PyFly: A Fast, Portable Aerodynamics Simulator

D. Garcia[a,b], M. Ghommem[c,*], N. Collier[d], B. Varga[e], V. M. Calo[f]

[a]*Basque Center for Applied Mathematics, (BCAM), Bilbao, Spain.*
[b]*University of the Basque Country (UPV/EHU), Leioa, Spain.*
[c]*Department of Mechanical Engineering, American University of Sharjah (AUS), Sharjah, United Arab Emirates.*
[d]*Oak Ridge National Laboratory, Oak Ridge, TN, USA.*
[e]*Earth Science & Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia.*
[f]*Applied Geology Department, Curtin University, Perth, Western Australia.*

Abstract

We present a fast, user-friendly implementation of a potential flow solver based on the unsteady vortex lattice method (UVLM), namely PyFly. UVLM computes the aerodynamic loads applied on lifting surfaces while capturing the unsteady effects such as added mass forces, growth of bound circulation, and wake while assuming that the flow separation location is known a priori. The computational framework uses the Python programming language to provide a easy handle user interface while the computational kernels are written efficiently in Fortran. The mixed-language approach enables high performance in terms of solution time and high flexibility in terms of easiness of code adaptation to different system configurations and applications. This computational tool is intended to predict the unsteady aerodynamic behavior of multiple moving bodies (e.g., flapping wings, rotating blades, suspension bridges...) subject to an incoming air. We simulate different aerodynamic problems to illustrate the usefulness and effectiveness of PyFly.

Keywords: Unsteady aerodynamics, numerical simulations, mixed-language approach, potential flow.

1. Introduction

The performance of aerodynamic systems, such as air vehicles, suspension structures, and wind turbines, could be assessed at the earliest stages of design through the deployment of computational tools. Recent advances in computer hardware and software have overcome the computational burden associated with the numerical integration of the equations governing the aerodynamic performance of the aforementioned systems. However, large scale and intensive computations still require the use of compiled languages (e.g., C/C++, Fortran) to obtain reasonable simulation times. Integrating the set of flow governing equations into a single large code (for instance, completely written in C++) may be complex for users and scientists and may lack flexibility and easiness in adapting different configurations and applications. The demanded flexibility can be hardly achieved by deploying such high-performance programming language. Python presents a convenient and open source working environment but remains inappropriate for intensive computation. As such, the mixed-language approach seeks to resolve the issues related to both performance and flexibility. The concept of a computational platform supporting data exchange between scripts programmed in different languages has been previously employed for multi-disciplinary optimization (e.g., pyOPT [1], and pyMDO [2]), high-fidelity simulations of nonlinear hyperbolic PDEs (e.g., SOLVCON [3]), and isogeometric analysis (e.g., PetIGA [4]).

In this paper, we present a fast and efficient numerical implementation of a potential flow solver based on the unsteady vortex lattice method (UVLM), we name this platform PyFly. This computational tool is designed to simulate the unsteady aerodynamic behavior of a wide range of moving and deforming bodies. The aerodynamic loads are obtained by pressure differences across the body surface resulting from acceleration- and circulation-based phenomena.

*Corresponding author
Email address: mghommem@aus.edu (M. Ghommem)

The UVLM formulation accounts for unsteady effects such as added mass forces, the growth of bound circulation, and the wake. This approach applies only to ideal fluids (i.e., those involved in incompressible, inviscid, and irrotational flows), where the separation lines are known a priori. UVLM has been widely used for the analysis and design of avian-like flapping wings in forward and hover flights [5–12], modeling wind turbines [13–18], dynamic analysis of offshore structures [19–21] and control and vibration suppression of civil engineering structures [22, 23].

While the numerical flow solver does not change for different applications, the requirements which different applications present can become complex to manage. For example in the case of flapping wings, there is the wing and its wake to manage. For flights at low altitude under the ground effect, we must also track the effect of the wing's mirror image. However, in the case of a wind turbine, each rotating blade and its wake is modeled as a separate object which interacts with the entire system. To simplify the user-burden in writing solvers for different scenarios, we implement a base grid class in Python which handles the grid itself and its wake. Furthermore, this class can compute its influence on other grid objects. This abstract approach makes the framework simple to use, while core components run in Fortran for efficiency. This freely-available software reproduces all numerical results discussed in this paper in the demo section [24].

The remainder of the paper is organized as follows: first, we briefly present the theoretical background of UVLM. Then, we describe the numerical implementation of PyFly and detail its main features. The work concludes with the simulation of a set of aerodynamic problems to illustrate the performance and flexibility of the computational tool. In closing, we summarize our findings and describe the future work.

2. UVLM: Theoretical Background

The unsteady vortex lattice method (UVLM) facilitates the study of the aerodynamic effects in slender bodies. The method predicts the dynamics of the problem by solving the flow around the bodies. The effects of the compressibility and viscosity are neglected in UVLM by considering an idealized potential flow.

Vorticity is the main fluid quantity used in UVLM to compute the dynamic evolution of fluids. The vorticity is generated along the boundary layer as a consequence of the viscous forces and advected downstream into the fluid. The region of the fluid enclosing the vorticity is called wake. In UVLM, the regions confining the vorticity are considered thin enough to be described as sheets of vorticity. This consideration agrees with the exclusion of viscous effects due to the assumed ideal potential flow. The vortex sheet description for the case of a lifting body submerged in a fluid is illustrated in Figure 1, where the red arrows are the circulation vectors of the vortex rings.

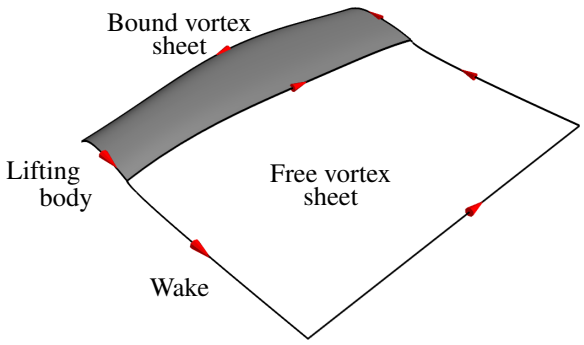


Figure 1: Bound vortex description.

The vortex sheet description consists of a bound vortex and a free vortex. The bound vortex sheet describes the boundary layer of the lifting body. The displacement of the bound vortex is prescribed as a result of the solid surface of the body and a jump in the pressure may exist across this segment of the sheet as a consequence of the interaction of the solid body with the incoming freestream. The free vortex sheet describes the wake and has no prescribed displacement implying the sheet deforms freely. In the free vortex, there is no jump in pressure due to the free deformation of the wake. The only flow separation considered in the vortex sheet description is performed at the trailing edge connecting the bound vortex and the free vortex, implying the Kutta condition is satisfied in UVLM.

2.1. UVLM scheme

In this section, we describe the numerical computation performed in UVLM. The method discretizes the vortex sheets used in the description of the problem in a lattice of vortex elements known as vortex rings. A typical discretization of a vortex sheet description of a lifting body problem is presented in Figure 2, where $\underline{p}^i_1$, $\underline{p}^i_2$, $\underline{p}^i_3$, $\underline{p}^i_4$ are the

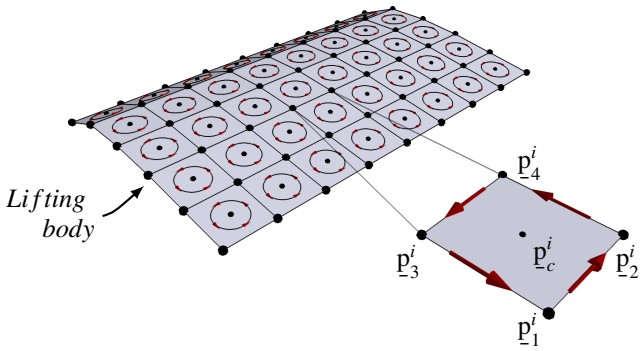


Figure 2: Bound vortex discretization.

corners of a vortex ring and $\underline{p}^i_c$ is the center point of the i-th vortex element. The vortex element is a grouping of four straight vortex segments with the same value in circulation.

To compute the circulation, we impose a no-penetration condition along the body surface, that is, a zero velocity in the normal direction to the quadrangle and is imposed at the center of each vortex element of the body. These conditions read

$$\left[(\underline{u}_\infty + \underline{u} - \underline{v}) \cdot \underline{n}\right]^b_j = \underline{0} \quad \text{in} \quad \Omega \tag{1}$$

where $\underline{u}_\infty$ refers to the freestream velocity, $\underline{u}$ is the velocity induced by the vorticity, $\underline{v}$ is the velocity of the body, and $\underline{n}$ is the normal vector at the $j-th$ vortex element of the bound vortex sheet discretization (b) located over the lifting body. The normal vector is computed using the cross product between the partial derivatives of the surface parametrization of the vortex sheet. The partial derivatives are computed at the center point $\underline{p}^j_c$

$$\underline{n}_j = \frac{\dfrac{\partial \underline{p}^j_c}{\partial \xi} \times \dfrac{\partial \underline{p}^j_c}{\partial \eta}}{\left| \dfrac{\partial \underline{p}^j_c}{\partial \xi} \times \dfrac{\partial \underline{p}^j_c}{\partial \eta} \right|} \tag{2}$$

where ξ and η refer to two different directions on the tangent plane to the surface at the center point.

The velocity $\underline{u}$ is split into two terms in order to make a distinction between the velocity induced by the vorticity of the bound vortex and the vorticity at the free vortex.

$$\underline{u}^b_j = \left(\underline{u}^{b,b}_j\right) + \left(\underline{u}^{b,f}_j\right) \tag{3}$$

where b refers to the bound segment and f to the free segment of the vortex sheet description. The first term is related with the velocity induced by the vorticity at the bound vortex while the second term is related to the free vortex. Substituting Equation (3) into Equation (1), we obtain

$$\left(\underline{u}^{b,b}_j + \underline{u}^{b,f}_j\right) \cdot \underline{n}_j = (\underline{u}_\infty - \underline{v})_j \cdot \underline{n}_j \quad \text{in} \quad \Omega, \tag{4}$$

thus, Equation 4 can be expressed as

$$\underline{u}^{b,b}_j \cdot \underline{n}_j = \left(\underline{V}_j - \underline{u}^{b,f}_j\right) \cdot \underline{n}_j \quad \text{in} \quad \Omega \tag{5}$$

where $\underline{V}_j = (\underline{u}_\infty - \underline{v})_j$.

UVLM employs the Biot-Savart equation in order to compute the velocity induced by the vorticity field; that is,

$$\underline{u}(\underline{y}_j) = \frac{1}{4\pi} \oint \frac{\Gamma_i(\underline{x}_i) d\ell(\underline{x}_i) \times (\underline{x}_i - \underline{y}_j)}{|\underline{x}_i - \underline{y}_j|^3} \tag{6}$$

In the Biot-Savart equation, Γ_i is the circulation of the i-th vortex element, $d\ell$ is the vortex filament length, and $\underline{u}$ is the velocity computed at the center point $\underline{p}_c$ of the j-th vortex element. The velocity induced by vorticity is expressed as follows

$$\underline{u}_j^{b,b} = \underline{\underline{A}}_{ij}^{b,b} \underline{\Gamma}_i^b \tag{7}$$

$$\underline{u}_j^{b,f} = \underline{\underline{A}}_{ij}^{b,f} \underline{\Gamma}_i^f \tag{8}$$

where

$$\underline{\underline{A}}_{ij}^{b,b/f} = \frac{1}{4\pi} \oint \frac{d\ell(\underline{x}_i) \times (\underline{x}_i - \underline{y}_j)}{|\underline{x}_i - \underline{y}_j|^3} \tag{9}$$

Figure 3 illustrates the velocity induced by one segment of the vorticity. The formulation of the Biot-Savart equation is given in Appendix A.

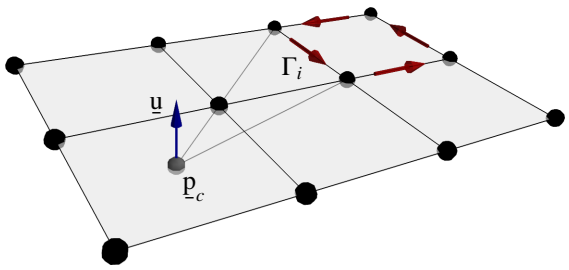


Figure 3: Velocity induce by vorticity.

Substituting Equations (7) and (8) in Equation (5), we obtain

$$\left(\underline{\underline{A}}_{ij}^{b,b} \Gamma_i^b\right) \cdot \underline{n}_j = \left(\underline{V}_j - \underline{u}_j^{b,f}\right) \cdot \underline{n}_j \tag{10}$$

Once Equation (10) is solved and the circulation in every bound segment of the sheet description (Γ_i^b) is computed, the circulation is advected into the wake. The advection carries in the downwash direction the vortex elements at the trailing edge. The value of the circulation remains constant during the advection to satisfy Kelvin's circulation theorem. Figure 4 illustrates the advection of circulation into the wake from time t_0 until time t_2.

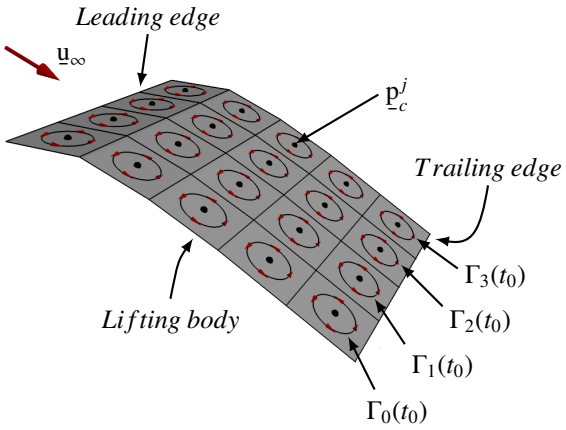
The wake location is updated to preserve the continuity of the pressure field across the sheet. The displacement of the local fluid particles at the wake is computed using the following equation

$$\underline{u}_j^f = \underline{\underline{A}}_{ij}^{f,f} \cdot \Gamma_i^f + \underline{\underline{A}}_{ij}^{f,b} \cdot \Gamma_i^b \tag{11}$$

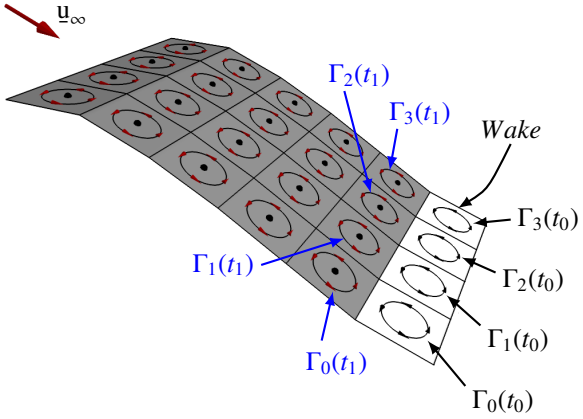
and the new position of the wake is

$$\underline{x}_i^f(t + \Delta t) = \underline{x}_i^f(t) + \underline{u}_i^f \Delta t \tag{12}$$

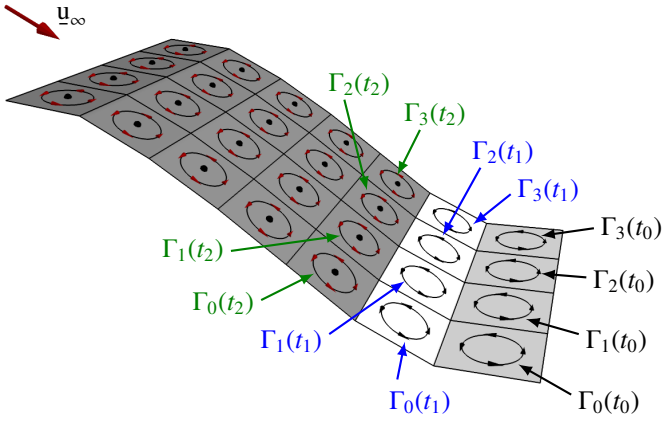
UVLM computes the circulation and updates the wake location at each time step. In the initial step, there is no wake and the term $\underline{u}_j^{b,f}$ in Equation (5) is assumed zero (case shown in Figure 4a).



(a) $t = t_0$



(b) $t = t_1$



(c) $t = t_2$

Figure 4: Advection of the circulation into the wake.

2.2. *Aerodynamic forces*

The aerodynamic loads induced by the interactions between the solid body and the incoming freestream are computed using the approach described by Katz and Plotkin [25]. This approach considers the leading-edge suction force.

The forces are computed as the sum of the contributions of each element to the loads. Equation (13) is used to computed the aerodynamic forces.

$$\underline{\mathrm{F}} = \sum_{k=1}^{N} \underline{\mathrm{F}}_k = \sum_{k=1}^{N} \Delta\underline{\mathrm{L}}_k \cdot \underline{\mathrm{e}}_k^{lift} + \Delta\underline{\mathrm{D}}_k \cdot \underline{\mathrm{e}}_k^{drag} \tag{13}$$

where $\Delta\underline{\mathrm{L}}_k$ is the lift contribution and $\Delta\underline{\mathrm{D}}_k$ is the drag contribution of the k-th element. The contribution to the lift and drag are applied along the local vectors $\underline{\mathrm{e}}_k^{lift}$ and $\underline{\mathrm{e}}_k^{drag}$.

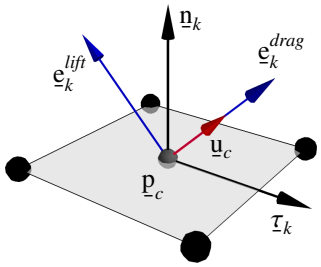


Figure 5: Vectors used in dynamic computation.

The local vectors $\underline{\mathrm{e}}_k^{lift}$ and $\underline{\mathrm{e}}_k^{drag}$ are obtained from the normal and tangential vector of the k-th element and are given as

$$\underline{\mathrm{e}}^{lift} = \underline{\underline{\mathrm{T}}}_k \cdot \underline{\underline{\mathrm{T}}}_{k,\alpha}^{T} \cdot \underline{\underline{\mathrm{T}}}_k^{T} \cdot \underline{\mathrm{n}}_k \tag{14}$$

$$\underline{\mathrm{e}}^{drag} = \underline{\underline{\mathrm{T}}}_k \cdot \underline{\underline{\mathrm{T}}}_{k,\alpha}^{T} \cdot \underline{\underline{\mathrm{T}}}_k^{T} \cdot \underline{\tau}_k \tag{15}$$

where

$$\underline{\mathrm{q}}_k = \underline{\tau}_k \times \underline{\mathrm{n}}_k \tag{16}$$

$$\underline{\underline{\mathrm{T}}}_k = \left(\underline{\tau}_k \; \underline{\mathrm{n}}_k \; \underline{\mathrm{q}}_k\right) \tag{17}$$

$$\underline{\underline{\mathrm{T}}}_k = \begin{pmatrix} \cos\alpha_k & \sin\alpha_k & 0 \\ -\sin\alpha_k & \cos\alpha_k & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{18}$$

and α_k is the angle of attack relative to the freestream direction computed as

$$\alpha_k = \arctan \frac{\underline{\mathrm{V}}_k \cdot \underline{\mathrm{n}}_k}{\underline{\mathrm{V}}_k \cdot \underline{\tau}_k} \tag{19}$$

where the variables $\underline{\mathrm{n}}_k$ and $\underline{\tau}_k$ are the normal and tangential vectors of the k-th element and $V_k = (u_\infty - \underline{\mathrm{v}})_k$. The tangential vector is computed using the partial derivative of the surface parametrization of the vortex sheet at the point $\underline{\mathrm{p}}_c^k$ in the chordwise direction (Equation (20)).

$$\tau_k = \frac{\partial \underline{\mathrm{p}}_c^k / \partial\eta}{\left|\partial \underline{\mathrm{p}}_c^k / \partial\eta\right|} \tag{20}$$

The lift and drag contributions are computed as

$$\Delta\underline{\mathrm{L}}_k = \rho \cdot b_k \cdot \left[\underline{\mathrm{V}}_k \left(\Gamma_k - \Gamma_l\right) + c_k \cdot \frac{\partial}{\partial t}\left(\frac{\Gamma_k + \Gamma_l}{2}\right)\right] \cdot \cos\alpha_k \tag{21}$$

$$\Delta\underline{\mathrm{D}}_k = \rho \cdot b_k \cdot \left[\left(\underline{\mathrm{u}}_k^{b,b} + \underline{\mathrm{u}}_k^{b,f}\right)\left(\Gamma_k - \Gamma_l\right) + c_k \cdot \frac{\partial}{\partial t}\left(\frac{\Gamma_k + \Gamma_l}{2}\right)\right] \cdot \sin\alpha_k \tag{22}$$

where k refers to the k-th vortex element and l refers to the vortex element upstream of the l-th vortex element. In case the k-th vortex element is located at the leading edge, the contributions are computed using the following expressions:

$$\Delta \underline{\mathrm{L}}_k = \rho \cdot b_k \cdot \left[\underline{\mathrm{V}}_k \left(\Gamma_k \right) + c_k \cdot \frac{\partial}{\partial t} \left(\frac{\Gamma_k}{2} \right) \right] \cdot \cos \alpha_k \tag{23}$$

$$\Delta \underline{\mathrm{D}}_k = \rho \cdot b_k \cdot \left[\left(\underline{\mathrm{u}}_k^{b,b} + \underline{\mathrm{u}}_k^{b,f} \right) \left(\Gamma_k \right) + c_k \cdot \frac{\partial}{\partial t} \left(\frac{\Gamma_k}{2} \right) \right] \cdot \sin \alpha_k \tag{24}$$

where ρ is the density of the fluid, b_k is the span and c_k is the chord of the k-th element. The term Γ is the circulation of the respective vortex element. The component of the force F in Equation (13) parallel to the x-axis is the drag while the component parallel to the z-axis is the lift.

3. Computational Tool Description

While fast runtime is usually the target of scientific software projects, we recognize that a simple user-interface is also an important aspect of a framework's usage. An efficient framework which is complex to understand and use will not reduce the solution time for an engineer, despite the fact that the resulting code executes quickly. However, languages which are easy to use are frequently orders of magnitude slower in terms of performance and simulation time. Neither situation is ideal. The goal of PyFly is to provide a friendly framework for aerodynamic simulations based on the UVLM which is also computationally efficient. We achieve this by using a mixed-language programming paradigm. We employ Python [26] for high-level management of grid objects and processing the aerodynamic quantities while Fortran is used for the computationally-demanding kernels which must run efficiently.

3.1. Enhanced PyFly

PyFly models the aerodynamics of lifting body problems. The framework implements UVLM and seeks to predict dominant aerodynamic effects at lower computational costs than fully resolved Navier-Stokes or Euler descriptions. As described in the previous section, to compute the contribution of each vortex element on a point sitting in the body surface, we use an analytical expression of the solution of the Biot-Savart equation. The solution expresses the influence of a straight segment of constant vorticity on a point. In one implementation, PyFly uses this analytical expression to compute the influence of the l-th segment of the i-th vortex element at the j-th point where the no penetration condition is enforced (y_j), thus we have

$$A_{ij}^{l} = \frac{1}{4\pi |\underline{\mathrm{h}}|} \left(\cos(\theta_{l,1}) + \cos(\theta_{l,2}) \right) \underline{\mathrm{e}} \tag{25}$$

Figure 6 illustrates the parameters used in Equation (25) to compute the influence of a straight segment of constant vorticity on a point.

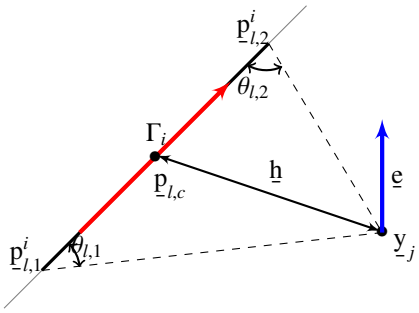


Figure 6: Influence of a straight segment of constant vorticity on a point.

PyFly is expected to handle a wide range of aerodynamic problems. However, as the domain grows, the number of pairwise interactions to be accounted for grows significantly. This leads to a significant increment in the computational expense for forming the algebraic system and inverting the resulting dense matrix. In an attempt to overcome this

potential computational burden, we decompose the domain into two regions and compute the contributions of the vortex segments in each part with two different mechanisms. In the proximal part of the domain to the target point, we use the analytical expression given in Equation (25) to compute the corresponding contribution. In the distal part of the domain, away from the point, we use a pointwise approximation to compute the remaining contribution. The process is recursively repeated until the contribution of all the vortex segments is evaluated on each target point. The partitioning of the domain allows us to develop a fast method to approximate the interactions and to solve the system system, which is schematically shown in Figure 7 and resembles multipole techniques.

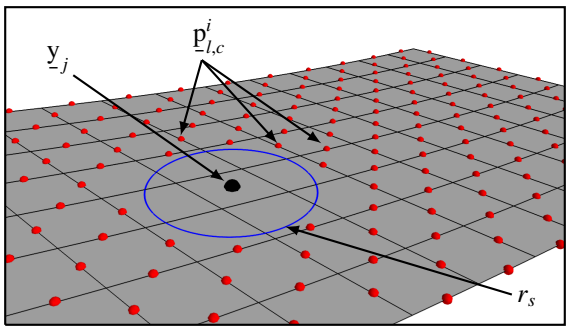


Figure 7: Enhanced implementation sketch based on mixed analytical/numerical computational approach to evaluate the impact of the vortex sheets on a target point.

The l-th vortex segment of the i-th vortex element is considered proximal to the target point if

$$\left|\underline{y}_j - \underline{p}^i_{l,c}\right| \leq r_s \tag{26}$$

where $\underline{y}_j$ is the target point, $\underline{p}^i_{l,c} = 0.5\left(\underline{p}^i_{l,1} + \underline{p}^i_{l,2}\right)$ and r_s is the close-to-target condition parameter that is computed using

$$r_s = N \cdot \Lambda_j \tag{27}$$

where the coefficient N is an integer number used to compute the value of the close-to-target condition in relation to the characteristic length Λ_j of the j-th vortex element. The diameter of a circle with the same area of the j-th vortex element is used as the characteristic length and is given by

$$\Lambda_j = 2 \cdot \sqrt{\frac{A^{elem}_j}{\pi}} \tag{28}$$

where A^{elem}_j refers to the area of the j-th vortex element.

The pointwise approximation is used when the vortex segment is further than r_s from the target point. The contribution is computed by the following expression

$$A^l_{ij} = \frac{1}{4\pi}\frac{\underline{r}_0 \times \underline{h}}{|\underline{h}|^3} \tag{29}$$

where $\underline{h} = \underline{p}_{l,c} - \underline{y}_j$ and $\underline{r}_0 = \underline{p}^i_{l,1} - \underline{p}^i_{l,2}$ which contribution corresponds to the analytical contribution of the vortex points.

The influence of the i-th vortex element at the point y_j is computed as the summation of the contributions of the four segments composing the vortex element.

$$A_{ij} = \sum_{l=1}^{4} A^l_{ij} \tag{30}$$

The pointwise approximation approach accelerates the numerical computation of the UVLM solution. In the next section, we demonstrate through a set of numerical examples the capability of this approach to speed up the aerodynamic simulations while maintaining good accuracy level.

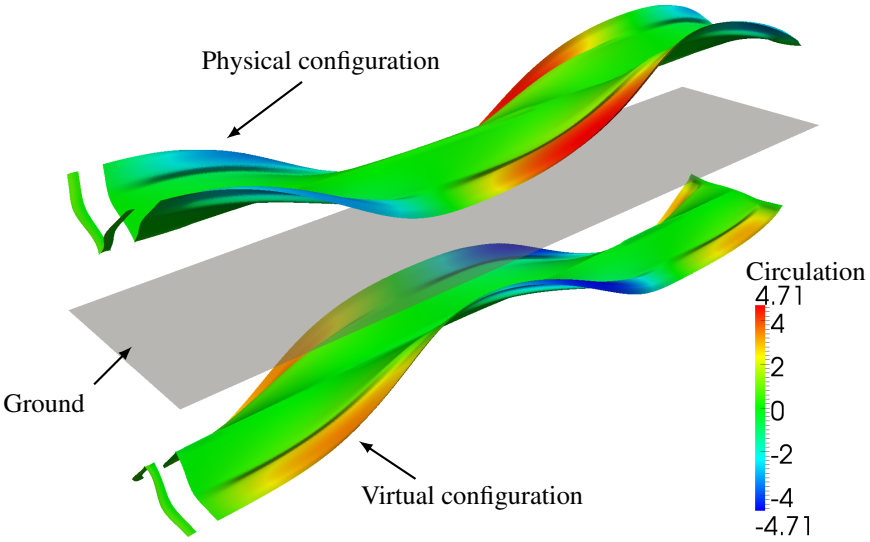


Figure 8: PyFly simulation of the ground effect: physical and virtual wings and wakes.

3.2. Additional features

Simulation of multi-body interactions. Our computational tool has also the capability to simulate the interactions between multiple bodies and their corresponding wakes. The modifications required to model these interactions are introduced to the matrix system obtained when imposing the no-penetration condition as follows:

$$\begin{pmatrix} \underline{\underline{A}}^{b-b}_{11} & \cdots & \underline{\underline{A}}^{b-b}_{1N} \\ \vdots & \ddots & \vdots \\ \underline{\underline{A}}^{b-b}_{N1} & \cdots & \underline{\underline{A}}^{b-b}_{NN} \end{pmatrix} \begin{pmatrix} \underline{\Gamma}^{b}_{1} \\ \vdots \\ \underline{\Gamma}^{b}_{N} \end{pmatrix} = - \begin{pmatrix} \underline{\underline{A}}^{wa-b}_{11} & \cdots & \underline{\underline{A}}^{wa-b}_{1N} \\ \vdots & \ddots & \vdots \\ \underline{\underline{A}}^{wa-b}_{N1} & \cdots & \underline{\underline{A}}^{wa-b}_{NN} \end{pmatrix} \begin{pmatrix} \underline{\Gamma}^{wa}_{1} \\ \vdots \\ \underline{\Gamma}^{wa}_{N} \end{pmatrix} + \underline{V}_{n} \tag{31}$$

where $\underline{\Gamma}^{b}_{i}$ and $\underline{\Gamma}^{wa}_{i}$ denote the vector collecting the circulation of the vortex elements of body i and wake i, respectively. For the influence matrices, $\underline{\underline{A}}^{b-b}_{ij}$ refers to the effect of body i on body j and $\underline{\underline{A}}^{wa-wi}_{ij}$ refers to the effect of wake i on body j. $\underline{V}_{n}$ collects the normal component of the velocity at each collocation point of all bodies. Furthermore, PyFly accounts for the interactions between the wakes when convecting the vortex segments from the trailing-edges of the bodies. In the next section, we will analyze two aerodynamic systems involving the interactions between multiple bodies.

Simulation of enclosure effects. PyFly simulates the enclosure effects (the proximity to obstacles such as ground and walls). To account for the ground effect using UVLM, an effective ground plane is introduced into the model using the method of images [25, 27, 28]. In this technique, the vortex lattices representing the lifting surface and wake are mirrored about the ground plane. In the simulation, a mirror-image of each vortex segment (with inverted circulation) is placed under the ground plane. As an example, we plot in Figure 8 the actual and virtual flapping wings and wakes. The color levels denote the vorticity circulation strength. As mentioned in the UVLM description, the vorticity in the wake is shed from the wing at an earlier time. The vortex distribution of the virtual lattices modeling the image lifting surface and wake create a secondary induced flowfield. The velocity at any point in space is obtained by summation over all vortex segments. At the ground plane, the vertical component of the velocities induced by the actual and virtual wings cancels and then the no-penetration condition is satisfied. More details on the implementation of the enclosure effects are given in [29].

B-spline representation of body shapes. PyFly describes the geometry of the simulated aerodynamic systems using B-spline parameterizations, which are the standard technology for describing geometries in computer-aided design (CAD) software. This shape representation facilitates the design optimization process by enabling mesh generation

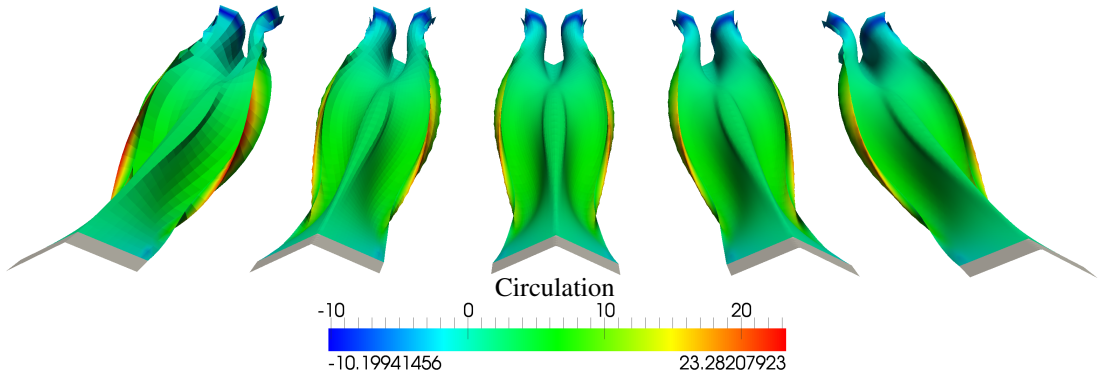


Figure 9: A flapping wing and its wake increasingly refined (left to right) using subdivision surfaces.

directly from the CAD model. The associated basis functions can be used to smoothly discretize aerodynamic system shapes with few degrees of freedom. The reader is referred to [30–33] for more details on B-splines.

Enhanced visualization. Due to the approximate nature of the UVLM method, computational grids tend to be coarse as a fully resolved grid is beyond the scope and intent of the method. As a consequence, the visualizations of the solutions are of low resolution while sharp and highly resolved snapshots are increasingly the norm in scientific publications, regardless of the accuracy of the underlying method.

To improve the asthetics of UVLM solutions, we use of subdivision surfaces [34] as a post-processing step. Subdivision surfaces are an industry standard in the field of computer animation and used by companies such as Pixar since the late 1990's. Subdivision surfaces represents a smooth surface by a coarse piecewise-linear polygonal mesh. A refined surface is then computed by applying a recursive process which divides each surface into smaller surfaces. Because the grids we use in our implementation of UVLM are structured, we use the Catmull-Clark scheme [35] which is a generalization of bi-cubic B-splines to arbitrary topologies.

In Figure 9, we show the effect of subdivision on the wake of a flapping wing. On the far left we show the raw output of the UVLM simulation, colored by values of the vorticity circulation. For each figure to the right, the subdivision scheme is applied to the grid resulting in a smoother representation. The number of applications of the subdivision surface refinement algorithm can be controlled when using the UVLMGrid member function `Plot`. We emphasize that while this process does change the geometry and data on the grid, the refined grid is guaranteed to lie within the convex hull of the original, limiting the deviation. We only use this process for visualization. Aerodynamic quantities are always computed from the raw solution.

3.3. PyFly implemetation

This section gives an overview on how the present implementation of PyFly serves the needs of users and developers seeking to simulate moving bodies interacting among themselves and air under potential flow assumptions. We design PyFly to reduce the overall complexity of applying a numerical method to realistic problems. The implementation is efficient in terms of runtime. While UVLM is a reduced-order method and not intended for high-fidelity modeling which requires extensive computational resources, the method does lend itself to studies in design optimization or inverse problems which would be intractable for full fidelity models. In these cases, the forward model is frequently executed and is therefore critical that the simulation be fast and reliable.

Additionally, the implementation must also be simple to use and apply to a range of problems. We pose this design goal given the understanding that efficient software which are difficult to install and use, may end up requiring more user time. We observe this principle in the prevalent use of Matlab in academic communities. Despite that it produces software which can take a comparatively large amount of time to execute, the users prefer it due to the simple implementation to solve numerical problems.

There is a substantial effort in the academic community to develop and provide open-source, scientific software, particularly using the Python language [26, 36–39]. In PyFly, we provide the user interface using Python to make our framework easy to use, while the core functionality is in Fortran. This results in an efficient and yet simple software to use.

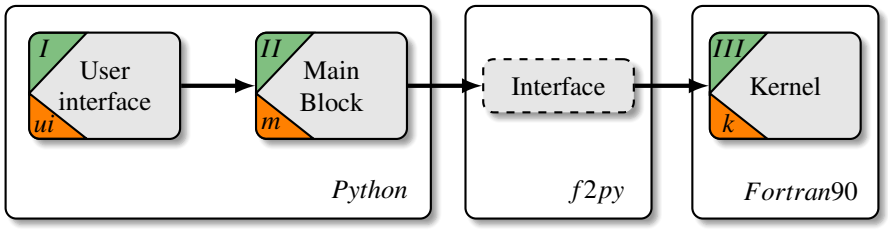


Figure 10: PyFly architecture. Interface between Python and FORTRAN 90 is accomplished using a tool called `f2py` [40].

PyFly consists of three main blocks as illustrated in Figure 10. The first block holds the user interface and corresponds to a Python file. This user interface file permits to initialize the problem by assigning the lifting body shape and motion as well as the fluid properties, the existence and location of walls (including ground, roof, or wall), and the method of solution (either using the entire geometry or using a mirror technique). The mirror technique reduces the domain problem depending on the symmetry of the geometry. For instance, a bird represented by two flapping wings can be reduced to a single flapping wing assuming that the shape and the motion of the two wings are symmetric. Moreover, by using the mirror technique the memory usage and cost to solve a problem may be reduced. The second block consists in the main block of PyFly framework. This part is written in Python and contains the subroutine used to build the matrix system corresponding to (10) in order to compute the circulation on the body surface elements along with the subroutines used to solve (11) and (12) governing the update mechanism of the wake location.

The third block is the kernel or core functionality of PyFly. The kernel computes the followings:

- The contribution of the vortex elements to the normal velocities on the body surface elements (matrix $\underline{\underline{A}}_{ij}^{b,b} \cdot \underline{n}_j$ and vector $\underline{u}_j^{b,f} \cdot \underline{n}_j$ in Equation (10))
- The motion velocity of the wake (matrices $\underline{\underline{A}}_{ij}^{f,f} \cdot \Gamma_i^f$ and $\underline{\underline{A}}_{ij}^{f,b} \cdot \Gamma_i^b$ in Equation (11))
- The aerodynamic loads applied on the lifting body (lift, drag, and pressure) and the generated power.

In Listing 1, we illustrate the structure of the uvlm.f90 file that corresponds to the kernel block. This FORTRAN file comprises two modules. The first module includes functions to compute the basic vectorial operations and the solution of (9); that is, the velocity induced for one segment of a vortex ring. The second module consists of the five following subroutines:

- SourceOnTarget: Computes the contribution to the velocity normal of all body elements induced by all vortex rings (body to body contribution).
- NormalVelocityAtTarget: Computes the normal velocity at various body elements center points induced by the designed vortex rings (wake to body contribution).
- VelocityAtPoints: Computes the velocity at various points induced by the designed vortex rings (wake/body to wake contribution).
- ComputeDownwash: Computes induced velocity on the body elements due to a wake vortex ring (information needed to compute the Aerodynamic forces).
- AerodynamicData: Computes the aerodynamic quantities (lift, drag, pressure and power).

Figure 10 shows the interface between Python and FORTRAN is accomplished using the tool `f2py` [40].

Listing 1: Kernel block (uvlm.f90)

```
! Module #1: Includes the vectorial products, the computation of cell data and the
! computation of Biot Savart law to obtain the relation between source and target.
module base
contains
! Function: Compute the cross product between two vectors, v1 X v2 = v3.
function CrossProduct(v1,v2) result (v3)
```

Computes → `v1×v2=v3.`

```
end function CrossProduct
! Function: Compute the dot product between two vectors, v1 . v2 = v3.
function DotProduct(v1,v2) result (dot)
```

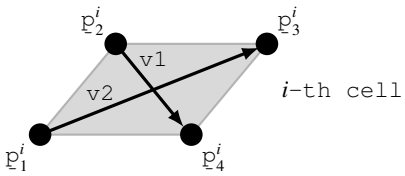
Computes → `v1·v2=dot.`

```
end function DotProduct
! Function: Compute the normals and the areas for each cell.
subroutine ComputeCellData(...)
```

$\underline{n}_j = \texttt{CrossProduct(v1,v2)}.$

$\texttt{area}_j = \sqrt{\texttt{DotProduct}\left(\underline{n}_j, \underline{n}_j\right)}.$

$\underline{n}_j = \underline{n}_j / \texttt{area}_j.$



i−th cell

```
end subroutine ComputeCellData
! Function: Compute the relation between the source and the target. The implementation
!           use the analytical formula of the Biot Savart law equation to compute the
!           relation between the source and the target when the distance between the
!           cells is smaller than the influence radius. When the distance between
!           cells is larger than the influence radius, the relation between the
!           source and the target is computed using a point-wise method.
            (This is explained in enhanced PyFly section)
function BSL(...) result (out)
```

Computes `out`

$$\begin{cases} A_{ij}^l = \dfrac{1}{4\pi|\underline{h}|}\left(\cos(\theta_{l,1}) + \cos(\theta_{l,2})\right)\underline{e} & \forall \left|\underline{y}_j - \underline{p}_{l,c}^i\right| \le r_s \\ A_{ij}^l = \dfrac{1}{4\pi}\dfrac{\underline{r}_0 \times \underline{h}}{|\underline{h}|^3} & \forall \left|\underline{y}_j - \underline{p}_{l,c}^i\right| > r_s \end{cases}$$

Contribution to the velocity at the j−th body element induced for one segment of the i−th vortex ring.

```
end function BSL
!
end module base
! ----------
! Module #2: Includes the functions used to compute the interaction between source
!            and target.
module influence
contains
!
! Subroutine: Computes the contribution to the velocity normal to all body elements
!             induced by all vortex rings (body to body contribution).
subroutine SourceOnTarget(...)
```

Computes matrix A = $\underline{\underline{A}}_{ij}^{b,b} \cdot \underline{n}_j$ (Equation 10). using Function BSL k

```
end subroutine SourceOnTarget
!
```

```
! Subroutine: Computes the normal velocity at various given body elements center points
!             induced by the designed vortex rings (wake to body contribution).
subroutine NormalVelocityAtTarget(...)
```

Computes vector b = $\underline{u}_j^{b,f} \cdot \underline{n}_j = \left(\underline{\underline{A}}_{ij}^{b,f} \underline{\Gamma}_i^f\right) \cdot \underline{n}_j$ (Equation 10). using Function BSL k

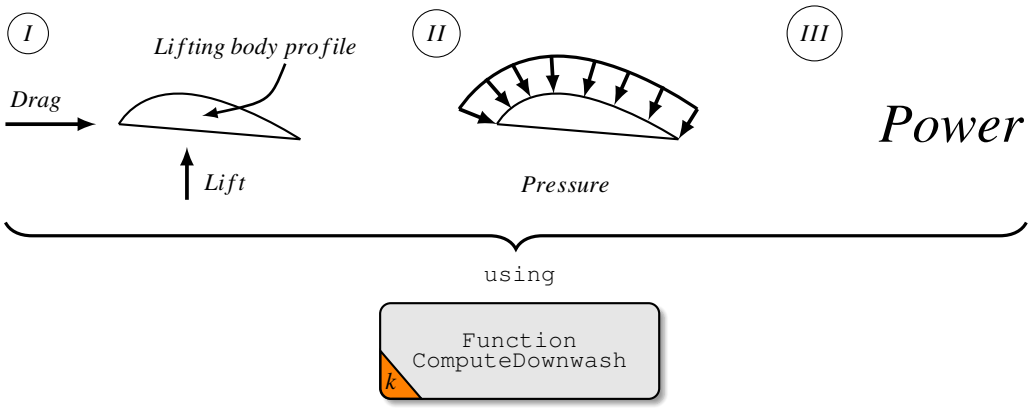
```
end subroutine NormalVelocityAtTarget
!
! Subroutine: Computes the velocity at various given points induced by the designed
!             vortex rings (wake/body to wake contribution).
subroutine VelocityAtPoints(...)
```

Computes displacement velocity vel = $\underline{u}_j^f = \underline{\underline{A}}_{ij}^{f,f} \cdot \Gamma_i^f + \underline{\underline{A}}_{ij}^{f,b} \cdot \Gamma_i^b$ (Equation 11). using Function BSL k

```
end subroutine VelocityAtPoints
!
! Function: Computes induced velocity on a body elements due to a wake vortex ring
!           (information needed to compute the Aerodynamic forces).
function ComputeDownwash(...) result (W_ind)
```

Computes vector velocity W_ind = $\underline{\underline{A}}_{ij}^{b,f} \underline{\Gamma}_i^f$ (Equation 10). using Function BSL k

```
end function ComputeDownwash
!
! Subroutine: Computes the aerodynamic forces (lift, drag, pressure and power).
subroutine AerodynamicData(...)
```

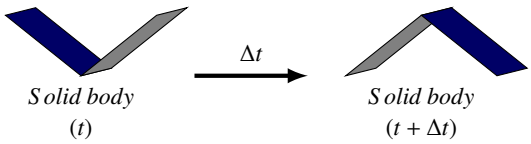



```
end subroutine AerodynamicData
!
end module influence
```

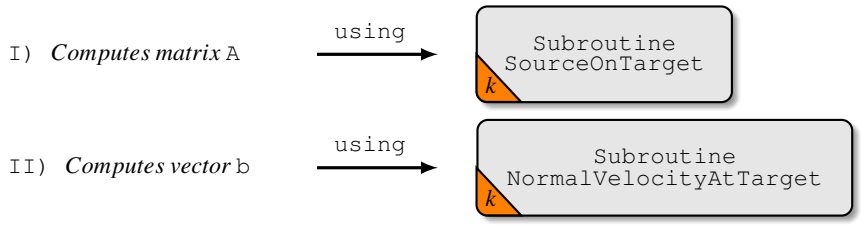
Listing 2 illustrates the structure of the grid.py file that corresponds to the PyFly main block. This Python file consists of a Pyhton class which includes six subroutines. The first subroutine initializes the Python class and create the required variables (allocating space) to build the system problem and solve it. The following subroutines are used to create the matrix system, update the lifting body location, convect the wake and plot the results (the circulation over the lifting body).

Listing 2: Main block (grid.py)

```
# Calling libraries
 Here are written the commands to call Python libraries
 as well as PyFly kernel.
####################
# Main Code
__all__ = ['UVGrid']
# Class: Contains all the information referred to the lifting body
#        and the wake.
class UVGrid():
    def __init__(...):

        Initialization of variables


# Function: Updating of the coordinates of the body's and ring
#           circulation s grids based on the movement the body presents.
    def Update(...):
```




```
        return
# Function: Computes the influence of the wing (A) and the wake (b) in the
#           wing circulation. Constructs the system Ax = b without considering
#           the absolute velocity of the lifting body.
    def Influence(...):
```

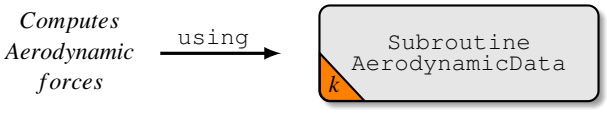



```
        return A,b
# Function: Computes the absolute velocity of the lifting body..
#           The absolute velocity is added to (b).
    def RelativeMotion(...):
```

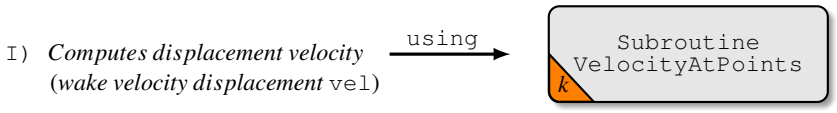
Computes vector `vel` $= \underline{V}_j = \left((\underline{u}_\infty - \underline{v})_j\right) \cdot \underline{n}_j$

```
        return -vel   → b=b-vel
```

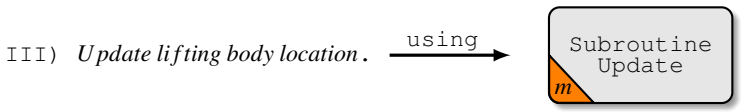
```
# Function: Updates of the wing and the wake after computing circulation over
#           the lifting body. Also computes the aerodynamic forces.
    @staticmethod
    def ConvectWake(...):
            # --- Aerodynamic Forces --- #
```




```
            # ------------------------- #
```



II) *Computes wake new location* $\underline{x}^f(t+\Delta t) = \underline{x}^f(t) + \texttt{vel} \cdot \Delta t$




```
        return
# Function: This function saves the information related with the position and
#           the circulation.
    def Plot(...):
```

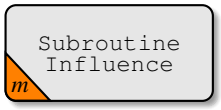
Plot results in a vtk file

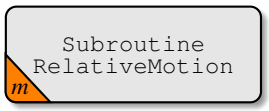
We employ a Python file as the user interface block. In this file, we call the essential libraries and we introduce the information related with the aerodynamic problem of interest. Listing 3 shows the Python file for computing the dynamics of two wings moving through one flap cycle. The numerical results of this case will be presented in the next section.

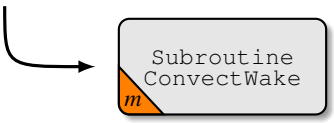
Listing 3: User interface block

```
# Calling libraries
```
Here are written the commands to call Python libraries
as well as PyFly main block and the python functions used to
build the solid body and describe solid body motion and fluid flow.

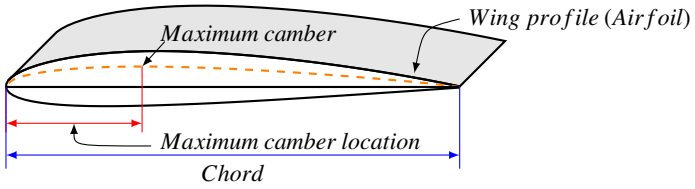
```
#
# ====================================================================#
# First step computation
# Initial coordinates of the circulations rings
def Update(pos,rot,t,t_period):
    grid.Update(pos[0,:],rot[0],t,t_period)
    return 0
# ====================================================================#
# - Setup the linear system
# Computes the influence of the wing (A) and the wake (b) in the wing circulation.
def Setup(grid,wall,t,dt,rs,fsv):
    A,b = grid.Influence(grid,wall,t,dt,rs)
    b += grid.RelativeMotion(t,dt,fsv=fsv)
    return A,b
# ====================================================================#

# - Linear Solver
# Computes the circulation on the solid body (the circulation in every cell).
def Solve(A,b):
    c = np.linalg.solve(A,-b)
    return c
# ====================================================================#
# - Convect the wake
# Computes the aerodynamic forces and update the body and
# wake coordinates and the wake circulation.
def Convect(step,grid,pos,wall,upwake,t,t_period,dt,rs,vref,fsv):
    UVGrid.ConvectWake(step,[grid],pos,rot,wall,upwake,t,t_period,dt,rs,vref,fsv=fsv)
    return 0

################## NOT TOUCH BEFORE THIS LINE ##################
#==============================================================#
##################        MAIN PROGRAM        ##################
# ======
# Flight of a single bird problem using Flapping wings.     ← Problem name
# ======
# - Air properties
# Density
```



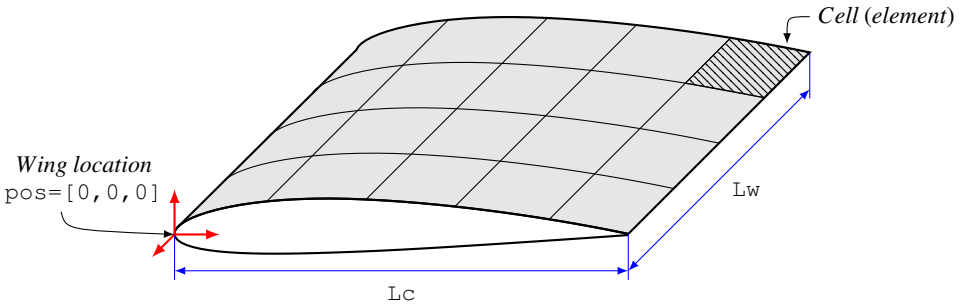




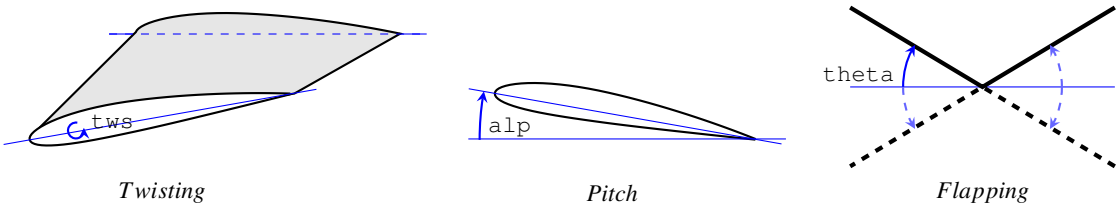
```
rho    = 1.225
# - Flow velocity      ← freestream vector [x, y, z]
fsv   = [0,0,0]
# ======
# - Airfoil Data NACA MPXX
M = 8.0       ← Maximum camber
P = 3.0       ← Location of maximum camber
```




```
# ======
# - Wing Data
# Cord and wingspan lengths
Lc,Lw = 1.,4.
# Cord and wingspan discretization
nc,nw = 6,10
# Cord and wingspan polynomial order
pc,pw = 1,1
# Wing geometry
wing = WING(M,P,Lc,Lw,nc,nw,pc,pw,0,0)      ← Function that build the wing mesh
# Wing position
pos = np.zeros((1,3))
pos[0,:] = [0.,0.,0.]
```




```
# Wake data
mxwake = 51    # Wake length
upwake = 10    # Maximum wake to update
# Walls data
wall = np.zeros((1,2))
wall[0,:] = [0,0.]
# Solid body moves
parm = np.zeros(5)
parm[0] = Lw     #       --> wingspan
parm[1] = 15.0   #theta --> flapping angle
parm[2] =  4.0   #tws   --> Twisting angle
parm[3] =  0.0   #alp   --> Pitch
parm[4] = 10.0   #       --> Velocity of solid body
```



Twisting *Pitch* *Flapping*

```
# ======
# - Transient Data
N_period = 40
Omega = 2.
t_period = (2.*np.pi/Omega)
dt = t_period/(N_period)
N_steps = 50
np.set_printoptions(precision=15)
# ====================================================================#
# - Python class initialization
grid = UVGrid(wing,trans,gyro,twist,flap,rho,parm,symaxis=2,symcord=pos[0,1]
                                                              ,maxwake=mxwake)
```

Solid body *Translation* *Rotation* *Morphing (Twisting)* *Transformation (Flapping & pitch)*

```
# ======
# - Initization of the solution loop
t = 0.
# ======
# - influence radius (rs)
# Maximum distance between cells where Biot savart analytical equation is used
# to compute the interaction between cells. Beyond that distance a point-wise
# approximation is used.
rs = 2*(user data)*np.sqrt((1.*Lc/nc)*(1.*Lw/nw)/np.pi)
# ===================
# - Initial coordinates of the circulations rings
Update(pos,rot,t,t_period)
# - Loop     ← Time loop
for step in range(N_steps):
    # - Setup the linear system
    A,b = Setup(grid,wall,t,dt,rs,fsv)
    # - Linear Solver
    c = Solve(A,b)
    # - Save computed circulation (c)
    grid.circ = c.reshape(grid.circ.shape)
    # - Convect the wake
    Convect(step,grid,pos,wall,upwake,t,t_period,dt,rs,vref,fsv)
    # - Time update
    t += dt
    # - Plot results at time t
    grid.Plot("Flap_wing",step)
# ====================================================================#
```

We implement the methodology presented in [41] to select properly the mesh and time-step sizes to achieve invariant UVLM simulation results under mesh refinement. The simulation results revealed that when selecting improperly the number of time steps per cycle, the UVLM solution fails to converge as the aerodynamic mesh is refined. As such, to remove mesh sensitivity from UVLM simulations, Ghommem et al. [41] found that one needs to first determine the appropriate time step in order to

obtain convergence of UVLM simulations when refining the aerodynamic mesh. Listing 4 shows a Python file that includes the computation of the number of time steps per flapping cycle and the number of chordwise and spanwise elements to produce mesh independent results. This listing is a simplified version of Listing 3 where the lines in blue corresponds to the statements required to perform the mesh and time refinements.

Listing 4: Aerodynamic mesh and time refinement

```
    ⋮
################## NOT TOUCH BEFORE THIS LINE ##################
#==============================================================#
###################        MAIN PROGRAM        ###################
# ======
# Flight of a single bird problem using Flapping wings.
# ======
    ⋮
# ============================================================#
# Cord and wingspan initial discretization
nc,nw = 6,10     ← Initial mesh discretization
# Time initial discretization
N_period = 40     ← Initial mesh discretization
# Level of refinement
reft = userdata     ← in time
refc = userdata     ← in cord direction
refw = userdata     ← in spanwise direction
# Tolerance of the error
epsilon = 1E-6
# - Mesh and time refinement loop
refcond = True     ← Refinement condition
refcase = 0     ← Refinement case (see below)
while (refcond):
    # - Refinement cases: (The approach starts with the refinement of the time step)
    # refcase==0     ← Initialization case
    if (refcase==1): N_period += reft     ← Refinement in time
    if (refcase==2): nc += refc     ← Refinement of the mesh in cord direction
    if (refcase==3): nw += refw     ← Refinement of the mesh in spanwise direction
    # - Wing geometry
    wing = WING(M,P,Lc,Lw,nc,nw,pc,pw,0,0)
    # - Transient Data
    Omega = 2.
    t_period = (2.*np.pi/Omega)
    dt = t_period/(N_period)
    N_steps = 2*N_period     ← Study using two full flapping periods
    # ============================================================#
    # - Python class initialization
        grid = UVGrid(wing,trans,gyro,twist,flap,rho,parm,symaxis=2,symcord=pos[0,1]
                                                                    ,maxwake=mxwake)
    # ======
    # - Initizalization of the solution loop
    t = 0.
    atime = []
    atime.append(t)
    # ======
    # - influence radius (rs)
    rs = 2*(user data)*np.sqrt((1.*Lc/nc)*(1.*Lw/nw)/np.pi)
    # ==================
    # - Initial coordinates of the circulations rings
```

```
    Update(pos,rot,t,t_period)
    # - Loop
    for step in range(N_steps+1):
        # - Setup the linear system
        A,b = Setup(grid,wall,t,dt,rs,fsv)
        # - Linear Solver
        c = Solve(A,b)
        # - Save computed circulation (c)
        grid.circ = c.reshape(grid.circ.shape)
        # - Convect the wake
        Convect(step,grid,pos,wall,upwake,t,t_period,dt,rs,vref,fsv)
        # - Time update
        t += dt
        atime.append(t/t_period)
    # - Compute stop refinement criteria
    if (refcase==0):
        # Initial step (save Lift and Drag coefficients)
        Clift0 = interp1d(atime[:-1],grid.Clift,kind='linear')
        Cdrag0 = interp1d(atime[:-1],grid.Cdrag,kind='linear')
        refcase = 1
    else:
        # Save current step
        Clift = interp1d(atime[:-1],grid.Clift,kind='linear')
        Cdrag = interp1d(atime[:-1],grid.Cdrag,kind='linear')
        # Compute error
        x = np.linspace(0,2,len(atime[:-1]))      ← Study using two full flapping periods
        elift = np.sqrt(np.mean(np.square(Clift0(x[1:-1])-Clift(x[1:-1]))))
        edrag = np.sqrt(np.mean(np.square(Clift0(x[1:-1])-Clift(x[1:-1]))))
        # Check stop criteria
        if (elift ≤ epsilon and edrag ≤ epsilon):
            # If stop criteria is satisfied during time refinement, the method
            # continues with the refinement of the mesh in the cord direction.
            if (refcase==1):
                refcase = 2
                continue
            # If stop criteria is satisfied during mesh refinement in the chord
            # direction, the method continues with the refinement of the mesh in
            # the spanwise direction.
            if (refcase==2):
                refcase = 3
                continue
            # If stop criteria is satisfied during mesh refinement in the spanwise
            # direction, the refinement method stops.
            if (refcase==3):
                refcond = False
                print "Time discretization:  N_period=>",N_period
                print "Mesh discretization:  [nc][nw]=>[",nc,"] [",nw,"]"
        else:
            # If the stop criteria is not satisfied the current Lift and Drag
            # coefficient are saved to compute the Frobenius norm error during
            # the following refinement.
            Clift0 = Clift
            Cdrag0 = Cdrag
# ======================================================================#
```

3.4. Summary of PyFly capabilities

Our numerical tool combines the use compiled language (Fortran) and user-interactive problem solving environment (Python) presents several advantages (but not restricted to):

- Easy-to use and open source tool. PyFly is based on open source scripting/compiled languages and users can employ a potential flow solver that is integrated into a user-friendly and accessible computational framework.
- High flexibility and high performance. PyFly is designed to cover a broad range of applications (e.g., flapping wings, formation flight, rotating blades, suspension bridges, wind turbines). The code interface uses Python to provide a flexible working environment for users. The code back-end uses Fortran, which is a low-level compiling language, to ensure reasonable simulation time and numerical approximations of the analytical equations. Additionally, pointwise approximations of the distal interaction speed up performance of the simulator.
- Efficient and reliable. The UVLM has been widely verified and used for design and performance assessment of flying systems, rotating blades, offshore structures, and suspension bridge decks [5, 6, 8, 10–23, 29, 41, 42]. Several studies demonstrated the capability of the UVLM to produce accurate results for attached flow cases and even remain trend-relevant in the presence of flow separation when comparing them against high-fidelity computational fluid dynamics simulations of Navier-Stokes system [8].
- Potential extensibility. PyFly can be coupled with structural models to account for the fluid-structure interactions (FSI) and body deformation. The extension of PyFly to include FSI schemes is still under development. PyFly is currently made available as an open source software tool that provides a powerful and extensible platform for the numerical simulations of aeroelastic systems. PyFly also provides a solid foundation for being coupled to multidisciplinary optimization tools for design purposes.
- Use of standard and accessible libraries. This includes mainly py2f tool used to communicate between Python and Fortran scripts and Lapack Fortran library used to solve the linear systems with dense matrices assembled based on the UVLM formulation. The developed software along with all numerical test problems discussed in the next section is made accessible for all users [24].
- Available documentation detailing the theoretical background of the UVLM and the numerical implementation along with demo problems for illustration.

4. Numerical Test Problems and Computational Performance Analysis

In this section, we consider a number of test problems of varying complexity to demonstrate the capability of the computational framework to handle a broad range of applications. Aerodynamic simulations of flapping wings in forward flight and rotating blades of wind turbine are used to verify PyFly. The main purpose of the tests is to compare the performance and accuracy of the proposed numerical approximation against standard UVLM implementation. The numerical tests are computed on an intel core i7-4702MQ (2.20GHz) 8 cores PC with 16GB RAM. The computational performance analysis is carried out in serial; that is, only one core is used per computation.

4.1. Flapping/twisting rectangular wing

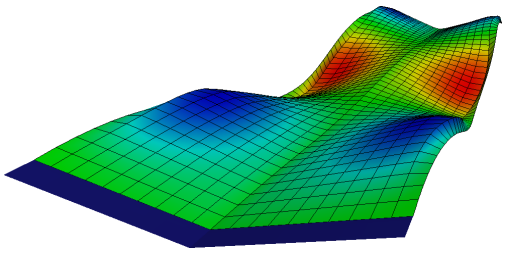
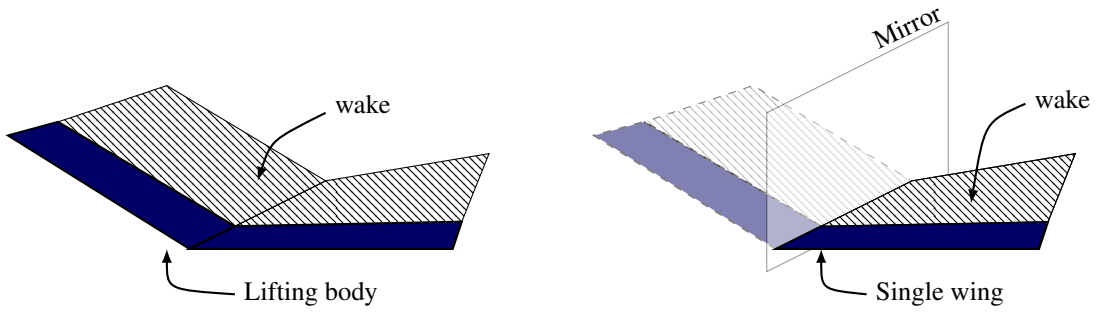


Figure 11: Flapping/twisting wing problem sketch.

The aerodynamic analysis of a flapping/twisting wing in forward flight is conducted first. A sketch of the problem showing the flapping wings along with the shed wake is presented in Figure 11. The wing has a rectangular shape with a root chord $c = 1\ cm$ and a wingspan $b = 8\ cm$. The transversal section of the wing consists of a NACA 0012 airfoil. The wing moves forward at a speed $\underline{v}$ and flaps with a maximum angle of θ_{max} and at a flapping frequency of ω_f. A linear deformation along the span of the

Parameter	
Flight velocity ($\underline{v}$ in m/s)	10
Flapping frequency (ω_f in Hz)	2
Reduce frequency ($k = \omega_f/2/\underline{u}_\infty$)	0.1
Flapping amplitude (θ_{max} in °)	15
Maximum twisting (β in °)	4
Twisting phase angle (Ψ_β in °)	90
Pitch (α in °)	0/4
Flapping period ($T = 2\pi/\omega$ in s)	π
Time step ($t = T/40$ in s)	$\pi/40$

Table 1: Flapping/twisting wing problem data.



(a) Full domain (b) Reduced domain

Figure 12: Full and reduced aerodynamic problem domain.

wing is introduced with a maximum twisting angle $\beta = 4°$ at the tip of the wing. The twisting is out of phase with the flapping by $\Psi_\beta = 90°$. The parameters used in the test case are presented in Table 1.

We reduce the problem size by assuming that the lifting body involves a symmetric behavior during the flapping flight. Figure 12 sketches the full and reduced problem domains. The reduced domain is discretized with six elements along the chordwise direction and ten elements in the direction of the wingspan (that corresponds to a single wing).

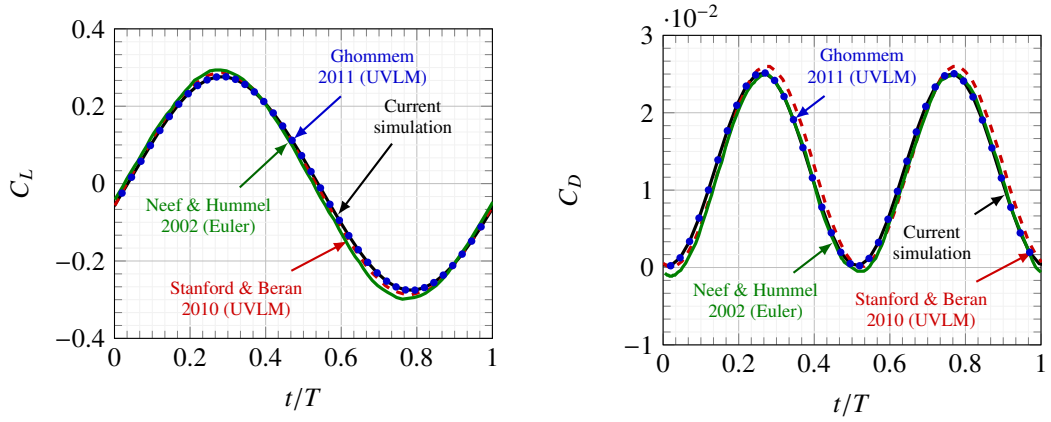
In Figure 13, we plot the lift and thrust coefficients for one flapping cycle. The simulations are performed for a pitch angle of 0^o and 4^o. The results obtained PyFly compare very well with those obtained from the Euler flow simulations preformed in [43] and previous studies using UVLM-based implementations [5, 6], demonstrating the capability of PyFly to predict accurately the unsteady aerodynamic loads generated from flapping wings interacting with air. The transient part of the simulations are removed since the approach requires some steps to stabilize before produce adequate results.

To quantify the performance of the enhanced PyFly, we estimate the cost and error to solve the problem for various values of the close-to-target parameter (influence radius r_s). We chose the case problem with an angle of attack (pitch) $\alpha = 4°$ to perform this analysis of performance. In each time step, PyFly builds an algebraic system, computes the bound circulation on the solid body elements (cells), updates the body position, transports the bound circulation into the free vortex and finds the new wake distribution. Figure 14 displays the time PyFly uses to perform all the operations. At the beginning, the computational time increases as the influence radius r_s grows. However, after r_s reaches a certain value ($r_s \approx 50$ times the characteristic length Λ of the solid elements), the simulation time becomes less sensitive to the influence radius growth and only a slight variation of the time it costs is observed.

The L^2 error norm of the circulation on the solid body is computed for all time steps in the simulations, and the maximum value among them is used as the estimate of the error. The circulation on the bound sheet is considered to conduct the error analysis as this circulation determines all the aerodynamics loads (as shown in Equations (23) and (24)) and the dynamics of the wake. The estimate of the error is computed using

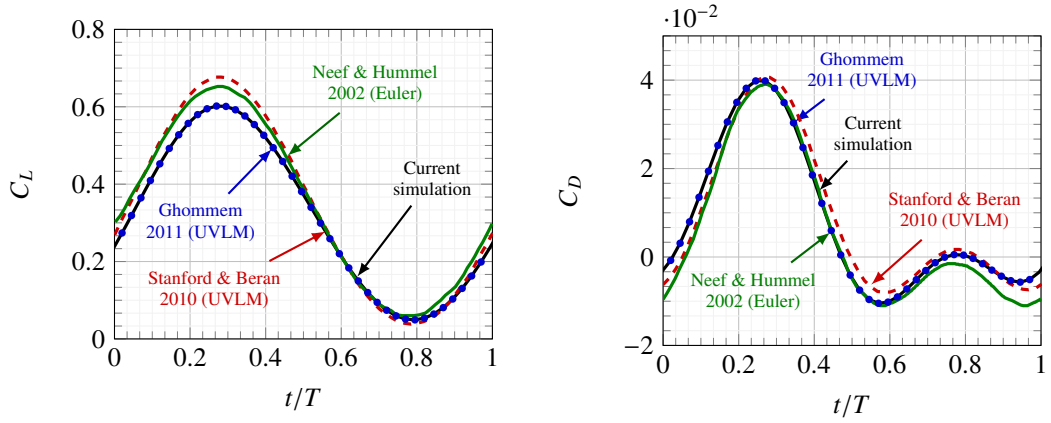
$$\|\bar{e}\|_{L^2} = \left\langle \left(\sum_{i=0}^{elements} |\Gamma^i_\infty - \Gamma^i_{r_s}|^2 \right)^{0.5} \; : \; 0 < rs < \infty \right\rangle^{max} \tag{32}$$

where Γ^i_∞ refers to the circulation of the i-th element on the bound sheet computed with the analytical method (traditional UVLM)



(a) Lift coefficient (Pitch $\alpha = 0°$)

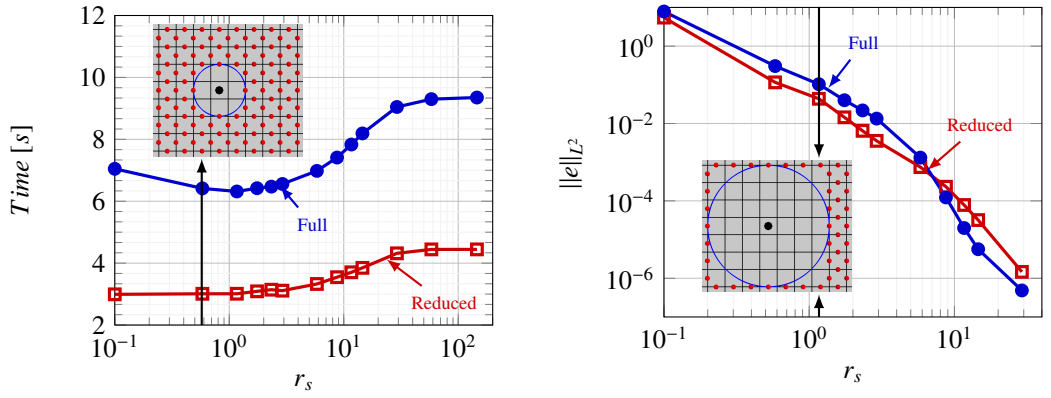
(b) Thrust coefficient (Pitch $\alpha = 0°$)



(c) Lift coefficient (Pitch $\alpha = 4°$)

(d) Thrust coefficient (Pitch $\alpha = 4°$)

Figure 13: Lift and Thrust (negative drag) coefficients: comparison against other flow solvers.



(a) Computational time.

(b) L^2 error norm of ciruclation γ.

Figure 14: Cost and Error estimates for the flapping/twisting wing problem.

and $\Gamma^i_{r_s}$ is the circulation computed with the enhanced PyFly and a close-to-target parameter r_s. Here $\langle\rangle^{max}$ refers to the maximum value of the error norm among all the time steps. In Figure 14(b), we show the error in the circulation for various values of r_s.

Contrary to the cost, the error reduces as the close-to-target parameter value is increased.

A relevant measure to analyze the performance of enhance PyFly is the time saving. Figure 15 illustrates the computational time saving relative to the numerical error when using various values r_s. The maximum saving in the computational time that enhanced PyFly yields when solving the current flapping/twisting wing problem is approximately 30% for both cases, using the full and the reduced domain. These values are not entirely practical since the error corresponding to those cases is high ($O(10^{-1})$). Reasonable savings in computational time are achieved when the parameter r_s is taken bigger than 20Λ. For instance, the error observed with $r_s = 30\Lambda$ is of order $O(10^{-3})$ while the saving reaches a maximum of 20%. If an error of order $O(10^{-6})$ is required in the solution, a saving in computational time is still observed (approximately 2% time saving).

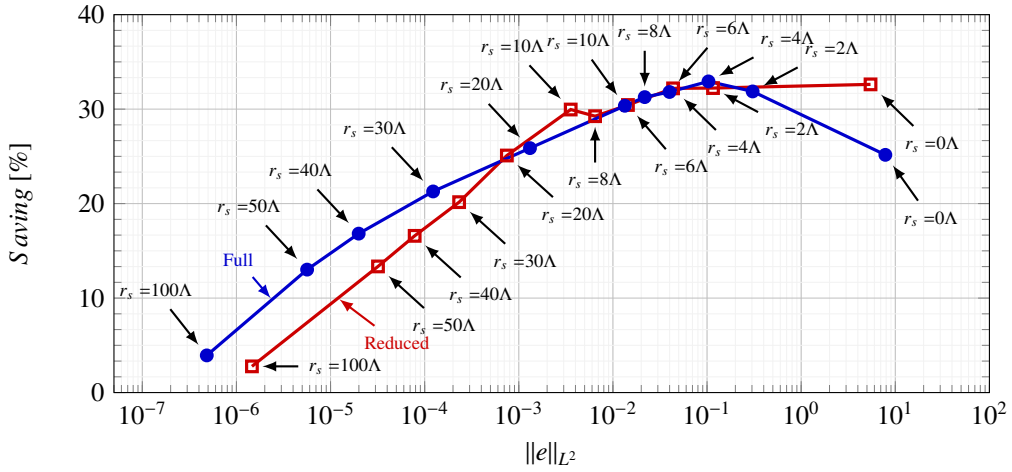


Figure 15: Plot of computational time vs error for the flapping/twisting wing problem.

The selection of the close-to-target parameter value depends on the solution accuracy and numerical performance requirements of the problem under study. For instance, in the study and analysis of birds' flight aiming at developing an understanding of their aerodynamic behaviour, one could reach a saving of 25% while considering an acceptable of order $O(10^{-3})$, however in engineering design problems where accuracy is a critical factor, such as the assessment of the aircraft stability, an error of order $O(10^{-6})$ should be targeted while a 2% in saving is still possible.

4.2. Flapping cambered wing

The second test problem considers cambered wings. This aerodynamic problem was previously simulated by Stanford and Beran in [6] and Ghommem et al. in [5]. The shape of the wing is rectangular with an aspect ratio of 6, and a transversal section consisting of a NACA 83XX airfoil. A sketch of the flapping wing along with the wake is presented in Figure 16. The parameters used to simulate the flapping cambered problem flight are presented in table 2.

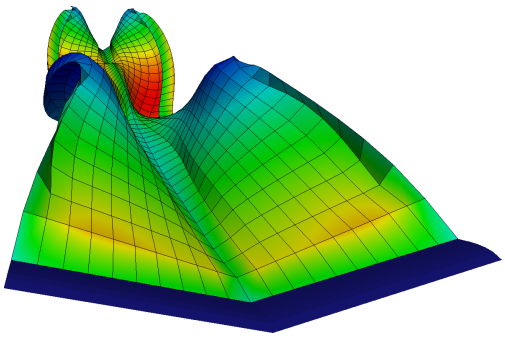


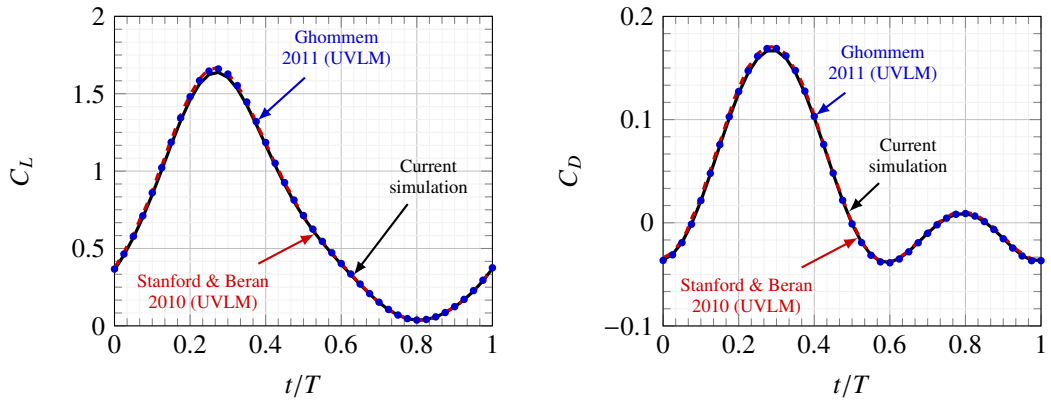
Figure 16: Flapping cambered wing problem sketch.

Parameter	
Flight velocity ($\underline{v}$ in m/s)	10
Flapping frequency (ω_f in Hz)	2
Reduce frequency ($k = \omega_F/2/\underline{u}_\infty$)	0.1
Flapping amplitude (θ_{max} in °)	45
Pitch (α in °)	5
Flapping period ($T = 2\pi/\omega$ in s)	π
Time step ($t = T/40$ in s)	$\pi/40$

Table 2: Flapping cambered wing problem data.

The problem is solved using both the full domain and a reduced domain accounting for the problem symmetry (that corresponds to a single wing). Each of the wings is discretized with six elements along the chordwise direction and ten elements along the spanwise direction. This implies that a total number of 120 grid cells is used to discretize the full domain.

Figure 17 illustrates the lift and thrust coefficients obtained from the simulation performed with PyFly, and resulted from previous studies [5, 6]. The aerodynamic coefficients obtained from the current simulations are in a good agreement with those obtained from the previous UVLM implementations [5, 6] implying that PyFly is a adequate tool for studying lifting body problems.



(a) Lift coefficient

(b) Thrust coefficient

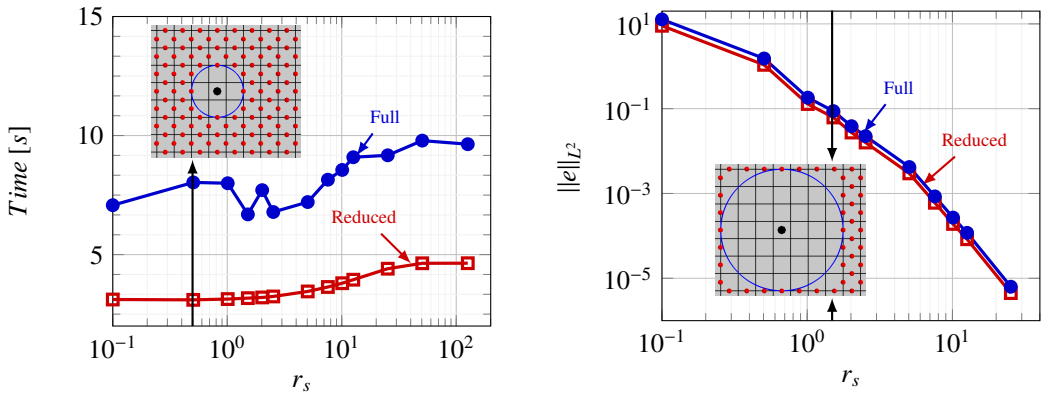
Figure 17: Lift and Thrust (negative drag) coefficients: comparison against other flow solvers.

Next, we study the performance of enhanced PyFly when solving the flapping cambered problem, we compute the cost of computation, the L^2 error with respect to the traditional UVLM method and the saving in computational time relative to the error of computation for various values of r_s. Figure 18 shows the variations of the simulation time and error estimate with the close-to-target parameter r_s while considering the full and reduced domains. Two sketches are added in Figure 18 to illustrate the difference between two pointwise approximations based on the value of the parameter r_s. We observe a significant effect of the reduction in the aerodynamic domain on the simulation time with a slight effect on the error. In Figure 19, we depict the variations of the computational saving with the error. Considering acceptable an error of order $O(10^{-3})$, the maximum saving in time for the flapping cambered wing problem is found equal to 24%. This value is obtained when the close-to-target parameter is set equal to $r_s = 20\Lambda$.

4.3. Three flapping wings: formation flight

The analysis of formation flights can be helpful to develop an understanding of the potential power saving that birds can achieve through organized patterns when travelling over long distances without stopping and feeding [42]. In this numerical test problem, we simulate the flight of three birds in a *V-shape* formation and in proximity of each other. This numerical example illustrates the capability of PyFly to handle multiple bodies.

The wings are represented as rectangular shaped bodies with a root chord of $c = 1$, an aspect ratio of $d/c = 3$ and a transversal section of a NACA 83XX airfoil. Figure 20 illustrates the *V-shape* formation flight. The birds fly at the same altitude separated by a distance of seven times the chord length ($7c$) while the bird in the middle is positioned ahead along the flight direction by a



(a) Computational time.

(b) L^2 error norm of ciruclation γ.

Figure 18: Cost and Error estimates for the flapping cambered wing problem.

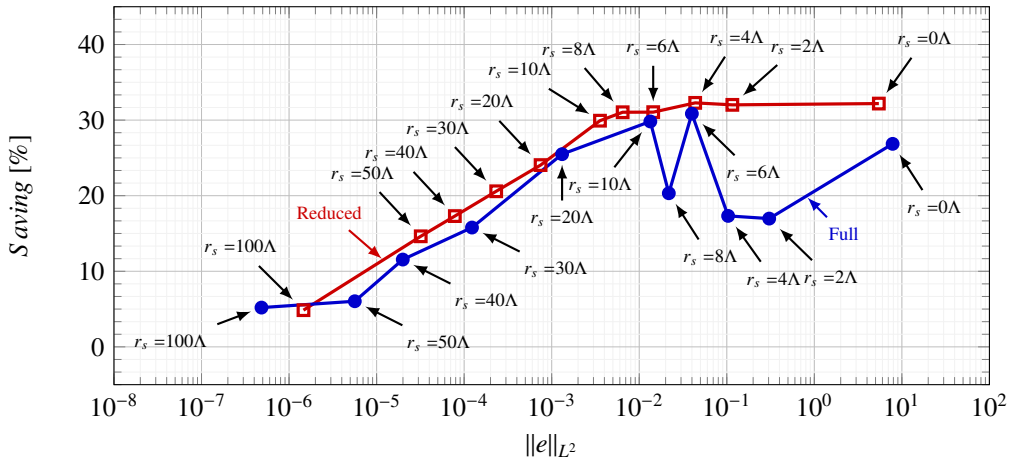


Figure 19: Plot of computational saving vs error for the flapping cambered wing problem.

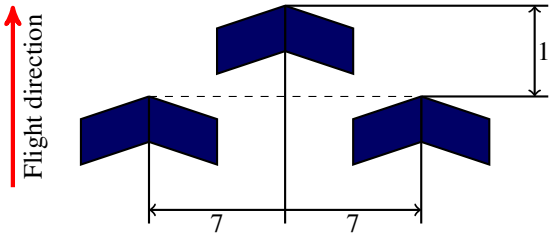


Figure 20: V-shape flight formation.

distance equal to the chord length of the wing (c). Figure 21 illustrates the flapping wings and generated wakes as obtained from PyFly. The wings of the three birds flap at the same frequency of ω_f while those move forward at the speed $\underline{v}$. The parameters used in the computation of the problem are presented in Table 3. The lifting bodies that represent the wings in the problem are discretized with six elements along the chordwise direction and ten elements in the span direction of the wings. Again, the problem is solved using the full domain and a reduced domain by considering the symmetry in the problem.

The problem is solved using various values of the close-to-target parameter r_s. Estimates of the cost and error obtained when varying the parameter r_s are presented in Figure 22. Similar trends of the error and cost are obtained when varying the value of the parameter r_s. A noticeable effect of reducing the aerodynamic mesh on the simulation time is observed while keeping the same

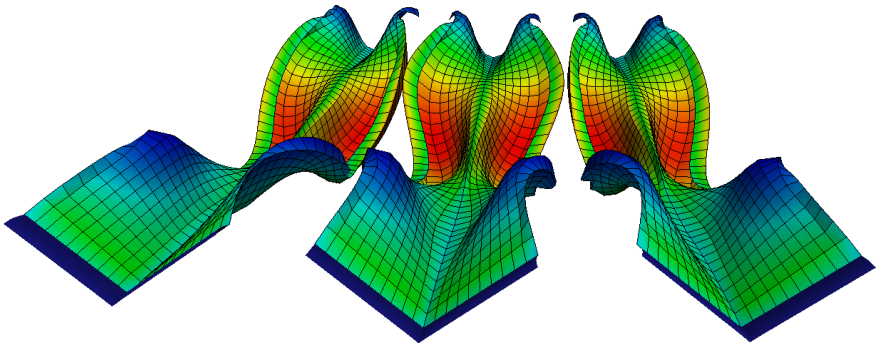
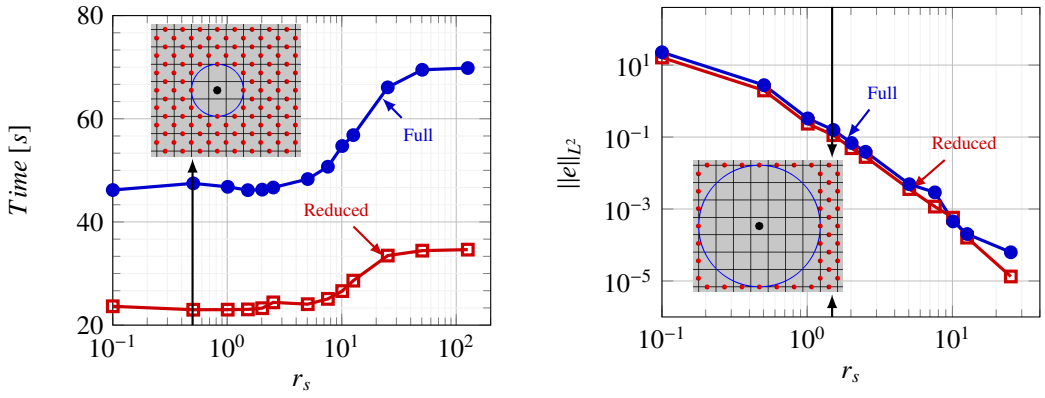


Figure 21: Three flapping bodies problem sketch.

Parameter	
Flight velocity ($\underline{v}$ in m/s)	10
Flapping frequency (ω_f in Hz)	2
Reduce frequency ($k = \omega_F/2/\underline{u}_\infty$)	0.1
Flapping amplitude (θ_{max} in °)	45
Pitch (α in °)	5
Flapping period ($T = 2\pi/\omega$ in s)	π
Time step ($t = T/40$ in s)	$\pi/40$

Table 3: Three flapping bodies problem data.

error level. In Figure 23, we plot the variations of the computational saving with the error. Setting an upper bound of the error of order $O(10^{-3})$, the computational saving are found between 25% and 30%.



(a) Computational time.

(b) L^2 error norm of ciruclation γ.

Figure 22: Cost and Error estimates for the three birds flapping wing problem.

4.4. Rotating blades

In this numerical example, we compute the aerodynamic response of rigid rotating blades subjected to an incoming steady freestream flow. We consider a wind turbine composed of three blades, each 120 degrees out of phase from each other. Figure 24 illustrates the blade geometry. It is discretized using six elements along the chordwise direction and 14 elements along the blade spanwise direction. The transversal section of the blade is a NACA83XX airfoil. Additional information on the parameters used in the computation of the rotating blades problem is presented in Table 4.

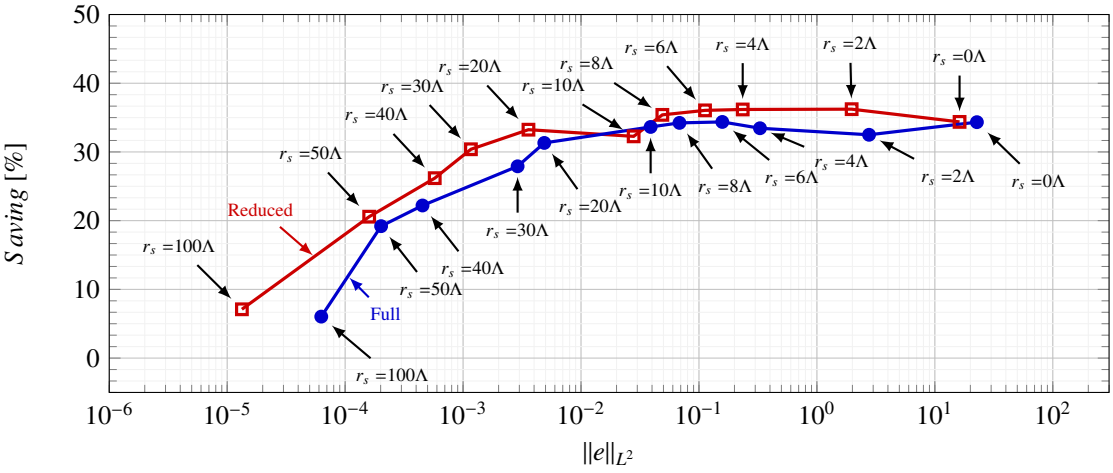


Figure 23: Plot of computational saving vs error for the three birds flapping wing problem.

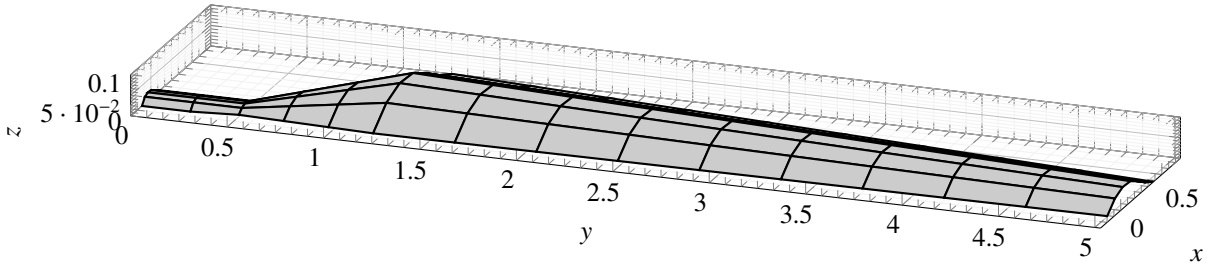


Figure 24: Blade geometry.

Parameter	
Freestream flow velocity ($\underline{u}_\infty$ in m/s)	10
Angular velocity (rotation) (ω_r in rad/s)	2
Average element length (h in m)	≈ 0.2
Time step (t in s)	$h/\|\|\underline{u}_\infty\|\|$

Table 4: Wind turbine problem data.

We use PyFly to simulate the interactions of the multiple grids (three blades) while varying the values of parameter r_s. Figure 25 shows the formation and shedding of the bound vortices into the wake. The roll-up of the wake vortices shed from the blades trailing edge can be observed.

In Figure 26, we plot the variations of the computational time and the error with the parameter r_s. As shown in Figure 26(a), we observe a slight variations in the computational time with r_s (up to $r_s = 5$) followed by a sharp increase. Then, the computational time stabilizes for $r_s \geq 20$. As for the error, a nearly-linear decreasing trend is observed when increasing the value of the parameter r_s. The curves shown in Figure 26 are presented in the log scale.

To show the tradeoffs between the possible computational saving (obtained from the pointwise approximation) and the loss in the aerodynamic solution accuracy, we plot in Figure 27 the variations of the computational saving with the error when varying the values of the parameter r_s. We observe a nearly-linear trend followed by a stabilization indicating that setting the parameter r_s greater than a certain value (8 Λ) does not speed up significantly the simulations but this has a noticeable effect on the accuracy of the solution as can be observed from the large errors. Assuming an error of approximation in the order of $O\left(10^{-3}\right)$, a maximum saving in time of 20% is obtained. These results demonstrate the capability of PyFly to speed up the aerodynamic simulations while keeping a good accuracy. Furthermore, the user has the capability to select the potential saving in the computational cost based on the required accuracy of the simulated aerodynamic problem and the objective of the study. For instance, optimization and sensitivity analyses require running the forward aerodynamic problem several times and then considering ways to speed up

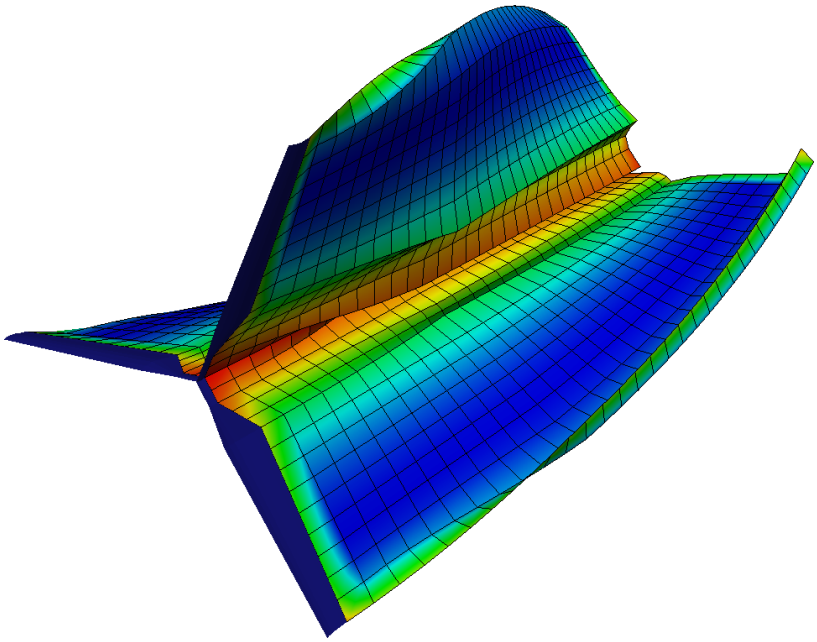
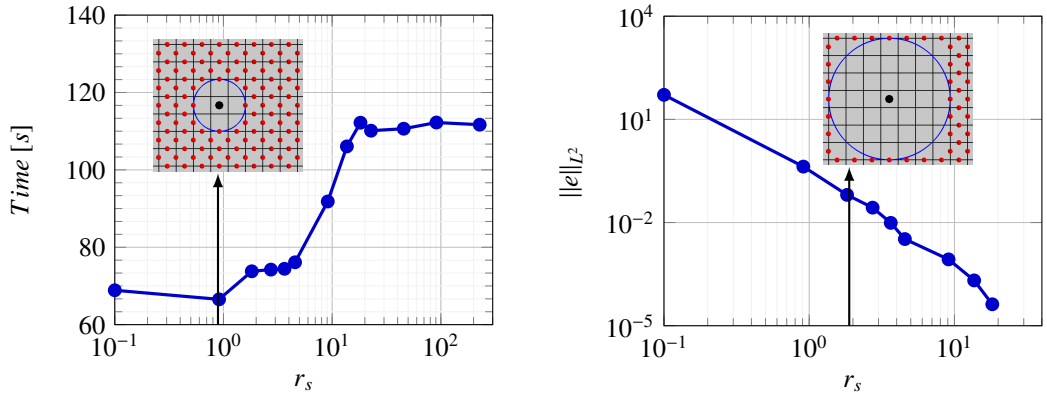


Figure 25: Rotating blades problem sketch.

the simulations such as the proposed pointwise approximation will be helpful to conduct such analyses within a reasonable time frame.



(a) Computational time.

(b) L^2 error norm of circulation γ.

Figure 26: Cost and Error estimates for the rotating blades problem.

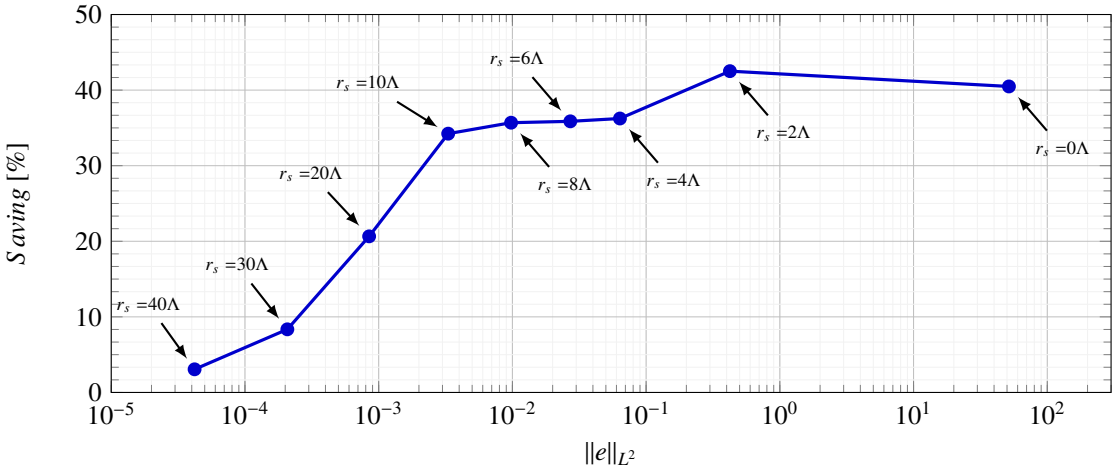


Figure 27: Plot of computational saving vs error for the rotating blades problem.

5. Conclusions

In this work, we discuss a fast and user-friendly computational framework based on the unsteady vortex lattice method (UVLM) for aerodynamic simulations. We used mixed-language programming approach combining Python for high-level management of grid objects and processing the aerodynamic quantities and Fortran for the computationally-demanding kernels. This mixed-programming approach enables us to reduce the overall complexity of applying UVLM to a wide range of problems. We then use PyFly to simulate a set of aerodynamic problems to demonstrate its flexibility and efficiency. We also showed the potential computational saving in the simulations thanks to the new implementation of UVLM. The future work will include the incorporation of the solid body flexibility using structural models to simulate the aeroelastic behavior.

Appendix A. Biot Savart Equation Formulation

The velocity field induced by vorticity satisfies Equations (33) and (34).

$$\nabla \cdot \underline{u} = \underline{0} \tag{33}$$

$$\nabla \times \underline{u} = \underline{\omega} \tag{34}$$

A potential field $\underline{\phi}$ is introduced to proceed with the computation of the velocity field. This field is related with velocity by Equation (35) in order to satisfy Equation (33).

$$\underline{u} = \nabla \times \underline{\phi} \tag{35}$$

The relation between potential field and vorticity is obtained by replacing Equation (35) in Equation (34).

$$\nabla \times \nabla \times \underline{\phi} = \underline{\omega} \tag{36}$$

Equation (36) is simplified by using condition $\nabla \cdot \underline{\phi} = \underline{0}$ and identity $\nabla \times \nabla \times \underline{a} = \nabla(\nabla \cdot \underline{a}) - \Delta \underline{a}$. The relation becomes

$$\Delta \underline{\phi} = -\underline{\omega} \tag{37}$$

Equation (37) has a Poisson's equation form allowing to solve $\underline{\phi}$ by using

$$\underline{\phi}(\underline{y}_j) = \int \omega(\underline{x}_i) \mathbf{G}(\underline{x}_i, \underline{y}_j)\, dV(\underline{x}_i) \tag{38}$$

where $\mathbf{G}$ is the Green's function for the Laplace's equation. Green's function describes the effect of the source located at $\underline{x}_i$ on the target at the point $\underline{y}_j$.

$$\mathbf{G}(\underline{x}_i, \underline{y}_j) = \frac{1}{4\pi |\underline{x}_i - \underline{y}_j|} \tag{39}$$

Velocity is computed by substituting Equation (38) in Equation (35)

$$\underline{u}(\underline{y}_j) = \frac{1}{4\pi}\nabla \times \int \frac{\omega(\underline{x}_i)}{|\underline{x}_i - \underline{y}_j|}\, dV(\underline{x}_i) \tag{40}$$

Vorticity ω is expressed in terms of circulation Γ by using Equation (41)

$$\Gamma = \int \omega\, dS \tag{41}$$

where circulation Γ is a measure of the rotation in the fluid. Substituting Equation (41) in Equation (40), the velocity field becomes

$$\underline{u}(\underline{y}_j) = \frac{1}{4\pi}\nabla \times \oint \frac{\Gamma d\ell(\underline{x}_i)}{|\underline{x}_i - \underline{y}_j|} \tag{42}$$

where $d\ell$ refers to the vortex segment. Biot-Savart equation is obtained by rearranging Equation (43)

$$\underline{u}(\underline{y}_j) = -\frac{1}{4\pi}\oint \Gamma(\underline{x}_i) d\ell(\underline{x}_i) \times \nabla \frac{1}{|\underline{x}_i - \underline{y}_j|} \tag{43}$$

$$\underline{u}(\underline{y}_j) = \frac{1}{4\pi}\oint \frac{\Gamma(\underline{x}_i) d\ell(\underline{x}_i) \times (\underline{x} - \underline{y})}{|\underline{x}_i - \underline{y}_j|^3} \tag{44}$$

References

[1] R. E. Perez, P. W. Jansen, J. R. R. A. Martins, pyOpt: a python-based object-oriented framework for nonlinear constrained optimization, Structural and Multidisciplinary Optimization 45 (1) (2012) 101 – 118.

[2] J. J. Alonso, P. LeGresley, E. van der Weide, J. R. R. A. Martins, J. J. Reuther, pymdo: A framework for high-fidelity multi-disciplinary optimization, in: 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 20044480, 2004.

[3] Y.-Y. Chen, D. L. Bilyeu, L. Yang, S.-T. J. Yu, Solvcon: A python-based cfd software framework for hybrid parallelization, in: 49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, AIAA 2011-1065, 2011.

[4] L. Dalcin, N. Collier, P. Vignal, A. M. A. Cortes, V. M. Calo, Petiga: A framework for high-performance isogeometric analysis, Computer Methods in Applied Mechanics and Engineering 308 (2016) 151–181.

[5] M. Ghommem, M. R. Hajj, D. T. Mook, B. K. Stanford, P. S. Beran, L. T. Watson, Global optimization of actively-morphing flapping wings, Journal of Fluids and Structures 30 (2012) 210–228.

[6] B. K. Stanford, P. S. Beran, Analytical sensitivity analysis of an unsteady vortex-lattice method for flapping-wing optimization, Journal of Aircraft 47 (2010) 647–662.

[7] M. Ghommem, N. Collier, A. H. Niemi, V. M. Calo, Shape optimization and performance analysis of flapping wings, in: Proceedings of The Eighth International Conference on Engineering Computational Technology, 2012.

[8] P. Persson, D. Willis, J. Peraire, Numerical simulation of flapping wings using a panel method and high-order Navier-Stokes solver, International Journal for Numerical Methods in Engineering 89 (2012) 1296–1316.

[9] E. C. Stewart, M. J. Patil, R. A. Canfield, R. D. Snyder, Aeroelastic shape optimization of a flapping wing, Journal of Aircraft 53 (2016) 636–650.

[10] M. L. Verstraete, S. Preidikman, B. A. Roccia, D. T. Mook, A numerical model to study the nonlinear and unsteady aerodynamics of bioinspired morphing-wing concepts, International Journal of Micro Air Vehicles 7 (2015) 327–345.

[11] A. T. Nguyen, J.-K. Kim, J.-S. Han, J.-H. Han, Extended unsteady vortex-lattice method for insect flapping wings, Journal of Aircraft 0 (2016) 1–10.

[12] J. D. Colmenares, O. D. Lpez, S. Preidikman, Computational study of a transverse rotor aircraft in hover using the unsteady vortex lattice method, Mathematical Problems in Engineering 2015, article ID 478457.

[13] C. G. Gebhardta, S. Preidikmana, J. Massaa, Numerical simulations of the aerodynamic behavior of large horizontal-axis wind turbines, International Journal of Hydrogen Energy 35 (2010) 6005–6011.

[14] F. Meng, H. Schwarze, F. Vorpahl, M. Strobel, A free wake vortex lattice model for vertical axis wind turbines: Modeling, verification and validation, Journal of Physics 555 (2014) 1–8.

[15] N. Sezer-Uzol, O. Uzol, Effect of steady and transient wind shear on the wake structure and performance of a horizontal axis wind turbine rotor, Wind Energy 16 (2013) 1–17.

[16] A. Rosenberg, A. Sharma, A prescribed-wake vortex lattice method for preliminary design of co-axial, dual-rotor wind turbines, Journal of Solar Energy Engineering 138 (2016) 1–9.

[17] B. F. Ng, H. Hesse, R. Palacios, J. M. R. Graham, E. C. Kerrigan, Aeroservoelastic state-space vortex lattice modeling and load alleviation of wind turbine blades, Wind Energy 18 (2015) 1317–1331.

[18] G. Tescione, C. S. Ferreira, G. van Bussel, Analysis of a free vortex wake model for the study of the rotor and near wake flow of a vertical axis wind turbine, Renewable Energy 87 (2016) 552–563.

[19] T. Sebastian, M. Lackner, Development of a free vortex wake method code for offshore floating wind turbines, Renewable Energy 46 (2012) 269–275.
[20] L. He, S. A. Kinnas, W. Xu, A vortex-lattice method for the prediction of unsteady performance of marine propellers and current turbines, International Journal of Offshore and Polar Engineering 23 (2013) 210–217.
[21] M. Jeona, S. Leea, S. Leeb, Unsteady aerodynamics of offshore floating wind turbines in platform pitching motion using vortex lattice method, Renewable Energy 65 (2014) 207–212.
[22] S. Preidikman, Numerical simulations of interactions among aerodynamics, structural dynamics, and control systems, Ph.D. thesis, Virginia Tech, Blacksburg, VA (1998).
[23] Z. Wang, Time-domain simulations of aerodynamic forces on three-dimensional configurations, unstable aeroelastic responses, and control by network systems, Ph.D. thesis, Virginia Tech, Blacksburg, VA (2004).
[24] D. Garcia, N. Collier, M. Ghommem, Pyfly, https://bitbucket.org/dgarcialozano/pyfly_v2 (2017).
[25] J. Katz, A. Plotkin, Low-Speed Aerodynamics, Cambridge University Press, MA, 2001.
[26] H. P. Langtangen, Python Scripting for Computational Science, Springer, 2010.
[27] A. O. Nuhait, M. F. Zedan, Numerical simulation of unsteady flow induced by a flat plate moving near ground, Journal of Aircraft 30 (1993) 611–617.
[28] A. O. Nuhait, D. T. Mook, Aeroelastic behavior of flat plates moving near the ground, Journal of Aircraft 47 (2010) 464–474.
[29] M. Ghommem, V. M. Calo, D. Garcia, Enclosure enhancement of flight performance, Theoretical & Applied Mechanics Letters. 4, doi: 10.1063/2.1406203.
[30] D. F. Rogers, An Introduction to NURBS With Historical Perspective, Academic Press, San Diego, CA, 2001.
[31] L. Piegl, W. Tiller, The NURBS Book (Monographs in Visual Communication), 2nd ed., Springer-Verlag, New York, 1997.
[32] G. Farin, NURBS Curves and Surfaces: From Projective Geometry to Practical Use, A. K. Peters, Ltd., Natick, MA, 1995.
[33] E. Cohen, R. Riesenfeld, G. Elber, Geometric Modeling with Splines. An Introduction, A K Peters Ltd., Wellesley, Massachusetts, 2001.
[34] J. Peters, U. Reif, Subdivision Surfaces, no. 3 in Geometry and Computing, Springer, 2008.
[35] E. Catmull, J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, Computer-Aided Design 10 (6) (1978) 350 – 355.
[36] SymPy Development Team, SymPy: Python library for symbolic mathematics (2012).
URL http://www.sympy.org
[37] T. Oliphant, Guide to NumPy, Trelgol Publishing, 2006.
[38] D. I. Ketcheson, K. T. Mandly, A. J. Ahmadia, A. Alghamdi, M. Q. D. Luna, M. Parsani, M. G. Knepley, M. Emmett, Pyclaw: Accessible, extensible, scalable tools for wave propagation problems, SIAM: SIAM Journal on Scientific Computing 34 (2012) 210–231.
[39] L. Dalcin, R. Paz, P. Kler, A. Cosimo, Parallel distributed computing using Python, Advances in Water Resources 34 (9) (2011) 1124 – 1139. doi:10.1016/j.advwatres.2011.04.013.
URL http://www.sciencedirect.com/science/article/pii/S0309170811000777
[40] P. Peterson, F2PY: a tool for connecting Fortran and Python, International Journal of Computational Science and Engineering 4 (4).
[41] M. Ghommem, N. Collier, A. H. Niemi, V. Calo, On the shape optimization of flapping wings and their performance analysis, Aerospace Science & Technology 32 (2014) 274–292.
[42] M. Ghommem, V. Calo, Flapping wings in line formation flight: a computational analysis, The Aeronautical Journal 118 (2014) 485–501.
[43] M. F. Neef, D. Hummel, Euler Solutions for a Finite-Span Flapping Wing in Mueller T. J. (ed.), Fixed and Flapping Wing Aerodynamics for Micro Air Vehicle Applications, American Institute of Aeronautics and Astronautics, Inc., Reston, 2004.