UCLouvain

École polytechnique de Louvain

Implementation of an immersed lifting line method in an exascale-ready implementation of the vortex particle-mesh method

Author: **William Grollinger**
Supervisors: **Philippe Chatelain, Matthieu Duponcheel**
Readers: **Philippe Chatelain, Matthieu Duponcheel, Vincent Legat, Jonathan Lambrechts**
Academic year 2022–2023
Master [120] in Mathematical Engineering

Acknowledgments

First of all, I would like to express my deepest gratitude to my two thesis supervisors, Professor Philippe Chatelain and Professor Matthieu Duponcheel, for their invaluable expertise and guidance throughout my research. I am profoundly grateful for their valuable advice and availability, which have been instrumental in shaping the outcomes of this thesis.

I would also like to extend my heartfelt thanks to Mr. Jonathan Lambrechts and Professor Vincent Legat for reviewing my work.

Furthermore, I am sincerely grateful to Pierre Balty for his dedicated mentoring, patient explanations of complex concepts, and ongoing advice, which played a crucial role in keeping me on track throughout this research project.

Lastly, I would like to acknowledge the Consortium des Équipements de Calcul Intensif (CESI) and the Université Catholique de Louvain (CISM/UCL) for generously providing computational resources through their supercomputing facilities. These resources have been vital in enabling the successful execution of the simulations and analyses involved in this work.

Contents

Introduction

Throughout the years, numerical methods have progressively gained greater significance within the realm of scientific research. Indeed, numerical methods find extensive application across a wide array of practical scenarios. Researchers are increasingly attracted to these tools, whether to simulate the dynamics of moving objects, analyzing the behavior of financial markets, or examining fluid flows around diverse geometries. Primarily employed for numerical analysis, these simulations aim to accurately model phenomena to study and predict their behavior. In the field of fluid mechanics, these simulations enable the researcher to study the behavior of flows around complex geometric objects. The many advantages of numerical simulations, which are precision, efficiency, and speed, have made them indispensable tools in the world of industry.

In the industrial world, numerical simulations are used to design and optimize a large number of items. For example, in the energy industry, this kind of modeling can be used to study the behavior and performance of a dam or to study the behavior of the flow around a wind turbine as shown in Figure 1. In this field in particular, the study of flow behavior in the wake of a wind turbine could play a key role in the future of energy. Indeed, as explained by Trigaux et al. [1], for logistical and environmental reasons, more and more wind farms are emerging around the

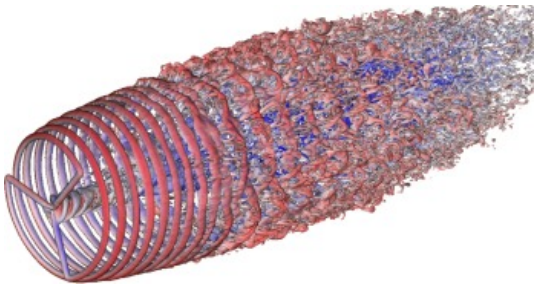


Figure 1: Numerical simulation of a fluid flow around a wind turbine. From [2].

world. Placing the different wind turbines without any reflection can lead to a power loss of 40% [1], so the study and optimization of these farms are really important. Such a study requires modeling the whole wind farm to understand the effect of numerous factors such as the positioning of wind turbines, the inclination of their blades, and their speed of rotation. Moreover, energy is not the only sector where numerical methods are used. They are also used in the aeronautical and aerospace industry to model and optimize the performance of aircraft, rockets, or ships.

The use of these numerical methods in the aforementioned domains requires a certain efficiency and precision that necessitates a significant computational capacity. Let us take the example of a wind farm. To model such a large domain while keeping sufficient accuracy to capture all the effects of the wind turbine on the fluid flow, it is necessary to use domains with a very large number of grid points. The complexity and computational cost of these methods can be extremely high. Therefore, in order to meet these needs, there are many more or less complicated techniques that allow us to obtain more or less precise results. We know, for example, the use of an adaptive time step or a non-uniform mesh that enables grid refinement in critical areas such as the blades. Another technique that can be used is the decomposition of the domain into different blocks that will be distributed among different processors, which will allow for parallel computation and therefore a considerable time-saving.

One way to model the blades of a wind turbine in a flow is to use Prandtl's theory of lifting lines [3]. This theory is a mathematical model applied to fluid mechanics, which allows us to represent a slender device as a line having a lifting force distribution along its length. Furthermore, to use this theory and implement it in a numerical simulation model, the use of a vortex particle-mesh method is necessary. This modeling method will allow us to integrate a discretization of the lifting line in the flow using particles.

Outline

Concretely, the goal of this master thesis is to develop a model of lifting lines in an already existing, exascale-ready implementation of the vortex particle-mesh method coded in a code called Murphy [4]. This code is a C++ code that combines efficiency and accuracy of results by using multiple processors. Therefore, this work consists of three chapters:

1. The first chapter is theoretical. We will go through the 3D Navier-Stokes equations and the Prandtl lifting line theory. Then we will explain the vortex

particle mesh method, its advantages and disadvantages, and how immersed lifting lines can be considered.

2. The second chapter aims to provide more details about Murphy, including the algorithm of immersed lifting line that is implemented and the adaptation to multi-processor implementation.

3. The third chapter will begin with the validation of the implemented model using another code named Vpm4x. We will then study the performance of the implementation.

To conclude the document, we will review the main theoretical results of the research and the main practical results showing the validity and performance of the algorithm. We will also propose some outlooks for readers.

Chapter 1

Methodology and Numerical Tools

This chapter aims to provide the readers with an overview of the numerical tools that will be used in this thesis. We begin by recalling the Navier-Stokes equations and their two formulations. Then, we will explain Prandtl's lifting line theory. Following that, we will explain the vortex particle-mesh method. Finally, we will look at a model, used to account for lifting lines on a flow using a Lagrangian discretization of the vortex sheet, which was developed by Caprace [5].

Contents

1.1 The 3D Navier-Stokes equations and the Prandtl lifting line theory

The Navier-Stokes (NS) equations in 3D

The Navier-Stokes equations were derived to model the flow of a Newtonian fluid. They describe the conservation of energy, mass, and momentum. Making the assumption of an uncompressible flow, the vector formulation in *velocity-pressure* of the NS equations is as follows

$$\nabla \cdot \mathbf{u} = 0 \tag{1.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla \mathbf{P} + \nu \nabla^2 \mathbf{u} \tag{1.2}$$

where ν is the kinematic viscosity, $\mathbf{u}$ is the velocity field and $\mathbf{P}$ is the reduced pressure. We can then define the 3-dimensional vorticity vector, $\boldsymbol{\omega}$ as follows

$$\boldsymbol{\omega} = \nabla \times \mathbf{u}. \tag{1.3}$$

By taking the curl of the equation (1.2), we can obtain the *vorticity-velocity* formulation of the Navier-Stokes equations which is given by the following partial vector differential equation

$$\frac{D\boldsymbol{\omega}}{Dt} = (\boldsymbol{\omega} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \boldsymbol{\omega}. \tag{1.4}$$

In this equation, the material derivative is expressed as

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla \tag{1.5}$$

The velocity field is $\mathbf{u} = \mathbf{u}_\omega + U_\infty$ where $\mathbf{u}_\omega$ can be obtained from the vorticity by solving the following Poisson equation

$$\nabla^2 u_\omega = -\nabla \times \boldsymbol{\omega}. \tag{1.6}$$

An important property of the vorticity that follows directly from equation (1.3) is that

$$\nabla \cdot \boldsymbol{\omega} = \nabla \cdot (\nabla \times \mathbf{u}) = 0 \tag{1.7}$$

This is called the *solenoidal* property. Theoretically, this important property must be verified at every time. However, in numerical simulation, inaccuracies can be observed which may lead to disrespecting this property.

The Prandtl lifting line theory

Lifting line theory was developed by Prandtl in 1918 [3]. It is a powerful tool that can be used to model a turbine blade, an aircraft wing, or any other such linear device. This tool aims to predict the lift distribution along the lifting device without having to solve the Navier-Stokes equations around the entire airfoil section.

As explained by Winckelmans [6], this theory considers the wing as a distribution of infinitesimally thin vortices along its span. These vortices represent the effect of airflow circulation around the wing, providing an approximation of the real flow behavior. As shown in figure 1.1, the device is represented by a line and its vortex wake. This line will be a source of circulation and will then generate forces and vorticity. In order to add the impact of these forces on the flow, we shed a vortex sheet in the flow containing vorticity information.

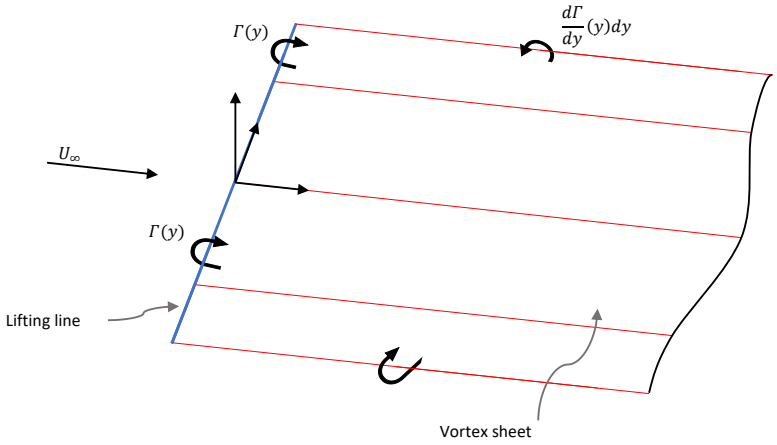


Figure 1.1: Lifting line and vortex sheet scheme.
Inspired from [6]

In figure 1.1 we observe $\Gamma(y)$ which is the circulation around the aerodynamic profile located at y. This circulation is related to the vorticity by the relation

$$\Gamma = \int_S \omega \cdot n dS \tag{1.8}$$

Due to the continuity of the vortex tubes, a vortex sheet must be shed and have a circulation per unit length, $\gamma(y) = -\frac{d\Gamma}{dy}(y)$. This Prandtl lifting line theory enables us to evaluate the lift forces exerted on the wing

$$L = \int_{-\frac{b}{2}}^{\frac{b}{2}} l(y)dy \tag{1.9}$$

Using the Kutta-Joukowski theorem [7] which gives the sectional lift density, $l(y) = \rho U_\infty \Gamma(y)$, we obtain

$$L = \rho U_\infty \int_{-\frac{b}{2}}^{\frac{b}{2}} \Gamma(y) dy \tag{1.10}$$

1.2 The Vortex Particle Mesh (VPM) method and the Immersed Lifting Line (ILL) model

The vortex particle-mesh method

Based on the *velocity-vorticity* formulation of the Navier-Stokes equations derived in section 1.1, the vortex-particle mesh (VPM) method is a method used to implement a numerical simulation of a fluid flow. As explained by Balty et al. [8], VPM is a hybrid of two different methods.

- The vortex particle (VP) method: This method consists of a Lagrangian representation of the domain composed of particles that will move in the domain. As explained by Balty et al. [8], this method is especially useful to process the advection term in the NS equations. Indeed, the VP method permits a robust adaptability, a larger time step, and therefore a more favorable change of CFL condition. However, such a method leads to a lot of inaccuracies due to particle distortions in the domain. Moreover, computing the velocity field from the vorticity field will implies to solve a more difficult problem, which will require more computational resources than using a mesh.
- The mesh method: Unlike the previous method, this one is mainly useful to compute finite differences efficiently and accurately. This method will then be used to recover the velocity from the Poisson equation and to compute the diffusion and stretching terms.

The hybrid aspect of the VPM method, therefore, brings the different advantages of each of the methods. First, we use particles to compute the advection, which increases the efficiency. Secondly, we use the mesh to perform finite differences to compute the Poisson equation and the right-hand side of the NS equations. This method will be much more accurate.

There are two particularities to using the VPM method. Firstly, as explained above, particles are moving and tend to deplete certain regions of the flow while

regrouping in others. This phenomenon is bad for the accuracy of the result. To avoid it, we need to remesh the particles frequently. Secondly, as we saw in section 1.1, vorticity has a *solenoidal* property. However, this property will not be preserved by the VPM method. Then, in order to maintain it true, Caprace [5] suggests periodically reprojecting the vorticity of the mesh onto a solenoidal space. This reprojection consists in computing a correction ∇A and adding it to the vorticity. This correction must be computed as $\nabla^2 A = -\nabla \cdot \omega$ so that the final vorticity field $\omega + \nabla A$ is divergence-free.

Large Eddy Simulations (LES)

In applications such as simulations of aircraft wings or wind turbine blades, the flow is typically turbulent and with a very high Reynolds number. As shown by Thiry [9], using full DNS methods to solve the flow leads to a total simulation cost scaled as Re^3. Such a turbulent simulation will then lead to a really high computational cost. To reduce the total cost, we instead use Large Eddy Simulations (LES) which, as explained by Balty et al. [8], modify the Navier-Stokes equations by low pass filtering. This modification saves computational resources but also removes the small-scale effects. In order to maintain a certain accuracy of results, we then need to take these small scales into account by adding a term to the Navier-Stokes equations. Assuming that small scales are isentropic, we can model them using a sub-grid scale model. Using it, the equation (1.4) becomes

$$\frac{D\boldsymbol{\omega}}{Dt} = (\boldsymbol{\omega} \cdot \nabla)\mathbf{u} + \nu\nabla^2\boldsymbol{\omega} + \nabla \cdot T^M \tag{1.11}$$

with

$$T^M = \nu_{SGS}(\nabla\boldsymbol{\omega}^S + (\nabla\boldsymbol{\omega}^S)^T) \tag{1.12}$$

In this equation, ν_{SGS} is the eddy viscosity obtained using the velocity field and $\boldsymbol{\omega}^S$ is the resolved small-scale part of the vorticity field that is obtained by high-pass filtering. For more details on the computation of the various terms, the readers are referred to [10].

The Immersed lifting line model

The VPM method explained in the previous section can account for a vortex generator in the flow. In our case, the vortex generator is a slender device representing a lifting body in the flow. There are several ways of modeling a linear

device in a flow, for example using the actuator line method (ALM). However, in our case, we use an immersed lifting line model described by Caprace in [5] which is inspired by Prandtl's lifting line theory.

Lift and circulation evaluation

Assuming a high aspect ratio device ($AR = \frac{b^2}{S}$), we can assume that the flow around the airfoil is essentially 2D. Taking an element of the line, we defined the sectional lift as L_{2D_i} and the sectional circulation as Γ_i. According to theory, we have

$$L_{2D_i} = \frac{1}{2}\rho U_{e_i}^2 c_i C_l(\alpha, Re) \tag{1.13}$$

where, U_e is the local effective upstream velocity, c is the aerodynamic chord and ρ is the fluid density. Finally, C_l is the lift coefficient, which depends on the angle of attack, α, and the Reynolds number, Re.

In order to relate the lift force with the circulation, Phillips et al. [11] suggest using the Kutta-Joukowski theorem, which in this situation can be written

$$L_{2D_i} = \rho U_0(t) \wedge \Gamma(t) \tag{1.14}$$

where U_0 is the projection on the sectional plan of the local velocity measured on the line element (Note that $U_e = U_0$).

Finally, we can combine these different equations in order to obtain the expression of the circulation

$$\Gamma = \frac{1}{2} U_0 c C_l. \tag{1.15}$$

Discretization of the vortex sheet and lagrangian treatment

The main aspects of the lagrangian treatment are shown in figure 1.3. First, we have the bound vorticity due to circulation on the line represented in blue. In this document, we will also refer to this vorticity as the "attached" vorticity. Assuming a representation of the line as in figure 1.2 we see that the "attached" vorticity only needs to be computed on the mid points. This vorticity has a strength equal to $\Gamma_i * ds$.

Secondly, we have the "shed" vorticity, which has two components. The first one, in red, is perpendicular to the line and is due to the spanwise gradient of circulation. This component has a strength of $\Gamma_{i+1} - \Gamma_i$ which depends on its release time. The second one, in orange, is parallel to the line and completes the

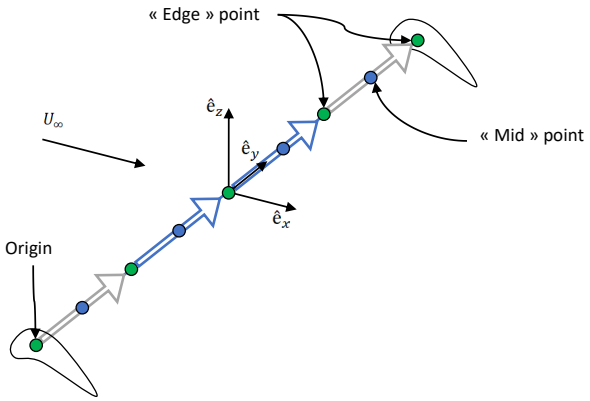


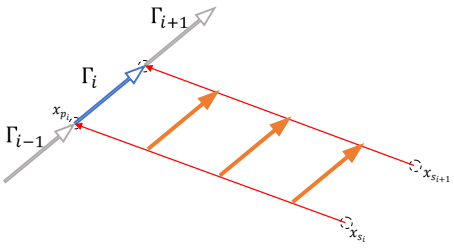
Figure 1.2: Line discretization, mid and edge points.

sheet of vorticity to ensure the solenoidal property. This part of the vorticity has a circulation per unit length equal to $\frac{\Gamma_i(t+\Delta t)-\Gamma_i(t)}{|x_s-x_p|}$.

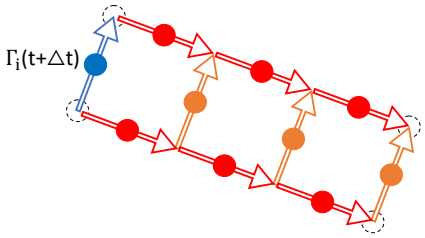
In order to add the impact of the sheet on the flow while taking into account the non-zero thickness of the lifting body, we will use a mollification region. This 1D mollification is introduced in the thickness direction by using several layers as we can observe in figure 1.3c. Each layer account for a fraction of the computed vorticity. To compute the different thickness weights for the different layers, we use a Gaussian function characterized by a parameter, σ. The weights, ω_j, for layer j, are computed in the following table as a function of $\frac{\sigma}{h}$.

j	-2	-1	0	1	2
$\frac{\sigma}{h} = 1$	0.0545	0.2442	0.4026	0.2442	0.0545

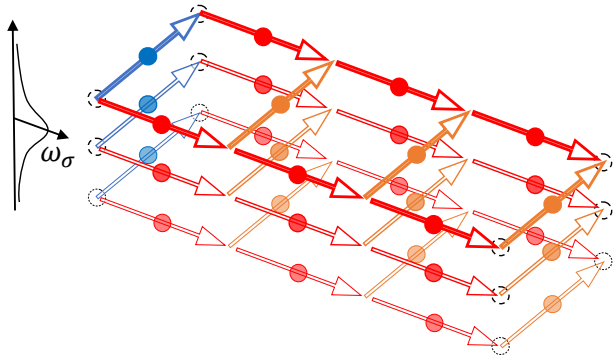
Table 1.1: Thickness weight as a function of the layer and $\frac{\sigma}{h}$



(a) Bound and shed vorticity component and tracer points.



(b) Discretization of the bound and shed vorticity using one layer.



(c) Discretization of the bound and shed vorticity using a mollification region.

Figure 1.3: Discretization of a lifting body and its vortex sheet. Inspired from [5]

Chapter 2

An Object-Oriented Implementation In Murphy

In this chapter, we will explore the main structures of the code. Then, we will explain the data structures that have been developed to enable the implementation of ILL in the code. Afterward, we will study in detail the steps of the algorithm used to implement the immersed lifting lines. Finally, we will examine how the code has been adapted for parallel computing with processor communication.

Contents

2.1 Introduction

Murphy (MUltiResolution multiPHYsics) is a flow solver developed by UCL-TFL and Van Rees Lab (MIT). It is "a multiresolution adaptive grid framework for numerical simulations on 3D block structured collocated grids with distributed computational architectures" [12]. This code is available on a GitHub repository [4]. In this code, a lot of researchers are developing different functionalities for various aims.

The study of the lifting lines and the elaboration of a functional numerical model was already realized a few years ago within the framework of a PhD at the Université Catholique de Louvain, [5]. This research led to the development of immersed lifting lines in a code named Vpm4x coded in the Fortran programming language. Initially developed by Chatelain [13], the code and its immersed lifting lines were then enhanced by Caprace [14]. The goal of the current master's thesis is to provide a functional implementation of the same immersed lifting lines but in a C++ code, Murphy. To achieve this, it was necessary to translate the existing Fortran implementation of the lifting lines into the already existing C++ code.

On the one hand, Fortran is a high-level programming language developed for scientific calculations. On the other hand, C++ is a high-level, object-oriented language that was developed based on C. In terms of syntax, the two languages have significant differences. Fortran uses explicit declarations for variables whereas C++ uses a strong typing system. C++ also uses classes and objects for object-oriented programming, whereas Fortran does not. The two models also differ in their variety of libraries.

As we can see, a translation from Fortran to C++ is not that simple. The original code must be adapted to be functional. This chapter will provide the reader with a clear view of the ILL implementation in Murphy.

2.2 Forest structured data

Murphy was developed to solve 3D fluid flows. This kind of resolution is very time and computational resources consuming. Therefore, we need to use code parallelism to reduce the time load. For this purpose, the computational domain is divided into several sub-domains called blocks. Each block can contain information and specific data associated with the part of the domain to which it corresponds. Afterward, these blocks are equally distributed among the available

processors, which will perform all computations at the same time. Performing the numerical solve of the Navier-Stokes equation on such a decomposition requires communication between the different blocks. Since two blocks can be on the same or different processors, there are two types of communication. The first one is communication between two blocks that are on the same processor. The second type is communication between two blocks that are not on the same processor. This second type of communication requires the use of MPI, a standardized library specification used for parallel computing, which provides a set of functions and routines enabling communication and coordination between multiple processors.

Representing the block decomposition of the domain requires appropriate data structures. There exist several types of appropriate data structures. The Fortran code, Vpm4x, uses a Cartesian decomposition of the domain into blocks which are stored linearly in memory. Such an implementation offers particularly good performance when communicating information from neighboring blocks but has major drawbacks such as poor load balancing when using non-uniform grid resolutions. Another interesting data structure, commonly used to represent the blocks decomposition of the domain, is the forest grid decomposition which is the choice of implementation made by Murphy's developers. Forest grid is a two-dimensional data structure that represents a three-dimensional domain. In order to implement such a structure, the developers used the p4est library. The readers are referred to [15] for more information on how forest grid decomposition works. Unlike the Vpm4x implementation, the forest grid offers a multitude of advantages. One major benefit is the significant improvement in load-balancing efficiency across multiple processors achieved through forest grid decomposition. This advantage becomes particularly important when blocks are subdivided into smaller units, leading to the creation of a non-uniform grid.

2.3 ILL data structures

As we have seen before, Murphy is an object-oriented code. Such code requires the use of data structures that will model the various parts of the problem. In what follows, readers will find details on the implementation of the data structures that have been added to Murphy in order to implement immersed lifting lines efficiently. Since the objective of the code is to be adaptable to new features in the future, we strive to be efficient and always implement things in such a way that the code can be easily adapted.

Curve/Line Structure

Since immersed lifting line theory consists in representing a blade by a line in the flow, we need a data structure that represents and describes a *"line"* object. To guarantee the flexibility needed to adapt the code to lifting curves and non-linear devices, we have chosen to create a global structure called *"curve"*, which will contain the required function declarations. With this structure as the parent class, we developed a derived class called *"line"*. This class inherits the attributes of *"curve"* and implements the different functions needed for its functionality. Below, is a list of the *line* class attributes:

- The **origin** position of the line corresponding to one of the two edges.
- The **length** of the line.
- The **direction** and **normal** vectors of the line. The line's reference frame is defined by these two vectors, as well as by the vector that is orthogonal to both of them.
- The angle of attack, $\boldsymbol{\alpha}$, which is defined with respect to the line's reference frame.
- The **number of segments** used to discretize the line.
- The **number of layers** used to mollify the vortex sheet discretization and the **space between them**.
- A *"vector"* of *"SeedShedPoints"* (*"SeedShedPoints"* class is explained later in this section) that represents **the discretization points** of the line. The choice of using *"vector"* brings us major advantages over simple lists. Indeed, vectors are useful when we want to save objects in contiguous memory. As we will often be iterating over different discretization points, it is useful to have them close together in memory. Moreover, using vectors makes it easier to add or remove elements.
- A **parametric expression** that provides the parametric equation of the line.

A *"line"* object also has several functions. The first one computes the discretization points of the line. The second one computes the number of discretization segments to be used (such as the length of a segment, $ds \approx h$), the number of layers, and the space between them. Lastly, additional functions are implemented to read files describing the line geometry.

Seed Shed Point Structure

The particles that will discretize the line as well as the tracers that will be dropped in the flow carry almost identical information. Knowing this, we chose to develop a unique data structure called *"SeedShedPoint"* to model these control points. This data structure is mainly characterized by the following attributes:

- The **position** of the point.
- Two **indexes**. The first one corresponds to the position along the line and the second one to the layer location of the particle. Using such indexing, we can identify the location of particles in the discretization and the mollification. Then, we can find tracers corresponding to the discretization points attached to the line.
- The **circulation** attached to the line. For tracers, this circulation corresponds to the circulation at the start of the time step, while for points attached to the line, this circulation is the current circulation.

Immersed Lifting Line Block Structure

As explained in the previous section, the physical domain is decomposed into several blocks over which each computation will be performed in parallel. Using such a decomposition requires the use of a block structure. In our case, we will call it the immersed lifting line block structure (*"IllBlock"*).

This class inherits from a particle block structure (*"PartBlock"*) already existing in Murphy. This *"PartBlock"* structure helps implement the VPM method for solving Navier-Stokes equations. Overall, the purpose of this *"IllBlock"* data structure is to implement all operations to be performed locally on the parts of the domain corresponding to the block.

Immersed Lifting Line Manager Structure

Since we want to develop parallel computing code using multiple processors, we need to be able to manage the blocks on each processor. To make it easy, we have chosen to develop a *"IllManager"* data structure. This manager will enable us to control the different operations to be applied to the different processor blocks. We will see in section 2.5 how this manager has also been used to perform the different communication operations between processors.

2.4 Algorithmic procedure

In the implementation of Murphy's VPM method, the Navier-Stokes equations are integrated using a low-storage third-order Runge-Kutta scheme. Let us assume that the differential equation we want to solve can be written as follows

$$\frac{d\omega}{dt} = f(\omega) \tag{2.1}$$

then, the low-storage Runge-Kutta scheme is

$$q_j = a_j q_{j-1} + h f(\omega_{j-1}) \tag{2.2}$$
$$\omega_j = \omega_{j-1} + b_j q_j \tag{2.3}$$

The values of the terms a_j and b_j for different Runge-Kutta orders can be easily computed. In our case, we use a third-order scheme for which we have the coefficients of table 2.1. For more details on how to obtain these coefficients and on the stability of this method, we invite readers to consult the document written by Williamson [16].

j	1	2	3
a_j	0	-0.5556	-1.1953
b_j	0.3333	0.9375	0.5333

Table 2.1: Coefficients of the low-storage third-order Runge-Kutta scheme.

In order to account for the lifting body in the flow when integrating the Navier-Stokes equations, it is necessary to model and add to the vorticity of the flow, the vorticity generated by the lifting body and its wake. Figure 2.1 presents a synthetic view of the algorithm. The following subsections aim to go through in detail the different operations that need to be carried out over a time step in order to model lifting bodies efficiently.

First sub-time step

The first sub-step is a little different from the next two. Indeed, as explained in section 1.2, the vorticity produced by the lifting body can be divided into two distinct parts. The attached vorticity (or bound vorticity) and the two components vortex sheet discretization. At the current time of this first sub-time step, the only

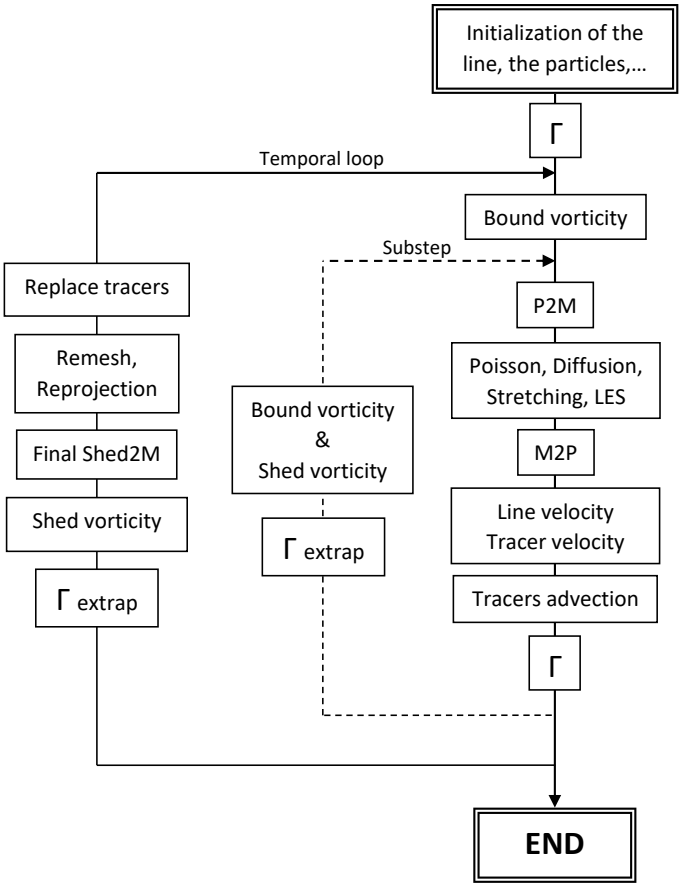


Figure 2.1: Main algorithm steps when including ILL.

part of the vorticity that has already been created is the bound vorticity. Therefore, the first thing to do is to compute this attached vorticity. How to compute it is described in subsection 2.4.1.

Once the bound vorticity has been computed, we need to interpolate this vorticity and the particle vorticity on the mesh. Following this, we can compute the velocity on the domain, by solving Poisson's equation (1.4). This equation is solved in the spectral domain using FLUPS [17, 18] (Fourier-based Library of Unbounded Poisson Solvers).

It is important to note that after solving the equation, it is necessary to subtract the attached vorticity from the mesh vorticity. Failure to remove it after each sub-time step would result in multiple instances of the same vorticity being added to the mesh, leading to errors.

Finally, with the velocity known on the mesh (and the particles by interpolation), it remains to compute the stretching term, the diffusion term, and the LES term, as well as advecting all the particles and updating the values of the RK3 terms. The final operation consists in updating the value of the line circulation and advecting the tracers of a sub-time step.

Second and third sub-time step

In the second and third sub-steps, after the tracers have been advected, it becomes necessary to compute the discretization of the vortex sheet in addition to the bound vorticity. To achieve this discretization, we need to compute $\Gamma(t_i)$, where t_i represents the current time, which is obtained by adding $dt_{i-1,i}$ (the sub delta time step between t_{i-1} and t_i) to the previous time t_{i-1}.

To compute $\Gamma(t_i)$, we need to know the velocity at t_i. However, to obtain this velocity, we need to solve Poisson's equation using the vorticity of the discretized vortex sheet added onto the mesh. This brings us back to the original problem, which requires the value of $\Gamma(t_i)$. To deal with this circular problem, we will approximate the circulation (further computational details provided in subsection 2.4.2). Finally, to maintain accuracy, we will update the circulation at the end of the sub-time step.

Once we have approximated the circulation, we can compute the attached vorticity as well as the shed vorticity parallel and perpendicular to the line, following the explanation given in section 2.4.1.

Then, in the same way as in the first sub-step, we interpolate the different vorticities on the mesh in order to solve the Poisson equation and obtain the velocity on the mesh. For the same reason as during the previous time step, solving Poisson's equation must necessarily be followed by a subtraction of the attached and shed vorticity from the mesh one.

Now that the velocity is known on the particles and the mesh, it remains to compute the stretching term, the diffusion term, and the LES term, as well as advecting all the particles, updating the values of the RK3 term, and updating the values of the line circulation. Finally, we conclude these sub-steps by marking the advection of the line tracers of one sub-time step.

End of a time step

To complete the full time step, it is now necessary to add the shed vorticity, obtained by discretizing the vortex sheet at this final time, to the mesh once and for all.

Moreover, at the end of each time step, if necessary, we perform a solenoidal reprojection and/or a remeshing. As explained in section 1.2, reprojection and remeshing are indispensable operations when using the VPM method. However, if we perform these operations at every time step, we lose the advantage of using particles. So, we need to find the right balance between too much reprojection and remeshing and not enough. Following Caprace et al. [14], we choose to use a remeshing frequency of 5 time steps and a solenoidal reprojection frequency of 20 time steps.

More precisely, as explained in section 1.2, solenoidal reprojection aims to maintain the vorticity field divergence-free. However, the lifting line present in the field is a source of vorticity. It is, therefore, necessary to disregard this vorticity production, and therefore vorticity divergence. To achieve this, we will reproject using a mask. Concretely, if we go back to the reprojection steps, we start by computing $\nabla\omega$ with the mask applied to it. Then we solve the Poisson equation, $\nabla^2 A = -\nabla\omega$, using finite differences (FLUPS solver). When solving the Poisson equation, if the domain is defined with unbounded boundary conditions, we use "even" boundary conditions, i.e. no divergence fluxes on the boundary, $\frac{\partial\omega}{\partial n} = 0$. Finally, once we have the correction ∇A we can apply it to the whole domain.

2.4.1 Vortex sheet discretization

Bound vorticity

Computing the bound vorticity requires computing the circulation on the "mid" particles of the line at the current time (details in section 2.4.2). Knowing it, we can compute ω_i using the following formula (associated to Figure 2.2)

$$\omega_i = \Gamma_i(t) \times dir \times \frac{ds}{h^3}. \tag{2.4}$$

In this equation, Γ_i is the circulation at time t+0, dir is the tangential direction to the line at the point location, ds is the length of a line element and h is the grid spacing (assuming a uniform grid). ω_i will then be distributed in the mollification region using the weights computed in section 1.2 (Table 1.1).

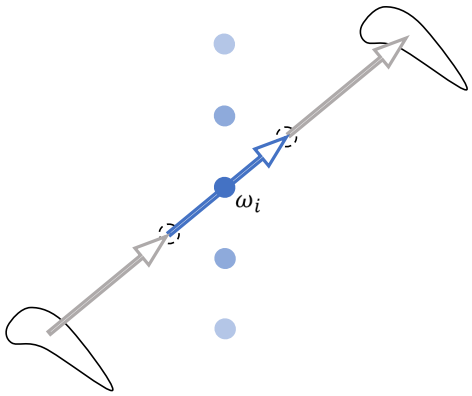


Figure 2.2: Attached vorticity scheme

Shed vorticity, parallel component

The first component of the vortex sheet discretization is the parallel component. As for the bound vorticity, we need to compute the circulation on the mid particles (shown in blue in Figure 2.3). Once this is done, we can compute the vorticity on each of the orange particles and their layer using the following equation

$$\omega_{i,j} = \frac{\Gamma(t + \Delta t) - \Gamma(t)}{Nh^3} \times axis \times l \times w_j. \tag{2.5}$$

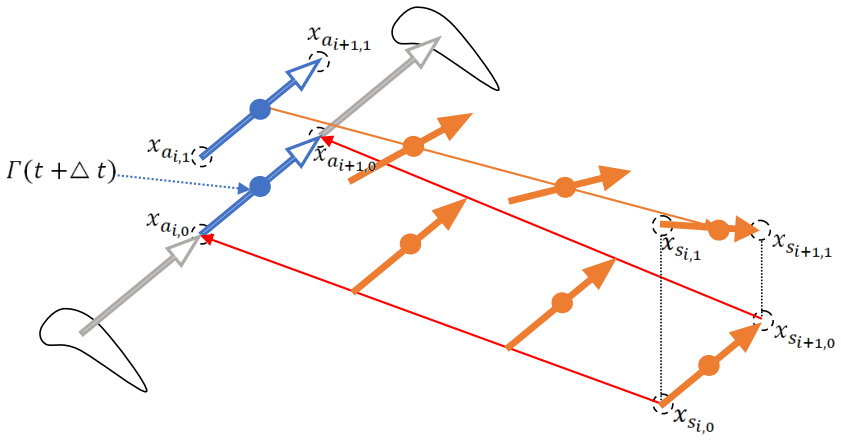


Figure 2.3: Parallel shed vorticity scheme

In the previous equation, h is the grid spacing and N is the number of particles

that will discretize the vortex sheet in the shedding direction. This number is computed so that the spacing between particles is approximately equal to h. Moreover, $axis$ is a direction. It is computed using a distance-weighted average, i.e., a linear interpolation. This interpolation is made between the vector $\overrightarrow{x_{a_{i,lay}} x_{a_{i+1,lay}}}$, which is the direction locally tangent to the line (spanwise direction), and $\overrightarrow{x_{s_{i,lay}} x_{s_{i+1,lay}}}$, which is also the spanwise direction, but the one associated with the tracers that have been advected. The latter begin their run at the locations of the "edge" particles at the start of each time step. The second direction is therefore defined as the direction of the vector between these two tracers, which depends on thickness. Finally, l is the length of the vector, computed in an equivalent way to $axis$, and w_{lay} are the weights computed in section 1.2.

Shed vorticity, perpendicular component

The second component of the vortex sheet discretization is the perpendicular component. Once again, we must first compute the circulation on the mid particles (shown in blue in figure 2.4). Knowing these, we can compute the perpendicular to the line component of the shed vorticity using the following equation

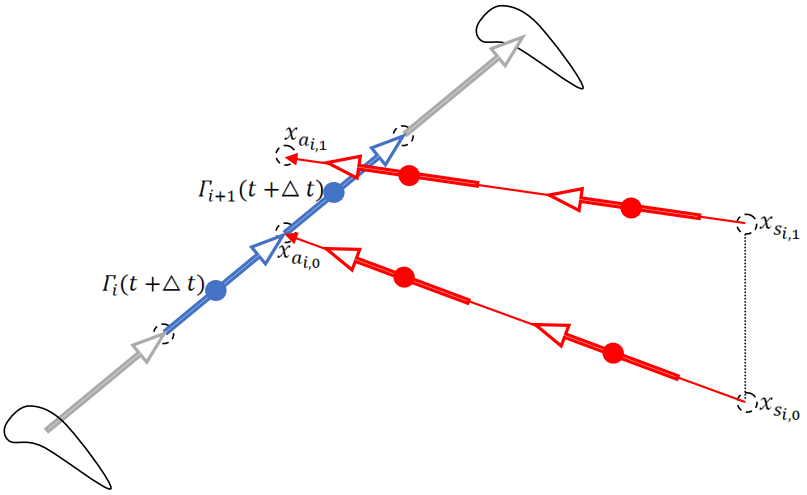


Figure 2.4: Perpendicular shed vorticity scheme

$$\omega_{i,j} = -\frac{\Big(\Gamma_i(t+\Delta t) - \Gamma_{i+1}(t+\Delta t)\Big) - \Big(\Gamma_i(t) - \Gamma_{i+1}(t)\Big)}{h^3} \times axis \times l \times w_j. \quad (2.6)$$

In this equation, h is the grid spacing and w_{lay} are the weights computed in the section 1.2. Moreover, $axis$ is the shedding direction $\overrightarrow{x_{a_{i,lay}} x_{s_{i,lay}}}$, which depends on the thickness. Finally, l is obtained by dividing the shedding axis length by the number of points in the shedding direction, N (which is once again computed to give a spacing of approximately h).

2.4.2 Circulation

As explained in the previous subsection, circulation is required to compute the discretization of the vortex sheet. We will compute the circulation of each line segment that discretizes the line using equation (1.15).

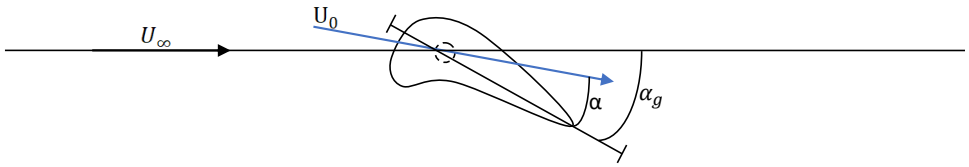


Figure 2.5: Schematic 2D view of the lifting line model and definition of the local effective angle of attack α

First, we need to obtain the upstream velocity U_0. This velocity could simply be computed by interpolating the mesh velocity over the particles, which represent a line element and are placed at its center ("mid" particles on Figure 1.2). However, as Churchfield et al. [19] explains, this method suffers from relatively large interpolation errors. Churchfield et al. [19] suggest moving the speed interpolation point to the front of the line. However, using such a method introduces a new parameter, the velocity interpolation location, which is not the intended purpose. Finally, as explained by Caprace et al. [20], the most consistent assessment of upstream velocity is to evaluate its "principal value", which is obtained by integrating the velocity over the mollification region. This velocity integral is computed as follows

$$U_0 = \sum_j w_j v(x_{a_{i,j}}) \tag{2.7}$$

where v is the velocity projected onto the plan perpendicular to the line and w_j are the layer weights.

From this velocity, we can compute the corresponding angle of attack as shown in Figure 2.5 and Reynolds number, $Re = \frac{U_0 c}{\nu}$. Finally, we can retrieve the value of the lift coefficient from a look-up to the polar data and compute the circulation.

Circulation extrapolation

At some point in the algorithm, we need to extrapolate the circulation. Assuming we have saved the values of gamma at the three previous sub-time steps, we compute Γ_{extrap} as follows

$$\Gamma_{extrap}(t_i) = \left(\frac{\Gamma(t_{i-1})}{t_i - t_{i-1}} + \frac{\Gamma(t_{i-2})}{t_i - t_{i-2}} + \frac{\Gamma(t_{i-3})}{t_i - t_{i-3}} \right) t_{tot} \tag{2.8}$$

with, $t_{tot} = \frac{1}{t_i - t_{i-1}} + \frac{1}{t_i - t_{i-2}} + \frac{1}{t_i - t_{i-3}}$.

2.4.3 Technical details

During the development of the code, we came across a few subtleties that we decided to implement so that we could enable or disable them. These include a filter on the angle of attack and the way the line normal is computed to place particles in the mollification region.

Smoothing with a filtering

The idea behind angle-of-attack filtering is to increase the angle of attack smoothly at the start, so as not to impose a too big change in the flow. Moreover, this filter is also applied in the case of an abrupt change in the angle of attack. In both cases, since we are dealing with a numerical simulation, we do not want any sharp variations that could lead to errors and that require finer resolution to be functional. So, we apply smoothing to the angle of attack. As explained, this smoothing can be activated or deactivated. The smoothing is computed as follows

$$\alpha_i = p\alpha_{true} + (1 - p)(\alpha_{i-1} + \frac{dt_{incr}}{dt} d\alpha_{i-1}) \tag{2.9}$$

where α_{true}, is the actual real value of alpha, α_{i-1} and $d\alpha_{i-1}$ are respectively the angle of attack at the start of the previous time step and the difference in angle of attack between the start of the previous time step and the one before. Finally, dt is the delta time step and dt_{incr} is the time step between the current evaluation of alpha and the time corresponding to the evaluation of α_{i-1}. Concerning the computation of p, we obtain it with, $p = min(1, dt_{incr} U_0 c)$.

Mollification direction

To obtain the position of the mollified points, we need to compute the direction normal to the line. This normal can be computed in two ways. The first way is to compute the normal to the shedding direction and the second way is to compute the normal to the local velocity. Here, following the vpm4x implementation, we choose to take the velocity normal as the direction for the mollified region. However, if we want to implement a normal to the shedding axis, we need to pay attention to some details. For example, when we are at the first sub-time steps, we do not have any shedding direction. An idea for computing the normal and remaining consistent is to take the shedding direction of the last time step.

2.5 Multi processor implementation

The Murphy implementation of ILL is adapted for use in a massively parallel architecture using multiple processors. To achieve this adaptation, we have chosen to use the method described below.

The idea is to select one processor among all and name it "Leader". In Murphy, this processor is chosen as the one containing the line origin. This "Leader" will hold all the line's updated information. It will therefore compute all operations relating to the line. Then, among the remaining processes, we can distinguish three types. Those containing part of the line ("Line Follower"), those within the line's zone of influence ("Simple Follower") (see explanation below), and others affected neither by the line nor by the vortex sheet ("Other"). The "Leader" and followers will form a communicator group where they will communicate information about the line together. Figure 2.6 is a representation of the overall functioning of the separation.

Let us look at the various operations to be performed on the lines. Firstly, all processors must determine their type ("Leader", "Line Follower", "Simple Follower", or "Other") and communicate this information to each other, so that the "Leader" knows who to communicate with. Secondly, the leader needs to know the velocity of the particles attached to the line. To achieve this, it computes the positions of the mollified region around the line and then sends these positions' information to all the processors concerned ("Line Follower" and "Simple Follower"). The latter then retrieves the position information, interpolates the velocity on these points, and sends the velocity information back to the "Leader". Finally, the "Leader" retrieves the velocities and computes the local velocities, U_0, of the attached points. Thirdly, the "Leader" will compute the circulation. Fourthly, the "Leader" needs to

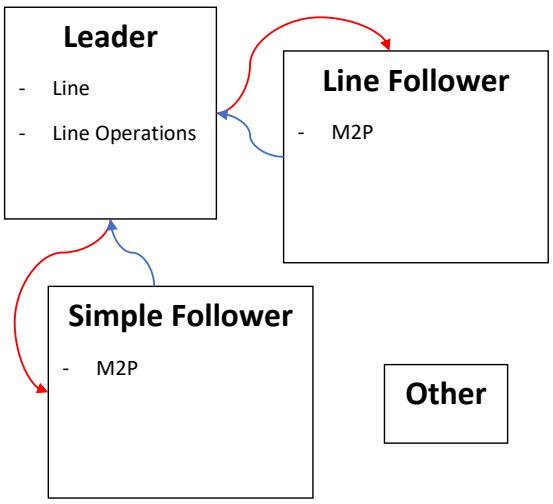


Figure 2.6: Processor decomposition and attribution

compute the velocity of the tracers in order to carry out their advection. To do this, the "Leader" sends the current position of the tracers to the followers, who in turn interpolate the velocity on these positions and send the velocity information back to the "Leader". Fifthly, as the "Leader" owns the entire updated line, he can compute the discretization of the entire vortex sheet and bound vorticity. Finally, it will send the discretization's position and vorticity information to the followers, who can then add and remove it from the mesh during the Poisson computation.

An important note on communication messages is that when the "Leader" sends information, whether it is the discretization of the vortex sheet, the position of points in the mollification region, or the position of tracers, it will send all the data identically to all the processors concerned. Only then, each processor will retain the information that is relevant to it.

For inter-process communication, we use the MPI library. In most of the above-mentioned steps, two kinds of operations are required. The first, called "BroadCast", is used to send information from one processor to a group of processors (in our case, from "Leader" to "Followers"). The second, called "Gather", is used to retrieve information sent by a group of processors to a single processor (in our case, from "Followers" to "Leader"). Both operations are implemented in the MPI library. So far, we have used blocking communications, i.e., communications that wait to be finalized before continuing.

Once all the processors have communicated information to each other, they still

have to communicate it to all their local blocks. In fact, as explained previously, we have two types of communication: between procs and between blocks. To do this blocks communication inside a processor, we use the *manager* class defined in the section 2.3. Each processor has its own object "manager". All "managers" will communicate together to perform the aforementioned inter-process communication. Once the manager has retrieved information on the processor, it distributes it to the various blocks that make it up, as shown in Figure 2.7. In this way, each block can use all the information contained in its domain.

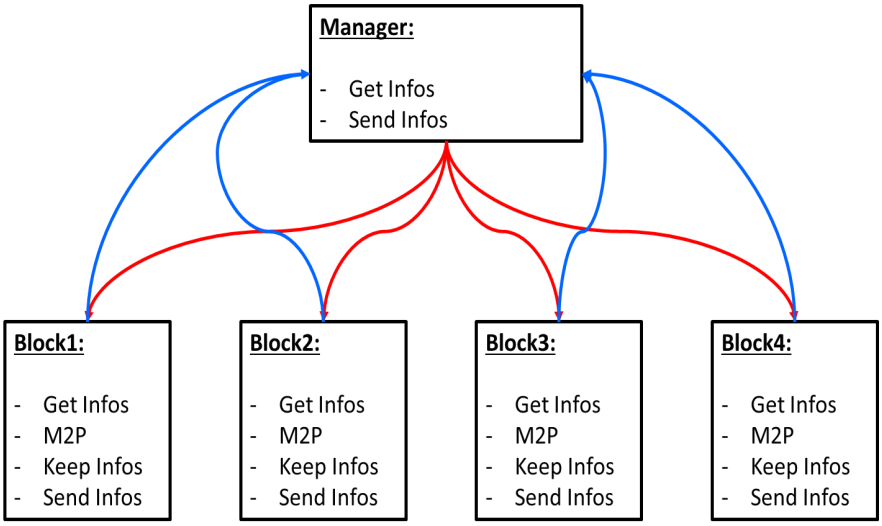


Figure 2.7: Inside processor organisation

Processor type assignation

How to determine the processor type? To find out if it is a "Leader", simply check whether it contains the origin. To find out if it is a "Line Follower", simply check whether it contains one of the points that discretize the line. Finally, if the processor does not fit into any of the above categories, we need to determine whether it is a processor in the near-line region. To do this, we compute the shortest distance between the line points and the blocks. We define this distance as the smallest distance between the line points and the block faces, as shown in Figure 2.8. Then, if this distance meets the following criteria

$$d \leq max(2 * CFL * h, 8 * \frac{\sigma}{h} * h) \tag{2.10}$$

where, $CFL = \frac{U_\infty dt}{h}$ and $\frac{\sigma}{h}$ is the Gaussian mollification parameter presented in section 1.2, the block is defined as "near line" and vice versa.

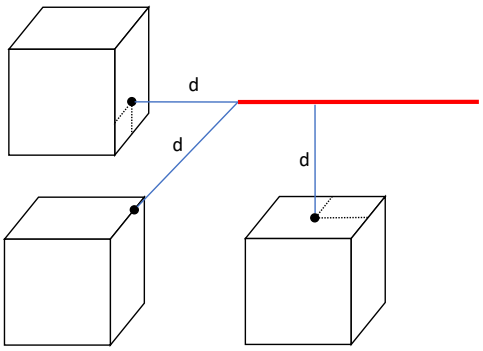


Figure 2.8: Distance d between a line and blocks

2.6 Conclusion

In conclusion, the implementation of immersed lifting lines in Murphy required an adaptation of the vpm4x one. Firstly, this chapter has enabled us to understand the overall functioning of Murphy, as well as the different data structures that have been added to implement lifting lines. Then, we reviewed the various parts of the algorithm in detail, as well as the different computational aspects present in the algorithm. Finally, we discussed the technique used to adapt the code to run on multiple processors.

Regarding future outlooks, we can identify four main points for consideration.

1. Firstly, all the code described above is based on the need for a uniform grid. However, as Murphy aims to be fast, we could imagine adapting ILL to a non-uniform grid, which would allow us, for example, to have a more refined grid near the line and a less refined grid far from the line. To achieve this, we would need to imagine a non-uniform discretization of the lift line and its wake. For example, this discretization could follow the local grid spacing. We would also need to consider modifying the spacing used between points in the mollification region to maintain consistency.

2. Secondly, an adaptation of the code to work with multiple or moving lifting lines. This adaptation is particularly useful for simulating wind farms or aircraft formation flights. No major modifications are required. However,

to minimize computation and communication costs, we could imagine distributing roles to the processors slightly differently. For example, we could determine one "Leader" per line, who would no longer necessarily be the processor containing the origin of the line, but, for example, different processors, to distribute the computational load between them.

3. Thirdly, as explained in section 2.3, we could imagine adapting the code to lifting curves. For this, we can imagine either a curve represented by linear segments locally or a higher-order curve between the curve's discretization points.

4. Fourthly, we could most probably improve the performance of the lifting lines by using a different implementation method for multi-processor operation and by optimizing communication. The possibilities of such improvements and their usefulness will be discussed in section 3.4.

Finally, we would like to point out that the ILL implementation could undoubtedly be coupled with a fully Eulerian method instead of a VPM method. We could mention here the Immersed Lifting Line approach, formulated by Scheurich et al. [21] using the vorticity transport model (VTM),[22], instead of VPM.

Chapter 3

Validation and Performance

This last Chapter aims to validate the implementation of the ILL method in Murphy. For this purpose, we will start by using the Vpm4x code to obtain results and compare them with those of Murphy. Then we will compare our results with those in the literature. Once these validations will be done, we will conclude with a performance study of the ILL implementation in Murphy.

Throughout this chapter, we consider the following dimensionless variables

$$u^* = \frac{u}{U_\infty} \qquad \omega^* = \frac{\omega L}{U_\infty} \qquad t^* = \frac{tU_\infty}{L} \tag{3.1}$$

where U_∞ is the free stream velocity and L is a characteristic length (in what follow we will take L equal to the wing span, b).

Contents

3.1 Introduction

As explained previously, this thesis aims to adapt an existing code called Vpm4x to an object-oriented implementation. To validate Murphy's ILL implementation, we will compare the results obtained by the Vpm4x code with those obtained by Murphy for identical case studies. In fact, we can assume that the results provided by Vpm4x are correct since the validity of this code has already been demonstrated by Caprace et al. in [14]. In a second step, we will re-obtain some results from [14] and check their equivalence, in order to enforce the validity of the code. Lastly, we will perform a small performance study of the implementation.

Case study

In this chapter, two test cases will be examined. In both cases, we will use a $CFL = \frac{U_\infty \Delta t}{h} = 1$ (the same as that used by Caprace et al. in [14]).

1. Firstly, we will consider a simple rectangular wing, with a constant chord, a span $b = 0.5$, and an aspect ratio $AR = 10$. As mentioned in section 1.2, we recall that $\Gamma = \frac{1}{2} U_\infty c C_L$. This wing will be placed in a domain of size $[2b \times 2b \times 8b]$ with $\frac{h}{b} = \frac{1}{32}$. We will use a uniform upstream velocity in the z direction, position the center of the wing at the domain entrance at $[b, b, b]$, and orient the wing in the y direction. In this case, we will use a Delft NACA0018 polar [23] with an arbitrary angle of attack of 6°. This arbitrary value was chosen as an acceptable angle-of-attack value for aircraft wings. However, it represents a relatively low value for wind turbine blades. The choice of this angle is not the most important, as the aim of this chapter is to demonstrate the accuracy of the code and not to study the impact of the angle of attack.

2. Secondly, we will consider the same elliptical wing as wing D presented in [14] with $b = 0.5$ and $AR = 10$. This is an elliptical wing such that, $\Gamma = \Gamma_0 * \sqrt{1 - \xi^2}$ with $\xi = \frac{y}{\frac{b}{2}}$ and $\Gamma_0 = \frac{2bU_\infty}{\pi AR} C_L$. This wing will be placed in a domain of size $[2b \times 2b \times 4b]$ with $\frac{h}{b} = \frac{1}{64}$. We will use a uniform upstream velocity in the z direction, place the wing in the center of the domain at $[b, b, 2b]$, and orient it in the y direction. In this case, we will use a polar such that $C_L(\alpha) = 2\pi\alpha$ and $C_D(\alpha) = 0$. Finally, we recall a theoretical result of PLL theory, namely that the downwash angle is assumed to be uniform over the span, $\epsilon_{ell} = \frac{2}{AR+2} \alpha_g$.

Moreover, although we introduced the LES term in section 1.2, we will not activate it in what follows, except when we will compare results with the literature in the second part of subsection 3.3. Indeed, after a few tests, we found that the error between Vpm4x and Murphy's results is a bit bigger when using LES. So, since the aim is to demonstrate the validity of the ILL implementation, we are going to demonstrate it without using LES to be as precise as possible. However, the additional error due to the LES term is relatively small and constant. The use of this term to compare results with those in the literature is therefore considered acceptable.

In addition, to determine the functionality of the multi-processor implementation, we will use a domain decomposition into 32 blocks, distributed between 32 processors for the rectangular wing domain, and a domain decomposition into 16 blocks, distributed between 16 processors for the elliptical wing domain.

Finally, since we aim to simulate a physical phenomenon, we will use a simulation domain with unbounded boundary conditions. To be closer to physics, we should use an inflow/outflow condition for the z direction. However, such a condition is beyond the scope of this thesis. Furthermore, the objective is to prove the functionality of the ILL and this can be done using simple unbounded boundary conditions. For this purpose, we will always compare results before any vorticity leaves the computational domain, as Murphy and Vpm4x boundary conditions are not necessarily implemented in the same way.

Physical expectations

Let us look at the physics of a wing and the expected flow. Of the two types of wings we are going to study here, the elliptical wing is the one that has the best properties. Indeed, unlike the rectangular wing, the elliptical wing, thanks to its shape, has an elliptical circulation distribution, which reduces both lift and drag when approaching the edge of the wing. The result is better aerodynamics. It can be defined as an ideal wing. So, we present here the different physical phenomena on the elliptical wing.

First of all, there are several phenomena to observe. The first is the presence and creation of a starting vortex. This vortex runs parallel to the wing and is created at the start of the simulation. Although it is not observable in reality, we will observe it during the simulation, as it is the result of circulation conservation. Figure 3.1 shows the flow around this elliptical wing. In this image, we can see the formation of two vortex tubes formed by a set of vortex lines. By continuity of the vortex tubes, a vortex sheet forms in the middle of these two tubes. In this figure,

(a) Slice $z = 0.5$ view at $t^* = 6.25$ (b) Lateral view at $t^* = 3.125$

Figure 3.1: Global view of the flow around an elliptical wing: views of ω^* with a saturation of 12.

we can also observe that the vortex sheet is oriented towards the bottom of the domain, as expected by physics.

Finally, the only purpose of the simulations presented here is to demonstrate the validity of the code. For this reason, the simulation domain is tail-limited. However, if we take a longer domain in the flow direction and run a longer simulation, we may observe crow instability. This phenomenon is due to flow instabilities. After a while, the two trailing vortices will interact with each other at a critical point and form vortex rings, as shown in figure 3.2.

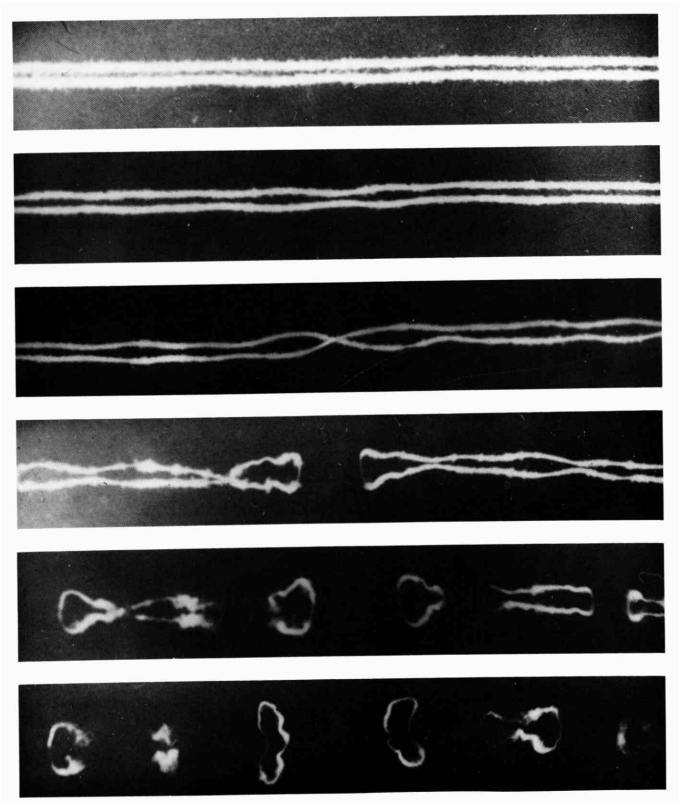


Figure 3.2: Crow instability of a pair of trailing vortices. From [24]

3.2 Rectangular wing

Figure 3.3, shows the distribution of the ω^* norm in the domain, obtained from Murphy.

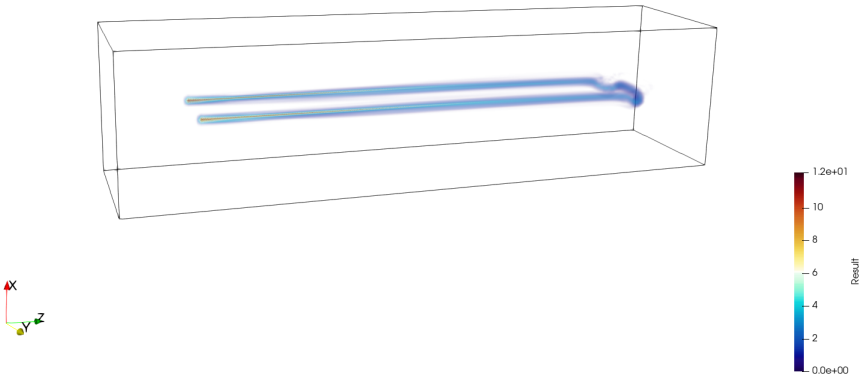
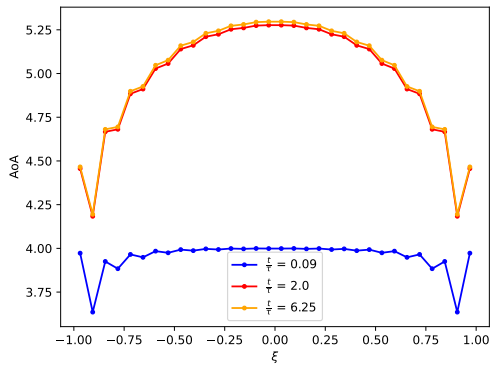


Figure 3.3: Space-developing simulation of the vortex-sheet rollup: 3D perspective view of ω^* at $t^* = 6.25$.

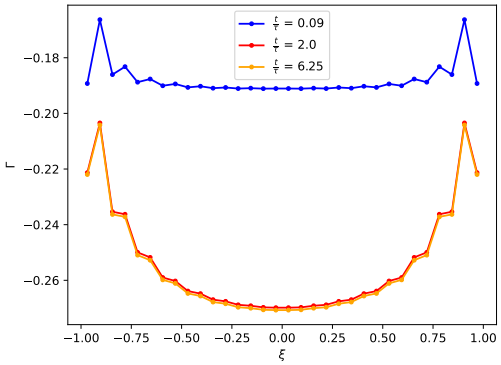
As explained in the introduction, we can observe the presence of the starting vortex on the right-hand side of the figure 3.3. We also see the two tip vortices created at both ends of the wing. To determine Murphy's validity more precisely, we will study the evolution of the line diagnostics such as angle of attack (angle between chord and local velocity), lift coefficient, and circulation. Figure 3.4 show us the evolution of these three diagnostics during simulations with the reference time $\tau = \frac{U_\infty T}{b} = 1$.

Figure 3.4a shows an increase in the angle of attack. This phenomenon can be attributed to two factors. Firstly, it is due to the application of a filter on the AoA (as explained in section 2.4.3). Secondly, it is also because of the downwash angle, ϵ. The downwash effect is due to the induced velocity in the region close to the line that is affected by the line. This effect is much more pronounced at the start of the simulation, due to the starting vortex. We also observe this effect on the steady-state angle of attack. Indeed, we see that the angle of attack stagnates at around 5.25 degrees where $\alpha_g = 6°$. Moreover, this effect can also be observed on both sides of the wing. Here, the downwash is due to the velocity induced by the tip vortices. As for the lift coefficient and circulation, they are proportional to the angle of attack. This is consistent with the equations used to obtain them, in which we observe this proportionality.

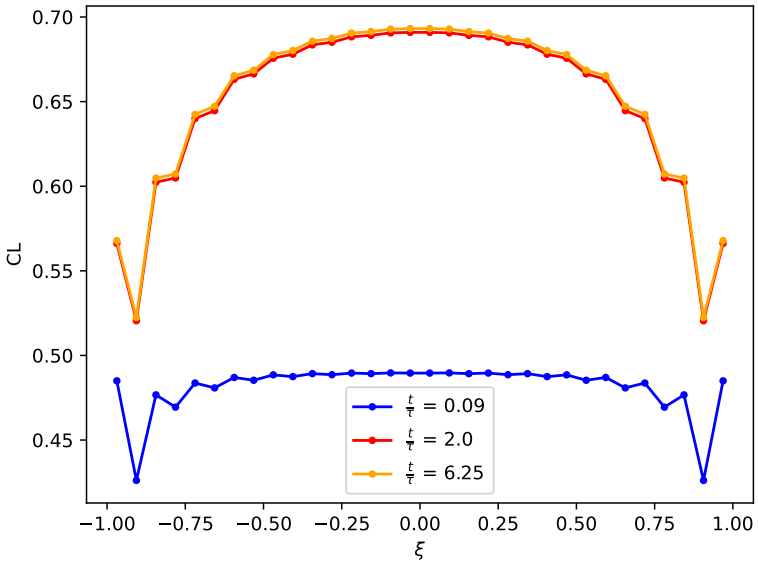
All the previous results seem coherent. We can now compute the difference



(a) angle of attack distribution [°]



(b) circulation distribution



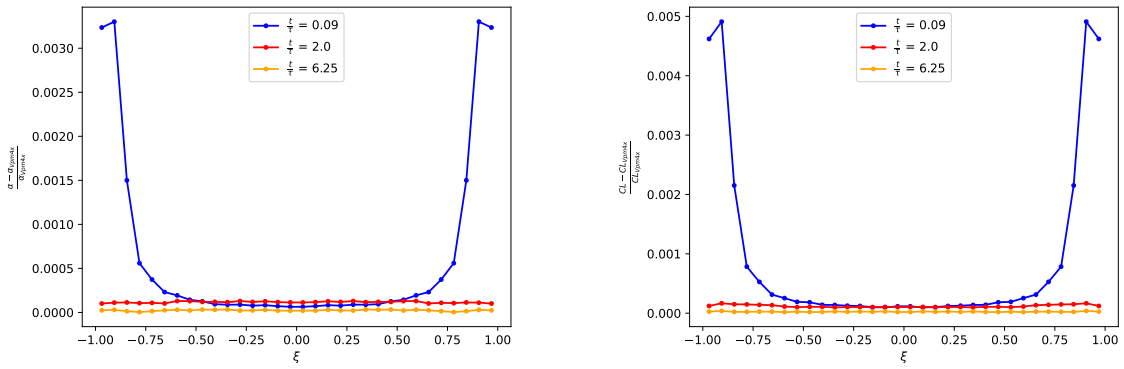
(c) lift distribution

Figure 3.4: Results of ILL for a rectangular wing for three different times and with constant $\frac{\sigma}{h} = 1$.

between the line diagnostics provided by Vpm4x and those provided by Murphy to determine their accuracy. To do this, we will use the following relative percentage error formula

$$err_{rel} = \frac{|diag_{Murphy} - diag_{Vpm4x}|}{diag_{Vpm4x}} 100. \tag{3.2}$$

Figure 3.5 shows the evolution of this error for angle of attack and lift along the line, over three different times.



(a) AoA error [%]

(b) lift error [%]

Figure 3.5: Error of ILL for a rectangular wing for three different times and with constant $\frac{\sigma}{h} = 1$.

These figures show a relative error of less than one-hundredth of a percent, which does not increase during the simulation. However, we note that at the start of the simulation, the line diagnostics are a bit more different between Murphy and Vpm4x. The origin of this small additional error may be either a small difference in the angle of attack filter between the two codes, or an amplification of the small error due to the huge amount of vorticity at the start of the simulation corresponding to the starting vortex. However, these slight differences at the start disappear when the flow becomes established which is the most important point.

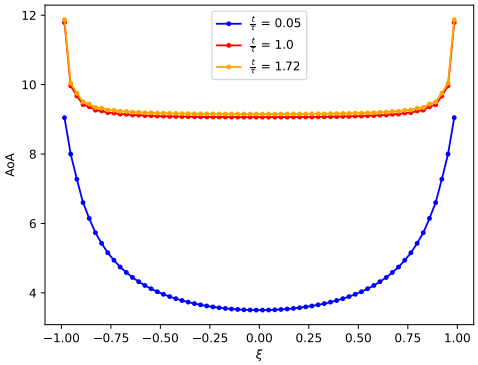
To conclude this first validation section, we have understood the observed evaluations of certain diagnostics and the global flow, and we have demonstrated the accuracy and functionality of the code.

3.3 Elliptical wing

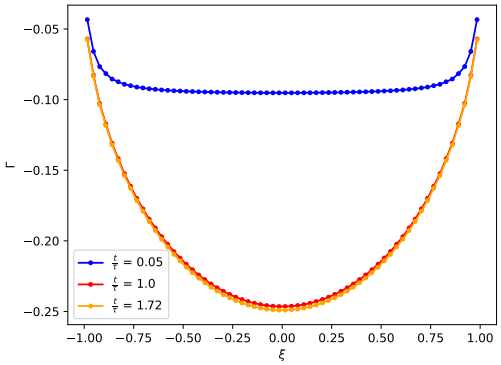
We can now study the flow around the elliptical wing to check the accuracy of the results once again.

To begin with, we will carry out the same validation as in the previous subsection. Figure 3.6 shows the evolution of the AoA, lift coefficient, and circulation. We can see an elliptical circulation distribution compared to the one in the rectangular wing case. We also observe the better refinement used in this case (as we use a $\frac{b}{h} = \frac{1}{64}$) because we do not observe the fluctuation of values at the wing tip.

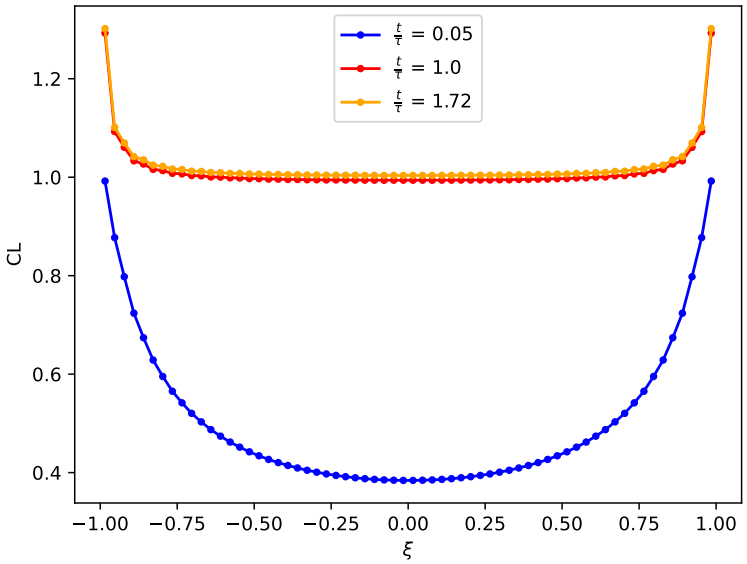
Then, Figure 3.7 shows the error between Murphy's results and those provided



(a) angle of attack distribution [°]



(b) circulation distribution

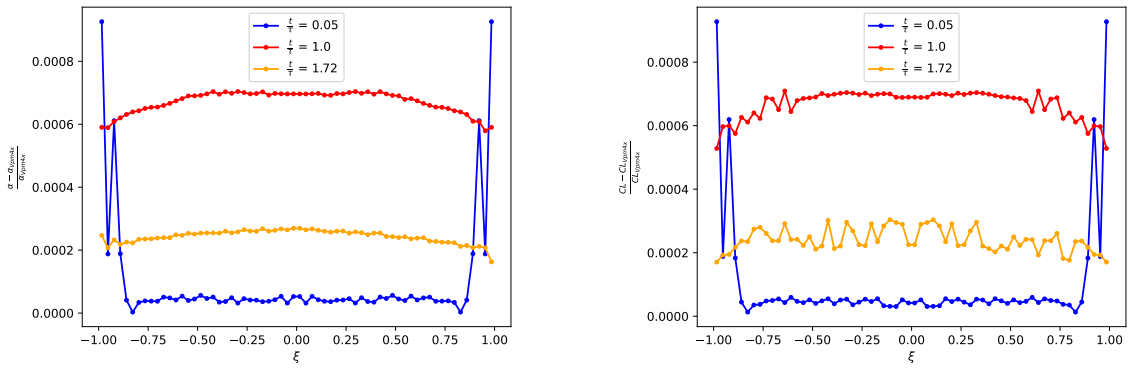


(c) lift distribution

Figure 3.6: Results of ILL for an elliptical wing for three different times and with constant $\frac{\sigma}{h} = 1$.

by the Vpm4x code. Once again, the error is very small, confirming the accuracy of the implementation. Moreover, we observe that better refinement seems to lead to closer results. This implies a convergence of the implemented method.

For a final validation, we will now compare the results with those obtained by Caprace et al. in [14]. In this document, Caprace et al. present the results obtained on the elliptical wing described above. Moreover, to be coherent with the procedure used in [14], we will linearly vary the AoA from $\alpha = 0$ to $\alpha = \alpha_g$ such



(a) AoA error [%] (b) lift error [%]

Figure 3.7: Error of ILL for an elliptical wing for three different times and with constant $\frac{\sigma}{h} = 1$.

that $C_L(\alpha_g) = 1$ in a time corresponding to $\tau = \frac{TU_\infty}{b} = 1$. Then, to demonstrate the validity of ILL Murphy's implementation, we will obtain two graphical results provided by Caprace et al..

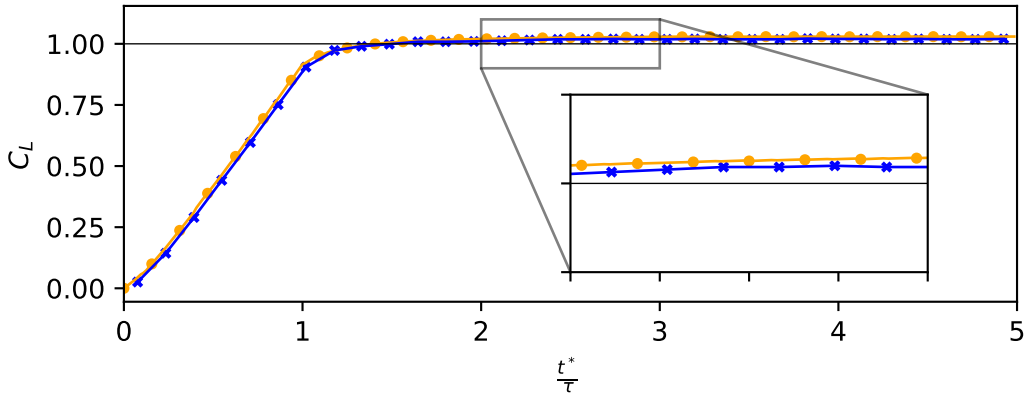


Figure 3.8: Time evolution of the lift coefficient developed by the ILL on the elliptical wing, as evaluated by Murphy (—●—) and as shown in [14] (—×—). The steady state value obtained using the classical PLL (———) is also displayed.

Firstly, figure 3.8 shows the evolution of the lift coefficient over time. In this figure, we observe a relatively good correspondence between the two curves. Slight differences may be due to the fact that the results obtained in the literature were obtained using an inflow/outflow condition whereas, as we have already explained, this feature is not implemented in Murphy.

Secondly, Figure 3.9 shows the evolution of the downwash angle along the line

when the flow is established. Once again, the literature results were obtained using inflow/outflow conditions, which may lead to small errors. However, we can observe better correspondence between the curves. These better results can be explained by the fact that we analyze the evolution of the downwash angle along the line, whereas figure 3.8 presents an evolution of the integral of the lift coefficient over time. Moreover, we observe smaller values than the expected uniform distribution ϵ_{ell} which can explain the bigger lift coefficient value than that predicted by the PLL observed in figure 3.8.

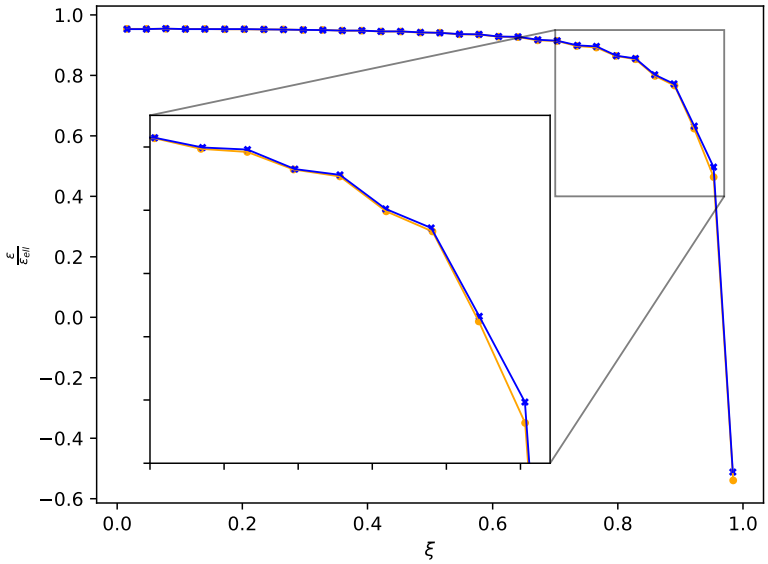


Figure 3.9: Down wash angle obtained by the ILL on the elliptical wing at steady states, as evaluated by Murphy (—●—) and as shown in [14] (—×—). The classical PLL prediction corresponds to a uniform downwash ϵ_{ell}.

To conclude this second validation, Murphy's results for the elliptical wing match those of Vpm4x and the literature. Thanks to this, we have demonstrated the validity of Murphy's ILL implementation.

3.4 ILL implementation performance

To reduce computational time, the lifting lines are implemented in such a way as to operate in parallel computation. As explained above, such an implementation requires not only communication between processors but also management of line information. In section 2.5, we saw the method used in Murphy to manage line

information. However, Vpm4x does not use the same technique. That is why it is interesting to evaluate the performance of our implementation, in order to determine the efficiency of the code, as well as to decide on potential improvements and their usefulness. To this end, we will present here a study of the time taken by various parts of the code during a simulation. We will study the time taken by the simulation carried out in section 3.2.

First of all, figure 3.10 shows us the distribution of time between operations linked to the lifting line computation and operations that are not, this being during the computation of sub-time steps. In this figure, we can see that the proportion of time taken up by ILL-related operations is around 40%. This seems a lot, so we are going to investigate where this value comes from. To do this, let us look at operations related to lifting lines. These include the mesh-to-particle interpolation performed before and after the solving of the Poisson equation at each sub-time step, as well as all the other operations, i.e. the communication and computational operations to determine tracer and attached particles velocities, to compute the circulation, and to advect tracer and discretize the vortex sheet.

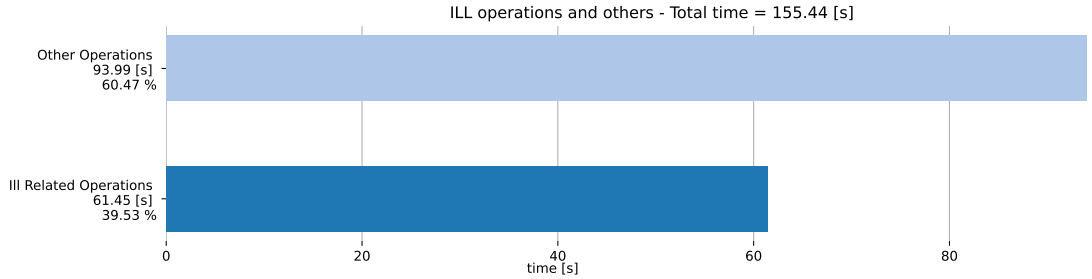


Figure 3.10: Time distribution between ILL related and ILL not related operations.

Figure 3.11 shows that only 4.7% of the time is spent on other operations. However, the remaining 95.3% is spent exclusively to compute interpolations. The optimization of these interpolations is beyond the scope of this thesis and is independent of the choices made when implementing lifting lines. We can therefore conclude that the operations relating exclusively to the lifting lines, and which are specific to the implementation carried out within the framework of this thesis, represent less than 2% ($39.53\% \cdot 4.71\% = 1.86\%$) of the total computation time. This allows us to assert that the performance of the implementation carried out as part of this work is more than adequate and that improvements will not bring any major performance gains.

Nevertheless, we can look at the distribution of these 2% between the different operations. Figure 3.12 shows the times taken by the different operations relating exclusively to ILL. Firstly, it should be noted that advection and circulation

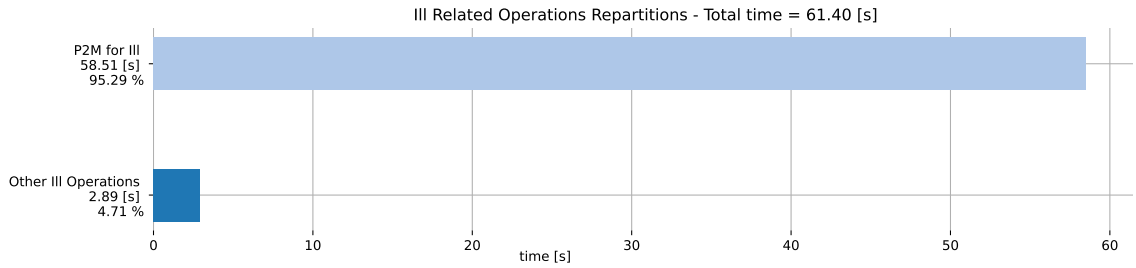


Figure 3.11: Time distribution of ILL-related operations between interpolation and other operations.

computation operations are not included in this figure, as they represent a minimal amount of time. Secondly, it can be seen that most of the time is spent computing tracer velocities. The figure also shows that in all three cases, most of the time is spent on communication, and very little on computation.

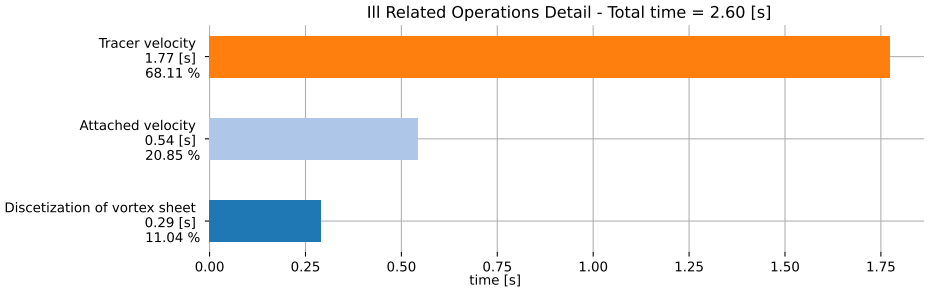


Figure 3.12: Time distribution between the specific to ILL operations. The hashed part of the bar correspond to the communication time and the not hashed to the computational time.

To conclude this section, other implementations are possible, such as the one implemented in Vpm4x, which involves sending the right information to the right processor and having each processor computing its own part of the line. However, given the extremely low computation time required for the ILL part of the code, it might be more interesting to work on optimizing other parts, such as interpolation, to improve performance. So here we have a functional implementation of ILL that can be solved in a very reasonable time.

Conclusion

In this work, we implemented the immersed lifting line model developed in the past in Vpm4x by Caprace during his PhD [5]. This model makes it possible to include a linear device in the flow, such as a wind turbine blade or an aircraft wing, using a circulation-generating line. We also obtained a series of results to demonstrate how the implementation works.

In a first step, we studied Murphy's overall operation and the adaptations needed to move from Fortran code to object-oriented code. We then presented in detail the entire algorithm, including the immersed lifting line model. As a reminder, this model allows us to lagrangianly discretize the different vorticities created by the lifting line that are released into the flow, and then add them to the flow.

Then, we saw how the algorithm could be adapted to multi-processor operation. This crucial step makes the code much more efficient, and in the future will enable much more complex simulations to be carried out in a reasonable time.

We then analyzed a set of results obtained by Murphy's simulation with an elliptical wing and a rectangular wing. These results were compared with those obtained by the Vpm4x simulation, as well as with those obtained by Caprace during his PhD thesis. All these analyses enabled us to demonstrate the accuracy of the code and validate it.

Lastly, we carried out a small performance study. This study enabled us to demonstrate that the part of the implementation carried out in this work represents only 2% of the total simulation time. It also showed that almost all of this 2% is due to inter-processor communication.

Finally, as an extension to this work, different things can be envisaged:

1. Firstly, as we have already seen, adapting the code to operate with multiple lifting lines as well as moving lines would be very useful for modeling wind farms or aircraft in formation.

2. Secondly, modifications could be made to couple the code with robotran (multibody solver), as has already been done in the past with Vpm4x [25, 26]. The use of a multibody solver could be useful for modeling several moving bodies, such as the various wind turbines that would make up a wind farm, whose layout we would like to optimize.

3. Thirdly, as we saw in Chapter 2, current lifting lines operate exclusively in uniform resolution. It might be worth considering modifying the code to operate in non-uniform resolution.

4. Fourthly, further optimization of the multiprocessor implementation would probably improve performance. However, as we have already seen, the gains would remain quite small. For this reason, such an improvement does not seem necessary.

5. Fifthly, in this paper, we validate the code by comparing the results with those of Vpm4x. This comparison assumes that the Vpm4x code is correct. Such an assumption seems acceptable, given the numerous uses of vpm4x code in scientific papers. However, we could ask ourselves whether this hypothesis is acceptable and therefore whether a code-by-code validation is a good validation. In order to certify the validity of the results, it would be interesting, in the future, to compare the results obtained with other results such as experimental ones.

This work, combining mechanical engineering, computer science, and mathematics, has also made us aware of the cost of large-scale numerical simulations and the importance of optimizing the code to reduce them. By bringing together numerous concepts from the three previous engineering fields, we have succeeded in establishing effective models for studying exciting phenomena that can lead to innovations in the technology sector.

Bibliography

[1] F. Trigaux, M. Moens, P. Chatelain, and G. Winckelmans. Tilted wind turbines in farm configuration for improved global efficiency. *Journal of Physics: Conference Series*, Vol. 1618, No. 6, 2020, pp. 062035. DOI: 10.1088/1742-6596/1618/6/062035 .

[2] P. Benard, A. Viré, V. Moureau, G. Lartigue, L. Beaudet, P. Deglaire, and L. Bricteux. Large-Eddy Simulation of wind turbines wakes including geometrical effects. *Computers & Fluids*, Vol. 173, 2018, pp. 133-139. DOI: https://doi.org/10.1016/j.compfluid.2018.03.015 .

[3] L. Prandlt. Tragflügeltheorie. *Göttinger Nachrichten, Mathematisch-Physikalische Klasse, Germany*, 1919, pp. 451-477.

[4] T. Gillis, P. Balty, J Gabbard, and M. Duponcheel. Murphy. `https://github.com/vanreeslab/murphy`.

[5] D.-G. Caprace. Modeling of lifting-dragging devices for large eddy simulation of space-developing wakes Application to wings, rotors and formation flight. PhD thesis. *Institute of Mechanics, Materials and Civil Engineering*, 2020.

[6] G. Winckelmans. Meca 2323 : Aerodynamics of external flows. *Course note*, 2019.

[7] W. Kutta. Auftriebskräfte in strömenden Flüssigkeiten. *Illustrierte Aeronautische Mitteilungen*, Vol. 6, No. 133, 1902, pp. 133–135.

[8] P. Balty and J. Waucquez. Numerical study of the aerodynamics and wakes of VAWTs in complex situations : double-rotor and offshore single rotor. *Ecole polytechnique de Louvain, Université catholique de Louvain*, 2019.

[9] O. Thiry. Investigation of wall shear stress models for large eddy simulation. PhD thesis. *Institute of Mechanics, Materials and Civil Engineering*, 2016.

[10] P. Chatelain, Y. Marichal, M. Duponcheel, M. Moens, and G. Winckelmans. Vortex methods applied to the large eddy simulation of wind turbine flows. 2015.

[11] W. F. Phillips and D. O. Snyder. Modern Adaptation of Prandtl's Classic Lifting-Line Theory. *Journal of Aircraft*, Vol. 37, No. 4, 2000, pp. 662–670. DOI: https://doi.org/10.2514/2.2649 .

[12] T. Gillis and W. M. van Rees. MURPHY – A scalable multiresolution framework for scientific computing on 3D block-structured collocated grids. *Journal of Scientific Computing*, Vol. 51, No. 6, 2022, pp. 1864–1872. DOI: 10.48550/ARXIV.2112.07537 .

[13] P. Chatelain, A. Curioni, M. Bergdorf, D. Rossinelli, W. Andreoni, and P. Koumoutsakos. Billion vortex particle direct numerical simulations of aircraft wakes. *Computer Methods in Applied Mechanics and Engineering*, Vol. 197, No. 13, 2008, pp. 1296–1304. DOI: 10.1016/j.cma.2007.11.016 .

[14] D.-G. Caprace, P. Chatelain, and G. Winckelmans. An Immersed Lifting and Dragging Line Model for the Vortex Particle-Mesh Method. *Theoretical and Computational Fluid Dynamics*, Vol. 34, No. 1, 2020, pp. 21-48. DOI: 10.1007/s00162-019-00510-1 .

[15] C. BURSTEDDE, L. C. WILCOX, and O. GHATTAS. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, Vol. 33, No. 3, 2011, pp. 1103–1133. DOI: 10.1137/100791634 .

[16] J. H. Williamson. Low-Storage Runge-kutta Schemes. *Journal of computational physics*, Vol. 35, 1980, pp. 48-56.

[17] P. Balty, P. Chatelain, and T. Gillis. FLUPS - A Flexible and Performant Massively Parallel Fourier Transform Library. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 34, No. 7, 2023, pp. 2011-2024. DOI: 10.1109/TPDS.2023.3254302 .

[18] D.-G. Caprace, T. Gillis, and P. Chatelain. FLUPS: A Fourier-Based Library of Unbounded Poisson Solvers. *SIAM Journal on Scientific Computing*, Vol. 43, No. 1, 2021, pp. C31–C60. DOI: 10.1137/19m1303848 .

[19] M.J. Churchfield, S. Schreck, L.A. Martinez-Tossas, C. Meneveau, and P.R. Spalart. An Advanced Actuator Line Method for Wind Energy Applications and Beyond. *35th Wind Energy Symposium, AIAA*, 2017, p. 1998. DOI: https://doi.org/10.2514/6.2017-1998 .

[20] D.-G. Caprace, P. Chatelain, and G. Winckelmans. Lifting Line with Various Mollifications: Theory and Application to an Elliptical Wing. *AIAA Journal*, Vol. 57, No. 1, 2019, pp. 1–12. DOI: https://doi.org/10.2514/1.J057487 .

[21] F. Scheurich and R. E. Brown. Modelling the aerodynamics of vertical-axis wind turbines in unsteady wind conditions. *Wind Energy*, Vol. 16, No. 1, 2013, pp. 91–107. DOI: 10.1002/we.532 .

[22] R. E. Brown and A. J. Line. Efficient High-Resolution Wake Modeling Using the Vorticity Transport Equation. *AIAA journal*, Vol. 43, No. 7, 2005, pp. 1434–1443. DOI: 10.2514/1.13679 .

[23] Timmer WA. Two-Dimensional Low-Reynolds Number Windtunnel Results for Airfoil NACA 0018. *Wind Engineering*, Vol. 32, No. 6, 2008, pp.525-537. DOI: 10.1260/030952408787548848 .

[24] S. C. Crow. Stability Theory for a Pair of Trailing Vortices. *AIAA journal*, vol. 8, No. 12, 1970, pp. 2172-2179.

[25] S. Buffin. Simulation of rotorcraft dynamics through the coupling of a multi-body solver and a Vortex Particle-Mesh method. *Ecole polytechnique de Louvain, Université catholique de Louvain*, 2016.

[26] P. Balty, D. -G. Caprace, J. Waucquez, M. Coquelet, and P. Chatelain. Multiphysics simulations of the dynamic and wakes of a floating Vertical Axis Wind Turbine. *Journal of Physics: Conference Series*, Vol. 1618, no. 1, 2020, p. 062053.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**