Università degli Studi di Genova

Department of Naval, Electric, and Electronic Engineering and Telecommunications

Master's Degree in Yacht Design

A Python-Implemented Vortex-Lattice Approach for Propeller Optimisation

Author: Lisa Martinez

Supervisor: Simone Saettone, Asst. Professor at the Polytechnic University of Madrid

Tutor: Stefano Gaggero, Assoc. Professor at the University of Genoa

Date: March 2024

Location: Madrid, Spain

To my mother and my brother

Abstract

The aim of this study is to create a Python-based program utilizing the vortex lattice method to enhance the efficiency of marine propellers while achieving a specified thrust.
The optimization procedure consists of solving a variational problem where the torque applied to the propeller is minimized for a given propeller thrust. In classical theory, this problem is addressed through an integral formulation where the propeller is represented as a lifting line with a continuous distribution of circulation. Betz (1927) [1] and Lerbs (1952) [2] provided solutions for this problem, respectively, for propellers in open water and in a radially varying wake, establishing two optimal criteria. To tackle the problem, Munk's displacement theorem is applied, and the problem is linearized. The method employed in this study to solve the problem is based on the approach outlined by Kerwin et al. (1986) [3].
In this method, the approach involves discretizing the continuous distribution of circulation, which allows for a direct solution to the problem without relying on classical theory assumptions. Unlike Kerwin et al., who employed a lifting line model for the propeller, this study utilizes the vortex lattice method. This method enables the integration of the entire blade's impact into the optimization process. The vortex lattice method entails representing the propeller blade with a grid of quadrilateral panels, each with constant circulation. Consequently, horseshoe vortices are formed, following helical trajectories. According to Munk's displacement theorem, specifying the chordwise distribution of circulation is necessary to solve the variational problem. However, it is noted that the primary contribution to the propeller blade's forces comes from the vortex located along the trailing edge, which combines the two shed vortices into a horseshoe vortex. Indeed, in accordance with Munk's displacement theorem, the form of the chordwise distribution of circulation has only a small influence on the results. By incorporating the entire blade in the optimization process, this study aims to examine the impact of propeller geometry on the optimal circulation distribution, thus providing a comparison with Olsen's findings [4].
The study compared the performance of four propellers, each with systematically varied skew and skew-induced rake, from the David W. Taylor Naval Ship Research and Development Center (DTNSRDC) series against the findings of Olsen (2001) [4]. This comparison was found to be highly satisfactory, revealing a consistent trend in the results.
In conclusion, this study presents an approach to optimizing the distribution of circulation along a propeller blade, leveraging the vortex lattice method to extend beyond the confines of classical theory. This methodology facilitates a detailed integration of propeller blade geometry into the optimization process, offering a deeper insight into how propeller geometry influences performance. Importantly, the use of Python, a free and open-source programming language, underscores the study's commitment to accessibility and reproducibility. The Python code developed for this project will be made available in the appendix, allowing others to replicate, verify, and build upon this work without financial barriers. The findings align with and expand upon previous research (Mishima and Kinnas 1997 [36]), notably demonstrating efficiency improvements with increased skew.

Riassunto

L'obiettivo di questo studio è creare un programma basato su Python che utilizzi il metodo vortex-lattice per migliorare l'efficienza delle eliche marine, mirando a ottenere una spinta specifica.
La distribuzione ottimale della circolazione è determinata risolvendo un problema variazionale in cui la coppia dell'elica è minimizzata per una data spinta. Nella teoria classica, questo problema viene affrontato attraverso una formulazione integrale, in cui l'elica è rappresentata come una linea portante con una distribuzione continua di circolazione. Betz (1927) [1] e Lerbs (1952) [2] hanno fornito soluzioni per questo problema, rispettivamente, per eliche in acque libere e con un flusso sulla scia che varia radialmente, stabilendo due criteri ottimali. Per affrontare il problema, si applica il teorema di Munk, e il problema viene linearizzato. Il metodo impiegato in questo studio per risolvere il problema si basa sull'approccio delineato da Kerwin et al. (1986) [3].
In questo metodo, la distribuzione continua della circolazione è discretizzata, consentendo la soluzione diretta del problema senza dipendere dalle ipotesi della teoria classica. A differenza di Kerwin et al., che hanno utilizzato un modello a linea portante per l'elica, questo studio utilizza il metodo vortex-lattice, consentendo l'integrazione dell'intera pala nell'ottimizzazione. L'utilizzo del metodo vortex-lattice comporta la rappresentazione della pala dell'elica con una griglia di pannelli quadrilateri con circolazione costante, risultando in vortici a ferro di cavallo che seguono traiettorie elicoidali. Secondo il teorema di Munk, è necessario specificare la distribuzione di circolazione lungo la corda per risolvere il problema variazionale. Si osserva che il principale contributo alle forze sulla pala dell'elica, proviene dal vortice situato lungo il bordo d'uscita, dove i due vortici liberi si combinano in un vortice a ferro di cavallo. Infatti, in accordo con il teorema di spostamento di Munk, la forma della distribuzione di circolazione lungo la corda ha solo una piccola influenza sui risultati. Considerando l'intera pala nel processo di ottimizzazione, è possibile esaminare e confrontare l'impatto della geometria dell'elica sulla distribuzione ottimale della circolazione, con i risultati di Olsen [4].
Lo studio ha confrontato le prestazioni di quattro eliche, ciascuna con Skew e Rake indotti da Skew, sistematicamente variati, della serie di eliche David W. Taylor Naval Ship Research and Development Center (DTNSRDC), con i risultati di Olsen (2001) [4]. Il confronto è risultato molto soddisfacente e ha rivelato una tendenza coerente nei risultati.
In conclusione, questo studio presenta un approccio per ottimizzare la distribuzione della circolazione lungo una pala d'elica, sfruttando il metodo vortex-lattice, per andare oltre i limiti della teoria classica. Questa metodologia facilita l'integrazione dettagliata della geometria della pala dell'elica nel processo di ottimizzazione, offrendo una visione più approfondita di come la geometria dell'elica influenzi le prestazioni. L'uso di Python, un linguaggio di programmazione libero e gratuito, sottolinea l'impegno dello studio verso l'accessibilità e la riproducibilità. Il codice Python sviluppato per questo progetto sarà reso disponibile in appendice, consentendo ad altri di replicare, verificare e sviluppare questo lavoro senza barriere finanziarie. I risultati si allineano e ampliano le ricerche precedenti (Mishima e Kinnas 1997 [36]), dimostrando, in particolare, miglioramenti dell'efficienza con l'aumento dello Skew.

Acknowledgement

I would like to say few words for the people who accompanied me along this path and made this project possible.

First and foremost to Simone Saettone, not only for being an excellent Supervisor and Professor during my time in Madrid, but also for being a valuable guide. Your support, understanding, and knowledge-sharing have been invaluable assets that I will carry over with me into my future. Without your help this would not have been possible. Your teachings have illuminated my academic path, and at the end of this journey, I am certain that you are one of the most professional individuals I have had the luck to meet.

I extend my thanks to all the people I met at the Universidad Politécnica de Madrid for welcoming me and making me feel at home from the very beginning. I will never forget the kindness of each one of you, especially Javier, Ricardo, Gustavo, Wenzhe, and Gaia.

I am grateful to my entire family for supporting my dreams. A special thanks to my parents for their unwavering support, for standing by me, and for loving me unconditionally.
To my mother, thank you for being there during tough times, for always understanding my actions, and for loving me unconditionally.
To my father, thank you for providing me with the tools and for raising the woman I am today, you are the example of my life.
To my brother, Andrea, thank you for every moment we've shared together and for understanding my absence without ever making it a burden.
To my uncle Alessio, thank you for being there for me both physically and emotionally, you have always been my benchmark in discovering the world.

A special thank to Rodrigo, although words are not enough to express my gratitude towards you. You were the first to share the joys and difficulties of this journey with me, always by my side. Thank you for your unconditional sweetness and love, for holding my hand when I was scared, and for simply being in my life. I love you.

To my lifelong friends, Carla and Camilla, centuries may pass and galaxies may separate us, but our bond will remain unbreakable. You are life's precious gift, and with you, I am certain that I am never alone.

Thanks to my friends in Spezia, the city that hosted me for five long years. To Selene, for everything, for every moment we shared that I hold dear. To Billy, our souls are so similar, I spent my best moments during my university time while we were in the same flat. To Davide, for being there for me in difficult times, even though our friendship is relatively new, I know it will last long. To Stefano, for always making me feel at home, for the talks and advice. To Ilario, a lifelong friend, for being like me and for understanding me without a word, our connection goes beyond words.

Finally, I would like to thank all the people I met this year in Madrid, especially Pierpaolo. What we shared will bind us forever.

Contents

List of Tables

List of Figures

1 Introduction

Despite the challenges posed by various global crises in recent years, including economic downturns, geopolitical tension, and pandemics, a significant portion of global trade-exceeding two-thirds-continues to be conducted through maritime transport. This includes vital trade involving food, energy, and other essential commodities. Specifically, maritime transport accounts for 77% of European foreign trade and 35% of European trade [15]. As of beginning of 2023, the total maritime fleet comprised of 105,500 vessels of at least 100 gross tonnage (GT), and offering a capacity of 2.3 billion deadweight tons (DWT), marking an increase of 70 million DWT compared to the preceding year.

Moreover, the volume of maritime trade has been on an upward trajectory, experiencing a 2.3% increase in 2023, with projections indicating a further growth rate of 2.1% over the coming five years. This trend not only highlights the critical role of maritime transport in enabling global trade but also brings to light the consequential rise in the average distance covered to transport goods.Such an expansion in maritime trade activities has precipitated a notable increase in carbon emissions, which, at the outset of 2023, were observed to be 20% higher than figures recorded two decades prior [15]. This exacerbates the urgency of addressing the environmental impact associated with maritime trade.

Given the current pace of trade expansion and increasing demand, emissions are forecasted to escalate by 50% to 250% by the year 2050, barring the implementation of effective mitigatory strategies. Presently, a staggering 98.8% [4] of the global shipping fleet is dependent on fossil fuels, with the combustion of marine fuel oil (HFO) releasing significant quantities of carbon dioxide (CO2), methane (CH4), and nitrous oxide (N2O) into the atmosphere. It's worth noting that merchant vessels emit approximately 16.14 grams of CO2 per kilometer for each ton of cargo transported, although it is observed that larger ships and cargoes generally achieve greater energy efficiency on a per-unit-load basis. This reality accentuates the pressing need to confront and mitigate the environmental consequences stemming from maritime trade, underscoring the urgency of action in this domain.

In response to growing environmental concerns, national and international organizations are stepping up to tackle the pollution caused by ships. At the 76th session of the Marine Environment Protection Committee (MEPC 76) in June 2021, changes were made to the International Convention for the Prevention of Pollution from Ships (MARPOL), which started being enforced at the end of 2022. The International Maritime Organization (IMO) is now requiring ships to adopt short-term actions to cut down on pollution [6]. The goal is to significantly lower emissions by 2050, aiming for a 40% reduction by 2030 compared to 2008, and aiming even higher with a 70% reduction by 2050. [23].

To put these new rules into practice, ships must calculate two things: the Energy Efficiency Existing Ship Index (EEXI) and the Carbon Intensity Indicator (CII). This approach, which started in 2013 for new ships with the Energy Efficiency Design Index (EEDI), is now being applied to all ships. Specifically, ships that are 400 gross tonnage or larger must complete the EEXI calculation. This is a big move towards making ships more energy-efficient and reducing their impact on the environment.
The EEXI is represented by the amount of CO2 emitted per unit of traffic volume and is calculated based on fuel consumption and other ship characteristics [7]. On the other hand, the CII determines the annual reduction factor required to ensure a continuous improvement in the

operational intensity of carbon emissions from a ship.

Various strategies proposed by the International Maritime Organization (IMO) aim to improve the energy efficiency of maritime vessels. These include leveraging renewable energy for power generation, adopting alternative fuels like liquefied natural gas (LNG) to lower emissions, refining ship design and equipment for better operational efficiency, imposing power restrictions, providing shore-based electricity supply, and introducing supportive measures, such as improvements in land-based transport and logistics management.

It is important to highlight that by concentrating on the hydrodynamic performance of ships, which entails the optimization of propeller design, it is possible to significantly increase ship efficiency. This approach is not only in alignment with the EEXI and CII regulations but also emphasizes the potential for substantial improvements in maritime energy efficiency.

1.1 Investigating Adopted Strategies

Decarbonization strategies for ships focus on two main areas: improving energy efficiency, derived from technical modifications to the ship's structures and operational adjustements for better navigation, and adopting new-generation clean fuels, specifically hydrogen, liquefied natural gas (LNG) and ammonia. Research from the Norwegian University of Science and Technology [9] has confirmed that emissions can be significantly reduced through these approaches, identifying six key mitigation strategies with substantial potential for emission reduction: hull design improvements (4-30%), economies of scale and advancements in power and propulsion (2-45%), optimizing speed (1-60%), adopting cleaner fuels such as LNG and ammonia (25-84%), exploring alternative energy sources (1-50%), and improving weather routing and scheduling (0.1-48%).

Within the power and propulsion strategies framework, a central approach to minimising emission in the maritime industry involves optimising propeller design [8]. This effort is integral to improving the efficiency of ship propulsion system [9], directly impacting fuel consumption and, consequently, emissions. Such optimisation is crucial for reducing the torque required while maintaining the same thrust, thereby increasing the propeller's efficiency. This approach directly addresses the need to enhance propeller performance by achieving grater propulsion efficiency with less energy cost. By optimising the design to minimise torque demands without compromising on thrust, ships can achieve smoother and more fuel-efficiency operations, significantly improving overall propeller and vessel efficiency.

1.2 Objective of the Thesis

As of the current date, the selection of codes available for marine propeller optimization, remains limited, with OpenProp, PROPAN, and Xfoil begin among the most notable. Despite their open-source status, their implementation in either MATLAB or Fortran requires either financial expenditure for a license or specialized programming expertise. This situation highlights the urgent need for the development of a new open-source code utilizing Python programming language. Such a development aims to offer a freely accessible and user-friendly alternative for the broader community, overcoming the limitations associated with proprietary platforms.

The choice to implement the method in Python is dictated by the fact that it is the most widely used programming language in the scientific field. This is because it is open to everyone and has a simple language that is easy to understand. Additionally, it has been developed over the years, providing a large library, manuals, and information developed by programmers. You can implement and automate many functions, and its extensive collection of libraries makes it usable across various fields.

2 Literature Review

2.1 Propeller Design

The design of propellers is a complex task that balances efficiency, power, and noise reduction, among other considerations. Primarily, there are three approaches to propeller design: the series approach, numerical methods, and experimental methods. Each has its advantages and is frequently used in combination with others to develop and refine propeller systems. The series approach utilizes data from previously tested propellers under various conditions to create new designs. It relies on systematic variations of essential propeller parameters such as diameter, pitch, blade number, and shape. Designers can consult charts or databases documenting the performance of different propeller geometries to select a design that closely meets their requirements. A renowned example is the Wageningen B-Series.
Numerical methods employ computational techniques to simulate the flow around propellers and predict their performance. Within the realm of numerical approaches for propeller design, two primary methods are significant: potential flow theory and Computational Fluid Dynamics (CFD). Potential flow theory simplifies the complex physics of fluid motion by assuming the fluid is inviscid and irrotational, effectively ignoring viscosity's effects. While this simplification reduces computational demands, it offers valuable insights into the flow field around propellers, particularly beneficial during the preliminary design phases for a broad exploration of the design space. CFD involves a range of computational techniques that solve the Navier-Stokes equations to simulate fluid flow with high fidelity. It captures complex flow phenomena, including turbulence, separation, and viscous effects, providing a detailed understanding of the flow around a propeller. However, the high computational cost of CFD, requiring more powerful computing resources and longer computation times, makes it less suitable for initial exploratory studies but invaluable for finalizing designs and conducting detailed performance analyses.
Experimental testing involves physically manufacturing a propeller and testing it in a controlled environment, such as towing tanks and cavitation tunnels. These tests yield essential data on the propeller's performance, including thrust, torque, and efficiency, along with insights into flow patterns, noise, and vibration levels. Experimental methods are often used to validate and refine designs derived from series or numerical simulations. Despite being expensive and time-consuming, experimental testing remains an indispensable part of the propeller design process, especially for final validation before production or for investigating new concepts.

Concentrating on potential flow theory, several key methodologies stand out. Among these, the lifting line model is particularly noteworthy for its simplicity in depicting propeller action. In this model, the intricate aerodynamic profiles of blade sections are elegantly replaced with a singular line vortex, providing a streamlined yet effective approach to understanding propeller dynamics. However, this simplification also serves as its primary limitation, as it fails to accurately capture three-dimensional flow effects and complex vortical patterns, especially near the blade tips. Moving on to the lifting surface model, this approach offers a more nuanced representation by considering the propeller blades as finite lifting surfaces. This method allows for a better approximation of the three-dimensional flow around the blades, capturing the essential aspects of blade geometry and its influence on performance. Despite its increased accuracy over the lifting line model, the lifting surface model is still hampered by its reliance on potential flow theory, which overlooks viscous effects and may not accurately predict performance in off-design conditions. [10].
Lastly, the boundary element method (BEM) represents a further advancement in modeling marine propellers. By discretizing the propeller blade and surrounding fluid domain into small elements, BEM can simulate the flow around the propeller with high fidelity, incorporating

both potential flow and viscous effects under certain formulations. This method is particularly effective in analyzing complex flow phenomena, such as cavitation and highly skewed flows. However, BEM's computational demand is significantly higher, requiring more sophisticated computational resources and longer processing times, which can be a considerable drawback for extensive parametric studies or real-time applications.
In summary, while each method has its pros and cons, the choice between the lifting line, lifting surface model, and boundary element method depends on the balance between computational efficiency and the level of detail required for accurate propeller performance prediction..

As mentioned previously, CFD provides detailed information about flow and pressure distributions, surpassing previous methods in its ability to capture complex fluid dynamics and interactions in marine propellers. Among the most commonly used solvers in CFD are the Reynolds-Averaged Navier-Stokes (RANS), Detached Eddy Simulation (DES), and Large Eddy Simulation (LES). These solvers offer different approaches to modeling turbulence, which is a key aspect in understanding the flow around marine propellers. In practice, fluid equations are substituted with discrete approximations at grid points, and the solution remains dependent on the spacing between grid points. Sometimes, the vortex lattice method or BEM method is coupled with a Reynolds-Averaged Navier-Stokes (RANS) method [11] , providing valuable information on the viscous and cavitation behavior of propellers in analytical cases.While CFD yields accurate results, its practical complexity and computational times make it challenging to implement automated optimization.

2.2 Lifting Line Theory

Betz (1919) [1] expanded upon Prandtl's lifting-line theory to establish the basis for determining the radial distribution of circulations [10]. In this theory, the lift generated by a wing or propeller blade results from the circulation development around the section, following the Kutta-Joukoski law (the flow separates from the trailing edge in a 'smooth' manner with a finite velocity value). Betz introduced a criterion for minimal energy loss, defining the concept of an optimum propeller. The optimum propeller develops a trailing vortex system, creating a rigid helicoidal surface that extends infinitely downstream from the blade. This surface must translate as a rigid entity in the downstream direction. While the Betz condition remains accurate for propellers operating in uniform flow, it begins to demonstrate limitations for heavily loaded propulsors.

Goldstein (1929) [24], solved the potential problem, following Prandtl's concept: the three-dimensional problem can be solved by concentrating circulation around the blades on individual lifting lines, and the flow in each radial section could be considered two-dimensional if the velocity induced by the free flow alters the field in which they are located. The solution proved successful for aircraft. However, it was unsatisfactory for marine propellers, which are designed with low aspect ratios to mitigate cavitation phenomena. Additionally, the onset flow for propeller rotation is typically non-uniform. In the initial stages, corrections were made to adjust the camber of 2-D sections to accommodate the induced curvature of the flow. This curvature results from the velocity induced by the trailing vortex sheet, which is greater at the trailing edge than at the leading edge.

Seventeen years later, Cox (1961) [25], published precise results of these corrections, which were obtained using computers. In summary, analytical methods for practical applications were not available before the 1950s. In 1952 [2], Lerbs introduced changes by extending lifting line theory to include propellers with arbitrary radial distributions of circulation under both uniform and radially varying inflow conditions. Subsequently, in the 1960s, this procedure was computerized.

2.3 Lifting-Surface Method

Lerbs' method continues to be utilized for radial distribution in the initial stages of design. During this period, Eckhart and Morgan (1955) [26], developed a combination of Lerbs' lifting-line theory and lifting-surface correction for camber and angle of attack, marking a significant advancement in lifting surface theory. As technology became more available, numerical methods for lifting surface evolved, including those developed by Kerwin (1961) and van Manen & Bakker (1962) [11]. However, these methods were based on simplifying assumptions that became inadequate with technological advancements.

During the early stages of lifting surface analysis, linear theory was employed to simplify the problem. Linear theory assumes that the blade and wake can be projected onto stream surfaces formed by the undisturbed flow. This was necessary for the design process, where only a partial understanding of the blade surface geometry is initially available. Determining the radial distribution of pitch, as well as the chordwise and radial distribution of camber, becomes necessary, and for calculating their induced velocity, sources and vortices must be positioned. However, in reality, the resulting blade surfaces often deviate from the assumptions made in linear theory. Therefore, the procedure computes the total fluid velocity at a number of points on the surface and then adjusts the surface in such a way as to annul its normal component.

Two calculation methods for the lifting surface are PROPLS, developed by Brockett (1981) [27], which directly integrates the resulting singular integrals, and PBD-lO, developed by Kerwin [11], which employs a vortex-lattice procedure. In Kerwin's method, the process starts with assuming the pitch and camber, then calculating the total flow velocity. Afterward, the surface is adjusted, the process is repeated using the new reference surface until convergence is obtained. In the vortex lattice approach, continuous distributions of vortices and sources are substituted with a series of concentrated rectilinear elements. These elements have endpoints positioned along the average surface of the blade. Velocities are subsequently computed at control points strategically positioned between these elements. Therefore, proper placement of control points and lattice elements is crucial.Vortex lattice methods are typically highly robust and James (1972) [17] and Lan (1974) [18] both provided rigorous demonstrations of the convergence of vortex-lattice methods in two-dimensional flow. James specifically addressed scenarios with constant vortex spacing, confirming that placing the control point at three-fourths of the element length yields the correct solution.

Subsequent advancements in propeller design were pioneered by Tsakonas et al. (1983) [28], Lee (1978) [29], van Gent (1977) [30], and Greeley (1982)[31]. These methods diverged from traditional approaches by acknowledging that induced velocities might not always be negligible compared to the initial flow velocity. They allowed for deviations in the positions, of the blade and the wake, of the trailing vortex from the undisturbed flow surface. The primary objective was to address the limitations of previous methods, particularly in their treatment of chord-wise lift, and to incorporate the effects of skew and rake into the analysis.

Lee and Kerwin et al. developed the vortex lattice code PUF-3 in its original form (1978) [29] then, Greeley and Kerwin expanded upon the existing approach by introducing a semi-empirical method aimed at forecasting the leading-edge separation point (1982). Greeley employed a program that utilizes a vortex lattice model for the blades, aligning with the design process outlined earlier. However, in this approach, each vortex element along the span is treated as an unknown and determined through collocation using an equal number of control points distributed across the blade. To model the strength of circulation/lift, a distribution of vortices is positioned on the mean surface of the blades. These vortices represent the circulation or lift generated by the blades. Additionally, to account for induced drag, several free trailing vortices are shed from each blade element.

Initially, the circulation distribution on the blades, and consequently in the wake, is determined based on an assumed wake geometry. This circulation distribution remains fixed while iteratively adjusting the position of the wake to align with the flow. This iterative process continues until convergence is achieved, indicating that the wake is accurately aligned with the flow.Once convergence is reached, the circulation distribution is recalculated based on the adjusted wake geometry, and the entire process is repeated. Iterations continue until the changes in the circulation distribution fall below a certain predefined tolerance level.

Brockett [27], calculates the induced velocities on the blades through one of direct numerical integrations. He assumes the blades to be thin, which allows the singularities distributed on both sides of the blades to collapse into a single surface. Additionally, he suggests defining the effective wake as the total velocity at any point in the fluid with a propeller in operation, subtracting the potential component of the propeller-induced velocity. This definition simplifies the propeller problem to determining the velocity potential in an unbounded fluid, satisfying the kinematic boundary condition on the propeller surface, along with kinematic and dynamic boundary conditions at the trailing edge and on the trailing vortex sheets behind the blades. However, he himself demonstrates the robustness of the convergence proofs of vortex-lattice methods in two-dimensional flow.

During recent years, development in studies on less conventional propeller designs and wake alignment has advanced. Leading figures in this area include Kerwin et al. (1986)[3], Andersen (1997)[32], developed the theory for tip-modified geometry, where the lifting line can be curved, in order to include the influence of skew and rake, and Jong (1991) [20]. Additionally, for investigations into energy coefficients and analyses under unsteady and off-design conditions, Caponetto (2000)[33], and Karim et al. (2001) [34], have made significant contributions.

2.4 Boundary Element Method

The boundary element method for propeller analysis has been developed in recent years to overcome two challenges of lifting surface analyses. The first relates to the occurrence of local errors near the leading edge, while the second concerns more widespread errors near the hub, where blades are closely spaced and relatively thick.

Although a local correction derived from Lighthill's work can address the first problem to some extent, the second problem persists. Boundary element methods, essentially panel methods, were initially introduced in the aircraft industry and later applied to propeller technology in the 1980s.
Hess and Valarezo (1985) [35], introduced an analysis method based on earlier work by Hess and Smith. Hoshino subsequently developed a surface panel method for hydrodynamic analysis of propellers operating in steady flow. These methods have achieved good agreement between theoretical and experimental results for blade pressure distributions and open water characteristics. Further advancements, such as those by Kinnas and colleagues at the University of Texas, Austin, have extended boundary element codes to solve for unsteady cavitating flow around propellers, considering non-axisymmetric inflow conditions and other factors such as mid-chord cavitation and unsteady tip vortex cavitation.

Additionally, efforts have been made to enhance slipstream flow prediction using iterative methods aligning the wake surface to local flow conditions. Within the framework of the MARIN-based Cooperative Research Ships organization, Vaz and Bosschers have developed a three-dimensional sheet cavitation model using a boundary element model of marine propellers. These developments aim to improve prediction accuracy under various conditions, including behind conditions and cavity volume variations influenced by non-cavitating propeller effects and viscous effects.

2.5 Computational Fluid Dynamics

During the past decade, significant advancements have been achieved in applying computational fluid dynamics (CFD) [13]. These advancements have enabled valuable insights into the viscous and cavitation behaviors of propellers, particularly in the analysis context. However, while progress has been made in using these methods for design purposes, widespread acceptance has not yet been attained. Various modeling approaches, including Reynolds Averaged Navier–Stokes (RANS) method, Large Eddy Simulation (LES), Detached Eddy Simulations (DES), and Direct Numerical Simulations (DNS), have been developed for analyzing flow around cavitating and non-cavitating propellers.

However, in practical propeller computations, computational efforts limit the application of many of these methods. RANS codes are favored due to their relatively lower computational times compared to other methods. Despite common features such as multi-grid acceleration and finite volume approximations, differences exist among practitioners in grid topology, cavitating flow modeling, and turbulence modeling.

2.6 Conclusion

The propeller optimization code employs the vortex-lattice approach. This method stands out for its computational efficiency, achieving significant savings in computational time during the design phase without compromising on accuracy. Over the years, the demonstrated functionality of the vortex lattice method has underscored its reliability in providing accurate approximations of propeller performance. Notably, it facilitates effective calculation of circulation on propeller blades, further highlighting its utility. The lifting line model was not selected because the vortex lattice method offers a superior capability to capture three-dimensional flow effects without significantly increasing computational time or complexity. Furthermore, the complexity of the Boundary Element Method (BEM) and the extensive computational demands of Computational Fluid Dynamics (CFD) rendered them unsuitable for the current project. Additionally, the prohibitive costs associated with physical model testing render such approaches impractical for the current project.

This study utilizes Kerwin's method (1986) [3] to establish the optimal distribution of circulation by minimizing torque for a given thrust through solving a variational problem. Essential to this approach is the incorporation of the entire blade's effect. Thrust and torque calculations for the propeller are executed using the vortex lattice method, accommodating nearly arbitrary propeller geometries. The method integrates a simple wake and blade alignment procedure akin to moderately loaded lifting lines, with thickness and hub effects omitted for simplicity. The study also considers skin friction drag. Providing input data such as propeller radius, hub radius, number of blades, chord length, skew, and rake distributions is required.

3 Potential Flow Theory

In fluid dynamics, the potential flow theory describes the velocity field of an inviscid, incompressible, and irrotational fluid as the gradient of a scalar function called potential, denoted as Φ:

$$\frac{\partial \Phi}{\partial x_i} = u_i \tag{1}$$

This equation defines each component of the velocity in terms of the local spatial partial derivative, in the direction of the velocity component.

3.1 Simplified Mathematical Models

In fluid dynamics, the behavior of fluids is governed by various forces and moments, similar to how rigid bodies are governed. However, in fluids, these forces are distributed continuously throughout the fluid rather than acting at specific points. This means that the motion of fluid particles and the distribution of forces are described continuously, assuming that the individual molecules can be treated as part of a continuum. The three principal forces are inertial, gravitational, and viscous. Typically, gravitational forces are ignored, and the fluid can be considered inviscid with a high Reynolds number, because viscous effects are limited to the boundary layer. Consequently, external forces are primarily due to the lifting surface in the fluid

Before delving into describing fluid flows with the velocity potential, it's crucial to introduce two foundational principles: the equations for conservation of mass and for the conservation of momentum. In this discussion, simplified forms following Newman's approach will be utilized. [13]. The principle of conservation of mass, when applied to a continuum of fluids in motion, asserts that within a three-dimensional volume in space—modeled as a cube—where mass can flow through each face of this geometric element, mass cannot be created or destroyed over time but is conserved. Consequently, the net inflow into the volume, subtracted from the net outflow from the volume, equals the net change in mass within the volume.

$$\frac{\partial}{\partial t} \int_V \rho \, dV + \int_S \rho(\vec{u} \cdot \vec{n}) \, dS = 0 \tag{2}$$

$\frac{\partial}{\partial t}$ represents the partial derivative with respect to time t.

$\int_V \rho \, dV$ denotes the integral of mass density ρ over volume V.

$\int_S \rho(\vec{u} \cdot \vec{n}) \, dS$ represents the integral of the mass flux $\rho\vec{u}$ across the surface S with the normal vector $\vec{n}$.

Similarly, the conservation of momentum states that, the sum of all forces acting on the fluid volume, must equal the rate of change of momentum density of fluid particles.

$$\frac{\partial}{\partial t} \int_V \rho\vec{u} \, dV + \int_S \rho\vec{u}(\vec{u} \cdot \vec{n}) \, dS = \sum \vec{F} \tag{3}$$

$\frac{\partial}{\partial t}$ represents the partial derivative with respect to time t.

$\int_V \rho\vec{u}\, dV$ denotes the integral of momentum density $\rho\vec{u}$ over volume V.

$\int_S \rho\vec{u}(\vec{u}\cdot\vec{n})\, dS$ represents the integral of the momentum flux $\rho\vec{u}(\vec{u}\cdot\vec{n})$ across the surface S with the normal vector $\vec{n}$.

$\sum \vec{F}$ represents the sum of all external forces acting on the system, such a surface and body forces.

The mass and momentum equations are sufficient to describe fluid motion, but the use of a differential representation is more practical. However, these equations are quite complex, nonlinear, and interconnected, which makes solving them a challenge. Although empirical evidence supports the Navier-Stokes equations for describing Newtonian fluids (where viscosity stays constant regardless of flow velocity or stress), finding analytical solutions is often difficult. To make progress in fluid dynamics, simplifications are often applied to the equations by neglecting certain terms or assuming their values to be zero. However, these simplifications may introduce errors into the analysis. Despite this, using simplified equations is often justified because they are easier to compute compared to the full equations. In the following, situations where such simplifications can prove advantageous, will be discussed.

- Inviscid flow
- Irrotational flow
- Incompressible flow

In numerous applications, it's common to assume, that the fluid density remains constant. This assumption holds true not just for liquid flows, where compressibility can often be neglected, but also for gases when the Mach number is below 0.3. Incompressible flow refers to motion that doesn't involve expansion. Additionally, if the flow is isothermal, the viscosity remains constant as well.
The flow can be treated as inviscid, because in flows far from solid surfaces, viscosity effects are typically minimal. When viscous effects are completely neglected, essentially assuming the stress tensor reduces to zero, the Navier-Stokes equations simplify to the Euler equations. Since the fluid is considered non-viscous, it doesn't stick to walls, allowing for slip at solid boundaries. At high velocities, the Reynolds number is very high, and viscous and turbulence effects only become significant in a small region near the walls. By incorporating a frictional drag coefficient, the friction drag between the fluid and the body is accounted for. Using the Euler equations, flow motion can be predicted accurately.

The continuity equation for a steady incompressible and inviscid fluid becomes:

$$\nabla\cdot\vec{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} = 0 \tag{4}$$

and the momentum equations:

$$\nabla\left(\frac{p}{\rho} + \frac{1}{2}|\vec{u}|^2\right) - \vec{u}\times\vec{w} = \frac{\vec{F}}{\rho} \tag{5}$$

where: $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ is the gradient operator.
$\vec{w} = \nabla\times\vec{u}$ is the vorticity.
p is the pressure.
F represents the external force exerted by the lifting surface in the fluid. The forces exerted

by the fluid on these surfaces, are equal in magnitude but opposite in direction, to the forces exerted by the surfaces on the fluid.

After these initial assumptions, the equations of motion become independent of time and rely solely on the Cartesian coordinates (x, y, z). Additionally, the studied body is fully submerged, the fluid is water, and it is assumed that the wake behind the hull is steady and axi-symmetric.

3.2 Irrotational Flow

To further simplify these equations, let's start by narrowing down the range of fluid motions and introducing the concept of circulation. By applying Kelvin's theorem to two points P_1 and P_2 within a connected region of the fluid, connected by paths forming a closed and continuous loop C, the circulation, denoted as Γ, is defined as the integral of tangential velocity around this closed contour C. This circulation remains constant if the fluid is subjected to conservative forces.

$$\Gamma = \int_C \vec{u}_i \, d\vec{x}_i \tag{6}$$

Thanks to Stokes's theorem, the circulation can be related to the vorticity vector. For a continuously differentiable vector $\vec{u}$, it holds that:

$$\int_S (\nabla \times \vec{u}) \cdot dS = \int_C \vec{u} \cdot d\vec{x} \tag{7}$$

In a frame tied to the body, where the velocity remains steady, it only varies with position and stays constant infinitely far away. As a result, the vorticity w remains zero across all points in the flow field, that can be traced back to infinity through streamlines. This outcome is a direct implication of Kelvin's theorem, stating that the circulation measured along any closed material line, remains constant over time.

Consequently, any motion starting from a stable condition, will persist as irrotational over time. The absence of rotation in a potential flow, arises from the fact that the curl of a gradient is always zero, causing circulation to vanish. Since the flow starts from a state of rest, circulation should remain zero, indicating that the integrand must be zero as well. Thus, the fluid's motion is irrotational:

$$\nabla \times \vec{v} = 0 \tag{8}$$

This conclusion holds significant implications because an irrotational vector field can be represented as the gradient of a scalar function. This assertion is a consequence of Helmholtz's theorem in vector analysis, which states that any continuous and finite vector field can be expressed as the sum of the gradient of a scalar function Φ and the curl of a zero-divergence vector, this vector vanishes identically, if the original vector field is irrotational. Therefore, if the velocity field is irrotational, it can be simplified to just the gradient of the scalar function Φ, also known as the velocity potential.

This simplification greatly aids in analyzing and understanding fluid motion, as it reduces the complexity of the vector field representation to a scalar function.

$$\nabla \Phi = \vec{u} \tag{9}$$

and inserting this in the continuity Equation (4), one obtains:

$$\nabla \cdot \vec{u} = \nabla \cdot \nabla \Phi = \nabla^2 \Phi = 0 \tag{10}$$

The motion can now be described by Laplace's equation:

$$\nabla^2 \phi = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2} = 0 \tag{11}$$

- It is a partial differential equation.
- It is a linear equation (for which superposition of effects applies), and the elementary solutions/functions, which are continuous and derivable except possibly at some contour points, can be used to derive solutions of more complex problems
- It is solvable, like any differential equation, once the boundary conditions are provided
  - Dirichlet Conditions: The value of the potential is imposed on the boundary.
  - Neumann Conditions: The value of the normal derivative of the potential is imposed on the boundary of the domain.

3.3 Kutta Condition

The derivation of the Laplace equation is valid only for simply connected regions, where, the circulation (line integral) of velocity, along a closed curve is always zero (6). This also guarantees the uniqueness of the solution, except for an additive constant, that does not affect the velocity problem, as they are the derivatives of the potential, indifferent to constants. If the region is multiply connected, it can be made simply connected by "cutting" the region itself. However, the circulation calculated around a non-reducible curve, is no longer zero: its value is constant for any curve surrounding the body and is constant along the cut. The uniqueness of the solution

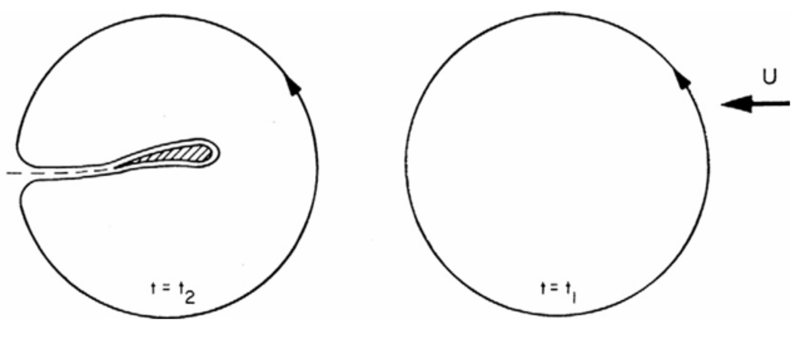


Figure 1: Simply connected region with a cut

to the potential problem, for regions made simply connected, is guaranteed if the intensity of this circulation is specified. To ensure the uniqueness of the solution, it is necessary to know the circulation and consider the nature of the physical phenomenon under study. Considering points 1 and 2, boundary conditions must be applied due to the discontinuity of the potential on the boundary:

- The normal derivative of the potential in the wake must not only be constant but also zero.
- The potential jump in the wake remains constant.
- The pressure must be equal across the cut, as per the Kutta condition $P_1 = P_2$.

3.4 Bernoulli Equation

The velocity is determined without requiring dynamic considerations; it simply needs to be kinematically compatible and respect the boundary conditions. The pressure is derived from the momentum equation, taking into account that the body force is conservative and can be expressed using a scalar function E, $\vec{F} = -\nabla E$. The Euler equation for incompressible and irrotational flow, with a conservative body force becomes:

$$\nabla(\frac{p+E}{\rho} + \frac{|\vec{u}|^2}{2}) = 0 \tag{12}$$

The terms in the brackets should be constant to satisfy the equations, and the equation often referred to as the Bernoulli equation is:

$$(\frac{p+E}{\rho} + \frac{|\vec{u}|^2}{2}) = C \tag{13}$$

From the momentum conservation equation, the integrated form of Bernoulli's equation, allows the derivation of pressure given the knowledge of velocity, completing the solution. Time does not explicitly appear in the Laplace equation due to its nature, which assumes an infinite propagation velocity of disturbances, causing the flow field to adapt instantaneously to changes in boundary conditions. However, it's important to note that time does appear in the expression for pressure and in the velocity field.

At this point, it's important to remember, as mentioned, that the Laplace equation is linear. This implies that the boundary problem can be separated into a value problem, for the undisturbed onset flow ϕ_{onset} and for the perturbed flow ϕ. Then, these two values can be summed. The potential flow for the onset flow can now be expressed as:

$$\Phi_{onset} = \vec{U} \cdot \vec{x} = U_{0,x}x + U_{0,y}y + U_{0,z}z \tag{14}$$

If the disturbance velocity of the body is small compared to the undisturbed flow, the equation can be linearised (Breslin and Andersen, 1994) [21]:

$$p_\infty - p = \rho U_0 u_x \tag{15}$$

Where u_x represents the axial component of disturbance velocity. Rewriting the equation in terms of pressure coefficient ΔC_p, it is formulated as follows :

$$\Delta C_p = \frac{p - p_\infty}{\frac{1}{2}\rho U_0^2} \approx 2\frac{u_x}{U_0} \tag{16}$$

3.5 Lifting surface

The importance of lifting surfaces in fluid mechanics, particularly in supporting aircraft, hydrofoil boats, and various control surfaces such as rudders and yacht sails, cannot be overstated. These surfaces are engineered to maneuver through the surrounding fluid at a slight angle of attack, thereby generating hydrodynamic lift forces. The aspect ratio, which measures the extent to which flow is influenced, by the three-dimensional nature of the surface, plays a crucial role. A high aspect ratio suggests flow that is largely independent of the transverse coordinate, while a lower aspect ratio indicates significant three-dimensional flow effects.

In the subsequent analysis, the scenario of a propeller operating under two-dimensional flow conditions is examined, where the boundary conditions imposed on the contour are applied. Two primary types of boundary conditions are addressed: a kinematic condition concerning the fluid velocity at the boundary and a dynamic condition related to the forces acting on the boundary. For a material boundary separating a fluid from another medium, the tangential velocity at the surface must remain continuous. Specifically, if the solid surface is stationary, the tangential velocity must be zero. In the case of an impermeable solid, it is assumed that there is no separation or interpenetration; thus, the normal velocities of the fluid and the boundary coincide. This is known as the kinematic condition or non-slip condition:

$$\nabla\Phi \cdot \vec{n} = 0 \tag{17}$$

Expanding upon the potential definition:

$$\frac{\partial \Phi}{\partial n} = -U_0 \cdot \vec{n} = 0 \tag{18}$$

where:

- $\vec{n}$ is the unit normal vector of the surface with direction from the surface into the fluid,
- U_0 is the velocity for the undisturbed onset flow.

The Kutta condition ensures that the velocity at the trailing edge remains finite, thereby mathematically enforcing the assumption of smooth tangential flow:

$$\nabla\Phi < \infty \ \textit{at trailing edge} \tag{19}$$

The influence of the body diminishes as the distance from it increases, therefore the perturbation potential decreases from a finite value to zero at infinity:

$$\nabla\Phi \rightarrow 0 \ \textit{at infinity} \tag{20}$$

In the general scenario, the perturbation potential, adheres to boundary conditions suitable for slender bodies with small angles of attack. Airfoils exemplify such slender profiles where separation effects remain negligible, allowing us to employ thin wing theory.

3.6 Linearised Thin Wing Theory

The thin wing theory, originally formulated for flow around two-dimensional wing sections, assumes a purely two-dimensional flow in this context, confined to the x-z plane as depicted in Figure 2, as the name implies, the theory is specifically tailored for slender profiles, with the additional condition that the angle of attack remains small. An illustrative profile is presented in the Figure 3:

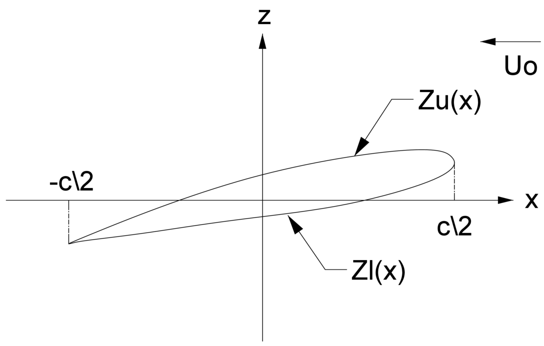


Figure 2: Notation for two-dimensional section

where $z_u(x)$ delineates the upper side of the profile, $z_l(x)$ denotes the lower side, and c represents the chord length.For the thin wing assumption to hold, both $z_u(x)$ and $z_l(x)$ should be significantly smaller than the chord length. Additionally, the slope of the profile, represented by $z'_u(x)$ and $z'_l(x)$, should be considerably less than one. If these conditions are fulfilled, the velocity boundary condition specified in Equation (18) can be linearised (Newman, 1978 [13]), thus simplifying the analysis:

$$\begin{aligned} \frac{\partial \phi}{\partial z} &= -U z'_u(x) \text{ on } z = 0_+, \quad -\frac{c}{2} \leq x \leq \frac{c}{2} \\ \frac{\partial \phi}{\partial z} &= -U z'_l(x) \text{ on } z = 0_-, \quad -\frac{c}{2} \leq x \leq \frac{c}{2} \end{aligned} \tag{21}$$

The singularities describing the foil in the linearised theory are located on the x-axis between $-\frac{c}{2} \leq x \leq \frac{c}{2}$. The question arises as to which singularities should be used to describe the profile. This can be determined by dividing the disturbance potential into even and odd components (Newman, 1978 [13]):

$$\begin{aligned} \phi(x, z) &= \phi_e(x, z) + \phi_o(x, z) \\ \phi_e(x, z) = \phi_e(x, -z) &= \frac{1}{2}[\phi(x, z) + \phi(x, -z)] \\ \phi_o(x, z) = -\phi_o(x, -z) &= \frac{1}{2}[\phi(x, z) - \phi(x, -z)] \end{aligned} \tag{22}$$

The boundary condition at $z = 0\mp$

$$\begin{aligned} \frac{\partial \phi_e}{\partial z} &= \mp\frac{1}{2}U(z_u'(x) - z_l'(x)) \quad \text{on } z = 0_{\pm}, \\ \frac{\partial \phi_o}{\partial z} &= -\frac{1}{2}U(z_u'(x) + z_l'(x)) \quad \text{on } z = 0_{\pm}, \end{aligned} \tag{23}$$

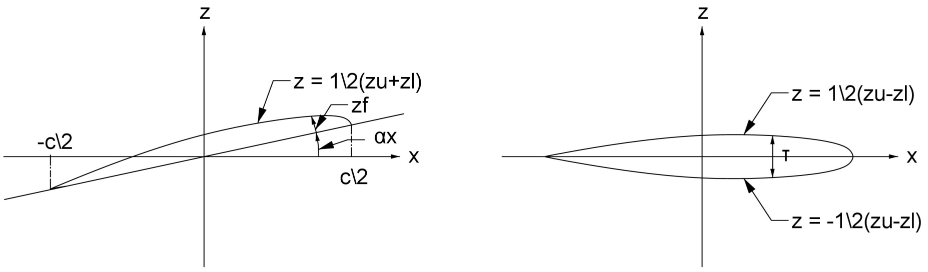


Figure 3: Left: Camber line with angle of attack α . Right: Symmetric section with thickness τ

The operator $\frac{\partial}{\partial z}$ is odd, implying that $\frac{\partial \phi_e}{\partial z}$ is odd and $\frac{\partial \phi_o}{\partial z}$ is even, with respect to z. These even and odd potentials correspond to two distinct physical scenarios. The the odd potential as the potential of an asymmetric flow, passing an arc with zero thickness defined by the curve $z = \frac{1}{2}U(z_u(x) + z_l(x))$, or the mean-camber line. Conversely, even potential as the potential for a symmetrical profile, having thickness $\tau = (z_u(x) - z_l(x))$, at zero angle of attack. Both scenarios are depicted in Figure 3.

By decomposing the original problem into two parts, one representing thickness effects and the other representing camber and angle of attack effects, Each aspect can be addressed separately. Since the pressure distribution is symmetric in the thickness problem, there is no lift force or moment involved. Therefore, thickness does not directly influence lift and moment, but only affects practical considerations when modifications of the pressure distribution, influence separation or cavitation.

The boundary condition for the even potential, as described in Equation (23), reveals an asymmetric vertical velocity along the projection of the profile on the x-axis. This asymmetric velocity arises from a distribution of sources along the projection, as discussed in works such as Breslin and Andersen (1994 [21]). Conversely, the boundary condition for the odd potential, as stated in Equation (23), necessitates a symmetric vertical velocity along the projection. Such symmetry in velocity is achieved through a distribution of vortices,which are crucial for lift generation, as also outlined in Breslin and Andersen (1994 [21]).

This explanation serves as a valuable reference, illustrating how a thin and horizontal profile can be represented by a distribution of sources and vortices, along its projection on the x-axis. As previously mentioned, the thickness is disregarded, and the foil is substituted with a distribution of circulation.

3.7 Circulation

Let's center our analysis on the flow over the mean-camber line and the resultant lift force and moment. The vertical position of the mean-camber line can be conveniently defined as: $z = \frac{1}{2}(z_u(x) + z_l(x)) = \alpha x + z_f(x)$. This establishes the corresponding boundary condition on the cut as :

$$\frac{\partial \phi_o}{\partial z} = -Uz'(x) = -U(\alpha x + z_f'(x)) \tag{24}$$

The boundary condition can be divided into two contributions: one from the angle of attack α and another from the camber line, represented by z_f. It's necessary to know the distribution along the chord. The distribution of circulation related to the angle of attack corresponds to a distribution for a flat plate:

$$\gamma_{FP}(x) = 2U_0\alpha\sqrt{\frac{\frac{c}{2}+x}{\frac{c}{2}-x}} \quad \text{for} \quad \frac{-c}{2} \leq x \leq \frac{c}{2} \tag{25}$$

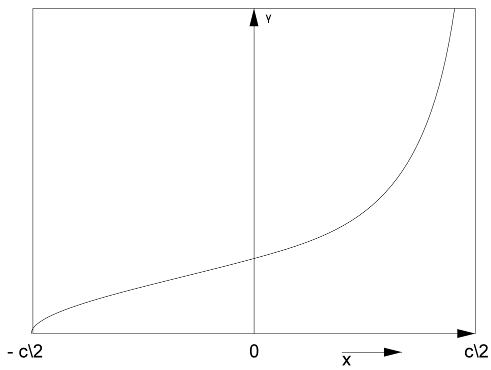


Figure 4: Vortex Distribution for flat plate

From the Equation (25), it's possible to note that the value of circulation is highly intense near the leading edge. As depicted in the Figure 4, the solution is not accurate at this point, but it is accurate for the rest of the profile. Therefore, it is usable.

Regarding the circulation distribution related to the camber line, it is determined by utilizing the linearized Bernoulli equation and a pressure distribution. The tangential velocity $u_x(x, z)$ on both sides of a planar distribution of circulation along the x-axis is given by (Breslin and Andersen [21]):

$$u_x(x, 0_\pm) = \mp\frac{2}{\gamma(x)} \quad \text{for} \quad -\frac{c}{2} \leq x \leq \frac{c}{2} \tag{26}$$

Inserting this in Equation (16) one has:

$$\gamma(x) = \frac{1}{2U_0}\Delta Cp(x) \tag{27}$$

The circulation is known when $\Delta C_p(x)$ is defined. For this purpose, the NACA series is utilized: the specific pressure distribution should be suitable regarding separation and cavitation. The factor 'a' denotes the fraction of the chord, measured from the leading edge, over which the pressure remains constant. Towards the trailing edge, the pressure linearly decreases to zero, creating what is often termed a rooftop pressure distribution.

The circulation distribution for these mean lines is:

$$\gamma_{RT}(x) = \begin{cases} \frac{(c/2+x)U_0}{c(1-a^2)} C_{L,i} & \text{per } -\frac{c}{2} \leq x \leq \frac{c}{2}(1-2a) \\ \frac{U_0}{1+a} C_{L,i} & \text{per } \frac{c}{2}(1-2a) \leq x \leq \frac{c}{2} \end{cases} \tag{28}$$

$C_{L,i}$ is the ideal lift coefficent. By combining the distributions of Equation (25) and Equation (28), the chordwise circulation distribution becomes known, for a profile characterized by an arbitrary rooftop pressure distribution and an arbitrary angle of attack, provided that the limits of the linear theory are respected. Having clarified the linearized thin-wing theory in two dimensions, it can be extended to the three-dimensional theory.

The three-dimensional flow varies also along spanwise direction, affecting circulation which depends on both chordwise and spanwise coordinates, with chordwise variations described by two-dimensional equations. Extending the linearized thin-wing theory to three dimensions is achievable, provided that the two-dimensional assumptions remain valid along the chord.

In steady two-dimensional flow, irrotationality is reached with an infinite vortex downstream of the propeller, compensating for propeller's chordwise circulation. For three-dimensional flow, this requirement is met by closed vortices with constant circulation. Therefore, in steady flow, when a body exhibits a circulation variation along its span, a vortex sheet forms behind it, merging the initially shed vortices with those created at the trailing edge. The circulation of this vortex sheet is given by $\frac{d\Gamma(y)}{dy}$, where $\Gamma(y)$ represents the total chordwise circulation at the spanwise coordinate y. As the vortex sheet is devoid of forces, it must move with the fluid, as per Helmholtz's theorem (a vortex line cannot start or end abruptly in a fluid).

Based on this theory, lifting surfaces such as propellers are modeled with vortex distributions on their surfaces and a wake vortex sheet. Despite the constraints, potential flow theory has been widely used and has proven reliable over time.

3.8 Distribution of Vortex

The method applied in this work divides the blade surface into a number of elements to describe the surface. The calculation is made using vortex segments distributed along the blade to represent its shape.
These vortex segments, form a gridwork that discretely represents the circulation distribution across the blade, by constructing blocks of constant strength vortices.
The wake of the propeller is modeled with a sheet of trailing vortices convected downstream with the mean flow.
Similar to the lifting line model, attention must be paid to how the trailing wake is modeled.

In determining the value of circulation, consideration must be given to the physical nature of the phenomena: the flow has to leave the trailing edge smoothly, satisfying Kutta's conditions:

- The vortex line cannot start or end abruptly,
- The vortex have to svanish into the flow,
- The velocity at trailng edge is finite.

$$\Gamma_{(T.E.)} = 0 \tag{29}$$

The circulation at the trailing edge must be zero, meaning that the pressure must be zero on the trailing edge. In practice, this condition is satisfied when the local streamline and the wake are parallel, which means the strength of the wake is equal to the strength of the panel at the trailing edge. Therefore, the vortex elements cannot end up on the wing but must vanish into the flow, ensuring that no force acts on them.

To satisfy the solution, Helmholtz's theorems are required:

$$\vec{Q} \times \vec{\Gamma}_{wake} = 0 \tag{30}$$

- $\vec{Q}$ is the local flow
- $\vec{\Gamma}_{wake}$ is the circulation of the horseshoe vortex

At any point of the wake, the free vortex must be parallel to the local flow, and to satisfy this condition:

- Each vortex has constant intensity,
- Each vortex can exist only as a closed (ring) line (infinite).

4 Optimisation Procedure

4.1 Introduction

The principle of lifting surfaces is applied to the design of modern propellers to achieve better lift-to-drag ratios. The objective of the optimization procedure is to determine the optimal radial distribution of circulation on the propeller, aiming to maximize efficiency, through the solution of a variational problem. By using this distribution of circulation, the corresponding pitch distribution is found. In other words, the goal is to design the propeller configuration that minimizes the power required to produce the desired thrust, thereby improving overall efficiency.

This methodology was initially developed by Prandtl and Betz in 1927 [1] for a single propeller operating in open water conditions, employing linear theory and a lifting line model with integral formulation. Betz identified the optimal circulation distribution, where the ratio between the pitch angle of the onset flow, and the pitch angle for the total inflow, remained constant.
Lerbs (1952) [2] further advanced the method by introducing a radially varying onset flow. In Lerbs' formulation, an induction factor was incorporated to calculate induced velocity from the trailing vortices, assumed with a helical shape. The shed vortices were aligned with the total flow at the lifting line, coinciding with the criteria established by Betz in the case of open water propellers.

Kerwin et al. (1986) [11], introduced a approach, that discretized the continuous distribution of circulation, allowing for the direct solution of the variational problem. Subsequently, Coney (1992 [12]) developed a vortex-lattice lifting line method, discretizing the continuous distribution of vortices along the lifting line. These advancements showcased significant advantages of the discrete model: linearization is not necessary, and it has the capability to handle theoretically unlimited complex propeller geometries.
In the current optimization procedure, a lifting-surface model is utilized, enabling the integration of the entire blade's effects into the optimization process. Consequently, the optimum distribution of loading is determined, followed by the calculation of the optimum distribution of pitch. For simplicity, the hub is neglected in the optimization procedure.

4.2 Geometry

4.2.1 Propeller Geometry

A brief explanation of the propeller blade geometry is appropriate at this point: the propeller is described in a Cartesian coordinate system which rotates with the propeller. The origin of the coordinate system is at the center of the propeller hub. The x-axis is positive upstream, the y-axis is positive to the port side and the z-axis completes the right-hand coordinate system, see Figure (5).

Consider a propeller comprised of Z identical, symmetrically arranged blades attached to a hub rotating at a constant angular velocity ω about the x-axis. The blade is formed starting with a mid-chord line defined parametrically by the radial distribution of the skew of the mid-chord line of the propeller $\phi_m(s)$, positive in the opposite direction of ϕ and rake $x_m(s)$. By advancing a distance $\pm\frac{1}{2}t$ along a helix of pitch angle $\beta(s)$, one obtains the blade's leading and trailing edges. The reference surface of propeller blade is described in function of arc length parameter along midchord line s, and dimensionless chordwise parameter t, while $c(s)$ is the chord length.

For the cylindrical coordinate system the radius r is positive away from the origin and the angle ϕ is measured from the z-axis and is positive in the same direction as ω. The x-coordinate for the cylindrical coordinate system is the same as for the Cartesian system. The cylindrical system is also shown in Figure (5).
For the blade surface the description in the Cartesian coordinate system:

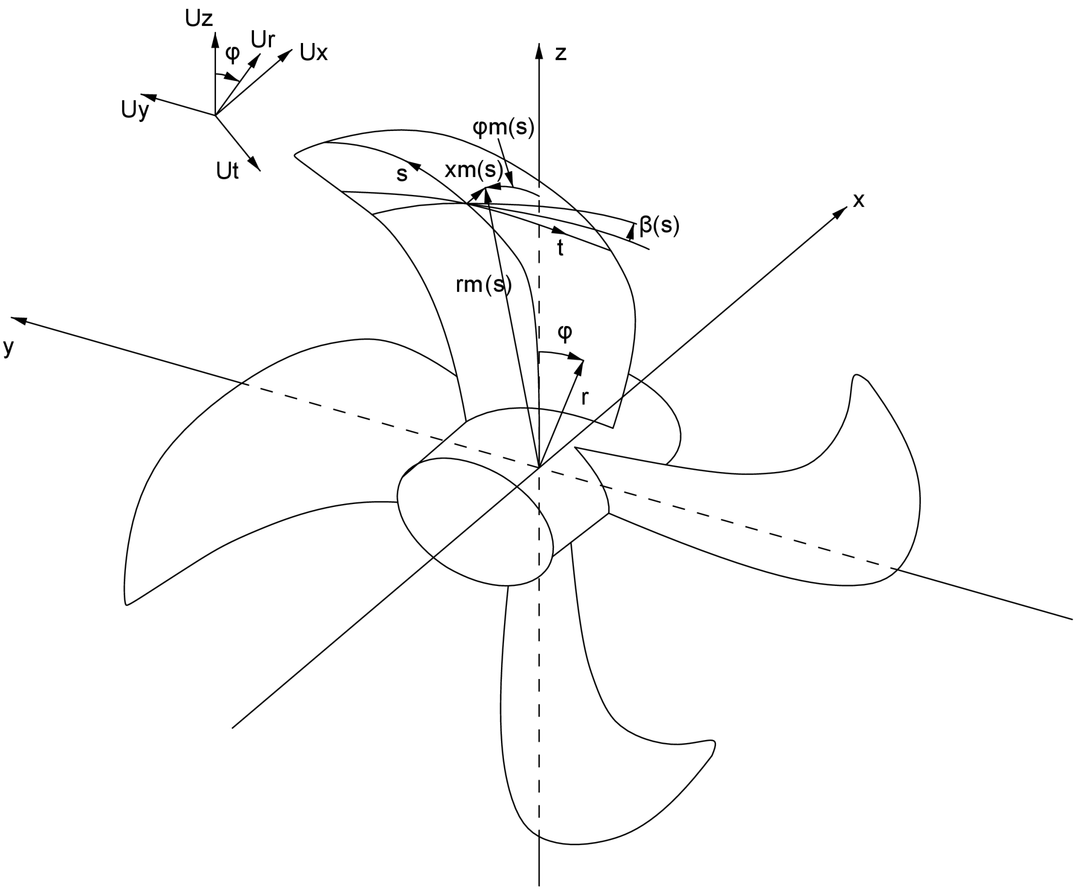


Figure 5: Coordinate system for the propeller

$$\phi(s,t) = -\phi_m(s) + \frac{c(s)}{r_m(s)} cos(\beta(s))t$$

$$\begin{aligned} x(s,t) &= x_m + c(s)sin(\beta(s))t \\ y(s,t) &= -r_m(s)sin(\phi(s,t)) \\ z(s,t) &= r_m(s)cos(\phi(s,t)) \end{aligned}$$

for $s_{hub} < s < s_{tip}$ and $-\frac{1}{2} = t_{(TrailingEdge)} < t < t_{(LeadingEdge)} = \frac{1}{2}$

β is the fluid pitch angle, of the propeller:

$$\beta(s) = tan^{-1}\left(\frac{U_{0,x}(s) - u_x(s)}{\omega r_m(s) - u_t(s) - U_{0,t}(s)}\right) \tag{31}$$

where:

- $U_{0,x}(s)$ is the x component of the onset flow,
- $u_x(s)$ is the total axial induced velocity,
- $r_m(s)$ is the radius for the propeller,
- $u_t(s)$ is the total tangential induced velocity,
- $U_{0,t}(s)$ is the tangential component of the onset flow.

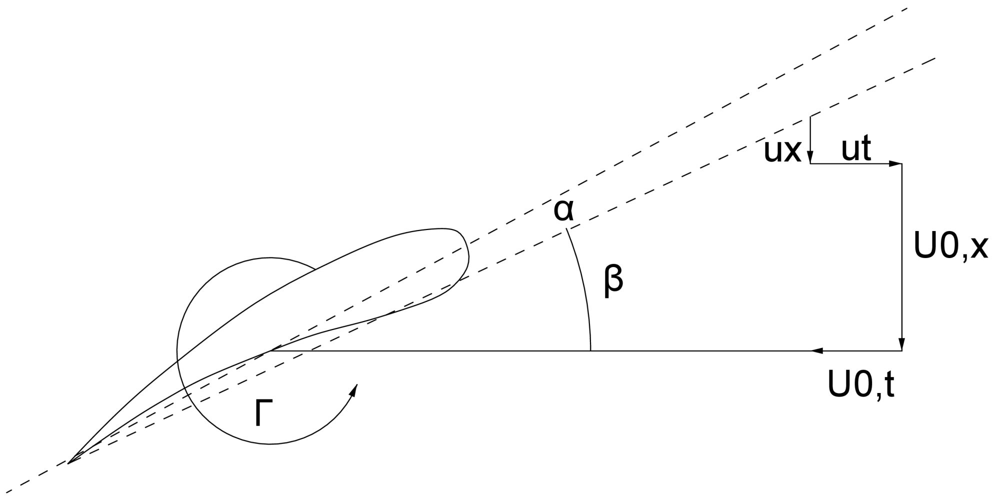


Figure 6: Velocity triangle for the propeller

4.2.2 Grid Generation

As the continuous distribution of circulation is replaced with a discrete distribution, the blade surface, is divided into a number of quadrilateral panels and the trailing vortex sheet is therefore, reduced to a number of trailing horseshoe vortices.

The circulation is positive counterclockwise along the sides of each panel. To satisfy Kelvin's circulation theorem, the circulation along these sides remains constant. The corners of the panels, or grid points, are labeled from $P1$ to $P4$ in the direction of circulation. The vectors along the sides are denoted as $\vec{l_1}$ to $\vec{l_4}$, so the $\vec{l_2}$ representing the vector from $P2$ to $P3$.

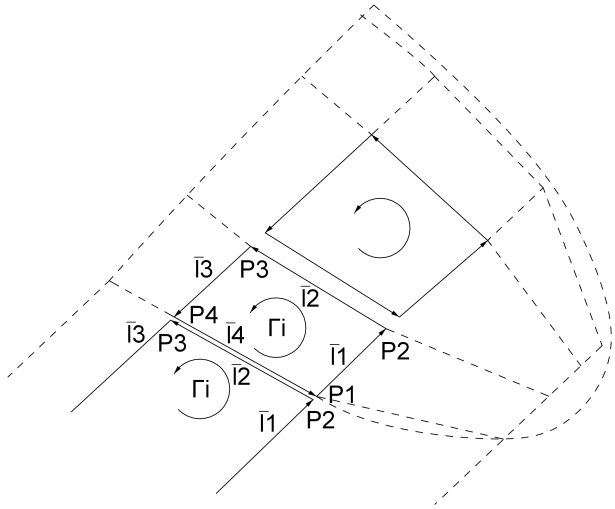


Figure 7: Description of a panel and a trailer

The radial discretization of the propeller follows James (1972) [17]. It's worth noting that the outermost grid points at the tips are shifted inward by one-quarter interval:

$$s_{gp,i} = \frac{4i-3}{4M_{sp}+2}(s_{tip} - s_{hub}) + s_{hub} \qquad \text{for } i = 1, 2, 3..., M_{sp}+1 \tag{32}$$

$$s_{cp,i} = \frac{1}{2}(s_{gp,i} + s_{gp,i+1}) \qquad \text{for } i = 1, 2, 3..., M_{sp} \tag{33}$$

The cosine discretization in the chord-wise direction, following Lan's method [18], is as follows:

$$t_{gp,1} = -\frac{1}{2} \qquad \text{located at T.E.} \tag{34}$$

$$t_{gp,i} = -\frac{1}{2}\cos\left(\frac{(i-\frac{3}{2})\pi}{N_{ch}}\right) \qquad \text{for } i = 2, 3..., N_{ch}+1 \tag{35}$$

$$t_{cp,i} = \frac{1}{2}(t_{gp,i} + t_{gp,i+1}) \qquad \text{for } i = 1, 2, 3..., N_{ch} \tag{36}$$

where N_{ch} is the number of chord-wise panels, M_{sp} is the number of span-wise panels, *gp* refers to grid points, *cp* refers to control points.

4.2.3 Horseshoe Vortex

As previously mentioned, the discretization process reduces the trailing vortex sheet to a finite number of horseshoe vortices, providing a simplified representation of the wing's vortex system. Each horseshoe vortex comprises two trailing wing-tip vortices, which extend infinitely downstream with the fluid flow, and a bound vortex, represented as a straight line positioned at the trailing edge. The wing-tip vortices contribute to the downwash component responsible for induced drag.

To satisfy the Kutta condition, the circulation of the horseshoe vortex equals the circulation of the adjacent trailing edge panel. For the propeller, it's assumed that the sides of the horseshoe, follow regular helices with constant pitch and radius.

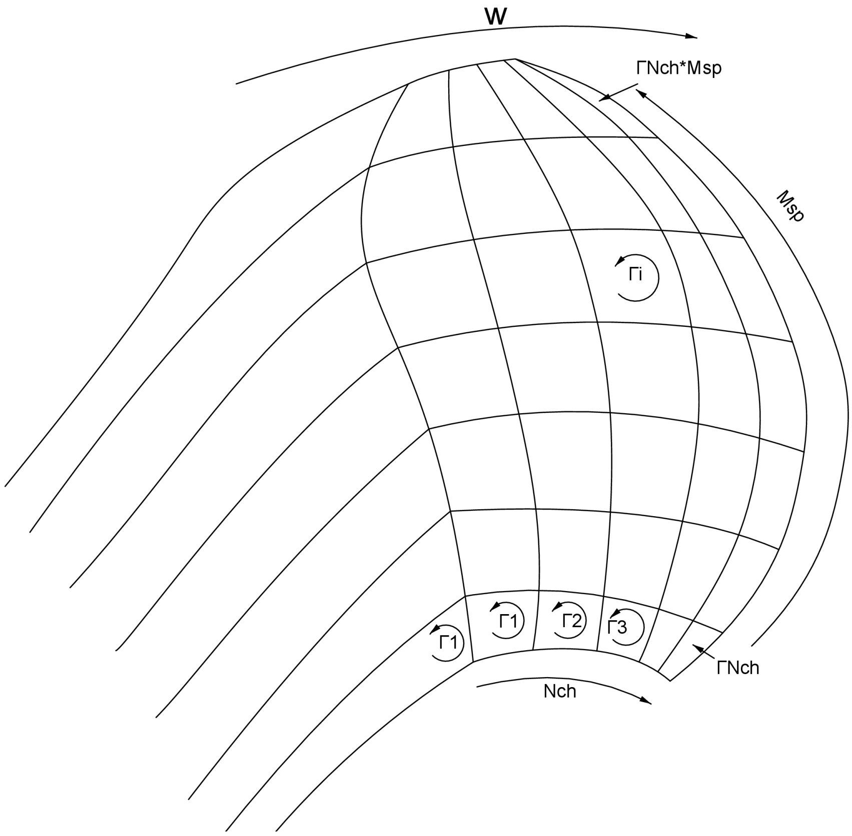


Figure 8: Example of grid, trailers and direction of the circulation for the propeller

Consequently, the horseshoe vortex can be described by:

$$\vec{x} = \begin{cases} x & -\infty < x < x_{(T.E.)} \\ -rsin\left(\frac{2\pi}{P}(x - x_{(T.E.)}) + \phi_{(T.E.)}\right) & y_{(T.E_{(Hub)})} < y < y_{(T.E_{(Tip)})} \\ rcos\left(\frac{2\pi}{P}(x - x_{(T.E.)}) + \phi_{(T.E.)}\right) & r_{(T.E_{(Hub)})} < r < r_{(T.E_{(Tip)})} \end{cases} \tag{37}$$

where:

- r is the radius of the grid points at the trailing edge,
- P is the pitch of the helix, which is equal to the pitch of the reference flow (see section 4.5): $P = 2\pi r \tan(\beta)$
- $\phi_{(T.E.)}$ is the phase angle of the helix,
- $x_{(T.E.)}$ is the x at the trailing edge.

4.3 Forces and Velocities Calculations

4.3.1 Force on the panel sides

The force on the panel sides is found by using the Kutta–Joukowski theorem:

$$\vec{F}_{Side} = \rho \vec{U}(\vec{x}) \times \vec{\Gamma}_{Side} \tag{38}$$

where:

- $\vec{\Gamma}_{Side}$ is the total circulation of the panel side, which is the difference in circulation for the two adjacent panels (for the leading edge panel is equal to $\vec{\Gamma}_{Panel}$),
- $\vec{U}(\vec{x})$ is the total velocity at the midpoint of the panel side. $\vec{U}(\vec{x}) = \vec{U}_0(\vec{x}) + \vec{u}(\vec{x})$

where:

- $\vec{U}_0(\vec{x})$ is the onset flow at the midpoint of the side,
- $\vec{u}(\vec{x})$ is the total induced velocity at the midpoint of the side.

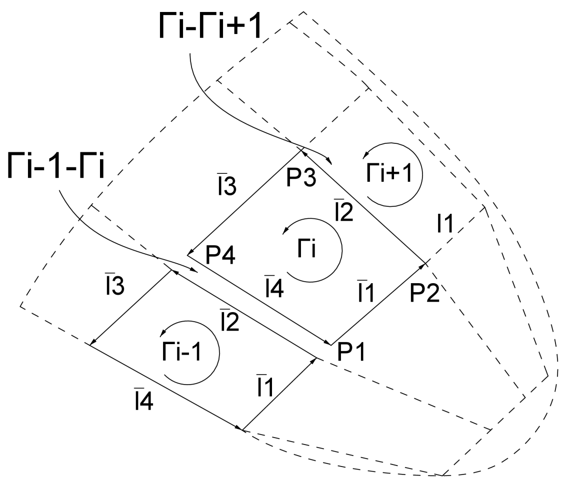


Figure 9: Description of the total circulation at the panel side

4.3.2 Onset Flow

The undisturbed flow is given in cylindrical coordinates, allowing for its rearrangement into Cartesian coordinates. It is assumed that the undisturbed flow depends solely on radial variation and is independent of longitudinal position; furthermore, it is assumed to be axi-symmetric. Therefore, the three Cartesian components can be expressed as follows:

$$\begin{aligned}\vec{U_0}(\vec{x}) = (-U_{0,x}(s), -U_{0,r}(s)\sin\phi - (U_{0,t}(s) - \omega r(s))cos\phi,\\ U_{0,r}(s)\cos\phi - (U_{0,t}(s) - \omega r(s))sin\phi)\end{aligned} \tag{39}$$

where:

- $U_{0,x}(s)$,$U_{0,z}(s)$,$U_{0,y}(s)$ are the wake velocities given in Cartesian coordinates,
- $U_{0,r}(s)$,$U_{0,t}(s)$ are the wake velocities given in cylindrical coordinates,
- $\omega r(s)$ is the tangential velocity caused by the rotation of the propeller(included because the coordinate system is fixed to the blade).

4.3.3 Induced velocities from the panels

The induced velocity from the panels is determined by applying the Biot-Savart law. This law is a general result of potential theory and describes both electromagnetic fields and inviscid, incompressible flows. In general terms the law can be stated (see Figure 10) as the velocity du induced at a point x of radius R from a segment $d\varepsilon$ of a vortex filament of strength Γ given by:

$$d\vec{u} = \frac{\Gamma}{4\pi}\frac{d\vec{\varepsilon} \times \vec{R}}{|\vec{R}|^3} \tag{40}$$

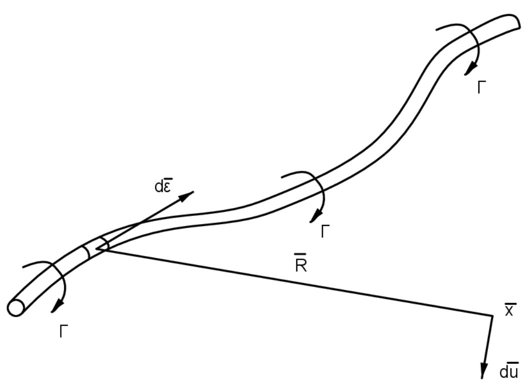


Figure 10: Application of the Biot–Savart law to a general vortex filament

To rearrange the expression for calculating the velocity induced by a single panel at the point $\vec{x}$, it can be expressed as follows:

$$\vec{u}_i(\vec{x}) = \frac{\Gamma_i}{4\pi}\sum_{k=1}^{4}\int_0^{s_k}\frac{d\vec{\varepsilon} \times \vec{R}}{\mid \vec{R} \mid^3} = \Gamma_i q_i(\vec{x}) \tag{41}$$

where Γ_i is the circulation of the panel, $d\vec{\varepsilon}$ is the length element along the panel side with the lenght s_k. $\vec{R}$ is the vector from the vortex element, q_i is defined as the velocity induced by the entire panel with a unit circulation.The numerical evaluation of the induced velocity involves the determination of the velocity induced by a unit circulation since at first the circulation is an unknown.

Considering that the panel sides are linear segments, the computational assessment of the induced velocity resulting from a panel side adheres to the methodology outlined by Olsen (2001) [4]:

$$\vec{u}(\vec{x}) = \frac{\Gamma}{4\pi}\frac{\vec{a} \times \vec{c}}{\mid \vec{a} \times \vec{c} \mid}\frac{1}{d}[\cos\alpha + \cos\beta] = \frac{\Gamma}{4\pi}\frac{\vec{a} \times \vec{c}}{\mid \vec{a} \times \vec{c} \mid}\frac{1}{d}\left[\frac{a-e}{b} + \frac{e}{c}\right]$$

The vector $\frac{(a \times c)}{|a \times c|}$ corresponds to a unit vector giving the direction of the induced velocity.

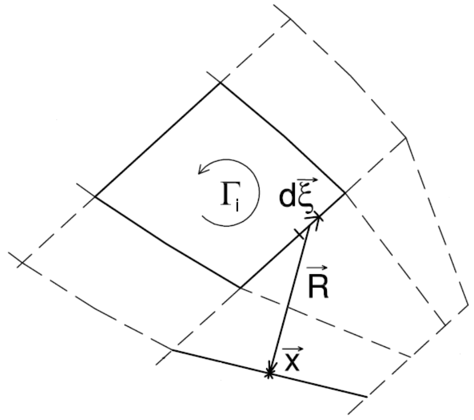


Figure 11: Description of the parameters used in the application of Biot-Savart law

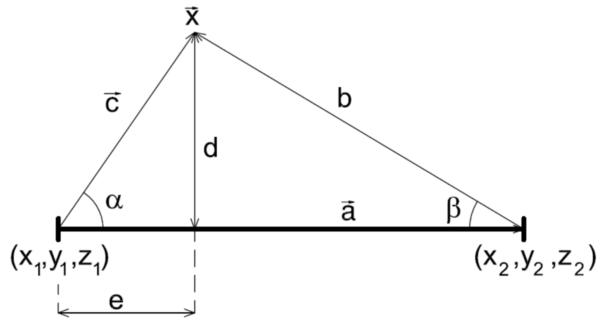


Figure 12: Parameters used to evaluate the induced velocity from a straight vortex

where:

- $a =\mid \vec{a} \mid= \sqrt{(x_2 - x_1) + (y_2 - y_1) + (z_2 - z_1)}$,
- $b = \sqrt{(x_2 - x) + (y_2 - y) + (z_2 - z)}$,
- $c = \sqrt{(x_1 - x) + (y_1 - y_1) + (z_1 - z)}$,
- $d = \sqrt{(c^2 - e^2)}$,
- $e =\mid \vec{e} \mid= \frac{a^2+c^2-b^2}{2a}$

4.3.4 Induced velocities from the horseshoe vortices

The induced velocity from the horseshoe vortices is divided into two parts:

- Transition wake:
  - Extends from the trailing edge of the propeller to four radii downstream.
  - The regular helix is approximated by a series of straight line vortices.
  - The induced velocity can be determined using the same method as for the panel sides.
- Ultimate wake:

– Includes the region from the end of the transition wake to infinitely downstream.
– The induced velocity in this region is calculated using the method developed by de Jong (1991) [20].

4.3.5 Total velocity

The total velocity at point $\vec{x}$ is the sum of the onset flow and the induced velocity from the propeller itself:

$$\vec{U}(\vec{x}) = \sum_{j=1}^{M_{sp}} \Gamma_{1+(j-1)N_{ch}} \sum_{i=1}^{N_{ch}} k_i \vec{q}_{i+(j-1)N_{ch}}(\vec{x}) + \vec{U}_0 \tag{42}$$

where:

- j is the counter for the span-wise panels,
- i is the counter for the chord-wise panels,
- $\Gamma_{1+(j-1)N_{ch}}$ is the circulation for the panel at the trailing edge,
- k_i is the weight function,
- $\vec{q}_{i+(j-1)N_{ch}}$ is the induced velocity from the panel $i + (j-1)N_{ch}$ with a unit circulation. The induced velocities from the trailing vortices are included in $\vec{q}_{i+(j-1)N_{ch}}$
- $\vec{U}_0$ is the onset flow.

4.4 Weight Function

Munk's displacement theorem states that the induced drag for a lifting surface depends solely on the total chord-wise circulation and not on the chord-wise distribution of the circulation. Hence, to specify the chord-wise distribution of circulation for the propeller, it becomes necessary to introduce the weight function. Essentially, the weight function establishes a relationship between the chord-wise panels to determine the chord-wise distribution of circulation for the propeller. For the propeller, the optimization problem is therefore simplified to finding the optimal distribution of total circulation for each chord-wise strip, which corresponds to the circulation of the horseshoe vortex.

In a discrete distribution of vortices, as depicted in Figure 13, the weight function is defined as follows:

$$\kappa_n = \frac{\Gamma_n}{\Gamma_{tot}} \tag{43}$$

The total circulation at grid point n, Γ_n, as shown in Figure 13, is the difference between the circulation of two adjacent panels. Meanwhile, the total circulation for the chordwise direction is given by: $\Gamma_{\text{tot}} = \int_{-c/2}^{c/2} \gamma(x), dx$, where γ represents the continuous distribution of circulation calculated earlier in Section 3.7, as a combination of the flat plate and rooftop distributions.

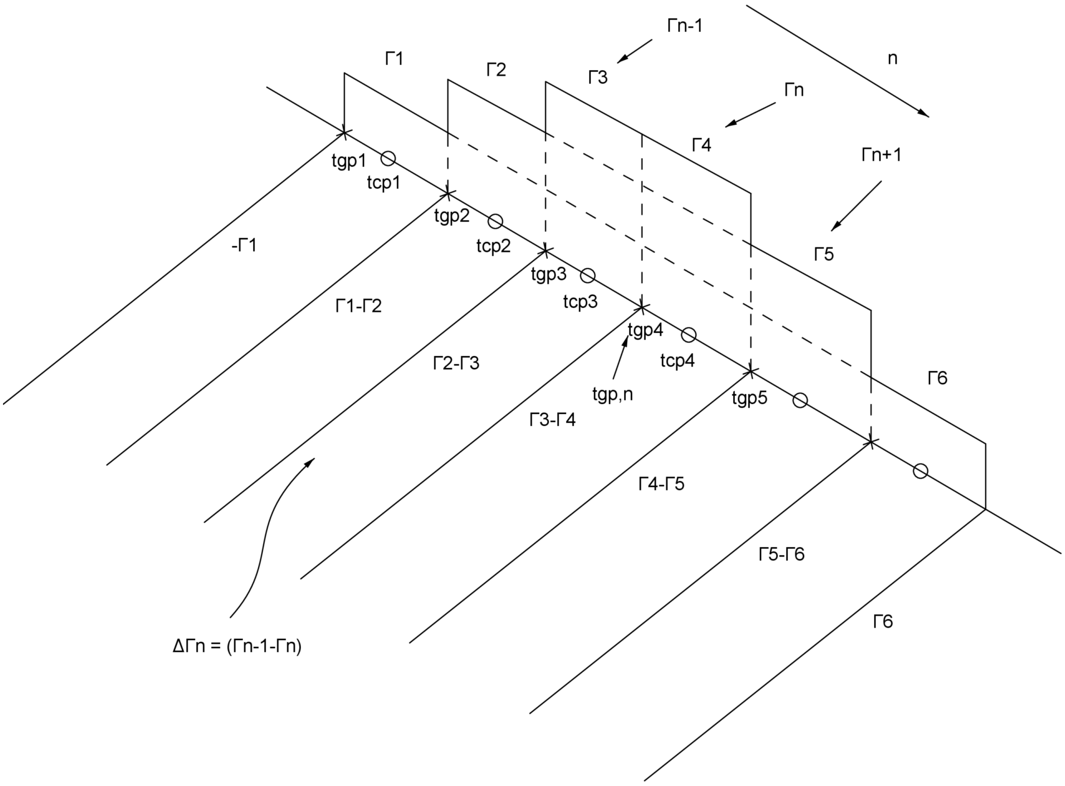


Figure 13: Description of total circulation at a panel side

For the discrete vortex, the cirulation can be approximated by:

$$\Gamma_n = c \int_{t_{cp,n-1}}^{t_{cp,n}} \gamma(t'), t' \approx \gamma(t_{gp,n})(t_{cp,n} - t_{cp,n-1}) \tag{44}$$

where $t_{cp,n}$ follows Lan (1974) [18], and represents the location of the control point, while $t_{gp,n}$ is described in Section 4.2.2.
The discrete circulation becomes:

$$\Gamma_n \approx \gamma(t_{gp,n}) C \sqrt{(\frac{1}{2} - t_{gp,n})(\frac{1}{2} + t_{gp,n})} \tag{45}$$

where C is a constant. Then, it's possible to write the relationship between the weight function κ and the circulation on the chord-wise panels as follows:

$$\begin{cases} \kappa_1 = 0 \\ \kappa_n = \frac{\Gamma_{n-1} - \Gamma_n}{\Gamma_{tot}} \qquad \text{for } i = 2, 3..., N_{ch} \\ \kappa_{N_{Ch+1}} = \frac{\Gamma_{N_{Ch+1}}}{\Gamma_{tot}} \end{cases} \tag{46}$$

This results in the weight function for the circulation of the panels:

$$\kappa_n = \sum_{i=n+1}^{N_{Ch+1}} ((1-\nu)\kappa_i^{RT} + \nu\kappa_i^{FT}) \qquad \text{for } i = 1, 2, 3..., N_{ch} \tag{47}$$

where:

- Γ_{tot} is the total circulation for the chord-wise distribution,
- κ_i^{RT} is the weight function for the flat plate pressure distribution ,
- κ_i^{FT},is the weight function for the rooftop plate pressure distribution,
- ν is the ratio of the pressure distribution. In our case $\nu = 0.5$.

4.5 Wake Alignment

The applied grid and wake alignment procedure assumes a constant pitch for the horseshoe vortices and disregards slipstream contraction. It also assumes that the blade and horseshoe vortices share the same pitch, determined by the total velocity at the midchord line of the blade. The pitch of the helix is based on the total velocity at the mid-chord line of the blade, which is located at $t = 0$. Consequently, the pitch angle of both the grid and horseshoe vortices is:

$$\beta_i(s) = tan^{-1}\left(\frac{U_{0,x}(s) - u_x(s)}{\omega r_m - u_t(s) - U_{0,t}(s)}\right) \tag{48}$$

where:

- $U_{0,x}(s)$ is the x component of the onset flow,
- $u_x(s)$ is the total axial induced velocity,
- ω is the angular velocity,
- r_m is the radius for the propeller,
- $u_t(s)$ is the total tangential induced velocity,
- $U_{0,t}(s)$ is the tangential component of the onset flow.

The applied alignment procedure corresponds to the wake alignment used in the moderately loaded lifting-line theory. However, unlike the lifting-line theory, the induced velocity from the bound vortices is included in the total induced velocity for the lifting-surface optimization. While it's assumed that the effects of these vortices are small, which holds true for a propeller without skew and rake, for a skewed propeller, this assumption becomes more questionable.

4.6 Thrust and Torque Calculation

As previously discussed, the forces on the propeller blades are found by the Kutta–Joukowski theorem. Therefore, the force on one side of the panel is calculated using the following expression:

$$\vec{F}_{Side} = \rho\vec{U}(\vec{x}) \times \vec{\Gamma}_{Side} = \rho\Gamma_{Side}\left(\vec{U}(\vec{x}) \times \vec{l}_{Side}\right) \tag{49}$$

where:

- $\vec{U}(\vec{x})$ is the total velocity calculated at the midpoint of the panel side,
- $\vec{l}_{Side}$ is the vector for the side,
- $\vec{\Gamma}_{Side}$ is the total circulation on the side.

The moment generated by one side of the panel can be expressed ass:

$$\vec{M}_{Side} = \vec{r}(\vec{x}) \times \vec{F}_{Side} \tag{50}$$

where:

- $\vec{r}(\vec{x})$ is the vector from the origin of the coordinate system to the midpoint of the side.

The total thrust, T, and torque Q, generated by the propeller are determined by summing the contributions from all the panel sides of all the blades. It's important to note that, due to the symmetric nature of the propeller and its operation under steady conditions, the forces generated by all the blades are identical. Therefore, the forces generated by the entire propeller can be calculated by multiplying the forces on one blade (the reference blade) by the number of blades Z.
Therefore, the thrust (x-component of the total force) is:

$$\begin{aligned} T = F_x =& \rho Z \sum_{m=1}^{M_{sp}} \Gamma_{1+(m-1)N_{ch}} \Bigg\{ \sum_{n=1}^{N_{ch}} \kappa_n \sum_{k=1}^{4} [l_{z,n+(m-1)N_{ch,k}} U_y(\vec{x}_{n+(m-1)N_{ch,k}}) \\ & - l_{y,n+(m-1)N_{ch,k}} U_z(\vec{x}_{n+(m-1)N_{ch,k}})] - l_{z,1+(m-1)N_{ch,4}} U_y(\vec{x}_{1+(m-1)N_{ch,4}}) \\ & + l_{y,1+(m-1)N_{ch,4}} U_z(\vec{x}_{1+(m-1)N_{ch,4}}) \Bigg\} \end{aligned} \tag{51}$$

where:

- l_x is the x-component of $\vec{l}$,
- l_y is the y-component of $\vec{l}$,
- l_z is the z-component of $\vec{l}$,
- U_x is the x-component of the total velocity,
- U_y is the y-component of the total velocity,
- U_z is the z-component of the total velocity,

- m is the span-wise index,
- n is the chord-wise index,
- k is the side index.

For example, $\vec{x}_{n+(m-1)N_{ch,k}}$ is the coordinate for the midpoint of side k of the panel number $n + (m-1)N_{ch,k}$.

The torque Q is the negative x-component of the total moment:

$$Q = -M_x = -\sum_{i=1}^{sides} (yF_z - zF_y) = Q_2 - Q_1 \tag{52}$$

where:

$$\begin{aligned} F_y = \rho Z \sum_{m=1}^{M_{sp}} \Gamma_{1+(m-1)N_{ch}} \Bigg\{ \sum_{n=1}^{N_{ch}} \kappa_n \sum_{k=1}^{4} [l_{x,n+(m-1)N_{ch,k}} U_z(\vec{x}_{n+(m-1)N_{ch,k}}) \\ - l_{z,n+(m-1)N_{ch,k}} U_x(\vec{x}_{n+(m-1)N_{ch,k}})] - l_{x,1+(m-1)N_{ch,4}} U_z(\vec{x}_{1+(m-1)N_{ch,4}}) \\ + l_{z,1+(m-1)N_{ch,4}} U_x(\vec{x}_{1+(m-1)N_{ch,4}}) \Bigg\} \end{aligned} \tag{53}$$

$$\begin{aligned} F_z = \rho Z \sum_{m=1}^{M_{sp}} \Gamma_{1+(m-1)N_{ch}} \Bigg\{ \sum_{n=1}^{N_{ch}} \kappa_n \sum_{k=1}^{4} [l_{y,n+(m-1)N_{ch,k}} U_x(\vec{x}_{n+(m-1)N_{ch,k}}) \\ - l_{x,n+(m-1)N_{ch,k}} U_y(\vec{x}_{n+(m-1)N_{ch,k}})] - l_{y,1+(m-1)N_{ch,4}} U_x(\vec{x}_{1+(m-1)N_{ch,4}}) \\ + l_{x,1+(m-1)N_{ch,4}} U_y(\vec{x}_{1+(m-1)N_{ch,4}}) \Bigg\} \end{aligned} \tag{54}$$

$$\begin{aligned} Q_1 = yF_z = \rho Z_P \sum_{m=1}^{M_{sp}} \Gamma_{1+(m-1)N_{ch}} \Bigg\{ \sum_{n=1}^{N_{ch}} \kappa_n \sum_{k=1}^{4} y_{n+(m-1)N_{ch,k}} [l_{y,n+(m-1)N_{ch,k}} U_x(\vec{x}_{n+(m-1)N_{ch,k}}) \\ - l_{x,n+(m-1)N_{ch,k}} U_y(\vec{x}_{n+(m-1)N_{ch,k}})] + y_{1+(m-1)N_{ch,4}} [-l_{y,1+(m-1)N_{ch,4}} U_x(\vec{x}_{1+(m-1)N_{ch,4}}) \\ + l_{x,1+(m-1)N_{ch,4}} U_y(\vec{x}_{1+(m-1)N_{ch,4}})] \Bigg\} \end{aligned} \tag{55}$$

$$\begin{aligned} Q_2 = zF_y = \rho Z_P \sum_{m=1}^{M_{sp}} \Gamma_{1+(m-1)N_{ch}} \Bigg\{ \sum_{n=1}^{N_{ch}} \kappa_n \sum_{k=1}^{4} z_{n+(m-1)N_{ch,k}} [l_{x,n+(m-1)N_{ch,k}} U_z(\vec{x}_{n+(m-1)N_{ch,k}}) \\ - l_{z,n+(m-1)N_{ch,k}} U_x(\vec{x}_{n+(m-1)N_{ch,k}})] + z_{1+(m-1)N_{ch,4}} [-l_{x,1+(m-1)N_{ch,4}} U_z(\vec{x}_{1+(m-1)N_{ch,4}}) \\ + l_{z,1+(m-1)N_{ch,4}} U_x(\vec{x}_{1+(m-1)N_{ch,4}})] \Bigg\} \end{aligned} \tag{56}$$

- F_y is the y-component of the force on side i,

- F_z is the z-component of the force on side i,
- Q_1 and Q_2 are introduced in order to make the expression above more readable.

The equations above satisfy the Kutta condition ($\Gamma_{(T.E.)} = 0$). Let's consider, for example, Equation (47) and its last two components. This part of the equation is employed to eliminate the contribution of segments at the trailing edge to the thrust, which was previously calculated in the same equation:

$$l_{x,1+(m-1)N_{ch,4}} U_z(\vec{x}_{1+(m-1)N_{ch,4}}) + l_{z,1+(m-1)N_{ch,4}} U_x(\vec{x}_{1+(m-1)N_{ch,4}}) \tag{57}$$

4.7 Optimum Circulation Distribution

As mentioned earlier, the objective of the optimization procedure is to determine the radial distribution of circulation on the propeller. This distribution allows the propeller to generate a specified thrust with minimal energy consumption. Consequently, minimizing the torque applied to the propeller becomes crucial. In essence, the objective is to achieve the highest efficiency for the propeller:

$$\eta = \frac{J}{2\pi} \frac{K_T}{K_Q} \tag{58}$$

where:

K_T is the thrust coefficient:

$$K_T = \frac{T_r}{\rho n^2 D^4} \tag{59}$$

K_Q is the torque coefficient:

$$K_Q = \frac{Q_t}{\rho n^2 D^5} \tag{60}$$

J is the advance number:

$$J = \frac{U_a}{nD} \tag{61}$$

T_r is the required thrust of the propeller : $\quad T_r = T_t - T_v$

- T_t is the total required thrust of the propeller
- T_v is the thrust owed to the skin friction drag of the propeller (negative).

U_a is the mean inflow to the propeller disc, ρ is the mass density of the water, n is the rate of revolution of the propeller, D is the propeller's diameter and Q_t is the total torque:

$$Q_t = (Q_2 - Q_1) + Q_v \tag{62}$$

where Q_v is the torque owed to the skin friction drag of the propeller (negative).

4.7.1 Skin Friction Drag

Skin friction drag is the portion of drag resulting from the friction between a fluid and the surface, of an object, moving through it. This drag arises within the boundary layer, due to the viscosity of the fluid. It is directly proportional to the surface area in contact with the fluid and increases with the square of the velocity. Additionally, it's important to note that form drag is disregarded in this context due to the vortex-lattice method's reliance on potential flow theory: The skin friction drag created by a panel of the propeller is:

$$dT_v = \frac{1}{2}\rho C_f A |V_T| V_T \tag{63}$$

$$dQ_v = \frac{1}{2}\rho C_f A |V_T| (y_P V_{T_z} - z_P V_{T_y}) \tag{64}$$

where:

- $C_f = 2 \cdot 0.004$ is the frictional drag coefficient for the two faces of the panel,
- A is the area of the panel,
- V_T is the total tangential velocity in the control point of the panel,
- y_P is the y-coordinate of the control point of the panel,
- z_P is the z-coordinate of the control point of the panel,
- V_{T_z} is the z-coordinate of the tangential velocity in the control point of the panel,
- V_{T_y} is the y-coordinate of the tangential velocity in the control point of the panel.

4.7.2 Variational Problem

The optimization procedure aims to determine the circulation distribution that enables the propeller to achieve a specified thrust while minimizing energy consumption. Therefore, the torque applied to the propeller should be minimized as well. This circulation distribution is obtained by solving a discrete variational problem, as outlined in Kerwin et al. (1986) [3].

The functional for this problem is given by:

$$H(\vec{\Gamma}, \lambda) = Q(\vec{\Gamma}) + \lambda(T(\vec{\Gamma}) - (T_r - T_v)) \tag{65}$$

where:

- $\vec{\Gamma}$ is the sought distribution of circulation,
- λ is the Lagrange multiplier,

Since the circulation on the blade is determined by the weight function and the circulation of the trailing vortices, the number of unknown circulations corresponds to the number of radial panels M_{sp}. The optimum distribution is that which minimises the functional H. Thus, the distribution can be found by setting the partial derivatives of $H(\vec{\Gamma}, \lambda)$ with respect to $\vec{\Gamma}$ and λ equal to zero.

This gives the following system of equations:

$$\begin{cases} \dfrac{\partial H}{\partial \Gamma_{1+(m-1)N_{ch}}} = \dfrac{\partial Q}{\partial \Gamma_{1+(m-1)N_{ch}}} + \lambda \dfrac{\partial T}{\partial \Gamma_{1+(m-1)N_{ch}}} = 0 \\ \dfrac{\partial H}{\partial \lambda} = T - (T_r - T_v) = 0 \end{cases} \tag{66}$$

The provided equations demonstrate that the optimization procedure is nonlinear. This nonlinearity arises due to the presence of products involving λ and $\vec{\Gamma}$, as well as the dependency of induced velocities on circulation. Therefore, the non-linear problem is linearised, which results in the following system of equations:

$$\begin{cases} \dfrac{\partial Q(\vec{\Gamma})}{\partial \Gamma_{1+(m-1)M_{sp}}} + \lambda^{t-1} \dfrac{\partial T(\vec{\Gamma})}{\partial \Gamma_{1+(m-1)M_{sp}}} + \lambda^{t} \dfrac{\partial T(U_0)}{\partial \Gamma_{1+(m-1)M_{sp}}} = - \dfrac{\partial Q(U_0)}{\partial \Gamma_{1+(m-1)M_{sp}}} \\ T(\vec{\Gamma}) = (T_r - T_v) - T(U_0) \qquad \text{for } m = 1, 2, ...M_{sp} \\ \end{cases} \tag{67}$$

where:

- $Q(\vec{\Gamma})$ refers to the parts of Q that are functions of the circulation,
- $T(\vec{\Gamma})$ refers to the parts of T that are functions of the circulation,
- $Q(U_0)$ refers to the parts of Q that are functions of the onset flow,
- $T(U_0)$ refers to the parts of T that are functions of the onset flow,
- t is the value of the current iteration,
- $t-1$ is the value from the previous iteration.

The primary challenge in determining the circulation distribution stems from the direct and indirect dependencies of thrust and torque on circulation. For instance, considering the thrust generated by one side:

$$\vec{T} = \rho\, \vec{U}(\vec{\Gamma}) \times \vec{\Gamma} \tag{68}$$

The equation above clearly illustrates the double dependence on circulation. Therefore, it becomes necessary to employ an iterative method for solving the variational problem in order to determine the radial distribution of circulation on the propeller.

As mentioned earlier, iterations are required to attain a solution to the problem. These iterations continue until the residual R^t falls below a certain limit.

$$R^t = \text{Max}\left(\left| 1 - \frac{\Gamma^{(t)}_{1+(m-1)M_{sp}}}{\Gamma^{(t-1)}_{1+(m-1)M_{sp}}} \right| \right) \qquad \text{for } m = 1, 2, ...M_{sp} \tag{69}$$

4.7.3 Optimisation Procedure

First of all, input parameters have to be specified in order to begin the optimisation procedure. These include:

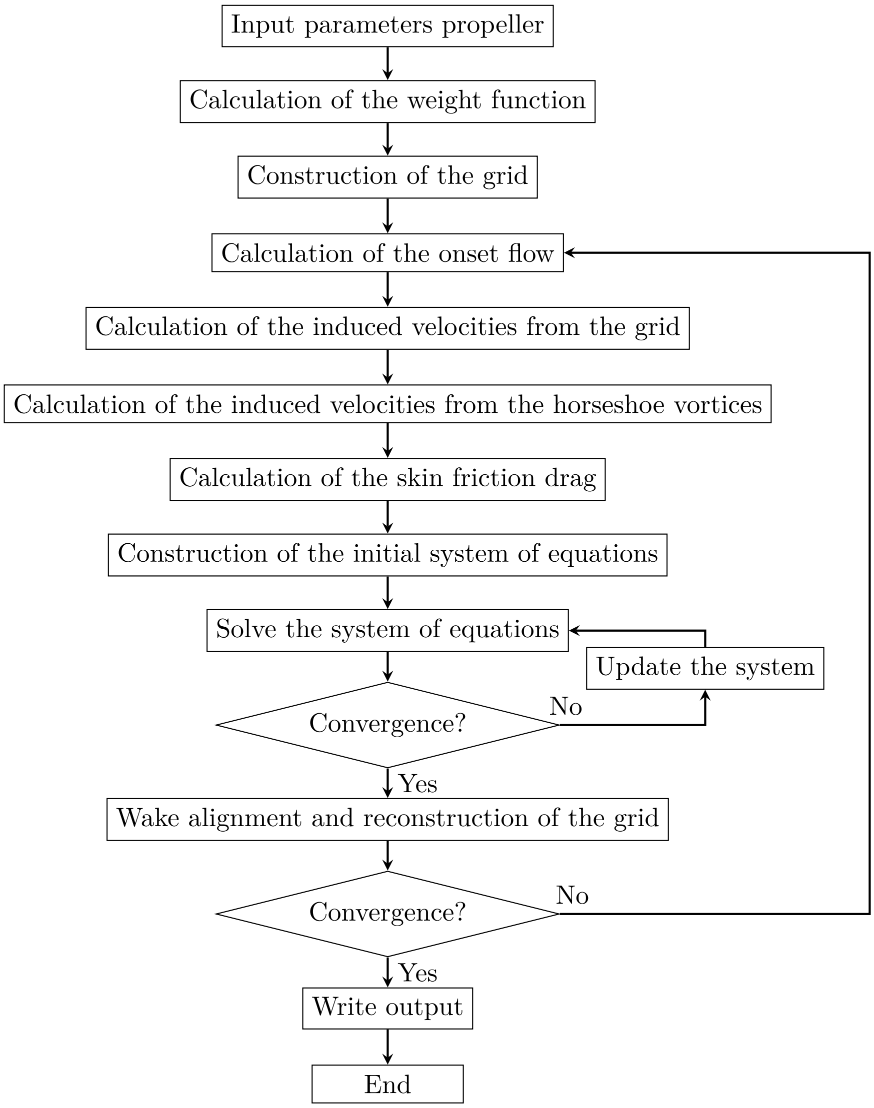
- main dimensions of the propeller (radius of the propeller, radius of the hub, number of blades, etc.),
- size of the grid (number of span-wise panels, number of chord-wise panels, etc.),
- geometry of the mid-chord line, which is specified through the distributions of radius, rake and skew. It is bear mentioning that these three parameters are function of the arc length parameter s,
- chord length distribution in order to construct the grid.
- design point (advance number, required thrust, etc.),
- onset flow,
- ratio between the flat plate and the rooftop distributions ν.

Given the initial input, the initial system of equations for the variational problem is constructed, according to Equation (67). Initially, the distribution of circulation is set to zero, and the Lagrange multiplier is set to -1 (Coney, 1992) [22]. The iteration for the variational problem continues until the residual, as described in Equation (69), falls below 10^{-5}, typically achieved in fewer than ten iterations. Once the variational problem has converged, the grid and the trailers are aligned according to Equation (48). Subsequently, the system of equations for the variational problem is updated with the new grid and wake geometry, and the variational problem is solved again. The alignment of the grid and the wake continues until the residual for the pitch distribution of the wake is less than 10^{-5}:

$$R^t_{align} = \text{Max}\left(\left|1 - \frac{P_m^{(t)}}{P_m^{(t-1)}}\right|\right) \qquad \text{for } m = 1, 2, ...M_{sp} + 1 \tag{70}$$

The number of iterations required for the wake alignment to converge varies based on the propeller geometry and loading, but generally, convergence is slower than for the variational problem. Once the wake alignment has converged, the distribution of circulation is saved to a file, and the program terminates.

The flow chart below, clearly shows the optimisation procedure for the propeller:



It is crucial to obtain the optimal distribution of circulation without altering either the wake or the grid. This necessity arises from the fact that aligning the wake for each iteration of the optimization procedure results in a heavily tip-loaded propeller, as demonstrated by Kerwin (1986) [3]. Consequently, during the circulation optimization procedure, the induced velocities remain fixed, as they are solely functions of the propeller's geometry.

5 Validation

The validation process of the computer program utilized the DTNSRDC propeller series, with additional information on the series available in Kerwin and Lee's work (1978) [29]. Subsequently, the validation results were compared with findings from Olsen (2001) [4]. For this comparison, four propellers from the series were selected. While these propellers share identical radial distribution of circulation, expanded blade area, and thickness distribution, variations in skew were introduced among them. Consequently, differences in pitch were observed.
The main dimensions and design points for the propellers are as follows:

$$Z = 5; R = 3.0m; \rho = 0.2; A_E/A_0 = 0.725; J = 0.889; K_{T,D} = 0.2055; C_{Th} = 0.662. \tag{71}$$

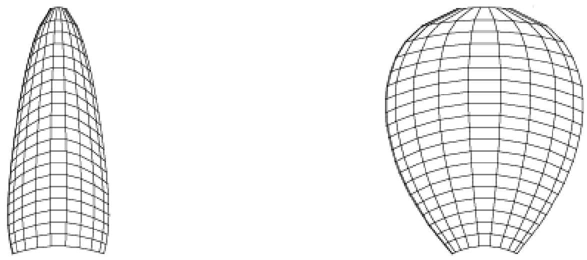


Figure 14: Grid for DC4381 Propeller, No Skew-No skew-induced rake. $M_{sp} \times N_{ch} = 20 \times 10$

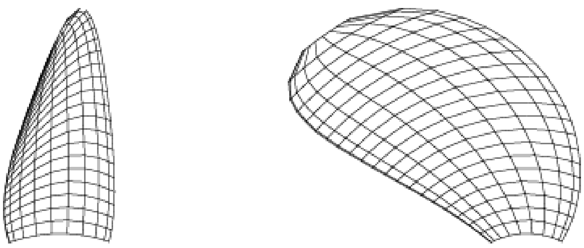


Figure 15: Grid for DC4497 Propeller, 36° Skew, No Skew-induced rake. $M_{sp} \times N_{ch} = 20 \times 10$

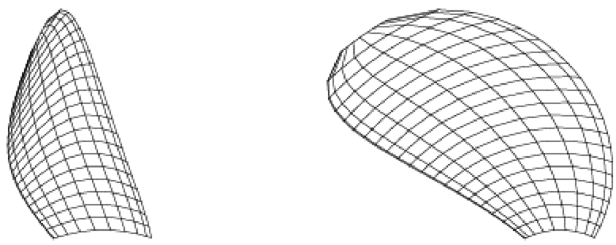


Figure 16: Grid for DC4382 Propeller, 36° Skew, Skew-induced rake. $M_{sp} \times N_{ch} = 20 \times 10$

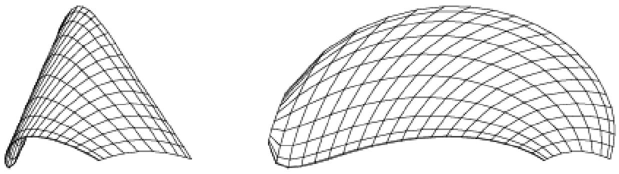


Figure 17: Grid for DC4383 Propeller, 72° Skew, Skew-induced rake. $M_{sp} \times N_{ch} = 20 \times 10$

The design thrust coefficient, $K_{T,D}$, is approximated from Kerwin and Lee (1978) [29], which also provides detailed geometry information of the propellers. The radius is chosen.
The four propellers include one reference propeller, which has no skew or rake (see Figure 14). The other two are connected, so they both have the same skew, but only one of them has skew-induced rake (see Figures 16 and 15). The last one has both skew and skew-induced rake, and it is different from the other two due to the skew being 72° instead of 36° (see Figure 15).

5.1 Grid Study

Initially, a grid study was conducted to validate the results by varying the number of panels. This approach aimed to assess both the consistency of the results and the impact of grid refinement. The parameter was varied with configurations such as $M_{sp} \times N_{ch} = 5 \times 5, \quad 20 \times 10, \quad 30 \times 20$. The grid study is done with the reference propeller, DC4381, and the linear theory is used. Hence, the grids of the propellers are aligned with the onset flow and the grid is not changed.

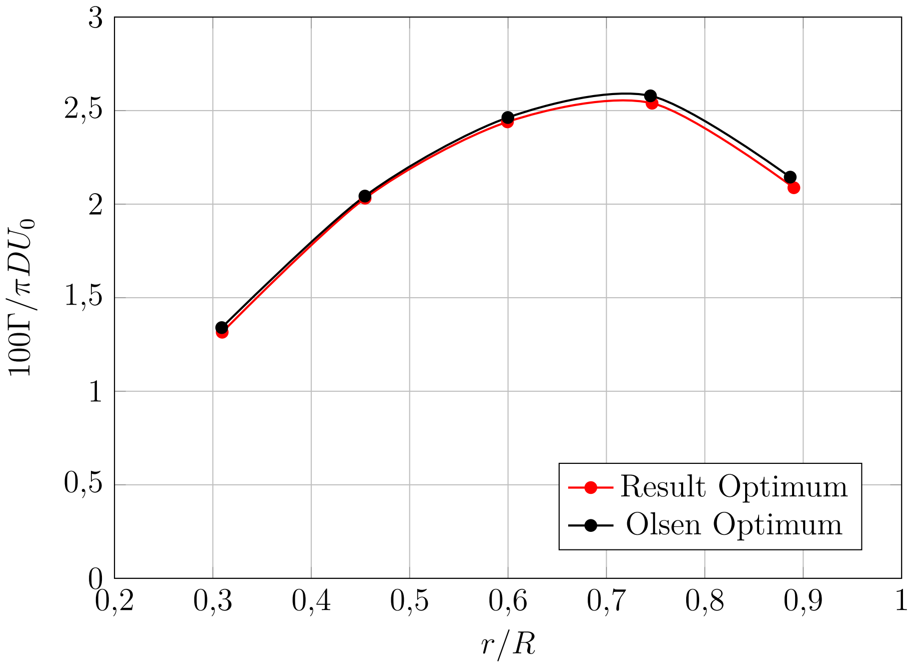


Figure 18: DC4381 $M_{sp} \times N_{ch} = 5 \times 5$

The tables below shows the optimised torque coeffcient $10K_Q$ for $DC4381$:

The table above illustrates a relative difference of 2.11% between the Olsen value and the validation value. Additionally, it is observed that the absolute difference decreases as the number

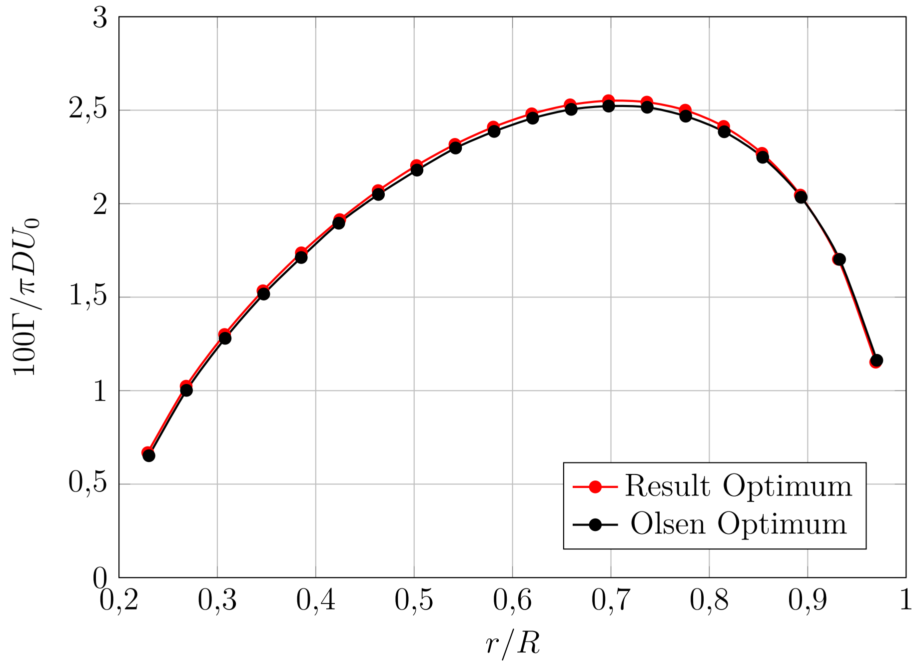


Figure 19: DC4381 $M_{sp} \times N_{ch} = 20 \times 10$

Table 1: Results from Grid Study

Grid cells	$10K_Q$ **(Olsen)**	$10K_Q$ **(Validation)**
5×5	0.3701	0.3587
20×10	0.3695	0.3611
30×20	0.3697	0.3619

of panels increases. From the grid study, it can be concluded that the number of grid points does not significantly impact the final outcome, but rather concerns the desired resolution of the solution. Furthermore, as the number of panels increases, the resulting values tend to converge.

5.1.1 Thrust Loading

The grid study involved varying the thrust coefficient to assess its agreement with Olsen's results, using a thrust coefficient of $C_{Th} = 2.0$. However, due to the utilization of linear theory, the results for high thrust loads ($C_{Th} = 2.0$) may not be accurate, as the alignment of the trailers was not considered. Nonetheless, these findings provide insights into the method's performance, under high thrust conditions.

Conversely, the lowest thrust load ($C_{Th} = 0.662$) is considered sufficiently low to justify the application of linear theory. Although slight differences may arise between results with and without wake alignment, these variances are assumed to be negligible for this thrust load. It's worth noting that the disparities between the two sets of results are minimal.

Table 2: Results obtained by varying thrust coefficient, $C_{Th} = 2$

Grid cells	$10K_Q$ **(Olsen)**	$10K_Q$ **(Validation)**
5×5	0.2725	0.2428
20×10	0.2715	0.2602

5.2 Advance Ratio

At this point, the aim is to compare Olsen's optimization of the lifting surface with the program's optimization, by varying the advance number. For this purpose, the reference propeller, DC4381, was analyzed for a constant thrust loading and a range of advance numbers. Calculations were performed for a thrust loading of 0.662 and a uniform onset flow with a velocity of $10.0m/s$. The chordwise loading is half flat plate and half rooftop ($\nu = 0.5$). The advance number was varied by changing the rotational speed of the propeller.

Table 3: Results obtained by varying Advance Ratio

J	**0.8**	**1.0**	**1.2**
n **(rps)**	2.083	1.667	1.398
w **(rad/sec)**	13.090	10.472	8.7266
K_T	0.16645	0.26008	0.37453
K_Q **Olsen**	0.02655	0.05371	0.09709
K_Q **Validation**	0.02600	0.05249	0.09484
η **Olsen**	0.79810	0.77074	0.73673
η **Validation**	0.81464	0.78822	0.75376

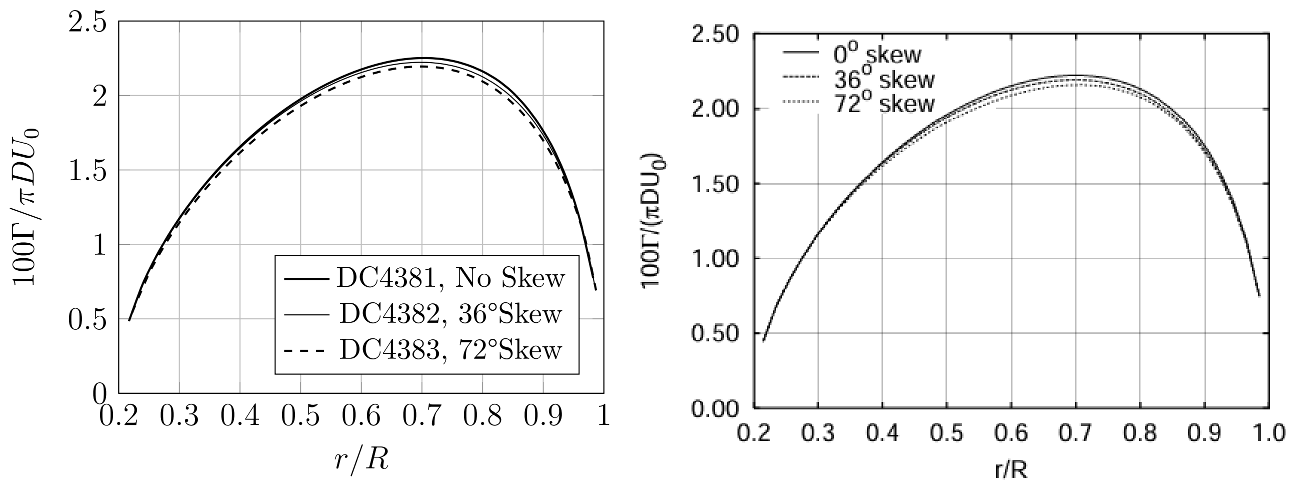
An increase in circulation occurs when the advance numbers increase (see Figures 21 and 23). This is expected, as the force on the blade is a function of speed and circulation (see Equation 38). Therefore, to maintain the same thrust, it is necessary for the circulation to increase, as the rotational speed decreases. Olsen provides an explanation of this phenomenon by examining the equations used and the force distribution on the lifting surface. In particular, he discusses the contribution of panels to thrust and torque, as well as their dependence on induced velocities and onset flow. He concludes that the circulation for optimizing the lifting surface load should be increasingly tip-loaded for decreasing advance numbers. For a better explanation, see Olsen [4].
As the advance numbers increase, there is an observed decrease in efficiency. This is attributed to the increasing relative magnitude of the axial velocity compared to the rotational velocity at higher advance numbers. Consequently, the torque also increases in relation to the thrust.

A grid consisting of 35 panels in the longitudinal direction and 20 panels in the chordwise direction was used, employing linear theory. The validation results were reported in Table (3). Also in this case, the differences in results are minimal.

5.3 Skew

For further comparison between Olsen's method and this program, the optimal distribution of circulation is considered. Figure (20) illustrates the circulation distributions for the propeller DC4381, as well as for two propellers with both skew and skew-induced rake, DC4382 and DC4383. From the figure, it is evident that the maximum value of circulation decreases with increasing skew. Hence, the shape of the propeller has a small but noticeable influence on the optimal distribution of circulation. Consequently, the efficiency of the skewed propeller is higher than that of the propeller without skew. Therefore, it can be concluded that the efficiency of the propeller is positively influenced by the increase in skew.
The design point used:

- $M_{sp} \times N_{ch} = 35 * 20$
- $J = 0.8$
- $U_a = 10m/s$
- $C_{Th} = 0.662$
- $K_T = 0.1664$



(a) $M_{sp} \times N_{ch} = 35 \times 20$

(b) Olsen optimum

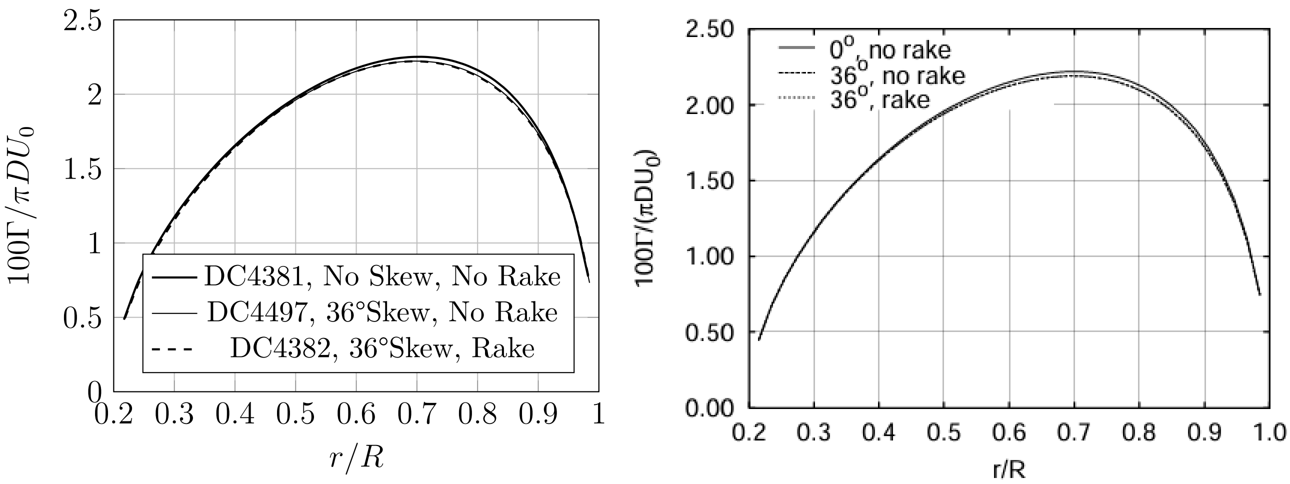
Figure 20: Comparison between results for the reference propeller, DC4381, the propeller with 36° skew and skew-induced rake, DC4382, and the propeller with 72° skew and skew-induced rake, DC4383.

Table 4: Results obtained by varying Skew

J	0.8		
K_T	0.16645		
Propeller	4381	4382	4383
Skew	0°	36°	72°
Indu.-rake	no	yes	yes
K_Q (Olsen)	0.02655	0.02641	0.2623
K_Q (Validacion)	0.02600	0.02602	0.02595
η (Olsen)	0.79810	0.80252	0.80786
η (Validacion)	0.81464	0.81507	0.81642

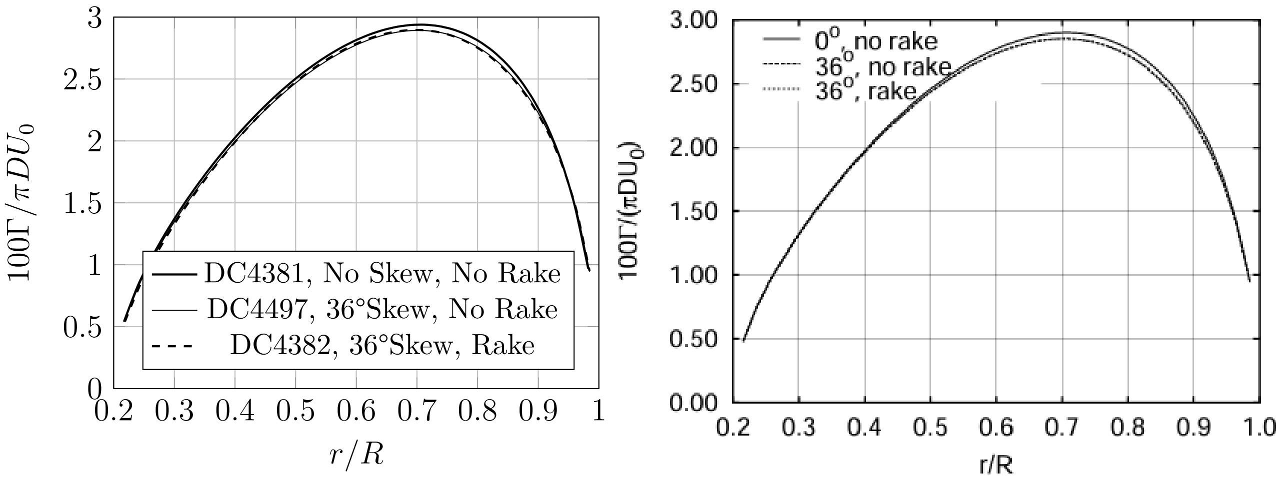
5.4 Skew-Induced Rake

Figures (21) and (23) compare the results for the skewed propellers with and without skew-induced rake, alongside the results for the reference propeller. The data is provided for $J = 0.8$ and $J = 1.0$. The Figures and Tables (5), illustrate that the circulation distribution and efficiency for the propellers with and without skew-induced rake are nearly identical. These results align with Munk's displacement theorem.



(a) $M_{sp} \times N_{ch} = 35 \times 20$ (b) Olsen Optimum

Figure 21: Comparison between results for the reference propeller,DC4381,and the two propellers with 36° skew, DC4382 which has skew-induced rake and DC4497 which has no rake. $J = 0.8$



(a) $M_{sp} \times N_{ch} = 35 \times 20$ (b) Olsen Optimum

Figure 22: Comparison between results for the reference propeller, DC4381, and the two propellers with 36° skew, DC4382 which has skew-induced rake and DC4497 which has no rake. $J = 1.0$.

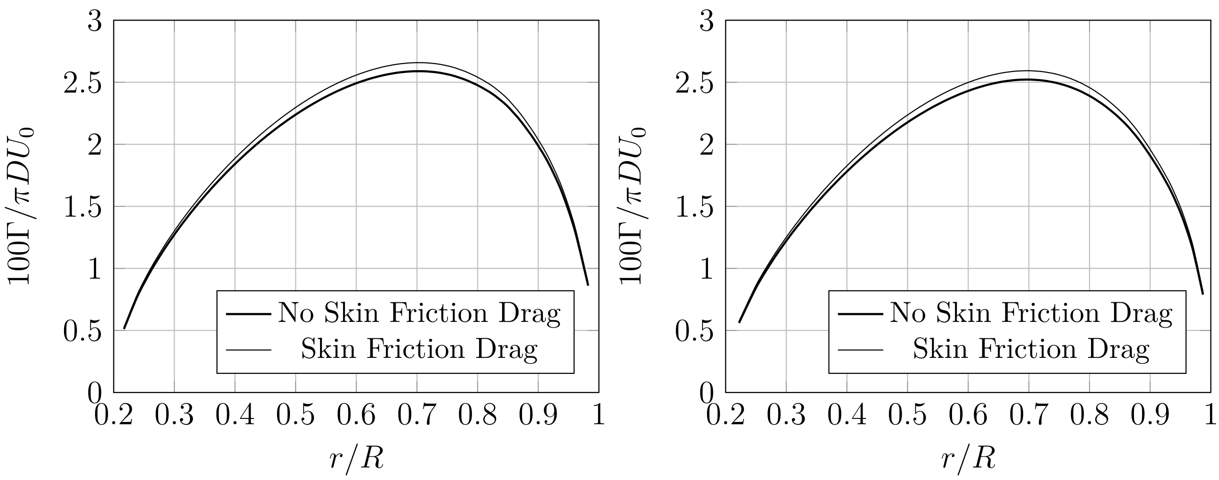
Table 5: Results obtained by varying Skew-induced Rake

J	0.8		
K_T	0.16645		
Propeller	4381	4382	4497
Skew	0°	36°	36°
Indu.-rake	no	yes	no
K_Q (Olsen)	0.02655	0.02641	0.02639
K_Q (Validation)	0.02600	0.02602	0.02598
η (Olsen)	0.79810	0.80252	0.80305
η (Validation)	0.81464	0.81507	0.81418

J	1		
K_T	0.26008		
Propeller	4381	4382	4497
Skew	0°	36°	36°
Indu.-rake	no	yes	no
K_Q (Olsen)	0.05371	0.05330	0.05324
K_Q (Validation)	0.05249	0.05236	0.05241
η (Olsen)	0.77074	0.77658	0.7743
η (Validation)	0.78822	0.79020	0.78948

5.5 Skin Friction Drag

The following analysis focuses on the influence of Skin Friction Drag. Two propellers were examined: the reference propeller, DC4381, and the propeller with skew but without induced rake, DC4497. The results clearly indicate that the inclusion of skin friction drag results in a more significant increase in torque and a decrease in efficiency. While potential flow theory provides a good representation of reality, incorporating a correction coefficient to account for the resistance generated in the boundary layer due to water viscosity brings our analysis closer to a more accurate depiction of reality.



(a) DC4381; $M_{sp} \times N_{ch} = 35 \times 20$

(b) DC4497; $M_{sp} \times N_{ch} = 35 \times 20$

Figure 23: Comparison between results for the reference propeller, DC4381, and the 36° skew, DC4497 which has no rake.

Table 6: Variation of Skin Friction Drag Results

Propeller	ω **(rad/sec)**	$10K_Q$ **(Inviscid)**	$10K_Q$ **(Viscid)**	η **(Inviscid)**	η **(Viscid)**
DC4381	1.875	0.0374	0.0420	0.8020	0.71499
DC4497	1.875	0.0373	0.04249	0.8045	0.70755

6 Conclusions

The outcome of this study demonstrates the successful development of a Python-based vortex lattice method to optimize propeller efficiency for a given thrust, proving its effectiveness across different grid resolutions and for various propeller loadings. Furthermore, the calculations indicate that the impact of the chordwise pressure distribution is negligible, in accordance with Munk's displacement theorem. Despite incorporating the entire blade into the optimization process, it becomes apparent that with the vortex-lattice method utilized, the majority of thrust and torque originate from the sides of the horseshoe vortices along the trailing edges, where the onset flow and induced velocities are fully incorporated. This observation is consistent with the principles outlined in Munk's theorem.

Comparison among the DTNSRDC propellers reveals variations in the distributions of circulation and torque. Notably, skew enhances efficiency, and further gains in efficiency can be achieved by eliminating skew-induced rake. Although the exact explanation behind this effect is not fully understood, some insights can be obtained by examining the combination of circulation distribution and total velocities at the trailing edge for different propellers. This comparison suggests a beneficial impact of skew on induced velocities at the trailing edge, resulting in higher efficiencies for skewed propellers. Further investigation is necessary to fully understand why propellers with skew demonstrate superior efficiency, although similar findings are documented in Mishima and Kinnas (1997) [36].

Furthermore, the calculations demonstrate that circulation and torque distributions are dependent on blade geometry. Additionally, incorporating the Skin Friction Drag coefficient separately in the calculation yields results closer to reality, accounting for a portion of drag neglected in potential flow theory.

7 References

[1] Betz, A. Prandtl, *L.,Schraubenpropeller mit Geringstem Enegieverlust* Goettnger Nachtrichten, pp. 193-217, March 1919

[2] H.W. Lerbs, *Moderately loaded propellers with a finite number of blades and an arbitrary distribution of circulation*, Trans. SNAME 60, 1952.

[3] J.E. Kerwin, W.B. Coney, C.-Y. Hsin, *Optimum Circulation Distribution for Single and Multi-Component Propulsors*, in Messalle, R. F. (editor), Proc. of Twenty-First American Towing Tank Conference, pp. 53–62, National Academy Press, Washington, D.C, 1986.

[4] A.S. Olsen *Optimisation of Propellers Using the Vortex-Lattice Method.* 2001.

[5] *EEXI and CII - Ship carbon intensity and rating system*, International Maritime Organization `https://www.imo.org/en/MediaCentre/HotTopics/Pages/EEXI-CII-FAQ.aspx`

[6] *EEXI – Energy Efficiency Existing Ship Index* DNV

[7] E.A. Bouman, E. Lindstad, A.I. Rialland, A.H. Strømman. *State-of-the-art technologies, measures, and potential for reducing GHG emissions from shipping – A review* Norwegian University of Science and Technology, 2017. `https://www.sciencedirect.com/science/article/pii/S1361920916307015#t0005`

[8] M. Issa, A. Ilinca, F. Martini, *Ship Energy Efficiency and Maritime Sector Initiatives to Reduce Carbon Emissions*. Institut Maritime du Québec à Rimouski, 2022.

[9] F. Vesting, *Marine Propeller Optimisation - Strategy and Algorithm Development*, CHALMERS UNIVERSITY OF TECHNOLOGY, 2015.

[10] W.B.Coney, *A METHOD FOR THE DESIGN OF A CLASS OF OPTIMUM MARINE PROPULSORS*, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 1989.

[11] J.E.KERWIN *MARINE PROPELLERS* Marine Annual Review of Fluid Mechanics. Vol. 18:367-403, January 1986.

[12] J. CARLTON *MARINE PROPELLERS AND PROPULSIONS*, Global Head of Marine Technology and Investigation, Lloyds Register, 2007.

[13] J.N. Newman, foreword by J. Grue. *Marine Hydrodynamics*, Massachusetts Institute of Technology, 40th anniversary edition, 2017.

[14] J. Katz, A. Plotktin *Low-Speed Aerodynamics, Second Edition*, Cambridge University, 2001.

[15] European Environment Agency, 2023. `https://www.eea.europa.eu/highlights/eu-maritime-transport-first-environmental`

[16] Review of Maritime Transport 2023, UNCTAD, United Nations publication, 2023.

[17] R.M. James. *On The Remarkable Accuracy Of The Vortex Lattice Method*, Computer Methods in Applied Mechanics and Engineering, 1(1):5979, 1972.

[18] C.E. Lan. *A Quasi-Vortex-Lattice Method in Thin Wing Theory*, 1974.

[19] *Towards a green and just transition*, UNCTAD, United Nations publication, 2023.

[20] K. De Jong. *On the Optimization and the Design of Ship Screw Propellers with and without End Plates*. PhD thesis, 1991.

[21] J.P. Breslin, P.Andersen, *Hydrodynamics of Ship Propellers*, Cambridge Ocean Technology Series 3, Cambridge University Press, Cambridge, UK. 1994.

[22] W.B. Coney, *Optimum Circulation Distributions for a Class of Marine Propulsors*, Journal of Ship Research, 36(3):210–222. 1992.

[23] *How much does the shipping industry contribute to global CO2 emissions?*, SINAY, Maritime Data Solution, 2023. `https://sinay.ai/en/how-much-does-the-shipping-industry-contribute-to-global-co2-emissions/#:~:text=In%202022%2C%20international%20shipping%20alone,contributor%20to%20global%20carbon%20pollution`

[24] S. Goldstein, *On the vortex theory of screw propellers*, Technical report, St. John's College, Cambridge, January 1929.

[25] G.G. Cox, *Corrections to the Camber of Constant Pitch Propellers*, Quaarterly Transactions of the Royal Institution of Naval Architects, Vol. 103, pp. 27-243, 1961.

[26] M.K. Eckhardt, W.B. Morgan, *A propeller design method* Trans. SNAME, 63, 1955.

[27] T.E. Brockett, *Lifting surface hydrodynamics for design of rotating blades*, Propellers '81, Symp. SNAME, 1981.

[28] S. Tsakonas, W.R. Jacobs, P. Liao, *Prediction of steady and unsteady loads and hydrodynamic forces on counter-rotating propellers* J.Ship Res., 27, 1983.

[29] J.E. Kerwin, Chang-Sup Lee. *Prediction of steady and unsteady marine propeller performance by numerical lifting-surface theory* Trans.SNAME, Paper No.8, Annual Meeting, 1978.

[30] W. van Gent, *On the Use of Lifting Surface Theory for Moderately and Heavily Loaded Ship Propellers* NSMB Report No. 536, 1977.

[31] D.A. Greeley, J.E. Kerwin, *Numerical methods for propeller design and analysis in steady flow*, Trans. SNAME, 90, 1982

[32] P. Andersen, *A Comparative Study of Conventional and Tip-Fin Propeller Performance, in Proc. Twenty-First Symposium on Naval Hydrodynamics*, pp. 930–945, National Academy Press, Washington, D.C. 1997.

[33] M. Caponnetto, *Optimisation and Design of Contra-Rotating Propellers*, in Proc.Propeller/Shafting 2000 Symposium, pp. 3.1–3.9, SNAME, Jersey City, N.J. 2000.

[34] M. Karim, M. Ikehata, K. Suzuki, H. Kai, *Application of Micro-Generic Algorithm (µGA) to the Optimal Design of Lifting Bodies*, J. Kansai Soc. of Naval Architects, Japan, 235:1–8. 2001.

[35] J.L. Hess, W.O. Valarezo, *Calculation of Steady Flow about Propellers by Means of a Surface Panel Method*, AIAA, Paper No. 85, 1985.

[36] S. Mishima, S.A Kinnas, *Application of a Numerical Optimization Technique to the Design of Cavitating Propellers in Nonuniform Flow*, Journal of Ship Research, 41(2):93–107, 1997.

8 Code

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This is the main.
"""


def main ():
    from sources.propeller_geometry import propeller_geometry

    (ir_prop, ix_prop, iskew_prop, ichord_prop, ithick_prop) = propeller_geometry()

    from sources.Grid_Generation_Propeller import Grid_Generation_Propeller
    (S_Distr_P, r_R_P, t_gp_P, s_gp_P, Grid_Points_P, Control_Points_P,
           N_Panel_P, N_Bound_Vortex_P, Horseshoe_P, Points_Trans_Wake_P
           )=Grid_Generation_Propeller()

    from sources.Weight_Function_Propeller_P import Weight_function_propeller
    Weight_P = Weight_function_propeller()

    from sources.Onset_Flow_Propeller_P import Onset_Flow_Propeller
    V_Onset_P = Onset_Flow_Propeller()

    from sources.Induced_Grid_Propeller_P import Induced_Grid_Propeller
    V_Grid_P = Induced_Grid_Propeller()

    from sources.Velocity_Total_No_Onset_Propeller_P import Velocity_Total_No_Onset_Propeller
    V_Ind_P, V_Tral_P = Velocity_Total_No_Onset_Propeller()

    from sources.System_Equations_Propeller_P import System_Equations_Propeller_P
    Gamma_TE_P_No_dim, R_Circ_P_R = System_Equations_Propeller_P()

    from sources.Advance_Ratio_P import Advance_Ratio_J
    Advance_ratio = Advance_Ratio_J()

    from sources.Skin_Friction_Drag_P import Skin_Friction_Drag
    T_fr_P, Q_fr_P = Skin_Friction_Drag()

    from sources.Efficiency_P import Efficiency
    Eff, K_T, K_Q = Efficiency()

    return Gamma_TE_P_No_dim


Gamma_TE_P_No_dim = main()
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine is tasked with creating and solving the system of
equations for the propeller analysis. It aligns the wake of the propeller,
ensuring that the flow dynamics are accurately represented and optimized.
"""


import numpy as np
import sources.Variables as Var
from sources.Weight_Function_Propeller_P import Weight_function_propeller
from sources.Onset_Flow_Propeller_P import Onset_Flow_Propeller
from sources.Mid_Vect_Propeller_P import Mid_Vect_Propeller
from sources.Induced_Grid_Propeller_P import Induced_Grid_Propeller
from sources.Velocity_Total_No_Onset_Propeller_P import Velocity_Total_No_Onset_Propeller
from sources.Gamma_Initialization import Gamma_It
from sources.Propeller_Pitch import pitch
from sources.Velocity_Total_Propeller_P import Velocity_Total_Propeller
from sources.Align_Wake_Propeller_P import Align_Wake_Propeller
from sources.Skin_Friction_Drag_P import Skin_Friction_Drag


def System_Equations_Propeller_P():
    Weight_P = Weight_function_propeller()
    Gamma_TE_P = Gamma_It()
    V_Onset_P = Onset_Flow_Propeller()
    V_Ind_P, V_Tral_P = Velocity_Total_No_Onset_Propeller()
    Points_Trans_Wake_P = np.loadtxt("output/Propeller_Points_Trans_Wake.txt",
                                     skiprows= 1, usecols= (1,2,3))

    # DECLARATION OF VARIABLES

    matr_T = np.zeros((Var.Msp+1, Var.Msp+1))
    matr_Q1 = np.zeros((Var.Msp+1, Var.Msp+1))
    matr_Q2 = np.zeros((Var.Msp+1, Var.Msp+1))
    matrix = np.zeros((Var.Msp+1, Var.Msp+1))
    rhsQ = np.zeros((Var.Msp+1,2))
    #Right hand side of the equation system
    rhs = np.zeros((Var.Msp+1,1))

    T_fr_P = 0.0
    # Thrust - Skin friction drag - Propeller
    Tr_P = 0.0

```

```python
    R_Circ_P = np.zeros((Var.Msp))
    # Radius where the circulation is calculated at the T.E.
    R_Circ_P_R = np.zeros((Var.Msp))
    # Dimensionaless radius where the circulation is calculated at the T.E.
    pitch_0 = np.zeros((Var.Msp+1,1))

    # INITIALIZATION

    Cs_T_r = (Var.Tr_P)/Var.rho/float(Var.Z_Blade_P)
    # Required thrust for each blade without rho (We don´t use rho in the system)

    iteration = 1

    # INITIALIZATION VARIABLE GAMMA

    Gamma_TE_P_No_dim = np.zeros((Var.Msp,1))
    #Distribution of circulation at the T.E. (Dimensionaless)
    Gamma_Panel_P = np.zeros((Var.Msp*Var.Nch))
    # Distribution of circulation on the blade

    # ALIGNMENT LOOP

    for j in range (Var.Msp+1):
        rhs[j,0] = 0.0
        rhsQ[j,0] = 0.0
        rhsQ [j,1] = 0.0
        for i in range (Var.Msp+1):
            matr_T[j,i] = 0.0
            matr_Q1[j,i] = 0.0
            matr_Q2[j,i] = 0.0
            matrix[j,i] = 0.0

    # SYSTEM OF EQUATIONS - DOUBLE LOOP USED TO CALCULATE &T(Uo),&Q1(Uo),&Q2(Uo)

    # This loop creates the system of equations (m = 1,2,3... Msp - Lines of the matrix)
    for m in range(Var.Msp):
        temp_T_0  = 0.0      # Initialization of the temporary variable used to calculate &T(Uo)
        temp_Q1_0 = 0.0      # Initialization of the temporary variable used to calculate &Q1(Uo)
        temp_Q2_0 = 0.0      # Initialization of the temporary variable used to calculate &Q2(Uo)

        for n in range (Var.Nch):       # First loop used to calculate the first sum (Nch)
            npln =  (n)+(m)*Var.Nch     # Counter used to select the right panel

            n_side = 4         # Number of sides for each panel
            if n == 0:
                n_side = 3
                # If we are considering the T.E. panel, instead of removing
                # the value of the T.E. side, we skip it

            temp_T_1 = 0.0     # Initialization of the temporary variable used to calculate &T
            temp_Q_11 = 0.0    # Initialization of the temporary variable used to calculate &Q1
            temp_Q_22 = 0.0    # Initialization of the temporary variable used to calculate &Q2

            for l in range(n_side):
                # Second loop used to calculate the second sum (4)
                xxn,xyn,xzn,xln,yln,zln = Mid_Vect_Propeller(npln,l)
                # This subroutine is used to calculate the midpoint

                temp_T_1 = temp_T_1 + zln*V_Onset_P[npln,l,1] - yln*V_Onset_P[npln,l,2]
                # Temporary variable used to calculate &T

                temp_Q_11 = temp_Q_11 + xyn*yln*V_Onset_P[npln,l,0] - xyn*xln*V_Onset_P[npln,l,1]
                # Temporary variable used to calculate &Q1

                temp_Q_22 = temp_Q_22 + xzn*xln*V_Onset_P[npln,l,2] - xzn*zln*V_Onset_P[npln,l,0]
                # Temporary variable used to calculate &Q2

            temp_T_0  = temp_T_0 + Weight_P[m,n] * temp_T_1
            temp_Q1_0 = temp_Q1_0 + Weight_P[m,n] * temp_Q_11
            temp_Q2_0 = temp_Q2_0 + Weight_P[m,n] * temp_Q_22
            # Temporary variable used to calculate T(Uo) (Nch Loop)
            # Temporary variable used to calculate Q1(Uo) (Nch Loop)
            # Temporary variable used to calculate Q2(Uo) (Nch Loop)

        matr_T [m,Var.Msp] = temp_T_0   # Value of &T(Uo) in the right position in the matrix (Temporary matrix matr_T)
        rhsQ[m,0] = - temp_Q1_0         # Value of &Q1(Uo) in the right position in the matrix (Temporary matrix rhs_Q)
        rhsQ[m,1] = - temp_Q2_0         # Value of &Q2(Uo) in the right position in the matrix (Temporary matrix rhs_Q)

    # Double loop used to calculate &T(Gam),&Q1(Gam),&Q2(Gam)

    # Loop used to select the line of the equation
    # (We don´t have the loop n because we already did that in Induced_Grid_Propeller)
    for m in range(Var.Msp):
        # Loop used to select the spanwise layer that induces velocity (Columns of the matrix) - Msp SUM
        for j in range (Var.Msp):
            temp_T_Gam = 0.0          # Initialization of the temporary variable used to calculate &T(Gam)
            temp_Q1_Gam = 0.0         # Initialization of the temporary variable used to calculate &T(Gam)
            temp_Q2_Gam = 0.0         # Initialization of the temporary variable used to calculate &T(Gam)

            #Loop used to select where the point is located (chordwise) - First SUM Nch
            for n in range(Var.Nch):
                npln = n + (m)*Var.Nch        #Panel where the point is located

                temp_T_1 = 0.0          # Initialization of the temporary variable used to calculate &T
                temp_Q_11 = 0.0         # Initialization of the temporary variable used to calculate &Q1
                temp_Q_22 = 0.0         # Initialization of the temporary variable used to calculate &Q2

                n_side = 4              # If we are considering the T.E. panel,
                if n == 0:              # instead of removing the value of the T.E. side, we skip it
                    n_side = 3

                for l in range (n_side):
```

```
                    # Loop used to select the side of the panel

                    xxn,xyn,xzn,xln,yln,zln =  Mid_Vect_Propeller(npln,l)
                    # This subroutine is used to calculate the midpoint

                    temp_T_1 = temp_T_1 + zln*V_Ind_P[j,npln,l,1] - yln*V_Ind_P[j,npln,l,2]
                    # Temporary variable used to calculate &T - Total thrust for that panel by j

                    temp_Q_11 = temp_Q_11 + xyn*yln*V_Ind_P[j,npln,l,0]- xyn*xln*V_Ind_P[j,npln,l,1]
                    # Temporary variable used to calculate &Q1 - Total torque 1 for that panel by j

                    temp_Q_22 = temp_Q_22 + xzn*xln*V_Ind_P[j,npln,l,2]- xzn*zln*V_Ind_P[j,npln,l,0]
                    # Temporary variable used to calculate &Q2 - Total torque 2 for that panel by j

                temp_T_Gam = temp_T_Gam + Weight_P[m,n] *  temp_T_1
                temp_Q1_Gam = temp_Q1_Gam + Weight_P[m,n] * temp_Q_11
                temp_Q2_Gam = temp_Q2_Gam + Weight_P[m,n] * temp_Q_22
                # Temporary variable used to calculate Q1 (Nch Loop)
                # Temporary variable used to calculate Q2 (Nch Loop)
                # Temporary variable used to calculate T (Nch Loop)

            for i in range (Var.Nch):
            # Loop used to select where the point is located (chordwise)
            # Second SUM Nch
                npli = i + (j)* Var.Nch

                temp_T_1 = 0.0
                temp_Q_11 = 0.0
                temp_Q_22 = 0.0
                # Initialization of the temporary variable used to calculate &T
                # Initialization of the temporary variable used to calculate &Q1
                # Initialization of the temporary variable used to calculate &Q2

                n_side = 4
                if i == 0:
                    n_side = 3
                    # If we are considering the T.E. panel,
                    # instead of removing the value of the T.E. side, we skip it

                for l in range(n_side):
                    xxi,xyi,xzi,xli,yli,zli = Mid_Vect_Propeller(npli,l)
                    # This subroutine is used to calculate the midpoint

                    temp_T_1 = temp_T_1 + zli*V_Ind_P[m,npli,l,1]- yli*V_Ind_P[m,npli,l,2]
                    # Temporary variable used to calculate &T
                    temp_Q_11 = temp_Q_11 + xyi*yli*V_Ind_P[m,npli,l,0] - xyi*xli*V_Ind_P[m,npli,l,1]
                    # Temporary variable used to calculate &Q1
                    temp_Q_22 = temp_Q_22 + xzi*xli*V_Ind_P[m,npli,l,2]- xzi*zli*V_Ind_P[m,npli,l,0]
                    # Temporary variable used to calculate &Q2

                temp_T_Gam = temp_T_Gam + Weight_P[j,i] * temp_T_1
                temp_Q1_Gam = temp_Q1_Gam + Weight_P[j,i] * temp_Q_11
                temp_Q2_Gam = temp_Q2_Gam + Weight_P[j,i] * temp_Q_22
                # Temporary variable used to calculate T (Nch Loop)
                # Temporary variable used to calculate Q1 (Nch Loop)
                # Temporary variable used to calculate Q2 (Nch Loop)

            matr_T[m,j] = temp_T_Gam
            matr_Q1[m,j] = temp_Q1_Gam
            matr_Q2[m,j] = temp_Q2_Gam
            # Value of &T(Gam) in the right position in the matrix (Temporary matrix matr_T)
            # Value of &Q1(Gam) in the right position in the matrix (Temporary matrix matr_T)
            # Value of &Q2(Gam) in the right position in the matrix (Temporary matrix matr_T)

    # SYSTEM OF EQUATIONS - LOOP

    V_Tot_P, V_Tot_No_Onset_P = Velocity_Total_Propeller ()
    # It is used it in order to update V_Tot_P with the new values of gamma

    # Loop for the T.E. panels (They don´t have the weight function)
    for m in range(Var.Msp):
        npl0 = (m)*Var.Nch
        n_side = 3
        temp_T_Gam = 0.0

        for l in range(n_side):
            xxm,xym,xzm,xlm,ylm,zlm = Mid_Vect_Propeller(npl0,l)
            # This subroutine is used to calculate the midpoint

            temp_T_Gam = temp_T_Gam + zlm*V_Tot_P[npl0,l,1] - ylm*V_Tot_P[npl0,l,2]
            # Temporary variable used to calculate T (Nch Loop)

        matr_T[Var.Msp,m] = temp_T_Gam
        # Value of &T (T.E.) in the right position in the matrix (Temporary matrix matr_T)

        # Loop for the other panels (They don´t have the weight function)
        for n in range(1, Var.Nch):
            npl1 = n + (m)*Var.Nch
            n_side = 4
            temp_T_2_Gam = 0.0

            # Loop for the other panels
            for l in range(n_side):
                xxm,xym,xzm,xlm,ylm,zlm =  Mid_Vect_Propeller(npl1,l)
                # This subroutine is used to calculate the midpoint
                temp_T_2_Gam = temp_T_2_Gam + zlm*V_Tot_P[npl1,l,1] - ylm*V_Tot_P[npl1,l,2]
                # Temporary variable used to calculate T (Nch Loop)
            matr_T[Var.Msp,m] = matr_T[Var.Msp,m] + Weight_P[m,n]*temp_T_2_Gam
            # Value of &T in the right position in the matrix (Temporary matrix matr_T)

    # CREATION OF THE MATRIX

```

```
    lamba_t_1 = Gamma_TE_P[Var.Msp]
    # Lagrange multiplier lambda t-1

    for i in range (Var.Msp):
        rhs[i,0] = rhsQ[i,0] - rhsQ[i,1]
        # rhs matrix

        matrix[i,Var.Msp] = matr_T[i,Var.Msp]       # System of equation (Left Matrix)
        matrix[Var.Msp,i] = matr_T[Var.Msp,i]       # System of equation (Left Matrix)

        for j in range (Var.Msp):
            matrix[i,j] = matr_Q1[i,j] - matr_Q2[i,j] + lamba_t_1*matr_T[i,j]
            # System of equation (Left Matrix)

    rhs[Var.Msp,0] = Cs_T_r + (abs(T_fr_P))/Var.rho
    # Total thrust required (Required + Skin Friction Drag Propeller)
    matrix[Var.Msp,Var.Msp] = 0.0

    # SOLVE THE SYSTEM OF EQUATIONS

    rhs = np.linalg.solve(matrix, rhs)  #it solves the system of equations
    #Computes the ""exact solution, x, of the well-determined, i.e.,
    # full rank, linear matrix equation ax = b.

    # CONVERGENS OF THE SYSTEM

    res_0 = 0.0
    # Loop used to check if the residual is below a certain small limit
    for i in range(Var.Msp+1):
        res_1 = abs(1-Gamma_TE_P[i]/rhs[i,0])

        if res_1 > res_0:
            res_0 = res_1
        Gamma_TE_P[i] = rhs[i,0]                # New values of circulation
    with open ("output/Propeller_Gamma_TE_P.txt","w") as file:
        for i in range (Var.Msp+1):
            file.write(f"{Gamma_TE_P[i]:13.9f}\n")

    while (res_0 > Var.epsi):
        V_Tot_P, V_Tot_No_Onset_P = Velocity_Total_Propeller ()
        # It is used it in order to update V_Tot_P with the new values of gamma

        # Loop for the T.E. panels (They don´t have the weight function)
        for m in range(Var.Msp):
            npl0 = (m)*Var.Nch
            n_side = 3
            temp_T_Gam = 0.0

            for l in range(n_side):
                xxm,xym,xzm,xlm,ylm,zlm = Mid_Vect_Propeller(npl0,l)
                # This subroutine is used to calculate the midpoint

                temp_T_Gam = temp_T_Gam + zlm*V_Tot_P[npl0,l,1] - ylm*V_Tot_P[npl0,l,2]
                # Temporary variable used to calculate T (Nch Loop)

            matr_T[Var.Msp,m] = temp_T_Gam
            # Value of &T (T.E.) in the right position in the matrix (Temporary matrix matr_T)

            # Loop for the other panels (They don´t have the weight function)
            for n in range(1, Var.Nch):

                npl1 = n + (m)*Var.Nch
                n_side = 4
                temp_T_2_Gam = 0.0

                # Loop for the other panels
                for l in range(n_side):
                    xxm,xym,xzm,xlm,ylm,zlm =  Mid_Vect_Propeller(npl1,l)
                    # This subroutine is used to calculate the midpoint
                    temp_T_2_Gam = temp_T_2_Gam + zlm*V_Tot_P[npl1,l,1] - ylm*V_Tot_P[npl1,l,2]
                    # Temporary variable used to calculate T (Nch Loop)
                matr_T[Var.Msp,m] = matr_T[Var.Msp,m] + Weight_P[m,n]*temp_T_2_Gam
                # Value of &T in the right position in the matrix (Temporary matrix matr_T)

        # CREATION OF THE MATRIX

        lamba_t_1 = Gamma_TE_P[Var.Msp]
        # Lagrange multiplier lambda t-1

        for i in range (Var.Msp):
            rhs[i,0] = rhsQ[i,0] - rhsQ[i,1]
            # rhs matrix

            matrix[i,Var.Msp] = matr_T[i,Var.Msp]       # System of equation (Left Matrix)
            matrix[Var.Msp,i] = matr_T[Var.Msp,i]       # System of equation (Left Matrix)

            for j in range (Var.Msp):
                matrix[i,j] = matr_Q1[i,j] - matr_Q2[i,j] + lamba_t_1*matr_T[i,j]
                # System of equation (Left Matrix)

        rhs[Var.Msp,0] = Cs_T_r + (abs(T_fr_P))/Var.rho
        # Total thrust required (Required + Skin Friction Drag Propeller)
        matrix[Var.Msp,Var.Msp] = 0.0

        # SOLVE THE SYSTEM OF EQUATIONS

        rhs = np.linalg.solve(matrix, rhs)     #it solves the system of equations
        #Computes the ""exact solution, x, of the well-determined, i.e.,
        # full rank, linear matrix equation ax = b.

        # CONVERGENS OF THE SYSTEM

```

```
        res_0 = 0.0
        # Loop used to check if the residual is below a certain small limit
        for i in range(Var.Msp+1):
            res_1 = abs (1-Gamma_TE_P[i]/ rhs[i,0])

            if res_1 > res_0:
                res_0 = res_1
            Gamma_TE_P[i] = rhs[i,0]                # New values of circulation
        with open ("output/Propeller_Gamma_TE_P.txt","w") as file:
            for i in range (Var.Msp+1):
                file.write(f"{Gamma_TE_P[i]:13.9f}\n")

    print('Iteration Propeller Number: {}'.format(iteration), 'Circulation on the propeller at the TE:')

    for i in range (Var.Msp+1):
        print (i,Gamma_TE_P[i])

    for i in range (Var.Msp):
        j = (i)*Var.Nch

        xx,xy,xz,xl,yl,zl = Mid_Vect_Propeller(j,3)
        #This subroutine is used to calculate the midpoint px,py,pz

        R_Circ_P[i] = np.sqrt(xy*xy + xz*xz)
        R_Circ_P_R[i] = R_Circ_P[i]/Var.Rad_P

        Gamma_TE_P_No_dim[i] = (Gamma_TE_P[i]*100)/(np.pi*2*Var.Rad_P*Var.V_Ship)

    with open("output/Propeller_Gamma_TE.txt","w") as file:
        file.write("     Gamma_Dim      Gamma_No_Dim     Radius\n")
        for i in range (Var.Msp):
            file.write("{:13.9f}    {:13.9f}    {:13.9f}\n.format.  {Gamma_TE_P[i]}   {Gamma_TE_P_No_dim[i]}   {R_Circ_P[i]}\n")

    with open ("output/Propeller_Gamma_TE_P.txt","w") as file:
        for i in range (Var.Msp+1):
            file.write(f"{Gamma_TE_P[i]}\n")

    with open("output/Propeller_Print_Gamma_TE.txt","w") as file:
        for i in range(Var.Msp):
            file.write(f" {Gamma_TE_P_No_dim[i]}\n")

    with open("output/Propeller_Print_Radius_TE.txt","w") as file:
        for i in range(Var.Msp):
            file.write(f"{R_Circ_P_R[i]}\n")

    # DISTRIBUTION OF CIRCULATION AT THE REST OF THE BLADE

    for i in range(Var.Msp):
        npl_TE = i * Var.Nch
        Gamma_Panel_P[npl_TE] = Gamma_TE_P[i]

        for j in range (1,Var.Nch):
            npl = j + i * Var.Nch
            Gamma_Panel_P [npl] = Gamma_Panel_P[npl_TE] * Weight_P[i,j]

    with open ("output/Propeller_Gamma_Blade.txt","w") as file:
        file.write("  Panel          Gamma\n")
        for i in range(Var.Msp*Var.Nch):
            file.write(f"   {i:3d}       {Gamma_Panel_P[i]:13.9f}\n")

    # ALIGNMENT OF THE WAKE

    pitch_0 = pitch()

    Points_Trans_Wake_P, Grid_Points_P, Control_Points_P = Align_Wake_Propeller()
    res_0 = 0.0
    # Initialization of the residual

    # Loop used to check if the residual is below a certain small limit
    for i in range(Var.Msp +1 ):
        i_1 = i+i*(Var.N_P_L)
        res_1 = abs(1 - (Points_Trans_Wake_P[i_1,2] / pitch_0[i]))

    if res_1 > res_0:
        res_0 = res_1

    while(iteration < 15): # If the the residual is greater than epsi the loop starts again
        while (res_0 > Var.epsi):
            V_Onset_P = Onset_Flow_Propeller()
            V_Grid_P = Induced_Grid_Propeller()
            V_Ind_P, V_Tral_P = Velocity_Total_No_Onset_Propeller()
            T_fr_P, Q_fr_P = Skin_Friction_Drag()

            iteration = iteration + 1

            for j in range (Var.Msp+1):
                rhs[j,0] = 0.0
                rhsQ[j,0] = 0.0
                rhsQ [j,1] = 0.0
                for i in range (Var.Msp+1):
                    matr_T[j,i] = 0.0
                    matr_Q1[j,i] = 0.0
                    matr_Q2[j,i] = 0.0
                    matrix[j,i] = 0.0

            # SYSTEM OF EQUATIONS - DOUBLE LOOP USED TO CALCULATE &T(Uo),&Q1(Uo),&Q2(Uo)

            # This loop creates the system of equations (m = 1,2,3... Msp - Lines of the matrix)
            for m in range(Var.Msp):
                temp_T_0  = 0.0     # Initialization of the temporary variable used to calculate &T(Uo)
                temp_Q1_0 = 0.0     # Initialization of the temporary variable used to calculate &Q1(Uo)
                temp_Q2_0 = 0.0     # Initialization of the temporary variable used to calculate &Q2(Uo)
```

```
            # First loop used to calculate the first sum (Nch)
            for n in range (Var.Nch):
                npln =  (n)+(m)*Var.Nch
                # Counter used to select the right panel

                n_side = 4       # Number of sides for each panel
                if n == 0:
                    n_side = 3
                    # If we are considering the T.E. panel, instead of removing
                    # the value of the T.E. side, we skip it

                temp_T_1 = 0.0
                temp_Q_11 = 0.0
                temp_Q_22 = 0.0
                # Initialization of the temporary variable used to calculate &T
                # Initialization of the temporary variable used to calculate &Q1
                # Initialization of the temporary variable used to calculate &Q2

                for l in range(n_side):
                    # Second loop used to calculate the second sum (4)
                    xxn,xyn,xzn,xln,yln,zln = Mid_Vect_Propeller(npln,l)
                    # This subroutine is used to calculate the midpoint

                    temp_T_1 = temp_T_1 + zln*V_Onset_P[npln,l,1] - yln*V_Onset_P[npln,l,2]
                    # Temporary variable used to calculate &T

                    temp_Q_11 = temp_Q_11 + xyn*yln*V_Onset_P[npln,l,0] - xyn*xln*V_Onset_P[npln,l,1]
                    # Temporary variable used to calculate &Q1

                    temp_Q_22 = temp_Q_22 + xzn*xln*V_Onset_P[npln,l,2] - xzn*zln*V_Onset_P[npln,l,0]
                    # Temporary variable used to calculate &Q2

                temp_T_0  = temp_T_0 + Weight_P[m,n] * temp_T_1
                temp_Q1_0 = temp_Q1_0 + Weight_P[m,n] * temp_Q_11
                temp_Q2_0 = temp_Q2_0 + Weight_P[m,n] * temp_Q_22
                # Temporary variable used to calculate T(Uo) (Nch Loop)
                # Temporary variable used to calculate Q1(Uo) (Nch Loop)
                # Temporary variable used to calculate Q2(Uo) (Nch Loop)

            matr_T [m,Var.Msp] = temp_T_0     # Value of &T(Uo) in the right position in the matrix (Temporary matrix matr_T)
            rhsQ[m,0] = - temp_Q1_0           # Value of &Q1(Uo) in the right position in the matrix (Temporary matrix rhs_Q)
            rhsQ[m,1] = - temp_Q2_0           # Value of &Q2(Uo) in the right position in the matrix (Temporary matrix rhs_Q)

        # Double loop used to calculate &T(Gam),&Q1(Gam),&Q2(Gam)

        # Loop used to select the line of the equation
        # (We don´t have the loop n because we already did that in Induced_Grid_Propeller)
        for m in range(Var.Msp):
            # Loop used to select the spanwise layer that induces velocity (Columns of the matrix) - Msp SUM
            for j in range (Var.Msp):
                temp_T_Gam = 0.0
                temp_Q1_Gam = 0.0
                temp_Q2_Gam = 0.0
                # Initialization of the temporary variable used to calculate &T(Gam)
                # Initialization of the temporary variable used to calculate &T(Gam)
                # Initialization of the temporary variable used to calculate &T(Gam)

                #Loop used to select where the point is located (chordwise) - First SUM Nch
                for n in range(Var.Nch):
                    npln = n + (m)*Var.Nch
                    #Panel where the point is located

                    temp_T_1 = 0.0
                    temp_Q_11 = 0.0
                    temp_Q_22 = 0.0
                    # Initialization of the temporary variable used to calculate &T
                    # Initialization of the temporary variable used to calculate &Q1
                    # Initialization of the temporary variable used to calculate &Q2

                    n_side = 4
                    if n == 0:
                        n_side = 3
                        # If we are considering the T.E. panel,
                        # instead of removing the value of the T.E. side, we skip it

                    for l in range (n_side):
                        # Loop used to select the side of the panel

                        xxn,xyn,xzn,xln,yln,zln =  Mid_Vect_Propeller(npln,l)
                        # This subroutine is used to calculate the midpoint

                        temp_T_1 = temp_T_1 + zln*V_Ind_P[j,npln,l,1] - yln*V_Ind_P[j,npln,l,2]
                        # Temporary variable used to calculate &T -
                        # Total thrust for that panel by j

                        temp_Q_11 = temp_Q_11 + xyn*yln*V_Ind_P[j,npln,l,0]- xyn*xln*V_Ind_P[j,npln,l,1]
                        # Temporary variable used to calculate &Q1 -
                        # Total torque 1 for that panel by j

                        temp_Q_22 = temp_Q_22 + xzn*xln*V_Ind_P[j,npln,l,2]- xzn*zln*V_Ind_P[j,npln,l,0]
                        # Temporary variable used to calculate &Q2 -
                        # Total torque 2 for that panel by j

                    temp_T_Gam = temp_T_Gam + Weight_P[m,n] *  temp_T_1
                    temp_Q1_Gam = temp_Q1_Gam + Weight_P[m,n] * temp_Q_11
                    temp_Q2_Gam = temp_Q2_Gam + Weight_P[m,n] * temp_Q_22
                    # Temporary variable used to calculate Q1 (Nch Loop)
                    # Temporary variable used to calculate Q2 (Nch Loop)
                    # Temporary variable used to calculate T (Nch Loop)

                for i in range (Var.Nch):
```

```
        # Loop used to select where the point is located (chordwise)
        # Second SUM Nch
            npli = i + (j)* Var.Nch

            temp_T_1 = 0.0
            temp_Q_11 = 0.0
            temp_Q_22 = 0.0
            # Initialization of the temporary variable used to calculate &T
            # Initialization of the temporary variable used to calculate &Q1
            # Initialization of the temporary variable used to calculate &Q2

            n_side = 4                 # If we are considering the T.E. panel,
            if i == 0:                 # instead of removing the value of the T.E. side, we skip it
                n_side = 3

            for l in range(n_side):
                xxi,xyi,xzi,xli,yli,zli = Mid_Vect_Propeller(npli,l)
                # This subroutine is used to calculate the midpoint

                temp_T_1 = temp_T_1 + zli*V_Ind_P[m,npli,l,1]- yli*V_Ind_P[m,npli,l,2]
                # Temporary variable used to calculate &T
                temp_Q_11 = temp_Q_11 + xyi*yli*V_Ind_P[m,npli,l,0] - xyi*xli*V_Ind_P[m,npli,l,1]
                #Temporary variable used to calculate &Q1
                temp_Q_22 = temp_Q_22 + xzi*xli*V_Ind_P[m,npli,l,2]- xzi*zli*V_Ind_P[m,npli,l,0]
                # Temporary variable used to calculate &Q2

            temp_T_Gam = temp_T_Gam + Weight_P[j,i] * temp_T_1
            temp_Q1_Gam = temp_Q1_Gam + Weight_P[j,i] * temp_Q_11
            temp_Q2_Gam = temp_Q2_Gam + Weight_P[j,i] * temp_Q_22
            # Temporary variable used to calculate T (Nch Loop)
            # Temporary variable used to calculate Q1 (Nch Loop)
            #  Temporary variable used to calculate Q2 (Nch Loop)

        matr_T[m,j] = temp_T_Gam
        matr_Q1[m,j] = temp_Q1_Gam
        matr_Q2[m,j] = temp_Q2_Gam
        # Value of &T(Gam) in the right position in the matrix (Temporary matrix matr_T)
        # Value of &Q1(Gam) in the right position in the matrix (Temporary matrix matr_T)
        # Value of &Q2(Gam) in the right position in the matrix (Temporary matrix matr_T)

# SYSTEM OF EQUATIONS - LOOP

V_Tot_P, V_Tot_No_Onset_P = Velocity_Total_Propeller ()
# It is used it in order to update V_Tot_P with the new values of gamma

# Loop for the T.E. panels (They don´t have the weight function)
for m in range(Var.Msp):
    npl0 = (m)*Var.Nch
    n_side = 3
    temp_T_Gam = 0.0

    for l in range(n_side):
        xxm,xym,xzm,xlm,ylm,zlm = Mid_Vect_Propeller(npl0,l)
        # This subroutine is used to calculate the midpoint

        temp_T_Gam = temp_T_Gam + zlm*V_Tot_P[npl0,l,1] - ylm*V_Tot_P[npl0,l,2]
        # Temporary variable used to calculate T (Nch Loop)

    matr_T[Var.Msp,m] = temp_T_Gam
    # Value of &T (T.E.) in the right position in the matrix (Temporary matrix matr_T)

    # Loop for the other panels (They don´t have the weight function)
    for n in range(1, Var.Nch):

        npl1 = n + (m)*Var.Nch
        n_side = 4
        temp_T_2_Gam = 0.0

        # Loop for the other panels
        for l in range(n_side):
            xxm,xym,xzm,xlm,ylm,zlm =  Mid_Vect_Propeller(npl1,l)
            # This subroutine is used to calculate the midpoint
            temp_T_2_Gam = temp_T_2_Gam + zlm*V_Tot_P[npl1,l,1] - ylm*V_Tot_P[npl1,l,2]
            # Temporary variable used to calculate T (Nch Loop)
        matr_T[Var.Msp,m] = matr_T[Var.Msp,m] + Weight_P[m,n]*temp_T_2_Gam
        # Value of &T in the right position in the matrix (Temporary matrix matr_T)

# CREATION OF THE MATRIX

lamba_t_1 = Gamma_TE_P[Var.Msp]        # Lagrange multiplier lambda t-1

for i in range (Var.Msp):
    rhs[i,0] = rhsQ[i,0] - rhsQ[i,1]   # rhs matrix

    matrix[i,Var.Msp] = matr_T[i,Var.Msp]       # System of equation (Left Matrix)
    matrix[Var.Msp,i] = matr_T[Var.Msp,i]       # System of equation (Left Matrix)

    for j in range (Var.Msp):
        matrix[i,j] = matr_Q1[i,j] - matr_Q2[i,j] + lamba_t_1*matr_T[i,j]
        # System of equation (Left Matrix)


rhs[Var.Msp,0] = Cs_T_r + (abs(T_fr_P))/Var.rho
# Total thrust required (Required + Skin Friction Drag Propeller)
matrix[Var.Msp,Var.Msp] = 0.0

# SYSTEM OF EQUATIONS - LOOP

rhs = np.linalg.solve(matrix, rhs)          #it solves the system of equations
#Computes the ""exact solution, x, of the well-determined, i.e.,
# full rank, linear matrix equation ax = b.

```

```
            # CONVERGENS OF THE SYSTEM

            res_0 = 0.0
            # Loop used to check if the residual is below a certain small limit
            for i in range(Var.Msp+1):
                res_1 = abs(1-Gamma_TE_P[i]/rhs[i,0])

                if res_1 > res_0:
                    res_0 = res_1
                Gamma_TE_P[i] = rhs[i,0]                # New values of circulation
            with open ("output/Propeller_Gamma_TE_P.txt","w") as file:
                for i in range (Var.Msp+1):
                    file.write(f"{Gamma_TE_P[i]:13.9f}\n")

            while (res_0 > Var.epsi):
                V_Tot_P, V_Tot_No_Onset_P = Velocity_Total_Propeller ()
                # It is used it in order to update V_Tot_P with the new values of gamma

                # Loop for the T.E. panels (They don´t have the weight function)
                for m in range(Var.Msp):
                    npl0 = (m)*Var.Nch
                    n_side = 3
                    temp_T_Gam = 0.0

                    for l in range(n_side):
                        xxm,xym,xzm,xlm,ylm,zlm = Mid_Vect_Propeller(npl0,l)
                        # This subroutine is used to calculate the midpoint

                        temp_T_Gam = temp_T_Gam + zlm*V_Tot_P[npl0,l,1] - ylm*V_Tot_P[npl0,l,2]
                        # Temporary variable used to calculate T (Nch Loop)

                    matr_T[Var.Msp,m] = temp_T_Gam
                    # Value of &T (T.E.) in the right position in the matrix (Temporary matrix matr_T)

                    # Loop for the other panels (They don´t have the weight function)
                    for n in range(1, Var.Nch):
                        npl1 = n + (m)*Var.Nch
                        n_side = 4
                        temp_T_2_Gam = 0.0

                        # Loop for the other panels
                        for l in range(n_side):
                            xxm,xym,xzm,xlm,ylm,zlm =  Mid_Vect_Propeller(npl1,l)
                            # This subroutine is used to calculate the midpoint
                            temp_T_2_Gam = temp_T_2_Gam + zlm*V_Tot_P[npl1,l,1] - ylm*V_Tot_P[npl1,l,2]
                            # Temporary variable used to calculate T (Nch Loop)
                        matr_T[Var.Msp,m] = matr_T[Var.Msp,m] + Weight_P[m,n]*temp_T_2_Gam
                        # Value of &T in the right position in the matrix (Temporary matrix matr_T)

                # CREATION OF THE MATRIX

                lamba_t_1 = Gamma_TE_P[Var.Msp]
                # Lagrange multiplier lambda t-1

                for i in range (Var.Msp):
                    rhs[i,0] = rhsQ[i,0] - rhsQ[i,1]
                    # rhs matrix

                    matrix[i,Var.Msp] = matr_T[i,Var.Msp]       # System of equation (Left Matrix)
                    matrix[Var.Msp,i] = matr_T[Var.Msp,i]       # System of equation (Left Matrix)

                    for j in range (Var.Msp):
                        matrix[i,j] = matr_Q1[i,j] - matr_Q2[i,j] + lamba_t_1*matr_T[i,j]
                        # System of equation (Left Matrix)

                rhs[Var.Msp,0] = Cs_T_r + (abs(T_fr_P))/Var.rho
                # Total thrust required (Required + Skin Friction Drag Propeller)
                matrix[Var.Msp,Var.Msp] = 0.0

                # SOLVE THE SYSTEM OF EQUATIONS

                rhs = np.linalg.solve(matrix, rhs)     #it solves the system of equations
                #Computes the ""exact solution, x, of the well-determined, i.e.,
                # full rank, linear matrix equation ax = b.

                # CONVERGENS OF THE SYSTEM

                res_0 = 0.0
                # Loop used to check if the residual is below a certain small limit
                for i in range(Var.Msp+1):
                    res_1 = abs(1-Gamma_TE_P[i]/rhs[i,0])

                    if res_1 > res_0:
                        res_0 = res_1
                    Gamma_TE_P[i] = rhs[i,0]                # New values of circulation
                with open ("output/Propeller_Gamma_TE_P.txt","w") as file:
                    for i in range (Var.Msp+1):
                        file.write(f"{Gamma_TE_P[i]:13.9f}\n")

            print('Iteration Propeller Number: {}'.format(iteration), 'Circulation on the propeller at the TE:')

            for i in range (Var.Msp+1):
                print (i,Gamma_TE_P[i])

            for i in range (Var.Msp):
                j = (i)*Var.Nch

                xx,xy,xz,xl,yl,zl = Mid_Vect_Propeller(j,3)
                #This subroutine is used to calculate the midpoint px,py,pz

                R_Circ_P[i] = np.sqrt(xy*xy + xz*xz)
                R_Circ_P_R[i] = R_Circ_P[i]/Var.Rad_P
```

```python

                Gamma_TE_P_No_dim[i] = (Gamma_TE_P[i]*100)/(np.pi*2*Var.Rad_P*Var.V_Ship)

            with open("output/Propeller_Gamma_TE.txt","w") as file:
                file.write("      Gamma_Dim      Gamma_No_Dim     Radius\n")
                for i in range (Var.Msp):
                    file.write(f"  {Gamma_TE_P[i]}   {Gamma_TE_P_No_dim[i]}   {R_Circ_P[i]}\n")

            with open ("output/Propeller_Gamma_TE_P.txt","w") as file:
                for i in range (Var.Msp+1):
                    file.write(f"{Gamma_TE_P[i]}\n")

            with open("output/Propeller_Print_Gamma_TE.txt","w") as file:
                for i in range(Var.Msp):
                    file.write(f" {Gamma_TE_P_No_dim[i]}\n")

            with open("output/Propeller_Print_Radius_TE.txt","w") as file:
                for i in range(Var.Msp):
                    file.write(f"{R_Circ_P_R[i]}\n")

            # DISTRIBUTION OF CIRCULATION AT THE REST OF THE BLADE

            for i in range(Var.Msp):
                npl_TE = i * Var.Nch
                Gamma_Panel_P[npl_TE] = Gamma_TE_P[i]

                for j in range (1,Var.Nch):
                    npl = j + i * Var.Nch
                    Gamma_Panel_P [npl] = Gamma_Panel_P[npl_TE] * Weight_P[i,j]

            with open ("output/Propeller_Gamma_Blade.txt","w") as file:
                file.write("  Panel          Gamma\n")
                for i in range(Var.Msp*Var.Nch):
                    file.write(f"   {i:3d}       {Gamma_Panel_P[i]:13.9f}\n")

            # ALIGNMENT OF THE WAKE

            pitch_0 = pitch()

            for i in range (Var.Msp+1):
                i_1 = i+i *(Var.N_P_L)
                pitch_0[i] = Points_Trans_Wake_P[i_1,2]
                # I need to save the old value of the pitch in order
                # to evaluate the residual for the pitch distribution

            Points_Trans_Wake_P, Grid_Points_P, Control_Points_P = Align_Wake_Propeller()
            res_0 = 0.0                                   # Initialization of the residual

            # Loop used to check if the residual is below a certain small limit
            for i in range(Var.Msp+1):
                i_1 = i + i*(Var.N_P_L)
                res_1 = abs(1 - (Points_Trans_Wake_P[i_1,2] / pitch_0[i]))
                if res_1 > res_0:
                    res_0 = res_1
        break

    return Gamma_TE_P_No_dim, R_Circ_P_R
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the advance ratio.
"""


import sources.Variables as Var
import numpy as np


def Advance_Ratio_J():

    r_R_P,X_P,Skew_P,Chord_P,Thick_P = np.loadtxt("input/grid.txt", unpack=True)
    U_O_P, U_R_P, U_T_P = np.loadtxt("input/onset.txt", unpack=True)

    n_0 = 50           # Number of intervals (It is used to find the "step length" for the composite Simpson's rule)
    n_1 = n_0          # Number of approximation values of the integral for the composite Simpson's rule
    h = (Var.Rad_P - Var.R_Hub_P)/n_0 # "step length"
    r_tmp = Var.R_Hub_P                # Initial value for the radius r

    Ua_tmp = 0           # Initialization of the approximation of the integral
    j = 1                # First value of j

    for i in range(n_1+1):                  # Simpson's rule used in order to solve the integral
        j = - j                             #It used to have 2 or 4 in the composite Simpson's rule
        Simpson = 3 + float(j)              # This value is 2 or 4
        if(i == 0 or i == n_1):             # This if is used to have 1 as coefficient if we are considering the first
            Simpson = 1                     # or the last value of the integral

        Ux_tmp = np.interp(r_tmp,r_R_P,U_O_P)
        # Linear interpolation used to find the value of the axial velocity
        Ua_tmp = Ua_tmp + (Ux_tmp * r_tmp) * Simpson
        # Composite Simpson's rule
        r_tmp = r_tmp + h

    # Advance velocity
    U_adv = 2 * h * Ua_tmp /(3*(Var.Rad_P**2 - Var.R_Hub_P**2))
    # Advance ratio
```

```
    Advance_ratio =  (U_adv * np.pi) / (Var.Omega * Var.Rad_P)
    # Wake fraction
    w_eff = 1 - U_adv/Var.V_Ship

    if (w_eff < 1.0 - 10):
        w_eff = 0

    # Open the file for writing
    with open("output/Propeller_Hydrodynamic_Characteristics.txt", "w") as file:
        # Write the header line
        file.write("{:2s}{:16s}{:4s}{:13s}{:4s}{:13s}\n".format("", "Advance velocity", "", "Advance ratio", "", "Wake fraction"))
        file.write("{:3s}{:13.9f}{:5s}{:13.9f}{:4s}{:13.9f}\n".format("", U_adv, "", Advance_ratio, "", w_eff))
    return Advance_ratio


Advance_ratio = Advance_Ratio_J()
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine contains a subroutine designed for optimizing
propeller flow through the generation of a new grid. It calculates induced
velocities at control points on mid-chord panels to determine a new beta
angle. This is crucial for the calculation of a new pitch, necessary for
grid generation. Pitch is interpolated at control points, with both blades
and trailers sharing the same pitch.
"""


import sources.Variables as Var
import numpy as np
from sources.Weight_Function_Propeller_P import Weight_function_propeller
from sources.Induced_Grid_Propeller_P import Induced_Grid_Propeller
from sources.Panel_Induced_Velocity_Propeller_Align import Panel_Induced_Velocity_Propeller_Align
from sources.Trailing_Vortices_Propeller_P import Trailing_Vortices_Propeller


def Align_Wake_Propeller():

    Grid_Points_P = np.zeros(((Var.Msp + 1) * (Var.Nch + 1), 3))
    Control_Points_P = np.zeros(((Var.Msp) * (Var.Nch), 3))
    Radius_cp_P = np.zeros((Var.Msp + 1))
    r_R_P,X_P,Skew_P,Chord_P,Thick_P = np.loadtxt("input/grid.txt", unpack= True)
    U_0_P, U_R_P, U_T_P = np.loadtxt("input/onset.txt",unpack=True)
    Gamma_TE_P = np.loadtxt("output/Propeller_Gamma_TE_P.txt")
    Control_Points_P = np.loadtxt('output/Propeller_Control_Points.txt')
    t_gp_P = np.loadtxt("output/Propeller_t_gp.txt", skiprows = 1)
    t_cp_P = np.loadtxt("output/Propeller_t_cp.txt", skiprows = 1)
    s_gp_P = np.loadtxt("output/Propeller_s_gp.txt", skiprows = 1)
    s_cp_P = np.loadtxt("output/Propeller_s_cp.txt", skiprows = 1)
    Weight_P = Weight_function_propeller()
    N_Bound_Vortex_P = np.loadtxt("output/Propeller_N_Bound_Vortex.txt",dtype= 'int')
    N_Bound_Vortex_P = N_Bound_Vortex_P.reshape((Var.Msp+1, 1))
    data_matrix = np.loadtxt("output/Propeller_Grid_Points_geom.txt")
    Radius_gp_P = data_matrix[:, 0]
    Chord_P_gp = data_matrix[:, 1]
    Rake_P_gp = data_matrix[:, 2]
    Skew_P_gp = data_matrix[:, 3]
    data_matrix = np.loadtxt("output/Propeller_Control_Points_geom.txt")
    Chord_P_cp = data_matrix[:, 0]
    Rake_P_cp = data_matrix[:, 1]
    Skew_P_cp = data_matrix[:, 2]

    # SUBROUTINE
    r_cp = np.zeros(Var.Msp)      # Radius in the control points of the propeller where the new pitch is computed
    tan_beta = np.zeros(Var.Msp)
    beta = np.zeros (Var.Msp)
    pitch_cp = np.zeros(Var.Msp)
    pitch_gp = np.zeros(Var.Msp+1)
    sin_b = np.zeros(Var.Msp + 1)
    cos_b = np.zeros(Var.Msp + 1)
    Theta_gp_P = np.zeros(Var.Nch + 1)
    Theta_cp_P = np.zeros(Var.Nch + 1)

    mid_point = (Var.Nch//2 + 1)

    # Loop used to select the closest control points to the midchord line (Chordwise)
    for j in range (Var.Msp):
        mid_point_cp = (mid_point + j * Var.Nch) -1

        p_x_mdp = Control_Points_P[mid_point_cp,0]     # X coordinate of the chosen control point of the propeller
        p_y_mdp = Control_Points_P[mid_point_cp,1]     # Y coordinate of the chosen control point of the propeller
        p_z_mdp = Control_Points_P[mid_point_cp,2]     # Z coordinate of the chosen control point of the propeller

        r_cp[j] = np.sqrt(p_y_mdp**2 + p_z_mdp**2)
        # Radius for the chosen control point of the propeller

        # VELOCITIES IN THE CONTROL POINTS FROM THE PANELS OF THE PROPELLER

        # Initialization of the variable used to store the induced velocity
        # from the panels of the propeller (x), (y), (z)
        u_x_panels = 0
        u_y_panels = 0
        u_z_panels = 0

        # Loop used to select the spanwise level that induces velocity
        # on the control points of the propeller
```

```
        for n in range (Var.Msp):

            # Initialization of the variable used to calculate the induced
            # velocity from the panels of the propeller (x), (y), (z)
            u_x_panels_0 = 0
            u_y_panels_0 = 0
            u_z_panels_0 = 0

            # Loop used to select the panel that induces velocity
            # on the control points of the propeller
            for m in range (Var.Nch):
                npl = m + n * Var.Nch

                u_x_temp,u_y_temp,u_z_temp = Panel_Induced_Velocity_Propeller_Align(npl,0,0,p_x_mdp,p_y_mdp,p_z_mdp)
                # Induced velocity from the selected panel on
                # the chosen control point of the propeller - No bound vortex

                u_x_panels_0 = u_x_panels_0 + Weight_P[n,m] * u_x_temp
                # Temporary variable used to calculate the induced velocity
                # from the panels of the propeller (x)
                u_y_panels_0 = u_y_panels_0 + Weight_P[n,m] * u_y_temp
                # Temporary variable used to calculate the induced velocity
                # from the panels of the propeller (y)
                u_z_panels_0 = u_z_panels_0 + Weight_P[n,m] * u_z_temp
                # Temporary variable used to calculate the induced velocity
                # from the panels of the propeller (z)

            u_x_panels = u_x_panels + Gamma_TE_P[n] * u_x_panels_0        # Induced velocity from the panels of the propeller (x)
            u_y_panels = u_y_panels + Gamma_TE_P[n] * u_y_panels_0        # Induced velocity from the panels of the propeller (y)
            u_z_panels = u_z_panels + Gamma_TE_P[n] * u_z_panels_0        # Induced velocity from the panels of the propeller (z)

        # VELOCITIES IN THE CONTROL POINTS FROM THE HORSESHOE VORTEX OF THE PROPELLER

        u_x_trail = 0
        # Initialization of the variable used to calculate the induced velocity
        # from the trailing vortices of the propeller (x)
        u_y_trail = 0
        # Initialization of the variable used to calculate the induced velocity
        # from the trailing vortices of the propeller (y)
        u_z_trail = 0
        # Initialization of the variable used to calculate the induced velocity
        # from the trailing vortices of the propeller (z)

        u_x_trail_1,u_y_trail_1,u_z_trail_1 = Trailing_Vortices_Propeller(0,p_x_mdp,p_y_mdp,p_z_mdp)
        # Induced velocity from the transition wake and from
        # the semi-infinite helicoidal vortex of the propeller (First)

        for n in range (Var.Msp):             # Loop used to select the trailing vortex that induces velocity
            n_1 = n + 1                       # on the control points of the propeller
            n_2 = (n+1) * (Var.Nch+1)

            u_x_trail_2,u_y_trail_2,u_z_trail_2 = Trailing_Vortices_Propeller(n_1,p_x_mdp,p_y_mdp,p_z_mdp)
            # Induced velocity from the transition wake and from the semi-infinite
            # helicoidal vortex of the propeller (Second) selected of the propeller

            u_x_trail = u_x_trail + Gamma_TE_P[n] * (u_x_trail_1 - u_x_trail_2)
            # Induced velocity from the horseshoe vortex of the propeller (x)
            # No bound vortex
            u_y_trail = u_y_trail + Gamma_TE_P[n] * (u_y_trail_1 - u_y_trail_2)
            # Induced velocity from the horseshoe vortex of the propeller (y)
            # No bound vortex
            u_z_trail = u_z_trail + Gamma_TE_P[n] * (u_z_trail_1 - u_z_trail_2)
            # Induced velocity from the horseshoe vortex of the propeller (z)
            # No bound vortex

            u_x_trail_1 = u_x_trail_2       # For the next loop
            u_y_trail_1 = u_y_trail_2       # For the next loop
            u_z_trail_1 = u_z_trail_2       # For the next loop

        # TOTAL INDUCED VELOCITY

        u_x_tot = u_x_trail + u_x_panels    # Total induced velocity on the propeller (x)
        u_y_tot = u_y_trail + u_y_panels    # Total induced velocity on the propeller (y)
        u_z_tot = u_z_trail + u_z_panels    # Total induced velocity on the propeller (z)

        # BETA AND PITCH AT THE CONTROL POINTS

        cos_theta = p_z_mdp / r_cp[j]
        sin_theta = p_y_mdp / r_cp[j]

        U_T_P_tot = - u_y_tot * cos_theta + u_z_tot * sin_theta
        # Total tangential induced velocity in the control points of the propeller

        U_0_P_beta = np.interp(r_cp[j],r_R_P,U_0_P)          # Wake (Axial) in the control points (s)
        U_T_P_beta = np.interp(r_cp[j],r_R_P,U_T_P)          # Wake (Tangential) in the control points (s)

        tan_beta[j] = abs(-U_0_P_beta + u_x_tot)/(Var.Omega * r_cp[j] - U_T_P_tot - U_T_P_beta)         # New tangent beta

        beta[j] = np.arctan(tan_beta[j])

        pitch_cp[j] = tan_beta[j] * 2 * np.pi * r_cp[j]

    with open("output/Propeller_Pitch_Control_Points.txt","w")as file:
        file.write(" Spanw.      Radius           Pitch/D\n")
        for j in range (Var.Msp):
            file.write(f" {j:3d}    {r_cp[j]:13.9f}    {pitch_cp[j]/(Var.Rad_P*2):13.9f}\n")

    with open ("output/Propeller_Beta.txt","w") as file:
        file.write("    Radius            Beta\n")
        for j in range (Var.Msp):
            file.write(" {:13.9f}{:3s}{:13.9f}\n".format(r_cp[j],"", np.arctan(tan_beta[j])))

```

```
    # INTERPOLATION OF THE PITCH

    if Var.Msp == 0:
        pitch_cp[Var.Msp-1] = 0

    # This loop is used to find the values of the pitch in the grid points
    # of the propeller (No tip - No Hub)
    for i in range (1,Var.Msp):
        pitch_gp[i] = np.interp(s_gp_P[i],s_cp_P,pitch_cp)

    pitch_gp[0] = (pitch_cp[1] - pitch_cp[0])/(s_cp_P[1]-s_cp_P[0])*(s_gp_P[0] - s_cp_P[0]) + pitch_cp[0] # Pitch at the hub

    pitch_gp[Var.Msp] = (pitch_cp[Var.Msp-1] - pitch_cp[Var.Msp-2])/(s_cp_P[Var.Msp-1] - s_cp_P[Var.Msp - 2] # Pitch at the tip
        )*(s_gp_P[Var.Msp] - s_cp_P[Var.Msp - 2])+ pitch_cp[Var.Msp - 2]

    with open("output/Propeller_Pitch_Grid_Points.txt","w") as file:
        file.write(" Spanw.     Pitch\n")

        for i in range (Var.Msp+1):
            file.write(f" {i:3d}{pitch_gp[i]:13.9f}\n")

    # GRID POINTS MATRIX -  CALCULATION OF BETA(S(R)),CHORD(S),SKEW(S) AND RAKE(S)

    for i in range (Var.Msp+1):

        ipl = ((i+1)*(Var.Nch+1))-(Var.Nch+1)
        # Counter used to order the Grid Points Matrix
        p_ref_gp = np.sqrt(pitch_gp[i]**2 + (2*np.pi*Radius_gp_P[i])**2)
        # Reference pitch (It has only a radial variation)
        sin_b[i] = pitch_gp[i]/p_ref_gp
        # sin(beta)
        cos_b[i] = 2*np.pi*Radius_gp_P[i]/p_ref_gp
        # cos(beta)
        for j in range (Var.Nch+1):

            npl = (j) + ipl       # Second counter to order the Grid Points Matrix
            Theta_gp_P[j] = -Skew_P_gp[i] + (t_gp_P[j] * Chord_P_gp[i] * cos_b[i]) / Radius_gp_P[i]

            # X(s,t)
            Grid_Points_P[npl, 0] = Rake_P_gp[i] + Chord_P_gp[i] * sin_b[i] * t_gp_P[j]
            # Y(s,t)
            Grid_Points_P[npl, 1] = - Radius_gp_P[i] * np.sin(Theta_gp_P[j])
            # Z(s,t)
            Grid_Points_P[npl, 2] = Radius_gp_P[i] * np.cos(Theta_gp_P[j])

    with open('output/Propeller_Grid_Points.txt', 'w') as file:
        for i in range((Var.Nch + 1) * (Var.Msp + 1)):
            file.write(f" {Grid_Points_P[i, 0]:.9f}      {Grid_Points_P[i, 1]:.9f}      {Grid_Points_P[i, 2]:.9f}\n")

     # GRID CONTROL POINTS MATRIX - CALCULATION OF BETA(S(R)),CHORD(S),SKEW(S) AND RAKE(S)

    for i in range (Var.Msp):           # Counter used to order the Grid Control Points Matrix
        ipl = ((i+1)*(Var.Nch))-(Var.Nch)

        Radius_cp_P[i] = 0.5 * (Radius_gp_P[i] + Radius_gp_P[i+1])

        p_ref_gp = np.sqrt(pitch_cp[i]**2 + (2*np.pi*Radius_cp_P[i])**2)        # Reference pitch (It has only a radial variation)

        sin_b[i] = pitch_cp[i]/p_ref_gp                     # sin(beta)
        cos_b[i] = 2*np.pi*Radius_cp_P[i]/p_ref_gp          # cos(beta)

        for j in range (Var.Nch):                           # t Loop
            npl = (j) + ipl                                 # Second counter to order the Grid Points Matrix
            Theta_cp_P[j] = - Skew_P_cp[i] + (t_cp_P[j] * Chord_P_cp[i] * cos_b[i]) / Radius_cp_P[i]

            # X(s,t)
            Control_Points_P[npl, 0] = Rake_P_cp[i] + Chord_P_cp[i] * sin_b[i] * t_cp_P[j]
            # Y(s,t)
            Control_Points_P[npl, 1] = - Radius_cp_P[i] * np.sin(Theta_cp_P[j])
            # Z(s,t)
            Control_Points_P[npl, 2] = Radius_cp_P[i] * np.cos(Theta_cp_P[j])

    with open('output/Propeller_Control_Points.txt', 'w') as file:
        for i in range((Var.Nch) * (Var.Msp)):
            file.write(f"{Control_Points_P[i, 0]:13.9f}      {Control_Points_P[i, 1]:13.9f}      {Control_Points_P[i, 2]:13.9f}\n")

    # CREATION OF THE TRANSITION WAKE (STRAIGHT LINE VORTICES)

    Points_Trans_Wake_P = np.zeros((((Var.N_P_L+1)*(Var.Msp+1)),3))

    for i in range(Var.Msp+1):
        i_1 = i+i*(Var.N_P_L)

        x_trans_wake = Grid_Points_P[N_Bound_Vortex_P[i,0],0]           # X value for the first point of the transition wake - T.E.
        y_trans_wake = Grid_Points_P[N_Bound_Vortex_P[i,0],1]           # Y value for the first point of the transition wake - T.E.
        z_trans_wake = Grid_Points_P[N_Bound_Vortex_P[i,0],2]           # Z value for the first point of the transition wake - T.E.

        pitch_trans_wake = pitch_gp[i]                            # Pitch at the T.E. (It has only a radial variation)

        r_trans_wake = np.sqrt(y_trans_wake**2 + z_trans_wake**2)        # Radius at the T.E.

        Points_Trans_Wake_P[i_1,0] = x_trans_wake      # Grid points for the transition wake (x) - T.E
        Points_Trans_Wake_P[i_1,1] = r_trans_wake      # Grid points for the transition wake (radius) - T.E
        Points_Trans_Wake_P[i_1,2] = pitch_trans_wake  # Grid points for the transition wake (pitch) - T.E

        delta_trans_wake = (-4 * Var.Rad_P - x_trans_wake)/(Var.N_P_L)      # The transition wake goes four radii downstream

        for j in range(Var.N_P_L):   # Loop used to divide the transition wake in N_P_L parts (N_P_L+1 points)
            i_2 = (i_1) + j+1

            # Grid points for the transition wake
            Points_Trans_Wake_P[i_2,0] = x_trans_wake + (j+1) * delta_trans_wake # (x)
```

```
            Points_Trans_Wake_P[i_2,1] = r_trans_wake                             # (radius)
            Points_Trans_Wake_P[i_2,2] = pitch_trans_wake                         # (pitch)

    with open("output/Propeller_Points_Trans_Wake.txt", "w") as file:
        file.write(f"{'Point':<8}{'x':<12}{'r':<20}{'p':<20}\n")
        for i in range(Var.Msp+1):
            i_1 = i+i*(Var.N_P_L)
            file.write(f"{i:<5}{Points_Trans_Wake_P[i_1,0]:13.9f}{Points_Trans_Wake_P[i_1 ,1]:13.9f}"
                       f"{Points_Trans_Wake_P[i_1,2]:13.9f}\n")

            for j in range(Var.N_P_L):
                i_2 = (i_1) + j+1
                file.write(f"{i:<5}{Points_Trans_Wake_P[i_2,0]:13.9f}{Points_Trans_Wake_P[i_2,1]:13.9f}"
                           f"{Points_Trans_Wake_P[i_2,2]:13.9f}\n")

    return Points_Trans_Wake_P, Grid_Points_P, Control_Points_P
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the area of a panel given its four
points.
"""


import numpy as np


def Area_Panel(x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3, x_4, y_4, z_4):


    s = 0                              # Initialization of the variable

    b_1 = x_4 - x_1                       # X value of the first vector of the panel (Point 1 and Point 4)
    b_2 = y_4 - y_1                       # Y value of the first vector of the panel (Point 1 and Point 4)
    b_3 = z_4 - z_1                       # Z value of the first vector of the panel (Point 1 and Point 4)

    e_1 = x_3 - x_1                       # First side (x)
    e_2 = y_3 - y_1                       # First side (y)
    e_3 = z_3 - z_1                       # First side (z)

    f_1 = x_2 - x_1                       # Second side (x)
    f_2 = y_2 - y_1                       # Second side (y)
    f_3 = z_2 - z_1                       # Second side (z)

    s_11 = f_2*b_3 - f_3*b_2              # X component of the first cross product
    s_12 = b_1*f_3 - f_1*b_3              # Y component of the first cross product
    s_13 = f_1*b_2 - f_2*b_1              # Z component of the first cross product

    s_21 = b_2*e_3 - b_3*e_2              # X component of the second cross product
    s_22 = e_1*b_3 - b_1*e_3              # Y component of the second cross product
    s_23 = b_1*e_2 - b_2*e_1              # Z component of the second cross product

    s = 0.5*(np.sqrt(s_11**2 + s_12**2 + s_13**2) + np.sqrt(s_21**2 + s_22**2 + s_23**2))  #Area of the panel

    return (s)
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the velocities Ux, Uy, Uz from a line
element (x1,y1,z1) to (x2,y2,z2) at the point (px,py,pz) using Biot-Savart's
law. The process applies to all blades of the propeller. Given the propeller's
symmetry, only the point on the reference blade is needed. The induced velocity
from the vortex is calculated with a unit circulation.
"""


import numpy as np


def Biot_Savart_Propeller(I_z, x_1, y_1, z_1, x_2, y_2, z_2, px, py, pz):

    U_x, U_y, U_z = 0.0, 0.0, 0.0

    d_theta = (2*np.pi)/float(I_z)  # Angle between the blades
    theta = 0.0                     # Angle for the first blade
    a_x = x_2 - x_1
    d_y = y_2 - y_1
    d_z = z_2 - z_1

    b_x = px - x_2
    c_x = px - x_1

    for i in range(I_z):
        cos_theta = np.cos(theta)
        sin_theta = np.sin(theta)

        a_y = d_y * cos_theta - d_z * sin_theta
        a_z = d_z * cos_theta + d_y * sin_theta

        b_y = py - y_2 * cos_theta + z_2 * sin_theta
        b_z = pz - z_2 * cos_theta - y_2 * sin_theta
```

```
        c_y = py - y_1 * cos_theta + z_1 * sin_theta
        c_z = pz - z_1 * cos_theta - y_1 * sin_theta

        a_length = np.sqrt(a_x*a_x + a_y*a_y + a_z*a_z)  # Lenght a
        b_length = np.sqrt(b_x*b_x + b_y*b_y + b_z*b_z)  # Lenght b
        c_length = np.sqrt(c_x*c_x + c_y*c_y + c_z*c_z)  # Lenght c

        a_c = a_x*c_x + a_y*c_y + a_z*c_z  # Dot product a.c (e)
        a_b = a_x*b_x + a_y*b_y + a_z*b_z  # Dot product a.b (c-e)

        ac_x = a_y*c_z - a_z*c_y  # X component of the cross product a^c
        ac_y = a_z*c_x - a_x*c_z  # Y component of the cross product a^c
        ac_z = a_x*c_y - a_y*c_x  # Z component of the cross product a^c

        aclen2 = ac_x*ac_x + ac_y*ac_y + ac_z*ac_z
        aclen = np.sqrt(aclen2)  # Module of the cross product a^c

        # This if is used to check the distance between the selected point and
        # the side. If they are too close we have to skip it
        if a_length != 0 and (aclen / a_length) > 1*10**(-5):

            cstac = a_c/c_length  # e/c
            cstab = a_b/b_length  # a-e / b
            cstv = 1.0 / (4.0 * np.pi * aclen2)
            cstv1 = cstv*cstac - cstv*cstab

            U_x = U_x + ac_x * cstv1  # Induced Velocity (x)
            U_y = U_y + ac_y * cstv1  # Induced Velocity (y)
            U_z = U_z + ac_z * cstv1  # Induced Velocity (z)

        theta = theta + d_theta

    return (U_x, U_y, U_z)
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the Ci-function used in 'De_Jong' by
rational approximations.
"""


import numpy as np


def Ci(xbar):


    f = (xbar**8+38.027264*xbar**6+265.187033*xbar**4+335.677320*xbar**2+38.102495)/(
    xbar**8+40.021433*xbar**6+322.624911*xbar**4+570.236280*xbar**2+157.105423)/xbar

    g = (xbar**8+42.242855*xbar**6+302.757865*xbar**4+352.018498*xbar**2+21.821899)/(
    xbar**8+48.196927*xbar**6+482.485984*xbar**4+1114.978885*xbar**2+449.690326)/xbar**2

    Cires=f*np.sin(xbar)-g*np.cos(xbar)

    return(Cires)
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the induced velocities Ux,Uy,Uz from a
semi infinitly vortex in the point px,py,pz. The calculations are made for 1
blade. The routine uses the helix radius r, pitch p and longitudinal starting
point x as input. The vortex starts at -infinity and stops at -x. The
calculations follows the procedure outlined by de Jong.
"""


import numpy as np


def De_Jong(x, r, p, phi, px, py, pz):

    from sources.Ci_P import Ci
    from sources.Si_P import si

    Ux = 0.0                # Initialization  of the variable U_x
    Uy = 0.0                # Initialization  of the variable U_y
    Uz = 0.0                # Initialization  of the variable U_z

    # CONSTANTS

    pbar=2*np.pi/p
    xbar=pbar*x
    x2bar=2*xbar
    xtld=pbar*px
    x2tld=2*xtld

    xsum=xbar+xtld
    xsum2=2*xsum*xsum
    xsum3=1.5*xsum2*xsum
```

```
    xsum4=xsum2*xsum2

    x2sum=2*xsum
    x2sum2=2*x2sum*x2sum
    x2sum3=1.5*x2sum2*x2sum
    x2sum4=x2sum2*x2sum2

    r2=r*r
    py2=py*py
    pz2=pz*pz
    rbar=r2+py2+pz2
    p2=pbar*pbar
    p3r2=p2*r2*pbar
    p4r2=3*p3r2*pbar
    p5r2=p4r2*pbar
    p2x2=p2/xsum2
    p4rb=3*p2*p2*rbar/xsum4

    cxtld=np.cos(xtld)
    sxtld=np.sin(xtld)
    cx2tld=np.cos(x2tld)
    sx2tld=np.sin(x2tld)
    cxbar=np.cos(xbar)
    sxbar=np.sin(xbar)
    cx2bar=np.cos(x2bar)
    sx2bar=np.sin(x2bar)
    cxbar2=cxbar*cxbar
    sxbar2=sxbar*sxbar

    cphi=np.cos(phi)
    sphi=np.sin(phi)
    cphi2=cphi*cphi
    sphi2=sphi*sphi
    phi2=2*phi
    c2phi=np.cos(phi2)
    s2phi=np.sin(phi2)

    Ci1 = Ci(xsum)
    Ci2 = Ci(x2sum)
    si1 = si(xsum)
    si2 = si(x2sum)

    cos11  = -cxtld*Ci1-sxtld*si1
    cos112 = -cx2tld*Ci2-sx2tld*si2

    sin11  = sxtld*Ci1-cxtld*si1
    sin112 = sx2tld*Ci2-cx2tld*si2

    cos12  = cxbar/xsum-sin11
    cos122 = cx2bar/x2sum-sin112

    sin12  = sxbar/xsum+cos11
    sin122 = sx2bar/x2sum+cos112

    cos13  = cxbar/xsum2-0.5*sin12
    cos132 = cx2bar/x2sum2-0.5*sin122

    sin13  = sxbar/xsum2+0.5*cos12
    sin132 = sx2bar/x2sum2+0.5*cos122

    cos23  = cxbar2/xsum2-sin122
    sin23  = sxbar2/xsum2+sin122

    cos14  = cxbar/xsum3-sin13/3
    cos142 = cx2bar/x2sum3-sin132/3

    sin14  = sxbar/xsum3+cos13/3
    sin142 = sx2bar/x2sum3+cos132/3

    cos24  = cxbar2/xsum3-4*sin132/3

    sin24  = sxbar2/xsum3+4*sin132/3

    cos15  = cxbar/xsum4-0.25*sin14
    cos152 = cx2bar/x2sum4-0.25*sin142

    sin15  = sxbar/xsum4+0.25*cos14
    sin152 = sx2bar/x2sum4+0.25*cos142

    cos25  = cxbar2/xsum4-2*sin142
    sin25  = sxbar2/xsum4+2*sin142

    # CALCULATION OF UX

    p3 = p2*pbar
    p4 = p3*pbar
    p5 = p4*pbar
    rbar3r = rbar+2*r2

    c0 = -p5r2*rbar/xsum4/2+p3r2/xsum2
    c1 = -p3*r*pz
    c2 = -p3*r*py
    c3 = 3*p5*r*pz*rbar3r/2
    c4 = 3*p5*r*py*rbar3r/2
    c5 = -p5r2*pz2
    c6 = -p5r2*py2
    c7 = -16*p5r2*py*pz

    Ux = (c0+(c1*cphi-c2*sphi)*cos13 + (c2*cphi+c1*sphi)*sin13 + (c3*cphi-c4*sphi)*cos15+(c4*cphi+c3*sphi)*sin15 +
          (c5*cphi2+c6*sphi2)*cos25 + (c6*cphi2+c5*sphi2)*sin25 + ((c5-c6)*8*s2phi+c7*c2phi)*sin152 - c7*s2phi*cos152)/4/np.pi

    # CALCULATION OF UY
```

```

    d0 =-p2*pz/xsum2+1.5*p4*pz*rbar/xsum4
    d1 = p2*r
    d2 = d1
    d3 = -3*p4*r*rbar/2
    d4 = 4*p4r2*pz
    d5 = p4r2*py
    d6 = -3*p4*r*py*pz
    d7 = 8*p4r2*py
    d8 = -3*p4*r*(py2+3*pz2+r2)/2
    d9 = p4r2*pz

    Uy = (d0+d1*cphi*cos13+d1*sphi*sin13+(d8*cphi-d6*sphi)*cos15 + (d8*sphi+d6*cphi)*sin15+0.25*d4*cphi*cphi*cos25 +
          0.25*d4*sphi*sphi*sin25 + (8*d5*c2phi+2*d4*s2phi)*sin152 - d7*s2phi*cos152-d1*sphi*cos12+d1*cphi*sin12-
          d3*sphi*cos14 + d3*cphi*sin14+d5*sphi*sphi*cos24+d5*cphi*cphi*sin24 + (d4*c2phi-0.5*d7*s2phi)*sin142-
          d4*s2phi*cos142)/4/np.pi

    # CALCULATION OF UZ

    e0 = p2*py/xsum2-1.5*p4*py*rbar/xsum4
    e1 = p2*r
    e2 = -e1
    e3 = -3*p4*r*rbar/2
    e4 = 4*p4r2*py
    e5 = p4r2*pz
    e6 = 3*p4*r*py*pz
    e7 = 3*p4*r*(3*py2+pz2+r2)/2
    e8 = -8*p4r2*pz
    e9 = -p4r2*py

    Uz = (e0+e1*cphi*cos12+e1*sphi*sin12+e1*sphi*cos13-e1*cphi*sin13 + e3*cphi*cos14+e3*sphi*sin14+
         (e4*c2phi-0.5*e8*s2phi)*sin142 - e4*s2phi*cos142+e5*cphi*cphi*cos24+e5*sphi*sphi*sin24 -
          e7*sphi*cos15+e7*cphi*sin15+e6*cphi*cos15+e6*sphi*sin15 - e8*s2phi*cos152+
          (2*e4*s2phi+e8*c2phi)*sin152 + e9*sphi*sphi*cos25+e9*cphi*cphi*sin25)/4/np.pi

    return (Ux,Uy,Uz)
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This function computes the propeller efficiency.
"""


import sources.Variables as Var
import numpy as np
from sources.Mid_Vect_Propeller_P import Mid_Vect_Propeller
from sources.Weight_Function_Propeller_P import Weight_function_propeller
from sources.Skin_Friction_Drag_P import Skin_Friction_Drag
from sources.Advance_Ratio_P import Advance_Ratio_J


def Efficiency():

    I_P_Points_P = (Var.Msp*Var.Nch)
    Panel, Gamma_Panel_P = np.loadtxt("output/Propeller_Gamma_Blade.txt",skiprows = 1,unpack= True)
    T_fr_P, Q_fr_P = Skin_Friction_Drag()
    Weight_P = Weight_function_propeller()
    V_Tot_P = np.loadtxt("output/Propeller_Velocity_Total.txt", skiprows=2, usecols= (2,3,4))
    V_Tot_P = np.reshape(V_Tot_P, (I_P_Points_P, 4, 3))
    Advance_ratio = Advance_Ratio_J()

    # THRUST AND TORQUE (WITHOUT SKIN FRICTION DRAG)

    Thr = 0
    Tor = 0

    T_tot_P = 0                 # Initialization of the temporary variable used to calculate T
    Tor_tot_P = 0

    Q1_tot = 0                  # Initialization of the temporary variable used to calculate Q1
    Q2_tot = 0                  # Initialization of the temporary variable used to calculate Q2

    for m in range (Var.Msp):            # Spanwise loop
        npl_TE = (m-1)*Var.Nch

        T_0 = 0           # Initialization of the temporary variable used to calculate T
        Q_10 = 0          # Initialization of the temporary variable used to calculate Q1
        Q_20 = 0          # Initialization of the temporary variable used to calculate Q2

        for n in range (Var.Nch):                 # Chordwise loop
            npl = n + (m-1)*Var.Nch

            T_00 = 0            # Initialization of the temporary variable used to calculate T
            Q_100 = 0           # Initialization of the temporary variable used to calculate Q1
            Q_200 = 0           # Initialization of the temporary variable used to calculate Q2

            for k in range (4):                 # Panel loop
                xkx,xky,xkz,xlk,ylk,zlk = Mid_Vect_Propeller(npl,k)
                # This subroutine is used to calculate the characteristics of the side k panel npl

                T_00 = T_00 + zlk*V_Tot_P[npl,k,1] - ylk*V_Tot_P[npl,k,2]
                # Thrust generated by side k panel npl without taking into account of the weight function

                Q_100 = Q_100 + xky*ylk*V_Tot_P[npl,k,0] - xky*xlk*V_Tot_P[npl,k,1]
                #Torque Q1 generated by side k panel npl without taking into account of the weight function

```

```
                Q_200 = Q_200 + xkz*xlk*V_Tot_P[npl,k,2]- xkz*zlk*V_Tot_P[npl,k,0]
                #Torque Q2 generated by side k panel npl without taking into account of the weight function

            T_0 = T_0 + Weight_P[m,n] * T_00
            # Thrust generated by the panel npl taking into account of the weight function
            Q_10 = Q_10 + Weight_P[m,n] * Q_100
            # Torque Q1 generated by the panel npl taking into account of the weight function
            Q_20 = Q_20 + Weight_P[m,n] * Q_200
            # Torque Q2 generated by the panel npl taking into account of the weight function

        xkx,xky,xkz,xlk,ylk,zlk = Mid_Vect_Propeller(npl_TE,3)

        T_tot_P = T_tot_P + Gamma_Panel_P[npl_TE]*T_0 - Gamma_Panel_P[npl_TE
        ]*zlk*V_Tot_P[npl_TE,3,1] + Gamma_Panel_P[npl_TE]*ylk*V_Tot_P[npl_TE,3,2]          # No thrust generated by T.E. side

        Q1_tot = Q1_tot + Gamma_Panel_P[npl_TE]*Q_10 - Gamma_Panel_P[npl_TE
        ]*xky*ylk*V_Tot_P[npl_TE,3,0] + Gamma_Panel_P[npl_TE]*xky*xlk*V_Tot_P[npl_TE,3,1]  # No torque generated by T.E. side

        Q2_tot = Q2_tot + Gamma_Panel_P[npl_TE]*Q_20 - Gamma_Panel_P[npl_TE
        ]*xkz*xlk*V_Tot_P[npl_TE,3,2] + Gamma_Panel_P[npl_TE]*xkz*zlk*V_Tot_P[npl_TE,3,0]  # No torque generated by T.E. side

    # EFFICIENCY

    Thr = Var.rho*float(Var.Z_Blade_P)*T_tot_P + T_fr_P*Var.Z_Blade_P    # Total thrust given by the propeller
    Tor = Var.rho*float(Var.Z_Blade_P)*Q1_tot - Var.rho*float(Var.Z_Blade_P)*Q2_tot + Q_fr_P*Var.Z_Blade_P
    # Total torque given by the propeller

    K_T = Thr / (Var.rho * (Var.Omega/(2*np.pi))**2 * (Var.Rad_P*2)**4)    # Thrust coefficient

    K_Q = Tor / (Var.rho * (Var.Omega/(2*np.pi))**2 * (Var.Rad_P*2)**5)    # Torque coefficient

    Eff = Advance_ratio * K_T / abs(2 * np.pi * K_Q)                       # Efficiency
    C_th = Thr/(0.5*Var.rho*Var.V_Ship**2*np.pi*Var.Rad_P**2)

    with open("output/Propeller_Efficiency.txt", mode='w') as file:
        file.write("Efficiency\n")
        file.write("{:13.9f}\n".format(Eff))

    with open("output/Propeller_Forces.txt", mode='w') as file:
        file.write("      K_T             K_Q             T             Q             Cth\n")
        file.write("{:13.9f}   {:13.9f}   {:10.1f}   {:10.1f}   {:13.9f}\n".format(K_T, -K_Q, Thr, Tor, C_th))
    return Eff, K_T, K_Q


Eff, K_T, K_Q = Efficiency()
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This section is dedicated to the initialization of the variable
Gamma_TE_P.
"""


import numpy as np
import sources.Variables as Var


def Gamma_It():

    Gamma_TE_P = np.zeros((Var.Msp+1))

    for j in range(Var.Msp+1):
        Gamma_TE_P[j] = 0.0
    Gamma_TE_P[Var.Msp] = -1      # lambda (t-1) initial
    with open("output/Propeller_Gamma_TE_P.txt", "w") as file:
        for i in range(Var.Msp+1):
            file.write(f"{Gamma_TE_P[i]:13.9f}\n")
    return Gamma_TE_P


Gamma_TE_P = Gamma_It()
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine is responsible for creating the initial grid for
the reference blade of the propeller, including Control Points & Grid Points.
Given the propeller's symmetry, generating the grid for the reference blade
alone suffices. Additionally, the subroutine undertakes the numbering of panels
and horseshoe vortices. Notably, the grid is aligned with the onset flow during
this phase.
"""


import math
import numpy as np
import sources.Variables as Var
import pandas as pd


def Grid_Generation_Propeller():

```

```
    r_R_P, X_P, Skew_P, Chord_P, Thick_P = np.loadtxt("input/grid.txt", unpack=True)
    U_0_P, U_R_P, U_T_P = np.loadtxt("input/onset.txt", unpack=True)

    """ MID-CHORD LINE """

    Midchord_line_P = np.zeros((Var.N_Iter,3))  # Create Matrix 3xN_Iter
    # Initial point (x, y, z)
    Midchord_line_P[0,0] = 0.0       # s line (x) - It begins at the hub-center but the first point is at the top of the hub
    Midchord_line_P[0,1] = 0.0       # s line (y) - It begins at the hub-center but the first point is at the top of the hub
    Midchord_line_P[0,2] = r_R_P[0]  # s line (z) assuming that r_R_P contain the initial z values

    # Cartesian coordinates for the blade surface
    for i in range(1, Var.N_Iter):
        Midchord_line_P[i, 0] = X_P[i]                             # x
        Midchord_line_P[i, 1] = -r_R_P[i] * math.sin(Skew_P[i])   # y
        Midchord_line_P[i, 2] = r_R_P[i] * math.cos(Skew_P[i])    # z

    s_tip = 0.0
    S_Distr_P = [0.0] * Var.N_Iter
    S_Distr_P[0] = math.sqrt(Midchord_line_P[0,1]**2 + Midchord_line_P[0,2]**2) # First value of the midchord line
    s_Hub_P = S_Distr_P[0]

    for i in range( Var.N_Iter - 1):  # This loop is used to find the length of the s line
        b = abs(Midchord_line_P[i+1,1])-abs(Midchord_line_P[i,1])       # Y Distance
        c = abs(Midchord_line_P[i+1,2])-abs(Midchord_line_P[i,2])       #Z Distance
        Prov = math.sqrt(b**2+c**2)
        S_Distr_P[i+1] = S_Distr_P[i] + Prov  # This is used to find the distribution of s, which is always costant

    s_tip = S_Distr_P[Var.N_Iter-1]

    data = np.column_stack([S_Distr_P, r_R_P])
    np.savetxt("output/Propeller_S_Distr.txt", data, fmt=['%13.9f','%13.9f'], delimiter= ' ', header = '    S_Distr        Radius')

    """ t FUNCTION     """

    t_gp_P=np.array([0.0]*(Var.Nch+1))    #Initialise the variable

    for i in range(Var.Nch + 1):
        t_gp_P[0] = -0.5
        t_gp_P[i] = -0.5 * np.cos(float(i+1-1.5)*3.14159274/float(Var.Nch))
        # (t) Grid points (always the same - it depends on Nch) - Cosine

    t_cp_P = np.array([0.0]*(Var.Nch),dtype=np.float64)  #Initialise the variable
    for i in range (Var.Nch):
        t_cp_P[i]= 0.5*(t_gp_P[i+1]+t_gp_P[i])  # (t) Control points (always the same - it depends on Nch) - Cosine


    data_t = np.column_stack([t_gp_P])
    np.savetxt("output/Propeller_t_gp.txt", data_t, fmt=['%13.9f'], delimiter= ' ', header = 't_gp')
    data_t = np.column_stack([t_cp_P])
    np.savetxt("output/Propeller_t_cp.txt", data_t, fmt=['%13.9f'], delimiter= ' ', header = 't_cp')

    """  s FUNCTION     """

    s_gp_P = np.array([0.0]*(Var.Msp+1),dtype=np.float64)       # (s) Grid points (always the same - it depends on Msp)
    for i in range(Var.Msp+1):
        aa = (i+1)*4.0 - 3.0 #Start with point after 0
        bb = 4.0*float(Var.Msp) + 2.0
        s_gp_P[i] = ((aa/bb)*(s_tip - s_Hub_P))+s_Hub_P

    s_cp_P = np.array([0.0]*(Var.Msp),dtype=np.float64)         # (s) Control points (always the same - it depends on Msp)
    for i in range(0,Var.Msp):
        s_cp_P[i] = 0.5 * (s_gp_P[i] + s_gp_P[i+1])

    data_s = np.column_stack([s_gp_P])
    np.savetxt("output/Propeller_s_gp.txt", data_s, fmt=['%13.9f'], delimiter= ' ', header = 's_gp')

    data_s = np.column_stack([s_cp_P])
    np.savetxt("output/Propeller_s_cp.txt", data_s, fmt=['%13.9f'], delimiter= ' ', header = 's_cp')

    """  GRID POINTS MATRIX - CALCULATION OF BETA(S),CHORD(S),SKEW(S) AND RAKE(S)  """

    #Initialise the variables
    Radius_gp_P = np.zeros(Var.Msp + 1)
    Chord_P_gp = np.zeros(Var.Msp + 1)
    Rake_P_gp = np.zeros(Var.Msp + 1)
    Skew_P_gp = np.zeros(Var.Msp + 1)
    sin_b = np.zeros(Var.Msp + 1)
    cos_b = np.zeros(Var.Msp + 1)
    Grid_Points_P = np.zeros(((Var.Msp + 1) * (Var.Nch + 1), 3))
    Theta_gp_P = np.zeros(Var.Nch + 1)

    # S Loop
    for i in range( Var.Msp +1):
        ipl = ((i+1) * (Var.Nch + 1)) - (Var.Nch + 1)

        Radius_gp_P[i] = np.interp(s_gp_P[i],S_Distr_P, r_R_P)  # Value of the radius in the grid points (s)
        U_0_P_gp = np.interp(s_gp_P[i],S_Distr_P, U_0_P)        # Wake (Axial) in the grid points (s)
        U_T_P_gp = np.interp(s_gp_P[i],S_Distr_P, U_T_P)        # Wake (Tangential) in the grid points (s)
        Chord_P_gp[i] = np.interp(s_gp_P[i],S_Distr_P, Chord_P) # Value of the chord in the grid points (s)
        Rake_P_gp[i] = np.interp(s_gp_P[i],S_Distr_P, X_P)      # Value of the rake in the grid points (s)
        Skew_P_gp[i] = np.interp(s_gp_P[i],S_Distr_P, Skew_P)   # Value of the skew in the grid points (s)

        V_tang = Var.Omega * Radius_gp_P[i] - U_T_P_gp  # Tangential velocity
        V_rel = np.sqrt(V_tang**2 + U_0_P_gp**2)        # Tangential velocity
        sin_b[i] = U_0_P_gp / V_rel                     # Sine (beta)
        cos_b[i] = V_tang / V_rel                       # Cosine (beta)

        # t Loop
        for j in range(Var.Nch+1):
            npl = (j) + ipl # Second counter used to order the Grid Points Matrix
            Theta_gp_P[j] = -Skew_P_gp[i] + (t_gp_P[j] * Chord_P_gp[i] * cos_b[i]) / Radius_gp_P[i]
```

```
            Grid_Points_P[npl, 0] = Rake_P_gp[i] + Chord_P_gp[i] * sin_b[i] * t_gp_P[j]   # X(s,t)
            Grid_Points_P[npl, 1] = -Radius_gp_P[i] * math.sin(Theta_gp_P[j])             # Y(s,t)
            Grid_Points_P[npl, 2] = Radius_gp_P[i] * math.cos(Theta_gp_P[j])              # Z(s,t)

    with open('output/Propeller_Grid_Points_Old.txt', 'w') as file:
        for i in range((Var.Nch + 1) * (Var.Msp + 1)):
            file.write(f"{Grid_Points_P[i, 0]:.9f}      {Grid_Points_P[i, 1]:.9f}      {Grid_Points_P[i, 2]:.9f}\n")
    with open('output/Propeller_Grid_Points.txt', 'w') as file:
        for i in range((Var.Nch + 1) * (Var.Msp + 1)):
            file.write(f"{Grid_Points_P[i, 0]:.9f}      {Grid_Points_P[i, 1]:.9f}      {Grid_Points_P[i, 2]:.9f}\n")
    with open('output/Propeller_Grid_Points_geom.txt', 'w') as file:
        for i in range((Var.Msp + 1)):
            file.write(f"{Radius_gp_P[i]:.9f}      {Chord_P_gp[i]:.9f}      {Rake_P_gp[i]:.9f}    {Skew_P_gp[i]:.9f}\n")

    """  CONTROL POINTS MATRIX - CALCULATION OF BETA(S(R)),CHORD(S),SKEW(S) AND RAKE(S)  """

    #Initialise the variable
    Radius_cp_P = np.zeros(Var.Msp + 1)
    Chord_P_cp = np.zeros(Var.Msp + 1)
    Rake_P_cp = np.zeros(Var.Msp + 1)
    Skew_P_cp = np.zeros(Var.Msp + 1)
    Theta_cp_P = np.zeros(Var.Nch + 1)
    Control_Points_P = np.zeros(((Var.Msp) * (Var.Nch), 3))

    for i in range(Var.Msp):
        Radius_cp_P[i] = 0.5*(Radius_gp_P[i]+Radius_gp_P[i+1]) # Value of the radius in the control point (s)
        U_0_P_cp = np.interp(s_cp_P[i],S_Distr_P, U_0_P)       # Wake (Axial) in the control points (s)
        U_T_P_cp = np.interp(s_cp_P[i],S_Distr_P, U_T_P)       #Wake (Tangential) in the control points (s)
        ipl = [0.0]
        ipl = ((i+1)*(Var.Nch))-(Var.Nch)              # Counter used to order the Control Points Matrix

        Chord_P_cp [i] = np.interp(s_cp_P[i],S_Distr_P, Chord_P)
        Rake_P_cp [i] = np.interp(s_cp_P[i],S_Distr_P, X_P)
        Skew_P_cp [i] = np.interp(s_cp_P[i],S_Distr_P, Skew_P)
        # Value of the chord in the control point (s)
        # Value of the rake in the control point (s)
        # Value of the skew in the control point (s)


        V_tang = Var.Omega * Radius_cp_P[i] - U_T_P_cp        # Tangential velocity
        V_rel = math.sqrt(V_tang**2 + U_0_P_cp**2)            # Relative velocity
        sin_b[i] = U_0_P_cp/V_rel                             # Sine (beta)
        cos_b[i] = V_tang/V_rel                               # Cosine (beta)

        #t loop
        for j in range(Var.Nch):
            npl = j+(ipl) # Second counter used to order the Control Points Matrix
            Theta_cp_P[j] = -Skew_P_cp[i] + (t_cp_P[j] * Chord_P_cp[i] * cos_b[i]) / Radius_cp_P[i]
            Control_Points_P[npl, 0] = Rake_P_cp[i] + Chord_P_cp[i] * sin_b[i] * t_cp_P[j]    # X(s,t)
            Control_Points_P[npl, 1] = -Radius_cp_P[i] * math.sin(Theta_cp_P[j])              # Y(s,t)
            Control_Points_P[npl, 2] = Radius_cp_P[i] * math.cos(Theta_cp_P[j])               # Z(s,t)

    with open('output/Propeller_Control_Points_Old.txt', 'w') as file:
        for i in range((Var.Nch) * (Var.Msp)):
            file.write(f"{Control_Points_P[i, 0]:.9f}      {Control_Points_P[i, 1]:.9f}      {Control_Points_P[i, 2]:.9f}\n")

    with open('output/Propeller_Control_Points.txt', 'w') as file:
        for i in range((Var.Nch) * (Var.Msp)):
            file.write(f"{Control_Points_P[i, 0]:.9f}      {Control_Points_P[i, 1]:.9f}      {Control_Points_P[i, 2]:.9f}\n")

    with open('output/Propeller_Control_Points_geom.txt', 'w') as file:
        for i in range((Var.Msp+1)):
            file.write(f"{Chord_P_cp [i]:.9f}      {Rake_P_cp [i]:.9f}      {Skew_P_cp [i]:.9f}\n")

    """  NUMERATION OF THE PANEL AND THE SIDE  """

    N_Panel_P = np.array([[0,1, Var.Nch + 2, Var.Nch +1]])     # Initialize the first panel

    t = 0
    for j in range(Var.Msp):
        for i in range(1,Var.Nch):

            t += 1
            t2 = t - 1

            N_Panel = N_Panel_P[t2] + 1
            N_Panel_P = np.append(N_Panel_P, [N_Panel], axis=0)

        if j != Var.Msp-1:
            t += 1
            t1 = t - 1
            N_Panel = N_Panel_P[t1] + 2
            N_Panel_P = np.append(N_Panel_P, [N_Panel], axis=0)

    with open ("output/Propeller_Numeration_Panel.txt","w") as file:
        for i in range((Var.Nch*Var.Msp)):
            file.write(f"{N_Panel_P[i, 0]:2d}   {N_Panel_P[i, 1]:2d}   {N_Panel_P[i, 2]:2d}   {N_Panel_P[i, 3]:2d}\n")

    """  COORDINATES FOR THE BOUND VORTICES (T.E. SIDE)  """

    # It is a matrix with the grid points at the T.E.
    N_Bound_Vortex_P = np.zeros((Var.Msp + 1, 1), dtype=int)
    for i in range (Var.Msp+1):
        N_Bound_Vortex_P[i] = i+i*(Var.Nch)

    np.savetxt("output/Propeller_N_Bound_Vortex.txt", N_Bound_Vortex_P, fmt="%3d")

    """  HORSESHOE VORTEX MATRIX    """

    Horseshoe_P = np.zeros(((Var.Msp),4),dtype=int)
    for i in range(Var.Msp):
```

```
        Horseshoe_P[i,0] = i                        # This value is used to access to the N_Bound_Vortex_P
                                                    # matrix in order to select the Bound vortex (0-6)(6-12)..
        Horseshoe_P[i,1] = i+1                      # This value is used to access to the N_Bound_Vortex_P
                                                    # matrix in order to select the Bound vortex (0-6)(6-12)..
        Horseshoe_P[i,2] =((i-1)+1)*Var.Nch         # Number of the T.E panel (0-5-10..)
        Horseshoe_P[i,3] = i                        # Number of the horseshoe vortex (0-1-2-3..)

    with open("output/Propeller_Horseshoe.txt","w") as file:
        for i in range(Var.Msp):
            file.write(f"   {Horseshoe_P[i,0]:2d}   {Horseshoe_P[i,1]:2d}   {Horseshoe_P[i,2]:2d}    {Horseshoe_P[i,3]}\n")

    """  COORDINATES FOR THE TRANSITION WAKE (STRAIGHT LINE VORTICES)  """

    # Initialize the variable
    Points_Trans_Wake_P = np.zeros((((Var.N_P_L+1)*(Var.Msp+1)),3))

    for i in range(Var.Msp+1):
        i_1 = i+i*(Var.N_P_L)
        x_trans_wake = Grid_Points_P[N_Bound_Vortex_P[i,0],0]
        y_trans_wake = Grid_Points_P[N_Bound_Vortex_P[i,0],1]
        z_trans_wake = Grid_Points_P[N_Bound_Vortex_P[i,0],2]
        # X value for the first point of the transition wake - T.E.
        # Y value for the first point of the transition wake - T.E.
        # Z value for the first point of the transition wake - T.E.

        r_trans_wake = np.sqrt(y_trans_wake**2 + z_trans_wake**2) # Radius at the T.E.
        U_O_P_trans_wake = np.interp(r_trans_wake, r_R_P, U_O_P)  # Wake (Axial) in the transition wake (s)
        U_T_P_trans_wake = np.interp(r_trans_wake, r_R_P, U_T_P)  # Wake (Tangential) in the transition wake (s)


        V_tang = Var.Omega*r_trans_wake - U_T_P_trans_wake      # Tangential velocity
        pitch_trans_wake = (2*np.pi*r_trans_wake*U_O_P_trans_wake)/V_tang  # Pitch at the T.E. (It has only a radial variation)
        Points_Trans_Wake_P[i_1,0] = x_trans_wake         # Grid points for the transition wake (x) - T.E
        Points_Trans_Wake_P[i_1,1] = r_trans_wake         # Grid points for the transition wake (radius) - T.E
        Points_Trans_Wake_P[i_1,2] = pitch_trans_wake     # Grid points for the transition wake (pitch) - T.E -
        # It is costant everywhere (right now) because V_tang does not take into
        # account of the induced velocity (due the fact that we don't know it yet)

        delta_trans_wake = (-4 * Var.Rad_P - x_trans_wake)/(Var.N_P_L)
        # The transition wake goes four radii downstream
        # Loop used to divide the transition wake in N_P_L parts (N_P_L+1 points)
        for j in range(Var.N_P_L):
            i_2 = (i_1) + j+1

            # Grid points for the transition wake
            Points_Trans_Wake_P[i_2,0] = x_trans_wake + (j+1) * delta_trans_wake # (x)
            Points_Trans_Wake_P[i_2,1] = r_trans_wake                            # (radius)
            Points_Trans_Wake_P[i_2,2] = pitch_trans_wake                        # (pitch)

    with open("output/Propeller_Points_Trans_Wake_Old.txt", "w") as file:
        file.write(f"{'Point':<8}{'x':<12}{'r':<20}{'p':<20}\n")
        for i in range(Var.Msp+1):
            i_1 = i+i*(Var.N_P_L)
            file.write(f"{i:<5}{Points_Trans_Wake_P[i_1,0]:13.9f}{Points_Trans_Wake_P[i_1,1]:13.9f}"
                       f"{Points_Trans_Wake_P[i_1,2]:13.9f}\n")

            for j in range(Var.N_P_L):
                i_2 = (i_1) + j+1
                file.write(f"{i:<5}{Points_Trans_Wake_P[i_2,0]:13.9f}{Points_Trans_Wake_P[i_2,1]:13.9f}"
                           f"{Points_Trans_Wake_P[i_2,2]:13.9f}\n")

    with open("output/Propeller_Points_Trans_Wake.txt", "w") as file:
        file.write(f"{'Point':<8}{'x':<12}{'r':<20}{'p':<20}\n")
        for i in range(Var.Msp+1):
            i_1 = i+i*(Var.N_P_L)
            file.write(f"{i:<5}{Points_Trans_Wake_P[i_1,0]:13.9f}{Points_Trans_Wake_P[i_1 ,1]:13.9f}"
                       f"{Points_Trans_Wake_P[i_1,2]:13.9f}\n")

            for j in range(Var.N_P_L):
                i_2 = (i_1) + j+1
                file.write(f"{i:<5}{Points_Trans_Wake_P[i_2,0]:13.9f}{Points_Trans_Wake_P[i_2,1]:13.9f}"
                           f"{Points_Trans_Wake_P[i_2,2]:13.9f}\n")

    return(S_Distr_P, r_R_P, t_gp_P, s_gp_P, Grid_Points_P, Control_Points_P,
           N_Panel_P, N_Bound_Vortex_P, Horseshoe_P, Points_Trans_Wake_P)


(S_Distr_P, r_R_P, t_gp_P, s_gp_P, Grid_Points_P, Control_Points_P, N_Panel_P, N_Bound_Vortex_P, Horseshoe_P, Points_Trans_Wake_P
      )=Grid_Generation_Propeller()
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine is tasked with creating the helix geometry.
"""


def Helix(x1, x2, y1, y2, dx):
    delx = x2 - x1
    a = (y2-y1-delx*dx)/delx/delx
    b = dx-2*a*x1
    c = y1 + a*x1*x1 - dx*x1
    return (a, b, c)
```

```
"""
```

```python
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine computes the induced velocities in the midpoints
of the segments (coefficient) from the entire grid of the propeller.
"""


import numpy as np
import sources.Variables as Var
from sources.Weight_Function_Propeller_P import Weight_function_propeller
from sources.Mid_Vect_Propeller_P import Mid_Vect_Propeller
from sources.Panel_Induced_Velocity_Propeller_P import Panel_Induced_Velocity_Propeller


def Induced_Grid_Propeller():
    Weight_P = Weight_function_propeller()
    I_P_Points_P = (Var.Msp*Var.Nch)

    V_Grid_P = np.zeros((Var.Msp,I_P_Points_P, 4,3))  # Induced velocity from the entire grid
    n_plaux = np.array(([0.0]*(Var.Msp)), dtype = int)
    npl = np.array(([0.0]*(Var.Msp)),dtype = int)

    for i in range (I_P_Points_P): # This loop selects the panel where the point px,py,pz is located

        for k in range (4): # This loop selects, inside the panel, the side where the point px,py,pz is located
            px,py,pz,v_px,v_py,v_pz = Mid_Vect_Propeller(i, k) # This subroutine is used to calculate the midpoint px,py,pz

            for j in range(Var.Msp):
                U_x = 0                 # Initialization of the variable U_x
                U_y = 0                 # Initialization of the variable U_y
                U_z = 0                 # Initialization of the variable U_z
                n_plaux = (Var.Nch)*(j)

                for h in range(Var.Nch): # This loop selects the panel (Chordwise) that induces velocity

                    npl = h + n_plaux
                    qx_pnl, qy_pnl, qz_pnl = Panel_Induced_Velocity_Propeller(npl,k,i,px,py,pz)

                    U_x += Weight_P[j,h] * qx_pnl   # Temporary induced velocity in the point px,py,pz
                    U_y += Weight_P[j,h] * qy_pnl   # due to the chordwise ring j (x),(y),(z)
                    U_z += Weight_P[j,h] * qz_pnl

                    V_Grid_P [j,i,k,0] = U_x         # Induced velocity in the point px,py,pz
                    V_Grid_P [j,i,k,1] = U_y         # due to the chordwise ring j (x),(y),(z)
                    V_Grid_P [j,i,k,2] = U_z

    with open("output/Propeller_Velocity_Grid.txt", "w") as file:
        file.write("{:>5s} {:>8s} {:>2s} {:>15s} {:>15s} {:>2s}\n".format("Point","Spanwise","Ux","Uy","Uz", ""))
        file.write("{:>8s} {:>7s}\n".format("(Panel)", "(Side)"))

        for i in range(I_P_Points_P):
            for k in range(4):
                for j in range(Var.Msp):
                    data_format = "{:>2d} {:>4d} {:>4d} {:>13.9f} {:>13.9f} {:>13.9f}\n"
                    file.write(data_format.format(i, k, j, V_Grid_P[j, i, k, 0], V_Grid_P[j, i, k, 1], V_Grid_P[j, i , k, 2]))
    return V_Grid_P


V_Grid_P = Induced_Grid_Propeller()
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the midpoint coordinates
(mid_x,mid,mid_z) and the vector for the panel side
(vector_x,vector_y,vector_z) of the reference blade of the propeller.

Parameters
 - n_pnl: number of the panel
 - n_side: number of the side
"""


import numpy as np
import sources.Variables as Var


def Mid_Vect_Propeller(n_pnl,n_side):

    Grid_Points_P = np.loadtxt("output/Propeller_Grid_Points.txt")
    N_Panel_P = np.loadtxt("output/Propeller_Numeration_Panel.txt",dtype='int')
    j1 = n_side
    j2 = (n_side+1)

    # Special case for n_side = 3
    if n_side == 3:
        j2 = 0

    mid_x = 0.5* (Grid_Points_P[N_Panel_P[n_pnl,j2],0] + Grid_Points_P[N_Panel_P[n_pnl,j1],0])      # Midpoint (x)
    mid_y = 0.5* (Grid_Points_P[N_Panel_P[n_pnl,j2],1] + Grid_Points_P[N_Panel_P[n_pnl,j1],1])      # Midpoint (y)
    mid_z = 0.5* (Grid_Points_P[N_Panel_P[n_pnl,j2],2] + Grid_Points_P[N_Panel_P[n_pnl,j1],2])      # Midpoint (z)

    vector_x = Grid_Points_P[N_Panel_P[n_pnl,j2],0] - Grid_Points_P[N_Panel_P[n_pnl,j1],0]      #Vector (x)
    vector_y = Grid_Points_P[N_Panel_P[n_pnl,j2],1] - Grid_Points_P[N_Panel_P[n_pnl,j1],1]      #Vector (y)
    vector_z = Grid_Points_P[N_Panel_P[n_pnl,j2],2] - Grid_Points_P[N_Panel_P[n_pnl,j1],2]      #Vector (z)
```

```python

    return mid_x,mid_y,mid_z,vector_x,vector_y,vector_z
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the normal vector for a panel
"""


import numpy as np


def Normal_Vector(x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3, x_4, y_4, z_4):
    a_1 = x_2 - x_3
    # X value of the first vector of the panel (Point 2 and Point 3)
    a_2 = y_2 - y_3
    # Y value of the first vector of the panel (Point 2 and Point 3)
    a_3 = z_2 - z_3
    # Z value of the first vector of the panel (Point 2 and Point 3)

    b_1 = x_4 - x_1
    # X value of the first vector of the panel (Point 1 and Point 4)
    b_2 = y_4 - y_1
    # Y value of the first vector of the panel (Point 1 and Point 4)
    b_3 = z_4 - z_1
    # Z value of the first vector of the panel (Point 1 and Point 4)

    x = a_2 * b_3 - a_3 * b_2  # X component of the cross product
    y = b_1 * a_3 - a_1 * b_3  # Y component of the cross product
    z = a_1 * b_2 - a_2 * b_1  # Z component of the cross product

    Norm = np.sqrt(x**2 + y**2 + z**2)  # Norm of the vector

    vector_x = x / Norm  # X component of the normal vector
    vector_y = y / Norm  # Y component of the normal vector
    vector_z = z / Norm  # Z component of the normal vector

    return (vector_x, vector_y, vector_z)
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine computes the onset flow at the midpoints of the
sides of the propeller. The onset flow is assumed to be axi-symmetric and
independent of the longitudinal position, meaning it has only a radial
variation.
"""


import numpy as np
import sources.Variables as Var
from sources.Mid_Vect_Propeller_P import Mid_Vect_Propeller


def Onset_Flow_Propeller():
    I_P_Points_P = (Var.Msp*Var.Nch)
    r_R_P, X_P, Skew_P, Chord_P, Thick_P = np.loadtxt("input/grid.txt", unpack=True)
    U_0_P, U_R_P, U_T_P = np.loadtxt("input/onset.txt", unpack=True)
    U_0_P_Onset = np.zeros((Var.Msp * Var.Nch, 4, 3))   #Onset Flow (x) - Propeller
    U_T_P_Onset = np.zeros((Var.Msp * Var.Nch, 4, 3))   #Onset Flow (y) - Propeller
    U_R_P_Onset = np.zeros((Var.Msp * Var.Nch, 4, 3))   #Onset Flow (z) - Propeller
    V_Onset_P = np.zeros((Var.Msp * Var.Nch, 4, 3))

    for j in range(I_P_Points_P): # This loop selects the panel where the point px,py,pz is located
        for k in range (4):       # This loop selects, inside the panel, the sides where the point px,py,pz is located
            xx, xy, xz, xl, yl, zl = Mid_Vect_Propeller(j, k)

            #This subroutine is used to calculate the midpoint px,py,pz
            r_sid = np.sqrt(xy*xy + xz*xz)           # Radius of the points px,py,pz

            U_0_P_Onset= np.interp(r_sid,r_R_P,U_0_P)    # Wake (Axial) in the midpoints (s)
            U_T_P_Onset= np.interp(r_sid,r_R_P,U_T_P)    # Wake (Tangential) in the midpoints (s)
            U_R_P_Onset= np.interp(r_sid,r_R_P,U_R_P)    # Wake (Radial) in the grid midpoints (s)

            V_Onset_P[j,k,0] = - U_0_P_Onset          # Onset Flow (x)
            V_Onset_P[j,k,1] = U_R_P_Onset*xy/r_sid - U_T_P_Onset*xz/r_sid + Var.Omega*xz  # Onset Flow (y)
            V_Onset_P[j,k,2] = U_R_P_Onset*xz/r_sid + U_T_P_Onset*xy/r_sid - Var.Omega*xy  # Onset Flow (z)

    with open("output/Propeller_Onset_Flow.txt", "w") as file:
        file.write("{:5s}{:>6s}{:12s}{:15s}\n".format("Point", "Ux", "Uy", "Uz"))
        file.write("{:<8s}{:<7s}\n".format("(Panel)", "(Side)"))

        for j in range(I_P_Points_P):
            for k in range(4):
                file.write("{:2d}{:4d}{:5s}{:13.9f}{:5s}{:13.9f}{:5s}{:13.9f}\n".format(
                    j, k, "", V_Onset_P[j, k, 0], "", V_Onset_P[j, k, 1], "", V_Onset_P[j, k, 2]))

    return V_Onset_P


V_Onset_P = Onset_Flow_Propeller()
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the induced velocities (coefficient)
from a panel in the point (px,py,pz) for all the blades of the propeller
without including the bound vortex.

Parameters:
 - n_pnl   : number of the panel that induces velocity in the point
             (px,py,pz)
 - mpnl    : number of the panel that containes the point px,py,pz.
 - msid    : number of the side that containes the point px,py,pz.
"""


import numpy as np
import sources.Variables as Var
from sources.Biot_Savart_Propeller_P import Biot_Savart_Propeller


def Panel_Induced_Velocity_Propeller_Align(n_pnl, msid, mpnl, px, py, pz):

    Grid_Points_P = np.loadtxt("output/Propeller_Grid_Points.txt")
    N_Panel_P = np.loadtxt("output/Propeller_Numeration_Panel.txt", dtype='int')

    # DECLARATION OF VARIABLES

    U_x = 0     # Initialization of the variable U_x
    U_y = 0     # Initialization of the variable U_y
    U_z = 0     # Initialization of the variable U_z

    delta_theta = 2*np.pi/float(Var.Z_Blade_P)

    x_10 = Grid_Points_P[N_Panel_P[n_pnl,0],0] # X value for the first point of the chosen panel of the propeller
    y_10 = Grid_Points_P[N_Panel_P[n_pnl,0],1] # Y value for the first point of the chosen panel of the propeller
    z_10 = Grid_Points_P[N_Panel_P[n_pnl,0],2] # Z value for the first point of the chosen panel of the propeller

    #Loop for the number of blades
    for j in range (Var.Z_Blade_P):
        theta_blade = float(j*delta_theta)
        cos_theta = np.cos(theta_blade)
        sin_theta = np.sin(theta_blade)

        x_1 = x_10    # X value for the first point of the chosen panel of the chosen blade of the propeller
        y_1 = y_10*cos_theta - z_10*sin_theta
        # Y value for the first point of the chosen panel of the chosen blade of the propeller
        z_1 = z_10*cos_theta + y_10*sin_theta
        # Z value for the first point of the chosen panel of the chosen blade of the propeller

        # 4 sides of the panel
        for i in range(4):
            i_2 = i + 1 if i < 3 else 0

            x_2  = Grid_Points_P[N_Panel_P[n_pnl,i_2],0] # X value for the second point of the chosen panel
                                                         # of the chosen blade of the propeller
            y_20 = Grid_Points_P[N_Panel_P[n_pnl,i_2],1] # Y value for the second point of the chosen panel
                                                         # of the chosen blade of the propeller
            z_20 = Grid_Points_P[N_Panel_P[n_pnl,i_2],2] # Z value for the second point of the chosen panel
                                                         # of the chosen blade of the propeller

            y_2 = y_20 * cos_theta - z_20 * sin_theta
            # Y value for the second point of the chosen panel of the chosen blade of the propeller
            z_2 = z_20 * cos_theta + y_20 * sin_theta
            # Z value for the second point of the chosen panel of the chosen blade of the propeller

            if i == 1:
                x_1, y_1, z_1 = x_2, y_2, z_2
                continue
            if i == 3:
                x_1, y_1, z_1 = x_2, y_2, z_2
                continue
            else:
                U_x_0, U_y_0, U_z_0 = Biot_Savart_Propeller(1, x_1, y_1, z_1, x_2, y_2, z_2, px, py, pz)

                # Update induced velocities
                U_x = U_x + U_x_0
                U_y = U_y + U_y_0
                U_z = U_z + U_z_0

                x_1, y_1, z_1 = x_2, y_2, z_2

    return(U_x, U_y, U_z)
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the induced velocities (coefficient)
from a panel at the point (px,py,pz) for all blades of the propeller.

Parameters:
- n_pnl: Number of the panel inducing velocity at the point (px,py,pz)
- mpnl: Number of the panel containing the point (px,py,pz)
- msid: Number of the side containing the point (px,py,pz)
"""

```

```python

from sources.Biot_Savart_Propeller_P import Biot_Savart_Propeller
import numpy as np
import sources.Variables as Var


def Panel_Induced_Velocity_Propeller(n_pnl, msid, mpnl, px, py, pz):
    Grid_Points_P = np.loadtxt("output/Propeller_Grid_Points.txt")
    N_Panel_P = np.loadtxt("output/Propeller_Numeration_Panel.txt", dtype='int')

    U_x, U_y, U_z = 0.0, 0.0, 0.0           # Initialization of the variable U_x, U_y, U_z

    delta_theta = 2*np.pi/float(Var.Z_Blade_P)

    x_10 = Grid_Points_P[N_Panel_P[n_pnl,0],0]  # X value for the first point of the chosen panel of the propeller
    y_10 = Grid_Points_P[N_Panel_P[n_pnl,0],1]  # Y value for the first point of the chosen panel of the propeller
    z_10 = Grid_Points_P[N_Panel_P[n_pnl,0],2]  # Z value for the first point of the chosen panel of the propeller

    for j in range(Var.Z_Blade_P):    #Loop for the number of blades
        theta_blade = (j) *delta_theta

        cos_theta = np.cos(theta_blade)
        sin_theta = np.sin(theta_blade)

        x_1 = x_10
        y_1 = y_10*cos_theta - z_10*sin_theta
        z_1 = z_10*cos_theta + y_10*sin_theta
        # X value for the first point of the chosen panel of the chosen blade of the propeller
        # Y value for the first point of the chosen panel of the chosen blade of the propeller
        # Z value for the first point of the chosen panel of the chosen blade of the propeller

        # 4 sides of the panel
        for i in range(4):

            i_2 = i + 1 if i < 3 else 0
            x_2  = Grid_Points_P[N_Panel_P[n_pnl,i_2],0]
            # X value for the second point of the chosen panel of the chosen blade of the propeller
            y_20 = Grid_Points_P[N_Panel_P[n_pnl,i_2],1]
            # Y value for the second point of the chosen panel of the chosen blade of the propeller
            z_20 = Grid_Points_P[N_Panel_P[n_pnl,i_2],2]
            # Z value for the second point of the chosen panel of the chosen blade of the propeller

            y_2 = y_20 * cos_theta - z_20 * sin_theta
            # Y value for the second point of the chosen panel of the chosen blade of the propeller
            z_2 = z_20 * cos_theta + y_20 * sin_theta
            # Z value for the second point of the chosen panel of the chosen blade of the propeller

            if n_pnl == mpnl and (j == 0) and i == msid:
                x_1 = x_2               # The second point becomes the first point (x)
                y_1 = y_2               # The second point becomes the first point (y)
                z_1 = z_2               # The second point becomes the first point (z)
            else:
                U_x_0, U_y_0, U_z_0 = Biot_Savart_Propeller(1, x_1, y_1, z_1, x_2, y_2, z_2, px, py, pz)
                # Induced velocity of that side of the panel of the propeller
                # (I_z = 1 because I have al ready created a loop)
                # Update induced velocity

                U_x = U_x + U_x_0
                U_y = U_y + U_y_0
                U_z = U_z + U_z_0

                x_1 = x_2               # The second point becomes the first point (x)
                y_1 = y_2               # The second point becomes the first point (y)
                z_1 = z_2               # The second point becomes the first point (z)

    return(U_x, U_y, U_z)
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description:  This subroutine is designed to open and process three specific
files containing data related to propeller characteristics, chord, and
velocities. It aims to facilitate the handling and  analysis of aerodynamic
properties through these datasets.
"""


import numpy as np
import pandas as pd
import sources.Variables as Var
from scipy.interpolate import CubicSpline


def propeller_geometry():
    # Read from exel as Data Frame
    data = pd.read_excel("input/geometry.xlsx", "geometry1", header=0)

    # Radius
    r_prop = data['Radius'].values
    # Cartesian coordinate
    x_prop = data['x'].values
    # Skew in Radians
    skew_prop = data['Skew (Rad)'].values
    # Chord
    chord_prop = data['Chord'].values
    # Thickness
    thick_prop = data['t'].values
```

```python
    # Axial onset flow
    u0_prop = data['U0'].values
    # Radial onset flow
    ur_prop = data['Ur'].values
    # Tangential onset flow
    ut_prop = data['Ut'].values

    # Linear interpolation for the Radius
    ir_prop = np.linspace(r_prop[0], max(r_prop), num=Var.N_Iter)

    # Cubic spline interpolation for various properties along radial axis:
    # x coordinate, skew, chord, thickness, axial onset flow, radial onset
    # flow, tangential onset flow.
    interpolator_x = CubicSpline(r_prop, x_prop, axis=0)
    ix_prop = interpolator_x(ir_prop)

    interpolator_skew = CubicSpline(r_prop, skew_prop, axis=0)
    iskew_prop = interpolator_skew(ir_prop)

    interpolator_chord = CubicSpline(r_prop, chord_prop, axis=0)
    ichord_prop = interpolator_chord(ir_prop)

    interpolator_thick = CubicSpline(r_prop, thick_prop, axis=0)
    ithick_prop = interpolator_thick(ir_prop)

    interpolator_u0 = CubicSpline(r_prop, u0_prop, axis=0)
    iu0_prop = interpolator_u0(ir_prop)

    interpolator_ur = CubicSpline(r_prop, ur_prop, axis=0)
    iur_prop = interpolator_ur(ir_prop)

    interpolator_ut = CubicSpline(r_prop, ut_prop, axis=0)
    iut_prop = interpolator_ut(ir_prop)

    # Define datasets
    dataset = list(zip(ir_prop, ix_prop, iskew_prop, ichord_prop, ithick_prop))
    dataset1 = list(zip(iu0_prop, iur_prop, iut_prop))

    # Write dataset to 'grid.txt'
    with open('input/grid.txt', 'w') as fileID:
        format = '{:10.5f} {:10.5f} {:10.5f} {:10.5f} {:10.5f}\n'
        for row in dataset:
            fileID.write(format.format(*row))

    # Write dataset1 to 'onset.txt'
    with open('input/onset.txt', 'w') as file:
        format = '{:10.5f} {:10.5f} {:10.5f}\n'
        for row in dataset1:
            file.write(format.format(*row))

    # Write chord_prop to 'chord.txt'
    np.savetxt('input/chord.txt', chord_prop, fmt='%10.5f')

    return ir_prop, ix_prop, iskew_prop, ichord_prop, ithick_prop


ir_prop, ix_prop, iskew_prop, ichord_prop, ithick_prop = propeller_geometry()
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This function saves the old propeller pitch to evaluate the
residual for the pitch distribution.
"""


import numpy as np
import sources.Variables as Var


pitch_0 = np.zeros((Var.Msp+1, 1))


def pitch():
    pitch_0 = np.zeros((Var.Msp+1,1))
    Points_Trans_Wake_P = np.loadtxt("output/Propeller_Points_Trans_Wake.txt", skiprows= 1, usecols= (1,2,3))
    for i in range (Var.Msp+1):
        i_1 = i+i*(Var.N_P_L)
        pitch_0[i] = Points_Trans_Wake_P[i_1,2]
    return pitch_0
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the Si-function used in 'De_Jong' by
rational approximations.
"""


import numpy as np


def si(xbar):
```

```
    f = (xbar**8+38.027264*xbar**6+265.187033*xbar**4+335.677320*xbar**2+38.102495) / (
    xbar**8+40.021433*xbar**6+322.624911*xbar**4+570.236280*xbar**2+157.105423)/xbar

    g = (xbar**8+42.242855*xbar**6+302.757865*xbar**4+352.018498*xbar**2+21.821899) / (
    xbar**8+48.196927*xbar**6+482.485984*xbar**4 +1114.978885*xbar**2+449.690326)/xbar**2

    sires=-f*np.cos(xbar)-g*np.sin(xbar)

    return(sires)
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine computes the skin friction drag at control points
of the propeller.
"""


import numpy as np
import sources.Variables as Var
from sources.Weight_Function_Propeller_P import Weight_function_propeller
from sources.Area_Panel_P import Area_Panel
from sources.Panel_Induced_Velocity_Propeller_P import Panel_Induced_Velocity_Propeller
from sources.Trailing_Vortices_Propeller_P import Trailing_Vortices_Propeller
from sources.Biot_Savart_Propeller_P import Biot_Savart_Propeller
from sources.Mid_Vect_Propeller_P import Mid_Vect_Propeller
from sources.Normal_Vector_P import Normal_Vector


def Skin_Friction_Drag():
    I_P_Points_P = (Var.Msp*Var.Nch)
    Weight_P = Weight_function_propeller()

    # DECLARATION OF VARIABLES

    V_Tot_P = np.loadtxt("output/Propeller_Velocity_Total.txt", skiprows=2, usecols= (2,3,4))
    V_Tot_P = np.reshape(V_Tot_P, (I_P_Points_P, 4, 3))
    N_Panel_P = np.loadtxt("output/Propeller_Numeration_Panel.txt",dtype='int')
    Grid_Points_P = np.loadtxt("output/Propeller_Grid_Points.txt")
    Control_Points_P = np.loadtxt('output/Propeller_Control_Points.txt')
    Radius, beta = np.loadtxt("output/Propeller_Beta.txt",skiprows = 1, unpack = True )
    Gamma_TE_P = np.loadtxt("output/Propeller_Gamma_TE_P.txt")
    Panel, Gamma_Panel_P = np.loadtxt("output/Propeller_Gamma_Blade.txt",skiprows = 1,unpack= True)
    r_R_P,X_P,Skew_P,Chord_P,Thick_P= np.loadtxt("input/grid.txt",unpack = True)
    U_0_P, U_R_P, U_T_P = np.loadtxt("input/onset.txt",unpack = True )
    S_Distr_P,r_R_P = np.loadtxt("output/Propeller_S_Distr.txt", skiprows= 1, unpack= True)
    Points_Trans_Wake_P = np.loadtxt("output/Propeller_Points_Trans_Wake.txt",
                         skiprows= 1, usecols= (1,2,3))

    radius_cp = np.zeros((I_P_Points_P))
    s_ring = np.zeros((Var.Msp))
    vector_panel = np.zeros((3))
    tangentialDirection = np.zeros((3))
    T_Skin_F = 0.0
    Q_Skin_F = 0.0
    vector_x = np.zeros((Var.Msp))
    vector_y = np.zeros((Var.Msp))
    vector_z = np.zeros((Var.Msp))
    r_cp_a = np.zeros((Var.Msp))
    cos_theta_c = np.zeros((Var.Msp))
    sin_theta_c = np.zeros((Var.Msp))
    u_x_tot_a = np.zeros((Var.Msp))
    u_y_tot_a = np.zeros((Var.Msp))
    u_z_tot_a = np.zeros((Var.Msp))
    u_tang_skin = np.zeros((Var.Msp))
    u_rel_skin = np.zeros((Var.Msp))
    Coeff_Corr_Camber = np.zeros((Var.Msp))
    Coeff_Corr_Alpha = np.zeros((Var.Msp))
    Coeff_Corr_Thick = np.zeros((Var.Msp))
    Thick_P_skin = np.zeros((Var.Msp))
    Chord_P_skin = np.zeros((Var.Msp))
    L_Ring = np.zeros((Var.Msp))
    L_Ring_x = np.zeros((Var.Msp))
    L_Ring_y = np.zeros((Var.Msp))
    L_Ring_z = np.zeros((Var.Msp))
    C_L_Local = np.zeros((Var.Msp))
    Camber_Dimless = np.zeros((Var.Msp))
    ideal_angle_attack = np.zeros((Var.Msp))
    angle_attack = np.zeros((Var.Msp))
    beta_temp_surface = np.zeros((Var.Msp))
    pitch_cp_final_surface = np.zeros((Var.Msp))

    T_Skin_f = 0.0  # Initialization of the variable used to calculate the viscous drag
    Q_Skin_f = 0.0  # Initialization of the variable used to calculate the viscous drag

    s_tot = 0
    for j in range (Var.Msp):
        s_r = 0

        for i in range(Var.Nch):
            npl = i + (j) * Var.Nch

            x_1 = Grid_Points_P[N_Panel_P[npl,0],0]      # X value of the edge number one of the panel j
            y_1 = Grid_Points_P[N_Panel_P[npl,0],1]      # Y value of the edge number one of the panel j
            z_1 = Grid_Points_P[N_Panel_P[npl,0],2]      # Z value of the edge number one of the panel j

            x_2 = Grid_Points_P[N_Panel_P[npl,1],0]      # X value of the edge number two of the panel j
```

```
            y_2 = Grid_Points_P[N_Panel_P[npl,1],1]      # Y value of the edge number two of the panel j
            z_2 = Grid_Points_P[N_Panel_P[npl,1],2]      # Z value of the edge number two of the panel j

            x_3 = Grid_Points_P[N_Panel_P[npl,3],0]      # X value of the edge number four of the panel j
            y_3 = Grid_Points_P[N_Panel_P[npl,3],1]      # Y value of the edge number four of the panel j
            z_3 = Grid_Points_P[N_Panel_P[npl,3],2]      # Z value of the edge number four of the panel j

            x_4 = Grid_Points_P[N_Panel_P[npl,2],0]      # X value of the edge number three of the panel j
            y_4 = Grid_Points_P[N_Panel_P[npl,2],1]      # Y value of the edge number three of the panel j
            z_4 = Grid_Points_P[N_Panel_P[npl,2],2]      # Z value of the edge number three of the panel j

            s_parz = Area_Panel (x_1,y_1,z_1,x_2,y_2,z_2,x_3,y_3,z_3,x_4,y_4,z_4)
            # Area of the panel where the control point is located

            s_r = s_r + s_parz

        s_ring [j] = s_r
        s_tot = s_tot + s_r

    Ae = s_tot
    Ao = np.pi * Var.Rad_P**2

    AeAo = Ae/Ao * Var.Z_Blade_P

    # SKIN FRICTION DRAG

    # Loop used to select all the control points of the propeller
    for j in range (I_P_Points_P):

        p_x_mdp = Control_Points_P[j,0]       # X coordinate of the chosen control point of the propeller
        p_y_mdp = Control_Points_P[j,1]       # Y coordinate of the chosen control point of the propeller
        p_z_mdp = Control_Points_P[j,2]        # Z coordinate of the chosen control point of the propeller

        x_1 = Grid_Points_P[N_Panel_P[j,0],0]     # X value of the edge number one of the panel j
        y_1 = Grid_Points_P[N_Panel_P[j,0],1]     # Y value of the edge number one of the panel j
        z_1 = Grid_Points_P[N_Panel_P[j,0],2]     # Z value of the edge number one of the panel j

        x_2 = Grid_Points_P[N_Panel_P[j,1],0]     # X value of the edge number two of the panel j
        y_2 = Grid_Points_P[N_Panel_P[j,1],1]     # Y value of the edge number two of the panel j
        z_2 = Grid_Points_P[N_Panel_P[j,1],2]     # Z value of the edge number two of the panel j

        x_3 = Grid_Points_P[N_Panel_P[j,3],0]     # X value of the edge number four of the panel j
        y_3 = Grid_Points_P[N_Panel_P[j,3],1]     # Y value of the edge number four of the panel j
        z_3 = Grid_Points_P[N_Panel_P[j,3],2]     # Z value of the edge number four of the panel j

        x_4 = Grid_Points_P[N_Panel_P[j,2],0]     # X value of the edge number three of the panel j
        y_4 = Grid_Points_P[N_Panel_P[j,2],1]     # Y value of the edge number three of the panel j
        z_4 = Grid_Points_P[N_Panel_P[j,2],2]     # Z value of the edge number three of the panel j

        radius_cp[j] = np.sqrt(p_y_mdp**2 + p_z_mdp**2)  # Radius for the chosen control point of the propeller

        cos_theta_c_skin = p_z_mdp/radius_cp[j]
        sin_theta_c_skin = p_y_mdp/radius_cp[j]

        # VELOCITIES IN THE CONTROL POINTS FROM THE ONSET FLOW

        U_O_Onset = np.interp (radius_cp[j],r_R_P,U_O_P)    # Wake (Axial) in the control points (s)
        U_T_Onset = np.interp (radius_cp[j],r_R_P,U_T_P)    # Wake (Tangential) in the control points (s)
        U_R_Onset = np.interp (radius_cp[j],r_R_P,U_R_P)    # Wake (Radial) in the control points (s)

        u_x_onset = - U_O_Onset      # Onset Flow (x)
        u_y_onset  = U_R_Onset*p_y_mdp/radius_cp[j] - U_T_Onset*p_z_mdp/radius_cp[j] + Var.Omega*p_z_mdp   # Onset Flow (y)
        u_z_onset  = U_R_Onset*p_z_mdp/radius_cp[j] + U_T_Onset*p_y_mdp/radius_cp[j] - Var.Omega*p_y_mdp   # Onset Flow (z)

        # VELOCITIES IN THE CONTROL POINTS FROM THE PANELS

        u_x_panels = 0
        # Initialization of the variable used to store the induced velocity from the panels of the propeller (x)
        u_y_panels = 0
        # Initialization of the variable used to store the induced velocity from the panels of the propeller (y)
        u_z_panels = 0
        # Initialization of the variable used to store the induced velocity from the panels of the propeller (z)

        # Loop used to select the spanwise level that induces velocity on the control points of the propeller
        for n in range (Var.Msp):
            u_x_panels_0 = 0
            #Initialization of the variable used to calculate the induced velocity from the panels of the propeller (x)
            u_y_panels_0 = 0
            # Initialization of the variable used to calculate the induced velocity from the panels of the propeller (y)
            u_z_panels_0 = 0
            # Initialization of the variable used to calculate the induced velocity from the panels of the propeller (z)

            # Loop used to select the panel that induces velocity on the control points of the propeller
            for m in range (Var.Nch):
                npl = m + (n) * Var.Nch

                u_x_temp,u_y_temp,u_z_temp = Panel_Induced_Velocity_Propeller (npl,5,0,p_x_mdp,p_y_mdp,p_z_mdp)
                # Induced velocity from the selected panel on the chosen control point of the propeller

                u_x_panels_0 = u_x_panels_0 + Weight_P[n,m] * u_x_temp
                # Temporary variable used to calculate the induced velocity from the panels of the propeller (x)
                u_y_panels_0 = u_y_panels_0 + Weight_P[n,m] * u_y_temp
                # Temporary variable used to calculate the induced velocity from the panels of the propeller (y)
                u_z_panels_0 = u_z_panels_0 + Weight_P[n,m] * u_z_temp
                # Temporary variable used to calculate the induced velocity from the panels of the propeller (z)


            u_x_panels = u_x_panels + Gamma_TE_P[n] * u_x_panels_0   # Induced velocity from the panels of the propeller (x)
            u_y_panels = u_y_panels + Gamma_TE_P[n] * u_y_panels_0   # Induced velocity from the panels of the propeller (y)
            u_z_panels = u_z_panels + Gamma_TE_P[n] * u_z_panels_0   # Induced velocity from the panels of the propeller (z)

        # VELOCITIES IN THE CONTROL POINTS FROM THE HORSESHOE VORTEX
```

```python

        u_x_trail = 0
        # Initialization of the variable used to calculate the induced velocity from the trailing vortices of the propeller (x)
        u_y_trail = 0
        # Initialization of the variable used to calculate the velocity from the trailing vortices of the propeller (y)
        u_z_trail = 0
        # Initialization of the variable used to calculate the induced velocity from the trailing vortices of the propeller (z)

        x_T_E_1 = Grid_Points_P[0,0]   # First point of the first trailing vortex of the propeller (x)
        y_T_E_1 = Grid_Points_P[0,1]   # First point of the first trailing vortex of the propeller (y)
        z_T_E_1 = Grid_Points_P[0,2]   # First point of the first trailing vortex of the propeller (z)

        u_x_trail_1,u_y_trail_1,u_z_trail_1 = Trailing_Vortices_Propeller(0,p_x_mdp,p_y_mdp,p_z_mdp)
        # Induced velocity from the transition wake and from the semi-infinite helicoidal vortex of the propeller (First)

        # Loop used to select the trailing vortex that induces velocity on the control points of the propeller
        for n in range (Var.Msp):
            n_1 = n + 1
            n_2 = (n+1) * (Var.Nch+1)

            u_x_trail_2,u_y_trail_2,u_z_trail_2 = Trailing_Vortices_Propeller(n_1,p_x_mdp,p_y_mdp,p_z_mdp)
            # Induced velocity from the transition wake and from the semi-infinite helicoidal vortex of the propeller (Second)

            x_T_E_2 = Grid_Points_P[n_2,0]  # Second point of the trailing vortex of the propeller (x)
            y_T_E_2 = Grid_Points_P[n_2,1]  # Second point of the trailing vortex of the propeller (y)
            z_T_E_2 = Grid_Points_P[n_2,2]  # Second point of the trailing vortex of the propeller (z)

            U_x_s,U_y_s,U_z_s =
                Biot_Savart_Propeller(Var.Z_Blade_P,x_T_E_1,y_T_E_1,z_T_E_1,x_T_E_2,y_T_E_2,z_T_E_2,p_x_mdp,p_y_mdp,p_z_mdp)
            # Induced velocity from the bound vortex selected of the propeller

            u_x_trail = u_x_trail + Gamma_TE_P[n] * (u_x_trail_1 - u_x_trail_2 + U_x_s)
            # Induced velocity from the horseshoe vortex of the propeller (x)
            u_y_trail = u_y_trail + Gamma_TE_P[n] * (u_y_trail_1 - u_y_trail_2 + U_y_s)
            # Induced velocity from the horseshoe vortex of the propeller (y)
            u_z_trail = u_z_trail + Gamma_TE_P[n] * (u_z_trail_1 - u_z_trail_2 + U_z_s)
            # Induced velocity from the horseshoe vortex of the propeller (z)

            x_T_E_1 = x_T_E_2   # For the next loop
            y_T_E_1 = y_T_E_2   # For the next loop
            z_T_E_1 = z_T_E_2   # For the next loop

            u_x_trail_1 = u_x_trail_2     # For the next loop
            u_y_trail_1 = u_y_trail_2     # For the next loop
            u_z_trail_1 = u_z_trail_2     # For the next loop

        # TOTAL INDUCED VELOCITY

        u_x_tot =  u_x_onset + u_x_trail + u_x_panels  # Total induced velocity on the propeller (x)
        u_y_tot =  u_y_onset + u_y_trail + u_y_panels  # Total induced velocity on the propeller (y)
        u_z_tot =  u_z_onset + u_z_trail + u_z_panels  # Total induced velocity on the propeller (z)

        # SKIN FRICTION DRAG

        s = Area_Panel (x_1,y_1,z_1,x_2,y_2,z_2,x_3,y_3,z_3,x_4,y_4,z_4)
        # Area of the panel where the control point is located

        point_x_2,point_y_2,point_z_2,vector_xx,vector_yy,vector_zz = Mid_Vect_Propeller(j,1)
        # This subroutine is used to calculate the midpoint of the panel side number 2
        point_x_4,point_y_4,point_z_4,vector_xx,vector_yy,vector_zz = Mid_Vect_Propeller(j,3)
        # This subroutine is used to calculate the midpoint of the panel side number 2

        vector_panel[0] = point_x_4 - point_x_2    # Tangent vector to the panel (x)
        vector_panel[1] = point_y_4 - point_y_2    # Tangent vector to the panel (y)
        vector_panel[2] = point_z_4 - point_z_2    # Tangent vector to the panel (z)

        vector_panel/= np.linalg.norm(vector_panel, ord=2)     # Unit tangent vector to the panel

        tangentialDirection[0] = 0.0                  # Tangent vector in yz plane
        tangentialDirection[1] = cos_theta_c_skin     # Tangent vector in yz plane
        tangentialDirection[2]= - sin_theta_c_skin    # Tangent vector in yz plane

        tangentialDirection/= np.linalg.norm(tangentialDirection, ord = 2)   # Unit tangent vector in yz plane

        V_tang = np.dot([u_x_tot, u_y_tot, u_z_tot], vector_panel)           # Tangent velocity to the panel

        dragInPanelDirection = Var.Skin_Coeff * 0.5 * Var.rho * (abs(V_tang)*V_tang) * s  # Tangent force to the panel

        T_Skin_F = T_Skin_F + dragInPanelDirection * vector_panel[0]    # Skin friction drag (Thrust) - X force to the panel

        Q_Skin_F = Q_Skin_F + dragInPanelDirection * np.dot(vector_panel, tangentialDirection) * radius_cp[j]
            # Skin friction drag (Torque) - X force to the panel

    T_fr_P =  T_Skin_F
    Q_fr_P = - Q_Skin_F

    with open ("output/Propeller_Drag.txt", "w") as file:
        file.write("  Drag T         Drag Q           Drag KT          Drag KQ \n")
        file.write(f"{(Var.Z_Blade_P * T_fr_P):9.1f}     {(Var.Z_Blade_P * Q_fr_P):9.1f}\
        {(Var.Z_Blade_P * T_fr_P / ((Var.Omega / (2 * np.pi))**2 * Var.rho * (Var.Rad_P * 2)**4)):0.6f}\
        {Var.Z_Blade_P * Q_fr_P / ((Var.Omega / (2 * np.pi))**2 * Var.rho * (Var.Rad_P * 2)**5):0.6f}\n")

    # OPTIMIZATION PARAMETERS - OUTPUT

    mid_point = Var.Nch//2

    # Loop used to select the closest control points to the midchord line (Chordwise)
    for j in range (Var.Msp):
        mid_point_cp = (mid_point) + (j) * Var.Nch

        p_x_mdp = Control_Points_P[mid_point_cp,0]
        p_y_mdp = Control_Points_P[mid_point_cp,1]
```

```
        p_z_mdp = Control_Points_P[mid_point_cp,2]
        # X coordinate of the chosen control point of the propeller
        # Y coordinate of the chosen control point of the propeller
        # Z coordinate of the chosen control point of the propeller

        x_1 = Grid_Points_P[N_Panel_P[mid_point_cp,0],0]
        y_1 = Grid_Points_P[N_Panel_P[mid_point_cp,0],1]
        z_1 = Grid_Points_P[N_Panel_P[mid_point_cp,0],2]
        # X value of the edge number one of the panel
        # Y value of the edge number one of the panel
        # Z value of the edge number one of the panel

        x_2 = Grid_Points_P[N_Panel_P[mid_point_cp,1],0]
        y_2 = Grid_Points_P[N_Panel_P[mid_point_cp,1],1]
        z_2 = Grid_Points_P[N_Panel_P[mid_point_cp,1],2]
        # X value of the edge number two of the panel
        # Y value of the edge number two of the panel
        # Z value of the edge number two of the panel

        x_3 = Grid_Points_P[N_Panel_P[mid_point_cp,3],0]
        y_3 = Grid_Points_P[N_Panel_P[mid_point_cp,3],1]
        z_3 = Grid_Points_P[N_Panel_P[mid_point_cp,3],2]
        # X value of the edge number four of the panel
        # Y value of the edge number four of the panel
        # Z value of the edge number four of the panel

        x_4 = Grid_Points_P[N_Panel_P[mid_point_cp,2],0]
        y_4 = Grid_Points_P[N_Panel_P[mid_point_cp,2],1]
        z_4 = Grid_Points_P[N_Panel_P[mid_point_cp,2],2]
        # X value of the edge number three of the panel
        # Y value of the edge number three of the panel
        # Z value of the edge number three of the panel

        vec_x,vec_y,vec_z = Normal_Vector (x_1,y_1,z_1,x_2,y_2,z_2,x_3,y_3,z_3,x_4,y_4,z_4)
        # This subroutine calculates the normal vector for the chosen panel

        vector_x[j] = vec_x       # X component of the vector
        vector_y[j] = vec_y       # Y component of the vector
        vector_z[j] = vec_z       # Z component of the vector

        r_cp_a[j] = np.sqrt(p_y_mdp**2 + p_z_mdp**2)   # Radius for the chosen control point of the propeller

        cos_theta_c[j] = p_z_mdp/r_cp_a[j]
        sin_theta_c[j] = p_y_mdp/r_cp_a[j]

        # ONSET

        U_0_P_Onset = np.interp(r_cp_a[j],r_R_P,U_0_P)    # Wake (Axial) in the midpoints (s)
        U_T_P_Onset = np.interp(r_cp_a[j],r_R_P,U_T_P)    # Wake (Tangential) in the midpoints (s)
        U_R_P_Onset = np.interp(r_cp_a[j],r_R_P,U_R_P)    # Wake (Radial) in the grid midpoints (s)

        u_x_onset = - U_0_P_Onset       # Onset Flow (x)
        u_y_onset = U_R_P_Onset*p_y_mdp/r_cp_a[j] - U_T_P_Onset*p_z_mdp/r_cp_a[j] + Var.Omega*p_z_mdp    # Onset Flow (y)
        u_z_onset = U_R_P_Onset*p_z_mdp/r_cp_a[j] + U_T_P_Onset*p_y_mdp/r_cp_a[j] - Var.Omega*p_y_mdp    # Onset Flow (z)

        # VELOCITIES IN THE CONTROL POINTS FROM THE PANELS OF THE PROPELLER

        u_x_panels = 0
        # Initialization of the variable used to store the induced velocity from the panels of the propeller (x)
        u_y_panels = 0
        # Initialization of the variable used to store the induced velocity from the panels of the propeller (y)
        u_z_panels = 0
        # Initialization of the variable used to store the induced velocity from the panels of the propeller (z)

        # Loop used to select the spanwise level that induces velocity on the control points of the propeller
        for n in range (Var.Msp):
            u_x_panels_0 = 0
            # Initialization of the variable used to calculate the induced velocity from the panels of the propeller (x)
            u_y_panels_0 = 0
            # Initialization of the variable used to calculate the induced velocity from the panels of the propeller (y)
            u_z_panels_0 = 0
            # Initialization of the variable used to calculate the induced velocity from the panels of the propeller (z)

            # Loop used to select the panel that induces velocity on the control points of the propeller
            for m in range (Var.Nch):
                npl = m + (n) * Var.Nch

                u_x_temp,u_y_temp,u_z_temp = Panel_Induced_Velocity_Propeller(npl,5,0,p_x_mdp,p_y_mdp,p_z_mdp)

                # Induced velocity from the selected panel on the chosen control point of the propeller
                u_x_panels_0 = u_x_panels_0 + Weight_P[n,m] * u_x_temp
                # Temporary variable used to calculate the induced velocity from the panels of the propeller (x)
                u_y_panels_0 = u_y_panels_0 + Weight_P[n,m] * u_y_temp
                # Temporary variable used to calculate the induced velocity from the panels of the propeller (y)
                u_z_panels_0 = u_z_panels_0 + Weight_P[n,m] * u_z_temp
                # Temporary variable used to calculate the induced velocity from the panels of the propeller (z)

            u_x_panels = u_x_panels + Gamma_TE_P[n] * u_x_panels_0    # Induced velocity from the panels of the propeller (x)
            u_y_panels = u_y_panels + Gamma_TE_P[n] * u_y_panels_0    # Induced velocity from the panels of the propeller (y)
            u_z_panels = u_z_panels + Gamma_TE_P[n] * u_z_panels_0    # Induced velocity from the panels of the propeller (z)

        # VELOCITIES IN THE CONTROL POINTS FROM THE HORSESHOE VORTEX OF THE PROPELLER

        u_x_trail = 0
        # Initialization of the variable used to calculate the induced velocity from the trailing vortices of the propeller (x)
        u_y_trail = 0
        # Initialization of the variable used to calculate the induced velocity from the trailing vortices of the propeller (y)
        u_z_trail = 0
        # Initialization of the variable used to calculate the induced velocity from the trailing vortices of the propeller (z)

        x_T_E_1 = Grid_Points_P[0,0]     # First point of the first trailing vortex of the propeller (x)
        y_T_E_1 = Grid_Points_P[0,1]     # First point of the first trailing vortex of the propeller (y)
```

```
        z_T_E_1 = Grid_Points_P[0,2]    # First point of the first trailing vortex of the propeller (z)

        u_x_trail_1,u_y_trail_1,u_z_trail_1 = Trailing_Vortices_Propeller(0,p_x_mdp,p_y_mdp,p_z_mdp)
        # Induced velocity from the transition wake and from the semi-infinite helicoidal vortex of the propeller (First)

        # Loop used to select the trailing vortex that induces velocity on the control points of the propeller
        for n in range (Var.Msp):
            n_1 = n + 1
            n_2 = (n+1) * (Var.Nch+1)

            u_x_trail_2,u_y_trail_2,u_z_trail_2 = Trailing_Vortices_Propeller(n_1,p_x_mdp,p_y_mdp,p_z_mdp)
            # Induced velocity from the transition wake and from the semi-infinite helicoidal vortex of the propeller (Second)

            x_T_E_2 = Grid_Points_P[n_2,0]  # Second point of the trailing vortex of the propeller (x)
            y_T_E_2 = Grid_Points_P[n_2,1]  # Second point of the trailing vortex of the propeller (y)
            z_T_E_2 = Grid_Points_P[n_2,2]  # Second point of the trailing vortex of the propeller (z)

            U_x_s,U_y_s,U_z_s = Biot_Savart_Propeller(Var.Z_Blade_P,x_T_E_1,y_T_E_1,
                                                  z_T_E_1,x_T_E_2,y_T_E_2,z_T_E_2,p_x_mdp,p_y_mdp,p_z_mdp)
            # Induced velocity from the bound vortex selected of the propeller

            u_x_trail = u_x_trail + Gamma_TE_P[n] * (u_x_trail_1 - u_x_trail_2 + U_x_s)
            # Induced velocity from the horseshoe vortex of the propeller (x)
            u_y_trail = u_y_trail + Gamma_TE_P[n] * (u_y_trail_1 - u_y_trail_2 + U_y_s)
            # Induced velocity from the horseshoe vortex of the propeller (y)
            u_z_trail = u_z_trail + Gamma_TE_P[n] * (u_z_trail_1 - u_z_trail_2 + U_z_s)
            # Induced velocity from the horseshoe vortex of the propeller (z)

            x_T_E_1 = x_T_E_2               # For the next loop
            y_T_E_1 = y_T_E_2               # For the next loop
            z_T_E_1 = z_T_E_2               # For the next loop

            u_x_trail_1 = u_x_trail_2       # For the next loop
            u_y_trail_1 = u_y_trail_2       # For the next loop
            u_z_trail_1 = u_z_trail_2       # For the next loop

        # TOTAL INDUCED VELOCITY

        u_x_tot_a[j] = u_x_onset + u_x_trail + u_x_panels    # Total induced velocity on the propeller (x)
        u_y_tot_a[j] = u_y_onset + u_y_trail + u_y_panels    # Total induced velocity on the propeller (y)
        u_z_tot_a[j] = u_z_onset + u_z_trail + u_z_panels    # Total induced velocity on the propeller (z)

        u_tang_skin[j] = - u_y_tot_a[j] * cos_theta_c[j] + u_z_tot_a[j] * sin_theta_c[j]
        # Total tangential induced velocity in the control points of the propeller
        u_rel_skin[j] = np.sqrt((u_x_tot_a[j]**2) + (u_tang_skin[j]**2))

    # THRUST FOR EACH STRIP

    # Spanwise loop
    for m in range (Var.Msp):
        npl_TE = (m)*Var.Nch

        L_0_x = 0   # Initialization of the temporary variable used to calculate the lift of the stator (x)
        L_0_y = 0   # Initialization of the temporary variable used to calculate the lift of the stator (y)
        L_0_z = 0   # Initialization of the temporary variable used to calculate the lift of the stator (z)

        # Chordwise loop
        for n in range (Var.Nch):
            npl = n + (m)*Var.Nch

            L_00_x = 0  # Initialization of the temporary variable used to calculate the lift of the stator (x)
            L_00_y = 0  # Initialization of the temporary variable used to calculate the lift of the stator (y)
            L_00_z = 0  # Initialization of the temporary variable used to calculate the lift of the stator (z)

            # Panel loop
            for k in range (4):

                xkx,xky,xkz,xlk,ylk,zlk = Mid_Vect_Propeller(npl,k)

                L_00_x = L_00_x + zlk*V_Tot_P[npl,k,1] - ylk*V_Tot_P[npl,k,2]  # Lift generated by side k panel npl (x)
                L_00_y = L_00_y + xlk*V_Tot_P[npl,k,2] - zlk*V_Tot_P[npl,k,0]  # Lift generated by side k panel npl (y)
                L_00_z = L_00_z - xlk*V_Tot_P[npl,k,1] + ylk*V_Tot_P[npl,k,0]  # Lift generated by side k panel npl (z)

            L_0_x = L_0_x + Weight_P[m,n] * L_00_x     # Lift generated by the panel npl (x)
            L_0_y = L_0_y + Weight_P[m,n] * L_00_y     # Lift generated by the panel npl (y)
            L_0_z = L_0_z + Weight_P[m,n] * L_00_z     # Lift generated by the panel npl (z)

        xkx,xky,xkz,xlk,ylk,zlk = Mid_Vect_Propeller(npl_TE,3)

        # Lift (Ring) - x
        L_Ring_x[m] = Gamma_Panel_P[npl_TE]*L_0_x - Gamma_Panel_P[
            npl_TE]*zlk*V_Tot_P[npl_TE,3,1] + Gamma_Panel_P[npl_TE]*ylk*V_Tot_P[npl_TE,3,2]
        # Lift (Ring) - y
        L_Ring_y[m] = Gamma_Panel_P[npl_TE]*L_0_y - Gamma_Panel_P[
            npl_TE]*xlk*V_Tot_P[npl_TE,3,2] + Gamma_Panel_P[npl_TE]*zlk*V_Tot_P[npl_TE,3,0]
           # Lift (Ring) - z
        L_Ring_z[m] = Gamma_Panel_P[npl_TE]*L_0_z + Gamma_Panel_P[
            npl_TE]*xlk*V_Tot_P[npl_TE,3,1] - Gamma_Panel_P[npl_TE]*ylk*V_Tot_P[npl_TE,3,0]
        # Total Lift of the ring m
        L_Ring[m] = L_Ring_x[m]*vector_x[m] + L_Ring_y[m]*vector_y[m] + L_Ring_z[m]*vector_z[m]

    # CORRECTION FACTORS

    for m in range (Var.Msp):
        Chord_P_skin[m] = np.interp(r_cp_a[m],S_Distr_P,Chord_P)       # Value of the chord
        Thick_P_skin[m] = np.interp(r_cp_a[m],S_Distr_P,Thick_P)       # Value of the thickness

    Thick_0 = (Thick_P_skin[1]-Thick_P_skin[0])/(r_cp_a[1]-r_cp_a[0])*(-r_cp_a[0]) + Thick_P_skin[0]  # Pitch at the hub

    for m in range (Var.Msp):
        Max_Skew = max(Skew_P)*180/np.pi
        a = (3.5*AeAo) / (np.sqrt(r_cp_a[m]/Var.Rad_P * np.tan(beta[m]))) * (r_cp_a[m]/Var.Rad_P - 0.5)**2
```

```python
        b = 0.71 * np.sqrt(r_cp_a[m]/Var.Rad_P * np.tan(beta[m])) + 0.56 * (
            AeAo)**2 + (r_cp_a[m]/Var.Rad_P)*(5-Var.Z_Blade_P)/Var.Z_Blade_P + 0.46

        Coeff_Corr_Camber[m] = a + b

        Coeff_Corr_Thick[m] = 2*(5 + Var.Z_Blade_P)*Var.Z_Blade_P*Thick_0/(Var.Rad_P*2) * AeAo * (1-r_cp_a[m]/Var.Rad_P)**2

        d = 1.2 * AeAo + 0.65 - 0.07*(2-np.pi*r_cp_a[m]/Var.Rad_P * np.tan(beta[m]))**3
        e = 55/np.sqrt(np.pi*r_cp_a[m]/Var.Rad_P * np.tan(beta[m]))*AeAo*(r_cp_a[
            m]/Var.Rad_P - 0.55)**4 + 1.2*r_cp_a[m]/Var.Rad_P*(5-Var.Z_Blade_P)/Var.Z_Blade_P
        f = 0.08 * Max_Skew * (1- 20 * abs((r_cp_a[m]/Var.Rad_P - 0.4)**3))

        Coeff_Corr_Alpha[m] = d + e + f

    with open("output/Propeller_Correction_Factors.txt","w") as file:
        file.write("   K_Camber    K_Thickness     K_Alpha\n")
        for m in range(Var.Msp):
            file.write(f"    {Coeff_Corr_Camber[m]:7.4f}      {Coeff_Corr_Thick[m]:7.4f}       {Coeff_Corr_Alpha[m]:7.4f}\n")

    for m in range (Var.Msp):
        C_L_Local[m] = abs(L_Ring[m])/(0.5*(u_rel_skin[m])**2*s_ring[m])       # Lift Coefficient
        Camber_Dimless[m] = (C_L_Local[m]*(1.0-Var.cny)) * 0.067 * Coeff_Corr_Camber[m] # / (1 - 0.83*Thick_P_skin[m]/Chord_P_skin[m])
        # Value of the camber
        ideal_angle_attack[m]= 1.40*(C_L_Local[m]*(1.0-Var.cny))/(180.0)*np.pi         # Ideal angle of attack - 2D
        angle_attack[m] = (C_L_Local[m]*Var.cny)/(np.pi*2.0) + ideal_angle_attack[m]  # Angle of attack -  2D
        beta_temp_surface[m] = angle_attack[m]*Coeff_Corr_Alpha[m] + beta[m] + Coeff_Corr_Thick[m]/180*np.pi
        pitch_cp_final_surface[m] = np.tan(beta_temp_surface[m]) * 2.0 * np.pi * r_cp_a[m]

    with open ("output/Propeller_Lift_Coefficient_Local_Parameters.txt","w") as file:
        file.write("  C_L_Local      Area          Beta          Angle of Attack         f/c        Ideal angle of attack      Radius\n")
        for m in range (Var.Msp):
                       file.write(f"  {C_L_Local[m]:7.4f}      {s_ring[m]:7.4f}    {beta[m]:13.9f}     {angle_attack[m]:13.9f}"
                        f"    {Camber_Dimless[m]:13.9f}       {ideal_angle_attack[m]:13.9f}      {r_cp_a[m]:13.9}\n")
    with open ("output/Propeller_Pitch_Control_Points_Angle_Add.txt","w") as file:
        file.write(" Spanw.      Radius           Pitch/D\n")
        for j in range (Var.Msp):
            file.write(f" {j:3d}     {r_cp_a[j]:13.9f}      {(pitch_cp_final_surface[j]/(Var.Rad_P*2)):13.9f}\n")

    return T_fr_P, Q_fr_P


T_fr_P,Q_fr_P = Skin_Friction_Drag()
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the induced velocities (coefficient)
from the transition wake and from the semi-infinite helicoidal vortex at a
single point for all propeller blades. Specifically, it addresses the selected
trailing vortex (n_tral_vortex) - options 1, 2, 3, 4 with Msp set to 3.
"""


import numpy as np
import sources.Variables as Var
from sources.De_Jong_P import De_Jong
from sources.Biot_Savart_Propeller_P import Biot_Savart_Propeller
from sources.Helix_P import Helix


def Trailing_Vortices_Propeller(n_tral_vortex, px, py, pz):
    Points_Trans_Wake_P = np.loadtxt("output/Propeller_Points_Trans_Wake.txt", skiprows= 1, usecols= (1,2,3))
    Grid_Points_P = np.loadtxt("output/Propeller_Grid_Points.txt")
    N_Bound_Vortex_P = np.loadtxt("output/Propeller_N_Bound_Vortex.txt",dtype= 'int')
    N_Bound_Vortex_P = N_Bound_Vortex_P.reshape((Var.Msp+1, 1))

    U_x = 0.0          # Initialization of the variable U_x
    U_y = 0.0          # Initialization of the variable U_y
    U_z = 0.0          # Initialization of the variable U_z

    # INDUCED VELOCITIES FROM SEMI-INFINITE HELICOIDAL VORTEX

    pyy = py
    pzz = pz

    k_2 = (n_tral_vortex + n_tral_vortex * Var.N_P_L)+Var.N_P_L
    # k_2 is the location in Points_Trans_Wake_P for the last point of that trailing vortex
    k_0 = k_2 - Var.N_P_L
    # k_0 is the location in Points_Trans_Wake_P for the first point of that trailing vortex (T.E.)
    k_2 = int(k_2)
    k_0 = int(k_0)
    n_tral_vortex = int (n_tral_vortex)

    x_1 = - Points_Trans_Wake_P[k_2,0]
    # First point for the semi-infinite helicoidal vortex (x) The - is because in infv the vortex starts at -infinity and stops at -x

    r_1 = Points_Trans_Wake_P[k_2,1]     # First point for the semi-infinite helicoidal vortex (Radius)

    p_1 = Points_Trans_Wake_P[k_2,2]     # First point for the semi-infinite helicoidal vortex (pitch)

    x_T_E = Points_Trans_Wake_P[k_0,0]
    r_T_E = Points_Trans_Wake_P[k_0,1]
    p_T_E = Points_Trans_Wake_P[k_0,2]

    delta_theta = 2*np.pi/float(Var.Z_Blade_P)

    for i in range(Var.Z_Blade_P):
```

```
        y_T_E = Grid_Points_P[N_Bound_Vortex_P[n_tral_vortex,0],1]
        theta = np.arcsin(- y_T_E / r_T_E) # Theta for the T.E. point
        xki = theta - 2*np.pi* x_T_E/p_T_E # Phase angle phi

        U_xx, U_yy, U_zz = De_Jong (x_1,r_1,p_1,xki,px, pyy, pzz)
        # This subroutine calculates the induced velocity for the ultimate wake that starts in x1,r1,z1 (De Jong) for the point
            px,pyy,pzz

        U_x = U_x + U_xx
        U_y = U_y + U_yy
        U_z = U_z + U_zz

        theta_blade = (float(i+1)*delta_theta)# This is the angle of the blade that induces velocity on the reference blade

        if (theta_blade < np.pi):
            pyy = py*np.cos(theta_blade) + pz*np.sin(theta_blade)
            pzz = pz*np.cos(theta_blade) - py*np.sin(theta_blade)
            # In order to calculate the induced velocity in the point px,py,pz from the semi-infinite
            # helicoidal vortices we do not change the helix (which is always located on the reference blade),
            # but we change the location of the point and we keep costant the relative distance between the semi-infinite helicoidal
                vortex
            # and the point (the rotation depends on the location of the blade that induces velocity)
        else:
            pyy = py*np.cos(2*np.pi-theta_blade) - pz*np.sin(2*np.pi-theta_blade)
            pzz = pz*np.cos(2*np.pi-theta_blade) + py*np.sin(2*np.pi-theta_blade)

    # INDUCED VELOCITIES FROM THE TRANSITION WAKE

    d_r = 0.0
    d_p = 0.0

    x_1 = Points_Trans_Wake_P[k_2,0]

    for i in range(Var.N_P_L):
            k2_i = k_2 - (i+1)

            x_2 = Points_Trans_Wake_P[k2_i,0]   # Second point for the selected side of the transition wake (x)
            r_2 = Points_Trans_Wake_P[k2_i,1]   # Second point for the selected side of the transition wake (radius)
            p_2 = Points_Trans_Wake_P[k2_i,2]   # Second point for the selected side of the transition wake (pitch)

            a_r,b_r,c_r = Helix(x_1,x_2,r_1,r_2,d_r)  # Calculates the coefficients a,b,c used in the polynomium
                                                      # for the radius for that side of the transition wake

            a_p,b_p,c_p = Helix(x_1,x_2,p_1,p_2,d_p)  # Calculates the coefficients a,b,c used in the polynomium
                                                      # for the pitch for that side of the transition wake

            delta_x = (x_2-x_1) / float(Var.sub_interv)

            x_11 = x_1   # First value of the element line (x) of the transition wake
            r_11 = r_1   # First value of the element line (radius) of the transition wake
            p_11 = p_1   # First value of the element line (pitch) of the transition wake

            theta_1 = (xki + (2*np.pi) * x_11) / p_11 # First value of the element line (theta) of the transition wake

            y_11 = - r_11 * np.sin(theta_1) # First value of the element line (y) of the transition wake
            # Theta is positive in the other direction (sin(-theta) = - sin(theta))

            z_11 = r_11 * np.cos(theta_1)   # First value of the element line (z) of the transition wake


            for j in range (Var.sub_interv):

                x_12 = x_11 + delta_x                  # Second value of the element line (x) of the transition wake
                r_12 = a_r*(x_12**2) + b_r*x_12 + c_r   # Second value of the element line (radius) of the transition wake
                p_12 = a_p*(x_12**2) + b_p*x_12 + c_p   # Second value of the element line (pitch) of the transition wake

                theta_2 = xki + (2*np.pi) * x_12 / p_12 # Second value of the element line (theta) of the transition wake
                y_12 = - r_12 * np.sin(theta_2)         # Second value of the element line (y) of the transition wake
                z_12 = r_12 * np.cos(theta_2)           # Second value of the element line (z) of the transition wake

                U_x_w,U_y_w,U_z_w = Biot_Savart_Propeller (Var.Z_Blade_P,x_11,y_11,z_11,x_12,y_12,z_12,px,py,pz)

                U_x = U_x + U_x_w
                U_y = U_y + U_y_w
                U_z = U_z + U_z_w

                x_11 = x_12
                y_11 = y_12
                z_11 = z_12

            x_1 = x_2
            r_1 = r_2
            p_1 = p_2

            d_r = 2 * a_r * x_2 + b_r
            d_p = 2 * a_p * x_2 + b_p

    return (U_x, U_y, U_z)
```

```
'''
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This module contains the fixed variables.
'''


```

```
# Density of water (kg/m^3)
rho = 1025
# Velocity of the ship (m/s)
V_Ship = 12.8611
# Convergence criteria
epsi = 0.0001
# Number of subdivisions of the input values
N_Iter = 500
# Total required thrust (N)
Tr = 3256000

# Preserved total required thrust for calculations
Tr_P = Tr
# Number of panels (spanwise)
Msp = 5
# Number of panels (chordwise)
Nch = 5
# Flat plate coefficient (0: pure rooftop, 0.5: half rooftop, 1: pure flat
# plate)
cny = 0.5

# Angular velocity (rad/s) - Propeller
Omega = 9.886
# Radius (m) - Propeller
Rad_P = 4.5
# Radius for the hub - Propeller
R_Hub_P = 0.189 * Rad_P
# Number of blades - Propeller
Z_Blade_P = 6
# Skin friction drag coefficient
Skin_Coeff = 0.008
# Number of straight line vortices (Transition Wake)
N_P_L = 5
# Number of subintervals for each line of the transition wake
sub_interv = 60
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine computes the induced velocities at the midpoints
of segments (coefficient) emanating from the horseshoe vortex across all blades
of the propeller. Furthermore, it calculates the total "velocity matrix" at
these midpoints, excluding the onset flow.
"""


import numpy as np
import sources.Variables as Var
from sources.Induced_Grid_Propeller_P import Induced_Grid_Propeller
from sources.Mid_Vect_Propeller_P import Mid_Vect_Propeller
from sources.Biot_Savart_Propeller_P import Biot_Savart_Propeller
from sources.Trailing_Vortices_Propeller_P import Trailing_Vortices_Propeller


def Velocity_Total_No_Onset_Propeller():

    Horseshoe_P = np.loadtxt("output/Propeller_Horseshoe.txt", dtype='int')
    Grid_Points_P = np.loadtxt("output/Propeller_Grid_Points.txt")
    N_Bound_Vortex_P = np.loadtxt("output/Propeller_N_Bound_Vortex.txt", dtype='int')
    N_Bound_Vortex_P = N_Bound_Vortex_P.reshape((Var.Msp+1, 1))

    I_P_Points_P = (Var.Msp*Var.Nch)
    V_Grid_P = Induced_Grid_Propeller()
    V_Tral_P = np.zeros((Var.Msp+1, I_P_Points_P, 4,3))
    V_Ind_P = np.zeros((Var.Msp, I_P_Points_P, 4,3))

    # HORSESHOE VORTEX

    for i in range (I_P_Points_P):
    # I used this loop in order to select the panel where the point xx,xy,xz is located

        N_P_V = Horseshoe_P[0,0]
        # First point of the first trailing vortex selected (Segment)

        for k in range (4):
        # I used this loop in order to select the side where the point xx,xy,xz
        #is located and to calculate the induced velocity
        # from the transition wake and from the semi-infinite helicoidal vortex for the selected trailing vortex (N_P_V)
            xx,xy,xz,v_xx,v_xy,v_xz = Mid_Vect_Propeller(i,k)   # This subroutine is used to calculate the midpoint xx,xy,xz

            U_x1,U_y1,U_z1 = Trailing_Vortices_Propeller (N_P_V,xx,xy,xz)
            # Induced velocity from the transition wake and from the semi-infinite helicoidal vortex (First)

            V_Tral_P[0,i,k,0] = U_x1
            V_Tral_P[0,i,k,1] = U_y1
            V_Tral_P[0,i,k,2] = U_z1

        x_1 = Grid_Points_P[N_Bound_Vortex_P[N_P_V,0],0] # X coordinate of the second point of the segment of the first trailing vortex
        y_1 = Grid_Points_P[N_Bound_Vortex_P[N_P_V,0],1] # Y coordinate of the second point of the segment of the first trailing vortex
        z_1 = Grid_Points_P[N_Bound_Vortex_P[N_P_V,0],2] # Z coordinate of the second point of the segment of the first trailing vortex

        for j in range(Var.Msp):                        # This loop is used to select the horseshoe vortex

            j_2 = j+1
```

```
            N_P_V_2 = Horseshoe_P[j,1]    # Second point of the trailing vortex selected (Segment)

            x_2 = Grid_Points_P[N_Bound_Vortex_P[N_P_V_2,0],0]  # X coordinate of the second point of the segment of the trailing
                vortex
            y_2 = Grid_Points_P[N_Bound_Vortex_P[N_P_V_2,0],1]  # Y coordinate of the second point of the segment of the trailing
                vortex
            z_2 = Grid_Points_P[N_Bound_Vortex_P[N_P_V_2,0],2]  # Z coordinate of the second point of the segment of the trailing
                vortex

            for k in range (4):
                xx,xy,xz,v_xx,v_xy,v_xz= Mid_Vect_Propeller(i,k)

                U_x2,U_y2,U_z2 = Trailing_Vortices_Propeller(N_P_V_2,xx,xy,xz)
                # Induced velocity from the transition wake and from the semi-infinite helicoidal vortex (Second)

                U_xs,U_ys,U_zs = Biot_Savart_Propeller(Var.Z_Blade_P,x_1,y_1,z_1,x_2,y_2,z_2,xx,xy,xz)
                # Induced velocity from the bound vortex selected

                V_Tral_P[j,i,k,0] = V_Tral_P[j,i,k,0] - U_x2 + U_xs  # X velocity induced from the horseshoe vortex
                V_Tral_P[j,i,k,1] = V_Tral_P[j,i,k,1] - U_y2 + U_ys  # Y velocity induced from the horseshoe vortex
                V_Tral_P[j,i,k,2] = V_Tral_P[j,i,k,2] - U_z2 + U_zs  # Z velocity induced from the horseshoe vortex


                V_Tral_P[j_2,i,k,0] = U_x2     # I need this value for the next i loop (U_x2 will be U_x1 for the next horseshoe
                    vortex)
                V_Tral_P[j_2,i,k,1] = U_y2     # I need this value for the next i loop (U_y2 will be U_z1 for the next horseshoe
                    vortex)
                V_Tral_P[j_2,i,k,2] = U_z2     # I need this value for the next i loop(U_y2 will be U_z1 for the next horseshoe
                    vortex)

            x_1 = x_2          # This is used in order to have the first point of the next bound vortex (x)
            y_1 = y_2          # This is used in order to have the first point of the next bound vortex (y)
            z_1 = z_2          # This is used in order to have the first point of the next bound vortex (z)

    # VELOCITY MATRIX

    for j in range(Var.Msp):
        for i in range(I_P_Points_P):
            for k in range (4):
                V_Ind_P[j,i,k,0] = V_Grid_P [j,i,k,0] + V_Tral_P[j,i,k,0]  # Total induced velocity without the onset flow (x)
                V_Ind_P[j,i,k,1] = V_Grid_P [j,i,k,1] + V_Tral_P[j,i,k,1]  # Total induced velocity without the onset flow (y)
                V_Ind_P[j,i,k,2] = V_Grid_P [j,i,k,2] + V_Tral_P[j,i,k,2]# Total induced velocity without the onset flow (y)

    # Open the file for Propeller_Velocity_Total_No_Onset
    with open("output/Propeller_Velocity_Total_No_Onset.txt", "w") as file:
        file.write("    Point          Spanwise       Ux              Uy               Uz\n")
        file.write("  (Panel)  (Side)\n")

        for i in range(I_P_Points_P):
            for k in range(4):
                for j in range(Var.Msp):
                    file.write(f"    {i:2d}    {k:4d}     {j:4d}    {V_Ind_P[j, i, k, 0]:13.9f}    {V_Ind_P[j, i, k, 1]:13.9f}
                        {V_Ind_P[j, i, k, 2]:13.9f}\n")

    # Open the file for Propeller_Velocity_Trailing_Vortices
    with open("output/Propeller_Velocity_Trailing_Vortices.txt", "w") as file:
        file.write("    Point          Spanwise       Ux              Uy               Uz\n")
        file.write("(Panel)    (Side)\n")

        for i in range(I_P_Points_P):
            for k in range(4):
                for j in range(Var.Msp):
                    file.write(f"    {i:2d}    {k:4d}     {j:4d}    {V_Tral_P[j, i, k,0]:13.9f}    {V_Tral_P[j, i, k, 1]:13.9f}
                        {V_Tral_P[j, i, k, 2]:13.9f}\n")

    return V_Ind_P, V_Tral_P


V_Ind_P, V_Tral_P = Velocity_Total_No_Onset_Propeller()
```

```
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine calculates the total velocities at the midpoints
of segments (Onset + Induced & Onset) for the reference blade of the propeller.
"""


from sources.Onset_Flow_Propeller_P import Onset_Flow_Propeller
from sources.Velocity_Total_No_Onset_Propeller_P import Velocity_Total_No_Onset_Propeller
import numpy as np
import sources.Variables as Var


def Velocity_Total_Propeller ():
    Gamma_TE_P = np.loadtxt("output/Propeller_Gamma_TE_P.txt")

    V_Ind_P, V_Tral_P = Velocity_Total_No_Onset_Propeller()
    V_Onset_P = Onset_Flow_Propeller()

    I_P_Points_P = (Var.Msp*Var.Nch)
    V_Tot_No_Onset_P = np.zeros((I_P_Points_P,4,3))
    V_Tot_P = np.zeros((I_P_Points_P,4,3))

    for i in range (I_P_Points_P):
        # This loop in used to select the panel where the point px,py,pz is located on the propeller

```

```python
        # This loop in used to select the side of the panel where the point px,py,pz is located on the propeller
        for k in range(4):

            u_x = V_Onset_P[i,k,0]    # Temporary variable used to store the onset flow (x)
            u_y = V_Onset_P[i,k,1]    # Temporary variable used to store the onset flow (y)
            u_z = V_Onset_P[i,k,2]    # Temporary variable used to store the onset flow (z)

            u_xx = 0        # Initialization of the variable used to store the induced velocity
                            # of the propeller without the onset flow (x)
            u_yy = 0        # Initialization of the variable used to store the induced velocity
                            # of the propeller without the onset flow (y)
            u_zz = 0        # Initialization of the variable used to store the induced velocity
                            # of the propeller without the onset flow (z)

            # This loop is used to calculate the total velocity
            # at the midpoint of the panel of the propeller (Spanwise)
            for j in range (Var.Msp):
                u_x = u_x + Gamma_TE_P[j] * V_Ind_P[j,i,k,0]
                # Onset Flow + induced velocity in the panel i side k of the propeller (x)
                u_y = u_y + Gamma_TE_P[j] * V_Ind_P[j,i,k,1]
                # Onset Flow + induced velocity in the panel i side k of the propeller (y)
                u_z = u_z + Gamma_TE_P[j] * V_Ind_P[j,i,k,2]
                # Onset Flow + induced velocity in the panel i side k of the propeller (z)

                # Induced velocity in the panel i side k of the propeller
                u_xx = u_xx + Gamma_TE_P[j] * V_Ind_P[j,i,k,0]
                u_yy = u_yy + Gamma_TE_P[j] * V_Ind_P[j,i,k,1]
                u_zz = u_zz + Gamma_TE_P[j] * V_Ind_P[j,i,k,2]

            # Induced velocity of the propeller without the onset flow
            V_Tot_No_Onset_P[i,k,0]  = u_xx
            V_Tot_No_Onset_P[i,k,1]  = u_yy
            V_Tot_No_Onset_P[i,k,2]  = u_zz

            # Total induced velocity from the propeller in the panel i side k of the propeller
            V_Tot_P[i,k,0] = u_x
            V_Tot_P[i,k,1] = u_y
            V_Tot_P[i,k,2] = u_z


    # Open the file for Propeller_Velocity_Total_No_Onset
    with open("output/Propeller_Velocity_Total_No_Onset_V.txt", "w") as file:
        file.write("  Point                Ux             Uy              Uz\n")
        file.write("(Panel)  (Side)\n")

        for i in range(I_P_Points_P):
            for k in range(4):
                file.write(f"  {i:2d}    {k:4d}    {V_Tot_No_Onset_P[i, k, 0]:13.9f}   {V_Tot_No_Onset_P[i, k, 1]:13.9f}
                     {V_Tot_No_Onset_P[i, k, 2]:13.9f}\n")

    # Open the file for Propeller_Velocity_Trailing_Vortices
    with open("output/Propeller_Velocity_Total.txt", "w") as file:
        file.write("  Point                Ux             Uy              Uz\n")
        file.write("(Panel)  (Side)\n")

        for i in range(I_P_Points_P):
            for k in range(4):
                file.write(f"  {i:2d}    {k:4d}    {V_Tot_P[i,k,0]:13.9f}   {V_Tot_P[i, k, 1]:13.9f}   {V_Tot_P[i, k, 2]:13.9f}\n")

    return (V_Tot_P, V_Tot_No_Onset_P)


V_Tot_P, V_Tot_No_Onset_P = Velocity_Total_Propeller()
```

```python
"""
Date: Q4 2023 - Q1 2024
Author: Lisa Martinez
Institution: Technical University of Madrid

Description: This subroutine computes the weight function for the propeller,
involving the declaration of variables and arrays.
"""


import numpy as np
import sources.Variables as Var


def Weight_function_propeller():


    t_gp_P = np.loadtxt("output/Propeller_t_gp.txt", skiprows=1)
    # Rooftop parameter a
    a_roof = 0.8
    # 2 / Pi
    pi_inv = 2/np.pi
    # Domain limit of the distribution of circulation (rooftop)
    t_rest = 0.5 - a_roof
    # First Denominator
    t_slop = 2/(1 - a_roof*a_roof)
    # Second Denominator
    pcst = 2/(1 + a_roof)
    # Rooftop Coefficient
    cny1 = 1 - Var.cny

    GF_tot = [0.0]                        # Weight equation´s numerator
    GF_tmp = np.array([0.0]*(Var.Nch))    # Temporary Weight Function 2
    GW = np.array([0.0]*(Var.Nch))        # Temporary Weight Funtion 2
    G_Faux =np.array([0.0]*(Var.Nch))
```

```
    Weight_P = np.zeros((Var.Msp,Var.Nch))

    #Weight equation´s numerator
    for i in range(Var.Nch+1):
        i_1 = i-1
        t_gp_P_1 = 0.5 + t_gp_P[i]              # Numerator - Flat plate distribution (gamma)
        t_gp_P_2 = 0.5 - t_gp_P[i]              # Denominator - Flat plate distribution (gamma)
        Gamma_fp =pi_inv * np.sqrt(t_gp_P_1/t_gp_P_2) * Var.cny  # Flat plate distribution (gamma)
        Gamma_rt = pcst * cny1                                   # Rooftop distribution (gamma)

        if t_gp_P[i]<t_rest:                                     # Rooftop distribution (gamma)
            Gamma_rt = t_slop * t_gp_P_1 * cny1

        Gamma_Tot = Var.cny*Gamma_fp + cny1*Gamma_rt # This is the combination of flat plate distribution
                                                     # and rooftop distribution (gamma)

        GF_tmp[i_1] = Gamma_Tot*np.sqrt(t_gp_P_1*t_gp_P_2)
        GF_tot = GF_tot + GF_tmp[i_1]                # Weight equation´s denominator

    # Loop for the weight function (circulation of the panels)
    for i in range(Var.Nch-1,-1,-1):
        G_Faux += GF_tmp[i]/GF_tot
        GW[i] = G_Faux[i]
    #Loop used to write the weight function for all the panels# (it is the same for each spanwise level)
    for j in range (Var.Msp):
        for i in range (Var.Nch):
            Weight_P[j,i] = GW[i]   # Weight Function - j = spanwise level - i = chordwise level - Propeller

    with open("output/Propeller_Weight_Function.txt", "w") as file:
        file.write("Panel  Weight Function\n")
        for i in range(Var.Nch):
            file.write(f"{i:3d}    {Weight_P[0,i]:13.9f}\n")
    return(Weight_P)


Weight_P = Weight_function_propeller()
```