SAPIENZA - UNIVERSITÁ DI ROMA

FACOLTÁ DI INGEGNERIA CIVILE E INDUSTRIALE

Ingegneria Aeronautica

FINAL YEAR THESIS

Modeling of Unsteady Vortex Lattice Method for wing flutter control in OpenMDAO

Author:
Giovanni Pesare

Supervisors:
Prof. Walter Lacarbonara
Prof. Joaquim R.R.A. Martins
Prof. Joseph Morlier

Co-supervisors:
Dr Andrea Arena
John Jasa

October 2016

Abstract

The main objective of this work is to implement a tool capable of performing dynamic aeroelastic analysis using the OpenMDAO framework. The starting point for this project is the open-source aerostructural analysis code *OpenAeroStruct*, implemented by the MDOlab at the University of Michigan. The initial version of this code performs steady linear aeroelastic analysis and optimization by solving for the aerodynamic forces using the classical Vortex Lattice Method (VLM) and by modeling the wing structure as a spatial beam with a tube section.

In order to achieve this objective, several modifications to both the structural and aerodynamic models are introduced and investigated. For the structural analysis, these modifications concern the structural matrix assembly, introduction of modal analysis, and resolution of equilibrium equation by the Newmark-Beta solver. For the aerodynamic analysis, the aerodynamic model is updated to its unsteady formulation (UVLM) and a time loop is created in order to perform the simulation in the time domain. Time-dependent problems have not been studied in depth using the multi-disciplinary optimization framework OpenMDAO and this innovation comprises important modifications in the *OpenAeroStruct* structure.

As a result of those modifications, the system flutter velocity can not be evaluated by analysing the time history of system variables and their Fast Fourier Transform (FFT). In fact, this analysis provides important information about the amplitude and frequency of signal oscillations, such as the wing tip displacement. Finally, a linear Tuned Mass Damper (TMD) is introduced in order to ameliorate the dynamic aeroelastic response of the system.

Acknowledgements

This thesis became a reality with the kind support and help of many people. I would like to extend my sincere thanks to all of them.

Foremost, I would like to express my gratitude to my advisor Prof. Walter Lacarbonara of "La Sapienza" University for the continuous support during my studies at "La Sapienza" University. His guidance helped me in all of my important academic decisions, as those about the double degree option or the thesis project in the United States. I'm honored to have worked with a Professor of his stature. I'm also grateful with Ing. Andrea Arena, for his invaluable help and support, especially during the breathtaking last week of this project.

I wish express my sincere thanks to Prof. Joaquim R.R.A. Martins and the MDOlab of University of Michigan, for welcoming me into their group during my period in Ann Arbor. In particular, I am very grateful to John Jasa, Shamsheer Chauhan, Eiríkur Jónsson and Cristina Riso for their daily support and their friendship. Without their passionate participation and input, the project could not have been successfully conducted.

On the French side, I place on record my sincere thanks to Prof. Guilhem Michon, who has accepted me into his laboratory for the first part of this project, and to Prof. Joseph Morlier, for having supported my candidature for the FondationSupaero "Scholarship of excellence". I would like to express my gratitude to all people I have met during my two years in France and to all my friends from Italy that have always supported me. A special thanks goes to "les italiens" Andrea Antolino, Andrea Guarriello, Luca Palomba, Stefano Romani and Francesco Romano for the great moments we spent together in Toulouse.

Finally, I must express my very profound gratitude to my parents and to my girlfriend Francesca for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Rome, 21 October 2016 Giovanni Pesare

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Aerostructural design

The field of aeroelasticity focuses on the study of systems that display a "mutual interaction among inertial, elastic and aerodynamic forces" [7]. Aeroelastic systems are characterized by a coupling between a structure and the surrounding fluid. Therefore, the response of the overall system is determined by the interaction between the coupled dynamics of the fluid and the structure. A common example of such a system is a wing moving through air. Since the fluid-structure coupling can become quite complex, numerical and theoretical consideration even of rather simple aeroelastic systems can become involved [8].

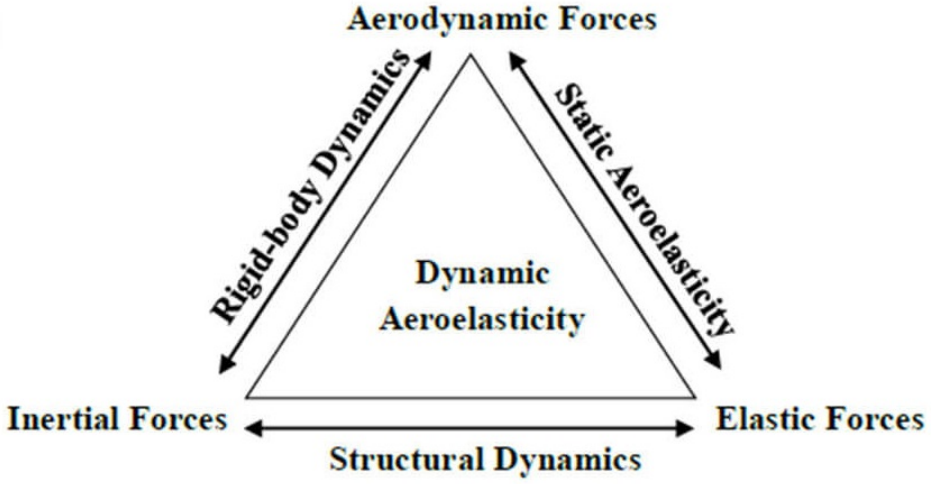


FIGURE 1.1: Aeroelastic interaction chart from Collar [1]

To facilitate our discussion, we begin with Collar's famous triangle diagram shown in Figure 1.1. The three vertices in the diagram are Elasticity, Aerodynamics, and Dynamics. Each one of them represents a subject of interest individually. Combining either two of them or all of them yields different aspects of aeroelastic analysis. When considering only elasticity and aerodynamics, it means that the wing is treated as an elastic body subjected to an aerodynamic force which is considered in steady state. This is a static aeroelasticity problem. Two typical examples are wing-divergence phenomenon and reversal of ailerons, which are static instabilities of a lifting surface of an aircraft in flight. At low speeds of flight, the effect of elastic deformation is small. But at higher speeds, the effect of elastic deformation can become large enough to cause a wing to be unstable, or to render a control surface ineffective, or even to reverse the sense of control. Aerodynamics and Dynamics together give us a classical aerodynamics analysis. In this case, the aircraft or the wing is assumed rigid; no elastic deformation is induced. This is the typical approach when analyzing helicopter or wind turbine blades. All three subjects together are needed to solve an unsteady aeroelastic problem such as flutter phenomena and buffeting.

Flutter is a dynamic instability occurring in an aircraft in flight. At a speed called the flutter speed, the interaction between elastic deformations and aerodynamics causes a sustained oscillation of the wing or control surfaces [9, 10]. The phenomenon of flutter has been observed from the early days of powered flights as well as in civil engineering. The oscillations associated with flutter can be very serious and sometimes catastrophic. A well-known flutter example happened on November 7, 1940, when the Tacoma Narrows bridge in the State of Washington was destroyed by wind action. Since then, aeroelasticity has been a very important factor in the design of long-span suspension bridges.

The oscillatory motion of a fluttering wing, in general, involves the coupling of several degrees of freedom. A rigid airfoil so constrained as to have only the deflection, which simulates the flexural degree of freedom, does not flutter. A rigid airfoil with only the torsional degree of freedom can flutter only if the angle of attack is at or near the stalling angle. The coupling between various degrees of freedom accounts for the phase shift between the motions in those degrees of freedom. Experiments on cantilever wings show that the flexural motions at all points across the span are approximately in phase with one another, and likewise the torsional motions are all approximately in phase, but the deflection is considerably out of phase from the torsional motion. This phase difference at the critical flutter speed brings about the flutter in such a way that energy can be extracted from the airstream passing by.

We focus on the flutter problem in this work. Flutter analysis is very challenging because of its inherent coupling and nonlinear characteristics. One can not determine the structural response without first knowing the applied loads, and on the other hand, one can not calculate the aerodynamic loads on a body without first knowing the state (position, velocity) of that body.

The approach followed in this thesis is the analysis in the time domain: the governing equations of the whole dynamic system will be integrated simultaneously and interactively in time. Then it is possible to plot nodal displacements and rotations to evaluate the pre- and post-flutter behaviour of the structure.

1.2 Aerodynamic methods

An important part of modeling the dynamics of an aircraft is its unsteady aerodynamics. Unsteady aerodynamics modeling, which involves computation of the forces exerted on a body in a time dependent air flow, has been important for reliable flutter analysis. There are several methods available to study unsteady aerodynamics, ranging from advanced, high fidelity computational fluid dynamics (CFD) solvers [11], to potential flow based aerodynamic strip theory, which utilizes 2-D infinite wing assumptions [12]. CFD methods provide models with very high number of states which are very computationally expensive to simulate and unsuitable for conceptual design. On the other hand, strip theory based methods provide computationally inexpensive models, but they lack the desired fidelity we may require for aeroelastic analysis.

Panel methods based on potential flow, developed in the 1960s and 1970s, are able to model the lifting characteristics of finite wings with reasonable accuracy while being computationally inexpensive [13]. The equation governing low-speed potential flows is Laplace's equation for the velocity potential. One of the key features of this linear differential equation is that a 3-D flow-field problem can be converted to a reduced-dimension equivalent one by distributing singularity (elementary) solutions over the surface where the flow potential must be found (Boundary-Value Problem). As a result, the numerical solution in terms of singularities is faster compared to field methods where the unknown quantities are distributed in the entire volume surrounding the body. The reduction of the 3-D computational domain to a surface problem has led to the rapid development of computer codes for the implementation of potential-flow methods, and thanks to their flexibility and relative economy they continue to be widely used despite the availability of more exact approaches.

In this context, the Vortex Lattice Method (VLM) was among the earliest methods utilizing computers to actually assist aerodynamicists in estimating aircraft aerodynamics. It models relevant 3-D flow physics, such as the accurate prediction of wing-tip effects [14], and the aerodynamic interference between wakes and lifting surfaces [15]. This method is based on the idea of a vortex singularity as the solution of Laplace's equation, and it is very easy to use and is capable of providing remarkable insight into wing aerodynamics and component interaction.

1.2.1 Vortex Lattice Method

The foundations of potential-flow Vortex-Lattice Methods (VLM) can be traced back to Helmholtz's seminal work on vortex flows and Joukowski's contributions on circulation [16], but the earliest formulations appeared in the 1930s. Rosenhead studied a two-dimensional vortex-layer by replacing it with a system of vortex filaments and he showed that the vortex sheet rolled up with time [17]. The term "vortex lattice" was coined by Falkner in 1943 [18]. The concept was simple but it relied on a numerical solution, so it was not until digital computers became available that practical implementations became widespread.

Hedman established the now classical (steady) VLM in 1965 [19]. He idealised the mean aerodynamic surface into small trapezoidal lifting elements each containing a horseshoe vortex with its bound spanwise element along the swept quarter-chord of the element, and locating the collocation points for the non-penetration boundary condition at the three-quarter chord. The downwash at each collocation point was computed through the Biot-Savart law, and compressibility was accounted for by means of the Prandtl-Glauert transformation. The method continued to be widely used in the following decades and alongside different implementations of the code various relevant numerical issues were also addressed: increasing convergence and accuracy [20], accounting for wake roll-up instead of assuming a flat wake [21], and modelling leading-edge separation [22].

As the VLM was initially limited to steady load calculations it was natural to develop an unsteady equivalent. Albano and Rodden [23] extended the VLM to harmonically oscillating surfaces for an assumed flat wake. Replacing the vortex sheet by one of (equivalent) oscillating doublets, the Doublet-Lattice Method (DLM) was obtained. In the last decades, it has been broadly used for unsteady load computations and the prevalent tool in subsonic aircraft aeroelasticity [24].

When referring to the Vortex-Lattice Method, it is a common misconception (particularly within the aeroelastic community) to assume that it is limited to the steady equivalent of the Doublet-Lattice Method. While this is true for the steady version, it is also known that a panel with a piecewise constant doublet distribution is equivalent to a vortex ring around its periphery [25]. Hence, VLM can be directly extended to non-stationary situations, giving rise to the time-domain Unsteady Vortex-Lattice Method (UVLM). The UVLM is mentioned in many textbooks on aerodynamics, but the most comprehensive description is possibly given by Katz and Plotkin [3], which is the fundamental reference for the numerical implementation used in this work and described in Chapter 3.

As opposed to the DLM, which is written in the frequency domain on a fixed geometry, the UVLM is formulated in the time domain and allows the shape of the force-free wake to be obtained as part of the solution procedure. The DLM offers a faster way of computing unsteady aerodynamic loads, but it is a linear method restricted to small

out-of plane harmonic motions with a flat wake. Hence, while the DLM has dominated in fixed-wing aircraft aeroelasticity, the UVLM has been gaining ground in situations where free-wake methods become a necessity because of geometric complexity, such as flapping-wing kinematics [26], rotorcraft [27], or wind turbines [28]. With the advent of novel vehicle configurations and increased structural flexibility for which the underlying assumptions of the DLM no longer hold, the UVLM constitutes an attractive solution for aircraft dynamics problems and has been recently exercised in problems such as unsteady interference [29], flutter suppression [30], gust response [31], optimisation [32], and coupled aeroelasticity and flight dynamics [33].

1.3 CRM wing

The Common Research Model (CRM) is a realistic example of aircraft conceived by NASA (*National Aeronautics and Space Administration*) for the purpose of validating specific applications of CFD [34]. NASA felt that a conventional configuration would be sufficiently challenging and relevant for aerodynamic prediction validation and the development went forward from this point. There exist many different configurations of the CRM, such as the one shown in Figure 1.2, which is composed of body, wing, nacelles, pylons, and a horizontal tail.

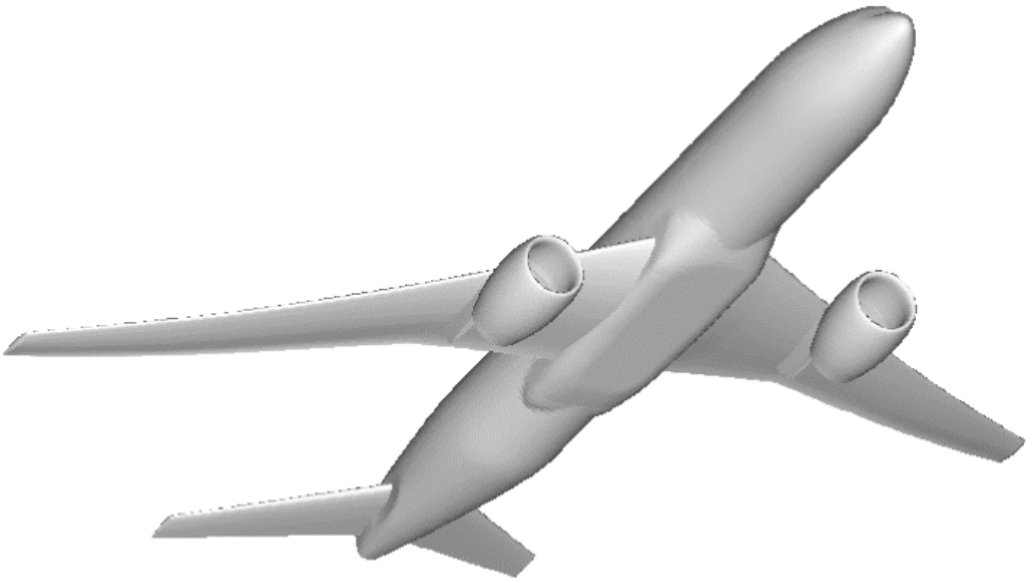


FIGURE 1.2: Common Research Model

The fuselage is representative of a wide-body commercial transport aircraft; it includes a wing-body fairing, as well as a scrubbing seal for the horizontal tail. The nacelle is a single-cowl, high bypass ratio, flow-through design with an exit area sized to achieve a natural unforced mass-flow-ratio typical of commercial aircraft engines at cruise.

The CRM is based on a transonic transport configuration designed to fly at a cruise Mach number of $M = 0.85$ with a nominal lift condition of $C_L = 0.50$ and at a Reynolds number of $Re = 40$ million per reference chord. In this work only the wing will be used and its nominal geometrical characteristics are provided below:

- span $= 58.763\,m$
- root chord $= 7.005\,m$
- reference surface area $= 15105.888\,m^2$

- aspect ratio = 9.0
- taper ratio = 0.275
- quarter-chord sweep = 35.0°

The CRM wing is shown in Figure 1.3, which includes a spanwise planform break (kink or 'Yehudi') at the 37% of the semi-span. It should be noted that the portion of the wing closest to the root is actually inside the fuselage.

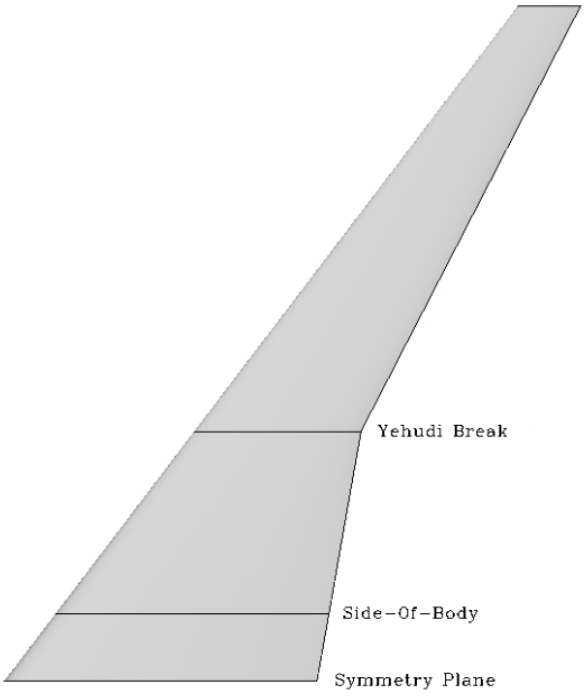


FIGURE 1.3: Planform of the CRM Wing

1.4 OpenMDAO

OpenMDAO is an open-source Multidisciplinary Design, Analysis, and Optimization (MDAO) framework, written in the Python programming language and developed by NASA [35]. OpenMDAO is a high-performance computing platform for systems analysis and optimization that allows users to decompose models, making them easier to build and maintain, while still solving them in a tightly-coupled manner with efficient parallel numerical methods. Moreover it allows users to combine analysis tools (or design codes) from multiple disciplines, at multiple levels of fidelity, and to manage the interaction between them.

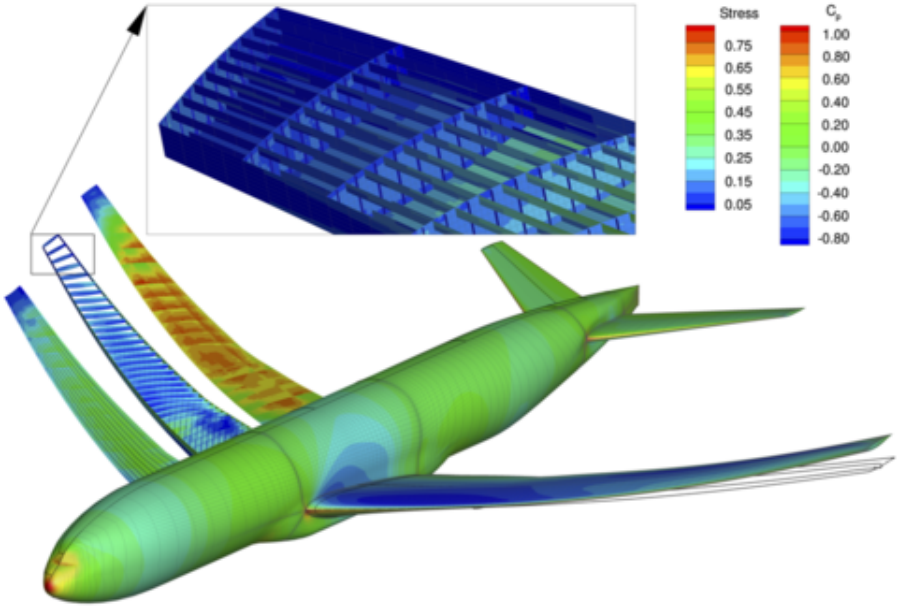


FIGURE 1.4: Multipoint High-Fidelity Aerostructural Optimization of a Transport Aircraft Configuration using OpenMDAO [2]

OpenMDAO is designed to separate the flow of information (dataflow) from the process in which analyses are executed (workflow), along with optimization algorithms and other advanced solution techniques. In the framework structure four specific constructs are used: Component, Assembly, Driver, and Workflow. The construction of system models begins with wrapping (or writing from scratch) various analysis codes as Components. A group of components is linked together inside an Assembly, specifying the dataflow between them. Once the dataflow is in place, it is possible to select specific Drivers (optimizers, solvers, design of experiments, etc.) and set up a Workflow to determine exactly how the problem should be solved.

OpenMDAO provides a library of sparse solvers and optimizers designed to work with its distributed-memory, sparse data-passing scheme. OpenMDAO ships with a numpy data-passing implementation that allows to get up and running quickly and provides efficient operation for serial execution. It also comes with an MPI-based implementation that

gives the possibility to run in parallel on a multi-core processor and high performance computing (HPC) environments.

In the optimization process, OpenMDAO works with both gradient-free (e.g., genetic algorithm, particle swarm) and gradient-based methods. With OpenMDAO it is possible to compute system-level gradients for Newton solvers and/or gradient-based optimizers by using its automatic analytic multidisciplinary derivatives. The user can provide analytic derivatives for components and OpenMDAO will compute the system-level gradients using the chain-rule across the entire model. It is not necessary to provide analytic derivatives for all of the components as OpenMDAO will finite-difference across components that are missing derivatives. Note that some parameters can have analytic derivatives and others can be finite-differenced, allowing for a fast and easy way to gain a lot of computational efficiency.

1.5 Thesis outline

The objective of the present work is to improve an existing open-source code (*OpenAeroStruct* [36]) developed by the MDOlab (*Multidisciplinary Design Optimization Laboratory*) at the University of Michigan. *OpenAeroStruct* is a lightweight model to perform aerostructural optimization using OpenMDAO. It is essentially composed by two main parts:

- a structural model that consists of a one-dimensional beam and a structural solver that can solve for the static deformation under aerodynamic loads and can optimize the thickness of each element;
- an aerodynamic solver that is based on the classical Vortex Lattice Method (VLM) with horseshoe vortex elements. Only steady flows can be examined and it can optimize the wing twist angle, taper ratio, dihedral angle, and span.

The leading purpose of this work is to improve *OpenAeroStruct* in order to perform a dynamic aeroelastic analysis for the first time in OpenMDAO. The thesis is divided into six main chapters.

Chapter 2 provides a detailed insight into the development of the structural solver. The mass, damping, and stiffness matrices are implemented for the FEM beam model and then the eigenvalue problem is constructed, solved, and verified.

In Chapter 3, the aerodynamic solver is presented. The existing steady VLM is updated in order to describe unsteady effects of the wake on the wing in the time domain.

The objective of Chapter 4 is to present the structure of the code written in OpenMDAO. The principal innovation is the loop in the time that is constructed to perform a dynamic aeroelastic analysis.

In Chapter 5, the numerical results of this study are verified and discussed in-depth. Moreover, studies of convergence for principal parameters are analyzed.

Finally, we draw conclusions from the results presented and examine how the objectives have been fulfilled. Additionally, we provide some suggestions on how to improve the aerostructural tool as a basis for future work.

Chapter 2

Structural model

A structural component is usually studied by dividing them in simple parts with well known properties. Then, by considering how those parts are linked to others, it is possible to create a model that describes properties of the entire component. The arrangement of mathematical relationships describing the structure can be approached in the different ways; the finite difference method, the transfer method and and variational methods such as Ritz method.

The aim of this chapter is to introduce one of the most used methods by engineers: the Finite Element Method (FEM). In this work, the wing is modeled as a beam using one-dimensional (1D) elements. For this reason, we only present the element formulation for 1D finite elements. In the first part, the stiffness and mass matrices of the wing are obtained by these two main steps:

- the characterization of the 1D finite elements, i.e. the mathematical description of their kinematics in relation to their equilibrium and deformation conditions, and
- the construction of the structure, i.e. the mathematical formulation of the equations that express the element belonging to a structure.

In the second part, we perform a modal analysis of a spatial beam with a tube cross section.

2.1 Finite Element Method

The basic concepts of Finite Element Analysis (FEA) can be summarized as follows:

1. Model the continuum (solid or fluid) as a finite collection of discrete elements.
2. Make simplifying assumptions about the displacement field (or the stress field) within the elements and across element boundaries.
3. Minimize the error introduced by the assumptions in step 2 using a suitable method.

If step 2 approximates displacements, the method is called the displacement or stiffness method; if step 2 approximates stresses, the method is referred to as the force or flexibility method. In the modern context, step 3 in the FE formulation is often accomplished using a variational principle, e.g., Hamilton's Principle. In some cases, working directly with the governing differential equations and using the method of weighted residuals gives the best approximation. This produces a finite element model type called Galerkin. Finite element formulations are therefore classified into three different classes, according to the approach used:

- *Direct method*: based on basic structural principles and Newton's Laws
- *Variational method(s)*: based on minimizing or finding the stationary values of a function
- *Method of weighted residuals*: based directly on the governing differential equation(s)

2.2 Characterization of 1D finite elements

In this section, the derivation of element equations for one-dimensional structural elements is considered. We refer to an element initially straight, identified by its two end points (nodes) 1 and 2, through which it exchanges actions with the outside. The element is associated with a local reference system (x, y, z) in which the x-axis coincides with the axis of the element and is directed from node 1 to node 2; y and z axes are perpendicular to the x axis and coincide with the principal directions of inertia of the element cross section, as shown in Figure 2.1.

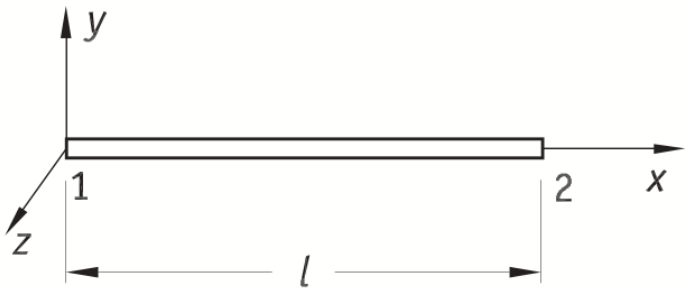


FIGURE 2.1: 1D finite element

These elements can be used for the analysis of systems such as planar trusses, spatial trusses, torsional bar, beams, grid systems, planar beams, and spatial beams [37]. A truss element (Section 2.2.1) is a bar that can only resist axial forces (compressive or tensile), can only deform in the axial direction, and is not able to carry transverse loads or bending moments. While a torsional bar (Section 2.2.2) can only rotate from applied torques around its longitudinal axis. In planar truss analysis, each of the two nodes can have components of displacement parallel to the x and y axes. In three-dimensional truss analysis, each node can have displacement components in the x, y, and z directions. A beam or a frame element (Section 2.2.3) is a bar that can resist axial forces, transverse loads, and bending moments. In the analysis of planar beams, each of the two nodes of an element will have two translational displacement components (parallel to x and y axes) and a rotational displacement (in the xy-plane). For a spatial beam element (Section 2.2.4), each of the two ends is assumed to have three translational displacement components (parallel to x, y, and z axes) and three rotational displacement components (one in each of the xy-, yz-, and zx-planes). Additionally, throughout this work, the structural members are assumed to be uniform and linearly elastic.

2.2.1 Truss element

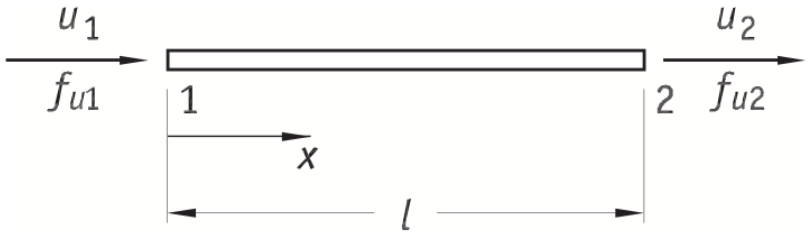


FIGURE 2.2: Truss bar element

Consider a finite element (displacement) formulation for the truss element in Figure 2.1. Substituting the Hooke's law and the strain-displacement relationship into force equilibrium for a truss with loads applied only at the ends, we obtain the differential equation that governs the truss element, as follows:

$$\sigma_x = E\epsilon_x \tag{2.1}$$

$$\epsilon_x = \frac{du}{dx} \tag{2.2}$$

$$A\sigma_x = f = constant \tag{2.3}$$

$$\frac{d}{dx}\left(AE\frac{du}{dx}\right) = 0 \tag{2.4}$$

In these equations the displacement u, the strain ϵ_x and the stress σ_x of the truss are all in the axial direction. In general, the longitudinal Young's modulus E and the element's sectional area A can change along the truss, but we will assume they are constant in the following derivations.

Now, let us assume that the displacement field can be approximated by a linear polynomial in x:

$$u(x) = a_1 + a_2 x = \begin{bmatrix} 1 & x \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \tag{2.5}$$

The generalized coordinates a_1 and a_2 can be evaluated in terms of end displacements, as follows:

$$\begin{aligned} u(0) &= u_1 = a_1 \\ u(l) &= u_2 = a_1 + a_2 l \quad \rightarrow \quad a_2 = (u_2 - u_1)/l \end{aligned}$$

In matrix form,

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1/l & 1/l \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.6}$$

Combining Equations 2.5 through 2.6 yields

$$u(x) = \begin{bmatrix} 1 & x \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1/l & 1/l \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} N_1(x) & N_2(x) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.7}$$

where the shape functions are

$$\begin{aligned} N_1(x) &= 1 - \tfrac{x}{l} \\ N_2(x) &= \tfrac{x}{l} \end{aligned} \tag{2.8}$$

Shape functions describe the shape of displacement field and are one of the main factors in governing the efficiency and accuracy of FEA.

By using Equations 2.2 and 2.1

$$\epsilon_x = \frac{du}{dx} = \begin{bmatrix} -1/l & 1/l \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.9}$$

$$\sigma_x = E\epsilon_x = \frac{E}{l} \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{E}{l}(-u_1 + u_2) \tag{2.10}$$

we obtain joint (nodal) forces:

$$[f] = \begin{bmatrix} f_{u1} \\ f_{u2} \end{bmatrix} = A\sigma_x \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \frac{AE}{l} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = [k]\,[u] \tag{2.11}$$

where the stiffness matrix for a truss bar element in local coordinates is

$$[k]_{truss} = \frac{AE}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{2.12}$$

The construction of the elemental mass matrix with distributed mass density can be carried out through different methods. The variational mass lumping formulation is presented and adopted in this thesis. This is done by taking the kinetic energy as part of the governing function. The kinetic energy of an element of mass density ρ that occupies the domain Ω and moves with velocity field $\tilde{v}$ is

$$T = \frac{1}{2}\int_{\Omega} \rho(\tilde{v})^T(\tilde{v})d\Omega \tag{2.13}$$

Following the conventional FEA philosophy, the element velocity field is interpolated using shape functions:

$$\tilde{v} = N_v\dot{u} \tag{2.14}$$

in which $\dot{u}$ are node velocities and N_v a shape function row vector. Inserting Equation 2.14 into Equation 2.15 and taking the node velocities out of the integral yields

$$T = \frac{1}{2}\dot{u}^T\int_{\Omega} \rho {N_v}^T N_v dV\dot{u} = \frac{1}{2}\dot{u}^T m\dot{u} \tag{2.15}$$

whence the element mass matrix is the Hessian of T:

$$m = \frac{\partial^2 T}{\partial\dot{u}\partial\dot{u}} = \int_{\Omega} \rho {N_v}^T N_v d\Omega \tag{2.16}$$

If we use the same shape functions used in the derivation of the stiffness matrix, $N_v = N$, the mass matrix defined in Equation 2.16 is called the consistent mass matrix or CMM, because the velocity field used in calculating the kinetic energy is consistent with the assumed displacement field.

Using the shape functions of Equation 2.8, the element mass matrix for the truss element in local coordinates is easily obtained as

$$[m]_{truss} = \int_0^l \rho A N^T N dx = \frac{\rho l A}{4}\int_0^1 \begin{bmatrix} 1-\frac{x}{l} \\ \frac{x}{l} \end{bmatrix}\begin{bmatrix} 1-\frac{x}{l} & \frac{x}{l} \end{bmatrix} dx = \frac{\rho A l}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{2.17}$$

2.2.2 Torsional bar

In this section we consider the torsional bar, which is another type of finite element. We must write angular moments as a function of displacements and rotations. The torsional bar is an element with only one degree of freedom, α_x (rotation around the x axe) for each node, as shown in Figure 2.3:

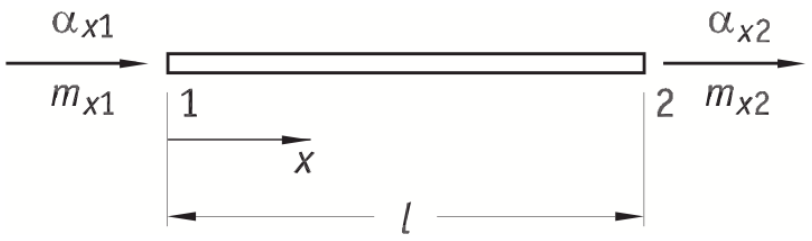


FIGURE 2.3: Torsional bar element

where m_{x1} and m_{x2} are the torques applied on nodes 1 and 2. Starting from the equilibrium equation

$$m_{x1} + m_{x2} = 0 \tag{2.18}$$

and the differential equation:

$$\frac{d\alpha}{dx} = \frac{m_x}{GJ} \tag{2.19}$$

where $d\alpha/dx$ is the axial rotation of the bar, m_x the torque that acts in the bar section, G the shear modulus of the material, and J the section rigidity. The cross-sectional rigidity for a general closed cross section is defined as

$$J = \frac{4A^2}{\int \frac{dl}{t}} \tag{2.20}$$

where dl is the infinitesimal length element along the section perimeter and t is its thickness. For a constant thickness tube, Equation 2.20 becomes

$$J = \frac{\pi D^3 t}{4} \tag{2.21}$$

Integrating Equation 2.19 along the bar length with m_x and G set constant with x, we obtain

$$\alpha_{x2} - \alpha_{x1} = \frac{l}{GJ} m_{x2} \tag{2.22}$$

Equations 2.18 and 2.22 can be rewritten in matrix form as follows:

$$\begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{x1} \\ \alpha_{x2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & \frac{l}{GJ} \end{bmatrix} \begin{bmatrix} m_{x1} \\ m_{x2} \end{bmatrix} \tag{2.23}$$

Simplifying the previous equation by multiplying through with the moments, we obtain:

$$\begin{bmatrix} m_{x1} \\ m_{x2} \end{bmatrix} = \frac{GJ}{l} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{x1} \\ \alpha_{x2} \end{bmatrix} \tag{2.24}$$

where the stiffness matrix of the torsion bar element is

$$[k]_{tors} = \frac{GJ}{l} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{2.25}$$

Following the same procedure presented in the previous section, we can obtain the mass matrix for a torsional bar:

$$[m]_{tors} = \frac{GI_x}{l} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{2.26}$$

where I_x is the moment of inertia of the section around the x axis.

2.2.3 Simple beam

As previously explained, a beam is a bar-like structural component whose primary function is to support transverse loading and transfer it to supports. Although beams are three-dimensional structures, we can approximate them as one-dimensional due to the bar-like geometry shown in Figure 2.4. This is one of the simplifications used in the development of classical beam theory, based on the Euler-Bernoulli assumption where shear deformations and shear strains are neglected. Other beam theories, such as Timoshenko beam theory, are more sophisticated.

Let the Young's modulus E, the section moment of inertia I_z around the z axis, and the mass density ρ be constant and consider flexure only (axial loads can be incorporated via the truss element just considered). A simple beam element has 2 degrees of freedom

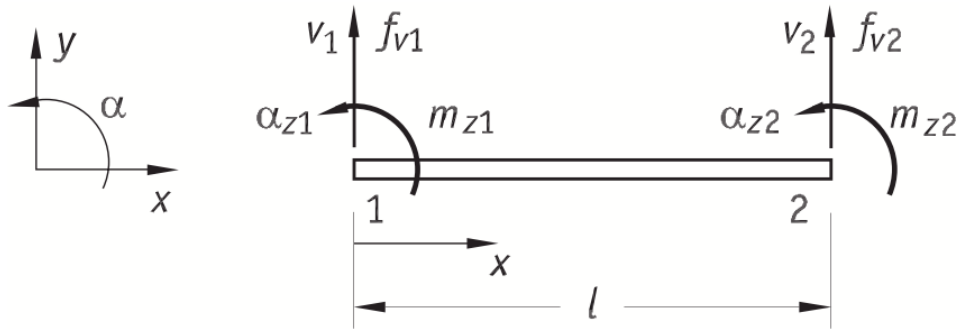


FIGURE 2.4: Simple beam element

(DOF) for each node (displacement v in the y direction and the rotation α_z around the z axis), or 4 DOFs in total.

Assume a displacement field of the form:

$$v(x,t) = a_1 + a_2 x + a_3 x^2 + a_4 x^3 = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = [\phi]^T [a] \tag{2.27}$$

which is exact for a statically loaded beam with end forces only. That is, the assumed displacement field satisfies the equilibrium equation from statics,

$$EI_z \frac{\partial^4 v}{\partial x^4} = 0 \tag{2.28}$$

In the dynamic case, the constants a_i are functions of time and have the role of generalized coordinates. They can be related to the nodal (end) displacements as follows:

$$\begin{aligned} v_1 &= v(0,t) = a_1 \\ \alpha_{z1} &= v'(0,t) = a_1 + a_2 l + a_3 l^2 + a_4 l^3 \\ v_2 &= v(l,t) = a_1 \\ \alpha_{z2} &= v(l,t) = a_2 + 2a_3 l + 3a_4 l^2 \end{aligned} \tag{2.29}$$

where small displacements and small slopes have been assumed. In matrix form, Equation 2.29 can be expressed as

$$[q] = \begin{bmatrix} v_1 \\ \alpha_{z1} \\ v_2 \\ \alpha_{z2} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & l & l^2 & l^3 \\ 0 & 1 & 2l & 3l^2 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = [H]\,[a] \tag{2.30}$$

Solving for a_i:

$$[a] = [H]^{-1}\,[q] \tag{2.31}$$

where

$$[H]^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3/l^2 & -2/l & 3/l^2 & -1/l \\ 2/l^3 & 1/l^2 & -2/l^3 & 1/l^2 \end{bmatrix} \tag{2.32}$$

Back-substituting Equations 2.31 and 2.32 into Equation 2.27 yields $w(x,t)$ in terms of the nodal displacements:

$$w(x,t) = [\phi]^T\,[H]^{-1}\,[q] = [N(x)]^T\,[q] \tag{2.33}$$

where

$$\begin{aligned} N_1(x) &= 1 - 3(x/l)^2 + 2(x/l)^3 \\ N_2(x) &= x - 2x^2/l + x^3/l^2 \\ N_3(x) &= 3(x/l)^2 - 2(x/l)^3 \\ N_4(x) &= -x^2/l + x^3/l^2 \end{aligned} \tag{2.34}$$

are shape or interpolation functions associated with the vector of coordinates q_i, which yields a detailed form of Equation 2.33

$$w(x,t) = N_1(x)v_1 + N_2(x)\alpha_{z1} + N_3(x)v_2 + N_4(x)\alpha_{z2} \tag{2.35}$$

The $N_i(x)$ functions are the so-called Hermite (cubic) polynomials: they are cubic functions that are defined in terms of the values of the function and its first derivative at the two end points of the interval $[0, l]$.

Now we can present a third way to build the stiffness matrix, by starting from the definition of strain energy in the beam:

$$U = \frac{1}{2}\int_0^l EI_z \left(\frac{\partial^2 v}{\partial x^2}\right)^2 dx \tag{2.36}$$

and

$$\frac{\partial^2 v}{\partial x^2} = \left[\phi''\right]^T [H]^{-1} [q] = \left[N''\right]^T [q] = [q]^T \left[N''\right] \tag{2.37}$$

$$\left(\frac{\partial^2 v}{\partial x^2}\right)^2 = [q]^T \left[N''\right] \left[N''\right]^T [q] = [q]^T \left([H]^{-1}\right)^T \left[\phi''\right] \left[\phi''\right]^T [H]^{-1} [q] \tag{2.38}$$

The strain energy can now be expressed in a form that allows direct identification of the stiffness matrix:

$$U = \frac{1}{2} [q]^T \left(\int_0^l EI_z \left[N''\right] \left[N''\right]^T dx\right) [q] = \frac{1}{2} [q]^T [k]_{elem} [q] \tag{2.39}$$

The stiffness matrix for the beam element is therefore

$$[k]_{beam} = \int_0^l EI_z \left[N''(x)\right] \left[N''(x)\right]^T dx \tag{2.40}$$

Proceeding, the elements of $[k]$ are readily evaluated, using Equation 2.40

$$k_{ij} = \frac{\partial^2 U}{\partial q_i \partial q_j} = \int_0^l EIN_i''(x)N_j''(x)dx \tag{2.41}$$

Substituting using Equations 2.34 and carrying out the integrations, we obtain the 4x4 stiffness matrix for a Euler-Bernoulli beam element:

$$[k]_{beam} = \frac{EI_z}{l^3} \begin{bmatrix} 12 & 6l & -12 & 6l \\ 6l & 4l^2 & -6l & 2l^2 \\ -12 & -6l & 12 & -6l \\ 6l & 2l^2 & -6l & 4l^2 \end{bmatrix} \tag{2.42}$$

We can use the same energetic method explained in Section 2.2.1 to obtain the mass matrix. However, we must take into account that for a slender beam, the kinetic energy contributed by the angular velocity of the cross section can be neglected. Starting from Equation 2.16

$$[m]_{elem} = \rho A \int_0^l [N(x)]^T [N(x)] \, dx \tag{2.43}$$

with elements

$$m_{ij} = \frac{\partial^2 T}{\partial \dot{q}_i \partial \dot{q}_j} = \rho A \int_0^l N_i(x) N_j(x) dx \tag{2.44}$$

Evaluating the integrals in Equation 2.44 yields the consistent mass matrix for the slender Euler-Bernoulli beam element:

$$[m]_{beam} = \frac{\rho A l}{420} \begin{bmatrix} 156 & 22l & 54 & -13l \\ 22l & 4l^2 & 13l & -3l^2 \\ 54 & 13l & 156 & -22l \\ -13l & -3l^2 & -22l & 4l^2 \end{bmatrix} \tag{2.45}$$

2.2.4 Spatial beam

A spatial beam element models a straight bar of an arbitrary cross section, which can deform not only in the axial direction but also in the directions perpendicular to the axis of the bar. The bar is capable of carrying both axial and transverse forces as well as moments. Therefore, a spatial beam element possesses the properties of both truss and simple beam elements. Spatial beam elements are found in most of our real world structural problems because there few structures that deform and carry loadings purely in axial or transverse directions. Lastly, because spatial beams are the most general form of element with a one-dimensional geometry, they are used in the *OpenAeroStruct* code.

A spatial beam element is modeled with 6 DOFs for each node: there are three translational displacements in the x, y and z directions, and three rotations with respect to the x, y and z axes. Therefore, for an element with two nodes, there are 12 DOFs in total, as shown in Figure 2.5. The element displacement vector for a beam element in spatial can be written as

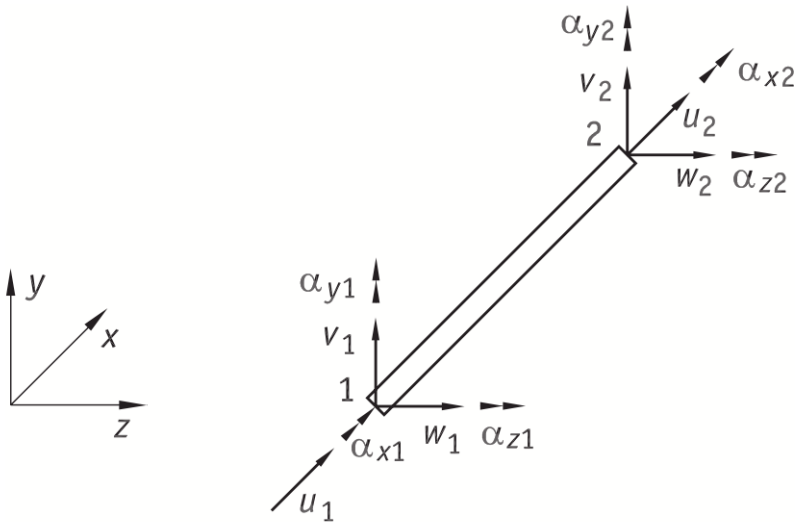


FIGURE 2.5: Spatial beam element

$$[d]_e = \begin{bmatrix} u_1 \\ v_1 \\ w_1 \\ \alpha_{x1} \\ \alpha_{y1} \\ \alpha_{z1} \\ u_2 \\ v_2 \\ w_2 \\ \alpha_{x2} \\ \alpha_{y2} \\ \alpha_{z2} \end{bmatrix} \tag{2.46}$$

As the y and z axes coincide with the principal axes of inertia, shear and flexural moments in both planes xy and xz can be considered independents. Therefore there is a considerable simplification in the matrix formulation. In fact mass and stiffness matrices for a spatial beam element can be obtained by superimposing the matrices of truss (Equations 2.12, 2.17), torsional bar (Equations 2.25, 2.26) and beams in xy (Equations 2.40, 2.45) and xz plans. Because of the huge matrices involved, the details will not be shown, but the 12x12 stiffness and consistent mass matrices are listed here as follows, and can be easily confirmed simply by inspection:

.

$$[k]_e = \begin{bmatrix} k_1 & 0 & 0 & 0 & 0 & 0 & -k_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12k_2^z & 0 & 0 & 0 & 6k_2^z l & 0 & -12k_2^z & 0 & 0 & 0 & 6k_2^z l \\ 0 & 0 & 12k_2^y & 0 & -6k_2^y l & 0 & 0 & 0 & -12k_2^y & 0 & -6k_2^y l & 0 \\ 0 & 0 & 0 & k_3 & 0 & 0 & 0 & 0 & 0 & -k_3 & 0 & 0 \\ 0 & 0 & -6k_2^y l & 0 & 4k_2^y l^2 & 0 & 0 & 0 & 6k_2^y l & 0 & 2k_2^y l^2 & 0 \\ 0 & 6k_2^z l & 0 & 0 & 0 & 4k_2^z l^2 & 0 & -6k_2^z l & 0 & 0 & 0 & 2k_2^z l^2 \\ -k_1 & 0 & 0 & 0 & 0 & 0 & k_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -12k_2^z & 0 & 0 & 0 & -6k_2^z l & 0 & 12k_2^z & 0 & 0 & 0 & -6k_2^z l \\ 0 & 0 & -12k_2^y & 0 & 6k_2^y l & 0 & 0 & 0 & 12k_2^y & 0 & 6k_2^y l & 0 \\ 0 & 0 & 0 & -k_3 & 0 & 0 & 0 & 0 & 0 & k_3 & 0 & 0 \\ 0 & 0 & -6k_2^y l & 0 & 2k_2^y l^2 & 0 & 0 & 0 & 6k_2^y l & 0 & 4k_2^y l^2 & 0 \\ 0 & 6k_2^z l & 0 & 0 & 0 & 2k_2^z l^2 & 0 & -6k_2^z l & 0 & 0 & 0 & 4k_2^z l^2 \end{bmatrix} \tag{2.47}$$

where

$$k_1 = \frac{EA}{l} \quad , \quad k_2^z = \frac{EI_z}{l^3} \quad , \quad k_2^y = \frac{EI_y}{l^3} \quad , \quad k_3 = \frac{GJ}{l} \tag{2.48}$$

and

$$[m]_e = \frac{\rho Al}{420} \begin{bmatrix} 140 & 0 & 0 & 0 & 0 & 0 & 70 & 0 & 0 & 0 & 0 & 0 \\ 0 & 156 & 0 & 0 & 0 & 22l & 0 & 54 & 0 & 0 & 0 & -13l \\ 0 & 0 & 156 & 0 & -22l & 0 & 0 & 0 & 54 & 0 & 13l & 0 \\ 0 & 0 & 0 & 140m_t & 0 & 0 & 0 & 0 & 0 & 70m_t & 0 & 0 \\ 0 & 0 & -22l & 0 & 4l^2 & 0 & 0 & 0 & -13l & 0 & -3l^2 & 0 \\ 0 & 22l & 0 & 0 & 0 & 4l^2 & 0 & 13l & 0 & 0 & 0 & -3l^2 \\ 70 & 0 & 0 & 0 & 0 & 0 & 140 & 0 & 0 & 0 & 0 & 0 \\ 0 & 54 & 0 & 0 & 0 & 13l & 0 & 156 & 0 & 0 & 0 & -22l \\ 0 & 0 & 54 & 0 & -13l & 0 & 0 & 0 & 156 & 0 & 22l & 0 \\ 0 & 0 & 0 & 70m_t & 0 & 0 & 0 & 0 & 0 & 140m_t & 0 & 0 \\ 0 & 0 & 13l & 0 & -3l^2 & 0 & 0 & 0 & 22l & 0 & 4l^2 & 0 \\ 0 & -13l & 0 & 0 & 0 & -3l^2 & 0 & -22l & 0 & 0 & 0 & 4l^2 \end{bmatrix} \tag{2.49}$$

with

$$m_t = \frac{I_x}{A} \tag{2.50}$$

2.3 Assembly of global matrices

The matrices formulated in the previous section are for a particular beam element in a specific orientation. A full structure usually comprises numerous beam elements of different orientations joined together. As such, their local coordinate system would vary from one orientation to another. To assemble the element matrices together, all the matrices must first be expressed in a common coordinate system, the global coordinate system.

The coordinate transformation matrix T gives the relationship between the displacement vector d_e (Equation 2.46) based on the local coordinate system and the displacement vector D_e for the same element, but based on the global coordinate system:

$$[d]_e = [T]\,[D]_e \tag{2.51}$$

Following Reference [38], T is defined as follows:

$$[T] = \begin{bmatrix} T_3 & 0 & 0 & 0 \\ 0 & T_3 & 0 & 0 \\ 0 & 0 & T_3 & 0 \\ 0 & 0 & 0 & T_3 \end{bmatrix} \tag{2.52}$$

$$[T_3] = \begin{bmatrix} l_x & m_x & n_x \\ l_y & m_y & n_y \\ l_z & m_z & n_z \end{bmatrix} \tag{2.53}$$

The components of the sub-matrix T_3 are the "direction cosines" that link the two orthonormal basis vectors of local and global coordinate systems. In particular, by adopting lowercase letters for the local coordinates vector (x, y, z) and uppercase letters for the global one (X, Y, Z), we have:

$$\begin{array}{lll} l_x = cos(x,X) & m_x = cos(x,Y) & n_x = cos(x,Z) \\ l_y = cos(y,X) & m_y = cos(y,Y) & n_y = cos(y,Z) \\ l_z = cos(z,X) & m_z = cos(z,Y) & n_z = cos(z,Z) \end{array} \tag{2.54}$$

At this point, we can build the stiffness matrix of a generic spatial beam in the global coordinate system, which is obtained from the static equilibrium equation written in the local system.

$$[k]_e\,[d]_e = [f]_e \tag{2.55}$$

The relation in Equation 2.51 is applicable to any vector. For the force vector:

$$[f]_e = [T]\,[F]_e \tag{2.56}$$

Substituting Equation 2.51 and Equation 2.56 in Equation 2.55, we obtain

$$[k]_e\,[T]\,[D]_e = [T]\,[F]_e \tag{2.57}$$

Pre-multiplying both members of Equation 2.57 by $[T]^{-1}$ and comparing it with the static equation in global coordinates, we obtain the stiffness matrix in the global coordinate system for a generic spatial beam element:

$$[T]^{-1}\,[k]_e\,[T]\,[D]_e = [F]_e \tag{2.58}$$

$$[K]_e = [T]^{-1}\,[k]_e\,[T] \tag{2.59}$$

In the same way, it is obtained the mass matrix in the global coordinate system:

$$[M]_e = [T]^{-1}\,[m]_e\,[T] \tag{2.60}$$

In practice, in order to obtain global matrices, we have to pre-multiply and multiply the element local matrices by $[T]^{-1}$ and $[T]$ respectively.

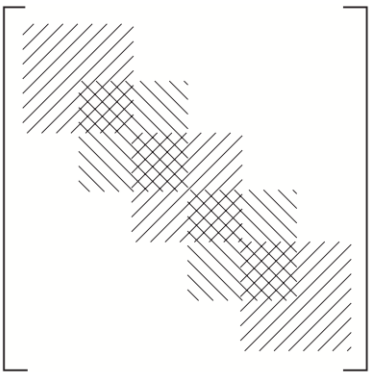


FIGURE 2.6: Assembly of global structural matrix

Once the matrices of all structural elements are in the same system of reference, we superimpose them to build the stiffness and mass matrices of the entire structure. Considering spatial beam elements, the displacement vector contains 6 DOFs for each node

of the structure. For example, dividing the beam in 10 finite elements gives 11 nodes and then 66 total DOFs. The dimension of the displacement vector for this structure is 66, while the dimension of stiffness and mass matrices is 66 x 66.

The resultant global mass and stiffness matrices are typically banded matrices as shown in Figure 2.6, where cross-hatched regions indicate where elements of sub-matrices add algebraically at a node. The ordering of the nodes (coordinates) influences the bandwidth, which is 6 in this case. Large-scale FE programs usually have bandwidth minimization routines that reorder the coordinates in an optimal manner. This is important in large problems because special algorithms have been developed for inverting and processing these sparse matrices.

2.4 Modal analysis

In this section we introduce the normal mode method, which decouples the system by taking advantage of the orthogonality conditions of a structural dynamic system.

We start from the dynamic equation for the undamped structure:

$$[M]\left[\ddot{d}\right] + [K]\,[d] = [F(t)] \tag{2.61}$$

where the vector of applied external forces $[F]$ in general varies in time.

2.4.1 Unforced response problem

Free vibration of a structure is its dynamic response in the absence of external forces, that is, when $[F(t)] = 0$. The governing equations for the structure in this case become homogeneous, so

$$[M]\left[\ddot{d}\right] + [K]\,[d] = 0 \tag{2.62}$$

To solve this set of partial differential equations we assume a solution of the form

$$[d] = \left[\bar{d}\right] e^{i\omega t} \tag{2.63}$$

Then we have also

$$\left[\ddot{d}\right] = -\omega^2 \left[\bar{d}\right] e^{i\omega t} \tag{2.64}$$

Substituting Equations 2.63-2.64 into the governing equations

$$-\omega^2\,[M]\left[\bar{d}\right] e^{i\omega t} + [K]\left[\bar{d}\right] e^{i\omega t} = 0 \tag{2.65}$$

$$[K]\left[\bar{d}\right] = \omega^2\,[M]\left[\bar{d}\right] \tag{2.66}$$

The set of n equations in Equation 2.66 is a generalized eigenvalue problem. Solving this set of equations yields n eigenvalues λ_i, $(i = 1, ..., n)$ and corresponding eigenvectors $[\bar{d}_i]$. We solve these equations by setting the following determinant to zero:

$$|[K] - \lambda [M]| = 0 \tag{2.67}$$

This equation has n roots, corresponding to the eigenvalues λ_i. The eigenvalues for this kind of problem are real and greater of equal than zero, since the mass matrix is symmetric and positive definite, and the stiffness matrix is also symmetric and at least positive semi-definite.

We can then substitute the eigenvalues into the eigenvalue problem:

$$|[K] - \lambda [M]| \left[\bar{d}\right] = 0 \tag{2.68}$$

Solving Equation 2.68 for each eigenvalue λ_i yields the corresponding eigenvector.

The eigenvalues are directly related to the natural vibration frequencies of the structure

$$\omega_i = \sqrt{\lambda_i} \quad , \qquad i = 1, 2, ..., n \tag{2.69}$$

Physically, for each frequency ω_i there is a mode shape $[\bar{d}_i]$, which is the shape of the structure when it vibrates at that frequency. The time history of the displacements is then given by

$$[d(t)] = \left[\bar{d}\right] e^{i\omega_i t} \tag{2.70}$$

If we put the eigenvectors in a matrix by columns, we build the "modal matrix" $[\Phi]$, which has the property of diagonalizing the mass matrix $[M]$ and the stiffness matrix $[K]$, due to the orthogonality conditions with respect to $[M]$ and $[K]$. Thus

$$[\Phi]^T [M] [\Phi] = [M_{rr}] \tag{2.71}$$

$$[\Phi]^T [K] [\Phi] = [K_{rr}] \tag{2.72}$$

where matrices $[M_{rr}]$ and $[K_{rr}]$ are diagonal.

From Equations 2.71-2.72 and Equation 2.66 it is clear that

$$[K_{rr}] = \omega_r^2 [M_{rr}] \tag{2.73}$$

2.4.2 Forced response problem

Based on the results of the previous section, the forced response of Equation 2.79 can be conveniently obtained by using the normal modes as the appropriate coordinate system. Introducing the transformation

$$[d] = [\Phi]\,[\eta] \tag{2.74}$$

Thus, by substituting Equation 2.74 in Equation 2.79:

$$[M]\,[\Phi]\,[\ddot{\eta}] + [K]\,[\Phi]\,[\eta] = [F(t)] \tag{2.75}$$

Now, we pre-multiply Equation 2.75 by $[\Phi]^T$

$$[\Phi]^T\,[M]\,[\Phi]\,[\ddot{\eta}] + [\Phi]^T\,[K]\,[\Phi]\,[\eta] = [\Phi]^T\,[F(t)] \tag{2.76}$$

and define

$$[\Phi]^T\,[F(t)] = [F_r(t)] \tag{2.77}$$

From Equations 2.71, 2.72, 2.73, 2.76 and 2.77 we finally obtain

$$[M_{rr}]\,[\ddot{\eta}] + \left[\omega_r^2 M_{rr}\right]\,[\eta] = [F_r(t)] \tag{2.78}$$

which means that the response of the system is governed by an uncoupled system of linear, second order differential equations with constant coefficients, which is a simple problem from a mathematical point of view. Thus the response is obtained by solving the following set of equations

$$M_{rr}\ddot{\eta}_r + \omega_r^2 M_{rr}\eta_r = F_r(t) \quad , \qquad r = 1, 2, ..., n \tag{2.79}$$

where $F_r(t)$ is the generalized excitation in any mode r.

In conclusion, since the system is linear and the principle of superposition applies, knowledge of $\eta_r(t)$, $r = 1, 2, ..., n$ enables one to obtain the response from Equation 2.74:

$$[d(t)] = [\Phi]\,[\eta(t)] \tag{2.80}$$

2.5 Damping matrix

It is possible to apply the same procedure explained in Section 2.4 to a damped system:

$$[M]\left[\ddot{d}\right] + [C]\left[\dot{d}\right] + [K][d] = [F(t)] \tag{2.81}$$

The normal mode transformation can be used again, however the success of the method in decoupling the system now depends on the properties of the damping matrix

$$[C_{rr}] = [\Phi]^T [C] [\Phi] \tag{2.82}$$

If the damping matrix $[C]$ is diagonal or can be assumed to be such, the response problem decouples again. A nondiagonal $[C]$ matrix requires a somewhat more complicated treatment of the dynamic response problem.

In this work, $[C]$ is assumed to be a linear combination of the mass and stiffness matrix:

$$[C] = \alpha [M] + \beta [K] \tag{2.83}$$

A damping matrix of this form is called a proportional damping matrix and this assumption was first introduced by Rayleigh [39]. A combination of Equations 2.82 through 2.83 yields

$$[C_{rr}] = [\Phi]^T [C] [\Phi] = [\Phi]^T [\alpha [M] + \beta [K]] [\Phi] = \left(\alpha [I] + \beta \left[\omega_r^2\right]\right) \tag{2.84}$$

and Equation 2.84 decouples into n equations of the form

$$\ddot{\eta_r} + \left(\alpha + \beta\omega_r^2\right)\dot{\eta_r} + \omega_r^2 \eta_r = F_r(t) \tag{2.85}$$

Thus, it is again possible to consider the response in each degree of freedom independently.

In order to evaluate the values of constants α and β, let's consider a 1-DOF system with a given amount ζ of modal critical damping associated with each mode

$$\ddot{\eta} + 2\zeta\omega\dot{\eta} + \omega^2\eta = F(t) \tag{2.86}$$

Comparing Equations 2.85 and 2.86 it is clear that

$$\zeta_j = \frac{\left(\alpha + \beta\omega_j^2\right)}{2\omega_j} \tag{2.87}$$

While the one-to-one correspondence between modal critical damping ratios and the constants of proportionality α and β can be obtained easily for a 2-DOF system, it can be quite problematic for the n-DOF case. In fact, if modal damping ζ_1, ζ_2,..., ζ_n is obtained from some experimental method, one will have in general n quantities (or equations) from which two parameters α and β have to be determined. This is an overdetermined problem and more sophisticated mathematical methods such as least squares techniques have to be used to determine the values of these two parameters for each equation of the system [40].

For this work, in order to compute the damping matrix of the system, a simplification has been made. In fact, we choose a mass proportional damping, as follows:

$$[C] = 2\frac{\zeta}{100}\bar{\omega}\,[M] \tag{2.88}$$

where ζ is the percent damping factor equal for all the system, and $\bar{\omega}$ is the first natural frequency (in rad/s) that acts in the correspondent direction of the considered degree of freedom. In practice, for all the lines relatives at the axial degree of freedom (u), we use the first natural frequency acting in the x direction. The same, for all the lines that have to multiply v or w, we use the first bending frequency of the system for both (because the simmetry), etc.

2.6 Newmark-Beta method

In this section we introduce an implicit time integration method for solving the forced differential system in Equation2.81. The Newmark-beta method is a generalized modification of the original Newmark method [41]. It is widely used in numerical evaluation of the dynamic response of structures and solids such as in finite element analysis to model dynamic systems.

2.6.1 Mean value theorem

The Newmark-beta method is based on the mean value theorem, which states that if a function f is continuous on the closed interval $[a, b]$, where $a < b$, and differentiable on the open interval (a, b), then there exist a point c in (a, b) such that [42]:

$$f'(c) = \frac{f(b) - f(a)}{b - a} \tag{2.89}$$

2.6.2 Numerical integration scheme

The Newmark beta equations for displacement and velocity at time $t + \Delta t$ are:

$$[d]_{t+\Delta t} = [d]_t + [\dot{d}]_t \Delta t + \left[(1-\beta)\,[\ddot{d}]_t + \beta[\ddot{d}]_{t+\Delta t}\right] \Delta t^2 \tag{2.90a}$$

$$[\dot{d}]_{t+\Delta t} = [\dot{d}]_t + \left[(1-\alpha)\,[\ddot{d}]_t + \alpha[\ddot{d}]_{t+\Delta t}\right] \Delta t \tag{2.90b}$$

where α and β are parameters that control the stability and accuracy of the method. In the original method $\alpha = 1/2$ and $\beta = 1/4$. We obtain the recurrence formula by writing Equation2.81 at time $t + \Delta t$, and substituting for $[\ddot{d}]_{t+\Delta t}$ and $[\dot{d}]_{t+\Delta t}$:

$$[\ddot{d}]_{t+\Delta t} + [C]\,[\dot{d}]_{t+\Delta t} + [K]\,[d]_{t+\Delta t} = [F(t + \Delta t)] \tag{2.91}$$

We first solve for $[\ddot{d}]_{t+\Delta t}$ in terms of $[d]_{t+\Delta t}$ and other vectors by using Equation 2.90a. Next we insert the result into equations 2.90b to express $[\dot{d}]_{t+\Delta t}$ in terms of $[d]_{t+\Delta t}$ and other vectors. Finally, substituting for $[\ddot{d}]_{t+\Delta t}$ and $[\dot{d}]_{t+\Delta t}$ in the equation of motion permits us to write the recurrence formula in terms of an effective stiffness and load vector.

$$[\bar{K}]\,[d]_{t+\Delta t} = [\bar{F}]_{t+\Delta t} \tag{2.92}$$

where the effective stiffness matrix and the effective load vector are:

$$[\bar{K}] = [K] + \frac{\alpha}{\beta\Delta t}[C] + \frac{1}{\beta\Delta t^2}[M] \tag{2.93a}$$

$$\begin{aligned}[\bar{F}] = [F]_{t+\Delta t} + [C]\left[\tfrac{1}{\beta\Delta t}[d]_t + \left(\tfrac{\alpha}{\beta} - 1\right)[\dot{d}]_t + \tfrac{\Delta t}{2}\left(\tfrac{\alpha}{\beta} - 2\right)[\ddot{d}]_t\right] \\ + [M]\left[\tfrac{1}{\beta\Delta t^2}[d]_t + \tfrac{1}{\beta\Delta t^2}[\dot{d}]_t + \left(\tfrac{1}{2\beta} - 1\right)[\ddot{d}]_t\right]\end{aligned} \tag{2.93b}$$

In Equation 2.92, $[d]_{t+\Delta t}$ on the left-hand side is unknown and all the terms on the right-hand side are known, hence Equation 2.92 represents the recurrence formula for computing $[d]_{t+\Delta t}$ at the end of a time step. Note that the effective stiffness matrix $[\bar{K}]$ given in Equation 2.93a depends on the system stiffness matrix, hence we cannot render $[\bar{K}]$ diagonal by lumping the damping and mass matrices, as for example it is possible with the central difference recurrence formula. Consequently, the Newmark algorithm is implicit, and we must solve a set of simultaneous equations at each time step.

The performance of the Newmark beta algorithm has been studied extensively. We know the algorithm to be unconditionally stable for $\alpha = 1/2$ and $\beta = 1/4$, as in the original method. For this reason, in this work we these α and β values. Thus we may select the time step Δt in the Newmark algorithm without concern for solution stability, but we must consider the effects of time step size on solution accuracy. The basic guideline is that the time step should be small enough so that the response in all modes that contribute significantly to the total response is calculated accurately. The selection of a time step in practical problems often represents a compromise between solution accuracy and computational expense.

Chapter 3

Vortex Lattice Method

This chapter presents the Vortex Lattice Method (VLM). Potential flow theory is introduced after a brief introduction to the basic concepts of fluid dynamics. The first part of the chapter concludes with a description of the steady VLM formulation based on vortex rings. The second part covers all aspects of the Unsteady Vortex Lattice Method (UVLM), following the exhaustive description provided by Katz and Plotkin in Reference [3].

3.1 Basics of fluid dynamics

A flow field can be classified in several ways. From a classical point of view, a flow field can be either viscous or inviscid, compressible or incompressible, or unsteady or steady [43].

The dynamic (shear) viscosity of a fluid expresses its resistance to shearing flows, where adjacent layers move parallel to each other with different speeds. For aerodynamic purposes, a flow is considered viscous when it is close to a solid boundary, like the wing of an airplane or in the wake downstream the airplane. The viscous flow region by solid boundaries is called a boundary layer.

A flow with constant density is called incompressible. In contrast, a flow where the density is variable is called compressible [5]. All aeronautical flow fields are compressible, but compressibility effects are only considered when the ratio between the local velocity of the fluid and the local speed of sound (Mach number) reaches a certain point, usually Mach 0.3.

Every flow field is also time-dependent, but there are some conditions that the fluid flow must satisfy for these effects to be important. The selection of a reference system is the most important. A body moving relative to a fixed reference system (e.g. Earth) always

generates an unsteady flow while the same flow field in a fixed-body reference system can produce a steady flow field. The state and motion of the fluid can be characterized by its velocity, pressure, and temperature. Hence, the flow is steady if all these entities are independent of time in all points of the chosen reference system.

3.1.1 Kinematic concepts

The kinematics of flow fields is geometrically described [43] with concepts such as:

- Pathline: is the path taken by a volume-less fluid particle
- Streamline: a line in the flow field with the property that the fluid velocity vector is pointing in its tangent direction. If flow is steady, pathlines and streamlines coincide.
- Stream tube: real or imaginary tube in the flow field bounded by streamlines
- Stream tube flow: 1D approximation in which the velocity, the density, the pressure are constant over the stream tube sections of the coordinate s along the stream tube (and the time)

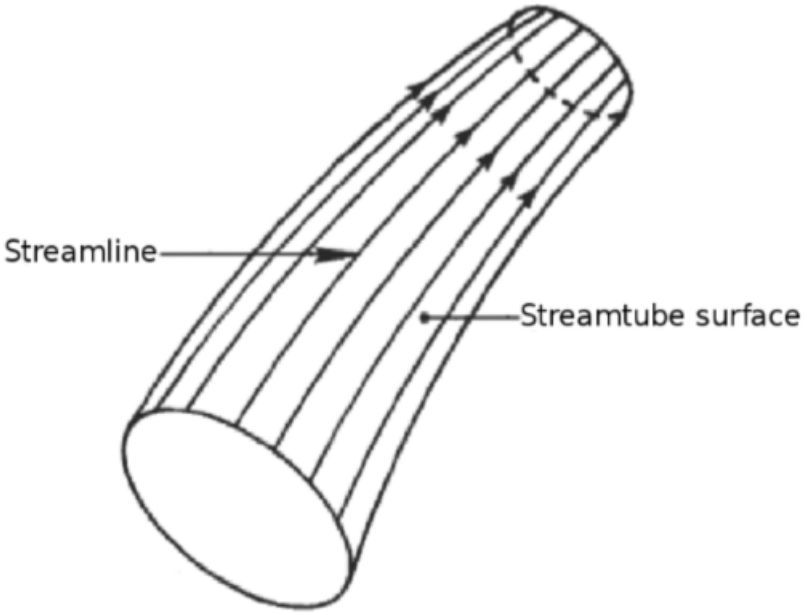


FIGURE 3.1: Cross section of a stream tube

3.1.2 The continuity equation

In fluid dynamics, the continuity equation states that, in any steady state process, the rate at which mass enters a system is equal to the rate at which mass leaves the system [44]. This is stated by the conservation of mass or continuity equation, which can be written in its general form as follows:

$$\frac{\partial \rho}{\partial t} + \vec{\nabla}(\rho \vec{q}) = 0 \tag{3.1}$$

where

- ρ is fluid density
- t is time
- $\vec{q}$ is the flow velocity vector field

$$\vec{q} = [u, v, w] \tag{3.2}$$

- $\vec{\nabla}$ is the flow velocity gradient operator, which, in Cartesian coordinates, is

$$\vec{\nabla} = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right] \tag{3.3}$$

In this context, this equation is also one of the Euler equations. Integrating Equation 3.1, we obtain its reduced form:

$$\rho(s)\vec{q}(s)A(s) = constant \tag{3.4}$$

with $A(s)$ that indicates the stream flow tube area in the section determined by the path coordinate s.

If ρ is a constant, as in the case of incompressible flow, the mass continuity equation simplifies to a volume continuity equation:

$$\vec{\nabla} \cdot \vec{q} = 0 \tag{3.5}$$

which means that the divergence of velocity field is zero everywhere. Physically, this is equivalent to saying that the local volume dilation rate is zero.

3.1.3 Bernoulli equation

The Bernoulli equation describes the behavior of a fluid moving along a streamline. It is obtained by integrating the Euler equations along a streamline for an inviscid, steady, and incompressible flow [45] and can be written in the traditional form as:

$$p + \frac{1}{2}\rho q^2 + \rho g z = constant\ along\ the\ streamline \tag{3.6}$$

Usually, the variation of the potential energy term $\rho g z$ can be neglected for air flow [46]. The physical meaning of the Bernoulli equation is clear; there is an inverse relation between the stream velocity module q and the pressure p.

This equation allows to calculate the forces in many aerodynamic problems. In Section 3.4 its unsteady form is posed.

3.2 Potential Flow Theory

3.2.1 Laplace's equation

We have seen that for an incompressible and inviscid flow, the continuity equation (Equation 3.5) is valid. Further, considering an irrotational flow, a velocity potential can be introduced:

$$\vec{q} = \vec{\nabla}\Phi \tag{3.7}$$

Hence, for an incompressible and irrotational flow, the following equation holds:

$$\vec{\nabla}^2\Phi = 0 \tag{3.8}$$

Equation 3.8 is known as Laplace's equation, which is a second order linear equation. The characteristics associated with linear equations make Laplace's useful for most of aerodynamic problems. It can be applied when the flow satisfies the potential conditions such that a velocity potential function exists.

One of the key features of Laplace's qquation is that it allows the equation governing the flow field to be converted from a 3D problem to a 2D problem by finding the potential on the surface. The solution is then found using this property by distributing singularities of unknown strength over discretized portions (panels) of the surface. Hence the flow field solution is found by representing the surface by a number of panels and solving a linear system of algebraic equations to determine the unknown strengths of the singularities.

For the Laplace's equation, a number of solutions can be found. Anderson [47] expounds this fact and states that:

> *"A complicated flow pattern for an irrotational, incompressible flow can be synthesized by adding together a number of elementary flows, which are also irrotational and incompressible."*

Such elementary flows may be a point source, a point sink, a doublet, or a vortex line. These may be superimposed in many ways including the formation of line sources, vortex sheets, etc. Today, the Vortex Lattice Method is one of a wide variety of methods that is based on these theories.

3.2.2 Kelvin's circulation theorem

Let C be a fluid curve in an incompressible, inviscid flow field (Figure 3.2). The circulation is defined as the line integral of the tangential velocity around a closed loop enclosing the airfoil [47].

$$\Gamma = \oint \vec{q} d\vec{s} \tag{3.9}$$

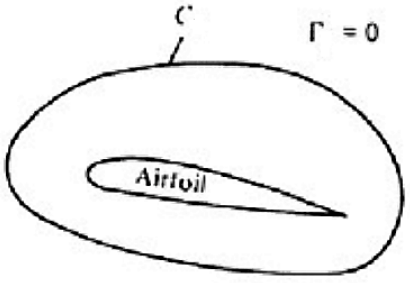


FIGURE 3.2: Fluid at rest relative to the airfoil

At some time t, this curves moves in space. If the flow is inviscid, since C is a fluid curve, from Euler equations is as follows:

$$\frac{d\Gamma}{dt} = 0 \tag{3.10}$$

Equation 3.11 is known as Kelvin's circulation theorem [48] and is a form of the angular momentum conservation principle. It states that the circulation is constant around any material curve in inviscid flow.

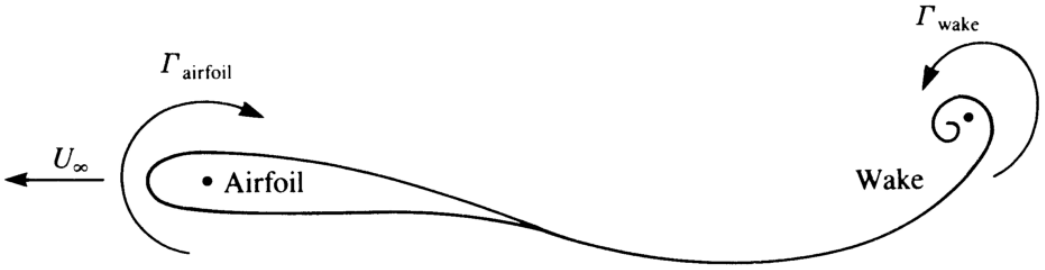


FIGURE 3.3: Circulation caused by an airfoil after it is suddenly set into motion [3]

In particular, for the airfoil in Figure 3.3 it states:

$$\frac{d\Gamma}{dt} = \frac{\Gamma_{airfoil} - \Gamma_{wake}}{\Delta t} = 0 \tag{3.11}$$

3.2.3 Helmholtz's theorems

Based on the results of the previous sections, von Helmholtz described the three-dimensional motion of fluid in the vicinity of vortex filaments. These three Helmholtz's theorems apply to inviscid flows and flows where the influence of viscous forces are small and can be ignored. They can be formulated as follows [49]:

- The strength of a vortex filament is constant along its length
- A vortex filament cannot start or end in a fluid, it must form a closed path or extend to infinity
- The fluid that forms a vortex tube continues to form a vortex tube and the strength of the vortex tube remains constant as the tube moves about (hence vortex elements, such as vortex lines, vortex tubes, vortex surfaces, etc., will remain vortex elements with time)

3.2.4 Kutta-Joukowski theorem

The Kutta–Joukowski theorem relates the lift generated by an airfoil to the speed of the airfoil through the fluid, the density of the fluid, and the circulation.

The theorem refers to two-dimensional flow around an airfoil and determines the lift generated by one unit of span. When the circulation Γ is known, the lift L per unit span of the airfoil can be calculated using the following equation:

$$L = \rho_\infty U_\infty \Gamma \tag{3.12}$$

where ρ_∞ and U_∞ are the fluid density and the fluid velocity far upstream of the airfoil which is now regarded fixed on a body-fixed frame, and Γ is the circulation defined in Equation 3.9.

Using vector notation, this can be expressed as

$$\vec{F} = \rho_\infty \vec{Q}_\infty \times \vec{\Gamma} \tag{3.13}$$

where $\vec{F}$ is the aerodynamic force per unit width and acts in the direction determined by the vector product and $\vec{Q}_\infty = [U_\infty, V_\infty, W_\infty]$. Note that positive $\vec{\Gamma}$ is defined according to the right-hand rule.

There are two main conclusions to highlight from Kutta-Joukowski theorem. The first is that, within the domain of the potential theory, the force over an airfoil is perpendicular

to the free stream velocity. The second is that the lift needs circulation in order to exist. For further information about the derivation of the equation, check Reference [50].

3.2.5 Boundary conditions

The potential flow theory states some boundary conditions that have to be fulfilled to solve Laplace's equation (Equation 3.8). The first requires zero normal velocity across the body's solid boundary surfaces:

$$(\vec{\nabla}\Phi + \vec{v}) \cdot \vec{n} = 0 \qquad (in\ X,\ Y,\ Z\ coordinates) \tag{3.14}$$

where, as already seen, $\vec{\nabla}\Phi$ is the variation of the velocity potential (or flow disturbance) in in the inertial frame, $-\vec{v}$ is the total velocity on the body surface and $\vec{n}$ is the normal vector to the surface. Note that $\vec{v}$ is defined with the minus sign so that the undisturbed flow velocity will be positive in the body's frame of reference.

The second boundary condition states that the flow disturbance produced by the motion of the lifting body in the fluid has to diminish far from the body. This is posed as:

$$\lim_{|\vec{R}-\vec{R}_0|\to\infty} \vec{\nabla}\Phi = 0 \tag{3.15}$$

where $\vec{R}$ is the position of the disturbed flow particle and $\vec{R}_0$ is the location of the body.

The Kutta-Joukowski condition (often shortened as the Kutta condition) is based on physical considerations. It states that the flow leaving the trailing edge of the airfoil or wing defines the shedding location for an infinitesimally thin wake in which all vorticity of the otherwise irrotational fluid is concentrated [5]. In inviscid, potential flow, it implies that there must be a stagnation point at the trailing edge (Figure 3.4). The implementation of the Kutta condition in a mathematical model is done in two main ways:

- The pressure jump over the trailing edge is prescribed as zero
- There is a streamline that leaves the airfoil trailing edge along the bisector to the trailing edge angle δ_{TE}

3.2.6 Biot-Savart law

A vortex line is a special kind of singularity solution of Equation 3.8. The infinite vortex line induces a flow field around a line with the induced flow perpendicular to the radius and the strength inversely proportional to the radius. The field is defined as:

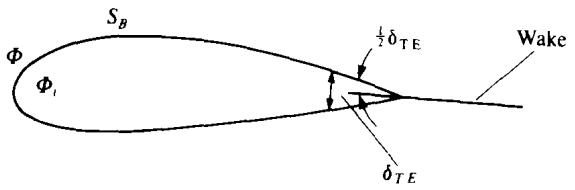


FIGURE 3.4: Kutta condition for the wake shape near the trailing edge

$$q = \frac{\Gamma}{2\pi r} e_\theta \tag{3.16}$$

where Γ is the field strength, r is the distance to the line and q is the induced velocity. When we consider a vortex segment, or a vortex line with finite length, this relation changes slightly. In this case the induced field is defined by the Biot-Savart Law:

$$d\vec{q} = \frac{\Gamma_n (d\vec{l} \times \vec{r})}{4\pi r^2} \tag{3.17}$$

Equation 3.17 can be integrated to give the induced velocity for a vortex segment of arbitrary length. This is done in [4] and the equation takes the following form:

$$\vec{q} = \frac{\Gamma_n}{4\pi} \frac{\vec{r_1} \times \vec{r_2}}{|\vec{r_1} \times \vec{r_2}|^2} \left[\vec{r_0} \cdot \left(\frac{\vec{r_1}}{r_1} - \frac{\vec{r_2}}{r_2} \right) \right] \tag{3.18}$$

The nomenclature is explained in Figure 3.5.

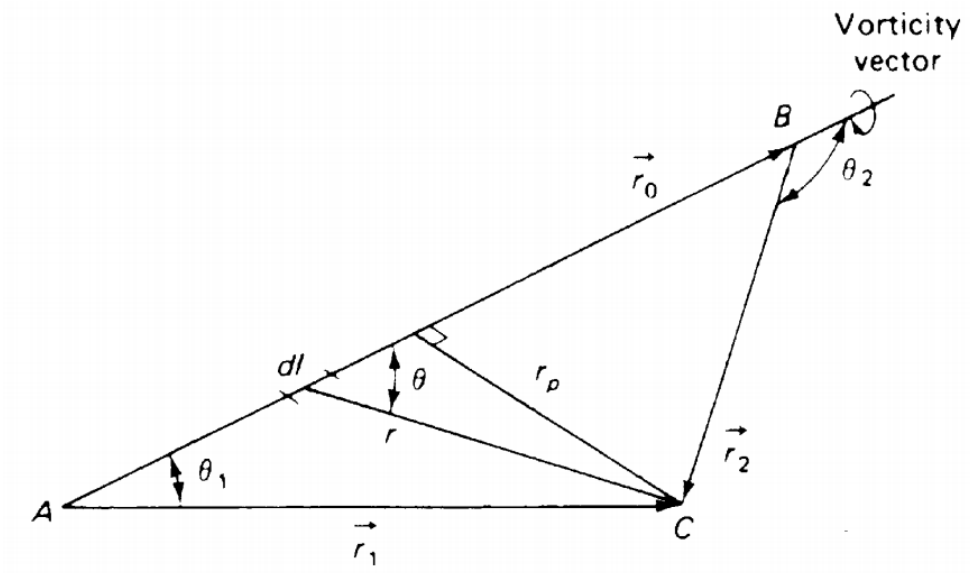


FIGURE 3.5: Nomenclature for calculating the downwash of a finite vortex segment [4]

These vortex segments can be utilized to build very intricate vortex systems, such as the mesh of vortex segments used in VLM. Traditional VLM uses three vortex segments to create a "horseshoe vortex" on every panel.

3.2.7 Horseshoe vortex

When creating an aerodynamic we must establish the properties that the model needs to fulfill. In the present case, these are:

- For a wing generating lift there must be a circulation around each curve encircling any airfoil section
- There is a trailing vortex downstream of each wingtip

The so called horseshoe vortex model, shown in Figure 3.6, satisfies these properties.

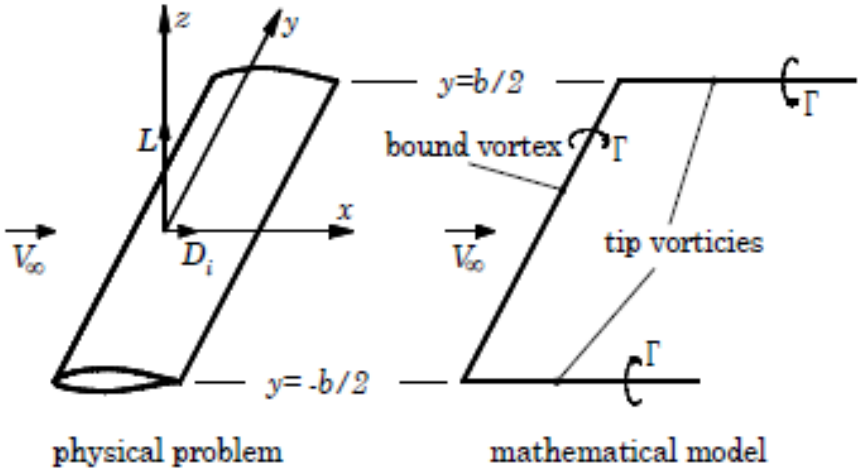
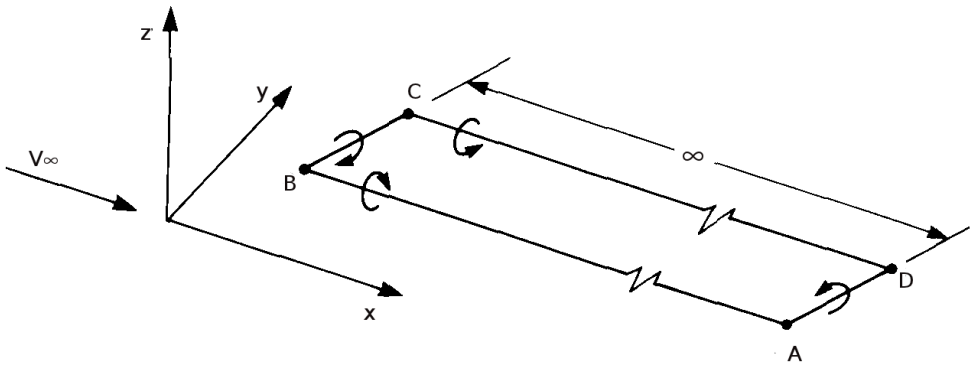


FIGURE 3.6: Physical problem and mathematical model for a horseshoe vortex [5]

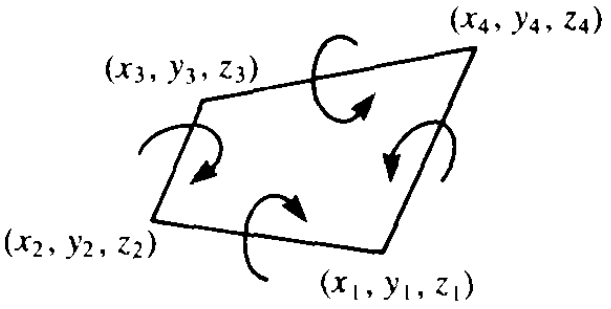
This horseshoe vortex consists of a bound vortex and two trailing vortices. There is also a starting vortex parallel to the bound vortex and located infinitely downstream of the wing. The bound vortex replaces the wing and becomes the source of lift through the Kutta-Joukowski equation (Equation 3.13). According to one of the Helmholtz vortex theorems, the bound vortex cannot end at the wing tips, so it continues downstream, creating the wake which is modeled by the trailing vortices. The two half-infinite vortices generate a downwash along the bound vortex, and this induced downwash is given by the Biot-Savart law.

3.2.8 Vortex rings

Vortex rings are a generalization of the horseshoe model. Theoretically, the horseshoe vortex is presented as a vortex ring with trailing segments of infinite length, as in Figure 3.7a. However, in practice, the horseshoe vortex is modeled as only three segments while the vortex ring is formed by four straight segments. The bound vortex is forward, the starting vortex is in the rear position and, linking both of them, there are two trailing vortices, as shown in Figure 3.7b.



(A) Horseshoe vortex



(B) Vortex ring

FIGURE 3.7: Vortex ring definition

One advantage of using vortex rings instead of horseshoe vortices is that the zero-normal-flow condition can be satisfied on the actual wing surface. The vortex rings formulation is used in this thesis. Another main reason to use these elements over horseshoe vortices is that vortex rings are less computationally expensive. Moreover, they can describe the wake rollup, as shown in the following sections.

3.3 Vortex ring formulation of the Vortex Lattice Method

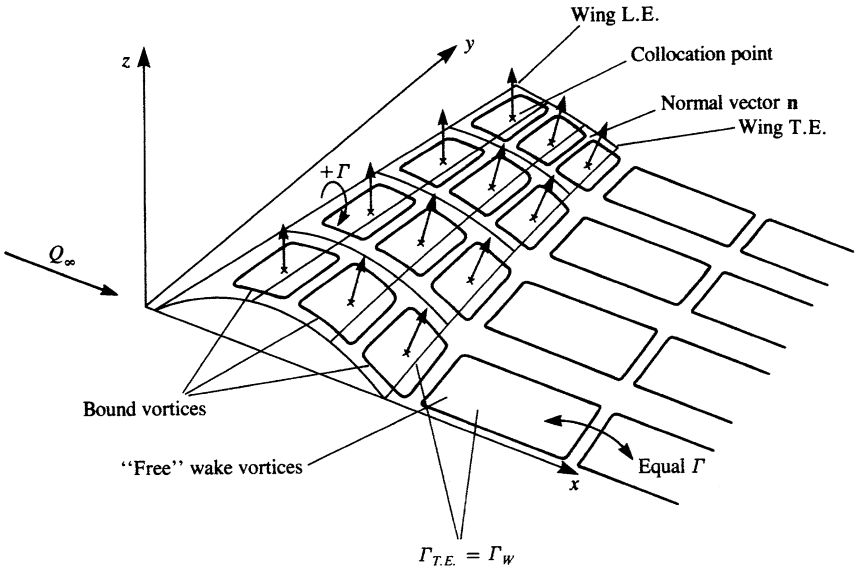


FIGURE 3.8: Vortex ring model for a thin lifting surface [3]

As introduced in the previous section, the vortex ring element could be defined by superimposing contributions from different straight vortex segments that form the vortex ring (Figure 3.7b). Doing this, it is possible to calculate the total induced velocity $\vec{q}$ at any arbitrary point a:

$$\vec{q}_a = [u_a, v_a, w_a] = [u_{1a}, v_{1a}, w_{1a}] + [u_{2a}, v_{2a}, w_{2a}] + [u_{3a}, v_{3a}, w_{3a}] + [u_{4a}, v_{4a}, w_{4a}] \quad (3.19)$$

The method by which the thin wing planform is divided into panels is shown in Figure 3.8. The lifting surface is divided into several spanwise and chordwise lattices and a vortex ring is placed in each panel, producing the vortex lattice. The leading segment of the vortex ring is placed on the panel's quarter chord line and the collocation point is at the center of the three-quarter chord line.

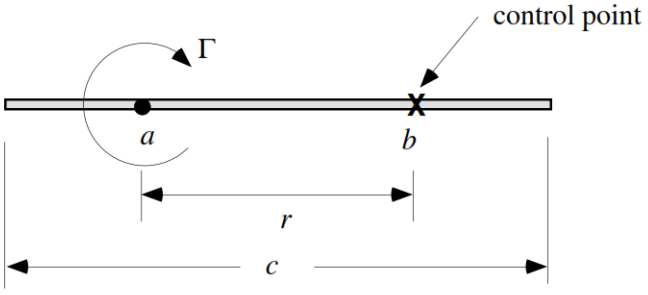


FIGURE 3.9: The notation for control point and vortex location analysis

This is shown in Figure 3.9, where

$$r = \frac{c}{2} \quad , \quad \frac{a}{c} = \frac{1}{4} \tag{3.20}$$

The $\frac{1}{4} - \frac{3}{4}$ rule is not a theoretical law, but simply a rule of thumb. It was discovered by Italian Pistolesi [51]. Mathematical derivations of more precise vortex/control point locations are available in [52], but the $\frac{1}{4} - \frac{3}{4}$ rule is widely used and has been proven to be sufficiently accurate in practice.

Because this is a steady implementation, the wake behind the wing is not discretized. In order to satisfy the Kutta condition, the wake at the trailing edge is replaced by a series of horseshoe vortices. The zero-normal-flow boundary condition is satisfied at the control point of each lattice.

The panel normal vector is also defined at the control points. The positive direction is defined according to the right-hand rule. One of the major differences between the VLM and panel methods is that in the former the singularities are placed on the mean surface of the wing, rather than on the actual surface. This distribution of vorticity seeks to emulate the changes in velocity induced on the flow as it traverses the upper and lower surfaces of a wing.

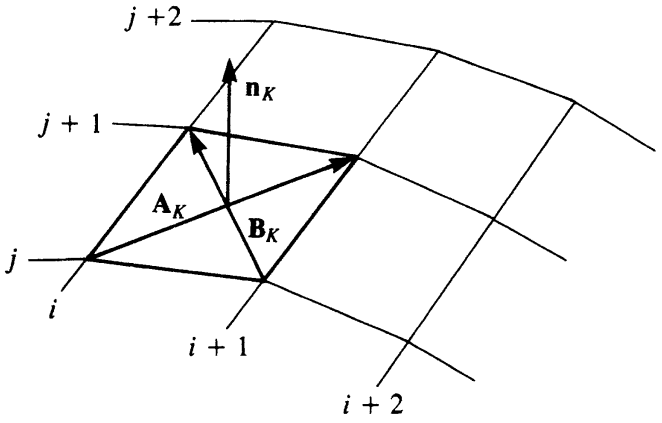


FIGURE 3.10: Definition of wing outward normal [3]

To compute the surface normal, we use a very simple and efficient method shown in Figure 3.10. The opposite corner points of the panel k define two vectors, $\vec{A_k}$ and $\vec{B_k}$, and their vector product is the surface normal. Hence:

$$\vec{n}_k = \frac{\vec{A_k} \times \vec{B_k}}{\left|\vec{A_k} \times \vec{B_k}\right|} \tag{3.21}$$

This method is precise when sufficient chordwise panels are used.

At this point, it is important to note that in situations when symmetry exists between the left and right halves of the body's surface, a rather simple method can be used to include these features in the numerical scheme. In terms of programming simplicity these modifications will affect only the influence coefficient calculation section of the code.

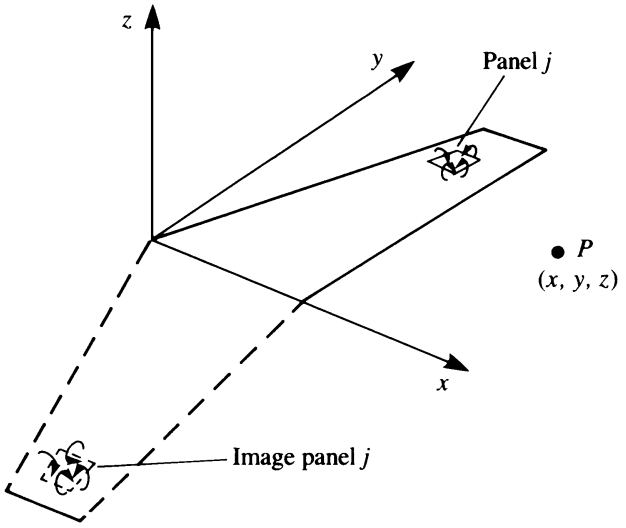


FIGURE 3.11: Image of the right-hand side of a symmetric wing model [3]

For example, consider the symmetric wing (left to right), shown in Figure 3.11, where only the right-hand half of the wing must be modeled. In order to evaluate the influence coefficients we first compute the dot product of the induced velocity of a particular vortex ring and the lattice normal vector. The induced velocity calculated is due to a unit strength vortex. Because symmetry is invoked (the distribution of circulation is symmetric), the system only has N equations rather than $2N$. Assuming that collocation point number 1 is being treated, the total velocity induced by the first vortex ring and its image is found to be:

$$[u, v, w]_{11} = [u_i, v_i, w_i]_{11} + [u_{ii}, v_{ii}, w_{ii}]_{11} \tag{3.22}$$

where i denotes the vortex ring and ii its image from the left semispan. In fact, in this case the image panel in the left half wing in Figure 3.11 will have the same strength of the correspondent one of the right half wing.

Then, the influence coefficient is:

$$a_{11} = [u, v, w]_{11} \cdot \vec{n}_1 \tag{3.23}$$

This process is repeated for each control point and each vortex ring until all the influence coefficients have been computed. In order to form the right-hand-side (RHS) of the

linear system of equations, the normal velocity components of the free-stream flow should be computed as follows:

$$RHS_i = -\vec{Q}_\infty \cdot \vec{n}_i = -\left[U_\infty, V_\infty, W_\infty\right] \cdot \vec{n}_i \tag{3.24}$$

Once the computation of the influence coefficients and right-hand-side vector is done, the zero-normal-flow boundary condition will result in the following system of linear equations:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{bmatrix} \begin{bmatrix} \Gamma_1 \\ \vdots \\ \Gamma_m \end{bmatrix} = \begin{bmatrix} RHS_1 \\ \vdots \\ RHS_m \end{bmatrix} \tag{3.25}$$

where m denotes the total number of vortex rings.

The solution of Equation 3.25 yields the value of the vortex strength for each panel, described by the vector $[\Gamma_1, \Gamma_2, ..., \Gamma_m]$. Thus, to compute the lift acting on each panel, the Kutta-Joukowski theorem is applied to each vortex-lattice panel:

$$\Delta L_j = \rho Q_\infty \Gamma_j \Delta y_j \tag{3.26}$$

where Δy_j is the panel bound vortex projection normal to the free stream.

Similarly, the induced drag calculation is:

$$\Delta D_j = -\rho w_{ind_j} \Gamma_j \Delta y_j \tag{3.27}$$

where the induced downwash w_{ind_j} at each collocation point j is computed by summing the velocity induced by all the trailing vortex segments.

The total lift and induced drag are then calculated by summing the individual panel contributions:

$$L = \sum_{j=1}^{N} \Delta L_j \tag{3.28a}$$

$$D_{ind} = \sum_{j=1}^{N} \Delta D_j \tag{3.28b}$$

Finally, the lift and induced drag coefficients could be computed as follows:

$$C_L = \frac{L}{\frac{1}{2}\rho S Q_\infty^2} \tag{3.29a}$$

$$C_{D_i} = \frac{D_{ind}}{\frac{1}{2}\rho S Q_\infty^2} \tag{3.29b}$$

3.4 Unsteady Vortex Lattice Method

Previously, we established that a velocity field can be obtained by solving the continuity equation, which does not include time-dependent terms. However, for unsteady conditions we introduce these terms through the boundary conditions [3]. This is the basis of unsteady potential flow theory. The same methods used to solve the steady case are applied to the unsteady case, though with slight modifications. These modification include a different treatment of the tangential flow boundary condition, incorporation of the unsteady Bernoulli equation, and a more complex wake than in the steady case.

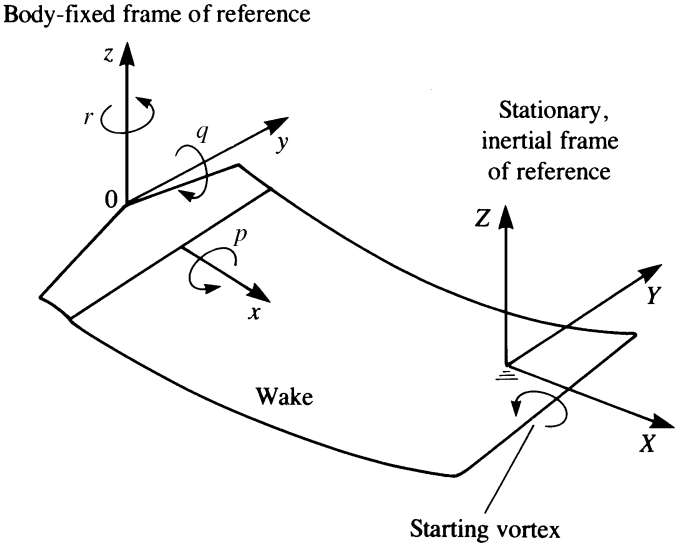


FIGURE 3.12: Inertial and body coordinates used to describe the motion of the body [3]

When treating time-dependent motions of bodies, the selection of the coordinate systems becomes very important. It is useful to describe the unsteady motion of the surface on which the zero-normal-flow boundary condition is applied in a body-fixed coordinate system (x, y, z); see for example the maneuvering wing depicted in Figure 3.12. The motion of the origin O of this coordinate system (x, y, z) is then prescribed in an inertial frame of reference (X, Y, Z) and is assumed to be known. For simplicity, assume that at $t = 0$ the inertial frame (X, Y, Z) coincides with the frame (x, y, z). Then, at $t > 0$, the relative motion of the origin of the body-fixed frame of reference is prescribed by its location $\vec{R}_0(t)$, and the instantaneous orientation $\vec{\Theta}(t)$, where (ϕ, θ, ψ) are the Euler rotation angles:

$$[X_0, Y_0, Z_0] = \vec{R}_0(t) \tag{3.30}$$

$$[\phi, \theta, \psi] = \vec{\Theta}(t) \tag{3.31}$$

For example, in the case of a constant-velocity flow of speed U_∞ in the positive x direction (in the wing's frame of reference), the function $\vec{R}_0(t)$ will be

$$[X_0, Y_0, Z_0] = [-U_\infty t, 0, 0] \tag{3.32}$$

which means that the wing is being translated in the negative X direction, so that the undisturbed flow velocity will be positive in the body's frame of reference.

In the inertial frame, the Laplace's equation (Equation 3.8) is always valid, like the two boundary conditions of Equations 3.14 and 3.15. Note that, since Equation 3.8 does not depend directly on time, the time dependency is introduced by Equation 3.14. In fact the location and orientation of $\vec{n}$ can vary with time.

For the unsteady flow Kelvin theorem supplies an additional equation that can be used to determine the stream-wise strength of the vorticity shed into the wake [3]. It states that in the potential flow region the angular momentum cannot change, so the circulation around the fluid curve enclosing the wing and the wake is conserved.

$$\frac{d\Gamma}{dt} = 0 \qquad (for\ any\ t) \tag{3.33}$$

The solution of this problem, which becomes time-dependent, is easier in the body-fixed coordinate system. Consequently, a transformation f between the two coordinate systems has to be established, based on the flight path information of Equations 3.30 and 3.31 as follows:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = f\left(X_0, Y_0, Z_0, \phi, \theta, \psi\right) \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.34}$$

Such a transformation should include the translation and the rotation of the (x, y, z) system [53] and has the following form:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi(t) & sin\phi(t) \\ 0 & -sin\phi(t) & cos\phi(t) \end{bmatrix} \begin{bmatrix} cos\theta(t) & 0 & -sin\theta(t) \\ 0 & 1 & 0 \\ sin\theta(t) & 0 & cos\theta(t) \end{bmatrix} \begin{bmatrix} cos\psi(t) & sin\psi(t) & 0 \\ -sin\psi(t) & cos\psi(t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X - X_0 \\ Y - Y_0 \\ Z - Z_0 \end{bmatrix} \tag{3.35}$$

Similarly, the kinematic velocity $\vec{V}$ of the lifting surface as viewed from the inertial frame of reference is given by:

$$\vec{V} = -\left(\vec{V}_0 + \vec{v}_{rel} + \vec{\Omega} \times \vec{r}\right) \tag{3.36}$$

where

- $\vec{V_0}$ is the velocity of the (x, y, z) system's origin, and must be resolved into the instantaneous (x, y, z) directions

$$\vec{V_0} = \left[\dot{X}_0, \dot{Y}_0, \dot{Z}_0\right] \tag{3.37}$$

- $\vec{r}$ is the position vector of the body's frame of reference

$$\vec{r} = [x, y, z] \tag{3.38}$$

- $\vec{\Omega}$ is the rate of rotation of the body's frame of reference

$$\vec{\Omega} = [p, q, r] \tag{3.39}$$

  where (p, q, r) are the angular velocity components, as shown in Figure 3.12

- $\vec{v}_{rel}$ is an eventual relative motion within the (x, y, z) system (e.g., small amplitude oscillation of the wing or its flap, in addition to the average motion of the body system)

$$\vec{v}_{rel} = [\dot{x}, \dot{y}, \dot{z}] \tag{3.40}$$

To an observer in the (x, y, z) frame, the velocity direction is opposite to the flight direction (as derived in the inertial frame) and therefore the minus sign appears in Equation 3.36. Now, if Equation 3.36 is substituted in the boundary condition of Equation 3.14, it follows:

$$\left(\vec{\nabla}\Phi - \vec{V_0} - \vec{v}_{rel} - \vec{\Omega} \times \vec{r}\right) \vec{n} = 0 \qquad (in\ x,\ y,\ z\ coordinates) \tag{3.41}$$

From this equation we notice that for incompressible flow, the instantaneous solution of Equation 3.8 is independent of time derivatives and, consequently, steady-state techniques can be used for the unsteady problem by substituting the boundary condition at every moment. However, the wake has to be treated differently. Its separation line is dictated by the Kutta condition and the velocity has to be finite along trailing edges of lifting surfaces.

The simplest way to obtain the wake strength is to apply the Kutta condition along the trailing edges of lifting wings:

$$\Gamma_{T.E.} = 0 \tag{3.42}$$

The validity of the assumption depends on the component of the kinematic velocity normal to the trailing edge. This must be much smaller than the characteristic velocity

for Laplace's equation to be valid [3]. Besides, the Kevin condition (Equation 3.33) can be used to calculate the change in the wake circulation.

The only equation that has to be defined yet is the unsteady Bernoulli equation. In the inertial frame of reference, from Equation 3.6, it becomes:

$$\frac{p_\infty - p}{\rho} = \frac{(\vec{\nabla}\Phi)^2}{2} + \frac{\partial \Phi}{\partial t} = \frac{1}{2}\left[\left(\frac{\partial \Phi}{\partial X}\right)^2 + \left(\frac{\partial \Phi}{\partial Y}\right)^2 + \left(\frac{\partial \Phi}{\partial Z}\right)^2\right] + \frac{\partial \Phi}{\partial t} \tag{3.43}$$

where

- p_∞ is the pressure in the free stream
- $\frac{(\vec{\nabla}\Phi)^2}{2}$ is the gradient of the velocity potential
- $\frac{\partial \Phi}{\partial t}$ is the velocity potential time derivative

The last term can be obtained in terms of the circulation, establishing the following relations [3]:

$$\Phi^{\pm} = \pm \int \frac{\Gamma}{2} dl \qquad (integrating\ from\ the\ leading\ edge) \tag{3.44}$$

Since for the vortex model that is used here $\Delta\Phi = \Gamma$, then:

$$\pm \frac{\partial \Phi_{ij}}{\partial t} = \pm \frac{1}{2} \frac{\partial \Gamma_{ij}}{\partial t} \tag{3.45}$$

Chapter 4

Numerical implementation

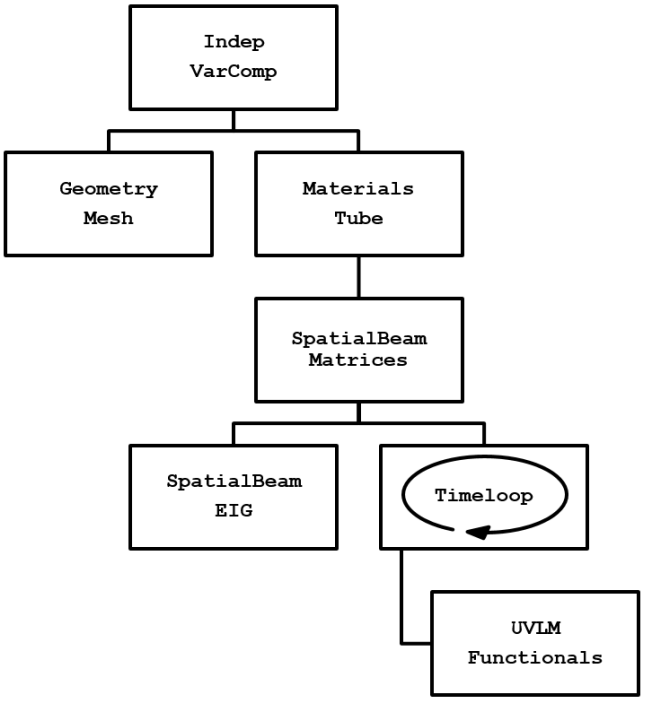
This chapter provides a brief overview of the core concepts and constructs involved in the code written for this project. As introduced in Section 1.4, the code is written in Python and is built in the framework OpenMDAO. This means that the code includes components and groups, organized by topics. A component is a computational class of OpenMDAO, where you build models and wrap external analysis codes. A group represents collections of components and other groups with data passing and a specified execution sequence. A detailed explanation of the OpenMDAO framework could be found in [54].

4.1 Structure of the code

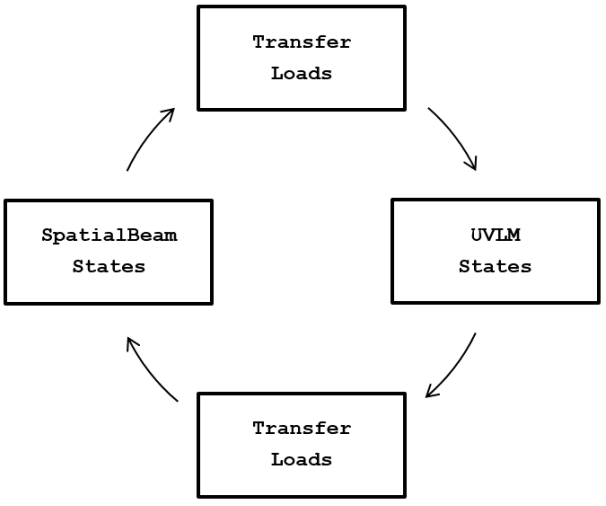
The aim of the present work is to improve the aerodynamic model of *OpenAeroStruct* and perform a dynamic aerostructural analysis. For this reason, the introduction of a time loop is needed. Moreover, in order to find the critical flutter velocity of the system, a parametric loop is performed. In this way, it is possible to run the analysis and achieve dynamic results for different flow velocities. The same loop could be used for the convergence studies if the time step duration and wake element length are provided.

The code is composed of four main parts:

- Main program
- Structural model
- Aerodynamic model
- Transfer functions



(A) Main program



(B) Time loop

FIGURE 4.1: *OpenAeroStruct* structure

The center of the program is the main: it includes definition of flow and beam properties, definition of the independent variables with the OpenMDAO component `IndepVarComp`, calls of components or groups, and the time loop, as shown in Fig.4.1. The listing of the main program is given in Appendix A and will be analyzed in detail in the next section.

In the structural part, the spatial beam introduced in Section 2.2.4 is modeled. Before the time loop, the material properties, as cross sections and moments of inertia, are evaluated in a component called `MaterialsTube`. Then, mass, damping and stiffness matrices are built in `SpatialBeamMatrices` and the modal analysis is performed with the component `SpatialBeamEIG`. During the loop, the structural model receives the vector of forces from the transfer function `TransferLoads` and the Newmark-Beta algorithm can solve the dynamic system. Finally, the resultant displacement vector is passed to the transfer function `TransferDisplacements`.

The aerodynamic part consists of the unsteady Vortex Lattice Method discussed in Section 3.4. Before the time loop, the aero-mesh for the first time step has to be created in the component `GeometryMesh`. For the rest of the loop, the aero-mesh is defined in `TransferDisplacements`.

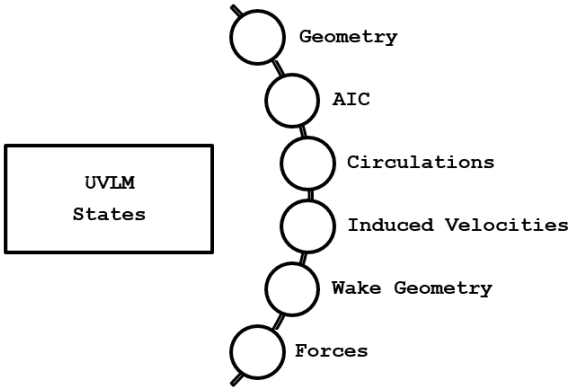


FIGURE 4.2: UVLM group with its components

All the aerodynamic components called in the time loop form the group `UVLMStates` (Fig.4.2). The output of this group is a matrix of aerodynamic loads that is passed to `TransferLoads`. After the loop, the components of the group `UVLMFunctionals` define the C_L and the C_D of the structure by using the aerodynamic forces of the last step of the iteration. Listings of aerodynamic components and groups are given in Appendix A.

4.2 Main program

4.2.1 Loop for parametric analysis

In order to evaluate the flutter velocity of the coupled system, the analysis has to be performed for a range of velocities. For this reason all the main program is closed in a `for` loop, and just before it, we have to declare the desired range of velocities in `velocities_vect` and the number of intervals in which this range has to be divided (`num_of_points`). At this point, in the loop the flow velocity `v` is defined.

```
18 num_of_points = 4
19 velocities_vect = numpy.linspace(50.0, 200.0, num=num_of_points)
20 #num_dt_vect = numpy.linspace(100, 400, num=num_of_points)
21
22 for i in xrange(num_of_points):
23
24     v = velocities_vect[i]
25     #v = 50.                          # flow speed [m/s]
26     v = float(v)
```

Listing 4.1: Main: loop for parametric analysis

As already mentioned, this loop could be used for others parameters. For example, in line 20 of Listing 4.1, the number of time steps can be parametrized by the vector `num_dt_vect`. It is useful for the time step convergence study. In this case, lines 19 and 24 should be commented and, conversely, the line 25 has to be uncommented, in order to fix the same flow velocity for each run.

4.2.2 Parameters for simulation in time

The unsteady VLM needs a time loop to get the effects of wake rollup in the energy exchange between structure and flow. Here we define duration of the simulation by fixing `final_t`, and the number of time steps `num_dt` in which we want to divide this interval. Moreover we define the wake influence on the profile by choosing the number of wake elements `num_w` that have to be considered. If `num_dt` is bigger than `num_w`, only the nearest wake elements have influence on the wing.

```
31     #num_dt = int(num_dt_vect[i])
32     num_dt = 100                     # number of time steps
33     final_t = 5.                     # time-simulation duration [s]
34     num_w = 50                       # number of (timewise) deforming wake elements
```

Listing 4.2: Main: parameters for simulation in time

4.2.3 Aerodynamic parameters

In this part of the code, we simply define the wing angle of attack `alpha` and the air density `rho`. It is also possible to fix the values of the zero lift coefficient C_{L_0} and the zero lift drag coefficient C_{D_0} in lines 41-42 of Listing 4.3.

```
39    rho = 1.225                    # air density [kg/m^3]
40    alpha = 5.                     # angle of attack [deg.]
41    CL0 = 0.
42    CD0 = 0.
```

LISTING 4.3: Main: aerodynamic parameters

4.2.4 Wing geometry and aerodynamic mesh

The 2D aerodynamic mesh is created following the shape of the selected wing. It is possible to choose the CRM wing, introduced in Section 1.3, or a simple rectangular wing. The choice is done by setting the boolean variable `CRM` in line 47 of Listing 4.4: we define it as `True` for the CRM wing, or `False` for the rectangular one.

```
47    CRM = True
48
49    if CRM:                        # Use the CRM wing model
50        wing = 'CRM'
51        npi = 4                    # number of points inboard
52        npo = 6                    # number of points outboard
53        full_wing_mesh = gen_crm_mesh(npi, npo, num_x=5)
54        num_x, num_y = full_wing_mesh.shape[:2]
55        num_y_sym = numpy.int((num_y + 1) / 2)
56        span = 58.7630524 # [m]
57        num_twist = 5
58
59    else:                          # Use the rectangular wing model
60        wing = 'RECT'
61        num_x = 4                  # number of spanwise nodes
62        num_y = 17                 # number of chordwise nodes
63        num_y_sym = numpy.int((num_y + 1) / 2)
64        span = 20. # [m]
65        chord = 1. # [m]
66        cosine_spacing = 0.
67        full_wing_mesh = gen_mesh(num_x, num_y, span, chord, cosine_spacing)
68        num_twist = numpy.max([int((num_y - 1) / 5), 5])
69
70    half_wing_mesh = full_wing_mesh[:, (num_y_sym-1):, :]
```

LISTING 4.4: Main: wing geometry and aerodynamic mesh

In the case of the CRM wing, it is necessary to decide the number of points inboard (between the root and the section with the Yehudi break shown in Figure 1.3) and the number of points outboard (in the rest of the wing until the tip). Note that the point of the central section between these two parts is in common. Then, for the case shown in Listing 4.4, along the semi-wing there are a total of 9 nodes, i.e. 8 finite elements. The aero-mesh is created by the function `gen_crm_mesh`, and at line 54 the number of chord-wise and span-wise nodes are defined, respectively `num_x` and `num_y`. Then, `num_y_sym` indicates the number of nodes along the semi-span. Finally, the span and the number of different twist angles along the wing are defined in lines 56-57.

For the rectangular wing, the module is similar, but here we have to define the chord length too. The function `gen_mesh` generates the rectangular aero-mesh in line 67.

In the end, in order to reduced the computational cost of the program, only the right half wing is considered for our analysis, and it is defined in line 70 in both cases.

4.2.5 Beam properties

In this section, the properties of the structural model are defined. The wing is modeled as a tube beam with a radius `r` and a thickness `thick` defined in lines 75-76 of Listing 4.5. It is possible to define the chordwise position of the elastic axe `fem_origin`, and the percentual damping coefficient `zeta` used in Equation 2.88. The following material properties can be defined immediately below: the Young's modulus `E`, the Poisson's coefficient `poisson`, the shear modulus `G` and the mass density `mrho`. In this particular example, a beam of Aluminium has been considered.

```
75    r = radii(half_wing_mesh)/5      # beam radius
76    thick = r / 5                    # beam thickness
77    fem_origin = 0.5                 # elastic axis position along the chord
78    zeta = 0.8                       # damping percentual coeff.
79
80    E = 70.e9 # [Pa]
81    poisson = 0.3
82    G = E / (2 * (1 + poisson))
83    mrho = 2800. # [kg/m^3]
```

Listing 4.5: Main: beam properties

4.2.6 Independent variables

In OpenMDAO, there are two ways to use a defined variable in a specified component. The first way is passing the variable as a direct input in the call of the component, as in other widely used numerical programming languages. The second and most useful way

is to define the variable as a design variable and call it in the component as a `param`. It also allows to use our variable to minimize the output when doing optimization. Note that the design variable has to be initialized.

```
88      indep_vars = [
89          ('span', span),
90          ('twist', numpy.zeros(num_twist)),
91          ('dihedral', 0.),
92          ('sweep', 0.),
93          ('taper', 1.0),
94          ('v', v),
95          ('alpha', alpha),
96          ('rho', rho),
97          ('r', r),
98          ('thick', thick)]
```

LISTING 4.6: Main: independent variables

The list of variables defining `indep_vars` in Listing 4.6 will be declared as design variables with the OpenMDAO component `IndepVarComp` The list comprises the span and the twist angle vector already defined in Listing 4.4, the geometrical angles that can be defined for the rectangular wing (`dihedral`, `sweep` and `taper`), the aerodynamic parameters (`v`, `alpha` and `rho`) and the structural ones (`r` and `thick`).

4.2.7 Calls of components

Once all the variables are defined, components and groups of the code are called in this part of the main program and regrouped in `root`. We could divide them in these groups: components before the time loop, those involved in the loop, and those that have to be run only after the loop.

The first call is for the OpenMDAO component `IndepVarComp` that, as said in the previous section, declares the variables of the vector `indep_vars` as design variables. After that, the structural variables relatives to the tube section are defined in `MaterialsTube`. Then, the first aero-mesh defined in Listing 4.4 is twisted in `GeometryMesh`. The output of this component is the design variable `mesh` which represents the aerodynamic mesh that starts the loop. For the structural part, `SpatialBeamMatrices` and `SpatialBeamEIG` has to be called before the loop, because the matrices of the dynamic problem and the modal analysis don't change during the aeroelastic analysis. Note that the call `SBEIG` allows to uses an instance of the component `SpatialBeamEIG` during the loop. This artifice is necessary for the mechanism of the Newmark-Beta solver. In fact its internal counter has to be updated externally, without re-call the original component that contains the first call of the algorithm. For sure, this mechanism could undergo to future improvements.

The time loop consists of a `for` loop that run until the actual number `n` of the time step reaches the prefixed total number of steps `num_dt`. For each step, a new group `SingleStep` is created, which includes the corresponding time step number appended to its name. This is clear in line 126 of Listing 4.7, where the string `name_step` represents the system name of the actual `SingleStep` group. Note that the first time step is indicated with the number 0. Moreover, all the groups `SingleStep` are regrouped in `coupled`, which in turn is added to `root` in lines 130-132. After the time loop, `UVLMFunctionals` is the last group added to `root`.

```
103     root = Group()
104
105     # Components before the time loop
106     root.add('indep_vars',
107              IndepVarComp(indep_vars),
108              promotes=['*'])
109     root.add('tube',
110              MaterialsTube(num_y_sym),
111              promotes=['*'])
112     root.add('mesh',
113              GeometryMesh(full_wing_mesh, num_twist),
114              promotes=['*'])
115     root.add('matrices',
116              SpatialBeamMatrices(num_x, num_y_sym, E, G, mrho),
117              promotes=['*'])
118     SBEIG = SpatialBeamEIG(num_y_sym, num_dt, final_t)
119     root.add('eig',
120              SBEIG,
121              promotes=['*'])
122
123     # Time loop
124     coupled = Group()
125     for t in xrange(num_dt):
126         name_step = 'step_%d'%t
127         coupled.add(name_step,
128                     SingleStep(num_x, num_y_sym, num_w, E, G, mrho, fem_origin, SBEIG, t),
129                     promotes=['*'])
130     root.add('coupled',
131              coupled,
132              promotes=['*'])
133
134     # Components after the time loop
135     root.add('vlm_funcs',
136              UVLMFunctionals(num_x, num_y_sym, CL0, CD0, num_dt),
137              promotes=['*'])
```

Listing 4.7: Main: calls of components

4.2.8 Run the program

The last part of the main program is the creation of `Problem` in line 143 of Listing 4.8, and the subjection of `root` to it. In lines 148-149 it is possible to record and save in a file database all the values of the variables in the analysis. This allows us to manipulate data without re-running long analysis. Finally, the problem is solved at the line 153.

```
143     prob = Problem()
144     prob.root = root
145     prob.print_all_convergence()
146
147     # Setup data recording
148     db_name = 'results/db/%s_%d_%d.db'%(wing, num_dt)
149     prob.driver.add_recorder(SqliteRecorder(db_name))
150
151     prob.setup()
152     view_tree(prob, outfile="aerostruct.html", show_browser=False)
153     prob.run_once()
```

Listing 4.8: Main: run the program

4.2.9 Graph partition tree

An useful tool for the conceptual chomprension of the code structure could be the partition tree created at line 152 of Listing 4.8. In fact, it schematizes the structure of the code and the mutual interaction between system variables, as shown in Figure 4.3.

In the left part of this conceptual tree, the design variables are indicated in grey, while components or groups are in blue. All of them are indicated with their system names which may not correspond with the general component/group name. In fact, all the classes and the variables in `coupled` contain the number of the relative time step. For simplicity, in the example of Figure 4.3 a single time step simulation has been run. Note that the tree can be collapsed with a simple click, in order to zoom the scheme relative to a specific component/group, as shown in Figure 4.4 for `step_0`.

In the right part of the tree are indicated all the relationships between system variables. Circles and squares indicates scalar and vectorial variables, while black and yellow colors refer respectively to explicit or implicit derivation of them. For example, in Figure 4.4, we conclude that the state `circulations_0` is an implicit vector (yellow square) and is directly related with `c_pts_0`, `normals_0`, `v_local_0` and `AIC_mtx_0` (red links), while it directly influences the system variable `v_ind_wing_0` and the entire component `forces_0` (green links).

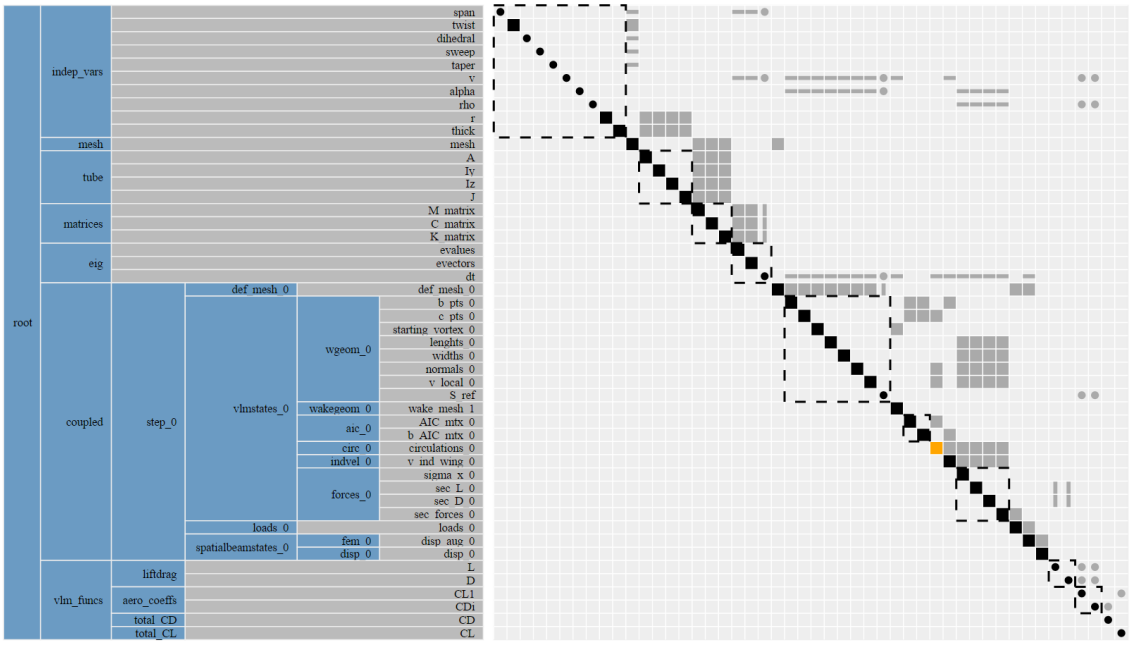


FIGURE 4.3: Partition tree for the aeroelastic problem with only one time step

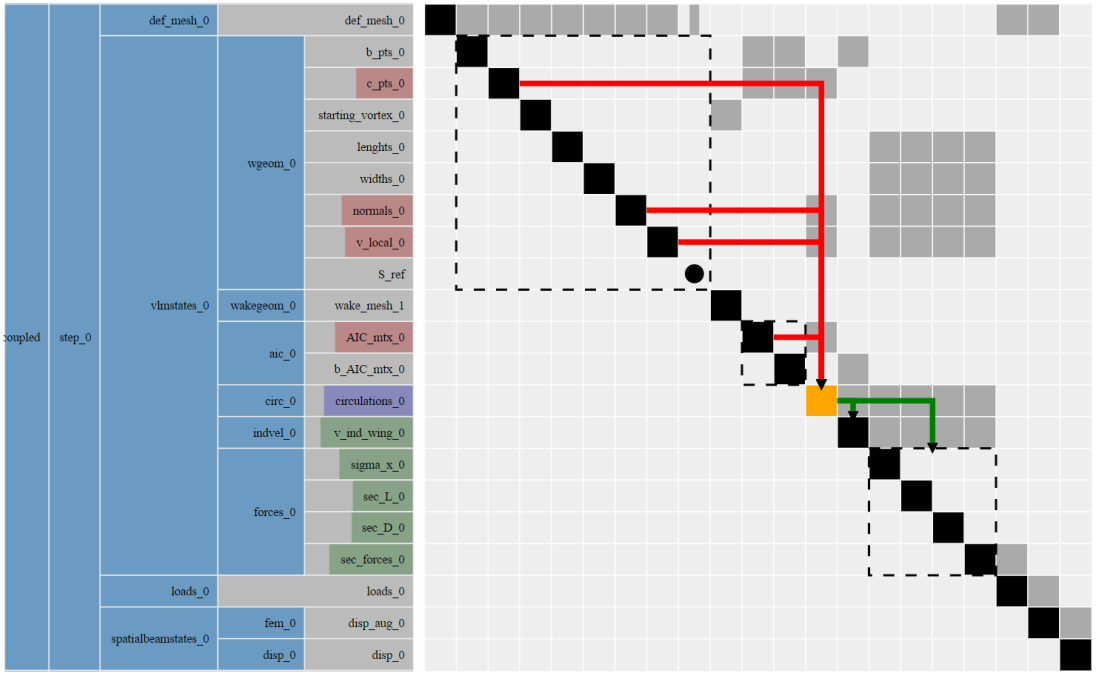


FIGURE 4.4: Collapsed partition tree for a single time step, with focus on circulation state variable relationships

Chapter 5

Results

In this chapter we present the results of this project whose main goal is the implementation of a tool for dynamic aeroelastic analysis in OpenMDAO. First, individual portions of the code have to be verified by comparing results with published articles. Then the coupled system has to be tested and verified with some well-known examples. After that, a brief but important note concerning system convergence is examined. Then, after having introduced the visualization tool, we present a general example of flutter analysis using *OpenAeroStruct*. Finally, the introduction of a Tuned Mass Damper is proposed in order to increase the flutter velocity of the system.

5.1 Code verification

In this section we want to verify the code by comparing results of structural, aerodynamic, and coupled models with some of those found in the literature.

5.1.1 Modal analysis verification

In order to verify the eigenvalues analysis performed in the structural part of the code, we model the example proposed in Reference [55]. Table 5.1 gives the structural and planform data for the wing model under investigation.

Using these data in the system of Equation 2.67, we obtain the eigenvalues listed in the second column of Table 5.4. The comparison between results obtained with *OpenAeroStruct* and those from [55] assures us that the global structural matrices are assembled in the correct way.

TABLE 5.1: Wing model data for modal analysis verification [55]

Half span	16 m
Chord	1 m
Mass per unit length	0.75 kg/m
Mom.Inertia (50% chord)	0.1 $kg\ m$
Spanwise elastic axis	50% chord
Center of gravity	50% chord
Bending rigidity	2 x 10^4 $N\ m^2$
Torsional rigidity	1 x 10^4 $N\ m^2$
Bending rigidity (chordwise)	4 x 10^4 $N\ m^2$

TABLE 5.2: Comparison of frequency results

Ref.[55] $[rad/s]$	OpenAeroStruct $[rad/s]$	Error %
2.243	2.243	0
14.057	14.056	+0.007
31.096	31.046	+0.161
31.718	31.718	0
39.380	39.356	+0.061

5.1.2 Unsteady Vortex Lattice Method verification

The UVLM is tested with the example proposed by Katz and Plotkin [3] using an uncambered rectangular wing in constant-speed forward flight. In this case the coordinate system is selected such that the x coordinate is parallel to the motion and the kinematic velocity components of Equation 3.35 become $[U(t), 0, 0]$. The angle of attack effect is taken care of by pitching the wing in the body frame of reference. For the planar wing all the normal vectors will be

$$[n] = [sin\alpha, 0, cos\alpha] \tag{5.1}$$

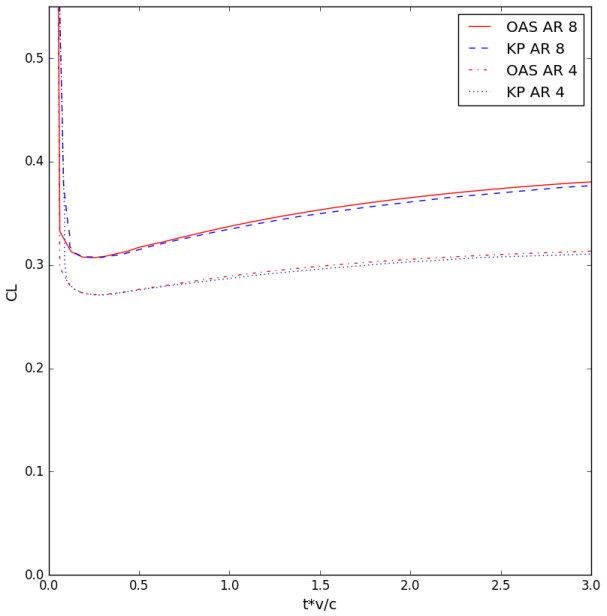
Consequently, the RHS vector of Equation 3.24 becomes

$$RHS_i = -\left[\left(U\left(t\right) + u_w\left(t\right)\right) sin\alpha + w_w cos\alpha\right]_i \tag{5.2}$$

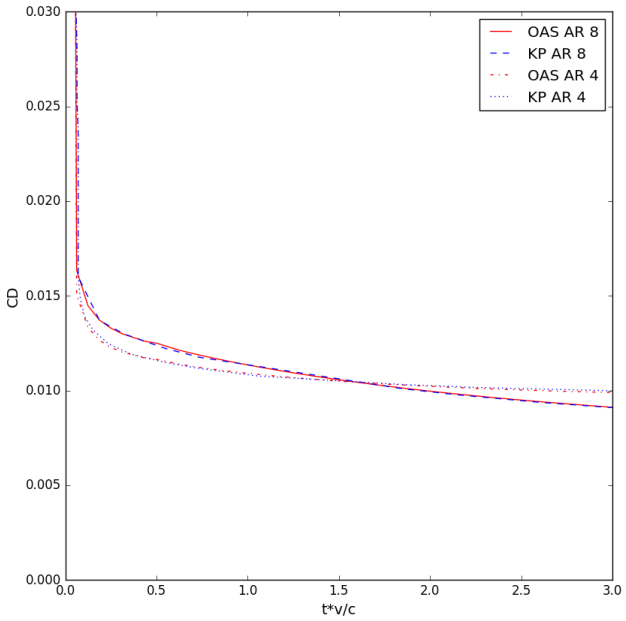
and here the wake influence will change with time.

For the numerical investigation, the wing is divided into 4 chordwise and 13 spanwise equally spaced panels. The time step is defined by the relation:

$$\frac{U_\infty \Delta t}{c} = \frac{1}{16} \tag{5.3}$$



(A) Lift coefficient



(B) Drag coefficient

FIGURE 5.1: Comparison between UVLM results of *OpenAeroStruct* (OAS) and Katz and Plotwin [3] (KP) for a rectangular wing with two different aspect ratio (AR)

Proceeding as explained in Section 3.3, the transient lift and drag coefficients variations with time are plotted in Figure 5.1. The results are in good agreement with those of Katz and Plotkin [3].

5.1.3 Flutter analysis verification

The flutter analysis of *OpenAeroStruct* has been verified by modeling the Goland wing [56] and comparing results with [57]. The wing data are described below:

TABLE 5.3: Goland wing data [56]

Half span	20 ft
Chord	6 ft
Mass per unit length	0.746 $slugs/ft$
Spanwise elastic axis	33% chord
Center of gravity	43% chord
Bending rigidity	23.65 x 10^6 $lb\ ft^2$
Torsional rigidity	2.39 x 10^6 $lb\ ft^2$

Note that all the data given in Table 5.3 are in the British Imperial system of units [58]. Running the analysis, we obtain results that are very close to those achieved in [57], by using the finite-state aerodynamic theory of Peters and Covorkers [59]. Moreover, we can compare them with the "exact" linear flutter speed of the cantilevered wing published in [56].

TABLE 5.4: Comparison of frequency results

Ref.[57] $[ft/s]$	Ref.[56] $[ft/s]$	OpenAeroStruct $[ft/s]$	Error with [56] %
445	450	437	3.33

The resultant flutter velocity of *OpenAeroStruct* is slightly less than the two references. This variation could be due to the different aerodynamic model.

Another comparison has been done with [6] where the ONERA aerodynamic stall model is used. The wing data are the same of Table 5.1 and the resultant flutter velocity is again lower than that from the reference. This is shown in Figure 5.2, where the flutter velocity has been achieved for different angles of attack.

Again, the different aerodynamic model might account for the small difference. Another possible motivation could be the big time step that has been used for this analysis. In fact, in order to visualize a convergence or a divergence of the signal amplitude, we need to simulate the interaction fluid-structure for a very large amount of time steps. It has been found a compromise between the simulation time and the precision of results.

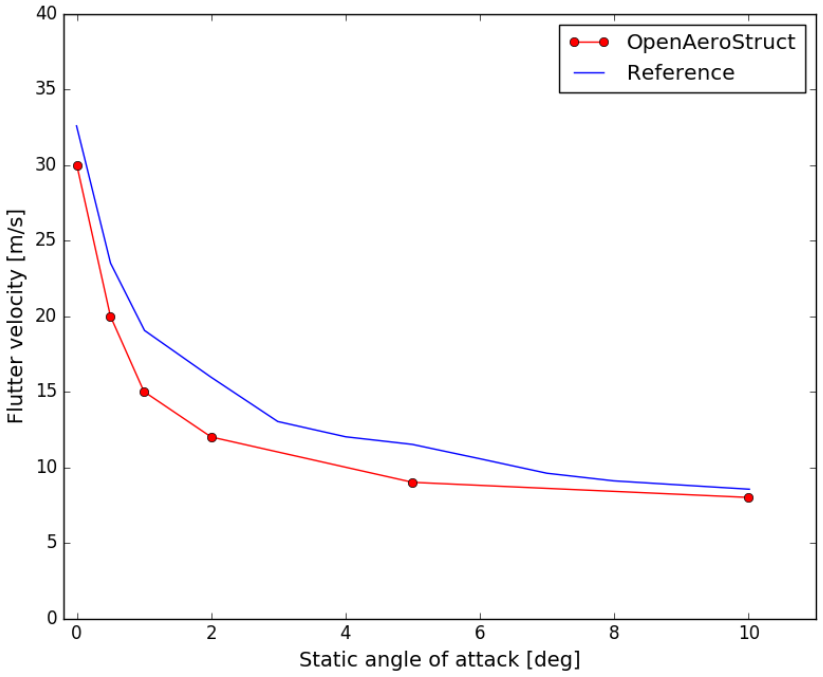
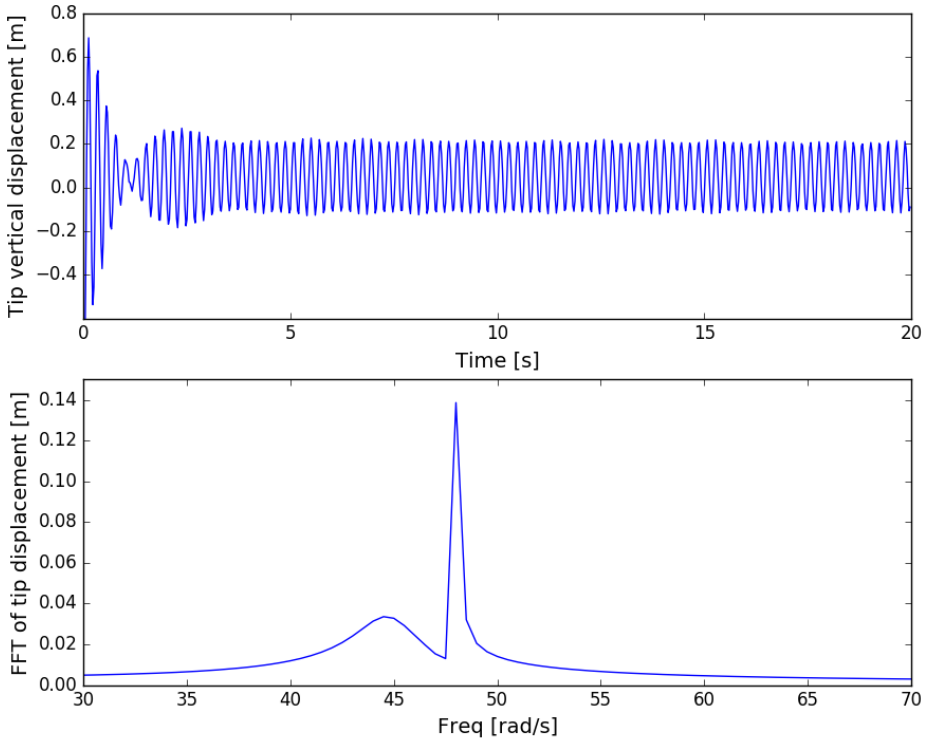


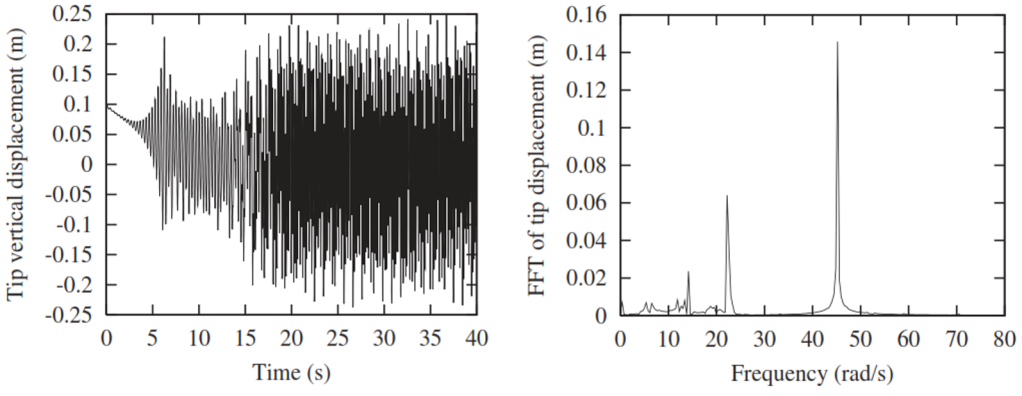
FIGURE 5.2: Flutter velocity at different angles of attack, compared with [6]

For the same example, we can plot the time history of the wing tip displacement, and the corresponding Fast Fourier Transform (Figure 5.3).

Comparing *OpenAeroStruct* results in Figure 5.3a with those of Tang and Dowell [6] of Figure 5.3b, we observe that in the time history the initial condition for the two simulations is different, but nevertheless the amplitudes of the wing tip vertical displacement are comparable, as well as the peak frequency in the FFT. The little differences could be due to numerical imprecision, as the use of not optimal parameters (num_dt, num_w, number of chordwise and spanwise elements).



(A) *OpenAeroStruct* results



(B) Tang and Dowell results [6]

FIGURE 5.3: Time history (above) and corresponding FFT analysis of tip vertical response for $U_\infty = 33\ m/s$, compared with [6]

5.2 Convergence studies

In this section we discuss the importance that a conscientious choice of analysis parameters could have in the results. The most important parameter is the time step of the simulation, which has to be adequately small in order to avoid numerical issues. In particular, the UVLM and the Newmark-Beta algorithm suffer majorly when using a large dt. Next, we show how to fix the number of wake elements num_w that influence the structure.

5.2.1 Time step sizing

Considering a general analysis performed with a fluid speed v on a wing with a chord c, we can plot the time history of the wing tip displacement. The dimensionless time is found by multiplying the time by v and dividing by c. Because this quantity is dimensionless, we do not need to specify the problem data, and this convergence analysis will be general. Note that also the displacement of the wing tip is divided by c.

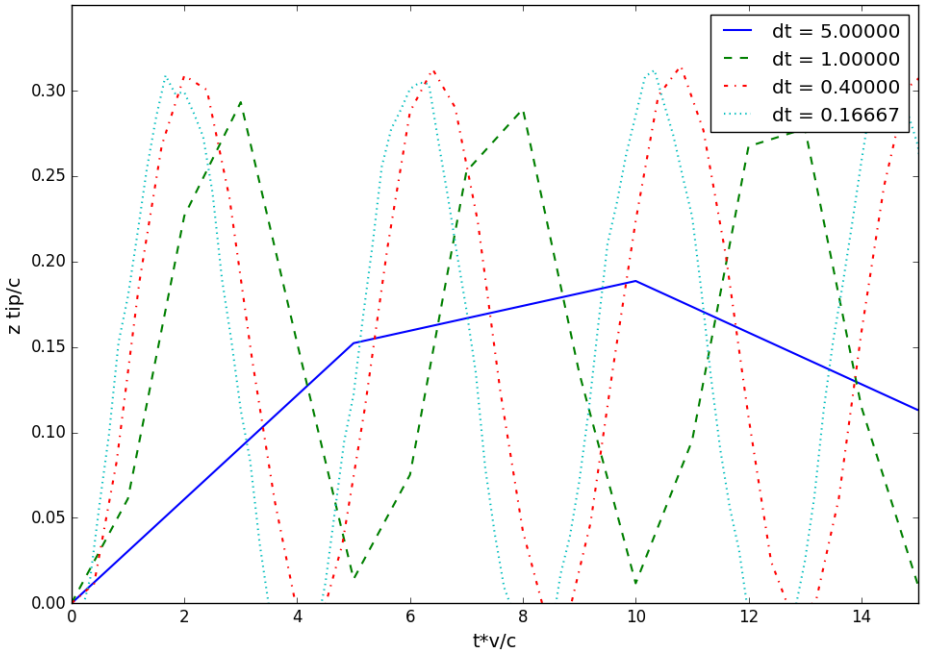


FIGURE 5.4: Example of inaccurate time step

In Figure 5.4 the importance of an adequate time step is evident. The time step influences both amplitude and frequency of the signal, which in this case is the wing tip displacement, and this is not acceptable. We need to find a maximum value of the dimensionless time step in order to assure good accordance with the real signal. Of course,

every problem is different and a compromise between accuracy and calculated velocity must be found.

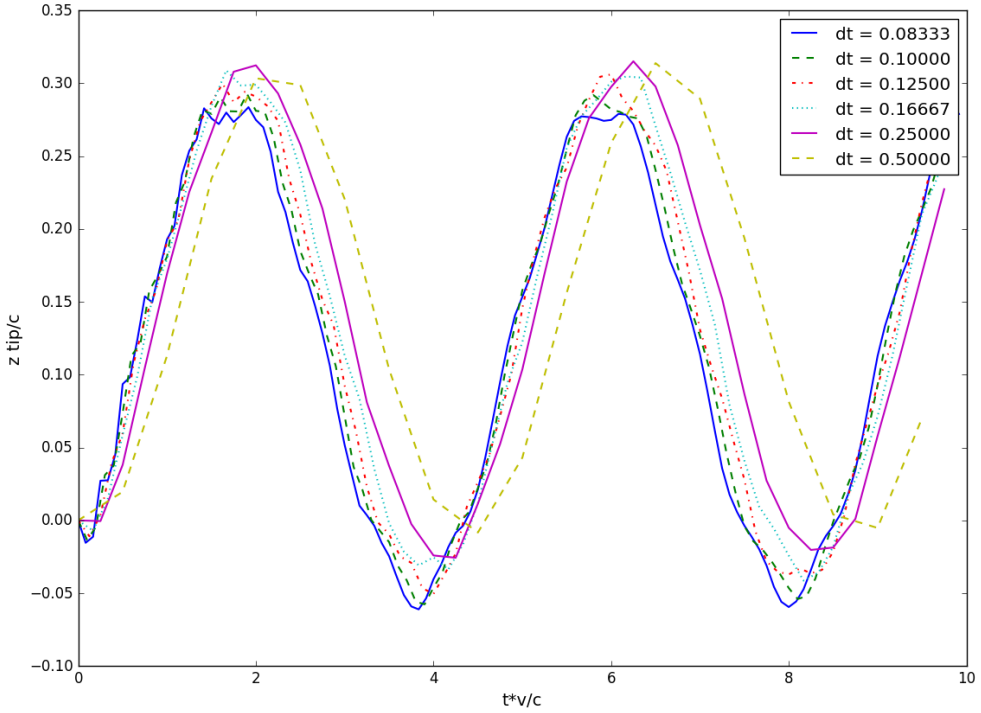


FIGURE 5.5: Time step convergence

In Figure 5.5 we have increased the number of simulation points by decreasing dt. We find that we have convergence and conclude that a dimensionless time step lower than 0.125 should be used.

5.2.2 Wake influence

Now we want to analyse the effect of the parameter num_w on our simulation. This parameter fixes the minimum number of wake panel rows that can influence the wing. After the time step $dt = num\_w$ the program starts to store only the position of the last num_w rows for each iteration.

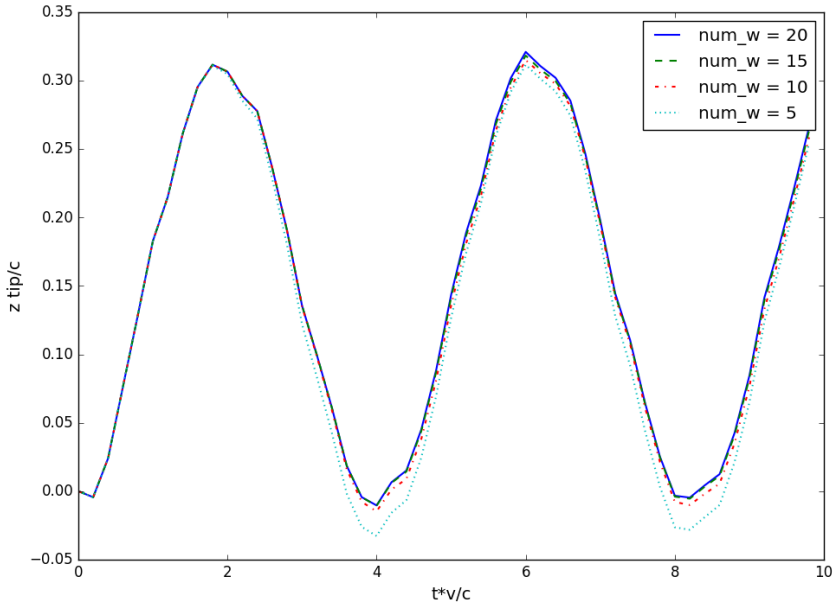


FIGURE 5.6: Convergence of the wake rows number

Figure 5.6 shows that at least 10 wake rows are required in order to reduce the resulting error of this approximation. Note that for an optimal visualization of the wake rollup, an higher num_w has to be taken into account. In fact this parameter has also another meaning: this is the number of wake rows that change their positions during the simulation. Then, with a little num_w there is no possibility to model the wake rollup.

5.3 Graphical representation

The code *OpenAeroStruct* includes a useful graphic tool that provides a visualization of the aero-mesh and the spatial beam, as shown in Figure 5.7. With this tool it is possible to change the orientation of the model by rotating it in a 3D view.

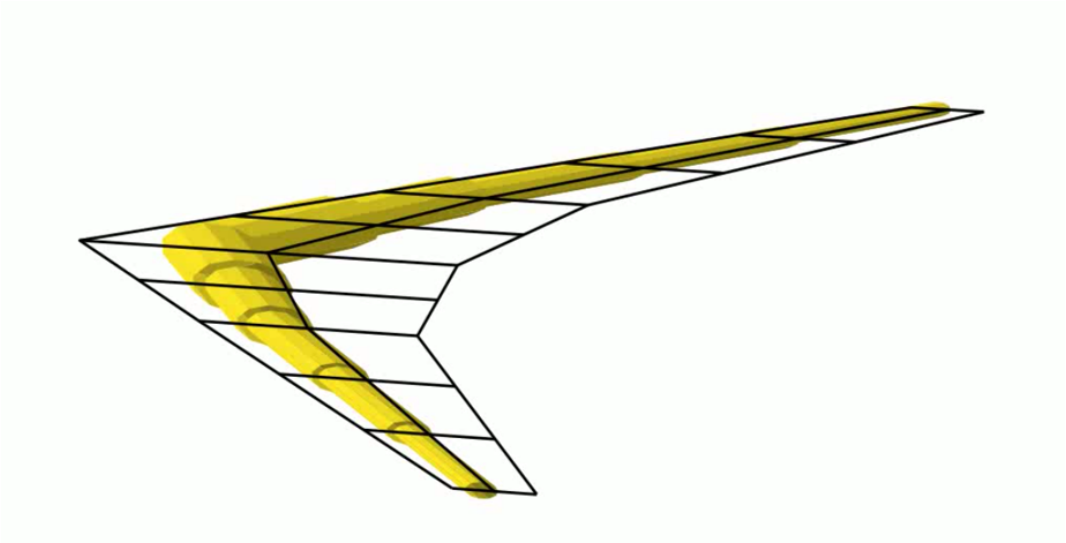


FIGURE 5.7: 3D visualization of the CRM wing aero-mesh with a tapered spatial beam

An important improvement to the code that has come from this project is the time dependence, and it has been included in the visualization tool too. It is possible to visualize the system configuration at each time step. It is also possible to show these different configurations in sequence by creating a movie (.mp4 format). Adding the wake mesh in the tool, it has been possible to visualize the wake rollup, as shown in Figure 5.8. In this case, the rectangular wing is moving to the left, leaving a wake behind in its previous positions. Fixing num_w to be a high number, wake nodes are influenced by the lifting surface panels for many time steps and this allows the wake rollup to build. The starting wake filament, parallel to the wing trailing edge, tends to curl onto itself, forming a visible vortex.

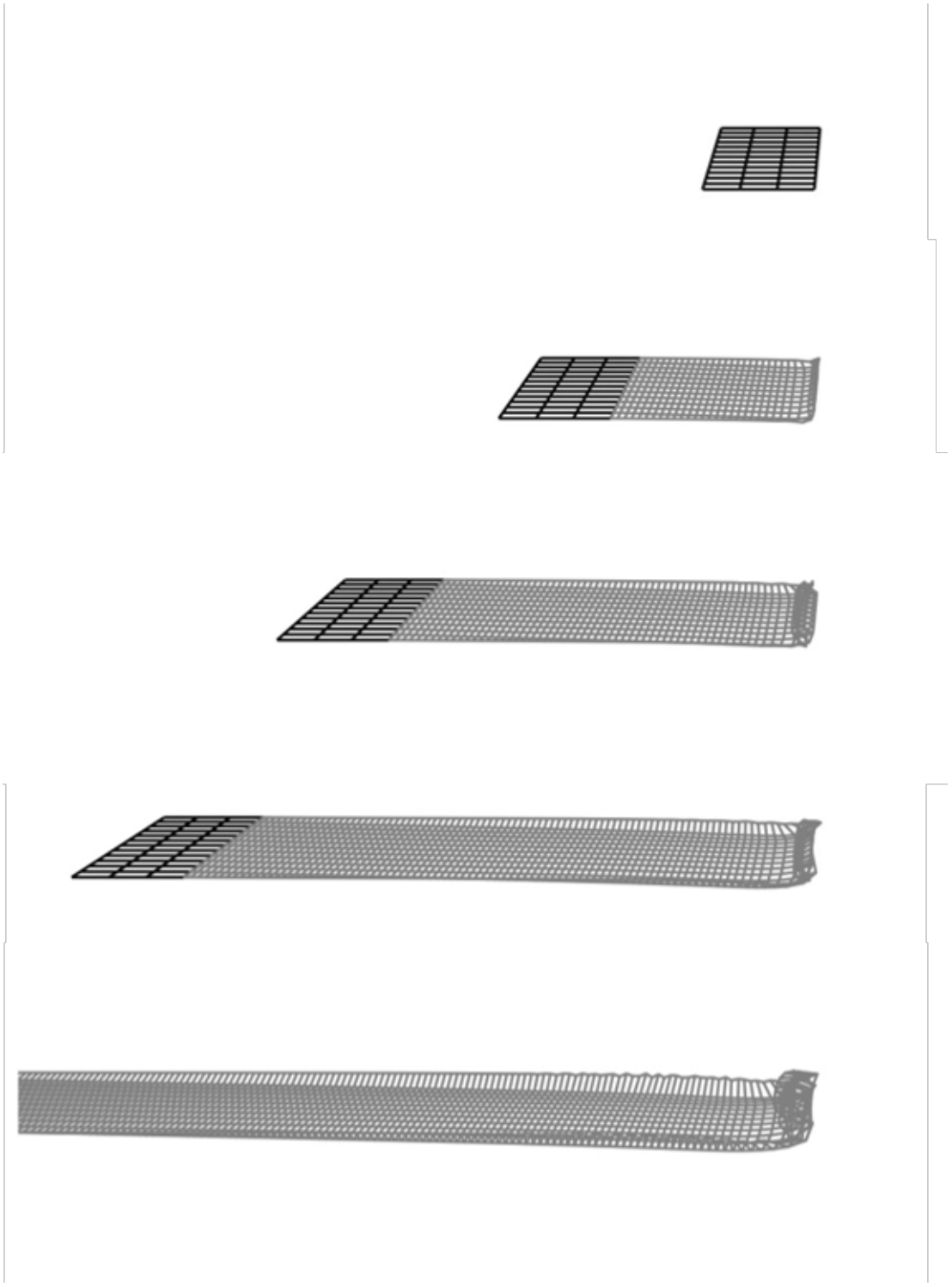


FIGURE 5.8: Wake rollup formation during the time simulation

5.4 Example of analysis

In this section we want to illustrate an example of application in order to explain how to determine the flutter velocity of the system. Let's consider a rectangular wing with values listed in Table 5.5.

TABLE 5.5: Example of *OpenAeroStruct* application

Half span	10 m
Chord	4 m
Spanwise elastic axis	50% chord
Center of gravity	50% chord
Tube radius	0.12 m
Tube thickness	0.024 m
Tube density	2800 $kg\ m^3$
Coeff. Poisson	0.3
Young module	70 10^9 Pa
Spanwise elements	10
Chordwise elements	4
Flow density	1.225 $kg\ m^3$

The flutter analysis is performed by varying the flow speed and analysing the time history of the system variables. If a convergence appear, this means that the system is stable in correspondence with that v. The first velocity at which we start to observe a constant or divergent behaviour, like augmentation of the signal amplitude, that is the sought flutter velocity.

In Figure 5.9 is shown the resultant time histories of the tip displacement for four different flow speed. It is clear that the aeroelastic instability occurs between 50 m/s and 70 m/s. Then, we can intensify the analysis in this range of velocities, like in Figure 5.10. Now the decision is a bit difficult, because it could be influenced by different parameters. It seems that the flutter velocity for this example is $v_f = 64\ m/s$. For velocity higher than v_f we observe a divergence in the harmonic response.

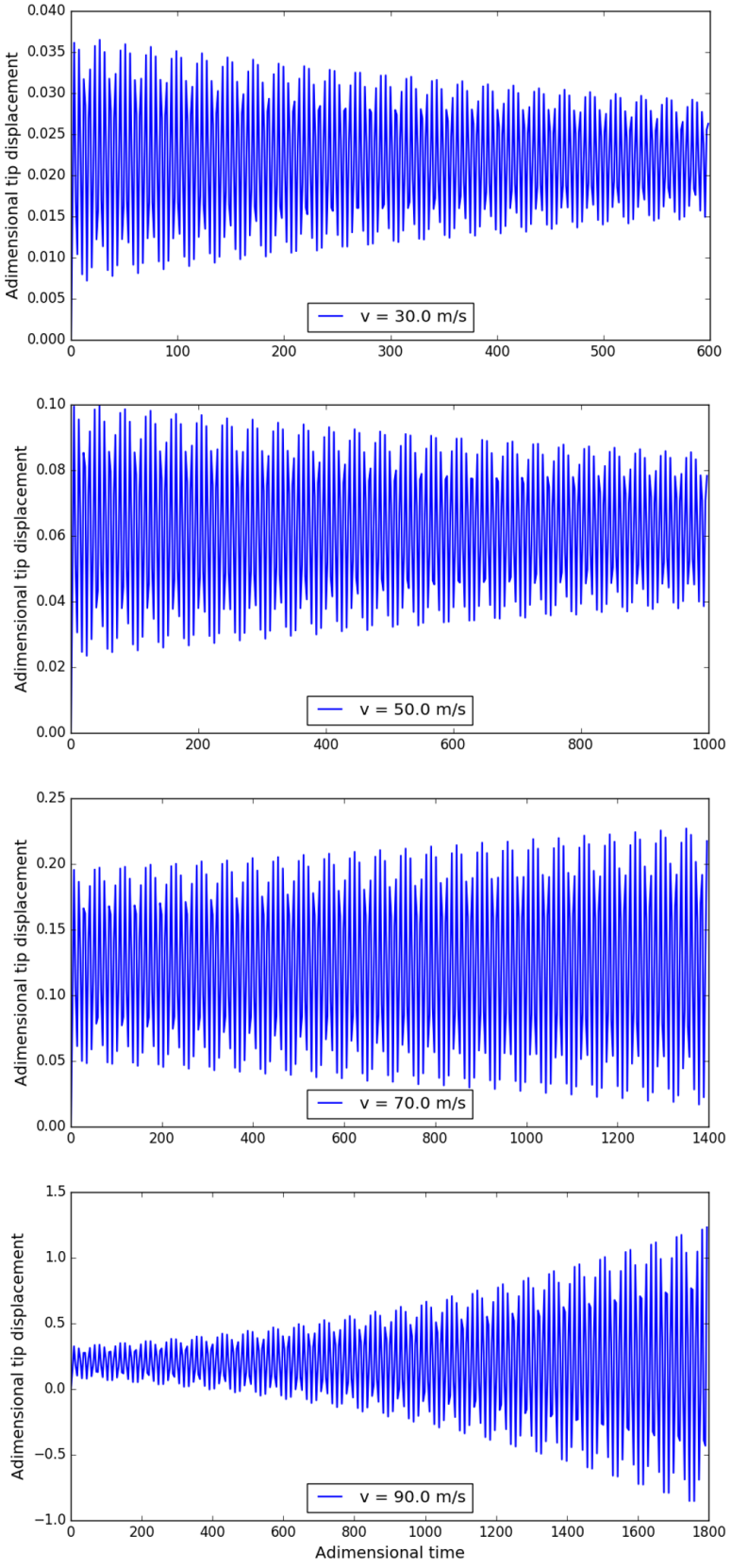


FIGURE 5.9: Aeroelastic analysis for different flow speeds

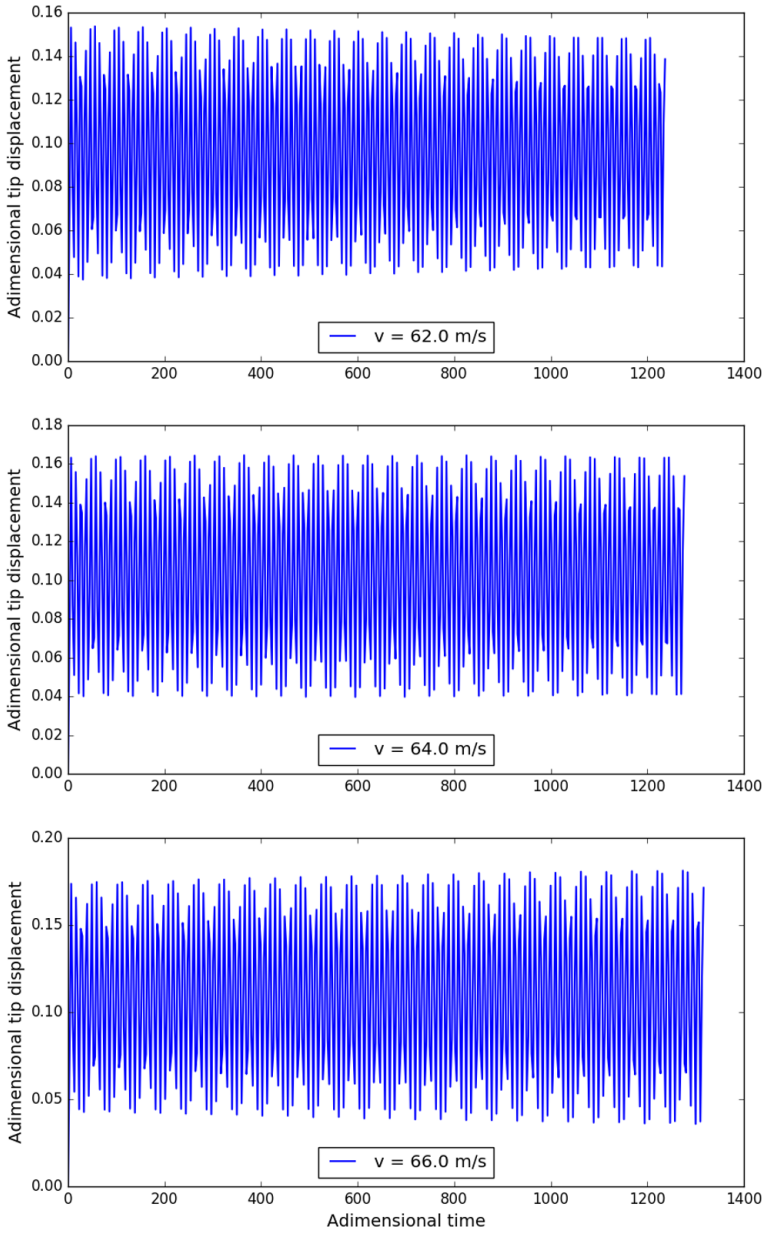


FIGURE 5.10: Research of the flutter velocity

5.5 Addition of a TMD

In this last section, we propose the addiction af a linear tuned mass damper (TMD) in order to increase the flutter velocity of the system. Further information about TMDs are given in Appendix B.

Considering the example proposed in [6] and already performed to verify the code in Section 5.1.3, we add a TMD in a specific position and it acts only on the spatial beam node of the correspondent section. We choose to fix its distance from the root equal to a sixth of the span, and in the chordwise direction we put it in the middle between the elastic axis and the trailing edge. Following [60], we impose that the factor between the half wing mass and the absorber mass has to be 100. Moreover, for the damping coefficient of the absorber we fix a value of 0.02.

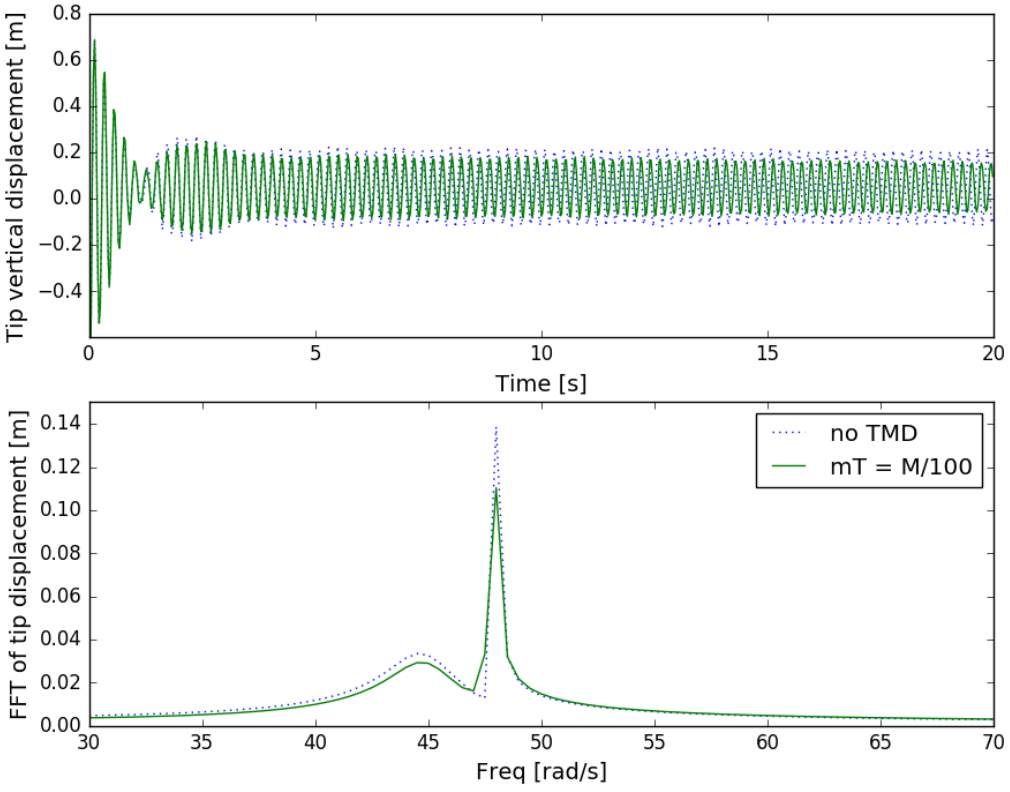


FIGURE 5.11: The effect of the TMD on flutter velocity and frequency

In Figure 5.11 it is shown the effect of the TMD addiction on the system at its flutter velocity (33 m/s). The amplitude of tip oscillations decreases, as it is possible to verify in the FFT. This convergence indicates that the flutter velocity of the system with the addiction of the TMD is higher.

Chapter 6

Conclusions and future work

The aim of this work was to improve the pre-existing open-source *OpenAeroStruct* code in order to perform dynamic aeroelastic analysis using the OpenMDAO framework. The previous version of this code could only perform steady linear aeroelastic analysis and optimization by modeling the wing as a spatial beam with a tube section and by obtaining aerodynamic forces with the classical Vortex Lattice Method (VLM).

Both structural and aerodynamic models have been largely modified. New contributions in the structural part include the introduction of mass and damping matrices, modal analysis, and the Newmark-Beta solver for the resolution of the equilibrium system. The Vortex Lattice Method has been updated in order to take in count unsteady effects due to wake effects (UVLM). The most important change in the code structure has been the inclusion of time dependency that has led to an increase in operational complexity and capabilities. Moreover, the main program and the visualization tool have been updated in order to take in count the time dependency. Finally, each of the code listings has been ordered, cleared of unnecessary lines, and fully commented.

After these modifications, with *OpenAeroStruct* it is possible to evaluate and visualize the time history of system variables, like displacements and forces, or the Fast Fourier Transform (FFT) of these signals. That allows the user to graphically estimate the system flutter velocity and gives important indications about amplitude and frequency of signal oscillations. The verification studies conducted in this thesis assure the validity of this code and the reliability of the generated results.

The last part of this project has demonstrated a possible application of this analysis tool. A linear Tuned Mass Damper (TMD) has been introduced in order to ameliorate the dynamic aeroelastic response of the system. The positive effect of this simple addiction has been shown and the optimization of TMD properties could be a fruitful avenue for future research.

The main improvement is the inclusion of these modeled dynamic effects in an optimization loop. In fact, what really makes *OpenMDAO* special is its automatic analytic multidisciplinary derivatives, which can be used to compute system-level gradients for Newton solvers and/or gradient-based optimizers. It would be possible, for example, to optimize the wing mass by constraining the flutter velocity or the tip displacement amplitude. Another important improvement would be the introduction of nonlinearities in the structural model that could lead to more precise and realistic results, especially when the small deformation assumption would not be satisfied.

In conclusion, the original objective of this project has been achieved: the implementation of a efficient and tested tool for dynamic aeroelastic analysis in OperMDAO with the ability to model unsteady wake effects and perform time-dependent simulations.

Appendix A

Python listings

Listing A.1: Main program

```
""" Code to run aerostructural analysis and evaluate flutter velocity."""

from __future__ import division
import numpy
from time import time

from openmdao.api import IndepVarComp, Problem, Group, SqliteRecorder
from geometry import GeometryMesh, gen_crm_mesh, gen_mesh
from materials import MaterialsTube
from spatialbeam import SpatialBeamMatrices, SpatialBeamEIG, radii
from timeloop import SingleStep
from uvlm import UVLMFunctionals, Plots
from openmdao.devtools.partition_tree_n2 import view_tree

#####################################
# Define loop for parametric analysis #
#####################################
num_of_points = 4
velocities_vect = numpy.linspace(50.0, 200.0, num=num_of_points)
#num_dt_vect = numpy.linspace(100, 400, num=num_of_points)

for i in xrange(num_of_points):

    v = velocities_vect[i]
    #v = 50.                        # flow speed [m/s]
    v = float(v)

    ###########################################
    # Define parameters for simulation in time #
    ###########################################
    #num_dt = int(num_dt_vect[i])
    num_dt = 100                    # number of time steps
    final_t = 5.                    # time-simulation duration [s]
    num_w = 50                      # number of (timewise) deforming wake elements

    ####################################
    # Define the aerodynamic parameters #
    ####################################
    rho = 1.225                     # air density [kg/m^3]
    alpha = 5.                      # angle of attack [deg.]
    CL0 = 0.
    CD0 = 0.

    ###########################################
    # Define wing geometry and aerodynamic mesh #
    ###########################################
    CRM = True

    if CRM:                         # Use the CRM wing model
        wing = 'CRM'
        npi = 4                     # number of points inboard
```

```
        npo = 6                     # number of points outboard
        full_wing_mesh = gen_crm_mesh(npi, npo, num_x=5)
        num_x, num_y = full_wing_mesh.shape[:2]
        num_y_sym = numpy.int((num_y + 1) / 2)
        span = 58.7630524 # [m]
        num_twist = 5

    else:                           # Use the rectangular wing model
        wing = 'RECT'
        num_x = 4                   # number of spanwise nodes
        num_y = 17                  # number of chordwise nodes
        num_y_sym = numpy.int((num_y + 1) / 2)
        span = 20. # [m]
        chord = 1. # [m]
        cosine_spacing = 0.
        full_wing_mesh = gen_mesh(num_x, num_y, span, chord, cosine_spacing)
        num_twist = numpy.max([int((num_y - 1) / 5), 5])

    half_wing_mesh = full_wing_mesh[:, (num_y_sym-1):, :]

    ##########################
    # Define beam properties #
    ##########################
    r = radii(half_wing_mesh)/5     # beam radius
    thick = r / 5                   # beam thickness
    fem_origin = 0.5                # elastic axis position along the chord
    zeta = 0.8                      # damping percentual coeff.

    E = 70.e9 # [Pa]
    poisson = 0.3
    G = E / (2 * (1 + poisson))
    mrho = 2800. # [kg/m^3]

    ###################################
    # Define the independent variables #
    ###################################
    indep_vars = [
        ('span', span),
        ('twist', numpy.zeros(num_twist)),
        ('dihedral', 0.),
        ('sweep', 0.),
        ('taper', 1.0),
        ('v', v),
        ('alpha', alpha),
        ('rho', rho),
        ('r', r),
        ('thick', thick)]

    ######################
    # Calls of components #
    ######################
    root = Group()

    # Components before the time loop
    root.add('indep_vars',
             IndepVarComp(indep_vars),
             promotes=['*'])
    root.add('tube',
             MaterialsTube(num_y_sym),
             promotes=['*'])
    root.add('mesh',
             GeometryMesh(full_wing_mesh, num_twist),
             promotes=['*'])
    root.add('matrices',
             SpatialBeamMatrices(num_x, num_y_sym, E, G, mrho),
             promotes=['*'])
    SBEIG = SpatialBeamEIG(num_y_sym, num_dt, final_t)
    root.add('eig',
             SBEIG,
             promotes=['*'])

    # Time loop
    coupled = Group()
    for t in xrange(num_dt):
        name_step = 'step_%d'%t
        coupled.add(name_step,
                    SingleStep(num_x, num_y_sym, num_w, E, G, mrho, fem_origin, SBEIG, t),
                    promotes=['*'])
    root.add('coupled',
             coupled,
             promotes=['*'])

```

```
    # Components after the time loop
    root.add('vlm_funcs',
             UVLMFunctionals(num_x, num_y_sym, CL0, CD0, num_dt),
             promotes=['*'])

    ###################
    # Run the program #
    ###################

    prob = Problem()
    prob.root = root
    prob.print_all_convergence()

    # Setup data recording
    db_name = 'results/db/%s_%d_%d.db'%(wing, num_dt)
    prob.driver.add_recorder(SqliteRecorder(db_name))

    prob.setup()
    view_tree(prob, outfile="aerostruct.html", show_browser=False)
    prob.run_once()
```

Listing A.2: Unsteady Vortex Lattice Method

```
""" Defines the aerodynamic analysis component using Unsteady Vortex Lattice Method (UVLM) """

from __future__ import division
import matplotlib.pyplot as plt
import numpy

from openmdao.api import Component, Group
from scipy.linalg import lu_factor, lu_solve

fortran_flag = False
if 1:
    fortran_flag = True
if fortran_flag:
    import lib


def norm(vec):
    """ Define the norm of the vector """
    return numpy.sqrt(numpy.sum(vec**2))

def calc_vorticity(A, B, P):
    """ Define the vorticity induced by the segment AB to P """
    rAP = P - A
    rBP = P - B
    rAP_len = norm(rAP)
    rBP_len = norm(rBP)
    cross = numpy.cross(rAP, rBP)
    may = numpy.sum(cross**2)

    r_cut = 1e-10
    cond = any([rAP_len < r_cut, rBP_len < r_cut, may < r_cut])
    if cond:
        return numpy.array([0., 0., 0.])

    return (rAP_len + rBP_len) * cross / \
           (rAP_len * rBP_len * (rAP_len * rBP_len + rAP.dot(rBP)))

def assemble_AIC_matrices(rings_mesh, points, a_mtx, b_mtx=numpy.array([]), also_b=False):
    """ Compute the influence that the ring vortex elements of the mesh have on a list of point c_pts
    The resultant matrice (a_mtx) is a matrix of influence coefficients.
    The matrice b_mtx can also be computed for v_wing induced velocity computation.
    b_mtx takes in count only chordwise vortex segments and the last mesh row that corresponds
    to the latest unsteady wake element """

    if fortran_flag:
        a_mtx, b_mtx = lib.ringsmtx(rings_mesh, points, also_b, True)

    else:
        m_nx = rings_mesh.shape[0]
        m_ny = rings_mesh.shape[1]
        c_nx = points.shape[0]
        c_ny = points.shape[1]

        # Chordwise loop through points
        for cp_i in xrange(c_nx):
            cp_loc_i = cp_i * (c_ny)
```

```
            # Spanwise loop through points
            for cp_j in xrange(c_ny):
                cp_loc = cp_j + cp_loc_i

                P = points[cp_i, cp_j]                          # local point

                # Chordwise loop through ring elements
                for el_i in xrange(m_nx - 1):
                    el_loc_i = el_i * (m_ny - 1)
                    # Spanwise loop through ring elements
                    for el_j in xrange(m_ny - 1):

                        el_loc = el_j + el_loc_i                # local ring element

                        A = rings_mesh[el_i, el_j + 0, :]       # ring element corners
                        B = rings_mesh[el_i, el_j + 1, :]
                        C = rings_mesh[el_i + 1, el_j + 1, :]
                        D = rings_mesh[el_i + 1, el_j + 0, :]

                        AB_vort = calc_vorticity(A, B, P)       # vortex segments
                        BC_vort = calc_vorticity(B, C, P)
                        CD_vort = calc_vorticity(C, D, P)
                        DA_vort = calc_vorticity(D, A, P)

                        # Influence of the other half wing (simmetry)
                        sym = numpy.array([1., -1., 1.])
                        P_sym = P * sym

                        AB_vort_sym = calc_vorticity(A, B, P_sym) * sym
                        BC_vort_sym = calc_vorticity(B, C, P_sym) * sym
                        CD_vort_sym = calc_vorticity(C, D, P_sym) * sym
                        DA_vort_sym = calc_vorticity(D, A, P_sym) * sym

                        # Computation of the influence coefficient that links the point with the element
                        a_mtx[cp_loc, el_loc, :] = AB_vort + BC_vort + CD_vort + DA_vort \
                                   + AB_vort_sym + BC_vort_sym + CD_vort_sym + DA_vort_sym

                        if also_b:
                            b_mtx[cp_loc, el_loc, :] = BC_vort + DA_vort + BC_vort_sym + DA_vort_sym
                            # Add influence of latest unsteady wake element
                            # (only in b_mtx, for induced drag computation)
                            if (el_i == m_nx - 2):
                                b_mtx[cp_loc, el_loc, :] += CD_vort + CD_vort_sym

        a_mtx /= 4 * numpy.pi
        if also_b:
            b_mtx /= 4 * numpy.pi

    return a_mtx, b_mtx



class Geometry(Component):
    """ Compute various geometric properties for VLM analysis
    def_mesh_t = deformed aero-mesh, at the time step t
    b_pts_t = vortex rings corners defined at quarters of panel 'chords', at the time step t
    c_pts_t = collocation points defined at three-quarters of panel 'chords', at the time step t
    starting_vortex_t = last row of b_pts matrix, representing the wake at t=0, at the time step t
    lenghts_t = lenght of aerodynamic panels, at the time step t
    widths_t = widhts of aerodynamic panels, at the time step t
    normals_t = normal vectors of aerodynamic panels, at the time step t
    v_local_t = local velocity of c_pts in the inertial frame, at the time step t
    S_ref = surface of the wing """

    def __init__(self, nx, n, t):
        super(Geometry, self).__init__()

        def_mesh_t = 'def_mesh_%d'%t
        b_pts_t = 'b_pts_%d'%t
        c_pts_t = 'c_pts_%d'%t
        starting_vortex_t = 'starting_vortex_%d'%t
        lenghts_t = 'lenghts_%d'%t
        widths_t = 'widths_%d'%t
        normals_t = 'normals_%d'%t
        v_local_t = 'v_local_%d'%t

        self.add_param('v', val=10.)
        self.add_param('alpha', val=3.)
        self.add_param('dt', val=0.1)
        self.add_param(def_mesh_t, val=numpy.zeros((nx, n, 3)))
        self.add_output(b_pts_t, val=numpy.zeros((nx, n, 3)))
        self.add_output(c_pts_t, val=numpy.zeros((nx-1, n-1, 3)))
```

```
        self.add_output(starting_vortex_t, val=numpy.zeros((1, n, 3)))
        self.add_output(lenghts_t, val=numpy.zeros((nx-1, n-1)), dtype="complex")
        self.add_output(widths_t, val=numpy.zeros((nx-1, n-1)))
        self.add_output(normals_t, val=numpy.zeros((nx-1, n-1, 3)), dtype="complex")
        self.add_output(v_local_t, val=numpy.zeros((nx-1, n-1, 3)), dtype="complex")
        if t == 0:
            self.add_output('S_ref', val=0.)

        self.deriv_options['form'] = 'central'

        self.num_x = nx
        self.num_y = n
        self.t = t

        self.unrotated_b = numpy.zeros((nx, n, 3))
        self.unrotated_c = numpy.zeros((nx-1, n-1, 3))
        self.all_lenghts = numpy.zeros((nx-1, n), dtype="complex")
        self.surf = val=numpy.zeros((nx-1, n-1))

        self.def_mesh_t = def_mesh_t
        self.b_pts_t = b_pts_t
        self.c_pts_t = c_pts_t
        self.starting_vortex_t = starting_vortex_t
        self.lenghts_t = lenghts_t
        self.widths_t = widths_t
        self.normals_t = normals_t
        self.v_local_t = v_local_t

    def get_lengths(self, A, B, axis):
        return numpy.sqrt(numpy.sum((B - A)**2, axis=axis))

    def solve_nonlinear(self, params, unknowns, resids):

        nx = self.num_x
        n = self.num_y
        t = self.t

        def_mesh_t = self.def_mesh_t
        b_pts_t = self.b_pts_t
        c_pts_t = self.c_pts_t
        starting_vortex_t = self.starting_vortex_t
        lenghts_t = self.lenghts_t
        widths_t = self.widths_t
        normals_t = self.normals_t
        v_local_t = self.v_local_t

        mesh = params[def_mesh_t]

        # distance of the last row of points chordwise from the trailing edge
        # last row of b_pts, defined with dist. This is also the starting vortex line
        dist_x = 0.3 * params['v'] * params['dt']
        self.unrotated_b[:-1, :, :] = mesh[:-1, :, :] * .75 + mesh[1:, :, :] * .25

        for j in xrange(n):
            self.unrotated_b[-1, j, :] = mesh[-1, j, :]
            self.unrotated_b[-1, j, 0] += dist_x

            self.unrotated_c = 0.5 * 0.25 * mesh[:-1,:-1, :] + \
                               0.5 * 0.75 * mesh[ 1:,:-1, :] + \
                               0.5 * 0.25 * mesh[:-1, 1:, :] + \
                               0.5 * 0.75 * mesh[ 1:, 1:, :]

        b_unrot = self.unrotated_b
        c_unrot = self.unrotated_c

        # Rotation of b_pts and c_pts due to the angle of attack alpha
        alpha_conv = params['alpha'] * numpy.pi / 180.
        cosa = numpy.cos(-alpha_conv)
        sina = numpy.sin(-alpha_conv)
        rot_x = numpy.array([cosa, 0, -sina])
        rot_z = numpy.array([sina, 0,  cosa])

        for i in xrange(nx):
            for j in xrange(n):
                unknowns[b_pts_t][i, j, 0] = b_unrot[i, j, :].dot(rot_x)
                unknowns[b_pts_t][i, j, 1] = b_unrot[i, j, 1]
                unknowns[b_pts_t][i, j, 2] = b_unrot[i, j, :].dot(rot_z)

        for i in xrange(nx-1):
            for j in xrange(n-1):
                unknowns[c_pts_t][i, j, 0] = c_unrot[i, j, :].dot(rot_x)
                unknowns[c_pts_t][i, j, 1] = c_unrot[i, j, 1]
```

```
                unknowns[c_pts_t][i, j, 2] = c_unrot[i, j, :].dot(rot_z)

        unknowns[starting_vortex_t] = unknowns[b_pts_t][-1, :, :]

        # lenght and widht of panels
        chords = self.get_lengths(mesh[-1, :, :], mesh[0, :, :], 1)
        self.all_lenghts[:-1, :] = self.get_lengths(b_unrot[1:-1, :, :], b_unrot[:-2, :, :], 2)
        # the total lenght have to be the chord lenght
        self.all_lenghts[-1, :] = self.get_lengths(mesh[-1, :, :] - b_unrot[-2, :, :], \
                                    mesh[0, :, :] - b_unrot[0, :, :], 1)

        unknowns[lenghts_t] = (self.all_lenghts[:, 1:] + self.all_lenghts[:, :-1])/2

        unknowns[widths_t] = self.get_lengths(b_unrot[:-1, 1:, :], b_unrot[:-1, :-1, :], 2)

        # Normals to the panels
        normals = numpy.cross(
        unknowns[b_pts_t][:-1,  1:, :] - unknowns[b_pts_t][ 1:, :-1, :],
        unknowns[b_pts_t][:-1, :-1, :] - unknowns[b_pts_t][ 1:,  1:, :], axis=2)

        norms = numpy.sqrt(numpy.sum(normals**2, axis=2))
        for ind in xrange(3):
            normals[:, :, ind] /= norms
        unknowns[normals_t] = normals

        # Panel velocity in the inertial frame
        for i in xrange(nx-1):
            for j in xrange(n-1):
                unknowns[v_local_t][i, j, 0] = params['v']

        # Surface of the half wing
        if t == 0:
            surf = unknowns[lenghts_t] * unknowns[widths_t]
            unknowns['S_ref'] = numpy.sum(surf)


class AIC(Component):
    """ Compute matrices of aerodynamic influence coefficients
    AIC_mtx = influence of wing vortex rings for wing panel circulations
    b_AIC_mtx = influence of wing chordwise vortex segments for wing induced velocity """

    def __init__(self, nx, n, t):
        super(AIC, self).__init__()

        b_pts_t = 'b_pts_%d'%t
        c_pts_t = 'c_pts_%d'%t
        AIC_mtx_t = 'AIC_mtx_%d'%t
        b_AIC_mtx_t = 'b_AIC_mtx_%d'%t

        self.add_param(b_pts_t, val=numpy.zeros((nx, n, 3)))
        self.add_param(c_pts_t, val=numpy.zeros((nx-1, n-1, 3)))
        size = (n-1) * (nx-1)
        self.add_output(AIC_mtx_t, val=numpy.zeros((size, size, 3), dtype="complex"))
        self.add_output(b_AIC_mtx_t, val=numpy.zeros((size, size, 3), dtype="complex"))

        self.deriv_options['form'] = 'central'

        self.a_mtx = numpy.zeros((size, size, 3), dtype="complex")
        self.b_mtx = numpy.zeros((size, size, 3), dtype="complex")

        self.b_pts_t = b_pts_t
        self.c_pts_t = c_pts_t
        self.AIC_mtx_t = AIC_mtx_t
        self.b_AIC_mtx_t = b_AIC_mtx_t

    def solve_nonlinear(self, params, unknowns, resids):

        b_pts_t = self.b_pts_t
        c_pts_t = self.c_pts_t
        AIC_mtx_t = self.AIC_mtx_t
        b_AIC_mtx_t = self.b_AIC_mtx_t

        self.a_mtx, self.b_mtx = assemble_AIC_matrices(params[b_pts_t], params[c_pts_t], self.a_mtx, self.b_mtx, True)

        unknowns[AIC_mtx_t] = self.a_mtx
        unknowns[b_AIC_mtx_t] = self.b_mtx


class Circulations(Component):
    """ Define wing and wake panels circulations
```

```
303     circ_t = wing panel circulations, at the time step t
        circ_wake_t = wake panel circulations, at the time step t
305     v_ind_wake_t = velocity of c_pts (wing panels) induced by wake panels, at the time step t """

307     def __init__(self, nx, n, nw, t):
            super(Circulations, self).__init__()
309
            lt = t - 1
311         c_pts_t = 'c_pts_%d'%t
            normals_t = 'normals_%d'%t
313         v_local_t = 'v_local_%d'%t
            AIC_mtx_t = 'AIC_mtx_%d'%t
315         v_ind_wake_t = 'v_ind_wake_%d'%t
            wake_mesh_t = 'wake_mesh_%d'%t
317         circ_t = 'circulations_%d'%t
            circ_lt = 'circulations_%d'%lt
319         circ_wake_t = 'circulations_wake_%d'%t
            circ_wake_lt = 'circulations_wake_%d'%lt
321
            if t < nw:
323             iw = t
            else:
325             iw = nw

327         self.add_param('dt', val=0.1)
            self.add_param(c_pts_t, val=numpy.zeros((nx-1, n-1, 3)))
329         self.add_param(normals_t, val=numpy.zeros((nx-1, n-1, 3), dtype="complex"))
            self.add_param(v_local_t, val=numpy.zeros((nx-1, n-1, 3), dtype="complex"))
331         size = (n-1) * (nx-1)
            self.add_param(AIC_mtx_t, val=numpy.zeros((size, size, 3), dtype="complex"))
333         self.add_state(circ_t, val=numpy.zeros((size), dtype="complex"))

335         # wake circulations
            # in the first step (dt=0) there is no wake and also there aren't previous results
337         # that have to be passed as input
            # in dt=1 there is the first evaluation of wake circulations
339         size_wake = (n-1) * t
            size_a1 = (n-1) * iw
341         if t > 0:
                self.add_param(wake_mesh_t, val=numpy.zeros((t+1, n, 3), dtype="complex"))
343             self.add_param(circ_lt, val=numpy.zeros((size), dtype="complex"))
                self.add_output(circ_wake_t, val=numpy.zeros((size_wake), dtype="complex"))
345             self.add_output(v_ind_wake_t, val=numpy.zeros((size, 3), dtype="complex"))

347             self.circ_wake = numpy.zeros((size_wake), dtype="complex")
                self.v_wake = numpy.zeros((size, 3), dtype="complex")
349             self.a1_mtx = numpy.zeros((size, size_a1, 3), dtype="complex")

351         # from dt=2 the wake circulations of the last dt have to be passed as input,
            # because we need to copy them
353         if t > 1:
                old_size_wake = (n-1) * lt
355             self.add_param(circ_wake_lt, val=numpy.zeros((old_size_wake), dtype="complex"))

357         self.deriv_options['form'] = 'central'

359         self.num_x = nx
            self.num_y = n
361         self.t = t
            self.iw = iw
363
            self.mtx = numpy.zeros((size, size), dtype="complex")
365         self.rhs = numpy.zeros((size), dtype="complex")

367         self.c_pts_t = c_pts_t
            self.normals_t = normals_t
369         self.v_local_t = v_local_t
            self.AIC_mtx_t = AIC_mtx_t
371         self.v_ind_wake_t = v_ind_wake_t
            self.wake_mesh_t = wake_mesh_t
373         self.circ_t = circ_t
            self.circ_lt = circ_lt
375         self.circ_wake_t = circ_wake_t
            self.circ_wake_lt = circ_wake_lt
377
        def _assemble_system(self, params, unknowns):
379         """ Define matrices for circulations """

381         normals_t = self.normals_t
            AIC_mtx_t = self.AIC_mtx_t
383         v_local_t = self.v_local_t
            v_ind_wake_t = self.v_ind_wake_t
```

```
385
        # mtx
387     self.mtx[:, :] = 0.
        for ind in xrange(3):
389         self.mtx[:, :] += (params[AIC_mtx_t][:, :, ind].T \
                               * params[normals_t][:, :, ind].flatten('C')).T
391
        # rhs
393     if (self.t == 0):
            v_c_pts = params[v_local_t].reshape(-1, params[v_local_t].shape[-1], order='C')
395     else:
            v_c_pts = params[v_local_t].reshape(-1, params[v_local_t].shape[-1], order='C') \
397                                       + unknowns[v_ind_wake_t]

399     norm = params[normals_t].reshape(-1, params[normals_t].shape[-1], order='C')

401     for k in xrange(self.rhs.shape[0]):
            self.rhs[k] = -norm[k].dot(v_c_pts[k])
403
    def solve_nonlinear(self, params, unknowns, resids):
405
        nx = self.num_x
407     n = self.num_y
        t = self.t
409     iw = self.iw
        c_pts_t = self.c_pts_t
411     v_local_t = self.v_local_t
        circ_t = self.circ_t
413     circ_lt = self.circ_lt
        circ_wake_t = self.circ_wake_t
415     circ_wake_lt = self.circ_wake_lt
        v_ind_wake_t = self.v_ind_wake_t
417     wake_mesh_t = self.wake_mesh_t

419     # wake circulations
        if t == 1:
421         unknowns[circ_wake_t] = params[circ_lt][(nx-2)*(n-1):]
        if t > 1:
423         self.circ_wake[:(n-1)] = params[circ_lt][(nx-2)*(n-1):]
            self.circ_wake[(n-1):] = params[circ_wake_lt][:]
425         unknowns[circ_wake_t] = self.circ_wake

427     # velocity of c_pts (wing panels) induced by wake panels (in inertial frame)
        if t > 0:
429         c_pts_inertial_frame = params[c_pts_t] - params[v_local_t] * params['dt'] * t

431         self.a1_mtx, _ = assemble_AIC_matrices(params[wake_mesh_t][:iw+1, :, :], c_pts_inertial_frame, self.a1_mtx)

433         for ind in xrange(3):
                self.v_wake[:, ind] = self.a1_mtx[:, :, ind].dot(unknowns[circ_wake_t][:(n-1)*iw])
435         unknowns[v_ind_wake_t] = self.v_wake

437     # wing circulations
        self._assemble_system(params, unknowns)
439     unknowns[circ_t] = numpy.linalg.solve(self.mtx, self.rhs)

441     resids = self.mtx.dot(unknowns[circ_t]) - self.rhs

443 def apply_nonlinear(self, params, unknowns, resids):

445     circ_t = self.circ_t
        self._assemble_system(params, unknowns)
447     circ = unknowns[circ_t]
        resids[circ_t] = self.mtx.dot(circ) - self.rhs
449
    def solve_linear(self, dumat, drmat, vois, mode=None):
451
        if mode == 'fwd':
453         sol_vec, rhs_vec = self.dumat, self.drmat
            t = 0
455     else:
            sol_vec, rhs_vec = self.drmat, self.dumat
457         t = 1

459     for voi in vois:
            sol_vec[voi].vec[:] = lu_solve(self.lup, rhs_vec[voi].vec, trans=t)
461


463
class InducedVelocities(Component):
465 """ Define induced velocities acting on each wing (v) or wake (w) panel
    v_ind_wing_t = velocity of c_pts (wing panels) induced by wing panels, at the time step t
```

```
    w_ind_t = velocity of wake_mesh (wake points) induced by wing and wake panels, at the time step t """

    def __init__(self, nx, n, nw, t):
        super(InducedVelocities, self).__init__()

        b_pts_t = 'b_pts_%d'%t
        b_AIC_mtx_t = 'b_AIC_mtx_%d'%t
        circ_t = 'circulations_%d'%t
        circ_wake_t = 'circulations_wake_%d'%t
        wake_mesh_t = 'wake_mesh_%d'%t
        v_ind_wing_t = 'v_ind_wing_%d'%t
        w_ind_t = 'w_ind_%d'%t

        if t < nw:
            iw = t
        else:
            iw = nw

        self.add_param('v', val=10.)
        self.add_param('dt', val=0.1)
        self.add_param(b_pts_t, val=numpy.zeros((nx, n, 3)))
        size = (nx-1) * (n-1)
        self.add_param(circ_t, val=numpy.zeros((size), dtype="complex"))
        self.add_param(b_AIC_mtx_t, val=numpy.zeros((size, size, 3), dtype="complex"))
        self.add_output(v_ind_wing_t, val=numpy.zeros((size, 3), dtype="complex"))

        size_wake_mesh = n * iw
        size_a3 = (n-1) * iw
        if t > 0:
            size_wake = (n-1) * t
            self.add_param(circ_wake_t, val=numpy.zeros((size_wake), dtype="complex"))
            self.add_param(wake_mesh_t, val=numpy.zeros((t+1, n, 3), dtype="complex"))
            self.add_output(w_ind_t, val=numpy.zeros((size_wake_mesh, 3), dtype="complex"))

            self.w_wake = numpy.zeros((size_wake_mesh, 3), dtype="complex")
            self.a3_mtx = numpy.zeros((size_wake_mesh, size_a3, 3), dtype="complex")

        self.deriv_options['form'] = 'central'

        self.num_y = n
        self.t = t
        self.iw = iw

        self.v_wing = numpy.zeros((size, 3), dtype="complex")
        self.w_wing = numpy.zeros((size_wake_mesh, 3), dtype="complex")
        self.a2_mtx = numpy.zeros((size_wake_mesh, size, 3), dtype="complex")

        if t < nw:
            self.wake_mesh_local_frame = numpy.zeros((t+1, n, 3), dtype="complex")
        else:
            self.wake_mesh_local_frame = numpy.zeros((nw+1, n, 3), dtype="complex")

        self.b_pts_t = b_pts_t
        self.b_AIC_mtx_t = b_AIC_mtx_t
        self.circ_t = circ_t
        self.circ_wake_t = circ_wake_t
        self.wake_mesh_t = wake_mesh_t
        self.v_ind_wing_t = v_ind_wing_t
        self.w_ind_t = w_ind_t

    def solve_nonlinear(self, params, unknowns, resids):

        n = self.num_y
        t = self.t
        iw = self.iw
        b_pts_t = self.b_pts_t
        b_AIC_mtx_t = self.b_AIC_mtx_t
        circ_t = self.circ_t
        circ_wake_t = self.circ_wake_t
        wake_mesh_t = self.wake_mesh_t
        v_ind_wing_t = self.v_ind_wing_t
        w_ind_t = self.w_ind_t

        # Wing (v)
        for ind in xrange(3):
            self.v_wing[:, ind] = params[b_AIC_mtx_t][:, :, ind].dot(params[circ_t])

        unknowns[v_ind_wing_t] = self.v_wing

        # Wake rollup (w)
        if t > 0:
```

```
549             # position of the wake mesh in local frame
                translation = numpy.array([params['v'] * params['dt'] * t, 0., 0.])
551             for i in xrange(self.wake_mesh_local_frame.shape[0]):
                    for j in xrange(n):
553                     self.wake_mesh_local_frame[i, j, :] = params[wake_mesh_t][i, j, :] \
                                                          + translation[:]
555
                self.a2_mtx, _ = assemble_AIC_matrices(params[b_pts_t], self.wake_mesh_local_frame[1:,:,], self.a2_mtx)
557             self.a3_mtx, _ = assemble_AIC_matrices(params[wake_mesh_t][:(iw+1), :, :],
                                       params[wake_mesh_t][1:(iw+1), :, :], self.a3_mtx)
559
                for ind in xrange(3):
561                 self.w_wing[:, ind] = self.a2_mtx[:, :, ind].dot(params[circ_t])
                    self.w_wake[:, ind] = self.a3_mtx[:, :, ind].dot(params[circ_wake_t][:iw*(n-1)])
563
                unknowns[w_ind_t] = self.w_wing + self.w_wake
565


567
    class WakeGeometry(Component):
569     """ Update position of wake mesh in the body frame, adding a line for each time step
        wake_mesh_nt = position of wake mesh points (wake rings corners), at the time step t+1 """
571
        def __init__(self, nx, n, nw, t):
573         super(WakeGeometry, self).__init__()

575         nt = t + 1
            starting_vortex_t = 'starting_vortex_%d'%t
577         wake_mesh_t = 'wake_mesh_%d'%t
            wake_mesh_nt = 'wake_mesh_%d'%nt
579         w_ind_t = 'w_ind_%d'%t

581         if t < nw:
                iw = t
583         else:
                iw = nw
585
            self.add_param('v', val=10.)
587         self.add_param('dt', val=0.1)
            self.add_param(starting_vortex_t, val=numpy.zeros((1, n, 3), dtype="complex"))
589         self.add_output(wake_mesh_nt, val=numpy.zeros((t+2, n, 3), dtype="complex"))

591         size_wake_mesh = n * iw
            if t > 0:
593             self.add_param(w_ind_t, val=numpy.zeros((size_wake_mesh, 3), dtype="complex"))
                self.add_param(wake_mesh_t, val=numpy.zeros((t+1, n, 3), dtype="complex"))
595
            self.deriv_options['form'] = 'central'
597
            self.num_y = n
599         self.t = t
            self.nw = nw
601         self.iw = iw

603         self.w_ind_t_resh = numpy.zeros((iw, n, 3), dtype="complex")
            self.new_wake_row = numpy.zeros((1, n, 3), dtype="complex")
605         self.old_wake = numpy.zeros((t+1, n, 3), dtype="complex")

607         self.starting_vortex_t = starting_vortex_t
            self.w_ind_t = w_ind_t
609         self.wake_mesh_t = wake_mesh_t
            self.wake_mesh_nt = wake_mesh_nt
611
        def solve_nonlinear(self, params, unknowns, resids):
613
            n = self.num_y
615         t = self.t
            nw = self.nw
617         iw = self.iw
            starting_vortex_t = self.starting_vortex_t
619         w_ind_t = self.w_ind_t
            wake_mesh_t = self.wake_mesh_t
621         wake_mesh_nt = self.wake_mesh_nt

623         # reshape of w_ind_t
            if t > 0:
625             for ind in xrange(3):
                    self.w_ind_t_resh[:, :, ind] = params[w_ind_t][:, ind].reshape(iw, n, order='C')
627
            # copy of old wake + addition of induced velocity
629         if t == 0:
                self.old_wake = params[starting_vortex_t]
```

```
631         else:
                self.old_wake = params[wake_mesh_t]
633
                unknowns[wake_mesh_nt][2:(iw+2), :, :] = self.old_wake[1:(iw+1), :, :] \
635                                                 + self.w_ind_t_resh * params['dt']
                unknowns[wake_mesh_nt][(iw+2):, :, :] = self.old_wake[(iw+1):, :, :]
637
            unknowns[wake_mesh_nt][1, :, :] = self.old_wake[0, :, :]
639
            # addition of a new wake row
641         for j in xrange(n):
                self.new_wake_row[0, j, 0] = params[starting_vortex_t][0, j, 0] - params['v'] * params['dt'] * (t+1)
643             self.new_wake_row[0, j, 1] = params[starting_vortex_t][0, j, 1]
                self.new_wake_row[0, j, 2] = params[starting_vortex_t][0, j, 2]
645
            unknowns[wake_mesh_nt][0, :, :] = self.new_wake_row
647


649
    class Forces(Component):
651     """ Define aerodynamic forces acting on each wing panel, evaluated by unsteady Bernoulli formula
        forces_L_t = lift acting on each wing panel, at the time step t
653     forces_D_t = induced drag acting on each wing panel, at the time step t
        sec_forces_t = sum of all the forces in each section. It is used in TransferForces """
655
        def __init__(self, nx, n, t):
657
            super(Forces, self).__init__()
659
            lt = t - 1
661         lenghts_t = 'lenghts_%d'%t
            widths_t = 'widths_%d'%t
663         normals_t = 'normals_%d'%t
            v_local_t = 'v_local_%d'%t
665         circ_t = 'circulations_%d'%t
            circ_lt = 'circulations_%d'%lt
667         v_ind_wing_t = 'v_ind_wing_%d'%t
            v_ind_wake_t = 'v_ind_wake_%d'%t
669         sigma_x_t = 'sigma_x_%d'%t
            sigma_x_lt = 'sigma_x_%d'%lt
671         sec_L_t = 'sec_L_%d'%t
            sec_D_t = 'sec_D_%d'%t
673         sec_forces_t = 'sec_forces_%d'%t

675         self.add_param('rho', val=3.)
            self.add_param('alpha', val=3.)
677         self.add_param('dt', val=0.1)
            self.add_param(lenghts_t, val=numpy.zeros((nx-1, n-1)))
679         self.add_param(widths_t, val=numpy.zeros((nx-1, n-1)))
            self.add_param(normals_t, val=numpy.zeros((nx-1, n-1, 3), dtype="complex"))
681         self.add_param(v_local_t, val=numpy.zeros((nx-1, n-1, 3), dtype="complex"))
            size = (nx-1) * (n-1)
683         self.add_param(circ_t, val=numpy.zeros((size), dtype="complex"))
            self.add_param(v_ind_wing_t, val=numpy.zeros((size, 3), dtype="complex"))
685
            self.add_output(sigma_x_t, val=numpy.zeros((nx-1, n-1), dtype="complex"))
687         self.add_output(sec_L_t, val=numpy.zeros((nx-1, n-1), dtype="complex"))
            self.add_output(sec_D_t, val=numpy.zeros((nx-1, n-1), dtype="complex"))
689         self.add_output(sec_forces_t, val=numpy.zeros((n-1, 3), dtype="complex"))

691         if t > 0:
                size_wake = (n-1) * t
693             self.add_param(v_ind_wake_t, val=numpy.zeros((size, 3), dtype="complex"))
                self.add_param(sigma_x_lt, val=numpy.zeros((nx-1, n-1), dtype="complex"))
695
            self.deriv_options['form'] = 'central'
697
            self.num_y = n
699         self.num_x = nx
            self.t = t
701
            self.velo = numpy.zeros((nx-1, n-1, 3), dtype="complex")
703         self.vind = numpy.zeros((nx-1, n-1, 3), dtype="complex")
            self.circ_mtx_x = numpy.zeros((nx-1, n-1), dtype="complex")
705         self.circ_mtx_y = numpy.zeros((nx-1, n-1), dtype="complex")
            self.dCirc_dt = numpy.zeros((nx-1, n-1), dtype="complex")
707         self.forces_L = numpy.zeros((nx-1, n-1), dtype="complex")
            self.forces_D = numpy.zeros((nx-1, n-1), dtype="complex")
709         self.sum_sections = numpy.zeros((n-1, 3), dtype="complex")

711         self.lenghts_t = lenghts_t
            self.widths_t = widths_t
```

```
        self.normals_t = normals_t
        self.v_local_t = v_local_t
        self.circ_t = circ_t
        self.v_ind_wing_t = v_ind_wing_t
        self.v_ind_wake_t = v_ind_wake_t
        self.sigma_x_t = sigma_x_t
        self.sigma_x_lt = sigma_x_lt
        self.sec_L_t = sec_L_t
        self.sec_D_t = sec_D_t
        self.sec_forces_t = sec_forces_t

    def solve_nonlinear(self, params, unknowns, resids):

        n = self.num_y
        nx = self.num_x
        t = self.t
        lenghts_t = self.lenghts_t
        widths_t = self.widths_t
        normals_t = self.normals_t
        v_local_t = self.v_local_t
        circ_t = self.circ_t
        v_ind_wing_t = self.v_ind_wing_t
        v_ind_wake_t = self.v_ind_wake_t
        sigma_x_t = self.sigma_x_t
        sigma_x_lt = self.sigma_x_lt
        sec_L_t = self.sec_L_t
        sec_D_t = self.sec_D_t
        sec_forces_t = self.sec_forces_t

        if t == 0:
            self.velo = params[v_local_t]
            self.vind = params[v_ind_wing_t][:, 2].reshape(nx-1, n-1, order='C')

        else:
            for ind in xrange(3):
                self.velo[:, :, ind] = params[v_local_t][:, :, ind] \
                    + params[v_ind_wake_t][:, ind].reshape(nx-1, n-1, order='C')
                self.vind = params[v_ind_wing_t][:, 2].reshape(nx-1, n-1, order='C') \
                    + params[v_ind_wake_t][:, 2].reshape(nx-1, n-1, order='C')

        circ_mtx = params[circ_t].reshape(nx-1, n-1, order='C')

        self.circ_mtx_x[0, :] = circ_mtx[0, :]
        self.circ_mtx_x[1:, :] = circ_mtx[1:, :] - circ_mtx[:-1, :]

        # Velocity-potential time derivative (dCirc_dt) is obtained by integrating
        # from the leading edge

        unknowns[sigma_x_t][0, :] = (0.5 * self.circ_mtx_x[0, :]) * params[lenghts_t][0, :]
        unknowns[sigma_x_t][1:, :] = (0.5 * self.circ_mtx_x[1:, :] + circ_mtx[:-1, :]) \
                                     * params[lenghts_t][1:, :]

        if t == 0:
            self.dCirc_dt = unknowns[sigma_x_t] / params['dt']
        else:
            self.dCirc_dt = (unknowns[sigma_x_t] - params[sigma_x_lt]) / params['dt']

        # Lift for each panel
        self.forces_L = params['rho'] * (self.velo[:, :, 0] * self.circ_mtx_x \
             + self.velo[:, :, 1] * self.circ_mtx_y + self.dCirc_dt) * params[widths_t] \
             * params[normals_t][:, :, 2]

        # Induced drag for each panel
        self.forces_D = params['rho'] * (-self.vind * self.circ_mtx_x
             + self.dCirc_dt * params[normals_t][:, :, 0]) * params[widths_t]

        unknowns[sec_L_t] = self.forces_L
        unknowns[sec_D_t] = self.forces_D

        # section forces for structural part
        projected_forces = numpy.array(params[normals_t], dtype="complex")
        for ind in xrange(3):
            projected_forces[:, :, ind] *= self.forces_L

        unknowns[sec_forces_t] = numpy.zeros((n-1, 3))
        for x in xrange(nx-1):
            self.sum_sections += projected_forces[x, :, :]
        unknowns[sec_forces_t] = self.sum_sections
        #F
        #print 'sum_sections', self.sum_sections

```

```
795
class LiftDrag(Component):
797     """ Calculate total lift and drag in force units based on section forces
        L = total lift of the wing, after the time loop
799     D = total induced drag of the wing, after the time loop """

801     def __init__(self, nx, n, num_dt):
            super(LiftDrag, self).__init__()
803
            sec_L_end = 'sec_L_%d'%(num_dt-1)
805         sec_D_end = 'sec_D_%d'%(num_dt-1)

807         self.add_param(sec_L_end, val=numpy.zeros((nx-1, n-1), dtype="complex"))
            self.add_param(sec_D_end, val=numpy.zeros((nx-1, n-1), dtype="complex"))
809         self.add_output('L', val=0.)
            self.add_output('D', val=0.)
811
            self.deriv_options['form'] = 'central'
813         #self.deriv_options['extra_check_partials_form'] = "central"

815         self.sec_L_end = sec_L_end
            self.sec_D_end = sec_D_end
817
        def solve_nonlinear(self, params, unknowns, resids):
819
            sec_L_end = self.sec_L_end
821         sec_D_end = self.sec_D_end

823         unknowns['L'] = numpy.sum(params[sec_L_end])
            unknowns['D'] = numpy.sum(params[sec_D_end])
825


827
class AeroCoeffs(Component):
829     """ Compute lift and drag coefficients
        CL1 = lift coefficient of the wing due to wing and wake circulations
831     CDi = drag coefficient of the wing due to wing and wake circulations """

833     def __init__(self):
            super(AeroCoeffs, self).__init__()
835
            self.add_param('S_ref', val=0.)
837         self.add_param('rho', val=0.)
            self.add_param('v', val=0.)
839         self.add_param('L', val=0.)
            self.add_param('D', val=0.)
841         self.add_output('CL1', val=0.)
            self.add_output('CDi', val=0.)
843
            self.deriv_options['form'] = 'central'
845         #self.deriv_options['extra_check_partials_form'] = "central"

847     def solve_nonlinear(self, params, unknowns, resids):

849         S_ref = params['S_ref']
            rho = params['rho']
851         v = params['v']
            L = params['L']
853         D = params['D']

855         unknowns['CL1'] = L / (0.5*rho*v**2*S_ref)
            unknowns['CDi'] = D / (0.5*rho*v**2*S_ref)
857


859
class TotalLift(Component):
861     """ Calculate total lift in force units
        CL = CL1 + CL0 = total lift coefficient of the wing """
863
        def __init__(self, CL0):
865         super(TotalLift, self).__init__()

867         self.add_param('CL1', val=1.)
            self.add_output('CL', val=1.)
869
            self.deriv_options['form'] = 'central'
871
            self.CL0 = CL0
873
        def solve_nonlinear(self, params, unknowns, resids):
875
            unknowns['CL'] = params['CL1'] + self.CL0
```

```
877
879
class TotalDrag(Component):
    """ Calculate total drag in force units
    CD = CDi + CD0 = total drag coefficient of the wing """
883
    def __init__(self, CD0):
885     super(TotalDrag, self).__init__()

887     self.add_param('CDi', val=1.)
        self.add_output('CD', val=1.)
889
        self.deriv_options['form'] = 'central'
891
        self.CD0 = CD0
893
    def solve_nonlinear(self, params, unknowns, resids):
895
        unknowns['CD'] = params['CDi'] + self.CD0
897


899
class UVLMStates(Group):
901     """ Group that contains the aerodynamic states """

903     def __init__(self, num_x, num_y, num_w, t):
        super(UVLMStates, self).__init__()
905
        name_wgeom = 'wgeom_%d'%t
907     name_aic = 'aic_%d'%t
        name_circ = 'circ_%d'%t
909     name_indvel = 'indvel_%d'%t
        name_wakegeom = 'wakegeom_%d'%t
911     name_forces = 'forces_%d'%t

913     self.add(name_wgeom,
                 Geometry(num_x, num_y, t),
915              promotes=['*'])
        self.add(name_aic,
917              AIC(num_x, num_y, t),
                 promotes=['*'])
919     self.add(name_circ,
                 Circulations(num_x, num_y, num_w, t),
921              promotes=['*'])
        self.add(name_indvel,
923              InducedVelocities(num_x, num_y, num_w, t),
                 promotes=['*'])
925     self.add(name_wakegeom,
                 WakeGeometry(num_x, num_y, num_w, t),
927              promotes=['*'])
        self.add(name_forces,
929              Forces(num_x, num_y, t),
                 promotes=['*'])
931


933
class UVLMFunctionals(Group):
935     """ Group that contains the aerodynamic functionals used to evaluate performance """

937     def __init__(self, num_x, num_y, CL0, CD0, num_dt):
        super(UVLMFunctionals, self).__init__()
939
        self.add('liftdrag',
941              LiftDrag(num_x, num_y, num_dt),
                 promotes=['*'])
943     self.add('aero_coeffs',
             AeroCoeffs(),
945              promotes=['*'])
        self.add('total_CL',
947              TotalLift(CL0),
                 promotes=['*'])
949     self.add('total_CD',
                 TotalDrag(CD0),
951              promotes=['*'])
```

Appendix B

Tuned Mass Damper (TMD)

A tuned mass damper (TMD) is a device mounted in structures to reduce the amplitude of mechanical vibrations. Their application can prevent discomfort, damage, or outright structural failure. Tuned mass dampers stabilize against violent motion caused by harmonic vibration. A TMD reduces the vibration of a system with a comparatively lightweight component so that the worst-case vibrations are less intense. Roughly speaking, practical systems are tuned to either move the main mode away from a troubling excitation frequency or to add damping to a resonance that is difficult or expensive to damp directly.

In the aeronautical field, it is possible to use a TMD in order to increase the flutter velocity of the system. We simply need to set the TMD frequency equal to the frequency where torsional and bending modes are coupled. This effectively moves these frequencies off of each other, delaying the aeroelastic instability to higher flow velocities.

In mathematics, the presence of a TMD consists of the addition of one degree of freedom, usually the displacement of the mass damper in the direction of application. In order to improve the flutter condition, this direction is perpendicular to the y axis of the wing. In this way the tuned mass damps the bending and torsional oscillations. In a simplified formulation, the TMD influences only the node of the spatial beam representing the section of application. Then, in the matrix representation of the dynamic system we have to add a row (and a column) for the TMD equation and modify the two lines of the beam node that correspond to its vertical displacement and its rotation around the beam axis. Following [60], the three equations could be easily written as:

$$(m_1 + m_2)\ddot{u}(t) + m_2\ddot{w}(t) - \eta m_2\ddot{\phi}(t) + c_1\dot{u}(t) + k_1 u(t) = 0 \tag{B.1a}$$

$$-\eta m_2\ddot{u}(t) - \eta m_2\ddot{w}(t) + (m_t + \eta^2 m_2)\ddot{\phi}(t) + c_3\dot{\phi}(t) + k_3\phi(t) = 0 \tag{B.1b}$$

$$m_2\ddot{u}(t) + m_2\ddot{w}(t) - \eta m_2\ddot{\phi}(t) + c_2\dot{w}(t) + k_2 w(t) = 0 \tag{B.1c}$$

where:

- m_1, c_1 and k_1 are mass, damping coefficient and stiffness relatives to the beam vertical displacement u
- m_t, c_2 and k_2 are are the same for the beam torsional degree of freedom ϕ
- m_3, c_3 and k_3 are are the same for the TMD relative displacement w
- η is the spanwise distance between the TMD and the half wing root

Bibliography

[1] AR Collar. The expanding domain of aeroelasticity. *Journal of the Royal Aeronautical Society*, 50(428):613–636, 1946.

[2] Gaetan KW Kenway and Joaquim RRA Martins. Multipoint high-fidelity aerostructural optimization of a transport aircraft configuration. *Journal of Aircraft*, 51(1): 144–160, 2014.

[3] Joseph Katz and Allen Plotkin. *Low-speed aerodynamics*, volume 13. Cambridge University Press, 2001.

[4] John J Bertin and Michael L Smith. Aerodynamics for engineers. 1998.

[5] John David Anderson Jr. *Fundamentals of aerodynamics*. Tata McGraw-Hill Education, 2010.

[6] DM Tang and EH Dowell. Effects of geometric structural nonlinearity on flutter and limit cycle oscillations of high-aspect-ratio wings. *Journal of fluids and structures*, 19(3):291–306, 2004.

[7] Robert Clark, David Cox, C Howard Jr, John W Edwards, Kenneth C Hall, David A Peters, Robert Scanlan, Emil Simiu, Fernando Sisto, Thomas W Strganac, et al. *A modern course in aeroelasticity*, volume 116. Springer Science & Business Media, 2006.

[8] Andrea Arena, Walter Lacarbonara, and Pier Marzocca. Nonlinear aeroelastic formulation and postflutter analysis of flexible high-aspect-ratio wings. *Journal of Aircraft*, 50(6):1748–1764, 2013.

[9] Raymond L Bisplinghoff and Holt Ashley. Half man, rl, aeroelasticity, 1955.

[10] YC Fung. The theory of aeroelasticity. *John Wily Sons*, 1955.

[11] John David Anderson and J Wendt. *Computational fluid dynamics*, volume 206. Springer, 1995.

[12] T Francis Ogilvie and Ernest O Tuck. A rational strip theory of ship motions: part i. Technical report, 1969.

[13] Thiemo M Kier. Comparison of unsteady aerodynamic modelling methodologies with respect to flight loads analysis. *AIAA Paper*, 6027:2005, 2005.

[14] Rafael Palacios, Joseba Murua, and Robert Cook. Structural and aerodynamic models in nonlinear flight dynamics of very flexible aircraft. *AIAA journal*, 48(11): 2648–2659, 2010.

[15] Joseba Murua, Rafael Palacios, and J Michael R. Graham. Assessment of wake-tail interference effects on the dynamics of flexible aircraft. *AIAA journal*, 50(7): 1575–1585, 2012.

[16] George Keith Batchelor. *An introduction to fluid dynamics*. Cambridge university press, 2000.

[17] L Rosenhead. The formation of vortices from a surface of discontinuity. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 134(823):170–192, 1931.

[18] Victor Montague Falkner. The calculation of aerodynamic loading on surfaces of any shape. Technical report, DTIC Document, 1943.

[19] Sven G Hedman. Vortex lattice method for calculation of quasi steady state loadings on thin elastic wings in subsonic flow. Technical report, DTIC Document, 1966.

[20] Richard M James. On the remarkable accuracy of the vortex lattice method. *Computer methods in applied mechanics and engineering*, 1(1):59–79, 1972.

[21] Dean T Mook and Ali H Nayfeh. Application of the vortex-lattice method to high-angle-of-attack subsonic aerodynamics. Technical report, SAE Technical Paper, 1985.

[22] Edward C Polhamus. A concept of the vortex lift of sharp-edge delta wings based on a leading-edge-suction analogy. 1966.

[23] Edward Albano and William P Rodden. A doublet-lattice method for calculating lift distributions on oscillating surfaces in subsonic flows. *AIAA journal*, 7(2):279–285, 1969.

[24] Jan Robert Wright and Jonathan Edward Cooper. *Introduction to aircraft aeroelasticity and loads*, volume 20. John Wiley & Sons, 2008.

[25] John L Hess. Calculation of potential flow about arbitrary three-dimensional lifting bodies. Technical report, DTIC Document, 1972.

[26] Lyle N Long and Tracy E Fritz. Object-oriented unsteady vortex lattice method for flapping flight. *Journal of Aircraft*, 41(6):1275–1290, 2004.

[27] A Röttgermann, R Behr, Ch Schöttl, and S Wagner. Calculation of blade-vortex interaction of rotary wings in incompressible flow by an unsteady vortex-lattice method including free wake analysis. In *Numerical Techniques for Boundary Element Methods*, pages 153–166. Springer, 1992.

[28] Mads Døssing. *Vortex lattice modelling of winglets on wind turbine blades*. PhD thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2007.

[29] Reza Karkehabadi. Aerodynamic interference of a large and a small aircraft. *Journal of Aircraft*, 41(6):1424–1429, 2004.

[30] Benjamin D Hall, Dean T Mook, Ali H Nayfeh, and Sergio Preidikman. Novel strategy for suppressing the flutter oscillations of aircraft wings. *AIAA journal*, 39 (10):1843–1850, 2001.

[31] Zhicun Wang, PC Chen, DD Liu, DT Mook, and MJ Patil. Time domain nonlinear aeroelastic analysis for hale wings. *AIAA paper*, 1640:2006, 2006.

[32] Bret Stanford and Philip Beran. Formulation of analytical design derivatives for nonlinear unsteady aeroelasticity. *AIAA journal*, 49(3):598–610, 2011.

[33] Joseba Murua, Rafael Palacios, and J Michael R. Graham. Assessment of wake-tail interference effects on the dynamics of flexible aircraft. *AIAA journal*, 50(7): 1575–1585, 2012.

[34] John C Vassberg, Mark A DeHaan, S Melissa Rivers, and Richard A Wahls. Development of a common research model for applied cfd validation studies. *AIAA paper*, 6919:2008, 2008.

[35] Justin Gray, Kenneth T Moore, and Bret A Naylor. Openmdao: An open source framework for multidisciplinary analysis and optimization. In *AIAA/ISSMO Multidisciplinary Analysis Optimization Conference Proceedings*, volume 5, 2010.

[36] *OpenAeroStruct* link. https://github.com/johnjasa/OpenAeroStruct, .

[37] Singiresu S Rao. *The finite element method in engineering*. Elsevier, 2010.

[38] Gui-Rong Liu and Siu Sin Quek. *The finite element method: a practical course*. Butterworth-Heinemann, 2013.

[39] Henri P Gavin. Structural element stiffness, mass, and damping matrices. *Duke University*, 2014.

[40] Ahid D Nashif, David IG Jones, and John P Henderson. *Vibration damping*. John Wiley & Sons, 1985.

[41] Nathan Mortimore Newmark. A method of computation for structural dynamics. In *Proc. ASCE*, volume 85, pages 67–94, 1959.

[42] Granino Arthur Korn and Theresa M Korn. *Mathematical handbook for scientists and engineers: Definitions, theorems, and formulas for reference and review*. Courier Corporation, 2000.

[43] Arnold M Kuethe and Chuen-Yen Chow. *Fondations of aerodynamics*. John Wiley & Sons, 1976.

[44] Joseph Pedlosky. *Geophysical fluid dynamics*. Springer Science & Business Media, 2013.

[45] A Rizzi. Background documentation for software&labwork course sd2610: Computational aerodynamics in aircraft design. *KTH, Aeronautical and Vehicle Engineering Department*, 2009.

[46] Enrique Mata Bueso. Unsteady aerodynamic vortex lattice of moving aircraft. 2011.

[47] John D Anderson. Jr,"introduction to flight", 1989.

[48] Pijush K Kundu and Ira M Cohen. Fluid mechanics 4th, 2008.

[49] James Lighthill. An informal introduction to theoretical fluid mechanics. 1986.

[50] José Meseguer Ruiz and Ángel Sanz Andrés. *Aerodinámica básica*. 2005.

[51] E Pistolesi. Ground effect-theory and practice. 1937.

[52] C Edward Lan. A quasi-vortex-lattice method in thin wing theory. *Journal of Aircraft*, 11(9):518–527, 1974.

[53] Bernard Etkin and Lloyd Duff Reid. *Dynamics of flight: stability and control*, volume 3. Wiley New York, 1996.

[54] Openmdao link. http://openmdao.readthedocs.io/en/1.7.2/index.html, .

[55] Mayuresh J Patil, Dewey H Hodges, and Carlos E S. Cesnik. Nonlinear aeroelasticity and flight dynamics of high-altitude long-endurance aircraft. *Journal of Aircraft*, 38(1):88–94, 2001.

[56] Martin Goland. The flutter of a uniform cantilever wing. *JOURNAL OF APPLIED MECHANICS-TRANSACTIONS OF THE ASME*, 12(4):A197–A208, 1945.

[57] Mayuresh J Patil and Dewey H Hodges. Nonlinear aeroelasticity and flight dynamics of aircraft in subsonic flow. In *Proceedings of the 21th Congress of International Council of the Aeronautical Sciences, Melbourne, Australia (September 1998)*, 1998.

[58] William L Hosch. *The Britannica Guide to Numbers and Measurement*. The Rosen Publishing Group, 2010.

[59] David A Peters and Mark J Johnson. Finite-state airloads for deformable airfoils on fixed and rotating wings. *ASME-PUBLICATIONS-AD*, 44:1–1, 1994.

[60] Walter Lacarbonara and Marek Cetraro. Flutter control of a lifting surface via visco-hysteretic vibration absorbers. *International Journal Aeronautical and Space Sciences*, 12(4):331–345, 2011.