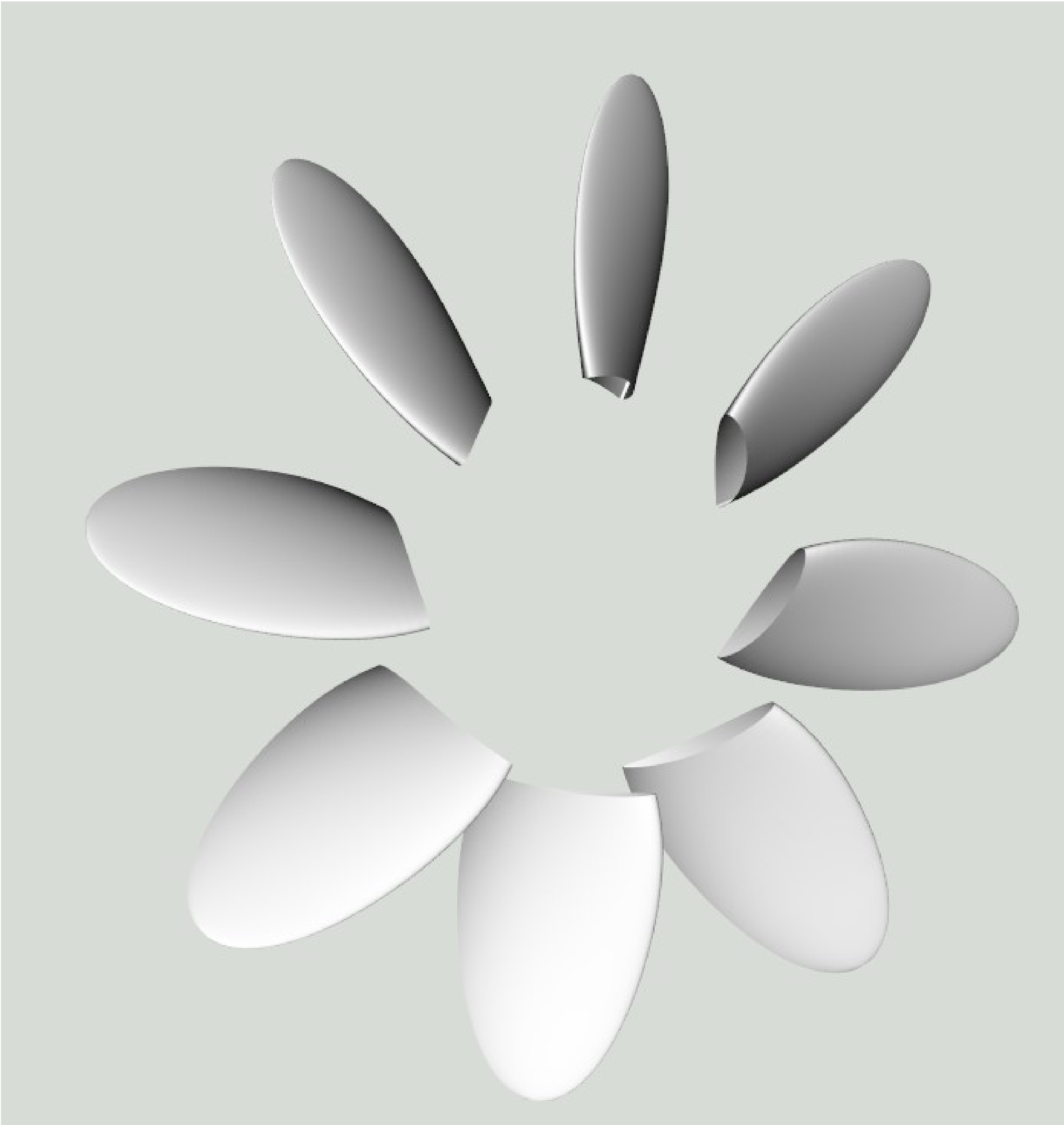
CHALMERS
UNIVERSITY OF TECHNOLOGY



A Chalmers University of Technology Conceptional Design Method for Propellers

An implementation of Drela method for minimum induced loss and Garrick and Watkins method for sound pressure level

Master’s thesis in Sustainable energy systems

GEORGE HELALEH

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Conceptional Design Method for propellers

An implementation of Drela method for minimum induced loss and Garrick and Watkins method for sound pressure level

GEORGE HELALEH

Department of Mechanics and Maritime Sciences
Division of Fluid Dynamics
Chalmers University of Technology
Gothenburg, Sweden 2025

Conceptional Design Method for propellers
An implementation of Drela method for minimum induced loss and Garrick and Watkins method for Sound pressure level
GEORGE HELALEH

Supervisor: Tomas Grönstedt, Department of Mechanics and Maritime Sciences.
Examiner: Tomas Grönstedt, Department of Mechanics and Maritime Sciences.

Master's Thesis 2025
The applied mechanics department
The fluid dynamics division
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Conceptional Design Method for propellers
An implementation of Drela method for minimum induced loss and Garrick and Watkins method for sound pressure level
GEORGE HELALEH
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Abstract

The ongoing development of electric aircraft aims to make the aviation industry more sustainable, with a focus on reducing energy consumption and noise emissions. Propellers, as a key component of electric airplanes, play a critical role in achieving these goals. This study focuses on designing optimized propellers for electric aircraft that maximize efficiency while minimizing energy loss and sound pressure levels (SPL). By employing Drela's propeller design methodology for aerodynamic optimization and Garrick and Watkins models for noise analysis.

The primary tool used in this study is OptoProp, a numerical simulation program developed in Python to optimize propeller efficiency. The methodology involves several key steps. First, the program is verified using a documented case to ensure its accuracy in predicting propeller performance. Following verification, a parameter sensitivity study is conducted to understand how critical design elements, such as blade count, diameter, and angular speed, influence efficiency and noise levels. The final step involves conducting noise simulations using the calculated aerodynamic data to analyze the relationship between propeller parameters and SPL, providing deeper insight into the effects of each parameter.

The results of the parameter study reveal several important trends. The propeller's diameter and blade count are pivotal in determining its overall efficiency. A larger propeller size enhances efficiency by reducing the power required and lowering the generated torque, both of which are critical for achieving sustainable performance. However, increasing the diameter also leads to a positive effect on SPL, as larger propellers tend to produce less noise due to their inverse relationship with the sound pressure level. While this trend offers a valuable approach to designing propellers with better performance and reduced noise levels, there is a threshold for both the diameter and blade count. Beyond this threshold, further increases in these parameters have detrimental effects on performance and can lead to suboptimal propeller designs. This suggests that simply increasing the size or number of blades does not always result in improved performance, and a balanced design is essential. On the other hand, the effect of angular speed on propeller performance is also significant. Increasing the angular speed improves efficiency up to a certain optimal value, after which further increases yield diminishing returns. However, angular speed has a negative impact on SPL, meaning that higher speeds generate more noise. This introduces a critical trade-off between performance and noise, as designers must be cautious not to increase the angular speed beyond the optimal level if low noise levels are a priority. The sensitivity of noise to angular speed makes it an important factor in designing propellers for electric aircraft, where minimizing noise pollution is a key concern.

In conclusion, the study provides valuable guidelines for designing electric aircraft propellers that balance high aerodynamic performance with low noise emissions. The findings suggest that an optimal propeller design must consider not only aerodynamic efficiency but also the impact of various parameters on SPL. The balance between propeller size, blade count, and angular speed is crucial for achieving sustainable performance, with thresholds for each parameter that must be respected to avoid diminishing returns in both efficiency and noise reduction. These insights contribute to the development of electric aircraft technology and offer a pathway toward more sustainable aviation solutions, where both environmental impact and noise pollution are minimized without compromising on performance.

Keywords: aerodynamics, aeroacoustic, Drela, Minimum induced loss, sound pressure level, Garrick and Watkins, noise

Acknowledgements

First and foremost, I would like to thank my supervisor, **Dr. Tomas Grönstedt**, unwavering support throughout this research. Their expertise and patience have been instrumental in shaping this work.
I am also grateful to **Alexandre Capitao Patrao** for his invaluable help with `python` and encouragement during this research. His input has greatly enriched the quality of this thesis.
I extend my sincere thanks to the faculty of the **Department of Mechanics and Maritime Sciences in chalmers university of technology** for providing me with the resources and opportunities necessary to conduct this research.
This thesis is dedicated to all those who have contributed to its completion, directly or indirectly. Thank you for being a part of this journey.

George Helaleh, Gothenburg, February 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AMT	Axial Momentum Theory
BET	Blade Element Theory
GHG	Green House Gases
GMT	General Momentum Theory
ICAO	International Civil Aviation Organization
IATA	International Air Transport Association
IEA	International Energy Agency
Optoprop	Optimum propeller
SPL	Sound Pressure Level

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices

Symbol	Description	Unit
$()_a$	Index for axial component	–
$()_t$	Index for tangential component	–

Parameters

Symbol	Description	Unit
B	Number of blades	–
Q	Torque	$\mathrm{N} \cdot \mathrm{m}$
$Q(r)$	Local torque	$\mathrm{N} \cdot \mathrm{m/m}$
R	Tip radius	m
T	Thrust	N
$T(r)$	Local thrust	N/m
V	Free stream velocity	m/s
$W(r)$	Local total velocity relative to blade	m/s
c	Fluid speed of sound	m/s
$cd(r)$	Local blade drag coefficient	–
$cl(r)$	Local blade lift coefficient	–
r	Radial coordinate	m
$\Gamma(r)$	Local blade circulation	$\mathrm{m^2/s}$
Ω	Rotation rate	rad/s
$\beta(r)$	Local geometric blade pitch angle	rad

η	Overall efficiency	–
λ	Advance ratio	–
$\lambda_w(r)$	Local wake advance ratio $(r/R)(W_a/W_t)$	–
μ	Fluid viscosity	$\mathrm{Pa \cdot s}$
$\phi(r)$	Local flow angle	rad
ρ	Fluid density	$\mathrm{kg/m^3}$
$u(r)$	Local externally-induced velocity at disk	m/s
$v(r)$	Local rotor-induced velocity at disk	m/s
$\alpha(r)$	Local angle of attack	rad

Nomenclature of acoustic study

Symbol	**Description**	**Unit**
$A(r)$	chordwise distribution of thrust acting on a radial element of a propeller blade	N/m
$b(r)$	width of propeller blade	m
c	velocity of sound	m/s
$F(r)$	chordwise distribution of forces perpendicular to the thrust of a propeller blade and giving rise to the torque	N.m/m
F_x, F_y, F_z	components of force vector $\bar{F}$	N
J_{mB}	Bessel function of first kind with index mB	
$k = \omega/c$	wave number	m^{-1}
M	Mach number V/c	
m	order of harmonic	-
n	propeller rotational speed	rps
p_Q	pressure due to torque	N/m^2
p_T	pressure due to thrust	N/m^2
p_{rms}	root mean square pressure	N/m^2
R_e	effective diameter	m
r, θ	polar coordinates in yz-plane	rad
S	$\sqrt{(x-x_l)^2 + \beta^2[(y-y_l)^2 + (z-z_l)^2]}$	m
S_o	$\sqrt{x^2 + \beta^2 y^2}$	m
s	$\sqrt{(x-x_l)^2 + (y-y_l)^2 + (z-z_l)^2}$	m
s_o	$\sqrt{x^2 + y^2}$	m

t	time	s
β	blade angle	rad
σ	$\frac{M(x-x_l)+S}{\beta^2}$	m
τ	$\frac{b}{r\Omega}$	s
τ_o	period $\frac{2\pi}{B\Omega}$	s
Ω	angular velocity	rad/s
ω	frequency of m harmonic, $mB\Omega$	Hz
ω_1	fundamental frequency, $B\Omega$	Hz

Contents

List of Figures

List of Tables

1

Introduction

Over the past century, the aviation industry has undergone numerous transformations. During the First and Second World Wars, there was a need to produce high-speed, low-noise aircraft to ensure safe returns [14]. In the middle of the last century, as transportation became more profitable, which in turn drove an increase in fuel consumption causing a notable rise in pollutant emissions. More recently, there has been an increased focus on reducing the environmental impact of the aviation sector and reducing the costs on the fuel. According to Our World In Data (OWID) website [2], aviation contributes 2.5% of global CO_2 emissions and 4% of global warming to date. Moreover, some reports indicate that if the situation continues unchanged, emissions could increase to 4-5% , as noted by Kreimeier[3], or even triple by 2050, as warned by the International Civil Aviation Organization (ICAO) [4].

A lot of global concerted efforts have been made to reduce greenhouse gas emissions from the aviation sector, in line with the Paris Agreement [5], which involves commitments from countries to reduce greenhouse gas emissions and enhance climate resilience through nationally determined contributions and international cooperation.

The agreement addressed two strategies to achieve the goal of reducing carbon pollution in the aviation sector: mitigation, which involves cutting emissions by using new clean technologies or improving the efficiency of existing technologies, and adaptation, which focuses on handling the impacts associated with climate change by implementing policies and logistics for infrastructure. As an example of adaptation measures in aviation infrastructure, the U.S. Federal Aviation Administration (FAA) detailed several steps for adopting a new framework to enhance airport performance in the United States of America [6].

The aviation sector has rapidly advanced in its mitigation strategies, with many innovative solutions and improvements finding their way into the industry. For example, using fuel derived from renewable resources, known as Sustainable Aviation Fuels (SAF), is helping to reduce the carbon footprint of flights [7]. Another approach in mitigation solutions is to use hydrogen or batteries for short flights, to avoid the need for fossil fuels on these routes [8]. Using lightweight composite materials reduces the total weight of the plane, which in turn decreases the need for power derived from fossil fuels [9]. Optimizing the design of propellers enhances their efficiency, which means a reduction in the amount of fossil fuel needed [10].

1.1 Efficiency and Environmental Impact

One of the promising approaches to improving environmental performance in the aviation sector is to enhance propeller design, thereby increasing aerodynamic efficiency and reducing the required power.
This improvement has undergone many development processes, starting in 1919 [14] with the Axial Momentum Theory (AMT) developed by Rankine and Froude, which assumes an axial flow passing through the propeller. This theory was extended by the German physicist Albert Betz to the General Momentum Theory (GMT) by including rotational motion in the flow analysis. Later, Glauert applied the Blade Element Theory (BET) to analyze the forces on the airfoil. Ludwig Prandtl then introduced the concept of vorticity and developed the Vortex Theory to optimize propeller design.
Many modern models are built on these conventional theories by using numerical methods such as those developed by Larrabee, Adkins, and Drela. These models play a vital role in enhancing the efficiency of propellers today, reducing their environmental impact and minimizing their carbon footprint.

1.2 Noise Pollution and Its Environmental Impact

Noise is one of the most significant negative effects on human health and is considered the second most harmful environmental factor after pollution, according to the European Commission website [12]. Constant exposure to noise not only causes discomfort but can also lead to hearing loss, sleep disturbances, and even cardiovascular disease. According to the European Commission, 12,000 premature deaths occur, 22 million Europeans experience prolonged discomfort, and 12,500 students suffer from learning disabilities due to environmental noise [12].
The World Health Organization - Regional Office for Europe has established guidelines for environmental noise [11]. The guidelines strongly recommend keeping noise levels below 45 dB during the day and below 40 dB at night to mitigate the negative effects of environmental noise on human health. Additionally, the organization encourages decision-makers to implement policies aimed at reducing environmental noise [11]. On the other hand, from a technological perspective, advanced propeller design can significantly enhance efforts to reduce environmental noise. Beginning with Lynam in 1919 and continuing through the work of many scientists, the sources of propeller noise are now well understood [34], this will reduce environmental noise by optimizing the design.

1.3 OPTOPROP

Optoprop is a code developed at Chalmers University of Technology (in-house) to facilitate the propeller design process, especially for open-rotor propellers [18].
The code is implemented primarily in MATLAB and partially in Python. The pur-

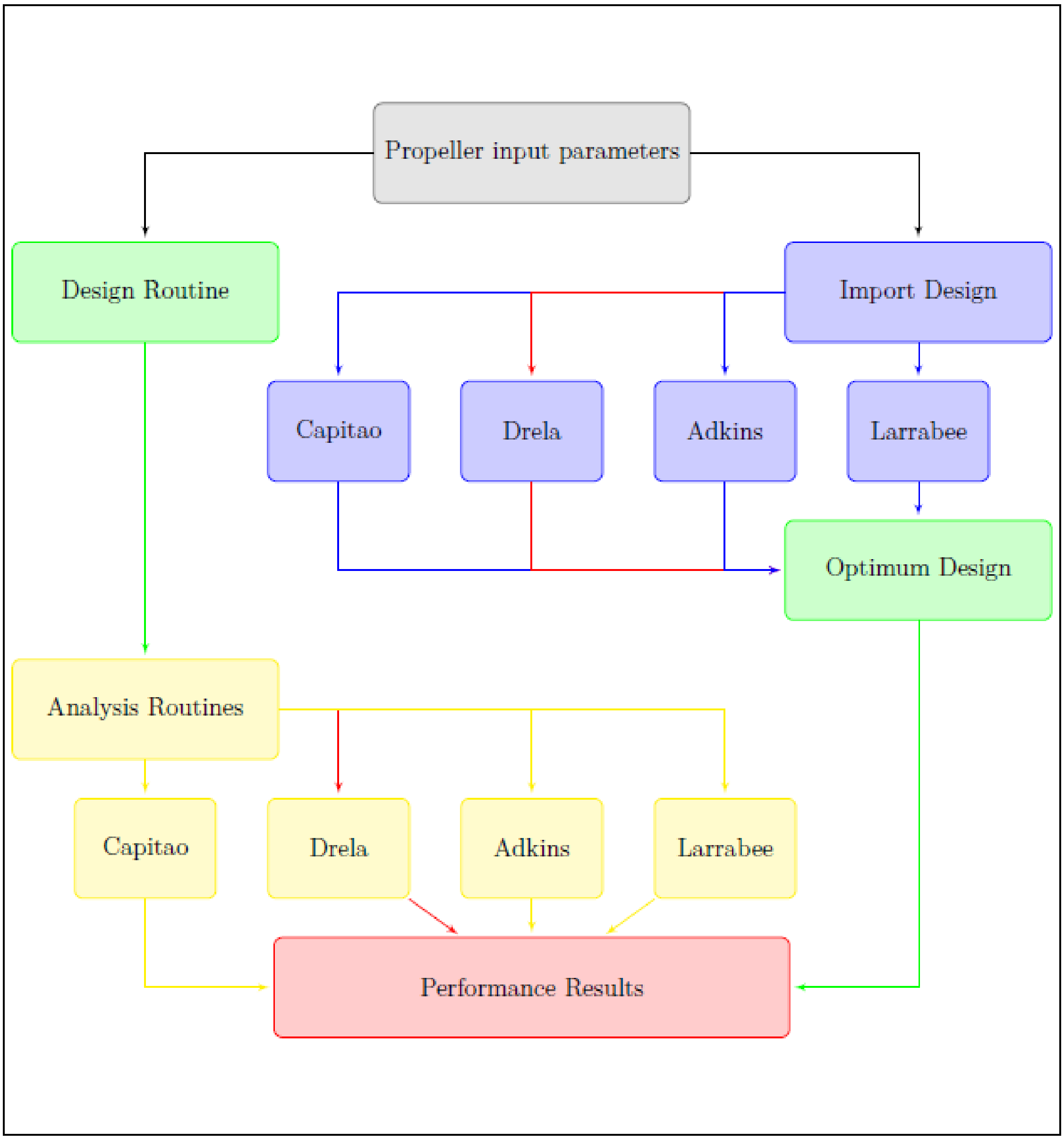


Figure 1.1: Optoprop code by Python.

pose of Optoprop is to advance the research and development of propeller design and analysis by utilizing various methods. This approach aims to enhance aerodynamic efficiency through the minimization of induced losses, ultimately leading to a reduction in the environmental impact of propellers.

1.4 Purpose and Contribution of the Thesis

This work aims to extend the Optoprop code, which was originally implemented in Python, to be compatible with the existing MATLAB code. Specifically, the Drela model of minimum induced loss for propellers will be integrated, along with additional types of data, such as other propeller types included in the MATLAB version 1.1. Additionally, a new model based on the Garrick and Watkins method will be incorporated to analyze the Sound Pressure Level (SPL) generated by the thrust and torque of the propeller.

2

Theory

2.1 Propeller Terminology

In order to ensure clarity and precision in the following sections, it is essential to establish a common understanding of key terms and concepts employed throughout this thesis. The main function of the propeller is to generate thrust and stabilize drag forces to facilitate the flying process of an aircraft. Thrust is the force that propels the aircraft in the axial direction, either forward or backward, and results from the propeller adding momentum to the surrounding airflow [14].

Thrust is affected by the forces applied to the blade of the propeller, which are the lift and drag forces. On one hand, the lift force is useful for overcoming gravitational force and is created by the pressure differences between the upper and lower surfaces of the propeller's cross-sectional shape. On the other hand, the drag force negatively impacts aircraft motion and is in the same direction as the airflow [14]. Fig. 2.1 illustrates the nomenclature of the airfoil [26].

- Leading edge: The section of the airfoil that is in contact with the airflow first.
- Trailing edge: The section of the airfoil that is in contact with the airflow last.
- Chord: The distance between the leading and trailing edges.
- Chord-line: A straight line joining the leading and trailing edges.
- Camber-line: A line joining the leading and trailing edges and equidistant from the top and bottom surfaces.
- Camber: The distance from the chord line to the camber line.
- Thickness: The distance from the top surface to the bottom surface.
- Torque: Also called a twisting force, it is parallel to the airflow.
- Angle of attack: The angle between the airflow and the chord line. The angle of attack is an important factor in increasing lift force.
- Stall: A condition where lift decreases and drag increases, causing the airflow to separate from the upper surface.
- Stall angle: The angle of attack at which the lift coefficient reaches its maximum value.
- Minimum induced loss situation: Propellers with minimal induced loss are the rotor counterparts to elliptically loaded wings. They are distinguished by the least kinetic energy loss within defined parameters of disc loading, the number of blades, and flight speed [17].
- Low disc loading:

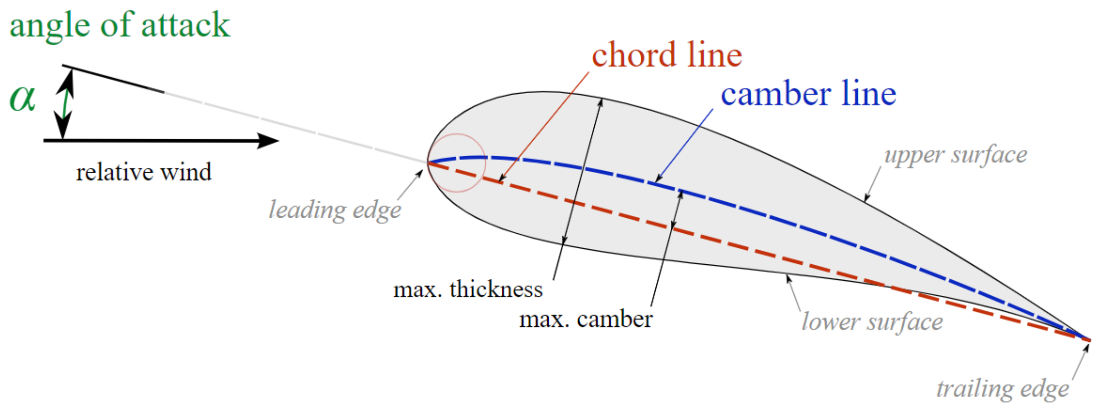


Figure 2.1: Airfoil diagram [15]

2.1.1 Velocities on airfoil

The velocity vector is expressed as the vector sum of three constituent vectors, each contributing to its overall magnitude and direction. The first velocity is the axial velocity, or aircraft velocity V_0; the second is the rotational velocity generated by the engine driving the propeller; and the last is the induced velocity of the environment, which includes the effects of other parts of the aircraft, such as the wings, on the propeller and the wake region. This is represented in Fig. 2.2 by W.

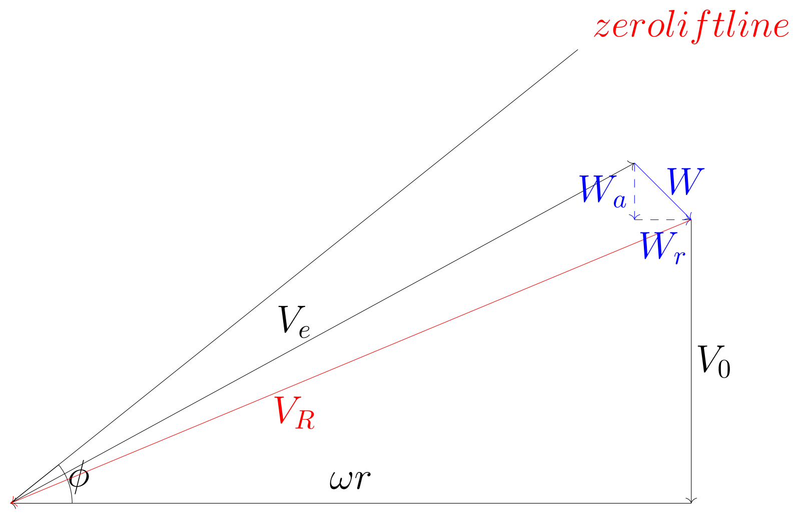


Figure 2.2: Illustration of velocities of airfoil

As depicted in Fig. 2.2, the velocity vector V_R changes instantaneously in both magnitude and direction. This variation occurs due to asymmetric incoming flow, which in turn changes because of variations in the angle of attack or, in some cases, the wake.

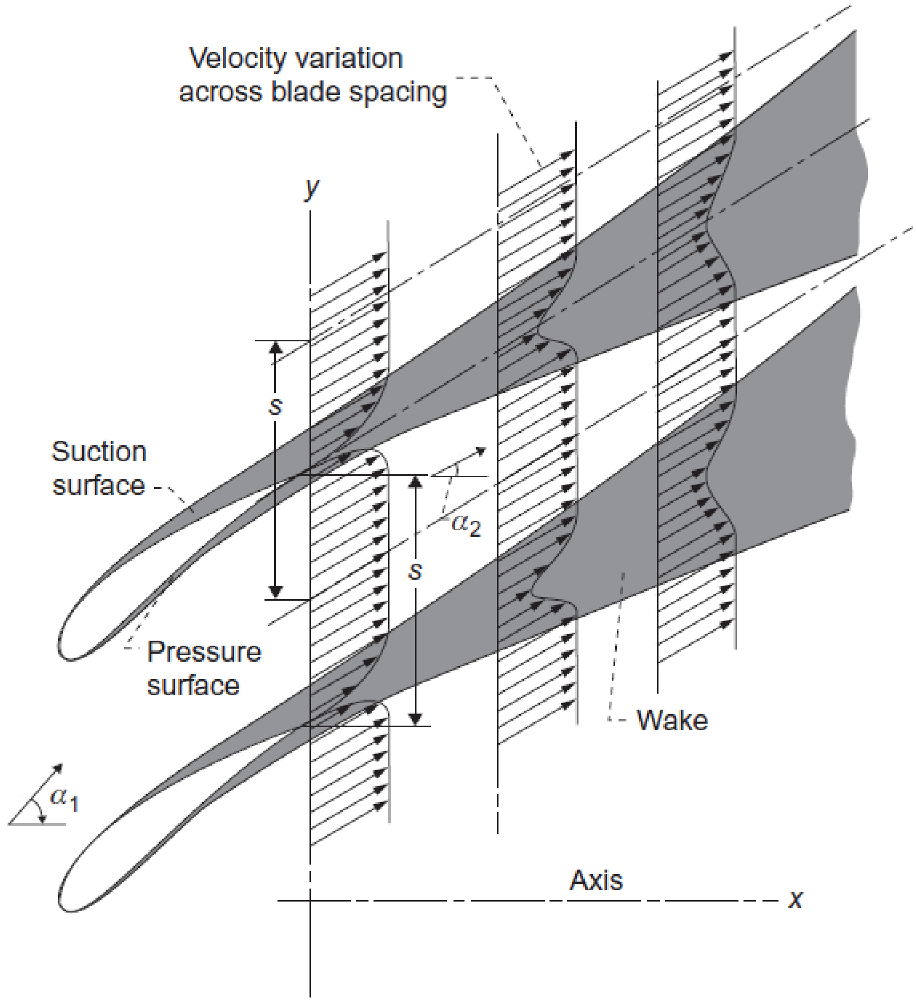


Figure 2.3: The flow through a blade cascade and the formation of the wakes [26]

2.1.2 Forces applied on airfoil:

The forces acting on the airfoil are of two types. The first is the pressure force, which arises due to the shape of the airfoil. The pressure force divides the airfoil into two surfaces: a high-pressure surface and a low-pressure surface. The difference in pressure between these surfaces leads to a resultant force.
The second force affecting the airfoil is friction. Friction results from viscous effects observed in the boundary layer. The effects on the boundary layer will be discussed in detail later.
As seen in Fig. 2.3, and by applying the momentum equation in the x and y directions, the equations can be written as follows

$$X = (p_h - p_l)S$$
$$Y = \rho S V^2 (vy_1 - vy_2)$$

Here $(_1)$ and $(_2)$ are index of the direction, not new variables. As previously stated, the lift force L is applied to the airfoil perpendicular to the chord line and toward

the low-pressure surface, as illustrated in Fig. 2.4. In contrast, the drag force D is applied to the airfoil parallel to the chord line and toward the trailing edge.
On the other hand, these forces can be represented in Cartesian coordinates (vertically and horizontally). To facilitate the equation, α_m is defined as the average angle of attack of the in/out flows:

$$\tan \alpha_m = \frac{1}{2}(\tan \alpha_1 + \tan \alpha_2).$$

$$T = L\cos(\alpha_m) - D\sin(\alpha_m) \tag{2.1}$$
$$Q = D\cos(\alpha_m) + L\sin(\alpha_m) \tag{2.2}$$

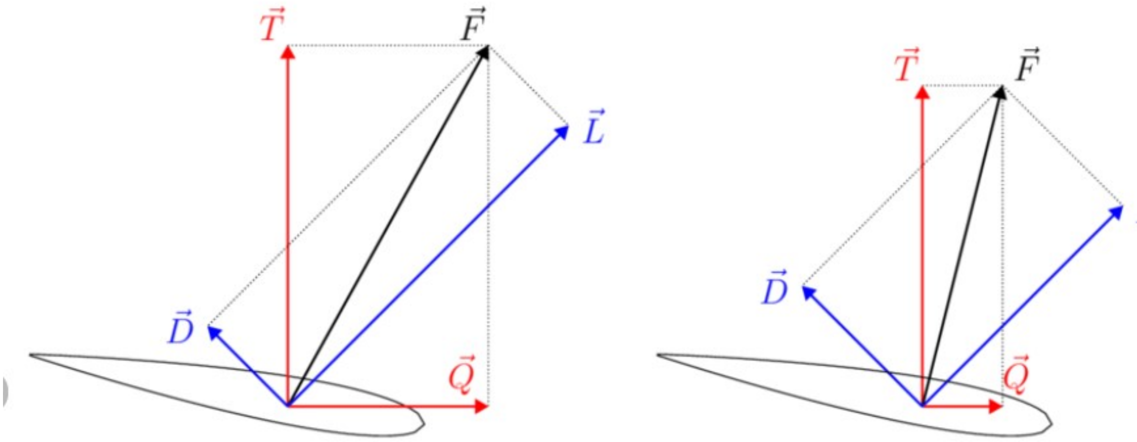


Figure 2.4: Schematic diagram of flow past blade element of height dr showing the forces acting and the incoming velocity relative to the blade element in a wind turbine, the direction of the force Q should be in the opposite for a propeller [27].

Following this, the definitions of lift and drag coefficients can be written as follows:

$$C_L = \frac{L}{\frac{1}{2}\rho V A}$$
$$C_D = \frac{D}{\frac{1}{2}\rho V A}$$

2.1.3 Propeller wake

As depicted in Fig 2.3, the flow downstream of the blade forms a wake region due to friction between the fluid and the blade. The integration of the wake at the trailing edge of the blade forms what is known as the slipstream of the propeller. The slipstream has a helical shape due to the constant axial and radial velocities, while the tangential velocity accelerates suddenly. The shape of the vortices in the wake plays a vital role in determining the circulation in the wake region and consequently the design of the propeller. It is important to define the concept of circulation in this section to ensure clarity for the later sections. Circulation is the integration of the fluid velocity around a closed curve: $\Gamma = \oint_S \mathbf{v} \cdot d\mathbf{S}$. This definition was first given by Zhukovsky [30] and later improved by Kutta through the so-called Kutta-Zhukovsky equation. This equation relates circulation to lift force and states that the circulation of a fluid with density ρ and relative velocity v between the airfoil and the fluid at infinity is equal to the lift force L.

$$L = \rho v \Gamma \tag{2.3}$$

The last equation is a fundamental concept in vortex theory, which is essential for designing optimal propellers.

2.1.4 Viscous Effects in the Boundary Layer on the Blade

The velocity vector is variable, and its magnitude changes between zero and the previous value V_R shown in Fig. 2.2. The difference, in this case, is due to the impact of the friction force, which causes particles closer to the blade to move with it. These particles will experience centrifugal acceleration similar to the blade.
For this reason, a radial component will be added to the velocity, resulting in the formation of radial flow in both separated flow regions and laminar separation bubbles. Additionally, there is a Coriolis force that affects particles moving in the radial direction, which will alter the shape of the propeller.
Therefore, there will be differences between the lift and drag coefficients in 2D and 3D calculations.

2.2 Review of the conventional theories to design propellers

Throughout the last century, the evolution of propeller design has been significantly influenced by the development of three pivotal theories. The first theory, General Momentum Theory (GMT), introduced in 1900, described how the flow moves around the propeller and the forces resulting from these effects. Following closely, the second theory, Blade Element Theory (BET), introduced in 1925, expanded upon the initial concepts, such as how the shape of the airfoils affects the flow of fluid particles and the resulting aerodynamic forces. As the field continued to grow, a third theory, Vortex Theory, emerged in the 1920s. The vortex theory accounts for the shape of the flow in the wake region and explains how this shape influences the design. These pioneering theories have been the foundation upon which modern propeller design theories are constructed.

2.2.1 Momentum Theories

Momentum theories stand as a pivot in the field of propeller design. They describe the induced flow velocities in both axial and radial directions. This is achieved by defining the axial and tangential induced factors (a, a'). The results emerging from the use of momentum theories are not perfectly accurate, but they satisfy Betz's condition perfectly [26]. To apply the theory effectively, one must consider the underlying assumptions, including the fact that the effect of airfoil shape is ignored, and thus, the effect of the wake area is neglected.

2.2.1.1 Axial Momentum Theory-Rankine Froude

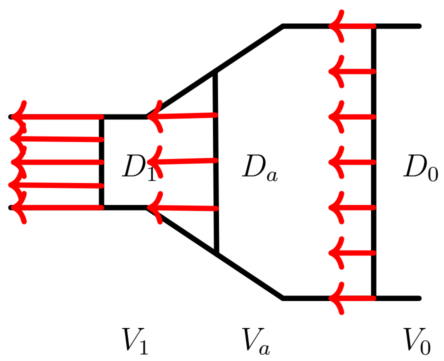


Figure 2.5: Axial Momentum Theory (AMT)

With the goal of enhancing the performance of marine propellers, Rankine and Froude presented their theory termed Axial Momentum Theory [14]. Axial Momentum Theory states that the upstream flow enters the propeller at section D_0 with a velocity of V_0, as illustrated in Fig. 2.5. The flow then crosses the disc of the propeller at D with a velocity of v_a. Finally, the downstream flow leaves the propeller at point D_1 with a velocity of v_1.
The difference in velocities occurs as a result of the jump in pressure at the actuator disk. The theory assumes that the velocity at the disk is the average value of the upstream and downstream velocities. Thus, a new factor called the *axial flow induction factor* $\bar{a}$ is defined, representing the relative change in velocity $\frac{(v_0-v)}{v_0} = 0.5$. The section area is represented as $s = 2\pi r$ since the propeller is a turbomachine. Thus, thrust (T), power (P), and efficiency (η) are calculated using the following equations:

$$T = \int_s \Delta p \, dx = A_a \, \rho \, v_a(v_1 - v_0)(1 + a) \tag{2.4}$$

$$P = \frac{1}{2} A_a \, \rho \, v_a({v_1}^2 - {v_0}^2)(1 + a) \tag{2.5}$$

$$\eta = \frac{v_0 \, T}{P} = \frac{1}{1 + (\frac{v_1 - v_0}{2v_0})} \tag{2.6}$$

Among the prominent weaknesses of this theory are the following assumptions:

1. The flow moves only in the axial direction (no radial movement).
2. The flow is incompressible, which aligns with the fundamental principles of the theory, primarily formulated for marine propellers.
3. Viscosity is not included (inviscid flow).

4. The flow moves uniformly along the slipstream tube to facilitate calculations.
5. Thrust is evenly distributed across the disc.

2.2.1.2 General Momentum Theory Of Betz

In this theory, the rotational motion of the flow must be included in the calculation, which may affect the value of pressure [26].

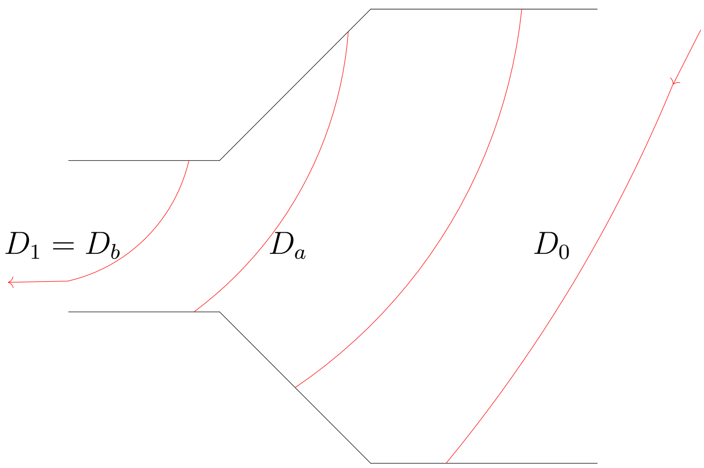


Figure 2.6: General momentum theory

In this theory, the nomenclature of the axial induction factor (a) is the same as in AMT, representing the changes in axial velocity due to the influence of the rotor or blades, i.e., $V_a = V_0(1 + a)$. Meanwhile, the nomenclature of the radial induction factor (a') represents the reduction in tangential velocity due to the influence of the rotor or blades and accounts for the circumferential redistribution of angular momentum, i.e., $V_{a,\mathrm{rad}} = V_{0,\mathrm{rad}}(1 + a')$.
The far upstream flow moves toward the disc with a velocity V_0. The velocity then increases to $V(1+a)$, where a is the axial interference factor. According to the conservation of mass, the relation for the flow particle is given by $dm = 2\pi r \rho V(1+a)$. The flow moves downstream to the trailing edge where the wake forms, and the velocity there reaches $V(1 + b)$, where b is the axial slipstream factor. In this theory, the b factor is approximately equal to $2a$. Ignoring radial flow, b is exactly equal to $2a$, as confirmed in the previous section describing the axial momentum theory. Thrust, torque, power, and efficiency can now be expressed by the following equations:

$$T = 2\pi r \rho V_0(1 + a)(2V_0 aF)$$
$$Q' = 2\pi r \rho V_0(1 + a)(2\Omega r a' F)$$

Among the prominent weaknesses of this theory is the fact that it does not consider the shape of the propeller.

2.2.2 The Blade Element Theory

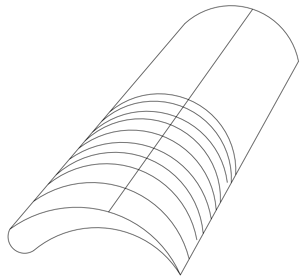


Figure 2.7: Blade element theory

The body of the propeller is divided into several segments (elements). Each segment is treated separately by applying aerodynamic forces to it as a 2D body and then summing the results at the end as seen in Fig. 2.7. In other words, at each element, the local velocity has unique components of tangential and axial velocities, which leads to a specific angle of attack at each element (local angle of attack AOA_r). The Blade Element Theory (BET) requires the lift and drag coefficients to be known [24]. As observed, this theory focuses on the shape of the propeller itself and neglects the effect of the surrounding environment. Thus, the effects of viscosity, stall, and compressibility are combined, which are referred to as Quasi-3D effects.

Furthermore, the problem with this theory is that it does not account for induced properties. The tendency of the fluid to rotate around itself, known as vorticity in the wake region, is neglected, so the results of this theory are not accurate.

2.2.3 Vortex Theory

This method presumes that the shape of the wake region is known. Accordingly, calculations of vorticity in the slipstream are possible, and the induced velocity can be determined as a result. Propellers with minimal induced loss are characterized by the least kinetic energy loss within defined parameters of disc loading, the number of blades, and the speed of flight. In other words, propellers with minimal induced loss represent rotor counterparts to elliptically loaded blades. This assumption can be right if the circulation distribution in the far wake is optimal, wherefore, the kinetic energy losses will diminish to the minimum magnitude. This case names minimum induced loss case [17].

In this thesis, three different methods will be discussed. The first one is established by Betz, Glauert, and Prandtl, while the second one is done by Goldstein, and the last one is founded by Theodorsen [14].

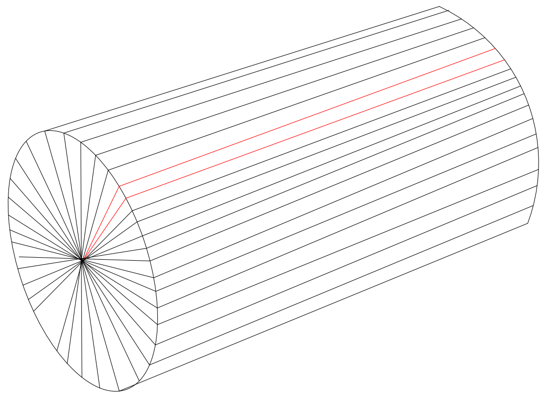


Figure 2.8: The vorticity forms a rigid cylindrical shape due to the infinite number of blades; the red lines represent a vortex element. The propeller is located in the center of the circle on the left side.

2.2.3.1 The theory of Betz, Glauert, and Prandtl

In this theory, there are two assumption are made, the first is having a constant circulation along the blade and the second one is to have an infinite number of blades,according to this theory, the vorticity that stems from the trailing edge of each propeller has a constant magnitude at each local position and moves downstream [14]. The combination of the all vorticity elements form a rigid cylindrical shape due to the assumption made of this theory to have an infinite number of blades and the neglect of the impact of the slipstream on it.
In other words, there are no interactions between the slipstream and the formed cylinder; therefore, the radial component of the slipstream will vanish.
Ignoring the radial component of the slipstream could be possible if the value of thrust or power per unit disc area is small; this is what one calls a light-load propeller blade [13]. The presuppositions of this theory:

1. The propeller has an infinite number of blades.
2. The circulation is constant along the blade span.
3. The vorticity generated by the trailing edge is represented as a rigid cylinder.
4. There is no interaction between the flow outside the cylinder and the flow inside.
5. The theory does not account for periodic effects because the shape of the wake region is represented as a rigid cylinder with no fluctuations.

According to the previous assumptions, the validity of this theory is limited to light-loaded propellers, where the contraction of the slipstream is significant and the rigid cylinder is ignored. The importance of this theory stems from its ability to calculate the induced velocities at the disc by knowing the velocity distribution in the wake region. The axial-induced velocity in the wake is double the axial-induced velocity at the disc. Two correction factors are presented in this theory: the first one concerns the profile of drag. Using the assumption of constant circulation distribution diminishes the thrust while increasing the torque, which results in a reduction of efficiency. The second correction factor addresses the assumption of a large number of blades (Prandtl factor) [31]. Prandtl paid attention to the tip losses, which were defined as the leakage of air near the edge of the propeller that causes the existence of a radial component in the slipstream and a perturbation (or reduction) in the constant circulation.

2.2.3.2 The Theory of Goldstein

This theory states that each blade has its own helical vortex, generated by its trailing edge. The circulation is not constant but varies in the radial direction. Furthermore, there is no interaction between vortex sheets and slipstream tubes. Thus, one can consider that the slipstream has a periodic nature.
The vortex surface of one blade is depicted in polar coordinates by aligning the helix axis with the reference axes r, θ, and z using the following equation:

$$\theta - \frac{\omega z}{V_\infty} = 0 \quad \text{or} \quad \pi \quad \text{for} \quad r < R \tag{2.7}$$

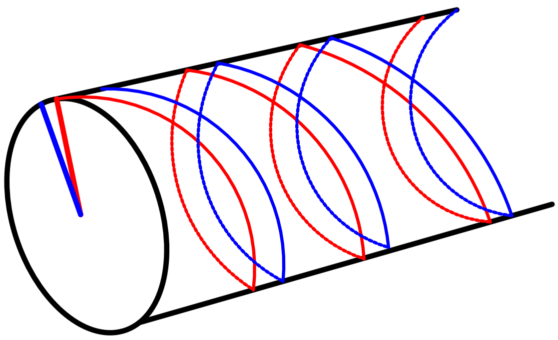


Figure 2.9: Vortex according to Goldstein. Each blade is aligned to a vortex.

Initially, Goldstein set the Laplace equation for the flow [32], $\nabla^2 v = 0$. Subsequently, he derived the velocity potential function for it. The potential field, as he suggested, is located far downstream and is applied under specific boundary conditions. Finally, he found that the solution to this function can be represented as a semi-infinite series of modified Bessel functions (Watson's Bessel functions).
In the validation process of this theory, the results showed that the distribution of circulation reached its minimum value when the fraction $\frac{V}{nD}$ (the so-called advanced ratio) had small magnitudes.
Goldstein compiled tables for two- and four-blade propellers and then explained a general method applicable to any number of blades by defining a non-dimensional function K that relates circulation Γ with the fraction r/R as follows:

$$K = \frac{B\omega\Gamma(\frac{r}{R})}{2\pi W V_\infty} \tag{2.8}$$

Where B is the blades number and Γ is the circulation, and W is the rearward displacment velocity of the helical vortex surface at infinity. This function K acts like the Prandtl correction factor in the previous theory. In other words, this function corrects the value of the induced velocity if an unlimited number of blades is assumed.

If the characteristics of the downstream flow are established through experimental data, the circulation distribution can be calculated using Kutta-Joukowski's law.

$$L = \rho V \Gamma \tag{2.9}$$

As a result, the calculations of local properties such as thrust, torque, and efficiency are possible. By integrating these values across the blade, one can obtain the total values [32].

2.2.3.3 The Theory of Theodorsen

Theodorsen's theory plays a vital role in proving that Betz's theorem is valid even for heavily loaded propellers in certain circumstances. This theory is heavily influenced by Goldstein's theory. It demonstrates that the radial component of circulation far behind the wake, in the case of heavily loaded propellers, is the same as that predicted by Goldstein's theory for lightly loaded propellers [33]. In other words, this theory models the heavily loaded propeller as a cascade of an infinite number of lightly loaded propellers.
According to Theodersen, the surface of the vortex equation is:

$$\theta = \frac{\omega z}{V_\infty + W} = 0 \, or \; \pi \tag{2.10}$$

2.3 Review of the modern models to design propellers

2.3.1 Larrabee

Larrabee built his theory based on the fact that the flow has a vortex shape, which harmonizes with the vortex theory. The prescribed shape of the vortex behind the wake region is helical (Archimedes' screw). The perpendicular velocity between the helicoidal sheet at time t and the same helicoidal sheet after Δt is constant, as seen in (2.10b). This constant velocity consists of two components: tangential velocity and displacement velocity (which has an axial direction) [16].

$$w_t = w_n \, sin(\phi) = v' sin(\phi) \, cos(\phi) \tag{2.11}$$

The vortex sheet is assumed to be changeable and forms a tip vortex, as seen in (2.10c) and (2.10d). The tip vortex, in turn, causes errors in the calculations, so Larrabee uses a correction factor (Prandtl factor).
Prandtl estimated the value of F using a mathematical formula for how a fluid flows in two dimensions around the edges of an infinite array of semi-infinite plates. These plates are very long in one direction and are moving straight up or down, not sideways. The average speed of the fluid being pulled along in the same direction as the moving plates is given by this formula [17]:

$$F = \frac{2}{\pi} \, acos(e^f) \tag{2.12}$$

In this formula, f is a number that depends on how far the edges of the plates are from each other:

$$f = \pi \times \left(\frac{\text{edge distance}}{\text{space between plates}} \right)$$

By applying the sheet spacing at the outer edges:

$$f = \frac{B}{2} \left(\frac{\sqrt{\lambda^2 + 1}}{\lambda} \right) \left(1 - \frac{r}{R} \right) \tag{2.13}$$

Larrabee used the same assumption that Betz used to find the optimal efficiency of the propeller. In other words, the induced velocity is equal to half of the displacement velocity. Moreover, Larrabee adopted the concept of axial and radial induced velocity factors (a, a').
As seen in Fig 2.10a, one can rewrite the trigonometric properties, including the velocities:

$$x = \frac{\Omega r}{V} \tag{2.14}$$

$$sin\phi = \frac{1}{\sqrt{1 + x^2}} \tag{2.15}$$

$$cos\phi = \frac{x}{\sqrt{1 + x^2}} \tag{2.16}$$

The definition of circulation according to Stokes is written by taking into consideration the prescribed assumption of the helical vortex as follows:

$$B\,\Gamma = 2\pi r\, F w_t = 2\pi r\, F\, v' sin(\phi)\, cos(\phi) \tag{2.17}$$

By substituting equations 2.14, 2.15, and 2.16 into the previous equation, one can find Goldstein's function G:

$$G = \frac{B\Omega\Gamma}{2\pi v' V} = \frac{x^2}{1 + x^2}\, F \tag{2.18}$$

By rearranging the equation for blade sectional thrust (2.2) using the notion of the drag-to-lift ratio ϵ, one can write:

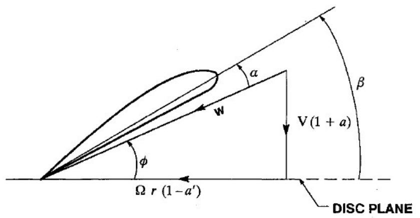
$$T = Lcos\phi(1 - \epsilon\, tan\phi) \tag{2.19}$$

On the other hand, blade sectional lift can be calculated using the Kutta–Joukowski theorem (2.9) by substituting it with circulation:

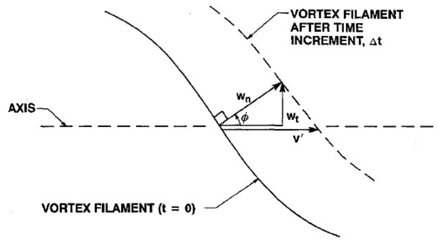
$$Lcos\phi(1 - \epsilon\, tan\phi) = B\rho W\Gamma cos\phi(1 - \epsilon\, tan\phi) \tag{2.20}$$

From Fig. 2.10e, one can find that the velocity $W \cos\phi$ can be replaced by the tangential velocity $\Omega r(1 - a')$, where a' represents the tangential induced factor mentioned earlier.

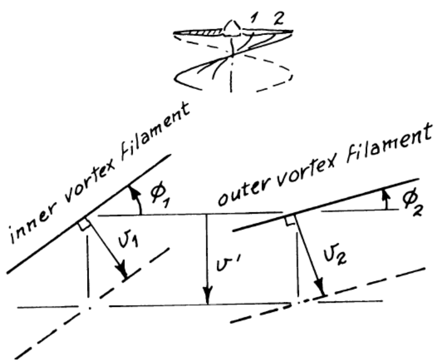
$$T = B\rho\Gamma\Omega r(1 - a')(1 - \epsilon\, tan\phi) \tag{2.21}$$



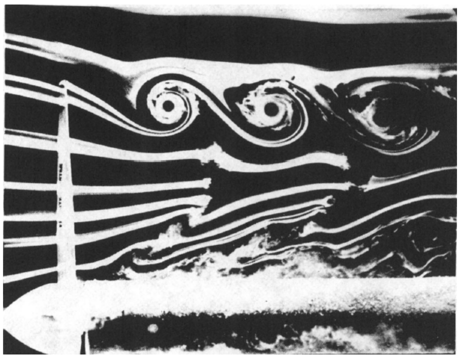
(a) Larrabee's design - velocity triangle



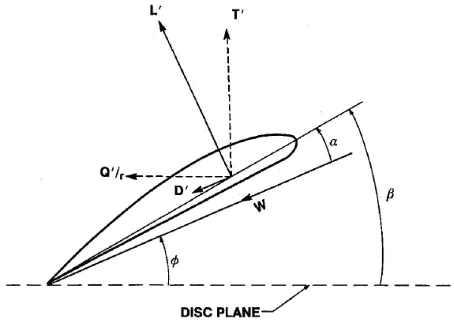
(b) Larrabee's design of minimum induced loss



(c) Betz Condition for Minimum Induced Loss



(d) Smoke Flow Visualization for Running Propeller (Prof. F.M.N. Brown)



(e) Larrabee - Force diagram for a blade element

Figure 2.10: Larrabee extraction theory [17]

Larrabee adopted Betz's assumption, which states that a' should equal the velocity in the wake region w_t divided by 2 to achieve optimal efficiency.

$$a' = \frac{1}{2}\frac{w_t}{\Omega r} = \frac{1}{2}\frac{v' sin(\phi)\, cos(\phi)}{\Omega r} = \frac{1}{2}\zeta\frac{1}{1+x^2} \tag{2.22}$$

ζ here represents the axial velocity displacement ratio $\frac{v_t}{V}$.
Thus, equation 2.21 can be rearranged,

$$T = B\rho\Gamma\Omega r(1 - \frac{1}{2}\zeta\frac{1}{1+x^2})(1 - \epsilon\, tan\phi) \tag{2.23}$$

$$T = 2\pi r\rho G V^2 \zeta(1 - \frac{1}{2}\zeta\frac{1}{1+x^2})(1 - \frac{\epsilon}{x}) \tag{2.24}$$

By setting a specific thrust coefficient $T_c = \frac{2T}{\rho V^2 \pi R^2}$, where R is the tip radius, and isolating the blade sectional thrust as follows:

$$T = \frac{1}{2}\rho V^2 \pi R \frac{dT_c}{d\xi} \tag{2.25}$$

where ξ is the nondimensional radius $\frac{r}{R}$, and this value can vary between $\xi_1 = 1$ when $r = R$ at the tip or $\xi_2 = \mathrm{HTR}$. Thus,

$$\frac{dT_c}{d\xi} = 4\xi G\zeta(1 - \frac{\epsilon}{x}) - 2\zeta^2\xi G(\frac{1-\epsilon}{1+x^2}) \tag{2.26}$$

By integrating the previous equation over ξ_1 and ξ_2, one can find the value of ζ.

$$\zeta = \frac{I_1}{2I_2}(1 + \sqrt{1 - \frac{4I_2T_c}{I_1^2}}) \tag{2.27}$$

$$I_1 = \int_{\xi_1}^{\xi_2} 4\xi G(1 - \frac{\epsilon}{x})d\xi \tag{2.28}$$

$$I_2 = \int_{\xi_1}^{\xi_2} 2\xi G\frac{(1 - \frac{\epsilon}{x})}{(1+x^2)}d\xi \tag{2.29}$$

By obtaining the value of the axial velocity displacement ratio, the axial induction factor can be calculated as:

$$a = \frac{1}{2}\zeta V\frac{x^2}{1+x^2} \tag{2.30}$$

In the end, the flow angle (ϕ) can be calculated using the equation:

$$\phi = arctan(\frac{V(1+a)}{\Omega r(1-a')}) \tag{2.31}$$

Among the prominent weaknesses in this theory are some assumptions, such as:

1. The theory is applicable to light-loading conditions.
2. The wake is represented as an infinite number of vortices.

2.3.2 Adkins

This theory was developed as a complement to McDonnell Douglas's work to facilitate and expand on Larrabee's model (heavy loading). Adkins built his model on both axial momentum theory and vortex theory. The displacement velocity ratio ζ concept is used similarly to Larrabee's but without any assumption of light load. Adkins used the Kutta–Joukowski equation that links Γ with L, and the general equation of circulation in the wake region. Assuming that both equations are equidistant, he derived the circulation distribution that minimizes the loss in kinetic energy. Adkins neglected the assumption of the Archimedes screw shape of

the vortex. Larrabee used the correction factor of Prandtl and Betz's condition for optimum design.
The sectional thrust and torque can be found according to the general momentum theory mentioned before, as shown in Fig. 2.10e and Fig. 2.10a.

$$T = 2\pi r\rho V(1+a)(2VaF) \tag{2.32}$$

$$Q = 2\pi r\rho V(1+a)(2\Omega r a' F) \tag{2.33}$$

Adkins uses the concept of displacement velocity like Larrabee, so the equations (2.14), (2.15), and (2.16) are implemented here as well.
Moreover, Adkins found the relation of circulation (Stokes' law) in Eq (2.17) and substituted it into the Kutta–Joukowski equation in Eq (2.18).
The last similar step between Larrabee and Adkins is the use of the drag-to-lift ratio ϵ to find the thrust and torque relations:

$$T = Lcos\phi(1 - \epsilon\, tan\phi) \tag{2.34}$$

$$Q = L\, sin\phi(1 + \frac{\epsilon}{tan\phi}) \tag{2.35}$$

The sectional thrust is calculated by two different methods previously: the general momentum method in Eq (2.33) and the last equation in Eq (2.33). By using the concept of axial velocity displacement ratio $\zeta = v'/V$, the axial and tangential induction factors can be calculated as:

$$a = \frac{\zeta}{2}cos^2\phi(1 - \epsilon tan\phi) \tag{2.36}$$

$$a' = \frac{\zeta}{2x}cos\phi\, sin\phi(1 + \frac{\epsilon}{tan\phi}) \tag{2.37}$$

Those equations with the Fig. 2.10a lead to:

$$tan\phi = \frac{(1 + \frac{\zeta}{2})}{x} = (1 + \frac{\zeta}{2})\frac{\lambda}{\xi} \tag{2.38}$$

By applying Betz's condition $r \tan\phi$ to ensure minimum induced loss, and to maintain the vortex shape similar to an Archimedes screw, the value ζ must be constant. The relationship between the speed ratio x, the axial velocity displacement ratio ζ, the advance ratio λ, and the nondimensional radius ξ is:

$$x = \frac{\Omega r}{V} = \frac{\frac{r}{R}}{\lambda} = \frac{\xi}{\lambda} \tag{2.39}$$

Since ζ is constant, as proved before, it is clear that equation 2.38 is a result of equations **??** and 2.37 without the influence of viscosity.

1. Specify the value of the nondimensional thrust T_c:

$$T_c = \frac{2T}{\rho V^2 \pi R^2} \tag{2.40}$$

2. On the other hand, this value can also be written as:

$$T_c = I_1\zeta - I_2\zeta^2 \tag{2.41}$$

3. The integrations I_1 and I_2 are:

$$I_1 = 4\xi G(1 - \epsilon \tan\phi) \tag{2.42}$$

$$I_2 = \lambda \frac{I_1}{2\xi}\left(1 + \frac{\epsilon}{\tan\phi}\right)\sin\phi\cos\phi \tag{2.43}$$

4. The constraint equation is:

$$\zeta = \frac{I_1}{2I_2} - \left[\left(\frac{I_1}{2I_2}\right)^2 - \frac{T_c}{I_2}\right]^{0.5} \tag{2.44}$$

It should be noted that the Prandtl correction factor in this theory differs in its calculation method from the Larrabee method:

$$F = \frac{2}{\pi} arccos(e^{-f}) \tag{2.45}$$

$$f = \frac{B}{2}\frac{(1-\xi)}{sin\phi_t} \tag{2.46}$$

$$tan\phi_t = \lambda(1 + \zeta/2) \tag{2.47}$$

$$tan\phi = tan\phi_t/\zeta \tag{2.48}$$

Among the prominent weaknesses in this theory are the following assumptions:
1. The theory neglects the effects of assuming that the flow is steady and that there is no turbulence.
2. The flow is assumed to be 2D rather than 3D.

2.3.3 Drela Method

This theory relies on classical blade-element and vortex theories without the need for momentum theory [19].

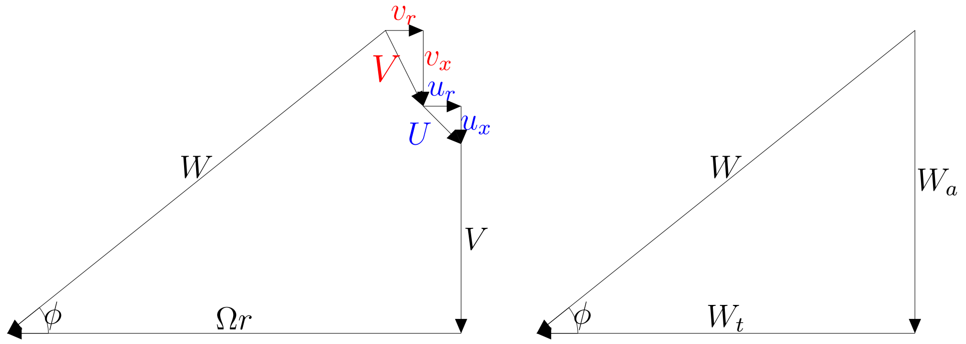


Figure 2.11: Drela velocity triangle

As seen in Fig. 2.11, the relative total velocity W consists of two components: tangential W_t and axial W_a. Changes in velocity occur due to both rotor movement and external factors. Drela combines the freestream velocity V with the axial upstream velocity u_a (the external local velocity), creating a new axial velocity component called U_a. Similarly, $\Omega r - u_t$ is combined with the tangential upstream velocity u_t, creating a new tangential velocity component called U_t. Together, these components can be referred to as the environmental velocity U. Thus, the velocities are initially calculated as follows:

$$W_a = V + u_a + v_a \tag{2.49}$$

$$W_t = \Omega r + u_t + v_t \tag{2.50}$$

$$W = \sqrt{W_a^2 + W_t^2} \tag{2.51}$$

Drela did not rely on any momentum theory to calculate the circulation in his approach. Instead, he used the same assumptions made by Kelvin [21] and Helmholtz's Theorem [22], which state that circulation is constant along a vortex tube for a barotropic, ideal fluid with conservative body forces. Drela assumed a constant circulation, $\Gamma(r)$, at radius r. Additionally, he employed the Biot–Savart law and its results, such as the induced velocity from a semi-infinite vortex being half of the induced velocity from infinite trailing vortices. This approach eliminates the need for the momentum equation. In the wake region, Drela utilized the concept of average velocity, $\overline{v}$, so that the circulation in the wake region is half of the circulation afterward, as illustrated in Fig. 2.12. Based on this hypothesis, the equation for circulation in the wake region is written as:

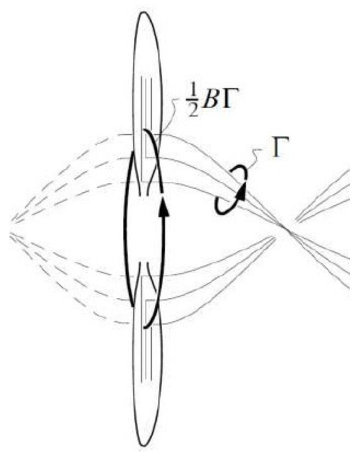


Figure 2.12: Flow geometry for blade element at radial station r [19].

$$2\pi r\overline{v}_t = \frac{1}{2}B\Gamma \rightarrow \overline{v}_t = \frac{B\Gamma}{4\pi r} \tag{2.52}$$

As seen in the previous equation, Drela used the concept of circumferentially-averaged tangential velocity, v_t, where he assumed that $\overline{\Gamma} = \Gamma/2\pi = rv_t$. This relation is applied in MIT's Aerospace Computational Design Laboratory's `MTFLOW` (empirical relation), where the factor K used in this case is $K = B/2r$, as referenced in the guidelines or in [20].

$$\bar{v_t} = v_t F \sqrt{1 + (\frac{4\lambda_w}{\pi B \xi})^2} \tag{2.53}$$

Also, Prandtl's factor F used by Drela is different from that in Larrabee's and Adkins methods and is calculated by the following equations:

$$F = \frac{2}{\pi} acos(e^{-f}) \tag{2.54}$$

$$f = \frac{B}{2}(1 - \xi)\frac{1}{\lambda_w} \tag{2.55}$$

$$\lambda_w = \xi tan(\phi) \tag{2.56}$$

$$\zeta = \frac{r}{R} \tag{2.57}$$

$$tan(\phi) = \frac{W_a}{W_t} \tag{2.58}$$

As indicated in the preceding equation, the advance ratio $\lambda = V/\Omega R$ is replaced by the advance ratio in the wake, λ_w. The value of λ_w is more accurate because it includes the induced velocities and has a specific value for each segment at radius r. As a result of this substitution, it can be observed that the local tangential induced velocity can be expressed in the following form:

$$v_t = \frac{B\Gamma}{4\pi r} \frac{1}{F\sqrt{1 + (\frac{4\lambda_w}{\pi B \xi})^2}} \tag{2.59}$$

On the other hand, from the trigonometric similarity relations, v_a can be calculated as:

$$v_a = v_t \frac{W_t}{W_a} \tag{2.60}$$

The assumption that v is perpendicular to W is important, as it will facilitate the solution, but it is only valid if a light load is applied to the rotor. In other words, if the helix shape contracts in the wake region, the pitch will not remain constant, leading to a variable distribution of circulation. This assumption introduces derivatives. The opposite case, as previously described by Goldstein, provides more accurate results.

2.3.3.1 Local analysis solution

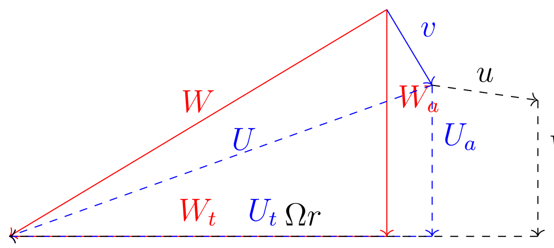


Figure 2.13: A schematic illustration is provided to clarify Drela's local velocity triangles.

As mentioned earlier, the first step in the analysis is to sum the local external induced velocity with the freestream and tangential velocities to create U_a and U_t. The resultant velocity, U, is the hypotenuse of two triangles: the first one (U_a, U_t, U) and the second one (W, V, U). As shown in Fig. 2.14, the shape (W, v, U_a, U_t) forms a cyclic quadrilateral, with its center located at the midpoint of U. The radius of this circle is $\frac{1}{2}U$. The specific radius that reaches the intersection of v and W defines an angle with the horizontal, called ψ. One should notice that the horizontal line passing through the center divides U_a into two equal halves, while the vertical line passing through the center similarly divides U_t.

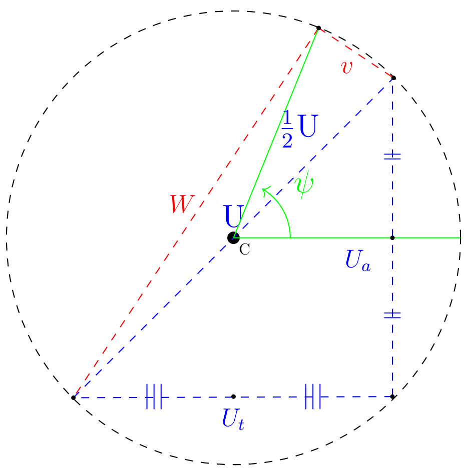


Figure 2.14: Velocity parameterization by the angle ψ.

Instead of directly iterating the circulation, Drela used the dummy variable ψ. This approach determined the dummy variable ψ and calculated all velocities based on it as seen in what follows:

$$U_a = V + u_a \tag{2.61}$$
$$U_t = \Omega r - u_t \tag{2.62}$$
$$U = \sqrt{U_a^2 + U_t^2} \tag{2.63}$$
$$\psi = arctan(\frac{U_a}{U_t}) \tag{2.64}$$
$$W_a(\psi) = \frac{1}{2}U_a + \frac{1}{2}U\sin(\psi) \tag{2.65}$$
$$W_t(\psi) = \frac{1}{2}U_a + \frac{1}{2}U\cos(\psi) \tag{2.66}$$
$$W(\psi) = \sqrt{W_t^2 + W_a^2} \tag{2.67}$$
$$v_a(\psi) = W_a - U_a \tag{2.68}$$
$$v_t(\psi) = U_t - W_t \tag{2.69}$$
$$\alpha(\psi) = \beta - \arctan\left(\frac{W_a}{W_t}\right) \tag{2.70}$$
$$Re(\psi) = \frac{\rho W C}{\mu} \tag{2.71}$$
$$Ma(\psi) = \frac{W}{a} \tag{2.72}$$

Also, the relation between circulation and tangential-induced velocity can be parameterized using the variable ψ.

$$\lambda_w(\psi) = \frac{r}{R}\frac{W_a}{W_t} \tag{2.73}$$
$$f(\psi) = \frac{B}{2}(1 - \frac{r}{R})\frac{1}{\lambda_w} \tag{2.74}$$
$$F(\psi) = \frac{2}{\pi}\arccos(e^{-f}) \tag{2.75}$$
$$\Gamma(\psi) = v_t(\psi)\frac{4\pi r}{B}F(\psi)\sqrt{1 + \left(\frac{4\lambda_w(\psi)R}{\pi B r}\right)^2} \tag{2.76}$$

The residuals will be calculated for the dummy variable ψ until convergence. After obtaining ψ, the general residual of thrust T or power P will decrease.

2.3.3.2 Local thrust and torque relations

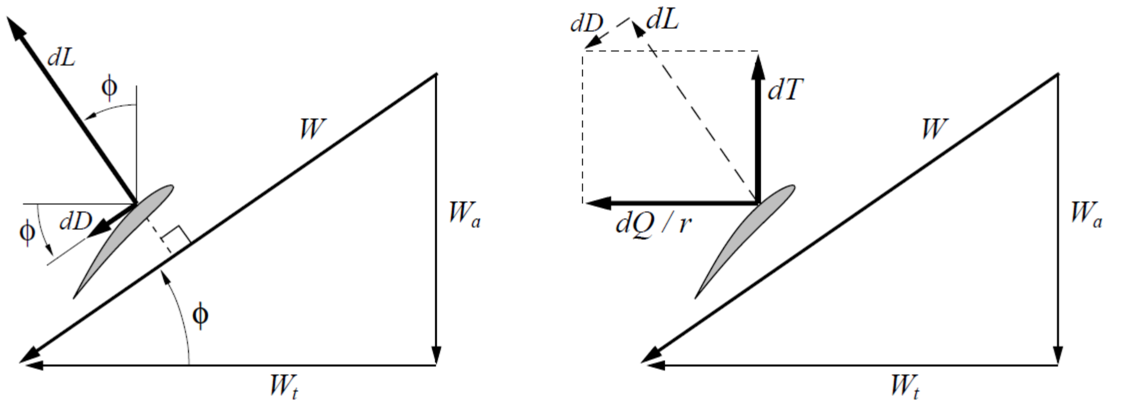


Figure 2.15: Blade lift and drag resolved into thrust and torque components [19].

The local efficiency can be calculated by substituting the velocities from the velocity triangle into the equations of local thrust and torque as follows:

$$dT = \rho B \Gamma (W_t - \epsilon W_a) dr \tag{2.77}$$

$$dQ = \rho B \Gamma (W_a - \epsilon W_t) r dr \tag{2.78}$$

$$\epsilon = \frac{C_d}{C_l} \tag{2.79}$$

$$\eta = \frac{V dT}{\Omega dQ} = \frac{V}{\Omega r} \frac{W_t - \epsilon W_a}{W_a - \epsilon W_t} \tag{2.80}$$

$$\eta_i = \frac{1 - (v_t/U_t)}{1 - (v_a/U_a)} \tag{2.81}$$

$$\eta_p = \frac{1 - (W_a/W_t)}{1 + (W_t/W_a)} \tag{2.82}$$

$$\eta = \eta_i \eta_p \tag{2.83}$$

This step is the final step. After obtaining the velocity triangles (2.14), the calculations of thrust, power, and efficiencies are performed.

2.3.4 Total Loads and Efficiency

After calculating the local parameters, the final step in blade element theory is to integrate over the entire blade, i.e., from the root to the tip radius.

$$T = \rho B \int_0^R \Gamma (W_t - \epsilon W_a) dr = \rho B \sum_r \Gamma (W_t - \epsilon W_a) \Delta r \tag{2.84}$$

$$Q = \rho B \int_0^R \Gamma (W_a + \epsilon W_t) dr = \rho B \sum_r \Gamma (W_a + \epsilon W_t) \Delta r \tag{2.85}$$

$$\eta = \frac{VT}{\Omega Q} \tag{2.86}$$

2.3.4.1 Comparison between Drela, Larrabee, and Adkins models

All three theories adopt the concept of vortex theory but differ in how they specify the shape of the vortex. Each approach requires the use of vortex theory to determine circulation in the wake region.
Larrabee relies on Goldstein's assumption, which states that the vortex formed behind the blade section takes the shape of a helix with a fixed pitch, similar to an Archimedean screw. This implies that the flow angle, ϕ, remains constant along the vortex filament. In contrast, Adkins adopts a more general vortex shape based on Theodorsen's prescription, where the flow angle in the far wake region matches the flow angle at the blade section.
From both Larrabee's and Adkins' perspectives, minimizing induced loss involves calculating the lift force. This can be done using general momentum theory and the Kutta-Joukowski equation, which requires knowing the circulation value in the wake region. Both methods use the concept of displacement velocity and derive the tangential velocity through geometric relationships, such as those found in velocity triangles.
On the other hand, Drela, like Larrabee, adopts Goldstein's assumption regarding vortex shape. He also relies on Helmholtz's theory, which states that circulation remains constant along a vortex tube for a barotropic or ideal fluid. Consequently, Drela's approach does not require momentum theories.
Drela uses an empirical equation to calculate the circumferentially-averaged tangential velocity for use in the circulation equation at the rotor section. Additionally, he employs a dummy variable, ψ, to determine the total local velocity, which is then used in the circulation equation for the wake region. Essentially, Drela's method relies on computational iteration to determine the residuals of circulation differences between the wake and rotor regions.
Furthermore, Drela uses the local advance ratio λ_w instead of the overall advance ratio λ, leading to more accurate results when calculating the Prandtl correction factor. It is worth noting that Drela's method is unique among the three in that it accounts for the presence of induced velocity from the rotor.

2.4 Acoustic Study

2.4.1 Literature Review

Since the First World War, reducing noise from propellers has been a major focus for scientists. The first attempt to propose a theory was made by Lynam and Webb in 1919 [34]. Their published paper stated that propeller noise occurs as a result of the constant motion of the pressure center or close to it. However, this prediction method was proven incorrect by Morfey's published review in 1973.

The first theory was developed by Guten, who demonstrated that the noise, according to Lynam, was directed correctly and also addressed the noise generation process (thrust).

Subsequently, Daming published two reports in 1937 and 1938, presenting his theory that noise resulted from thickness. Later, in 1940, he published another theory proposing that noise was due to thrust and torque. He adopted Guten's approach but replaced the thrust and torque with a full distribution of these factors over the disk.

In 1942, Guten reconsidered his previous theory and incorporated the effect of blade thickness alongside the full distribution of thrust and torque.

In 1953, Garrick and Watkins expanded upon Guten's research on noise from thrust and torque by incorporating the effects of subsonic forward flight.

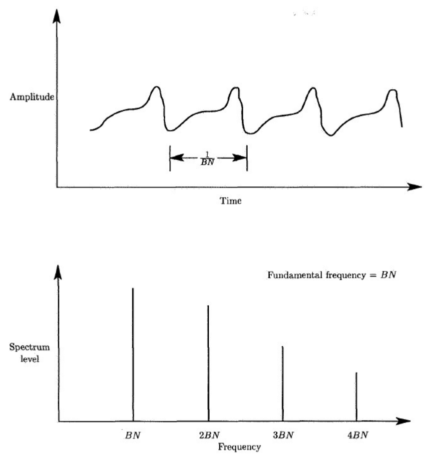
2.4.2 Propeller Noise Characteristics

The noise of the propeller can be categorized into three criteria [35]:

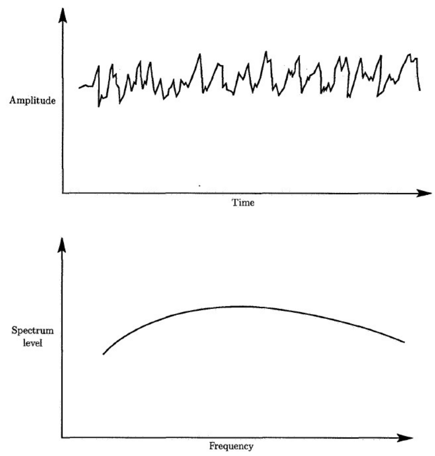
1. Harmonic noise: This type of noise has a periodic function, meaning it has a pulse shape that repeats after a specific interval, which is called the frequency $(1/BN)$. See Fig. 4.4.

2. Broadband noise: This type of noise changes its shape randomly and does not follow a routine. This occurs because each frequency has a unique amplitude. See Fig. 4.5.

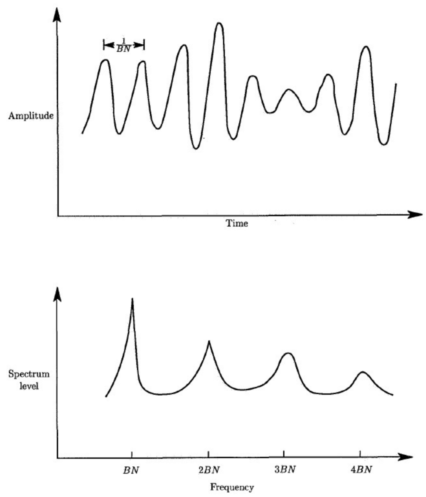
3. Narrow-band random noise: The shape looks periodic but it is not repeated. See Fig. 4.6.



(a) Characteristics of propeller rotational noise



(b) Characteristics of propeller Broadband noise



(c) Characteristics of propeller narrow-band random noise

Figure 2.16: Propeller Noise Characteristics [35]

2.4.3 Propeller Noise-Generating Mechanisms

2.4.3.1 Steady Sources

If an observer is located on the blade, the steady sources will seem constant relative to time. The noise generated by this type of source is periodic due to the rotational movement of the blade.
In general, noise sources are classified into three types [35]:

- Thickness: This type is linear. As the blade rotates, it periodically pushes a volume of air, which generates noise. The amplitude of this noise depends on the blade's volume, while the frequency depends on the rotational speed and the airfoil shape.
- Loading: This type is also linear. As mentioned before, when the blade moves, lift and drag forces are generated due to the pressure differences, which can

be represented as thrust and torque. The pressure anomaly is the real source of this type of noise.
- Quadrupole Sources: This type is non-linear and represents the effects of viscosity and propagation, which are not included in the thickness and loading types.

2.4.3.2 Unsteady Sources

The unsteady source is time-dependent, involving both periodic and irregular changes in the load on the blades [35]. A common example of this type of unsteady source is the effect of the angle of attack on the blade loading. When the incoming flow varies, it alters the propeller's core, causing the local angle of attack of each blade to fluctuate. Consequently, this leads to changes in the load on the blades.
Changes in loading can occur once per revolution or multiple times per revolution. The change in inflow that doesn't depend on time leads to changes in blade loading, which repeat exactly with every rotation. The resulting periodic unsteady noise is generated when the blade revolution frequencies occur harmonically.

2.4.4 Garrick and Watkins

The OPTPPROP model is designed to calculate the thrust value for the optimal design of a low-speed propeller, making the Garrick and Watkins method the best choice, as it neglects the effects of thickness and viscosity as sources of noise at low speeds [36].

2.4.4.1 Extension of Lamb's equation

The Garrick and Watkins theory builds on Lamb's pressure formula [37], which calculates the pressure at any point in the field resulting from a periodic force. They extended this theory by considering the force to be moving, rather than fixed, and applied it as a three-component force.

$$p = \frac{A}{4\pi}\frac{\partial}{\partial x}\frac{e^{i\omega\left(t-\frac{s}{c}\right)}}{S} \tag{2.87}$$

The value P represents the pressure at (x, y, z). The term $Ae^{i\omega t}$ is an external periodic force in one dimension, while S represents the distance shown in Fig. 2.17. This is applied by using an arbitrary force with three components: F_x, F_y, and F_z.

$$p = \frac{1}{4\pi}\left[\frac{\partial}{\partial x}\frac{F_x\left(t-\frac{s}{c}\right)}{s} + \frac{\partial}{\partial y}\frac{F_y\left(t-\frac{s}{c}\right)}{s} + \frac{\partial}{\partial z}\frac{F_z\left(t-\frac{s}{c}\right)}{s}\right] \tag{2.88}$$

The equation 2.88 derived by Lamb is for a concentrated periodic force in a fixed field in space. This equation is extended by incorporating forward speed, assuming the concentrated force moves with a uniform velocity V along with the coordinates, as follows:

$$p = \frac{1}{4\pi}\left[\frac{\partial}{\partial x}\frac{F_x\left(t-\frac{\sigma}{c}\right)}{s} + \frac{\partial}{\partial y}\frac{F_y\left(t-\frac{\sigma}{c}\right)}{s} + \frac{\partial}{\partial z}\frac{F_z\left(t-\frac{\sigma}{c}\right)}{s}\right] \tag{2.89}$$

where;

$$S = \sqrt{(x-x_l)^2 + \beta^2(y-y_l)^2 + (z-z_l)^2} \tag{2.90}$$

$$\sigma = \frac{M(x-x_l)+s}{\beta^2} \tag{2.91}$$

$$\beta = \sqrt{1-M^2} \tag{2.92}$$

The geometric meanings of S, s and σ are illustrated in Fig 2.17,

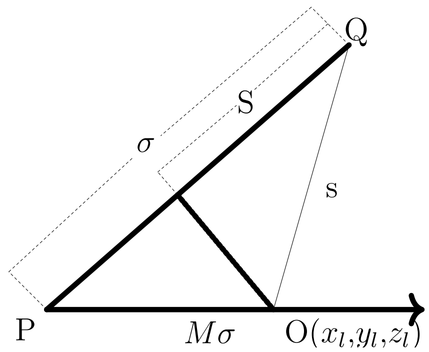


Figure 2.17: The geometric meaning of S, s and σ [29]

The origin or the location of the force is $O(x_l, y_l, z_l)$, while the points of the field are $Q(x, y, z)$. The effect at point Q at time t arises from the force's action when it was at position P. The Mach number M is used to specify the location P by the relation $M = \frac{OP}{OQ}$, where σ is the phase radius. The value S is determined by dropping O perpendicularly onto PQ.
By substituting S, σ, and $k = \omega/c$, the resultant equation 2.93 represents the pressure at any field point linked to a moving force.

$$p = \frac{1}{4\pi}\left(F_x\frac{\partial}{\partial x} + F_y\frac{\partial}{\partial y} + F_z\frac{\partial}{\partial z}\right)\frac{e^{-ik\sigma}}{S} \tag{2.93}$$

2.4.4.2 Disturbance Forces in the Propeller Plane

As mentioned earlier in the literature review section, Garrick and Watkins adopted the effective radius approach, similar to Guten [38].
This means that the propeller is located in the YZ plane, with the X axis passing through the core of the propeller. The propeller moves forward in the X direction with a velocity of V. By transitioning to polar coordinates to represent the location of the center where the force is concentrated:

$$x_l = 0, \tag{2.94}$$
$$y_l = r\cos(\theta), \tag{2.95}$$
$$z_l = r\sin(\theta). \tag{2.96}$$

If the propeller rotates, thrust and torque forces will be generated. As a result, the medium will produce periodic forces, equivalent in magnitude but in the opposite direction to the thrust T and torque Q. These forces are the source of noise, and the relationships between them are as follows:

$$\left.\begin{aligned} \frac{dT}{dr} &= BA(r) \\ \frac{dQ}{dr} &= BrF(r) \end{aligned}\right\} \tag{2.97}$$

Where $A(r)$ is the chordwise distribution of thrust acting on a radial element of a propeller blade, $F(r)$ is the chordwise distribution of forces perpendicular to the thrust of the propeller blade that giving rise to the torque and B is the number of blades. By isolating thrust and torque:

$$\left.\begin{aligned} T &= B\int_0^R A(r)dr \\ Q &= B\int_0^R rF(r)dr \end{aligned}\right\} \tag{2.98}$$

Since the force acts on a rectangular area (width/chord), this area will be denoted as $r\,dr\,d\theta$ for each element. Assuming the overlapping begins at $t = 0$ and ends at $t = \tau = \frac{b}{r\Omega}$, and for the second blade, the process begins at $t = \tau_0 = \frac{2\pi}{B\Omega}$, the force equation can be rewritten as a Fourier series:

$$F_1^*(t) = \begin{cases} A(r)\frac{rd\theta}{b}dr & \text{for } 0 < t < \tau \\ 0 & \text{for } \tau < t < \tau_0 \end{cases} \tag{2.99}$$

$$F_1^*(t) = A_0 + \sum_{m=1}^{\infty} A_m \cos(mB\Omega t - \epsilon_m) \tag{2.100}$$

and for the second force:

$$F_2^*(t) = \begin{cases} F(r)\frac{rd\theta}{b}dr & \text{for } 0 < t < \tau \\ 0 & \text{for } \tau < t < \tau_0 \end{cases} \tag{2.101}$$

$$F_2^*(t) = B_0 + \sum_{m=1}^{\infty} B_m \cos(mB\Omega t - \eta_m) \tag{2.102}$$

In which:

$$\left.\begin{aligned} A_m &= \frac{2}{m\pi} A(r) \frac{r}{b} \sin m\pi \frac{\tau}{\tau_0} dr d\theta \\ B_m &= \frac{2}{m\pi} F(r) \frac{r}{b} \sin m\pi \frac{\tau}{\tau_0} dr d\theta \\ \epsilon_m &= \eta_m = m\pi \frac{\tau}{\tau_0} = \frac{mB\Omega\tau}{2\tau_0} \end{aligned}\right\} \tag{2.103}$$

Here, A_0 and B_0 are constants representing the instantaneous average values of thrust and torque, while ϵ_m and η_m are relatively small values. After rotating by an angle of θ, the forces will shift as follows:

$$\left.\begin{aligned} F_1(t) =& F_1^*(t - \frac{\theta}{\Omega}) = A_0 + \sum_{m=1}^{\infty} A_m \cos(mB\Omega t - mB\theta - \epsilon_m) \\ F_2(t) =& F_2^*(t - \frac{\theta}{\Omega}) = B_0 + \sum_{m=1}^{\infty} B_m \cos(mB\Omega t - mB\theta - \eta_m) \end{aligned}\right\} \tag{2.104}$$

One can notice that the values of $\frac{m\pi\tau}{\tau}$ can be written as:

$$\frac{m\pi\tau}{\tau} = \frac{mB\Omega\tau}{2} = \frac{mBb}{2r} \tag{2.105}$$

which is small if the harmonic number m is small. Thus, substitution of $\sin\left(\frac{m\pi\tau}{\tau_0}\right)$ with $\frac{m\pi\tau}{\tau_0}$ is accepted. Moreover, the relations in 2.102 will be:

$$\left.\begin{aligned} A_m &= \frac{B}{\pi} A(r) dr d\theta = \frac{dT}{dr} dr d\theta \\ B_m &= \frac{B}{\pi} F(r) dr d\theta = \frac{1}{r}\frac{dQ}{dr} dr d\theta \end{aligned}\right\} \tag{2.106}$$

2.4.4.3 The Sound Pressure Field

The comprehensive form of the formula that describes the periodic forces associated with thrust and torque can be written in the complex form, knowing that $w = mB\Omega$, as follows:

$$\left.\begin{aligned} F_1 &= A_0 + \sum_{m=1}^{\infty} A_m e^{i(\omega t - mB\theta - \epsilon_m)} \\ F_2 &= B_0 + \sum_{m=1}^{\infty} B_m e^{i(\omega t - mB\theta - \eta_m)} \end{aligned}\right\} \tag{2.107}$$

The result of substituting the simplified values of A_m and B_m from equation 2.106 into the expression for the periodic forces applied to a medium element is expressed as:

$$\left.\begin{aligned} F_{x,m} &= -\frac{1}{\pi}\frac{dT}{dr} e^{(\omega t - mB\theta)}\, dr\, d\theta \\ F_{y,m} &= -\frac{1}{\pi}\frac{1}{r}\frac{dQ}{dr} \sin\theta\, e^{(\omega t - mB\theta)}\, dr\, d\theta \\ F_{z,m} &= \frac{1}{\pi}\frac{1}{r}\frac{dQ}{dr} \cos\theta\, e^{(\omega t - mB\theta)}\, dr\, d\theta \end{aligned}\right\} \tag{2.108}$$

Those equations 2.108 can calculate the pressure at any point (x, y, z) in the field for each element. To obtain the total pressure over the disk, two integrations should be performed: the first over dr with limits from 0 to R, and the second over $d\theta$ with limits from 0 to 2π.

$$P_T = -\frac{1}{4\pi^2}\int_0^R\int_0^{2\pi}\frac{dT}{dr}\,e^{i(\omega t - mB\theta)}\,\frac{\partial}{\partial x}\,\frac{e^{-ik\sigma}}{S}\,dr\,d\theta \tag{2.109}$$

$$P_Q = -\frac{1}{4\pi^2}\int_0^R\int_0^{2\pi}\frac{1}{r}\frac{dQ}{dr}\,e^{i(\omega t - mB\theta)}\,\frac{\partial}{\partial x}\,\frac{e^{-ik\sigma}}{S}\,dr\,d\theta \tag{2.110}$$

Thus, the total oscillating pressure is obtained by adding the thrust pressure and the torque pressure:

$$P = P_T + P_Q \tag{2.111}$$

To exclude the partial parts from the equation, the geometrical values in Eq2.92 are used as follows:

$$\left.\begin{aligned}\frac{\partial}{\partial x}\frac{e^{-ik\sigma}}{S} &= \frac{e^{-ik\sigma}}{S}\left(-\frac{ikM}{\beta^2} - \frac{ikx}{\beta^2 S} - \frac{x}{S^2}\right)\\ \frac{1}{r}\frac{\partial}{\partial\theta}\frac{e^{-ik\sigma}}{S} &= \frac{e^{-ik\sigma}}{S}\left(-\frac{ik}{S} - \frac{\beta^2}{S^2}\right)(y\,\sin\theta - z\cos\theta)\end{aligned}\right\} \tag{2.112}$$

Hence:

$$P_T = \frac{e^{i\omega t}}{4\pi^2}\int_0^R\int_0^{2\pi}\frac{dT}{dr}\,e^{-mB\theta}\,\frac{e^{-ik\sigma}}{S}\left(-\frac{ikM}{\beta^2} - \frac{ikx}{\beta^2 S} - \frac{x}{S^2}\right)dr\,d\theta \tag{2.113}$$

$$P_Q = \frac{e^{i\omega t}}{4\pi^2}\int_0^R\int_0^{2\pi}\frac{1}{r}\frac{dQ}{dr}\,e^{-mB\theta}\,\frac{e^{-ik\sigma}}{S}\left(-\frac{ik}{S} - \frac{\beta^2}{S^2}\right)(y\,\sin\theta - z\cos\theta)\,dr\,d\theta \tag{2.114}$$

$$= \frac{e^{i\omega t}}{4\pi^2}\int_0^R\int_0^{2\pi}\frac{1}{r}\frac{dQ}{dr}\,e^{-mB\theta}\,\frac{e^{-ik\sigma}}{S}\,imB\,dr\,d\theta \tag{2.115}$$

2.4.4.4 Implementing the Effective Radius R_e

The calculation is based on the premise that the forces are applied to an annular ring instead of a full circle, in order to simplify the equations by canceling one of the double integrations (over the propeller radius). To achieve this, a variable called the effective radius, denoted as R_e, is defined. This effective radius is a function of the diameter, given by $R_e = 0.8 \times \frac{D}{2}$. The loading position in polar coordinates (r, θ) can be expressed as follows:

$$x_l = 0 \tag{2.116}$$

$$y_l = R_e\cos(\theta) \tag{2.117}$$

$$z_l = R_e\sin(\theta) \tag{2.118}$$

Thus pressure arising from thrust and torque equations will be written respectively:

$$p_T = \frac{e^{i\omega t}}{4\pi^2}\int_0^{2\pi} T(\frac{ikM}{\beta^2} + \frac{ikx}{\beta^2 S} + \frac{x}{S^2})\frac{e^{-i(mB\theta+k\sigma)}}{S}d\theta \tag{2.119}$$

$$p_Q = \frac{-e^{i\omega t}}{4\pi^2}\int_0^{2\pi} \frac{Q}{R_e^2}imB\frac{e^{-i(mB\theta+k\sigma)}}{S}d\theta \tag{2.120}$$

Thereafter, the root-mean-square pressure p'_{rms}, which is calculated as the square root of the average of the squares of instantaneous pressure values over time, can be calculated from the equation:

$$p'_{rms} = \frac{\sqrt{2}}{2}|p'| = \frac{\sqrt{2}}{2}|p'_T + p'_Q| = \frac{\sqrt{2}}{8\pi^2}(A^2 + B^2)^{\frac{1}{2}} \tag{2.121}$$

Where A and B are calculated as follows:

$$A = \int_0^{2\pi}\{\frac{Tx}{S^2}sin(mB\theta + k\sigma) + [T\frac{k}{\beta^2}(M + \frac{x}{S}) - Q\frac{mB}{R_e^2}]sin(mB\theta + k\sigma)\}\frac{d\theta}{S} \tag{2.122}$$

$$B = \int_0^{2\pi}\{-\frac{Tx}{S^2}sin(mB\theta + k\sigma) + [T\frac{k}{\beta^2}(M + \frac{x}{S}) - Q\frac{mB}{R_e^2}]cos(mB\theta + k\sigma)\}\frac{d\theta}{S} \tag{2.123}$$

Finally, The sound pressure level (SPL) measures sound intensity relative to a reference pressure, typically 20 µPa, and is expressed in decibels (dB). the sound pressure level is calculated by the equation:

$$SPL = 20\,log_{10}(\frac{p'_{rms}}{p_{ref}}) \tag{2.124}$$

2.4.5 Near Field

The pressure arising from thrust and torque equations will be written respectively:

$$p_T = \frac{e^{i\omega t}}{4\pi^2}\int_0^{2\pi} T\left(\frac{ikM}{\beta^2} + \frac{ikx}{\beta^2 S} + \frac{x}{S^2}\right)\frac{e^{-i(mB\theta+k\sigma)}}{S}\,d\theta \tag{2.125}$$

$$p_Q = \frac{-e^{i\omega t}}{4\pi^2}\int_0^{2\pi} \frac{Q}{R_e^2}imB\frac{e^{-i(mB\theta+k\sigma)}}{S}\,d\theta \tag{2.126}$$

$$S = \sqrt{(x - x_l)^2 + \beta^2(y - y_l)^2 + (z - z_l)^2} \tag{2.127}$$

$$\sigma = \frac{M(x - x_l) + S}{\beta^2} \tag{2.128}$$

$$\beta = \sqrt{1 - M^2} \tag{2.129}$$

Thereafter, the root-mean-square pressure p'_{rms} can be calculated from the equation:

$$p'_{\mathrm{rms}} = \frac{\sqrt{2}}{2}|p'| = \frac{\sqrt{2}}{2}|p'_T + p'_Q| = \frac{\sqrt{2}}{8\pi^2}(A^2 + B^2)^{\frac{1}{2}} \tag{2.130}$$

Where A and B are calculated as follows:

$$A = \int_0^{2\pi} \left\{ \frac{Tx}{S^2} \sin(mB\theta + k\sigma) + \left[T\frac{k}{\beta^2}\left(M + \frac{x}{S}\right) - Q\frac{mB}{R_e^2} \right] \sin(mB\theta + k\sigma) \right\} \frac{d\theta}{S} \tag{2.131}$$

$$B = \int_0^{2\pi} \left\{ -\frac{Tx}{S^2} \sin(mB\theta + k\sigma) + \left[T\frac{k}{\beta^2}\left(M + \frac{x}{S}\right) - Q\frac{mB}{R_e^2} \right] \cos(mB\theta + k\sigma) \right\} \frac{d\theta}{S} \tag{2.132}$$

Thereafter, the sound pressure level is calculated by the equation:

$$SPL = 20\,\log_{10}\left(\frac{p'_{\mathrm{rms}}}{p_{\mathrm{ref}}}\right) \tag{2.133}$$

2.4.6 Far Field

As it was implemented in the near field section, the first step will be the equation of pressure generated from thrust and torque respectively:

$$p'_T = T\frac{e^{i\omega t}}{2\pi S_0} i^{mB} e^{-i\frac{k}{\beta^2}(S_0 + Mx)} \frac{ik}{\beta^2}\left(M + \frac{x}{S_0}\right) J_{mB}\left(\frac{kzR_e}{S_0}\right) \tag{2.134}$$

$$p'_Q = -\frac{Q}{R_e^2}\frac{e^{i\omega t}}{2\pi S_0} i^{mB+1} mB e^{-i\frac{k}{\beta^2}(S_0 + Mx)} J_{mB}\left(\frac{kzR_e}{S_0}\right) \tag{2.135}$$

$$S_0 = \sqrt{x^2 + \beta^2 z^2} \tag{2.136}$$

Here, J_{mB} represents the Bessel function. Thereafter, the calculations of the frequency ΩB, which is needed to calculate p'_{rms} in the last step, are as follows:

$$p'_{\mathrm{rms,FarField}} = \frac{m\omega_1}{2\sqrt{2}\pi a S_0}\left| T\left(M + \frac{x}{S_0}\right)\frac{1}{\beta^2} - Q\frac{Ba}{\omega_1 R_e^2} \right| J_{mB}\left(\frac{kzR_e}{S_0}\right) \tag{2.137}$$

3

Methods

This thesis aims to extend Optoprop to include Drela's method in the Python version and to conduct an acoustic analysis for design.

3.1 Optoprop code

Optoprop is a code developed by Alexander Capitao Patrao in 2017 as an in-house tool at Chalmers University of Technology [18]. It was designed to facilitate achieving minimum induced loss efficiency by implementing momentum/vortex theory, specifically for designing open propellers Fig. 3.1. The code was originally implemented in a MATLAB environment and includes four unique models to achieve minimum induced loss design *(the yellow line)* or to analyze the performance of an existing propeller *(the blue line)*, as seen in Fig. 3.1. Three of these methods were discussed in the theory section: the LArrabee method 2.3.1, the Adkins method 2.3.2, and the Drela method 2.3.3. The fourth method, known as Capitao Patrao, will not be included in this thesis.

The user should provide the program with some parameters as follows:

- **Geometric variables**: Number of blades, diameter of the propeller, and hub-to-tip ratio (HTR).
- **Performance target for the design**: The potential options are thrust coefficient (C_T), power coefficient (C_P), thrust (T), power (P), or `'noSwirl'`.
- **Atmospheric conditions**: Height above sea level (H) and temperature difference (dT).
- **Engine conditions**: Free stream axial Mach number (M_{axial}) (the value should be between 0.3 and 1.1) and rotational velocity (rpm).
- **Design options**: Specify whether the design should run with a specific chord value or with a lift coefficient (c_l) value (the value should be chosen between 0 and 0.5).
- **Thickness**: Should be chosen between 4% and 21%.
- **Airfoil camber**.
- **Blade loading**: Starting guess if the chord is pre-specified, or actual loading if the chord is a variable (c_{Ld}).
- **Blade mechanical parameters**: Density of the blade material [kg/m^3].
- **Upstream flow conditions**: Axial and tangential upstream velocity for Drela and Patrao methods.

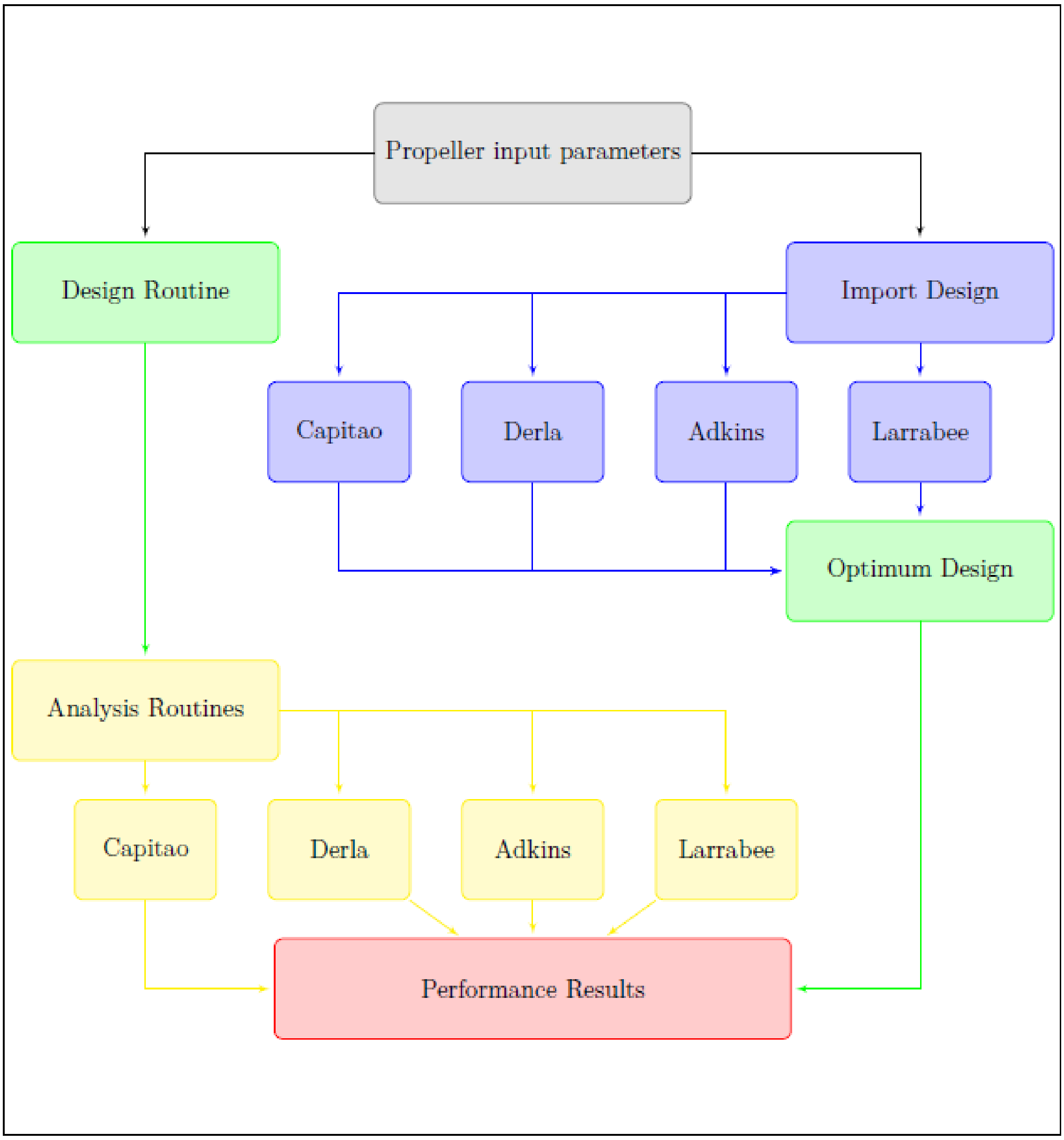


Figure 3.1: Optoprop code structure.

3.1.1 Drela Model for minimum induced loss - Design

Calculate the local externally-induced velocities at the disk, U_a, U_t, and U, which are required to determine $\psi_{Initial}$.

1. The first step is to choose an initial value for $\bar{\eta}$ together with an initial geometry. To reduce the process time, the Bisection method will be implemented, and the Larrabee method will be used to set the β and c values. The value of $\bar{\eta}$ is used to calculate the general residual, which could be C_T, C_P, thrust, or power, as follows:

$$R_T = T_c(\bar{\eta}) - T_{c,spec} \tag{3.1}$$

$$R_P = P_c(\bar{\eta}) - P_{c,spec} \tag{3.2}$$

2. Perform an iterative process to calculate the desired value of $\bar{\eta}$ by calculating the velocities U_a, U_t, and U using Eqs. 2.61, 2.62, 2.63, and then calculate the dummy variable ψ using Eq. 2.64. Afterwards, compute the velocity triangle W_a, W_t, and W using Eqs. 2.65, 2.66, 2.67:

$$R_{\bar{\eta}} = \overline{\eta} - \frac{VW_T}{\Omega r W_a} \to 0$$

3. The values of W_a and W_t depend on the local variable β in the case of a prespecified chord and on two variables in the case of a prespecified c_l, so the algorithm is divided into two paths:
   - **Prespecified chord**: In this path, the program uses the Newton-Raphson method to find the values of β and $\bar{\eta}$ as follows:

   $$\delta\beta = -\frac{R_{\bar{\eta}}}{\partial R_{\bar{\eta}}/\partial\beta}$$

   - **Prespecified** C_l: In this path, the program will find the values of c and β using the Newton-Raphson method:

   $$\begin{bmatrix} \delta\beta \\ \delta c \end{bmatrix} = \begin{bmatrix} \frac{\partial R_{C_l}}{\partial\beta} & \frac{\partial R_{C_l}}{\partial c} \\ \frac{\partial R_{\overline{\eta}}}{\partial\beta} & \frac{\partial R_{\overline{\eta}}}{\partial c} \end{bmatrix}^{-1} \cdot \begin{bmatrix} R_{C_l} \\ R_{\overline{\eta}} \end{bmatrix}$$

   Then, it will calculate the values of C_l and $\bar{\eta}$:

   $$R_{\bar{\eta}} = \overline{\eta} - \frac{VW_T}{\Omega r W_a} \to 0$$
   $$R_{C_l} = C_l - C_{l,spec} \to 0$$

4. Calculate the performance for the current geometry and the value of the residual $R_{\bar{\eta}}$. Perturb β, calculate the derivative of $R_{\bar{\eta}}$, and compute a new value for β. Iterate until convergence is reached for β, and calculate the final thrust T.
5. Compare the obtained thrust with the target thrust, and increase $\bar{\sigma}$ if the obtained thrust is too high, or decrease $\bar{\sigma}$ if the obtained thrust is too low.
6. Go back to step 2 and iterate until $\bar{\sigma}$ has converged.

3.1.2 Drela Model for Minimum Induced Loss - Analysis

1. Start by calculating the initial guess for the dummy variable ψ, since the values of the locally externally induced velocity at the disk and the free-stream velocity are known using the following equations:

$$U_a = V + u_a$$
$$U_t = \Omega r - u_t$$
$$U = \sqrt{U_a^2 + U_t^2}$$
$$\psi_{initial} = \arctan\left(\frac{U_a}{U_t}\right)$$

2. The next step is to calculate the velocity triangle and associated values (Re, Ma) using the guessed value of ψ:

$$W_a = \frac{1}{2}U_a + \frac{1}{2}U\sin(\psi)$$
$$W_t = \frac{1}{2}U_a + \frac{1}{2}U\cos(\psi)$$
$$W = \sqrt{W_t^2 + W_a^2} \rightarrow Re, Ma$$

3. Furthermore, calculate the flow angle ϕ, angle of attack α, F factor, and associated values such as ($v_a = W_a - U_a$, $v_t = U_t - W_t$):

$$\phi = \arctan\left(\frac{W_a}{W_t}\right)$$
$$\lambda_w = \frac{r}{R}\frac{W_a}{W_t}$$
$$\alpha = \beta - \phi$$

4. With α known, calculate c_l and c_d.
5. Calculate the circulation Γ:

$$\Gamma = v_t \frac{4\pi r}{B} F \sqrt{1 + \left(\frac{4\lambda_w}{\pi\xi B}\right)^2}$$

6. Calculate the residual for ψ and iterate until convergence. Steps 2–4 will be repeated:

$$R_\psi = \Gamma - \frac{1}{2} W\, C\, C_l \rightarrow 0 \tag{3.3}$$

7. Once ψ is known, calculate thrust, torque, and efficiency.

3.1.3 Drela Model for No Swirl

The velocity U_a can be calculated by Eq. 2.62, and it can also be expressed as $U_a = \Omega r(1 - \alpha'_\infty)$, where α'_∞ is the tangential induced factor for the upstream. To cancel the swirl, the values of α'_∞ and α'_∞ must be equivalent in magnitude but in opposite directions. For this scenario, the option to set thrust, torque, and general efficiency will not be possible. The iteration loop will proceed as follows:

3.1.3.1 Prespecified Chord

$$R_{swirl} = \frac{a'}{-a'_\infty} - 1 \rightarrow 0$$

The local variable here is β. Newton-Raphson method will be used as mentioned before:

$$\delta\beta = -\frac{R_{\overline{\eta}}}{\frac{\partial R_{\overline{\eta}}}{\partial \beta}}$$

3.1.3.2 Prespecified C_l

In addition to the residuals for canceling the swirl, R_{swirl}, the residual for C_l is also needed:

$$R_{swirl} = \frac{a'}{-a'_\infty} - 1 \to 0$$

$$R_{C_l} = C_l - C_{l,\text{specified}} \to 0$$

The local variables here are β and c. The Newton-Raphson method is also implemented here to reduce computation time:

$$\begin{bmatrix} \delta\beta \\ \delta c \end{bmatrix} = \begin{bmatrix} \frac{\partial R_{C_l}}{\partial \beta} & \frac{\partial R_{C_l}}{\partial c} \\ \frac{\partial R_{swirl}}{\partial \beta} & \frac{\partial R_{swirl}}{\partial c} \end{bmatrix}^{-1} \cdot \begin{bmatrix} R_{C_l} \\ R_{swirl} \end{bmatrix}$$

3.2 Acoustic Extension

The user can choose to perform an acoustic analysis by selecting "True" in the input file of the student design. The extension works across all routines of `OPTOPROP` and extracts values such as the number of blades β, diameter D, axial Mach number M, speed of sound a, revolutions per minute (RPM), angular velocity of the propeller Ω, thrust T, and torque Q. The user can decide whether to include time in the calculation of P_{rms} or to use the simplification assumed by Garrick and Watkins. The extension will output the values of both SPL and P_{rms} for both near-field and far-field conditions as follows:

3.2.1 Near Field

- The first step is to choose the boundary of the field. The chosen coordinate system is polar, where the propeller is located at the center, and the observers are located on a circle with a radius of $2D$, as shown in Fig. 3.2:

$$\begin{aligned} x &= 2D\cos\phi \\ y &= 2D\sin\phi \\ z &= 0 \end{aligned}$$

- The code calculates the effective radius $R_e = 0.4D$.
- The location of the propeller, or the location of the force, is given by x_l, y_l, and z_l.
- The p_{rms} is calculated using Eq. (2.137) for the simplified case, or by using Eqs. (2.134) and (2.135) when including time.
- The values of S_0, σ, and β are calculated using Eqs. (2.136) and (2.129), respectively.

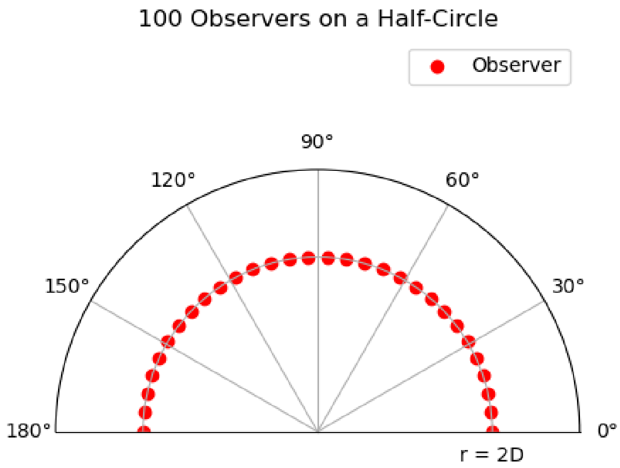


Figure 3.2: Observer location in the near field.

3.2.2 Far Field

1. The chosen coordinate system is Cartesian, where the propeller is located $100D$ above the observer's line, and the observers are located on the x-axis at $[-100D, 100D]$, as seen in Fig. 3.3

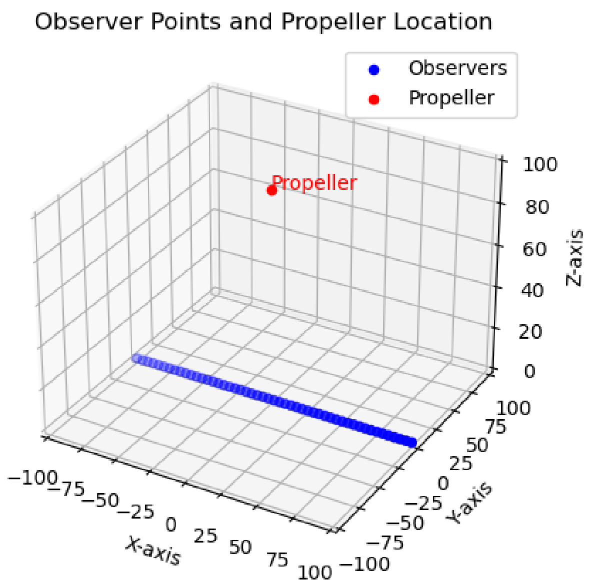


Figure 3.3: Observer location in the far field.

2. The code calculates the effective radius $R_e = 0.4D$.
3. The location of the propeller or the force is given by x_l, y_l, and z_l:

$$x_l = 0 \tag{3.4}$$
$$y_l = 0 \tag{3.5}$$
$$z_l = 100D \tag{3.6}$$

4. The p_{rms} is calculated using Eq. (2.121) for the simplified case, or by using Eqs. (2.121), (2.122), and (2.123), or by Eqs. (2.126) and (2.125) when including time.
5. The values of S, σ, and β are calculated using Eqs. (2.127), (2.128), and (2.129), respectively.

4

Results

4.1 Drela model

4.1.1 Validation of Drela implementation

To ensure the accuracy of the Python implementation of Drela's method, a comparison with the MATLAB implementation was conducted. The details of the propeller `NACA16` are provided in Table 4.1.

Blade Count	Diameter (m)	HTR	Height (m)	M_{Axial}	$C_{T_{SPEC}}$
2	0.254	0.177	1000	0.35	0.35

Table 4.1: Parameters of the tested propeller `NACA16`

The results of thrust, power, and efficiency obtained using different methods are shown in Table 4.2.

	T (N)	P (W)	η (%)	Discloading (N/m^2)
Larrabee Python Design	2.1774	357.2	71.7763	5536.7
Larrabee Python Analysis	1.5971	266.4	70.6045	5581.8
Adkins Python Design	2.1774	372.3	68.8705	5770.3
Adkins Python Analysis	2.1842	373.0	68.9607	5780.8
Drela Python Design	2.1759	334.2	76.66	5179.8
Drela Python Analysis	2.1764	334.2	76.67	5180.5
Drela MATLAB Design	2.1759	334.2	76.66	5179.8
Drela MATLAB Analysis	2.1764	334.2	76.67	5180.5

Table 4.2: Comparison between the Drela method code in Python and the Larrabee, Adkins, and Drela methods in MATLAB for both design and analysis routines

The differences between the Larrabee, Adkins, and Drela methods arise because Drela assumes constant circulation instead of using momentum equations in the wake region. However, the differences between MATLAB and Python implementations are negligible, indicating that the Python code is working correctly. Another set of data from the Capitao Patrao report [18], where OPTOPROP was used to examine the `GPS621` propeller, is shown in Table 4.3.

Blade Count	Diameter (m)	M_{Axial}	V_{Free} (m/s)	RPM	Thrust (N)	$C_{T_{SPEC}}$
8	0.75	0.75	222.4	4997	333.2	0.4

Table 4.3: Parameters of the `GPS621` propeller [18]

The results of applying Drela's method are presented in Table 4.4.

	Thrust (N)	Power (W)	Efficiency (%)
Drela Python	334.9	96.6	77.1
Drela MATLAB	334.9	96.6	77.1
Error	0	0	0

Table 4.4: Comparison between the Drela method code in Python and Drela methods in MATLAB for the `GPS621` propeller

4.1.2 Effects of parameters

In this section, a sensitivity study will be conducted for several parameters, including Diameter (D), blade count (B), angular speed (RPM), and Mach number (M), to illustrate their effects on the thrust and efficiency of the propeller. The analysis aims to provide guidelines for achieving a propeller with high performance and low noise.

The parameters of two different propellers used in this study are shown in Table 4.5. The atmospheric conditions and engine parameters are provided in Table 4.6.

Blade Count	Diameter (m)	HTR	$C_{T_{SPEC}}$
6	2.2	0.25	0.137
8	2.0	0.30	0.20

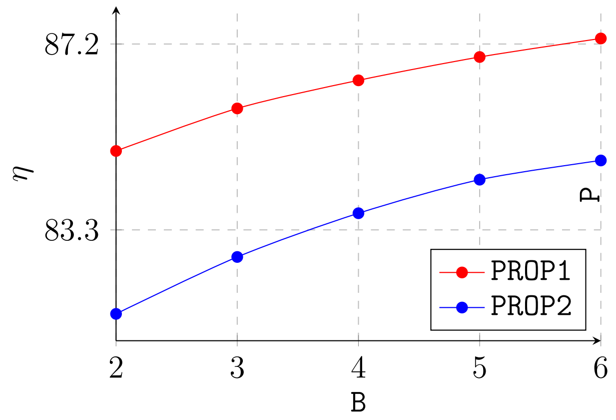
Table 4.5: Parameters of the tested propellers

Height (m)	Mach Number	RPM
5150.62	0.35	2200

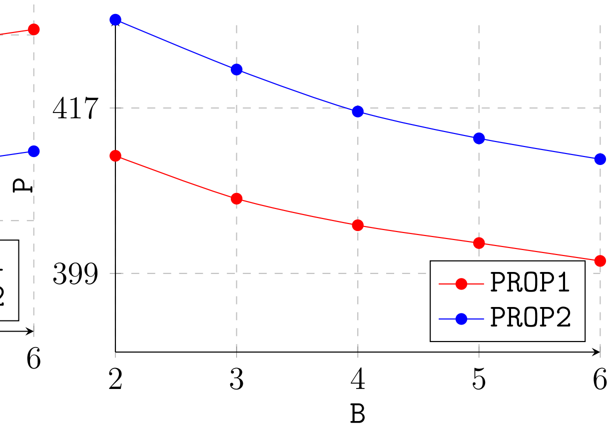
Table 4.6: Atmospheric conditions and engine parameters

4.1.2.1 The Effects of Blade Number on Efficiency, Power, and Torque

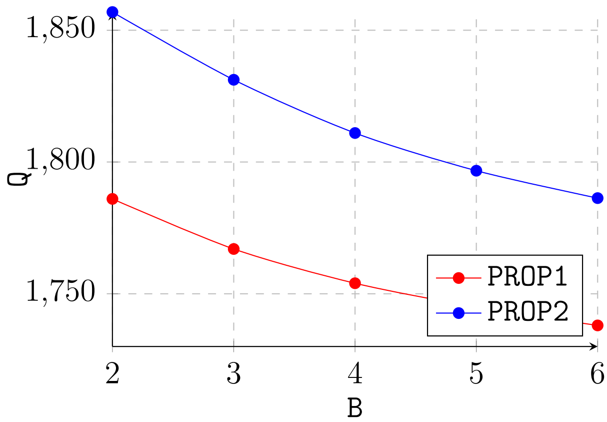
The values of blade numbers used are $B = [2, 3, 4, 5, 6]$.



(a) The effect of blade number on efficiency



(b) The effect of blade number on power



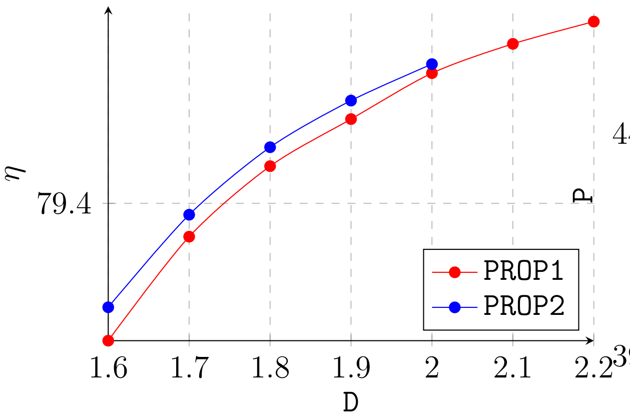
(c) The effect of blade number on torque

Figure 4.1: Effects of blade number on efficiency, power, and torque of the propeller

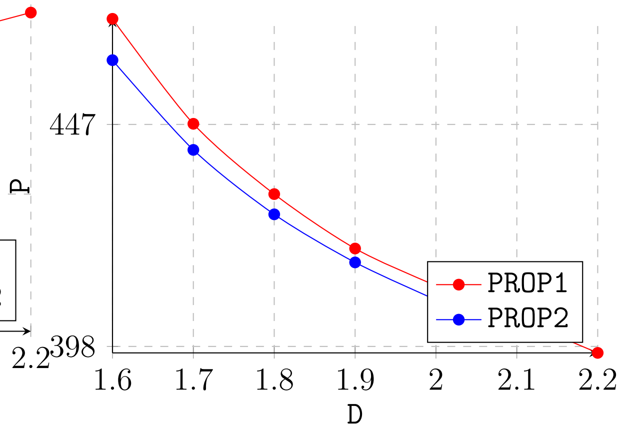
Fig. 4.1a, Fig. 4.1b, and Fig. 4.1c show that increasing the number of blades enhances the efficiency while decreasing the torque and power until reaching an optimal efficiency. These results are consistent with theoretical expectations.

4.1.2.2 The effect of diameter on efficiency, power, and torque

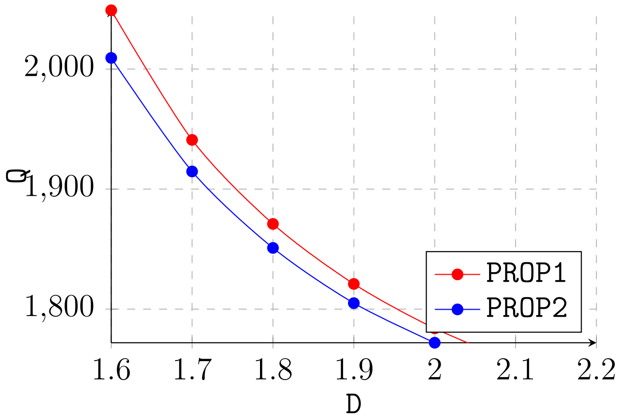
The values of diameters used are $D = [0.254, 0.5, 1, 1.5, 2, 2.5, 3]$ for the first propeller, and $D = [0.254, 0.5, 1, 1.5, 2]$ for the second propeller.



(a) The effect of diameter on efficiency



(b) The effect of diameter on power



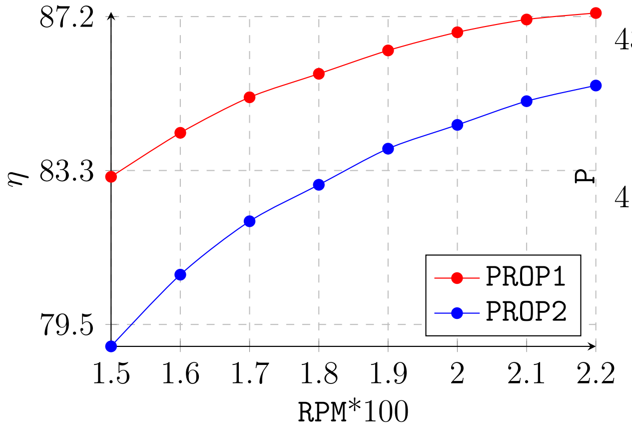
(c) The effect of diameter on torque

Figure 4.2: Effects of diameter on efficiency, power, and torque of the propeller

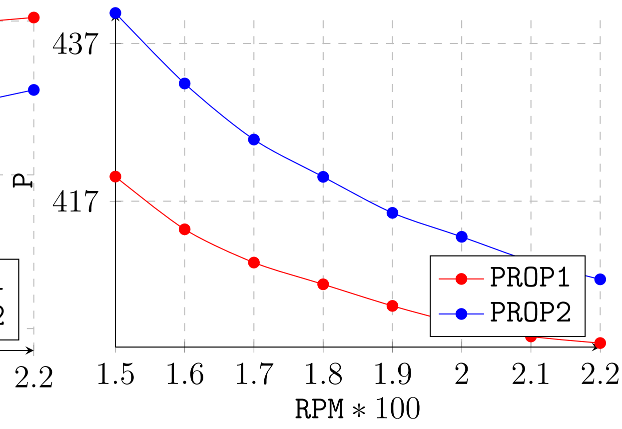
The plots reveal that efficiency increases with diameter up to an optimal point, as shown in Fig. 4.2a. Meanwhile, torque and power decrease with increasing diameter, as illustrated in Fig. 4.2b and Fig. 4.2c.

4.1.2.3 The effects of angular Speed on efficiency, power, and torque

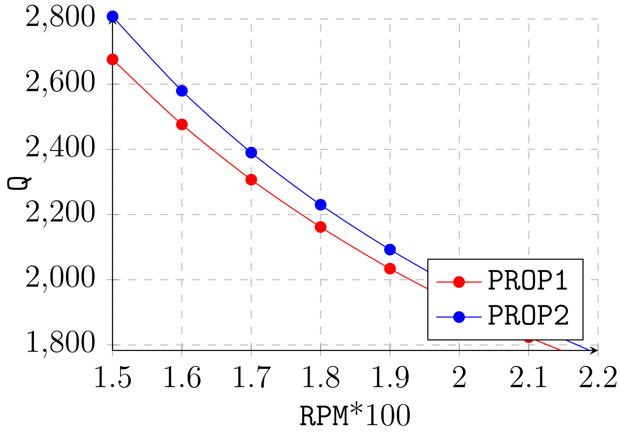
The values of angular speeds used are rpm = [1500, 1800, 2000, 2200] for both propellers. The graph shows that as angular speed increases, efficiency rises proportionally, demonstrating a positive linear relationship with a steep incline, as shown in Fig. 4.3a. In contrast, Fig. 4.3b and Fig. 4.3c illustrate that power and torque decrease consistently with increasing angular speed.



(a) The effect of angular speed on efficiency



(b) The effect of angular speed on power



(c) The effect of angular speed on torque

Figure 4.3: Effects of angular speed on efficiency, power, and torque of the propeller

4.2 Acoustic model

4.2.1 Validation of Garrick and Watkins implementation

To validate the extension, data calculated using Computational Fluid Dynamics (CFD) and the Ffowcs Williams and Hawkings (FW-H) equations are compared with results presented in a thesis from Chalmers [39]. The details of the propeller used are listed in Table 4.7.

Blade Count	6	
Diameter	2.2	m
Rotational Speed	2200	RPM
Flight Mach Number	0.35	-
Speed of Sound	319.9125	m/s
Efficiency	88.0193	%
Thrust	3125.4	N
Torque	1725.619	Nm

Table 4.7: Propeller details

The data for p_{rms} and SPL in both the nearfield and farfield are presented in an appendix of [39]. Fig. 4.4, Fig. 4.5, Fig. 4.6 and Fig. 4.7 show the calculated results from the extension compared to the referenced data.

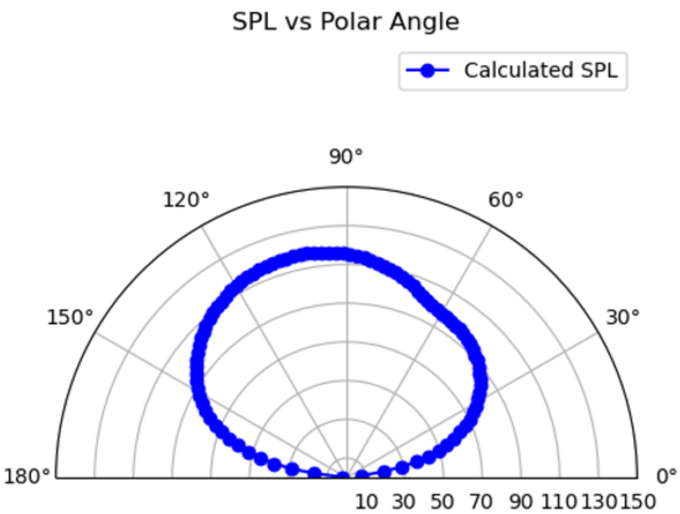


Figure 4.4: SPL in the near field

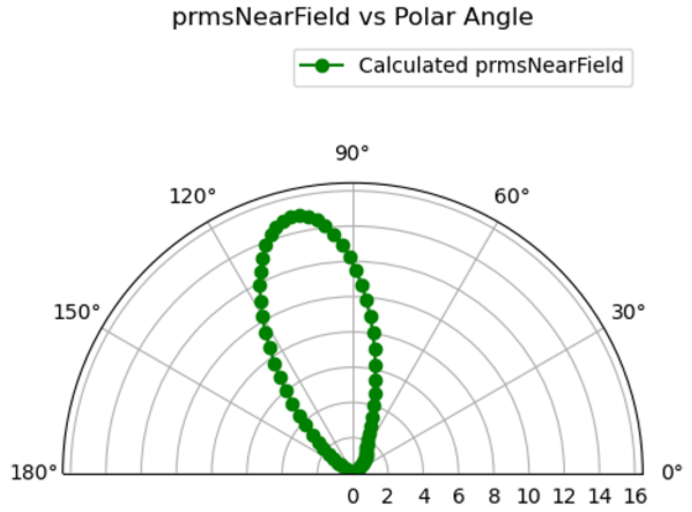


Figure 4.5: P_{rms} in the near field

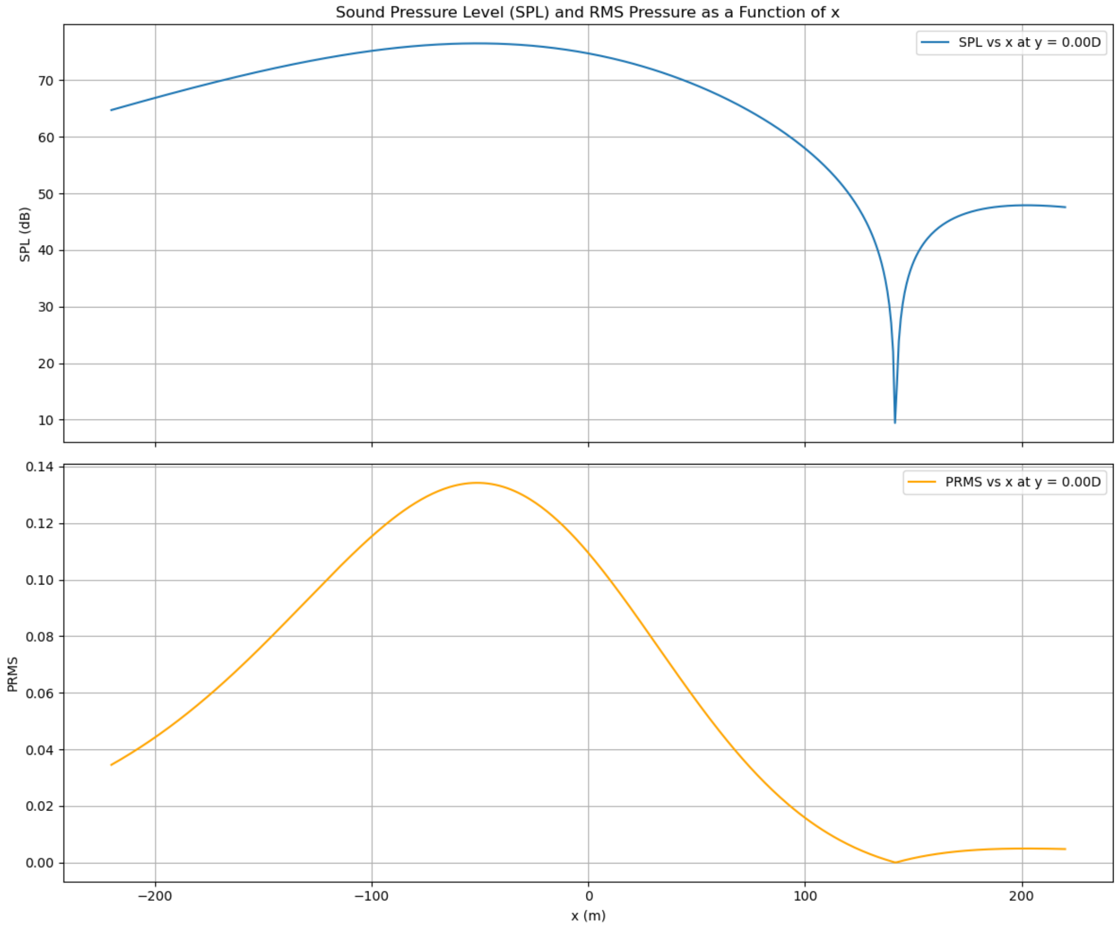


Figure 4.6: P_{rms} and SPL in the far field

The results are sufficiently close to the referenced data, validating the extension.

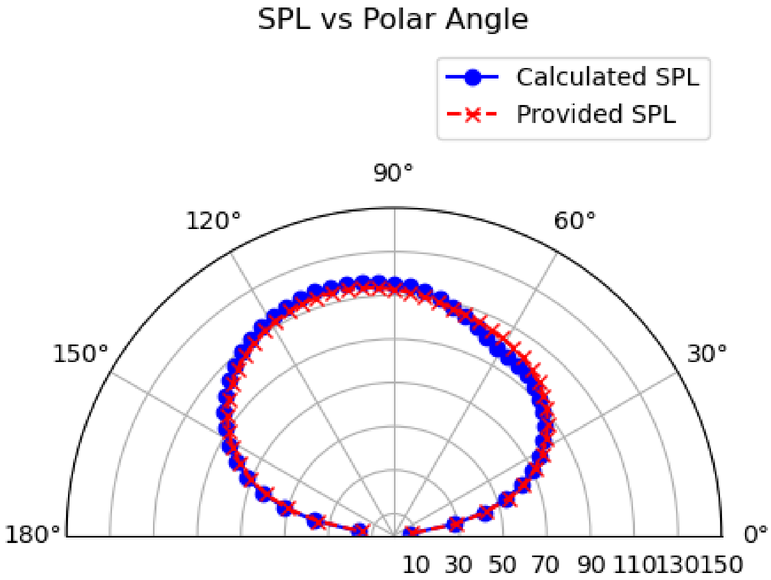
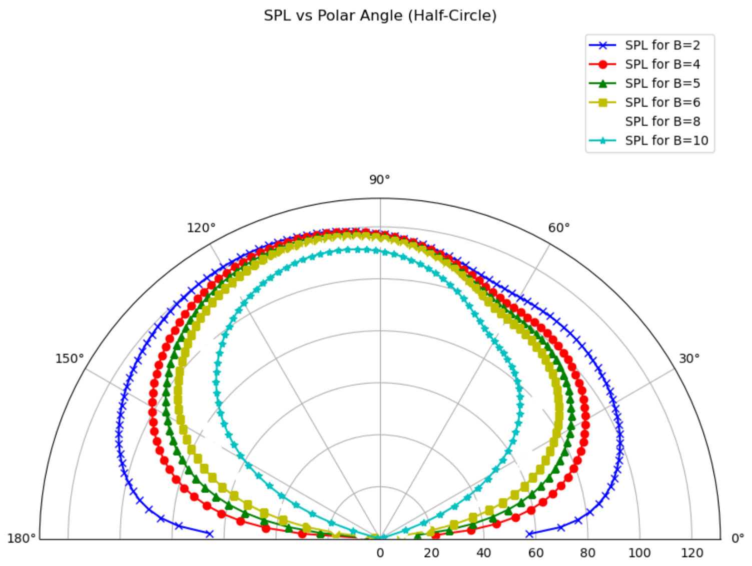


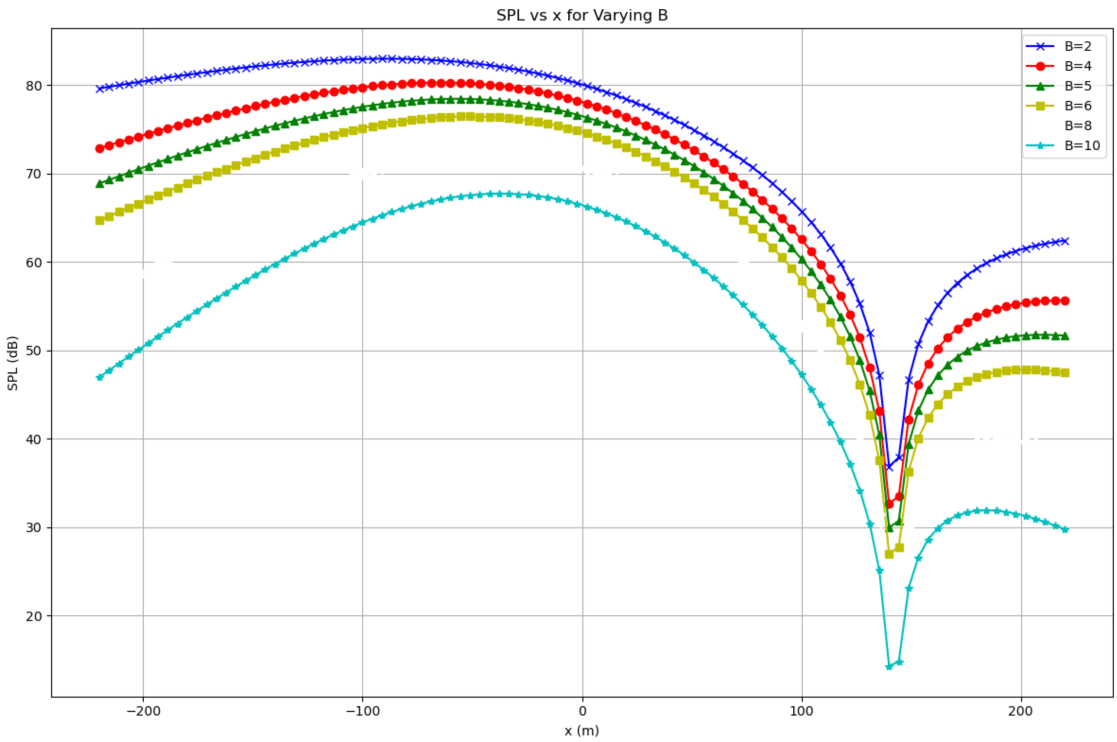
Figure 4.7: Comparison of SPL calculated using CFD+FW-H and results from the code

4.2.1.1 The effect of blade count, diameter and rotational speed on SPL

The values of blade counts that have been used are $B = [\,2,\ 3,\ 4,\ 6,\ 8,\ 10\,]$. In the second case, the values of diameters that have been used are $D = [\,1.5,\ 2,\ 2.2,\ 2.5,\ 2.8,\ 3\,]$, while the values of thrust used are $T = [\,1500,\ 2000,\ 2500,\ 3000,\ 3500,\ 4000\,]$.Finally the last sensitivity study will be done for angular speed $RPM = [\,1500,\ 1800,\ 2000,\ 2200,\ 2500,\ 3000\,]$.

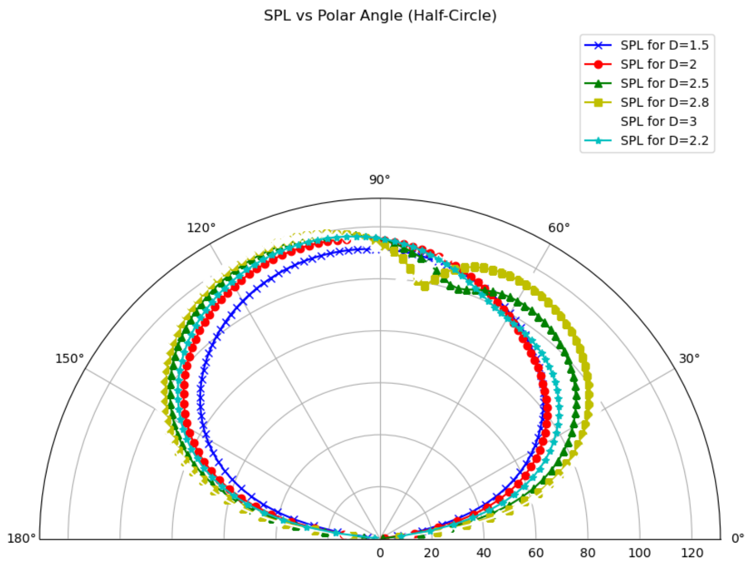


(a) Blade Counts effect on SPL in near field

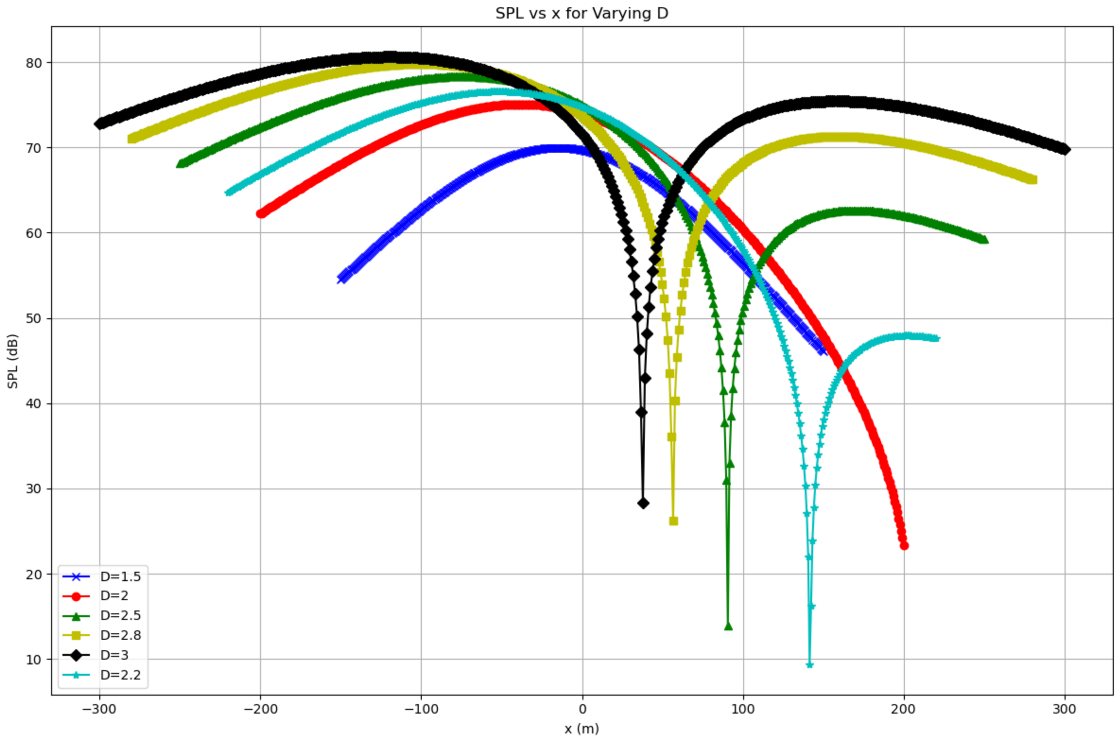


(b) Blade Counts effect on SPL in far field

Figure 4.8: blade counts effect on SPL

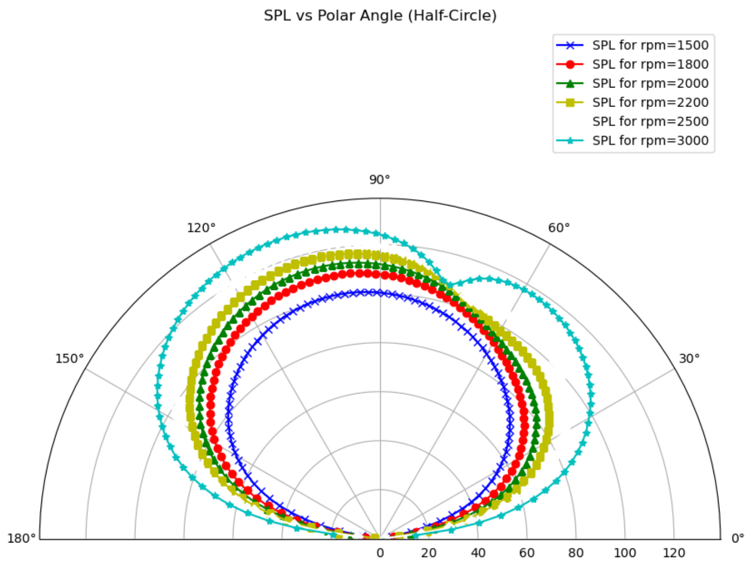


(a) diameter effect on SPL in near field

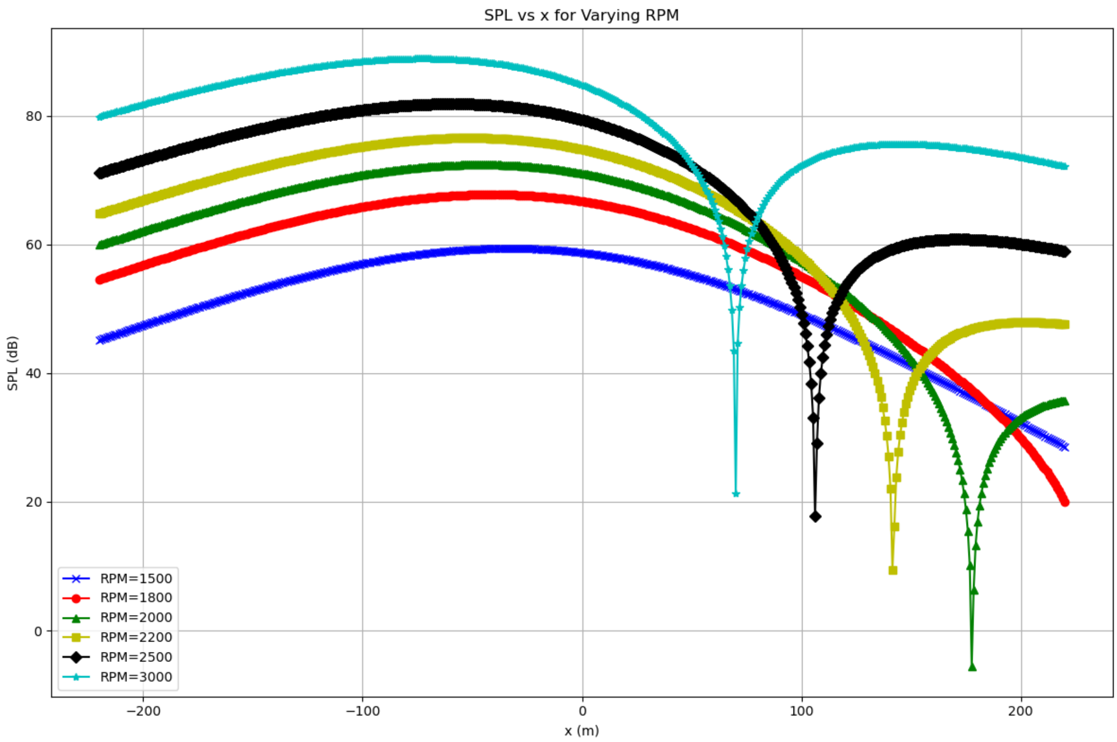


(b) Diameter effect on SPL in far field

Figure 4.9: Diameter effect on SPL

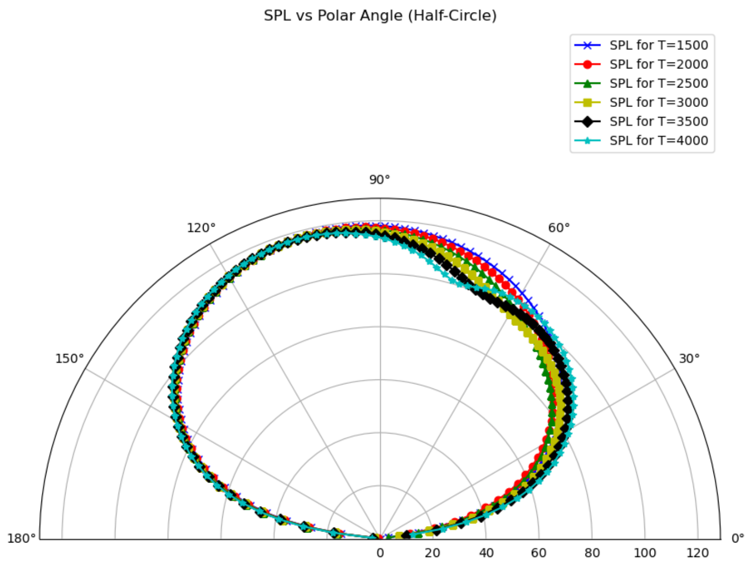


(a) Rotational speed effect on SPL in near field

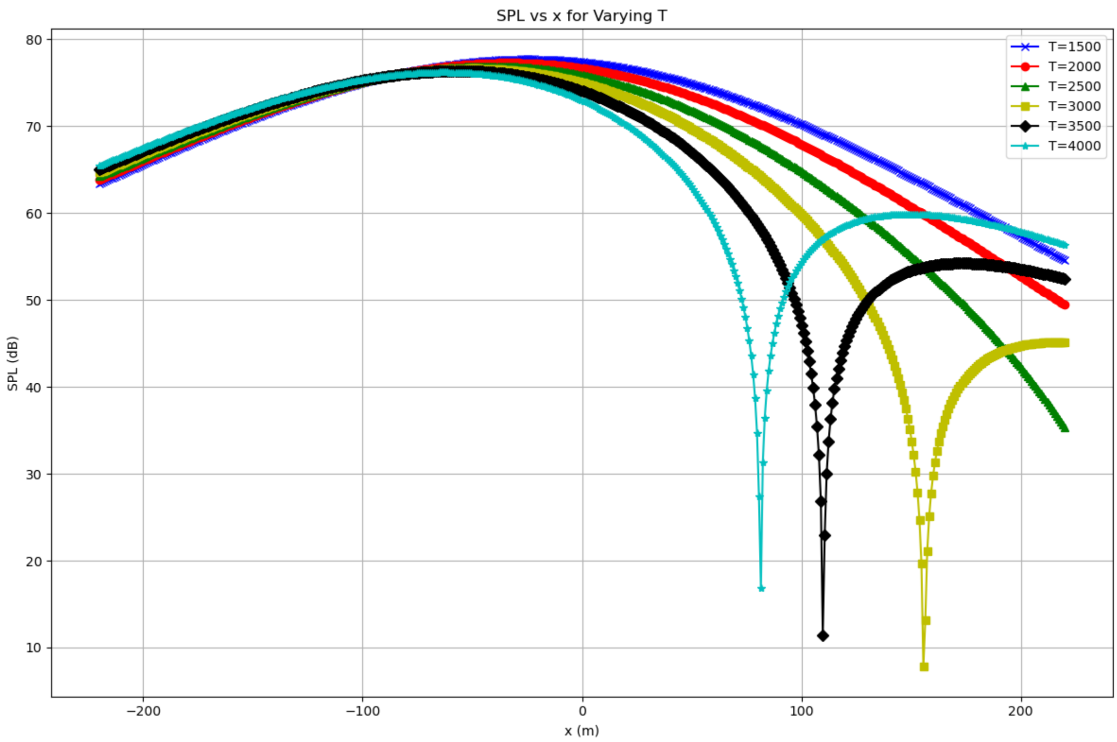


(b) Rotational speed effect on SPL in far field

Figure 4.10: Rotational speed effect on SPL



(a) Thrust effect on SPL in near field



(b) Thrust effect on SPL in far field

Figure 4.11: Thrust effect on SPL

Obviously, an increase in the number of blades leads to a reduction in SPL in both the nearfield 4.8a and farfield 4.8b. This pattern also applies to the diameter, which has an indirect relationship with SPL. This means that we can reduce the noise in both the farfield 4.9a and nearfield 4.9b by increasing the diameter of the propeller. RPM is the only parameter that reverses this behavior; in other words, SPL increases or decreases with RPM in both nearfield 4.10a and farfield 4.10b . Generally, if thrust increases for any reason other than an increase in engine angular speed, SPL will be lower.

5

Conclusion

5.1 Discussion on the effects of parameters on performance and noise footprint

As discussed in the results section, the diameter of the propeller and the blade count are crucial factors affecting efficiency. Increasing the propeller diameter generally leads to higher efficiency, reduced power consumption, and lower generated torque. However, it also influences the sound pressure level (SPL), which shows an inverse relationship with diameter. These findings suggest that optimizing propeller diameter and blade count can significantly enhance design performance and reduce noise levels. There is a threshold for both parameters beyond which further increases lead to diminished returns and potential negative impacts on performance. Conversely, while increasing angular speed improves performance up to a certain optimal point, it adversely affects SPL. Thus, if low noise is a priority, increasing angular speed should be done with caution to avoid excessive noise levels.

5.2 Conclusion

- The implementation of the Python code was successful, demonstrating robust functionality and producing results that closely match those obtained from the MATLAB implementation. This confirms the accuracy of the developed Python model in simulating the relevant aerodynamic and performance characteristics.
- Furthermore, we confirm the well-known trend that increasing the propeller diameter generally leads to improved performance. Specifically, larger diameters tend to enhance propeller efficiency, reduce the power required to achieve a given thrust, and lower the generated torque.
- The findings related to the effect of blade number and angular speed are also consistent with existing aerodynamic theory and align well with prior experimental and numerical studies in the field.
- Finally, the results of the acoustic analysis show good agreement with the original referenced data, further validating the model's predictive capability. The accuracy of the simulation in this aspect confirms its usefulness for noise estimation and related studies.

5.3 Future work

Several recommendations for future research are as follows:

- Incorporate the Capitao Patrao method, which addresses external induced loss effects for both upstream and downstream flows. This method could provide more accurate results compared to the Drela method, which assumes constant circulation. Alternatively, consider using Adkins' method, which integrates the momentum equation for a more comprehensive analysis.
- Explore additional airfoil types by collecting and analyzing more data to assess their impact on performance and noise.
- Although increasing diameter does not significantly affect noise compared to RPM, the `OptoProp` acoustic study incorporates parameters such as diameter and HTR to optimize design and SPL. Implementing linear programming to set SPL as a constraint will allow for the specification of desired SPL values to determine the appropriate diameter for an optimal design. This approach aims to balance performance with a low noise footprint.
- Investigate the effect of blade thickness in the extended acoustic study to understand its influence on both performance and noise.

Bibliography

[1] Zondervan, GJD, *A review of propeller modelling techniques based on Euler methods*, *Series 01: Aerodynamics 05*, 1998, Delft University Press.

[2] Our World In Data (OWID), What share of global CO emissions comes from? aviation. *https://ourworldindata.org/global-aviation-emissions*.

[3] Kreimeier, M., Stumpf, E., Gottschalk, D., *Economical Assessment of Air Mobility on Demand Concepts with Focus on Germany*, in: 16th AIAA Aviation Technology, Integration, and Operations Conference, American Institute of Aeronautics and Astronautics, Washington, D.C. (2016). `https://doi.org/10.2514/6.2016-3304`.

[4] Fleming, Gregg G. (USA DOT Volpe), de Lépinay, Ivan (EU EASA), and Schaufele, Roger (USA FAA), *Environmental Trends in Aviation to 2050* International Civil Aviation Organization-ICAO, 2024.

[5] Paris Agreement, *United Nations Framework Convention on Climate Change*, T.I.A.S. No. 16-1104, Dec. 12, 2015.

[6] United States, *2021 Aviation Climate Action Plan*, U.S. Department of Transportation, 2021.

[7] International Civil Aviation Organization (ICAO), *Sustainable Aviation Fuels (SAF)*, 2021. Retrieved from https://www.icao.int/environmental-protection/pages/SAF.aspx.

[8] Airbus, *ZEROe: Towards the World's First Zero-Emission Commercial Aircraft*, 2020. Retrieved from https://www.airbus.com/en/innovation/zero-emission/hydrogen/zeroe.

[9] Boeing, *Advanced Composite Materials in Boeing Aircraft*, Aero Magazine, 2019. Retrieved from

[10] Langley, R., & McCormick, B. W., *Optimization of Propeller Design for Improved Efficiency*, Journal of Aircraft, 51(6), 2033-2045, 2014. Retrieved from https://arc.aiaa.org/doi/10.2514/1.C032559.

[11] World Health Organization (WHO) Europe, *Environmental Noise Guidelines for the European Region*, 2018. Retrieved from `https://www.euro.who.int/en/health-topics/environment-and-health/noise/publications/2018/environmental-noise-guidelines-for-the-european-region`.

[12] European Commission, *Noise*, Year. Retrieved from `https://environment.ec.europa.eu/topics/noise_en`.

[13] Adkins, Charles N and Liebeck, Robert H, *Design of optimum propellers*, *Journal of Propulsion and Power*, **10**(5), 676–682, 1994.

[14] Zondervan, GJD, *A review of propeller modelling techniques based on Euler methods*, *Series 01: Aerodynamics 05*, 1998, Delft University Press.

[15] Birajdar, Mahasidha R., and Sandip A. Kale, *Effect of leading edge radius and blending distance from leading edge on the aerodynamic performance of small wind turbine blade airfoils*, 2015, International Journal of Energy and Power Engineering, 4(5-1), pp. 54–58.

[16] Larrabee, E Eugene, *Practical design of minimum induced loss propellers*, *SAE Transactions*, 2053–2062, 1979, JSTOR.

[17] Larrabee, E Eugene, *Five years experience with minimum induced loss propellers-part I: Theory*, *SAE Transactions*, 1130–1137, 1984, JSTOR.

[18] Patrao, ALEXANDRE CAPITAO, *Implementation of blade element momentum/vortex methods for the design of aero engine propellers*, *Research report 2017: 06, Mechanics and Maritime Sciences (M2) Chalmers*, 2017.

[19] Drela, Mark, *QPROP formulation*, *Massachusetts Inst. of Technology Aeronautics and Astronautics, Cambridge, MA*, 2006.

[20] Drela, Mark, *A User's Guide to MTFLOW 2.01*, *Massachusetts Institute of Technology, Cambridge, MA*, 2010.

[21] Dolzhansky, F.V. and Dolzhansky, F.V., *Equations of Motion of an Ideal Incompressible Fluid; Kelvin's Circulation Theorem*, Fundamentals of Geophysical Hydrodynamics, pp. 3-11, 2013.

[22] Rohrlich, F., *The Validity of the Helmholtz Theorem*, American Journal of Physics, 72(3), pp. 412-413, 2004.

[23] Breuer, K., Drela, M., Fan, X. and Di Luca, M., *Design and performance of an ultra-compact, low-speed, low turbulence level, wind tunnel for aerodynamic and animal flight experiments*, Experiments in Fluids, 63(11), p.169, 2022.

[24] Gibbs-Smith, Charles Harvard and others, *Sir George Cayley's Aeronautics, 1796-1855*, 1962, HM Stationery Office.

[25] Howard, Fred, *Wilbur and Orville: a biography of the Wright brothers*, 2013, Courier Corporation.

[26] Dixon, S Larry and Hall, Cesare, *Fluid mechanics and thermodynamics of turbomachinery*, 2013, Butterworth-Heinemann.

[27] Make, Michel, *Predicting scale effects on floating offshore wind turbines*, *MSc Thesis*, 2014.

[28] Goldstein, Sydney, *On the vortex theory of screw propellers*, *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, **123**(792), 440–465, 1929, The Royal Society London.

[29] Garrick, IE and Watkins, Charles E, *A theoretical study of the effect of forward speed on the free-space sound-pressure field around propellers*, 1953.

[30] Petrov, A.G., Sukhov, A.D., Sibgatullin, I.N. and Britov, A.D., "Analytical and numerical methods for Zhukovsky airfoils aerodynamics coefficients," in *2022 Ivannikov Ispras Open Conference (ISPRAS)*, pp. 62-64, December 2022, IEEE.

[31] Branlard, E., Dixon, K., & Gaunaa, M., *Vortex methods to answer the need for improved understanding and modelling of tip-loss factors*, *IET Renewable Power Generation*, *7*(4), pp. 311-320, 2013.

[32] Goldstein, S., *On the vortex theory of screw propellers*, *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, *123*(792), pp. 440-465, 1929.

[33] Theodorsen, T., *Theory of propellers*, McGraw-Hill Book Company, Inc., 1948.

[34] F. Bruce Metzger *A review of propeller noise prediction methodology 1919-1994*

[35] Hubbard, H.H. ed., *1991. Aeroacoustics of flight vehicles: theory and practice (Vol. 1). National Aeronautics and Space Administration, Office of Management, Scientific and Technical Information Program.*

[36] Garrick, I. E. and Watkins, C. E., *A Theoretical Study of the Effect of Forward Speed on the Free-Space Sound-Pressure Field Around Propellers*, Tech. rep., *NACA*, October 1953. [PDF]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19930092212.pdf, accessed: 2020-02-04.

[37] Lamb, H., *Hydrodynamics*, Sixth ed., *Cambridge University Press*, 1932, p. 502.

[38] Gutin, L., *On the Sound Field of a Rotating Propeller*, Tech. rep., *NACA TM llg5*, 1948. (From Phys. Z. der Sowjetunion, Bd. 9, Heft 1, 1936, pp. 57-71).

[39] Haidari, M.H., Hövik, F., Lago, C., Lindé, F.N. and Sjöberg, J., *Low-Noise Propeller Design for Future Electric Aircraft*, 2020.

A

Appendix 1

```python
import numpy as np
from pathlib import Path

# I - Input ############################################################
#Program task:
caseName = 'propeller_20221025_ACP'
programMode = 'design'                     #either 'design' or 'analysis'"
specRunType = 'DL'                         #Design Adkings: DA, Design Larabee: DL, Design Drela : DD
                                           #Analysis Adkings: AA, Analysis Larabee: AL,Analysis Drela: AD
propellerForAnalysisPath = Path.cwd() / 'output' / 'propeller_20221025_ACP_propellerDesign.json'

#Geometry
B = 5
nPoints = 25
D = 10*25.4e-3
HTR =  0.9*50e-3/D #50e-3                          #actual hub Diameter is 50 mm but we scaling it down a bit

#TARGET DESIGN PERFORMANCE
preSpecifiedPerformance = 'CT'      #Specify 'CT', 'CP', 'thrust', 'power', or 'noSwirl' (only available for drela methods)
targetPerformance =   0.3                    #In terms of either CT, CP, thrust [N], power [W], None if noSwirl

#atmospheric conditions
H =  0                                   #height above sea level
dT = 0                                         #temp change

#Engine conditions
M_axial = 30/340.29                        #freestream axial mach number
rpmChoice = 'rpm'                          #either 'rpm', 'tipMach', or 'J'
tipRelMach = 0.35
rpm = 8000                                 #rotational velocity

#Upstream flow conditions
axialVelocityUpstream = 0*np.ones(nPoints)
tangentialVelocityUpstream = 0*np.ones(nPoints)
```

```

#AIRFOIL SPEC
#airfoil type
airfoilChoice = 'NACA16'                        #'NACA_16' 'noDrag'

#airfoil chord
preSpecChord = False                            #Is chord pre-specified? True or False
chordChoice = 'linear'                          #'linear', 'quadratic', 'SR7L', 'GE36'
chordRoot = 10                                  #as percent of diameter
chordTip = 2                                    #as percent of diameter
typeOfParabolaChord = 'hub'

activityFactorChoice = 0                        #(0) as is, (1) for custom
activityFactorTarget = 120

#airfoil thickness
thickRoot = 21                                  #as percent of chord
thickTip = 8                                    #as percent of chord
thicknessChoice = 'linear'                      #'linear', 'quadratic', 'SR7L', 'GE36'
typeOfParabolaThickness = 'tip'

#airfoil camber
camberRoot = 0.3
camberTip = 0.3
camberChoice = 'linear'                         #'linear', 'quadratic', 'SR7L' (GE36 not available since airfoil not known)
typeOfParabolaCamber = 'tip'

#Blade loading (starting guess if chord is pre-spec, actual loading if chord is a variable)
cldRoot = 0.3
cldTip = 0.4
cldChoice = 'linear'                            #'linear', 'quadratic'
typeOfParabolacl = 'tip'

#Blade mechanical inputs
bladeDensity = 1520                             #Density for blade material [kg/m^3]

#############################################################
acoustic_study='yes'                            # yes or empty
# Near field observers
XNum1 = 101                                     #NUmber of observers in xand z axes
ZNum1 = 1
#Far field observers
XNum2 = 101                                     #NUmber of observers in xand z axes
ZNum2 = 1
```

Listing A.1: Input student design

```
import sys
```

```python
import matplotlib.pyplot as plt
import vtkplotlib as vpl
import time
import pdb
t_1=time.time()

from src.ISAdataFunction import ISAdataFunction
from src.PropellerProperties import PropellerGeometry, OperatingPoint, generateListOfOPsForSweep
from airfoils.airfoilAerodata import initiateAirfoilData
from src.designSolvers import LarrabeeDesign, AdkinsDesign, DrelaDesign
from src.analysisSolvers import AdkinsAnalysis ,DrelaAnalysis
from src.postProcess import PostProcess, PostProcessSweep
from src.geometryGeneration import fullPropellerGeometry
from src.AcousticStudy import AcousticStudy

plt.close('all')
vpl.close()

## Load input file
import inputFile_students_design as inputData

## Program running mode
if inputData.programMode == 'design':

    ## Design ############################################################
    ## Pre-calc
    propellerGeom = PropellerGeometry(inputData,inputData.programMode)      #Initiate propeller geometry
    airfoilAerodynamics = initiateAirfoilData(propellerGeom)          #Initiate airfoil aerodynamic data
    ISAData = ISAdataFunction(inputData.H)          #generate ISA data object with atmospheric properties
    operatingPoint = OperatingPoint(inputData, propellerGeom, ISAData)      #Initiate propeller operating point

    ## Design calc
    if inputData.specRunType == 'DL':
        designCalc = LarrabeeDesign(propellerGeom, airfoilAerodynamics, operatingPoint, ISAData)
        designCalc.solve()
    elif inputData.specRunType == 'DA':
        designCalc = AdkinsDesign(propellerGeom, airfoilAerodynamics, operatingPoint, ISAData)
        designCalc.solve()
    elif inputData.specRunType == 'DD':
        designCalc = DrelaDesign(propellerGeom, airfoilAerodynamics, operatingPoint, ISAData)
        designCalc.solve()
    else:
            sys.exit('Invalid option set in specRunType in the input file!')

    ## Post-process design
```

```
    propellerGeom.plotAirfoilDistributions()
    propellerGeom.savePropellerDesignProperties(caseName=inputData.
caseName)
    postObj = PostProcess(propellerGeom, operatingPoint, designCalc
, inputData.programMode, caseName=inputData.caseName)

    ## Output geometry
    fullPropellerGeometryObj = fullPropellerGeometry(inputData,
propellerGeom)
    fullPropellerGeometryObj.estimateCentrifugalStress(
operatingPoint.omega, rhoBlade=inputData.bladeDensity)
    fullPropellerGeometryObj.writeCSVs(caseName=inputData.caseName)
    fullPropellerGeometryObj.exportToSTL(caseName=inputData.
caseName)
    fullPropellerGeometryObj.plot3D()
    fullPropellerGeometryObj.outputCurveFilesForCAD(caseName=
inputData.caseName)




elif inputData.programMode == 'analysis':

    ## Analysis
############################################################
    ## Pre-calc
    propellerGeom = PropellerGeometry(inputData,inputData.
programMode)      #Initiate propeller geometry
    airfoilAerodynamics = initiateAirfoilData(propellerGeom)
        #Initiate airfoil aerodynamic data
    ISAData = ISAdataFunction(inputData.H)
        #generate ISA data object with atmospheric properties
    operatingPoint = OperatingPoint(inputData, propellerGeom,
ISAData)      #Initiate propeller operating point


    ## Analysis calc
    if inputData.specRunType == 'AL':
        pass
    elif inputData.specRunType == 'AA':
        analysisCalc = AdkinsAnalysis(propellerGeom,
airfoilAerodynamics, operatingPoint, ISAData)
        analysisCalc.solve()
    elif inputData.specRunType == 'AD':
         analysisCalc = DrelaAnalysis(propellerGeom,
airfoilAerodynamics, operatingPoint, ISAData)
         analysisCalc.solve()
    else:
        sys.exit('Invalid option set in specRunType in the input
file!')

    ## Post-process calc
    propellerGeom.plotAirfoilDistributions()
    postObj = PostProcess(propellerGeom, operatingPoint,
analysisCalc, inputData.programMode, caseName=inputData.caseName
)
```

```

    ## Plot 3D geometry
    fullPropellerGeometryObj = fullPropellerGeometry(inputData, propellerGeom)
    fullPropellerGeometryObj.plot3D()

elif inputData.programMode == 'sweep':

    propellerGeomList = []
    calcList = []

    ## Pre-calc
    propellerGeom = PropellerGeometry(inputData,inputData.programMode)      #Initiate propeller geometry
    propellerGeom.pitchBlade(inputData.deltaBeta75)
    airfoilAerodynamics = initiateAirfoilData(propellerGeom)        #Initiate airfoil aerodynamic data
    ISAData = ISAdataFunction(inputData.H)          #generate ISA data object with atmospheric properties
    operatingPointList = generateListOfOPsForSweep(inputData, propellerGeom, ISAData, inputData.typeOfOPSweep)

    ##Sweep advance ratio
    OPcounter = 0
    for OP in operatingPointList:
        OPcounter += 1
        print('Solving OP%i' % (OPcounter))
        if inputData.specRunType == 'AL':
            pass
        elif inputData.specRunType == 'AA':
            analysisCalc = AdkinsAnalysis(propellerGeom, airfoilAerodynamics, OP, ISAData)
            analysisCalc.solve()
            calcList.append(analysisCalc)
            propellerGeomList.append(propellerGeom)
        else:
            sys.exit('Invalid option set in specRunType in the input file!')



    postSweepObj = PostProcessSweep(propellerGeomList, operatingPointList, calcList)
t_2=time.time()
tot_time=t_2-t_1
print('Time = ', tot_time)

if inputData.acoustic_study == 'yes':
#     pdb.set_trace()
    if inputData.programMode == 'design':
        case = AcousticStudy(inputData , operatingPoint , ISAData , PostProcess , designCalc)
        case.solve()
    elif inputData.programMode == 'analysis':
        case = AcousticStudy(inputData, operatingPoint , ISAData , PostProcess , analysisCalc)
```

```
        case.solve()
```

Listing A.2: OPTOPROP

```
"""
Created on Tue Sep 19 23:34:20 2023

@author: georg
"""
import sys
import numpy as np
import settings as settings

class DrelaDesign(LarrabeeDesign):
    def __init__(self, propellerGeom, airfoilAerodynamics,
   designOperatingPoint, ISAData,maxIterCl = settings.maxIterCl):

        self._propellerGeom = propellerGeom
        self._airfoilAerodynamics = airfoilAerodynamics
        self._designOperatingPoint = designOperatingPoint
        self._ISAData = ISAData

        self.maxIterCl = maxIterCl

        self.B = propellerGeom.B
        self.D = propellerGeom.D
        self.tipradius=propellerGeom.D/2
        self.nPoints = propellerGeom.nPoints
        self.r = propellerGeom.r
        self.nonDimRadius = propellerGeom.nonDimRadius
        self.cld = propellerGeom.cld
        self.preSpecChord = propellerGeom.preSpecChord
        self.propellerGeom = propellerGeom
        self.chord = np.zeros_like(self.r)  #assigns initial values
    for chord
        self.airfoilAerodynamics = airfoilAerodynamics

        self.speedOfSound = ISAData.a
        self.rho = ISAData.rho
        self.nu = ISAData.nu

        self.speedRatio = designOperatingPoint.speedRatio
        self.vAxial = designOperatingPoint.vAxial
        self.axialVelocityUpstream=designOperatingPoint.
   axialVelocityUpstream
        self.tangentialVelocityUpstream=designOperatingPoint.
   tangentialVelocityUpstream
        self.omega = designOperatingPoint.omega
        self.preSpecifiedQuantity = designOperatingPoint.
   preSpecifiedQuantity
        if self.preSpecifiedQuantity == 'thrust':
            self.TC = designOperatingPoint.TC
        elif self.preSpecifiedQuantity == 'power':
            self.PC = designOperatingPoint.PC
        elif self.preSpecifiedQuantity == 'noSwirl':
            self.preSpecifiedQuantity = 'power'
```

```
            self.dQdrEstimate=2*np.pi*self.r**2*self.rho*(self.vAxial+self.axialVelocityUpstream)*self.tangentialVelocityUpstream
            self.powerSoughtEstimate=np.trapz(self.omega*self.dQdrEstimate , self.r)
            self.PC=2*self.powerSoughtEstimate/(self.rho*self.vAxial**3*np.pi*(self.D/2)**2)
        else:
            sys.exit('Hey doofus! You forgot to specify thrust or torque properly!')


    def solve(self, printResiduals=True):
        if self.preSpecifiedQuantity == 'noSwirl':
            print('Running Larrabee design method for initial guess, (i.e. beta and chord)')
            self.runInitialGuessNoSwirl()
            print('Running Drela design method...')
            if self.preSpecChord:
                self.DrelaDesignMinSwirl(self.beta,self.chord, Maxiteration = settings.drelaDesign_maxChordIter, relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.propellerGeom.setBladeAngleDistribution(self.beta)
            elif self.preSpecChord == False:
                self.DrelaDesignMinSwirl(self.beta,self.chord, Maxiteration = settings.drelaDesign_maxChordIter, relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.propellerGeom.chordNonDim = self.chord/self.D
                self.propellerGeom.setBladeAngleDistribution(self.beta)
                self.propellerGeom.calcAF()
        elif self.preSpecifiedQuantity == 'power':
            print('Running Larrabee design method for initial guess, (i.e. beta and chord)')

            self.runInitialGuess()

            print('Running Drela design method...')
            if self.preSpecChord:
                self.DrelaDesignMIL(solverTolLimit = settings.drelaDesign_cl_solverTolLimit, relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.propellerGeom.setBladeAngleDistribution(self.beta)
            elif self.preSpecChord == False:
                self.DrelaDesignMIL(solverTolLimit = settings.drelaDesign_cl_solverTolLimit, relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.propellerGeom.chordNonDim = self.chord/self.D
                self.propellerGeom.setBladeAngleDistribution(self.beta)
                self.propellerGeom.calcAF()
        elif self.preSpecifiedQuantity == 'thrust':
            print('Running Larrabee design method for initial guess, (i.e. beta and chord)')
```

```
            self.runInitialGuess()
            print('Running Drela design method...')
            if self.preSpecChord:
                self.DrelaDesignMIL(solverTolLimit = settings.drelaDesign_cl_solverTolLimit, relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.propellerGeom.setBladeAngleDistribution(self.beta)
            elif self.preSpecChord == False:
                self.DrelaDesignMIL(solverTolLimit = settings.drelaDesign_cl_solverTolLimit, relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.propellerGeom.chordNonDim = self.chord/self.D
                self.propellerGeom.setBladeAngleDistribution(self.beta)
                self.propellerGeom.calcAF()
        else:
            sys.exit('\tError in preSpecChord value!')

        print('Drela design method finished.')

    def runInitialGuess(self):
        self.uAxial=self.vAxial+self.axialVelocityUpstream
        self.uTangential=self.omega*self.r+self.tangentialVelocityUpstream
        self.BetaInitial=np.arctan2(self.uAxial,self.uTangential)+np.pi / 360
        self.initialGuess = LarrabeeDesign(self._propellerGeom, self._airfoilAerodynamics, self._designOperatingPoint, self._ISAData)
        self.initialGuess.solve(printResiduals=(False))
        if self.preSpecChord== True:
            self.beta = self.initialGuess.beta
            self.cld = self.initialGuess.cld
            self.chord=self.initialGuess.chord
        elif self.preSpecChord== False:
            self.chord=self.initialGuess.chord
            self.beta = self.initialGuess.beta

    def runInitialGuessNoSwirl(self):
        self.uAxial=self.vAxial+self.axialVelocityUpstream
        self.uTangential=self.omega*self.r+self.tangentialVelocityUpstream
        self.BetaInitial=np.arctan2(self.uAxial,self.uTangential)+np.pi / 360
        if self.preSpecChord== True:
            self.initialGuess = LarrabeeDesign(self._propellerGeom, self._airfoilAerodynamics, self._designOperatingPoint, self._ISAData)
            self.initialGuess.solve(printResiduals=(False))
            self.beta = self.initialGuess.beta
            self.cld = self.initialGuess.cld
            self.chord=self.initialGuess.chord
            self.preSpecifiedQuantity = 'noSwirl'
        elif self.preSpecChord== False:
            self.initialGuess = LarrabeeDesign(self._propellerGeom,
```

```
    self._airfoilAerodynamics, self._designOperatingPoint, self.
   _ISAData)
            self.initialGuess.solve(printResiduals=(False))
            self.chord=self.initialGuess.chord
            self.beta = self.initialGuess.beta
            self.preSpecifiedQuantity = 'noSwirl'

    def DrelaDesignMIL(self, solverTolLimit = settings.
   drelaDesign_cl_solverTolLimit, relaxationFactor = settings.
   drelaDesign_cl_relaxationFactor):

        self.tolofeta = 1                     #convergence
        self.dragToLiftRatioNew = np.ones(self.nPoints)
        self.EtaindResidualTrend = []                        #
   clResidualTrend
        self.relativeMachNumber=np.zeros(self.nPoints)
        self.alpha = np.zeros(self.nPoints)
        self.cl = np.zeros(self.nPoints)
        self.cd = np.zeros(self.nPoints)
        self.dragToLiftRatio = np.zeros(self.nPoints)

        #Bisection method for Eta induced
        self.EtaIndLo=0.6
        self.EtaIndHi=1.2
        self.EtaItr = 0
#        pdb.set_trace()
        self.Maxiteration = settings.drelaDesign_maxChordIter
        if hasattr(self,'Rtheta'): #if Rtheta exists since previous
    iteration use it
            pass
        else:
            self.Rtheta=0.1

        while self.tolofeta > solverTolLimit:
            print('Eta Iteration = ',self.EtaItr)
            self.EtaItr += 1
            self.EtaInd=(self.EtaIndHi+self.EtaIndLo)/2

            #self.beta_chord_tuple=self.beta
            #get the presint thrust or power according to the
   initial values( beta and chord)
            self.Present_SpecifiedQuantity=self.designResidual(self
   .beta , self.chord,self.EtaInd,self.Maxiteration ,
   relaxationFactor )
            self.T_pre=self.Present_SpecifiedQuantity[0]
            self.P_pre=self.Present_SpecifiedQuantity[1]
            #calculating R_theta (to spicify the direction of eta)
            if self.preSpecifiedQuantity == 'thrust':
                self.Rtheta=self.T_pre-self.TC
                print('Risiduals of theta = ',self.Rtheta)
            elif self.preSpecifiedQuantity == 'power':
                self.Rtheta=self.P_pre-self.PC
                print('Risiduals of theta = ',self.Rtheta)
            else:
                print("Drela design routine doesn't converge")

```

```
            if self.Rtheta<0:
                self.EtaIndHi=self.EtaInd
            elif self.Rtheta>0:
                self.EtaIndLo =self.EtaInd
            #calculating the residuals of eta
            self.tolofeta = max([abs((self.EtaIndHi-self.EtaIndLo)/self.EtaIndHi)])
            self.EtaindResidualTrend.append(self.tolofeta)

            if self.EtaItr >= self.maxIterCl:
                print('Drela design routine did not converge, exceeded amount of iterations.')
                break
        #check conve for preSpecifiedQuantity:
        if self.preSpecifiedQuantity == 'thrust':
            if abs(self.T_pre/self.TC -1) < 1e-3:
                print('Design converged to specified thrust!')
            else:
                print('Design DID NOT CONVERGE to specified thrust!')
        elif self.preSpecifiedQuantity == 'power':
            if abs(self.P_pre/self.PC -1) < 1e-3:
                print('Design converged to specified power!')
            else:
                print('Design DID NOT CONVERGE to specified power!')
        self.Final_design=self.psiresconverged(self.psiInitial , self.beta,self.chord , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)

        print('Drela design routine converge!')


    def DrelaDesignMinSwirl(self, beta,chord, Maxiteration = settings.drelaDesign_maxChordIter, relaxationFactor = settings.drelaDesign_cl_relaxationFactor):

        # set the values of beta, chord and eta induced
        self.relaxationFactor=relaxationFactor
        self.beta=beta
        self.chord=chord

        self.Maxiteration=Maxiteration

        #pre-calculation to get an initial value for psi
        self.uAxial=self.vAxial+self.axialVelocityUpstream
        self.uTangential=self.omega*self.r+self.tangentialVelocityUpstream
        self.u = np.sqrt(self.uAxial**2+self.uTangential**2)
        self.psiInitial=np.arcsin(self.uAxial/self.u)

        if self.preSpecChord==True:
            self.loopConvergence =  []                     #1
            self.loopconvergence= 1                        #1
            self.desItr=0
            self.loopConvergenceTemp=1
```

```
            self.stepSize=1
            self.pertSize=1e-6

            self.beta1=self.beta

            while self.loopConvergenceTemp > 1e-4:
                print('Design Iteration = ',self.desItr)
                self.desItr += 1
                #set a condition to not exceed the maximum iteration value
                if self.desItr>self.Maxiteration:
                    print('Drela residual for beta and chord has not converged!')

                self.first_point=self.psiresconverged(self.psiInitial , self.beta1,self.chord , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.psi= self.first_point[0]
                self.inducedTangentialVelocity1= self.first_point[9]
                self.Rr2 = self.inducedTangentialVelocity1 / self.tangentialVelocityUpstream -1
                # self.cl_p1 = self.first_point[3]
                # self.inducedTangentialVelocity1 = self.first_point[9]
                pertSize=1e-6
                #Rr1 derivative wrt beta
                self.beta2=self.beta1*(1+pertSize)
                self.second_point=self.psiresconverged(self.psiInitial , self.beta2,self.chord , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.inducedTangentialVelocity2= self.second_point[9]

                self.Rr2_perturb = self.inducedTangentialVelocity2 / self.tangentialVelocityUpstream -1
                self.dRr2_perturb = (self.Rr2_perturb - self.Rr2) / ( pertSize * self.beta1)
                #calculate new delta for beta
                self.deltaBeta=-1*self.Rr2/self.dRr2_perturb
                # calculate convergence
                self.loopConvergenceTemp = np.max(np.abs(self.deltaBeta) / np.abs(self.beta1))

                self.loopConvergence.append(self.loopConvergenceTemp)
                self.beta1=self.beta1+self.deltaBeta*self.stepSize
            self.beta=self.beta1
            self.output=True
            self.finaldesign=self.psievaluation(self.psi,self.beta, self.chord,self.output)
            self.TC_p=self.finaldesign[7]
            self.PC_p=self.finaldesign[8]
            return self.TC_p, self.PC_p ,self.loopConvergence,self.desItr,self.psiInitial,self.beta,self.chord

```

```
    elif self.preSpecChord==False:
        # initilise the residuals components
        self.deltaBeta = np.zeros(self.nPoints)
        self.deltaChord = np.zeros(self.nPoints)
        self.betaConv =  np.zeros(self.nPoints)
        self.chordConv = np.zeros(self.nPoints)
        self.loopConvergence =  []                  #1
        self.loopconvergence= 1                     #1
        self.desItr=0
        self.loopConvergenceTemp=1

        #set steps value (those values will be changed according to the distance )
        self.stepSizeChord=0.2
        self.stepSizeBeta=0.5
        self.pertSize=1e-6

        self.beta1=self.beta
        self.chord1=self.chord
        while self.loopConvergenceTemp > 1e-4:
#                print('desItr',self.desItr)
            self.desItr += 1
            #changing the step size according to the loop convergence
            if self.loopConvergenceTemp <=0.2:
                self.stepSizeBeta=1
                self.stepSizeChord=0.5
                self.pertSize=1e-6
                self.relaxationFactor=1
            elif self.loopConvergenceTemp <=0.1:
                self.stepSizeBeta=1
                self.stepSizeChord=1
                self.pertSize=1e-6
                self.relaxationFactor=1
            #set a condition to not exceed the maximum iteration value
            if self.desItr>self.Maxiteration:
                print('Drela residual for beta and chord has not converged!')

            self.first_point=self.psiresconverged(self.psiInitial , self.beta1,self.chord1 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
            self.psi= self.first_point[0]
            self.cl_p1 = self.first_point[3]
            self.inducedTangentialVelocity1 = self.first_point[9]

            self.Rr1 = self.cl_p1 - self.cld
            self.Rr2 = self.inducedTangentialVelocity1 / self.tangentialVelocityUpstream -1

            pertSize=self.pertSize
            #Rr1 derivative wrt beta
            self.beta2=self.beta1*(1+pertSize)
            self.chord2=self.chord1
```

```
                self.second_point=self.psiresconverged(self.psiInitial , self.beta2,self.chord2 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.cl_p2 = self.second_point[3]
                self.dRr1_dbeta = (self.cl_p2 - self.cl_p1) / (self.beta2 - self.beta1)
                #Rr1 derivative wrt chord
                self.dRr1_dchord = np.zeros(self.nPoints)
                #Rr2 derivative wrt beta
                self.beta3=self.beta1*(1+pertSize)
                self.chord3=self.chord1
                self.second_point=self.psiresconverged(self.psiInitial , self.beta3,self.chord3 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.cl_p3 = self.second_point[3]
                self.inducedTangentialVelocity3 = self.second_point[9]

                self.Rr2_pert_beta = self.inducedTangentialVelocity3 / self.tangentialVelocityUpstream -1
                self.dRr2_dbeta = (self.Rr2_pert_beta - self.Rr2 )/(self.beta3 - self.beta1)
                #Rr2 derivative wrt chord
                self.beta4=self.beta1
                self.chord4=self.chord1*(1+pertSize)
                self.second_point=self.psiresconverged(self.psiInitial , self.beta4,self.chord4 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.cl_p4 = self.second_point[3]
                self.inducedTangentialVelocity4 = self.second_point[9]
                self.Rr2_pert_chord = self.inducedTangentialVelocity4 / self.tangentialVelocityUpstream -1
                self.dRr2_dchord = (self.Rr2_pert_chord - self.Rr2)/(self.chord4 - self.chord1)

                for k in range(self.nPoints):
                    self.jacobian = np.array([[self.dRr1_dbeta[k], self.dRr1_dchord[k]], [self.dRr2_dbeta[k], self.dRr2_dchord[k]]])
                    if np.linalg.matrix_rank(self.jacobian) == self.jacobian.shape[0]:
                        self.deltaVariables = -np.linalg.inv(self.jacobian) @ np.array([self.Rr1[k], self.Rr2[k]])
                        self.deltaBeta[k] = self.deltaVariables[0]
                        self.deltaChord[k] = self.deltaVariables[1]

                # calculate convergence
                self.deltabetaoverbeta=np.abs(self.deltaBeta) / np.abs(self.beta1)
                self.betaConv = np.max(self.deltabetaoverbeta)
                self.chordConv = np.max((np.abs(self.deltaChord) / np.abs(self.chord1)))
                self.loopConvergenceTemp = np.max([self.betaConv, self.chordConv])
                self.loopConvergence.append(self.
```

```
loopConvergenceTemp)
                self.beta1=self.beta1+self.deltaBeta*self.
stepSizeBeta
                self.chord1=self.chord1+self.deltaChord*self.
stepSizeChord
            self.beta=self.beta1
            self.chord=self.chord1
            self.output=True
            self.finaldesign=self.psievaluation(self.psi,self.beta
, self.chord,self.output)
            self.TC_p=self.finaldesign[7]
            self.PC_p=self.finaldesign[8]
            return self.TC_p, self.PC_p ,self.loopConvergence,self.
desItr,self.psiInitial,self.beta,self.chord

        self.Final_design=self.psiresconverged(self.psiInitial ,
self.beta,self.chord , relaxationFactor = settings.
drelaDesign_cl_relaxationFactor)

        print('Drela design routine converge!')

    def designResidual(self,beta,chord,EtaInd, Maxiteration =
settings.drelaDesign_maxChordIter, relaxationFactor = settings.
drelaDesign_cl_relaxationFactor):
        # set the values of beta, chord and eta induced
        self.relaxationFactor=relaxationFactor
        self.beta=beta
        self.chord=chord
        self.EtaInd = EtaInd
        self.Maxiteration=Maxiteration

        #pre-calculation to get an initial value for psi
        self.uAxial=self.vAxial+self.axialVelocityUpstream
        self.uTangential=self.omega*self.r+self.
tangentialVelocityUpstream
        self.u = np.sqrt(self.uAxial**2+self.uTangential**2)
        self.psiInitial=np.arcsin(self.uAxial/self.u)

        if self.preSpecChord==True:
            self.loopConvergence =  []                      #1
            self.loopconvergence= 1                         #1
            self.desItr=0
            self.loopConvergenceTemp=1
            self.stepSize=1
            self.pertSize=1e-6

            self.beta1=self.beta

            while self.loopConvergenceTemp > 1e-4:
#                print('desItr',self.desItr)
                self.desItr += 1
                #set a condition to not exceed the maximum
iteration value
                if self.desItr>self.Maxiteration:
                    print('Drela residual for beta and chord has
not converged!')
```

```

                self.first_point=self.psiresconverged(self.psiInitial , self.beta1,self.chord , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.psi= self.first_point[0]
                self.Wa1= self.first_point[6]
                self.Wt1= self.first_point[7]
                cosntant=self.vAxial/(self.r*self.omega)
                self.eta_induced_p1 =cosntant* (self.Wt1)/(self.Wa1)
                self.Rr2 = self.EtaInd - self.eta_induced_p1

                pertSize=1e-6
                #Rr1 derivative wrt beta
                self.beta2=self.beta1*(1+pertSize)
                self.second_point=self.psiresconverged(self.psiInitial , self.beta2,self.chord , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.Wa2= self.second_point[6]
                self.Wt2= self.second_point[7]
                self.eta_induced_p2 = cosntant * (self.Wt2)/(self.Wa2)
                self.Rr2_perturb = self.EtaInd - self.eta_induced_p2
                self.dRr2_perturb = (self.Rr2_perturb - self.Rr2) / ( pertSize * self.beta1)
                #calculate new delta for beta
                self.deltaBeta=-1*self.Rr2/self.dRr2_perturb
                # calculate convergence
                self.loopConvergenceTemp = np.max(np.abs(self.deltaBeta) / np.abs(self.beta1))

                self.loopConvergence.append(self.loopConvergenceTemp)
                self.beta1=self.beta1+self.deltaBeta*self.stepSize
            self.beta=self.beta1
            self.output=True
            self.finaldesign=self.psievaluation(self.psi,self.beta, self.chord,self.output)
            self.TC_p=self.finaldesign[7]
            self.PC_p=self.finaldesign[8]
            return self.TC_p, self.PC_p ,self.loopConvergence,self.desItr,self.psiInitial,self.beta,self.chord

        elif self.preSpecChord==False:
            # initilise the residuals components
            self.deltaBeta = np.zeros(self.nPoints)
            self.deltaChord = np.zeros(self.nPoints)
            self.betaConv =  np.zeros(self.nPoints)
            self.chordConv = np.zeros(self.nPoints)
            self.loopConvergence =  []                     #1
            self.loopconvergence= 1                        #1
            self.desItr=0
            self.loopConvergenceTemp=1

            #set steps value (those values will be changed
```

```
according to the distance )
            self.stepSizeChord=0.2
            self.stepSizeBeta=0.5
            self.pertSize=1e-6

            self.beta1=self.beta
            self.chord1=self.chord
            while self.loopConvergenceTemp > 1e-4:
#                print('design Itr',self.desItr)
                self.desItr += 1
                #changing the step size according to the loop convergence
                if self.loopConvergenceTemp <=0.2:
                    self.stepSizeBeta=1
                    self.stepSizeChord=0.5
                    self.pertSize=1e-6
                    self.relaxationFactor=1
                elif self.loopConvergenceTemp <=0.1:
                    self.stepSizeBeta=1
                    self.stepSizeChord=1
                    self.pertSize=1e-6
                    self.relaxationFactor=1
                #set a condition to not exceed the maximum iteration value
                if self.desItr>self.Maxiteration:
                    print('Drela residual for beta and chord has not converged!')

                self.first_point=self.psiresconverged(self.psiInitial , self.beta1,self.chord1 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.psi= self.first_point[0]
                self.cl_p1 = self.first_point[3]
                self.upstreamEfficiency1 = self.first_point[4]
                self.inducedEfficiency1 = self.first_point[5]
                self.eta_induced_p1 = self.upstreamEfficiency1 * self.inducedEfficiency1

                self.Rr1 = self.cl_p1 - self.cld
                self.Rr2 = self.EtaInd - self.eta_induced_p1

                pertSize=self.pertSize
                #Rr1 derivative wrt beta
                self.beta2=self.beta1*(1+pertSize)
                self.chord2=self.chord1
                self.second_point=self.psiresconverged(self.psiInitial , self.beta2,self.chord2 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
                self.cl_p2 = self.second_point[3]
                self.upstreamEfficiency2 = self.second_point[4]
                self.inducedEfficiency2 = self.second_point[5]
                self.eta_induced_p2 = self.upstreamEfficiency2 * self.inducedEfficiency2
                self.dRr1_dbeta = (self.cl_p2 - self.cl_p1) / (self.beta2 - self.beta1)
                #Rr1 derivative wrt chord
```

```python
            self.dRr1_dchord = np.zeros(self.nPoints)
            #Rr2 derivative wrt beta
            self.beta3=self.beta1*(1+pertSize)
            self.chord3=self.chord1
            self.second_point=self.psiresconverged(self.psiInitial , self.beta3,self.chord3 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
            self.cl_p3 = self.second_point[3]
            self.upstreamEfficiency3 = self.second_point[4]
            self.inducedEfficiency3 = self.second_point[5]
            self.eta_induced_p3 = self.upstreamEfficiency3 * self.inducedEfficiency3
            self.Rr2_pert_beta = self.EtaInd - self.eta_induced_p3
            self.dRr2_dbeta = (self.Rr2_pert_beta - self.Rr2 )/(self.beta3 - self.beta1)
            #Rr2 derivative wrt chord
            self.beta4=self.beta1
            self.chord4=self.chord1*(1+pertSize)
            self.second_point=self.psiresconverged(self.psiInitial , self.beta4,self.chord4 , relaxationFactor = settings.drelaDesign_cl_relaxationFactor)
            self.cl_p4 = self.second_point[3]
            self.upstreamEfficiency4 = self.second_point[4]
            self.inducedEfficiency4 = self.second_point[5]
            self.eta_induced_p4 = self.upstreamEfficiency4 * self.inducedEfficiency4
            self.Rr2_pert_chord = self.EtaInd - self.eta_induced_p4
            self.dRr2_dchord = (self.Rr2_pert_chord - self.Rr2 )/(self.chord4 - self.chord1)

            for k in range(self.nPoints):
                self.jacobian = np.array([[self.dRr1_dbeta[k], self.dRr1_dchord[k]], [self.dRr2_dbeta[k], self.dRr2_dchord[k]]])
                if np.linalg.matrix_rank(self.jacobian) == self.jacobian.shape[0]:
                    self.deltaVariables = -np.linalg.inv(self.jacobian) @ np.array([self.Rr1[k], self.Rr2[k]])
                    self.deltaBeta[k] = self.deltaVariables[0]
                    self.deltaChord[k] = self.deltaVariables[1]

            # calculate convergence
            self.deltabetaoverbeta=np.abs(self.deltaBeta) / np.abs(self.beta1)
            self.betaConv = np.max(self.deltabetaoverbeta)
            self.chordConv = np.max((np.abs(self.deltaChord) / np.abs(self.chord1)))
            self.loopConvergenceTemp = np.max([self.betaConv, self.chordConv])
            self.loopConvergence.append(self.loopConvergenceTemp)
            self.beta1=self.beta1+self.deltaBeta*self.stepSizeBeta
            self.chord1=self.chord1+self.deltaChord*self.
```

```
stepSizeChord
            self.beta=self.beta1
            self.chord=self.chord1
            self.output=True
            self.finaldesign=self.psievaluation(self.psi,self.beta
, self.chord,self.output)
            self.TC_p=self.finaldesign[7]
            self.PC_p=self.finaldesign[8]
            return self.TC_p, self.PC_p ,self.loopConvergence,self.
desItr,self.psiInitial,self.beta,self.chord


 def psiresconverged(self, psiInitial , beta ,chord ,
relaxationFactor = settings.drelaDesign_cl_relaxationFactor):

      self.uAxial=self.vAxial+self.axialVelocityUpstream
      self.uTangential=self.omega*self.r+self.
tangentialVelocityUpstream
      self.u = np.sqrt(self.uAxial**2+self.uTangential**2)

      if hasattr(self,'psiInitial'): #if psiInitial exists since
previous iteration use it
            self.psi=self.psiInitial
      else:
            self.psiInitial=np.arctan(self.uAxial/self.uTangential)
            self.psi=self.psiInitial

      self.beta= beta
      self.chord= chord


      self.psiResidualTrend = []
      self.PsiResidual=1
      self.Newiteration=0

      while np.any(self.PsiResidual > 1e-4):
            print('New iteration of psi = ',self.Newiteration)
            self.psi1=self.psi       #we can put it before while
loop
            self.output=False


            self.Rpsi1=self.psievaluation(self.psi1,self.beta ,
self.chord ,self.output)

            self.psi2=self.psi1*(1+1e-8)

            self.Rpsi2=self.psievaluation(self.psi2,self.beta ,
self.chord ,self.output)

            self.derivative = (self.Rpsi2-self.Rpsi1)/(self.psi2-
self.psi1)

            self.deltaPsi = -1*self.Rpsi1/self.derivative
            self.psiTemp = self.psi1 + self.deltaPsi
            #self.psiTemp = self.psiTempFun(self.psi1 , self.beta ,
```

```
 self.chord ,psiPerturb = 1e-6 )

            self.values=(self.psiTemp - self.psi1)
            self.values=self.values/ self.psiTemp
            #self.values=self.values.flatten()
             #delete
            self.PsiResidual = np.linalg.norm(self.values, ord=np.
inf)

            self.psiResidualTrend.append(self.PsiResidual)
            self.psiNew = self.relaxationFactor*self.psiTemp + (1-
self.relaxationFactor)*self.psi1
            self.psi=self.psiNew
            self.Newiteration+=1
            if self.Newiteration>200:
                break
                print('Drela residual for beta and chord has not
converged!')
        #psievaluation
        self.output=True
        self.presentvalues= self.psievaluation(self.psiNew,self.
beta , self.chord,self.output)
        self.cl=self.presentvalues[1]
        self.dEtaUpdr =self.presentvalues[2]
        self.dEtaIndr=self.presentvalues[3]
        self.Wa = self.presentvalues[9]
        self.Wt = self.presentvalues[10]
        self.va = self.presentvalues[11]
        self.vt = self.presentvalues[12]


        return self.psiNew,self.Newiteration,self.psiResidualTrend,
self.cl,self.dEtaUpdr,self.dEtaIndr,self.Wa,self.Wt,self.va,self
.vt
    def psievaluation(self,psi,beta , chord,output):
        self.output=output
        self.psi=psi
        self.beta = beta
        self.chord = chord
        self.alpha = np.zeros(self.nPoints)
        self.cl = np.zeros(self.nPoints)
        self.cd = np.zeros(self.nPoints)
        self.dragToLiftRatio = np.zeros(self.nPoints)
        self.phi= np.zeros(self.nPoints)

        self.Wa=(self.uAxial+self.u*np.sin(self.psi))/2          #
        self.Wt=(self.uTangential+self.u*np.cos(self.psi))/2     #
        self.W=np.sqrt(self.Wa**2+self.Wt**2)
        #calcutating renolds and mach
        self.Re=self.W*self.chord/self.nu
        self.relativeMachNumber=self.W/self.speedOfSound
        self.phi=np.arctan(self.Wa/self.Wt)
        #induced velocities
        self.va=self.Wa-self.uAxial
        self.vt=self.uTangential-self.Wt
        #Alpha
```

```
        self.alpha=self.beta-self.phi
        #Prandelt method
        self.Lambdaw=(self.r/self.tipradius)*(self.Wa/self.Wt)
        self.f=self.B/2*(1-self.r/self.tipradius)/self.Lambdaw
        self.F=2/np.pi*np.arccos(np.exp(-self.f))

        for i, airfoil in enumerate(self.airfoilAerodynamics):
            _ , self.cl[i], self.cd[i] = airfoil.interpolateUsingAlpha(self.relativeMachNumber[i], self.alpha[i])
        #circulation equation
        self.Gamma=self.vt*(4*np.pi*self.r/self.B)*self.F*np.sqrt(1+(4*self.Lambdaw*self.tipradius/(np.pi*self.B*self.r))**2)
        self.Rpsi=self.Gamma-0.5*self.W*self.chord*self.cl

        if output == True:
            self.va=self.va
            self.vt=self.vt
            self.rps=self.omega/(2*np.pi)
            self.a=self.va/self.vAxial
            self.aPrime=self.vt/(self.omega*self.r)
            #local efficiencies
            self.ua=self.uAxial-self.vAxial
            self.ut=self.omega*self.r -self.uTangential
            self.dragToLiftRatio=self.cd/self.cl
            #efficiencies
            self.dEtaPdr=(1-self.dragToLiftRatio*self.Wa/self.Wt)/(1+self.dragToLiftRatio*self.Wt/self.Wa)
            self.dEtaIndr=(1-self.vt/self.uTangential)/(1+self.va/self.uAxial)
            self.dEtaUpdr=(1-self.ut/(self.omega*self.r))/(1+self.ua/self.vAxial)

            self.dLdr = 0.5*self.B*self.rho*self.W**2*self.cl*self.chord
            self.dDdr = 0.5*self.B*self.rho*self.W**2*self.cd*self.chord

            self.dTdr = self.dLdr*np.cos(self.phi)-self.dDdr*np.sin(self.phi)
            self.dQdr=self.r*(self.dLdr*np.sin(self.phi) + self.dDdr*np.cos(self.phi))

            self.dPdr = self.dQdr*self.omega
            self.dEtadr = self.vAxial*self.dTdr/self.dPdr
            self.T = np.trapz(self.dTdr,self.r)
            self.Q = np.trapz(self.dQdr,self.r)
            self.P = self.Q*self.omega
            self.eta=self.T*self.vAxial/self.P

            self.freeStreamVelocity=self.vAxial
            self.TC_p = 2*self.T/(self.rho*self.freeStreamVelocity**2*np.pi*self.tipradius**2)
            self.PC_p = 2*self.P/(self.rho*self.freeStreamVelocity**3*np.pi*self.tipradius**2)

            ##velocities calc:
```

```
            omega=self.omega
            self.romega= self.r*omega# delete the extra step
            self.swirlVelDisk = self.romega - self.Wt
            self.swirlAngle = np.pi/2 - np.arctan2(self.Wa,self.swirlVelDisk)
            B=self.B
            self.ultimateWakeTangentialVelocity = self.vt*self.F*np.sqrt(1 + (4*self.Lambdaw*self.tipradius/(np.pi * B*self.r)))
            rho=self.rho
            self.massFlow =np.trapz(self.r * self.Wa * 2 * np.pi * rho, self.r)

            self.averageWa = 1/self.massFlow*np.trapz(rho*self.Wa**2*2*np.pi*self.r, self.r)

            self.averageInducedAdvanceRatio = self.averageWa/(self.rps*(2*self.tipradius))
            self.swirlLossAtDisk = np.trapz(self.rho*self.Wa*(0.5*self.swirlVelDisk**2)*2*np.pi*self.r,self.r)/self.P
            return self.Rpsi,self.cl,self.dEtaUpdr ,self.dEtaIndr,self.T,self.P,self.eta,self.TC_p,self.PC_p,self.Wa,self.Wt,self.va,self.vt
        else:
            return self.Rpsi
```

Listing A.3: Design Solver

B

Appendix 2

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import settings as settings
import pdb


class DrelaAnalysis:
    def __init__(self, propellerGeom, airfoilAerodynamics,
    operatingPoint, ISAData, maxIter=settings.drelaAnalysis_MaxIter,
    relaxationFactor = settings.drelaAnalysis_relaxationFactor):

        self._propellerGeom = propellerGeom
        self._airfoilAerodynamics = airfoilAerodynamics
        self._operatingPoint = operatingPoint
        self._ISAData = ISAData
        self.relaxationFactor = relaxationFactor
        self.maxIter = maxIter

        self.B = propellerGeom.B
        self.D = propellerGeom.D
        self.nPoints = propellerGeom.nPoints
        self.r = propellerGeom.r
        self.nonDimRadius = propellerGeom.nonDimRadius
        self.bladeAngle = propellerGeom.bladeAngle
        self.chord = propellerGeom.chordNonDim*propellerGeom.D

        self.airfoilAerodynamics = airfoilAerodynamics

        self.speedOfSound = ISAData.a
        self.rho = ISAData.rho
        self.nu = ISAData.nu

        self.speedRatio = operatingPoint.speedRatio
        self.vAxial = operatingPoint.vAxial
        self.omega = operatingPoint.omega
        self.axialVelocityUpstream=operatingPoint.
    axialVelocityUpstream
        self.tangentialVelocityUpstream=operatingPoint.
    tangentialVelocityUpstream
        self.rps = operatingPoint.rps
        self.solidity = self.B*self.chord/(2*np.pi*self.r)


```

```
    def solve(self):
        print('Running Drela analysis method...')
        self.design=self.psiresconverge(self, self.bladeAngle ,self.chord )
        print('Drela analysis method finished.')

    def psiresconverge(self, solverTolLimit = settings.drelaAnalysis_solverTolLimit, relaxationFactor = settings.drelaAnalysis_relaxationFactor, printResiduals=True):
        self.uAxial=self.vAxial+self.axialVelocityUpstream
        self.uTangential=self.omega*self.r+self.tangentialVelocityUpstream
        self.u = np.sqrt(self.uAxial**2+self.uTangential**2)

        if hasattr(self,'psiInitial'): #if psiInitial exists since previous iteration use it
            self.psi=self.psiInitial
        else:
            self.psiInitial=np.arctan(self.uAxial/self.uTangential)
            self.psi=self.psiInitial
        self.beta=self.bladeAngle
        self.psiResidualTrend = []
        self.PsiResidual=1
        self.Newiteration=0
        while np.any(self.PsiResidual > 1e-4):
            print('New iteration',self.Newiteration)
            self.psi1=self.psi      #we can put it before while loop
            self.output=False


            self.Rpsi1=self.psievaluation(self.psi1,self.beta ,self.chord ,self.output)

            self.psi2=self.psi1*(1+1e-8)

            self.Rpsi2=self.psievaluation(self.psi2,self.beta ,self.chord ,self.output)

            self.derivative = (self.Rpsi2-self.Rpsi1)/(self.psi2-self.psi1)

            self.deltaPsi = -1*self.Rpsi1/self.derivative
            self.psiTemp = self.psi1 + self.deltaPsi

            self.values=(self.psiTemp - self.psi1)
            self.values=self.values/ self.psiTemp

            self.PsiResidual = np.linalg.norm(self.values, ord=np.inf)

            self.psiResidualTrend.append(self.PsiResidual)
            self.psiNew = self.relaxationFactor*self.psiTemp + (1-self.relaxationFactor)*self.psi1
            self.psi=self.psiNew
            self.Newiteration+=1
```

```
            if self.Newiteration>200:
                break
                print('Drela residual for beta and chord has not converged!')
        #psievaluation
        self.output=True
        self.presentvalues= self.psievaluation(self.psiNew,self.beta , self.chord,self.output)
        self.cl=self.presentvalues[1]
        self.dEtaUpdr =self.presentvalues[2]
        self.dEtaIndr=self.presentvalues[3]
        self.Wa = self.presentvalues[9]
        self.Wt = self.presentvalues[10]

        return self.psiNew,self.Newiteration,self.psiResidualTrend,self.cl,self.dEtaUpdr,self.dEtaIndr,self.Wa,self.Wt


    def psievaluation(self,psi,beta , chord,output):
        self.output=output
        self.psi=psi
        self.beta = beta
        self.chord = chord
        self.tipradius=self.D/2
        self.alpha = np.zeros(self.nPoints)
        self.cl = np.zeros(self.nPoints)
        self.cd = np.zeros(self.nPoints)
        self.dragToLiftRatio = np.zeros(self.nPoints)
        self.phi= np.zeros(self.nPoints)

        self.Wa=(self.uAxial+self.u*np.sin(self.psi))/2          #
        self.Wt=(self.uTangential+self.u*np.cos(self.psi))/2     #
        self.W=np.sqrt(self.Wa**2+self.Wt**2)
        #calcutating renolds and mach
        self.Re=self.W*self.chord/self.nu
        self.relativeMachNumber=self.W/self.speedOfSound
        self.phi=np.arctan(self.Wa/self.Wt)
        #induced velocities
        self.va=self.Wa-self.uAxial
        self.vt=self.uTangential-self.Wt
        #Alpha
        self.alpha=self.beta-self.phi
        #Prandelt method
        self.Lambdaw=(self.r/self.tipradius)*(self.Wa/self.Wt)
        self.f=self.B/2*(1-self.r/self.tipradius)/self.Lambdaw
        self.F=2/np.pi*np.arccos(np.exp(-self.f))

        for i, airfoil in enumerate(self.airfoilAerodynamics):
            _ , self.cl[i], self.cd[i] = airfoil.interpolateUsingAlpha(self.relativeMachNumber[i], self.alpha[i])
        #circulation equation
        self.Gamma=self.vt*(4*np.pi*self.r/self.B)*self.F*np.sqrt(1+(4*self.Lambdaw*self.tipradius/(np.pi*self.B*self.r))**2)
        self.Rpsi=self.Gamma-0.5*self.W*self.chord*self.cl

        if output == True:
```

```
            self.rps=self.omega/(2*np.pi)
            self.a=self.va/self.vAxial
            self.aPrime=self.vt/(self.omega*self.r)
            #local efficiencies
            self.ua=self.uAxial-self.vAxial
            self.ut=self.omega*self.r -self.uTangential
            self.dragToLiftRatio=self.cd/self.cl
            #efficiencies
            self.dEtaPdr=(1-self.dragToLiftRatio*self.Wa/self.Wt)/(1+self.dragToLiftRatio*self.Wt/self.Wa)
            self.dEtaIndr=(1-self.vt/self.uTangential)/(1+self.va/self.uAxial)
            self.dEtaUpdr=(1-self.ut/(self.omega*self.r))/(1+self.ua/self.vAxial)

            #performance: Calculate dTdr and dQdr
            self.dLdr = 0.5*self.B*self.rho*self.W**2*self.cl*self.chord
            self.dDdr = 0.5*self.B*self.rho*self.W**2*self.cd*self.chord

            self.dTdr = self.dLdr*np.cos(self.phi)-self.dDdr*np.sin(self.phi)
            self.dQdr=self.r*(self.dLdr*np.sin(self.phi) + self.dDdr*np.cos(self.phi))

            self.dPdr = self.dQdr*self.omega
            self.dEtadr = self.vAxial*self.dTdr/self.dPdr
            self.T = np.trapz(self.dTdr,self.r)
            self.Q = np.trapz(self.dQdr,self.r)
            self.P = self.Q*self.omega
            self.eta=self.T*self.vAxial/self.P

            self.freeStreamVelocity=self.vAxial
            self.TC_p = 2*self.T/(self.rho*self.freeStreamVelocity**2*np.pi*self.tipradius**2)
            self.PC_p = 2*self.P/(self.rho*self.freeStreamVelocity**3*np.pi*self.tipradius**2)

            ##velocities calc:
            omega=self.omega
            self.romega= self.r*omega# delete the extra step
            self.swirlVelDisk = self.romega - self.Wt
            self.swirlAngle = np.pi/2 - np.arctan2(self.Wa,self.swirlVelDisk)
            B=self.B
            self.ultimateWakeTangentialVelocity = self.vt*self.F*np.sqrt(1 + (4*self.Lambdaw*self.tipradius/(np.pi * B*self.r)))
            rho=self.rho
            self.massFlow =np.trapz(self.r * self.Wa * 2 * np.pi * rho, self.r)

            self.averageWa = 1/self.massFlow*np.trapz(rho*self.Wa**2*2*np.pi*self.r, self.r)

            self.averageInducedAdvanceRatio = self.averageWa/(self.
```

```
rps*(2*self.tipradius))
            self.swirlLossAtDisk = np.trapz(self.rho*self.Wa*(0.5*
self.swirlVelDisk**2)*2*np.pi*self.r,self.r)/self.P
            return self.Rpsi,self.cl,self.dEtaUpdr ,self.dEtaIndr,
self.T,self.P,self.eta,self.TC_p,self.PC_p,self.Wa,self.Wt
        else:
            return self.Rpsi
```

Listing B.1: Input student design

C

Appendix 3

```
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 13 15:37:47 2023

@author: georg
"""

# -*- coding: utf-8 -*-

import sys
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import scipy.special
#from scipy.special import jn  # Bessel function in SciPy
import pdb

class AcousticStudy:
    def __init__(self, inputData, operatingPoint, ISAData,
     PostProcess, calc):

        self.operatingPoint = operatingPoint
        self._ISAData = ISAData
        self._PostProcess = PostProcess
        self.PropNum=0
        self.B =inputData.B
        self.D =inputData.D
        self.M_axial = inputData.M_axial

        self.XNum1   = inputData.XNum1
        self.ZNum1   = inputData.ZNum1
        self.XNum2   = inputData.XNum2
        self.ZNum2   = inputData.ZNum2
        self.speedOfSound = ISAData.a #319.9125 #
        self.rpm = inputData.rpm #2200#
        self.omega = self.rpm*2*np.pi/60
        self.T = calc.T #3125.4 #
        self.Q = calc.Q #1725.619 #
        self.R_e= 0.4 * self.D
        self.xfar1 = np.linspace(-10 * self.D, 10 * self.D, self.
     XNum1)
        self.zfar1 = np.array(2*self.D)

        self.x1, self.y1, self.z1 = [], [], []
```

```python

        self.xfar2 = np.linspace(-100 * self.D, 100 * self.D, self.XNum2)
        self.zfar2 = np.array(10*self.D)

        self.PropXCoord=[ 0*self.D]
        self.PropZCoord=[ 2*self.D]

        self.x2, self.y2, self.z2 = [], [], []
        self.theta =np.linspace(0 , 2*np.pi , 360)
        self.x3, self.y3, self.z3 = [], [], []


        #self.eta=calc.

        self.m=2
        self.TNum=128
        self.AZNum = 3601
        self.Time = np.linspace(0, 2 * np.pi / (self.m * self.B * self.omega), self.TNum)  # time vector
        self.kc = self.m * self.B * self.omega / self.speedOfSound

        self.Re = 0.4* self.D
        self.Beta = np.sqrt(1- self.M_axial**2)


    def solve(self):
        print('Running Acoustic study option...')

        self.xyz_1(self.XNum1,self.ZNum1, self.xfar1)
        self.xyz_2(self.XNum2,self.ZNum2, self.xfar2)
        self.PT1 = np.zeros((self.TNum, self.ONum1), dtype=complex)
        self.PQ1 = np.zeros((self.TNum, self.ONum1), dtype=complex)

        self.TP1 = np.zeros((self.TNum, self.ONum1), dtype=complex)
        self.TPProp1 = np.zeros((self.TNum, self.ONum1), dtype=complex)
        self.TPrms1 = np.zeros(( self.ONum1), dtype=complex)
        self.SPL1 =   np.zeros(( self.ONum1), dtype=complex)
        self.PT2 = np.zeros((self.TNum, self.ONum2), dtype=complex)
        self.PQ2 = np.zeros((self.TNum, self.ONum2), dtype=complex)
        self.TP2 = np.zeros((self.TNum, self.ONum2), dtype=complex)
        self.TPProp2 = np.zeros((self.TNum, self.ONum2), dtype=complex)
        self.TPrms2 = np.zeros(( self.ONum2), dtype=complex)
        self.SPL2 =   np.zeros(( self.ONum2), dtype=complex)
        self.NearFieldObservers()
        self.FarFieldObservers()
        self.near=self.nearfieldmainfunc()
        self.TPrms1 = self.near[0]
        self.SPL1   = self.near[1]
        self.far =self.farfieldmainfunc()

        self.a = np.zeros((4, self.XNum1))  # Assuming the size of a is (5, ONum1)
        self.b = np.zeros((4, self.XNum2))  # Assuming the size of
```

```
b is (5, ONum2)

        self.a[3, :] = self.SPL1
        self.b[3, :] = self.SPL2

        # SPLAZ1 initialization
        self.SPLAZ1 = [None] * self.m  # Predefine a list for
    harmonic number dimensions
        self.SPLAZ2 = [None] * self.m

        self.TPrms2 = self.far[0]
        self.SPL2   = self.far[1]

        #print(SPL1.shape)
        self.SPLAZ1[0] = np.reshape(self.SPL1,  self.XNum1)
        self.SPLAZ2[0] = np.reshape(self.SPL2, self.XNum2)
        self.SPL1fun(self.SPLAZ1[0])
        self.SPL2_fun(self.SPLAZ2[0])
        self.c = np.zeros((4, self.XNum1))  # Assuming the size of
    a is (5, ONum1)
        self.d = np.zeros((4, self.XNum2))  # Assuming the size of
    b is (5, ONum2)

        self.c[3, :] = self.TPrms1
        self.d[3, :] = self.TPrms2
        self.TPrmsAZ1 = [None] * self.m  # Predefine a list for
    harmonic number dimensions
        self.TPrmsAZ2 = [None] * self.m
        # Loop through the harmonic numbers

        #print(SPL1.shape)
        self.TPrmsAZ1[0] = np.reshape(self.TPrms1, self.XNum1)
        self.TPrmsAZ2[0] = np.reshape(self.TPrms2, self.XNum2)


        self.TPrms1_fun(self.TPrmsAZ1[0])
        self.TPrms2_fun(self.TPrmsAZ2[0])
        print('Acoustic study finished...')



    def xyz_1(self, XNum1,ZNum1, xfar1 ):
        # Number of near field observers
        for j in range(self.XNum1):
            self.x1.append(self.xfar1[j])
            self.y1.append(-4 * self.D)
            self.z1.append(0)
        self.ONum1 = self.XNum1
        return self.x1,self.y1,self.z1,self.ONum1

    def xyz_2(self, XNum2,ZNum2, xfar2 ):
        # Number of far field observers
        for j in range(self.XNum2):
            self.x2.append(self.xfar2[j])
            self.y2.append(-100 * self.D)
            self.z2.append(0)
```

```
        self.ONum2 = self.XNum2
        return self.x2,self.y2,self.z2,self.ONum2

    def xyz_3(self,Re,theta):
        for i in range(self.theta):
            self.x3.append(0)
            self.y3.append(self.R_e * np.cos(self.theta))
            self.z3.append(self.R_e * np.sin(self.theta))
        self.ONum3 = len(self.theta)
        return self.x3,self.y3,self.z3,self.ONum3


    def NearFieldObservers(self):
        fig1 = plt.figure(3)
        ax1 = fig1.add_subplot(111, projection='3d')
        ax1.scatter(self.x1, self.y1, self.z1, c='r')
        ax1.set_title('Near Field Observers')

    def FarFieldObservers(self):
        fig2 = plt.figure(4)
        ax2 = fig2.add_subplot(111, projection='3d')
        ax2.scatter(self.x2, self.y2, self.z2, c='k')
        ax2.set_title('Far Field Observers')
        plt.show()

    def nearfieldmainfunc(self):
        pref = 2e-5
        # observers
        for j in range(self.ONum1):
            for t in range(self.TNum):
                    # observers
                self.ox = self.x1[j]
                self.oy = self.y1[j]
                self.oz = self.z1[j]
                # sources positions over time
                self.stheta = np.linspace(0, 2 * np.pi, self.AZNum)
                self.sx = self.PropXCoord[0]
                self.sy = self.Re * np.cos(self.stheta)
                self.sz = self.Re * np.sin(self.stheta) + self.PropXCoord[0]
                # distance calculation
                self.S = np.sqrt((self.ox - self.sx)**2 + self.Beta**2 * ((self.oy - self.sy)**2 + (self.oz - self.sz)**2))
                self.Sigma = (self.M_axial * (self.ox - self.sx) + self.S) / self.Beta**2
                # pressure components
                self.AS = self.T * (1j * self.kc * self.M_axial / self.Beta**2 + 1j * self.kc * self.ox / (self.Beta**2 * self.S) + self.ox / self.S**2) * np.exp(-1j * (self.m * self.B * self.stheta + self.kc * self.Sigma)) / self.S
                self.BS = self.Q * 1j * self.m * self.B / self.Re**2 * np.exp(-1j * (self.m * self.B * self.stheta + self.kc * self.Sigma)) / self.S
                # integration using the Trapezoidal Rule
                self.A_f = (2 * np.sum(self.AS) - self.AS[0] - self.AS[-1]) * (2 * np.pi / self.AZNum) / 2
```

```
                self.B_f = (2 * np.sum(self.BS) - self.BS[0] - self.BS[-1]) * (2 * np.pi / self.AZNum) / 2
                self.PT1[t, j] = np.exp(1j * self.m * self.B * self.omega * self.Time[t]) * self.A_f / (4 * np.pi**2)
        #          pdb.set_trace()
                self.PQ1[t, j] = -np.exp(1j * self.m * self.B * self.omega * self.Time[t]) * self.B_f / (4 * np.pi**2)
                self.TP1[t, j] = self.PT1[t, j] + self.PQ1[t, j]

                self.TPProp1[t, j] = np.sum(self.TP1[t, j])
 #           pdb.set_trace()
            self.TPrms1[j] = np.sqrt(np.mean(self.TPProp1[t, j]**2)) / np.sqrt(2)  # calculates rms value
            self.SPL1[j] = 20 * np.log10(self.TPrms1[j] / pref)  # converts the pressure to decibel
        return self.TPrms1 , self.SPL1

    def farfieldmainfunc(self):
        pref = 2e-5
        for j in range(self.ONum2):
            for t in range(self.TNum):
                # observers
                self.ox = self.x2[j]
                self.oy = self.y2[j]
                self.oz = self.z2[j]
                # sources positions over time
                self.stheta = np.linspace(0, 2 * np.pi, self.AZNum)
                self.sx = self.PropXCoord[0]
                self.sy = self.Re * np.cos(self.stheta)
                self.sz = self.PropZCoord[0] +self.Re * np.sin(self.stheta) #self.PropZCoord[j]*0 +
                #
                self.S0 = np.sqrt(self.ox**2 + self.Beta**2 * self.oy**2)
                #
                self.PT2[t, j] = np.exp(1j * self.m * self.B * self.omega * self.Time[t]) * self.T / (2 * np.pi * self.S0) * (1j)**(self.m * self.B) * np.exp(-1j * self.kc / self.Beta**2 * (self.S0 + self.M_axial * self.ox)) * 1j * self.kc / self.Beta**2 * (self.M_axial + self.ox / self.S0) * scipy.special.jn(self.m * self.B, self.kc * self.Re * self.oy / self.S0)
                self.PQ2[t, j] = -np.exp(1j * self.m * self.B * self.omega * self.Time[t]) * self.Q / (2 * np.pi * self.S0 * self.Re**2) * (1j)**(self.m * self.B + 1) * self.m * self.B * np.exp(-1j * self.kc / self.Beta**2 * (self.S0 + self.M_axial * self.ox)) * scipy.special.jn(self.m * self.B, self.kc * self.Re * self.oy / self.S0)
                self.TP2[t, j] = self.PT2[t, j] + self.PQ2[t, j]
                self.TPProp2[t, j] = np.sum(self.TP2[t, j])  # sum of both propellers pressure
#            pdb.set_trace()

            self.TPrms2[j] = np.sqrt(np.mean(self.TPProp2[t, j]**2)) / np.sqrt(2)  # calculate rms
            self.SPL2[j] = 20 * np.log10(self.TPrms2[j] / pref)  # converts to decibel
```

```
        return self.TPrms2,self.SPL2

    def SPL1fun(self,SPLAZ1):
#          pdb.set_trace()
        plt.figure(5)
        plt.grid(True)
        plt.plot(self.xfar1, SPLAZ1, 'r')
        plt.xlabel('x, [m]')
        plt.ylabel('SPL near field, dB')
        plt.xticks(fontname='Times New Roman', fontsize=16)
        plt.yticks(fontname='Times New Roman', fontsize=16)
        plt.box(False)
        plt.show()

    def SPL2_fun(self,SPLAZ2):
        plt.figure(6)
        plt.grid(True)
        plt.plot(self.xfar2, SPLAZ2, 'r')  # Use xfar1 instead of x1
        plt.xlabel('x, [m]')
        plt.ylabel('SPL far field, dB')
        plt.xticks(fontname='Times New Roman', fontsize=16)
        plt.yticks(fontname='Times New Roman', fontsize=16)
        plt.box(False)
        plt.show()

    def TPrms1_fun(self,TPrmsAZ1):
        plt.figure(7)
        plt.grid(True)
        plt.plot(self.xfar1, TPrmsAZ1, 'r')
        plt.xlabel('x, [m]')
        plt.ylabel('TPrms near field')
        plt.xticks(fontname='Times New Roman', fontsize=16)
        plt.yticks(fontname='Times New Roman', fontsize=16)
        plt.box(False)
        plt.show()

    def TPrms2_fun(self,TPrmsAZ2):
        plt.figure(8)
        plt.grid(True)
        plt.plot(self.xfar2, TPrmsAZ2, 'r')
        plt.xlabel('x, [m]')
        plt.ylabel('TPrms far field')
        plt.xticks(fontname='Times New Roman', fontsize=16)
        plt.yticks(fontname='Times New Roman', fontsize=16)
        plt.box(False)
        plt.show()

   # def prmsNearFieldSphere(self):
```

Listing C.1: Acoustic Study

D

Appendix 4

Table D.1: Baseline acoustic data gathered through the use of CFD+FW-H

Polar Angle (Radians)	SPL (dB)
0.1256637	8.422101
0.1884956	29.29372
0.2513274	43.92092
0.3141593	55.0761
0.3769911	63.99216
0.439823	71.32462
0.5026548	77.46245
0.5654867	82.65432
0.6283185	87.06778
0.6911504	90.82039
0.7539822	93.99747
0.8168141	96.66359
0.8796459	98.87098
0.9424778	100.6676
1.00531	102.1068
1.068142	103.2602
1.130973	104.2327
1.193805	105.1667
1.256637	106.2107
1.319469	107.4465
1.382301	108.8368
1.445133	110.2613
1.507964	111.5931
1.570796	112.7419
1.633628	113.6559
1.69646	114.3087
1.759292	114.6878
1.822124	114.7869
1.884956	114.6014
1.947787	114.1259
2.010619	113.3532
2.073451	112.2735
2.136283	110.8731
2.199115	109.1343
2.261947	107.034
2.324779	104.5427
2.38761	101.6219
2.450442	98.22186
2.513274	94.27713
2.576106	89.69999
2.638938	84.37006
2.70177	78.11701
2.764602	70.68964
2.827433	61.6968
2.890265	50.48117
3.015929	14.90499

E

Appendix 5

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 13 15:37:47 2023

@author: georg
"""

# -*- coding: utf-8 -*-

import sys
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import scipy.special
#from scipy.special import jn  # Bessel function in SciPy
import pdb
class SoundPressureField:
    def __init__(self, m, B, D, M, c0, rpm, T, Q, x_range, y_range, num_points=100):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.c0 = c0
        self.rpm = rpm
        self.T = T
        self.Q = Q
        self.x_range = x_range
        self.y_range = y_range
        self.num_points = num_points

        # Generate x and y meshgrid
        self.x, self.y = np.meshgrid(
            np.linspace(-self.x_range * D, self.x_range * D, self.num_points),
            np.linspace(0, self.y_range * D, self.num_points)
        )

        self.omega = 2 * np.pi * self.rpm / 60  # Angular speed in radians per second
        self.k = self.m * self.B * self.omega / self.c0  # Wave number
        self.beta = np.sqrt(1 - self.M**2)
        self.R_e = 0.8 * (self.D / 2)  # Effective radius of the
```

```
source
        self.z = 100 * self.D  # Distance to the observation plane
        self.omega1 = self.B * self.omega

        self.prmsFarField = None
        self.SPLFarField = None


    def compute_fields(self):

        nu = self.m * self.B  # Order of Bessel function

        # Initialize output arrays with zeros
        self.prmsFarField = np.zeros_like(self.x)
        self.SPLFarField = np.zeros_like(self.x)
        self.pref = 2e-5  # Reference pressure (20 * 10^(-6) Pa)

        # Compute fields
        for i in range(self.x.shape[0]):
            for j in range(self.x.shape[1]):
                S0 = np.sqrt(self.x[i, j]**2 + self.y[i, j]**2 + self.beta**2 * self.z**2)
                Z = self.k * self.z * self.R_e / S0  # Argument for the Bessel function

                # Calculate the root-mean-square pressure level and SPL
                self.prmsFarField[i, j] = self.m * self.omega1 / (np.sqrt(8) * np.pi * self.c0 * S0) * \
                    np.abs(self.T * (self.M + self.x[i, j] / S0) / self.beta**2 - self.Q * self.B * self.c0 / (self.omega1 * self.R_e**2)) * \
                    jv(nu, Z)

                self.SPLFarField[i, j] = 20 * np.log10(self.prmsFarField[i, j] / self.pref)

    def plot_spl(self):
        if self.prmsFarField is None or self.SPLFarField is None:
            raise ValueError("Fields have not been computed. Call compute_fields() first.")

        y_fixed_index = self.SPLFarField.shape[0] // 2  # Choose the middle of the y range for visualization

        # Create a figure with two subplots
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10), sharex=True)

        # Plot SPL vs x
        ax1.plot(self.x[0, :], self.SPLFarField[y_fixed_index, :], label='SPL vs x at y = {:.2f}D'.format(self.y[0, y_fixed_index]))
        ax1.set_ylabel('SPL (dB)')
        ax1.set_title('Sound Pressure Level (SPL) and RMS Pressure as a Function of x')
```

```
        ax1.legend()
        ax1.grid(True)

        # Plot prmsFarField vs x
        ax2.plot(self.x[0, :], self.prmsFarField[y_fixed_index, :], color='orange', label='PRMS vs x at y = {:.2f}D'.format(self.y[0, y_fixed_index]))
        ax2.set_xlabel('x (m)')
        ax2.set_ylabel('PRMS')
        ax2.legend()
        ax2.grid(True)

        plt.tight_layout()
        plt.show()

    def run(self):
        self.compute_fields()
        self.plot_spl()
#################################################################
#################################################################
class NearFieldSoundPressure:
    def __init__(self, m, B, D, M, T, Q, rpm, c0, Phi):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.T = T
        self.Q = Q
        self.rpm = rpm
        self.c0 = c0
        self.Phi = Phi

        # Precompute values used across methods
        self.omega = 2 * np.pi * self.rpm / 60
        self.k = self.m * self.B * self.omega / self.c0
        self.beta = np.sqrt(1 - self.M**2)
        self.Re = 0.8 * (self.D / 2)
        self.pref = 2e-5

    def calculate_prms_and_SPL(self):
        prmsNearField = np.zeros(len(self.Phi))
        SPLNearField = np.zeros(len(self.Phi))

        # Define the parameterized functions x_l, y_l, z_l
        x_l = lambda theta: 0
        y_l = lambda theta: self.Re * np.cos(theta)
        z_l = lambda theta: self.Re * np.sin(theta)

        for i in range(len(self.Phi)):
            x = 2 * self.D * np.cos(self.Phi[i])
            y = 2 * self.D * np.sin(self.Phi[i])
            z = 0

            # Define the S and sigma functions
            S = lambda theta: np.sqrt((x - x_l(theta))**2 + self.beta**2 * ((y - y_l(theta))**2 + (z - z_l(theta))**2))
```

```python
            sigma = lambda theta: (self.M * (x - x_l(theta)) + S(theta)) / self.beta**2

            # Define the integrand functions funA and funB
            funA = lambda theta: (
                self.T * x / S(theta)**2 * np.cos(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.sin(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            funB = lambda theta: (
                -self.T * x / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.cos(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            # Perform the integration
            intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
            intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)

            # Calculate prmsNearField and SPLNearField
            prmsNearField[i] = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
            SPLNearField[i] = 20 * np.log10(prmsNearField[i] / self.pref)

        self.prmsNearField = prmsNearField
        self.SPLNearField = SPLNearField

    def calculate_spl_vs_x(self, x_values):
        SPL_vs_x = np.zeros_like(x_values)

        # Define fixed Phi for simplicity
        phi_fixed = np.pi / 4  # For example, take Phi = 45 degrees

        for idx, x in enumerate(x_values):
            y = 0  # Assuming y = 0 for the sake of simplicity
            z = 0

            # Define the S and sigma functions
            S = lambda theta: np.sqrt((x - 0)**2 + self.beta**2 * (y**2 + z**2))
            sigma = lambda theta: (self.M * (x - 0) + S(theta)) / self.beta**2

            # Define the integrand functions funA and funB
            funA = lambda theta: (
                self.T * x / S(theta)**2 * np.cos(self.m * self.B *
```

```
 theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.sin(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            funB = lambda theta: (
                -self.T * x / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.cos(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            # Perform the integration
            intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
            intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)

            # Calculate prms and SPL
            prms = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
            SPL_vs_x[idx] = 20 * np.log10(prms / self.pref)

        return SPL_vs_x

    def plot_results(self):
        # Provided SPL and Phi values for comparison
        spl_provided = [
            8.422101, 29.29372, 43.92092, 55.0761, 63.99216, 71.32462,
            77.46245, 82.65432, 87.06778, 90.82039, 93.99747, 96.66359,
            98.87098, 100.6676, 102.1068, 103.2602, 104.2327, 105.1667,
            106.2107, 107.4465, 108.8368, 110.2613, 111.5931, 112.7419,
            113.6559, 114.3087, 114.6878, 114.7869, 114.6014, 114.1259,
            113.3532, 112.2735, 110.8731, 109.1343, 107.034, 104.5427,
            101.6219, 98.22186, 94.27713, 89.69999, 84.37006, 78.11701,
            70.68964, 61.6968, 50.48117, 35.8084, 14.90499
            ]

        phi_provided = [
            0.1256637, 0.1884956, 0.2513274, 0.3141593, 0.3769911, 0.439823,
            0.5026548, 0.5654867, 0.6283185, 0.6911504, 0.7539822, 0.8168141,
            0.8796459, 0.9424778, 1.00531, 1.068142, 1.130973, 1.193805,
```

```
            1.256637, 1.319469, 1.382301, 1.445133, 1.507964, 1.570796,
            1.633628, 1.69646, 1.759292, 1.822124, 1.884956, 1.947787,
            2.010619, 2.073451, 2.136283, 2.199115, 2.261947, 2.324779,
            2.38761, 2.450442, 2.513274, 2.576106, 2.638938, 2.70177,
            2.764602, 2.827433, 2.890265, 2.953097, 3.015929
            ]

    # Plotting
        fig = plt.figure(figsize=(14, 6))

        # Polar plot
        ax1 = fig.add_subplot(121, polar=True)
        ax1.plot(self.Phi, self.SPLNearField, linestyle='-', color='b', label='Calculated SPL')
        ax1.set_title("SPL vs Polar Angle")
        ax1.set_ylim(0, max(max(self.SPLNearField), max(spl_provided)) + 10)  # Set radial limits based on max SPL
        ax1.set_thetamin(0)
        ax1.set_thetamax(180)
        ax1.set_rticks([10, 30, 50, 70, 90, 110, 130, 150])  # Example ticks
        ax1.set_yticklabels(["10", "30", "50", "70", "90", "110", "130", "150"])
        ax1.legend(loc='upper right')

        # SPL vs X-Axis plot
        x_values = np.linspace(-2 * self.D, 2 * self.D, 100)
        SPL_vs_x = self.calculate_spl_vs_x(x_values)

        ax2 = fig.add_subplot(122)
        ax2.plot(x_values, SPL_vs_x, color='g', linestyle='-')
        ax2.set_title("SPL vs X-Axis")
        ax2.set_xlabel("X-Axis (Distance)")
        ax2.set_ylabel("SPL (dB)")
        ax2.grid(True)

        plt.tight_layout()
        plt.show()

    def run(self):
        self.calculate_prms_and_SPL()
        self.plot_results()
#################################################################
#################################################################
class NearFieldSoundPressure:
    def __init__(self, m, B, D, M, T, Q, rpm, c0, NumOfObserver):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.T = T
        self.Q = Q
```

```python
        self.rpm = rpm
        self.c0 = c0

        # Generate Phi using NumOfObserver
        self.Phi = np.linspace(0, np.pi, NumOfObserver)

        # Precompute values used across methods
        self.omega = 2 * np.pi * self.rpm / 60
        self.k = self.m * self.B * self.omega / self.c0
        self.beta = np.sqrt(1 - self.M**2)
        self.Re = 0.8 * (self.D / 2)
        self.pref = 2e-5

    def calculate_prms_and_SPL(self):
        prmsNearField = np.zeros(len(self.Phi))
        SPLNearField = np.zeros(len(self.Phi))

        # Define the parameterized functions x_l, y_l, z_l
        x_l = lambda theta: 0
        y_l = lambda theta: self.Re * np.cos(theta)
        z_l = lambda theta: self.Re * np.sin(theta)

        for i in range(len(self.Phi)):
            x = 2 * self.D * np.cos(self.Phi[i])
            y = 2 * self.D * np.sin(self.Phi[i])
            z = 0

            # Define the S and sigma functions
            S = lambda theta: np.sqrt((x - x_l(theta))**2 + self.beta**2 * ((y - y_l(theta))**2 + (z - z_l(theta))**2))
            sigma = lambda theta: (self.M * (x - x_l(theta)) + S(theta)) / self.beta**2

            # Define the integrand functions funA and funB
            funA = lambda theta: (
                self.T * x / S(theta)**2 * np.cos(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.sin(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            funB = lambda theta: (
                -self.T * x / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.cos(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            # Perform the integration
            intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
            intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6, epsabs
```

```
=1e-6)

            # Calculate prmsNearField and SPLNearField
            prmsNearField[i] = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
            SPLNearField[i] = 20 * np.log10(prmsNearField[i] / self.pref)

        self.prmsNearField = prmsNearField
        self.SPLNearField = SPLNearField

    def plot_results(self):
        # Provided spl and phi values for comparison
        spl_provided = [
            8.422101, 29.29372, 43.92092, 55.0761, 63.99216, 71.32462,
            77.46245, 82.65432, 87.06778, 90.82039, 93.99747, 96.66359,
            98.87098, 100.6676, 102.1068, 103.2602, 104.2327, 105.1667,
            106.2107, 107.4465, 108.8368, 110.2613, 111.5931, 112.7419,
            113.6559, 114.3087, 114.6878, 114.7869, 114.6014, 114.1259,
            113.3532, 112.2735, 110.8731, 109.1343, 107.034, 104.5427,
            101.6219, 98.22186, 94.27713, 89.69999, 84.37006, 78.11701,
            70.68964, 61.6968, 50.48117, 35.8084, 14.90499
            ]

        phi_provided = [
            0.1256637, 0.1884956, 0.2513274, 0.3141593, 0.3769911, 0.439823,
            0.5026548, 0.5654867, 0.6283185, 0.6911504, 0.7539822, 0.8168141,
            0.8796459, 0.9424778, 1.00531, 1.068142, 1.130973, 1.193805,
            1.256637, 1.319469, 1.382301, 1.445133, 1.507964, 1.570796,
            1.633628, 1.69646, 1.759292, 1.822124, 1.884956, 1.947787,
            2.010619, 2.073451, 2.136283, 2.199115, 2.261947, 2.324779,
            2.38761, 2.450442, 2.513274, 2.576106, 2.638938, 2.70177,
            2.764602, 2.827433, 2.890265, 2.953097, 3.015929
            ]

    # Create a figure with two subplots (polar plots)
        fig, axs = plt.subplots(2, 1, subplot_kw=dict(projection='polar'))

    # First subplot: SPL vs Phi
        axs[0].plot(self.Phi, self.SPLNearField, marker='o', linestyle='-', color='b', label='Calculated SPL')
```

```
        axs[0].set_title("SPL vs Polar Angle")
        axs[0].set_ylim(0, max(max(self.SPLNearField), max(spl_provided)) + 10)  # Set radial limits based on max SPL
        axs[0].set_thetamin(0)
        axs[0].set_thetamax(180)
        axs[0].set_rticks([10, 30, 50, 70, 90, 110, 130, 150])  # Example ticks
        axs[0].set_yticklabels(["10", "30", "50", "70", "90", "110", "130", "150"])
        axs[0].legend(loc='upper right')

    # Second subplot: prmsNearField vs Phi
        axs[1].plot(self.Phi, self.prmsNearField, marker='o', linestyle='-', color='g', label='Calculated prmsNearField')
        axs[1].set_title("prmsNearField vs Polar Angle")
        axs[1].set_ylim(0, max(self.prmsNearField) + 0.1 * max(self.prmsNearField))  # Set radial limits based on max prmsNearField
        axs[1].set_thetamin(0)
        axs[1].set_thetamax(180)
        #axs[1].set_rticks([0.01, 0.05, 0.1, 0.15, 0.2, 0.25])  # Adjust these values based on your data
        #axs[1].set_yticklabels(["0.01", "0.05", "0.1", "0.15", "0.2", "0.25"])
        axs[1].legend(loc='upper right')

    # Adjust layout to prevent overlap
        plt.tight_layout()

    # Show the plot
        plt.show()
#################################################################
#################################################################
class NearFieldSoundPressureSphere:
    def __init__(self, m, B, D, M, T, Q, rpm, c0, azimuth, elevation):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.T = T
        self.Q = Q
        self.rpm = rpm
        self.c0 = c0
        self.azimuth = azimuth
        self.elevation = elevation

        # Precompute values used across methods
        self.omega = 2 * np.pi * self.rpm / 60
        self.k = self.m * self.B * self.omega / self.c0
        self.beta = np.sqrt(1 - self.M**2)
        self.Re = 0.8 * (self.D / 2)
        self.pref = 2e-5

        # Initialize result arrays
        self.prmsNearFieldSphere = np.zeros((len(self.azimuth), len(self.elevation)))
```

```
        self.SPLNearFieldSphere = np.zeros((len(self.azimuth), len(self.elevation)))

        # Initialize field point coordinates
        self.x = np.zeros((len(self.azimuth), len(self.elevation)))
        self.y = np.zeros((len(self.azimuth), len(self.elevation)))
        self.z = np.zeros((len(self.azimuth), len(self.elevation)))

    def calculate_prms_and_SPL(self):
        # Define the parameterized functions x_l, y_l, z_l
        x_l = lambda theta: 0
        y_l = lambda theta: self.Re * np.cos(theta)
        z_l = lambda theta: self.Re * np.sin(theta)

        for i in range(len(self.azimuth)):
            for j in range(len(self.elevation)):
                # Field points, default radius 2D
                self.x[i, j] = 2 * self.D * np.cos(self.elevation[j])
                self.y[i, j] = 2 * self.D * np.cos(self.azimuth[i]) * np.sin(self.elevation[j])
                self.z[i, j] = 2 * self.D * np.sin(self.azimuth[i]) * np.sin(self.elevation[j])

                # Define the S and sigma functions
                S = lambda theta: np.sqrt((self.x[i, j] - x_l(theta))**2 + self.beta**2 * ((self.y[i, j] - y_l(-theta))**2 + (self.z[i, j] - z_l(theta))**2))
                sigma = lambda theta: (self.M * (self.x[i, j] - x_l(theta)) + S(theta)) / self.beta**2

                # Define the integrand functions funA and funB
                funA = lambda theta: (
                    self.T * self.x[i, j] / S(theta)**2 * np.cos(self.m * self.B * theta + self.k * sigma(theta)) +
                    (self.T * self.k / self.beta**2 * (self.M + self.x[i, j] / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                    np.sin(self.m * self.B * theta + self.k * sigma(theta))
                ) / S(theta)

                funB = lambda theta: (
                    -self.T * self.x[i, j] / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                    (self.T * self.k / self.beta**2 * (self.M + self.x[i, j] / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                    np.cos(self.m * self.B * theta + self.k * sigma(theta))
                ) / S(theta)

                # Perform the integration
                intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
                intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6,
```

```
                epsabs=1e-6)

                # Calculate prmsNearFieldSphere and SPLNearFieldSphere
                self.prmsNearFieldSphere[i, j] = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
                self.SPLNearFieldSphere[i, j] = 20 * np.log10(self.prmsNearFieldSphere[i, j] / self.pref)

    def plot_results(self):
        # Convert azimuth and elevation to radians if not already
        azimuth_rad = self.azimuth
        elevation_rad = self.elevation

        # Create a meshgrid for spherical coordinates
        azimuth_grid, elevation_grid = np.meshgrid(azimuth_rad, elevation_rad)

        # Convert spherical to Cartesian coordinates for plotting
        X = np.sin(elevation_grid) * np.cos(azimuth_grid)
        Y = np.sin(elevation_grid) * np.sin(azimuth_grid)
        Z = np.cos(elevation_grid)

        # Plot SPL on a 3D spherical surface
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

        # Use surface plot to map SPL to the spherical coordinates
        surf = ax.plot_surface(X, Y, Z, facecolors=plt.cm.viridis(self.SPLNearFieldSphere.T / np.max(self.SPLNearFieldSphere)), rstride=1, cstride=1, antialiased=False, shade=False)

        # Customize the plot
        ax.set_title("SPL Near Field - Spherical Coordinates")
        ax.set_xlabel("z")
        ax.set_ylabel("Y")
        ax.set_zlabel("x")

        # Add a color bar for SPL values
        m = plt.cm.ScalarMappable(cmap=plt.cm.viridis)
        m.set_array(self.SPLNearFieldSphere)
        plt.colorbar(m, shrink=0.5, aspect=5)

        plt.show()

    def run(self):
        self.calculate_prms_and_SPL()
        self.plot_results()
##################################################################
##################################################################
# Define D (you can set D to any value you want)
D = 1  # Example: D = 1, you can change it according to your needs

# Generate 100 observer points along the x-axis between -100D and 100D
x_observers = np.linspace(-100 * D, 100 * D, 50)
```

```
y_observers = np.zeros(50)  # All y coordinates are 0 since points are on x-axis
z_observers = np.zeros(50)  # All z coordinates are 0 since points are on x-axis

# Propeller location
propeller_position = np.array([0, 0, 100 * D])

# Plot the observer points and the propeller location
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot observer points
ax.scatter(x_observers, y_observers, z_observers, color='blue', label='Observers')

# Plot propeller location
ax.scatter(*propeller_position, color='red', label='Propeller')

# Annotate the propeller location
ax.text(*propeller_position, 'Propeller', color='red')

# Set labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
ax.set_title('Observer Points and Propeller Location')

# Set limits for better visualization
ax.set_xlim([-100 * D, 100 * D])
ax.set_ylim([-100 * D, 100 * D])
ax.set_zlim([0, 100 * D])

# Show legend
ax.legend()

# Show the plot
plt.show()
```

Listing E.1: Acoustic Study

F

Appendix 5

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 13 15:37:47 2023

@author: georg
"""

# -*- coding: utf-8 -*-

import sys
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import scipy.special
#from scipy.special import jn  # Bessel function in SciPy
import pdb
class SoundPressureField:
    def __init__(self, m, B, D, M, c0, rpm, T, Q, x_range, y_range, num_points=100):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.c0 = c0
        self.rpm = rpm
        self.T = T
        self.Q = Q
        self.x_range = x_range
        self.y_range = y_range
        self.num_points = num_points

        # Generate x and y meshgrid
        self.x, self.y = np.meshgrid(
            np.linspace(-self.x_range * D, self.x_range * D, self.num_points),
            np.linspace(0, self.y_range * D, self.num_points)
        )

        self.omega = 2 * np.pi * self.rpm / 60  # Angular speed in radians per second
        self.k = self.m * self.B * self.omega / self.c0  # Wave number
        self.beta = np.sqrt(1 - self.M**2)
        self.R_e = 0.8 * (self.D / 2)  # Effective radius of the
```

```
source
        self.z = 100 * self.D  # Distance to the observation plane
        self.omega1 = self.B * self.omega

        self.prmsFarField = None
        self.SPLFarField = None


    def compute_fields(self):

        nu = self.m * self.B  # Order of Bessel function

        # Initialize output arrays with zeros
        self.prmsFarField = np.zeros_like(self.x)
        self.SPLFarField = np.zeros_like(self.x)
        self.pref = 2e-5  # Reference pressure (20 * 10^(-6) Pa)

        # Compute fields
        for i in range(self.x.shape[0]):
            for j in range(self.x.shape[1]):
                S0 = np.sqrt(self.x[i, j]**2 + self.y[i, j]**2 + self.beta**2 * self.z**2)
                Z = self.k * self.z * self.R_e / S0  # Argument for the Bessel function

                # Calculate the root-mean-square pressure level and SPL
                self.prmsFarField[i, j] = self.m * self.omega1 / (np.sqrt(8) * np.pi * self.c0 * S0) * \
                    np.abs(self.T * (self.M + self.x[i, j] / S0) / self.beta**2 - self.Q * self.B * self.c0 / (self.omega1 * self.R_e**2)) * \
                    jv(nu, Z)

                self.SPLFarField[i, j] = 20 * np.log10(self.prmsFarField[i, j] / self.pref)

    def plot_spl(self):
        if self.prmsFarField is None or self.SPLFarField is None:
            raise ValueError("Fields have not been computed. Call compute_fields() first.")

        y_fixed_index = self.SPLFarField.shape[0] // 2  # Choose the middle of the y range for visualization

        # Create a figure with two subplots
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10), sharex=True)

        # Plot SPL vs x
        ax1.plot(self.x[0, :], self.SPLFarField[y_fixed_index, :], label='SPL vs x at y = {:.2f}D'.format(self.y[0, y_fixed_index]))
        ax1.set_ylabel('SPL (dB)')
        ax1.set_title('Sound Pressure Level (SPL) and RMS Pressure as a Function of x')
```

```
        ax1.legend()
        ax1.grid(True)

        # Plot prmsFarField vs x
        ax2.plot(self.x[0, :], self.prmsFarField[y_fixed_index, :],
    color='orange', label='PRMS vs x at y = {:.2f}D'.format(self.y
    [0, y_fixed_index]))
        ax2.set_xlabel('x (m)')
        ax2.set_ylabel('PRMS')
        ax2.legend()
        ax2.grid(True)

        plt.tight_layout()
        plt.show()

    def run(self):
        self.compute_fields()
        self.plot_spl()
##################################################################
##################################################################
class NearFieldSoundPressure:
    def __init__(self, m, B, D, M, T, Q, rpm, c0, Phi):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.T = T
        self.Q = Q
        self.rpm = rpm
        self.c0 = c0
        self.Phi = Phi

        # Precompute values used across methods
        self.omega = 2 * np.pi * self.rpm / 60
        self.k = self.m * self.B * self.omega / self.c0
        self.beta = np.sqrt(1 - self.M**2)
        self.Re = 0.8 * (self.D / 2)
        self.pref = 2e-5

    def calculate_prms_and_SPL(self):
        prmsNearField = np.zeros(len(self.Phi))
        SPLNearField = np.zeros(len(self.Phi))

        # Define the parameterized functions x_l, y_l, z_l
        x_l = lambda theta: 0
        y_l = lambda theta: self.Re * np.cos(theta)
        z_l = lambda theta: self.Re * np.sin(theta)

        for i in range(len(self.Phi)):
            x = 2 * self.D * np.cos(self.Phi[i])
            y = 2 * self.D * np.sin(self.Phi[i])
            z = 0

            # Define the S and sigma functions
            S = lambda theta: np.sqrt((x - x_l(theta))**2 + self.
    beta**2 * ((y - y_l(theta))**2 + (z - z_l(theta))**2))
```

```
            sigma = lambda theta: (self.M * (x - x_l(theta)) + S(theta)) / self.beta**2

            # Define the integrand functions funA and funB
            funA = lambda theta: (
                self.T * x / S(theta)**2 * np.cos(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.sin(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            funB = lambda theta: (
                -self.T * x / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.cos(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            # Perform the integration
            intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
            intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)

            # Calculate prmsNearField and SPLNearField
            prmsNearField[i] = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
            SPLNearField[i] = 20 * np.log10(prmsNearField[i] / self.pref)

        self.prmsNearField = prmsNearField
        self.SPLNearField = SPLNearField

    def calculate_spl_vs_x(self, x_values):
        SPL_vs_x = np.zeros_like(x_values)

        # Define fixed Phi for simplicity
        phi_fixed = np.pi / 4  # For example, take Phi = 45 degrees

        for idx, x in enumerate(x_values):
            y = 0  # Assuming y = 0 for the sake of simplicity
            z = 0

            # Define the S and sigma functions
            S = lambda theta: np.sqrt((x - 0)**2 + self.beta**2 * (y**2 + z**2))
            sigma = lambda theta: (self.M * (x - 0) + S(theta)) / self.beta**2

            # Define the integrand functions funA and funB
            funA = lambda theta: (
                self.T * x / S(theta)**2 * np.cos(self.m * self.B *
```

```
 theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.sin(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            funB = lambda theta: (
                -self.T * x / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.cos(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            # Perform the integration
            intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
            intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)

            # Calculate prms and SPL
            prms = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
            SPL_vs_x[idx] = 20 * np.log10(prms / self.pref)

        return SPL_vs_x

    def plot_results(self):
        # Provided SPL and Phi values for comparison
        spl_provided = [
            8.422101, 29.29372, 43.92092, 55.0761, 63.99216, 71.32462,
            77.46245, 82.65432, 87.06778, 90.82039, 93.99747, 96.66359,
            98.87098, 100.6676, 102.1068, 103.2602, 104.2327, 105.1667,
            106.2107, 107.4465, 108.8368, 110.2613, 111.5931, 112.7419,
            113.6559, 114.3087, 114.6878, 114.7869, 114.6014, 114.1259,
            113.3532, 112.2735, 110.8731, 109.1343, 107.034, 104.5427,
            101.6219, 98.22186, 94.27713, 89.69999, 84.37006, 78.11701,
            70.68964, 61.6968, 50.48117, 35.8084, 14.90499
            ]

        phi_provided = [
            0.1256637, 0.1884956, 0.2513274, 0.3141593, 0.3769911, 0.439823,
            0.5026548, 0.5654867, 0.6283185, 0.6911504, 0.7539822, 0.8168141,
            0.8796459, 0.9424778, 1.00531, 1.068142, 1.130973, 1.193805,
```

```
            1.256637, 1.319469, 1.382301, 1.445133, 1.507964, 1.570796,
            1.633628, 1.69646, 1.759292, 1.822124, 1.884956, 1.947787,
            2.010619, 2.073451, 2.136283, 2.199115, 2.261947, 2.324779,
            2.38761, 2.450442, 2.513274, 2.576106, 2.638938, 2.70177,
            2.764602, 2.827433, 2.890265, 2.953097, 3.015929
            ]

    # Plotting
        fig = plt.figure(figsize=(14, 6))

        # Polar plot
        ax1 = fig.add_subplot(121, polar=True)
        ax1.plot(self.Phi, self.SPLNearField, linestyle='-', color='b', label='Calculated SPL')
        ax1.set_title("SPL vs Polar Angle")
        ax1.set_ylim(0, max(max(self.SPLNearField), max(spl_provided)) + 10)  # Set radial limits based on max SPL
        ax1.set_thetamin(0)
        ax1.set_thetamax(180)
        ax1.set_rticks([10, 30, 50, 70, 90, 110, 130, 150])  # Example ticks
        ax1.set_yticklabels(["10", "30", "50", "70", "90", "110", "130", "150"])
        ax1.legend(loc='upper right')

        # SPL vs X-Axis plot
        x_values = np.linspace(-2 * self.D, 2 * self.D, 100)
        SPL_vs_x = self.calculate_spl_vs_x(x_values)

        ax2 = fig.add_subplot(122)
        ax2.plot(x_values, SPL_vs_x, color='g', linestyle='-')
        ax2.set_title("SPL vs X-Axis")
        ax2.set_xlabel("X-Axis (Distance)")
        ax2.set_ylabel("SPL (dB)")
        ax2.grid(True)

        plt.tight_layout()
        plt.show()

    def run(self):
        self.calculate_prms_and_SPL()
        self.plot_results()
#################################################################
#################################################################
class NearFieldSoundPressure:
    def __init__(self, m, B, D, M, T, Q, rpm, c0, NumOfObserver):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.T = T
        self.Q = Q
```

```
        self.rpm = rpm
        self.c0 = c0

        # Generate Phi using NumOfObserver
        self.Phi = np.linspace(0, np.pi, NumOfObserver)

        # Precompute values used across methods
        self.omega = 2 * np.pi * self.rpm / 60
        self.k = self.m * self.B * self.omega / self.c0
        self.beta = np.sqrt(1 - self.M**2)
        self.Re = 0.8 * (self.D / 2)
        self.pref = 2e-5

    def calculate_prms_and_SPL(self):
        prmsNearField = np.zeros(len(self.Phi))
        SPLNearField = np.zeros(len(self.Phi))

        # Define the parameterized functions x_l, y_l, z_l
        x_l = lambda theta: 0
        y_l = lambda theta: self.Re * np.cos(theta)
        z_l = lambda theta: self.Re * np.sin(theta)

        for i in range(len(self.Phi)):
            x = 2 * self.D * np.cos(self.Phi[i])
            y = 2 * self.D * np.sin(self.Phi[i])
            z = 0

            # Define the S and sigma functions
            S = lambda theta: np.sqrt((x - x_l(theta))**2 + self.beta**2 * ((y - y_l(theta))**2 + (z - z_l(theta))**2))
            sigma = lambda theta: (self.M * (x - x_l(theta)) + S(theta)) / self.beta**2

            # Define the integrand functions funA and funB
            funA = lambda theta: (
                self.T * x / S(theta)**2 * np.cos(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.sin(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            funB = lambda theta: (
                -self.T * x / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                (self.T * self.k / self.beta**2 * (self.M + x / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                np.cos(self.m * self.B * theta + self.k * sigma(theta))
            ) / S(theta)

            # Perform the integration
            intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
            intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6, epsabs
```

```
=1e-6)

            # Calculate prmsNearField and SPLNearField
            prmsNearField[i] = np.sqrt(2) / (8 * np.pi**2) * np.sqrt(intA**2 + intB**2)
            SPLNearField[i] = 20 * np.log10(prmsNearField[i] / self.pref)

        self.prmsNearField = prmsNearField
        self.SPLNearField = SPLNearField

    def plot_results(self):
        # Provided spl and phi values for comparison
        spl_provided = [
            8.422101, 29.29372, 43.92092, 55.0761, 63.99216, 71.32462,
            77.46245, 82.65432, 87.06778, 90.82039, 93.99747, 96.66359,
            98.87098, 100.6676, 102.1068, 103.2602, 104.2327, 105.1667,
            106.2107, 107.4465, 108.8368, 110.2613, 111.5931, 112.7419,
            113.6559, 114.3087, 114.6878, 114.7869, 114.6014, 114.1259,
            113.3532, 112.2735, 110.8731, 109.1343, 107.034, 104.5427,
            101.6219, 98.22186, 94.27713, 89.69999, 84.37006, 78.11701,
            70.68964, 61.6968, 50.48117, 35.8084, 14.90499
            ]

        phi_provided = [
            0.1256637, 0.1884956, 0.2513274, 0.3141593, 0.3769911, 0.439823,
            0.5026548, 0.5654867, 0.6283185, 0.6911504, 0.7539822, 0.8168141,
            0.8796459, 0.9424778, 1.00531, 1.068142, 1.130973, 1.193805,
            1.256637, 1.319469, 1.382301, 1.445133, 1.507964, 1.570796,
            1.633628, 1.69646, 1.759292, 1.822124, 1.884956, 1.947787,
            2.010619, 2.073451, 2.136283, 2.199115, 2.261947, 2.324779,
            2.38761, 2.450442, 2.513274, 2.576106, 2.638938, 2.70177,
            2.764602, 2.827433, 2.890265, 2.953097, 3.015929
            ]

    # Create a figure with two subplots (polar plots)
        fig, axs = plt.subplots(2, 1, subplot_kw=dict(projection='polar'))

    # First subplot: SPL vs Phi
        axs[0].plot(self.Phi, self.SPLNearField, marker='o', linestyle='-', color='b', label='Calculated SPL')
```

```
        axs[0].set_title("SPL vs Polar Angle")
        axs[0].set_ylim(0, max(max(self.SPLNearField), max(spl_provided)) + 10)  # Set radial limits based on max SPL
        axs[0].set_thetamin(0)
        axs[0].set_thetamax(180)
        axs[0].set_rticks([10, 30, 50, 70, 90, 110, 130, 150])  # Example ticks
        axs[0].set_yticklabels(["10", "30", "50", "70", "90", "110", "130", "150"])
        axs[0].legend(loc='upper right')

    # Second subplot: prmsNearField vs Phi
        axs[1].plot(self.Phi, self.prmsNearField, marker='o', linestyle='-', color='g', label='Calculated prmsNearField')
        axs[1].set_title("prmsNearField vs Polar Angle")
        axs[1].set_ylim(0, max(self.prmsNearField) + 0.1 * max(self.prmsNearField))  # Set radial limits based on max prmsNearField
        axs[1].set_thetamin(0)
        axs[1].set_thetamax(180)
        #axs[1].set_rticks([0.01, 0.05, 0.1, 0.15, 0.2, 0.25])  # Adjust these values based on your data
        #axs[1].set_yticklabels(["0.01", "0.05", "0.1", "0.15", "0.2", "0.25"])
        axs[1].legend(loc='upper right')

    # Adjust layout to prevent overlap
        plt.tight_layout()

    # Show the plot
        plt.show()
#################################################################
#################################################################
class NearFieldSoundPressureSphere:
    def __init__(self, m, B, D, M, T, Q, rpm, c0, azimuth, elevation):
        self.m = m
        self.B = B
        self.D = D
        self.M = M
        self.T = T
        self.Q = Q
        self.rpm = rpm
        self.c0 = c0
        self.azimuth = azimuth
        self.elevation = elevation

        # Precompute values used across methods
        self.omega = 2 * np.pi * self.rpm / 60
        self.k = self.m * self.B * self.omega / self.c0
        self.beta = np.sqrt(1 - self.M**2)
        self.Re = 0.8 * (self.D / 2)
        self.pref = 2e-5

        # Initialize result arrays
        self.prmsNearFieldSphere = np.zeros((len(self.azimuth), len(self.elevation)))
```

```
        self.SPLNearFieldSphere = np.zeros((len(self.azimuth), len(self.elevation)))

        # Initialize field point coordinates
        self.x = np.zeros((len(self.azimuth), len(self.elevation)))
        self.y = np.zeros((len(self.azimuth), len(self.elevation)))
        self.z = np.zeros((len(self.azimuth), len(self.elevation)))

    def calculate_prms_and_SPL(self):
        # Define the parameterized functions x_l, y_l, z_l
        x_l = lambda theta: 0
        y_l = lambda theta: self.Re * np.cos(theta)
        z_l = lambda theta: self.Re * np.sin(theta)

        for i in range(len(self.azimuth)):
            for j in range(len(self.elevation)):
                # Field points, default radius 2D
                self.x[i, j] = 2 * self.D * np.cos(self.elevation[j])
                self.y[i, j] = 2 * self.D * np.cos(self.azimuth[i]) * np.sin(self.elevation[j])
                self.z[i, j] = 2 * self.D * np.sin(self.azimuth[i]) * np.sin(self.elevation[j])

                # Define the S and sigma functions
                S = lambda theta: np.sqrt((self.x[i, j] - x_l(theta))**2 + self.beta**2 * ((self.y[i, j] - y_l(-theta))**2 + (self.z[i, j] - z_l(theta))**2))
                sigma = lambda theta: (self.M * (self.x[i, j] - x_l(theta)) + S(theta)) / self.beta**2

                # Define the integrand functions funA and funB
                funA = lambda theta: (
                    self.T * self.x[i, j] / S(theta)**2 * np.cos(self.m * self.B * theta + self.k * sigma(theta)) +
                    (self.T * self.k / self.beta**2 * (self.M + self.x[i, j] / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                    np.sin(self.m * self.B * theta + self.k * sigma(theta))
                ) / S(theta)

                funB = lambda theta: (
                    -self.T * self.x[i, j] / S(theta)**2 * np.sin(self.m * self.B * theta + self.k * sigma(theta)) +
                    (self.T * self.k / self.beta**2 * (self.M + self.x[i, j] / S(theta)) - self.Q * self.m * self.B / self.Re**2) *
                    np.cos(self.m * self.B * theta + self.k * sigma(theta))
                ) / S(theta)

                # Perform the integration
                intA, _ = quad(funA, 0, 2 * np.pi, epsrel=1e-6, epsabs=1e-6)
                intB, _ = quad(funB, 0, 2 * np.pi, epsrel=1e-6,
```

```
    epsabs=1e-6)

                # Calculate prmsNearFieldSphere and
    SPLNearFieldSphere
                self.prmsNearFieldSphere[i, j] = np.sqrt(2) / (8 *
    np.pi**2) * np.sqrt(intA**2 + intB**2)
                self.SPLNearFieldSphere[i, j] = 20 * np.log10(self.
    prmsNearFieldSphere[i, j] / self.pref)

    def plot_results(self):
        # Convert azimuth and elevation to radians if not already
        azimuth_rad = self.azimuth
        elevation_rad = self.elevation

        # Create a meshgrid for spherical coordinates
        azimuth_grid, elevation_grid = np.meshgrid(azimuth_rad,
    elevation_rad)

        # Convert spherical to Cartesian coordinates for plotting
        X = np.sin(elevation_grid) * np.cos(azimuth_grid)
        Y = np.sin(elevation_grid) * np.sin(azimuth_grid)
        Z = np.cos(elevation_grid)

        # Plot SPL on a 3D spherical surface
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

        # Use surface plot to map SPL to the spherical coordinates
        surf = ax.plot_surface(X, Y, Z, facecolors=plt.cm.viridis(
    self.SPLNearFieldSphere.T / np.max(self.SPLNearFieldSphere)),
    rstride=1, cstride=1, antialiased=False, shade=False)

        # Customize the plot
        ax.set_title("SPL Near Field - Spherical Coordinates")
        ax.set_xlabel("z")
        ax.set_ylabel("Y")
        ax.set_zlabel("x")

        # Add a color bar for SPL values
        m = plt.cm.ScalarMappable(cmap=plt.cm.viridis)
        m.set_array(self.SPLNearFieldSphere)
        plt.colorbar(m, shrink=0.5, aspect=5)

        plt.show()

    def run(self):
        self.calculate_prms_and_SPL()
        self.plot_results()
#################################################################
#################################################################
# Define D (you can set D to any value you want)
D = 1  # Example: D = 1, you can change it according to your needs

# Generate 100 observer points along the x-axis between -100D and
    100D
x_observers = np.linspace(-100 * D, 100 * D, 50)
```

```
y_observers = np.zeros(50)  # All y coordinates are 0 since points
    are on x-axis
z_observers = np.zeros(50)  # All z coordinates are 0 since points
    are on x-axis

# Propeller location
propeller_position = np.array([0, 0, 100 * D])

# Plot the observer points and the propeller location
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot observer points
ax.scatter(x_observers, y_observers, z_observers, color='blue',
    label='Observers')

# Plot propeller location
ax.scatter(*propeller_position, color='red', label='Propeller')

# Annotate the propeller location
ax.text(*propeller_position, 'Propeller', color='red')

# Set labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
ax.set_title('Observer Points and Propeller Location')

# Set limits for better visualization
ax.set_xlim([-100 * D, 100 * D])
ax.set_ylim([-100 * D, 100 * D])
ax.set_zlim([0, 100 * D])

# Show legend
ax.legend()

# Show the plot
plt.show()
```

Listing F.1: Acoustic Study