31   42.6875;
vertexP_40   52.0479;
vertexP_41   41.8723;
vertexP_50   19.1539;
vertexP_51   -2.6547;
vertexP_60   -0.63;
vertexP_61  -50.9719;
vertexP_70  -45.54;
vertexP_71  -33.3296;
```

```
vertexP_80  -22.4129;
vertexP_81    6.211;
```

Opti-3TV

```
vertexT_1   (1.01123  0  0);
vertexT_2   (1.364221  1.56872  0);
vertexT_3   (-0.237299  2.135153  0);
vertexT_4   (-1.058026  1.14231  0);
vertexT_5   (-0.403801  0  0);
vertexT_6   (-0.771681  -2.208536  0);
vertexT_7   (0.657202  -2.017376  0);
vertexT_8   (0.788637  -0.885882  0);

vectorL     ( -0.886304  -0.769395  -0.532257  -0.160631  0.439791  1.032496  1.551331  2.137403  2.676154
↪  3.149649  3.602139  4.144286  4.829633  5.6471  6.35806  6.737344  7.170279  7.608127  8.067062  8.540076
↪  9.004454  9.5542  10.016889  10.303246  10.442499  );
sectionL    ( -0.402683  0.14363  0.732744  1.291637  1.845197  2.402088  2.909307  3.378755  3.882915  4.498431
↪  5.239434  5.984322  6.536117  6.956251  7.390286  7.83906  8.303796  8.775462  9.285683  9.862695  );
```

Opti-3BV

```
vertexT_1   (1.379027  0  0);
vertexT_2   (1.28184  1.561356  0);
vertexT_3   (-0.479927  1.168758  0);
vertexT_4   (-1.409107  1.436631  0);
vertexT_5   (-1.762727  0  0);
vertexT_6   (-1.099568  -1.227981  0);
vertexT_7   (-0.356301  -1.765628  0);
vertexT_8   (0.651136  -1.748194  0);

vectorL     ( -1.250868  -1.098322  -0.748601  -0.185399  0.566909  1.104601  1.553474  2.203879  2.831937
↪  3.307407  3.649391  4.128831  4.722861  5.317601  5.879464  6.363501  6.781599  7.190536  7.604564  8.13421
↪  8.856372  9.627753  10.12574  10.387632  10.512064  );
sectionL    ( -0.557591  0.18187  0.823112  1.333734  1.886142  2.510619  3.057752  3.478564  3.899613  4.43065
↪  5.018891  5.59392  6.115492  6.569421  6.985898  7.40258  7.882226  8.505363  9.239638  9.946507  );

vertexP_10     8.9957;
vertexP_11    18.0512;
vertexP_20    46.916;
vertexP_21    54.3312;
vertexP_30    25.4048;
vertexP_31    49.8925;
vertexP_40    49.7479;
vertexP_41    55.0296;
vertexP_50    17.4586;
vertexP_51   -21.6402;
vertexP_60   -15.2348;
vertexP_61   -71.8715;
vertexP_70   -64.8478;
```

```
vertexP_71   -34.1564;
vertexP_80   -13.1204;
vertexP_81   -13.7782;
```

Opti-4P

```
vertexP_10   22.7788;
vertexP_11   43.8552;
vertexP_20   48.79;
vertexP_21   44.8839;
vertexP_30   36.9395;
vertexP_31   51.2835;
vertexP_40   51.9625;
vertexP_41   30.8545;
vertexP_50   41.3201;
vertexP_51    4.123;
vertexP_60   11.3296;
vertexP_61  -76.4937;
vertexP_70  -68.8433;
vertexP_71  -54.9809;
vertexP_80  -27.2127;
vertexP_81  -24.3432;
```

Opti-4TV

```
vertexT_1   (0.977396  0  0);
vertexT_2   (0.926911  0.595098  0);
vertexT_3   (-0.092456  2.163341  0);
vertexT_4   (-1.587724  1.420676  0);
vertexT_5   (-0.610922  0  0);
vertexT_6   (-0.77071  -2.185893  0);
vertexT_7   (0.827756  -2.167554  0);
vertexT_8   (0.717327  -1.011291  0);

vectorL     ( -0.754808  -0.634615  -0.404945  -0.079267  0.27069  0.644631  1.201535  1.872656  2.436956
↪  2.967008  3.401311  3.997061  4.788173  5.649904  6.391244  6.876934  7.352635  7.729882  8.179159  8.678481
↪  9.137171  9.528697  9.783547  9.989324  10.106836  );
sectionL    ( -0.301466  0.093199  0.466283  0.935465  1.537404  2.148928  2.696565  3.186897  3.714053  4.4037
↪  5.216965  6.004906  6.623021  7.110266  7.540158  7.959607  8.429212  8.903334  9.32798  9.700926  );
```

Opti-4BV

```
vertexT_1   (1.533621  0  0);
vertexT_2   (1.659796  0.554517  0);
vertexT_3   (-0.136571  1.587864  0);
vertexT_4   (-1.603568  0.991935  0);
```

```
vertexT_5   (-1.469895  0  0);
vertexT_6   (-1.048337  -0.444792  0);
vertexT_7   (0.156036  -1.908189  0);
vertexT_8   (1.528573  -1.811072  0);

vectorL     ( -1.030235  -0.890947  -0.58648  -0.107557  0.388664  0.725946  1.25634  2.015074  2.755729
↪  3.392056  3.879355  4.267987  4.634681  4.97275  5.40339  6.037202  6.762332  7.391485  7.882299  8.359993
↪  9.006944  9.58936  9.891243  10.077471  10.190266  );
sectionL    ( -0.430918  0.128  0.558729  1.008703  1.644468  2.380301  3.063336  3.625385  4.068646  4.449227
↪  4.80638  5.200393  5.732567  6.399573  7.067145  7.630581  8.127652  8.687832  9.287967  9.781241  );

vertexP_10    -4.9421;
vertexP_11    -1.2233;
vertexP_20    46.916;
vertexP_21    35.0314;
vertexP_30    43.8581;
vertexP_31    32.4889;
vertexP_40    49.7479;
vertexP_41    41.6068;
vertexP_50    30.8738;
vertexP_51    17.9182;
vertexP_60     9.9336;
vertexP_61   -14.4528;
vertexP_70   -17.0571;
vertexP_71   -62.4231;
vertexP_80   -59.1712;
vertexP_81   -53.2851;
```