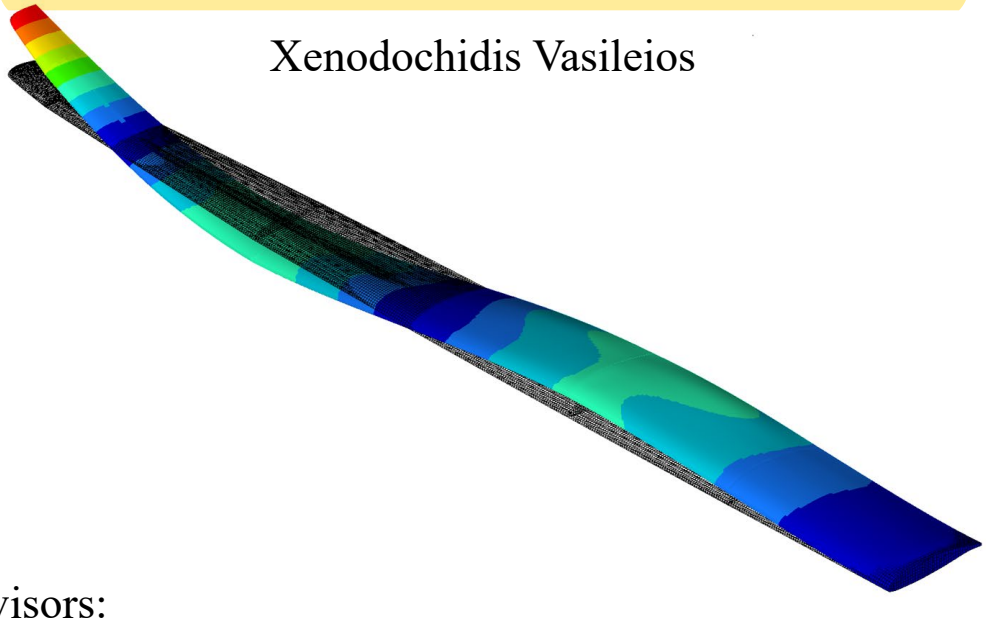
Aristotle University of Thessaloniki

Department of Mechanical Engineering

Aeroelastic Flutter Optimization of Wing Structures

Note that this is only a draft of the thesis and does not fully represent the final work.
Small corrections and a final review are still in order!

Xenodochidis Vasileios



Supervisors:

Dr. Dimitrios Giagopoulos, Associate Professor

Josef Koutsoupakis Ph.D. Candidate

Submitted in part fulfilment of the requirements for the
degree of Master of Science in Mechanical Engineering of the
Aristotle University of Thessaloniki, February 2025

Abstract

This thesis presents an aeroelastic analysis of a wing structure with the objective of optimizing its composite material to maximize flutter speed while minimizing mass. The study utilizes Altair's OptiStruct solver to compute flutter curves and assess the aeroelastic stability of the structure. Various optimization algorithms, implemented in Python, were employed to explore the design space and identify optimal configurations that balance structural mass and aeroelastic performance. Furthermore, a neural network model was developed to predict flutter speed based on key structural and material parameters, to see the potential that a surrogate model has in predicting the flutter speed. The results demonstrate significant improvements in both flutter speed and weight reduction, highlighting the potential of advanced optimization techniques and machine learning in aeroelastic design.

Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει μια αεροελαστική ανάλυση μιας πτέρυγας με στόχο τη βελτιστοποίηση του πολυστρωματικού σύνθετου υλικού της για τη μεγιστοποίηση της ταχύτητας πτερυγισμού ελαχιστοποιώντας παράλληλα τη μάζα. Η μελέτη χρησιμοποιεί τον επιλυτή OptiStruct της Altair για να αξιολογήσει την αεροελαστική σταθερότητα της δομής. Διάφοροι αλγόριθμοι βελτιστοποίησης, που υλοποιήθηκαν στην Python, χρησιμοποιήθηκαν για να εξερευνήσουν το χώρο σχεδιασμού και να εντοπίσουν βέλτιστους συνδυασμούς παραμέτρων που παράγουν αποδεκτά χαρακτηριστικά πτερυγισμού ενώ ταυτόχρονα ελαχιστοποιούν τη μάζα της κατασκευής. Επιπλέον, αναπτύχθηκε ένα μοντέλο νευρωνικού δικτύου για την πρόβλεψη της ταχύτητας πτερυγισμού με βάση βασικές δομικές και υλικές παραμέτρους, για να αναδειχθούν οι δυνατότητες που έχει ένα υποκατάστατο μοντέλο στην πρόβλεψη της ταχύτητας πτερυγισμού. Τα αποτελέσματα δείχνουν σημαντικές βελτιώσεις τόσο στην ταχύτητα πτερυγισμού όσο και στη μείωση βάρους, τονίζοντας τις δυνατότητες των προηγμένων τεχνικών βελτιστοποίησης και της μηχανικής μάθησης στον αεροελαστικό σχεδιασμό.

Table of Contents

List of Tables

List of Figures

1 Introduction

The Introduction chapter describes the aim of the thesis. It also defines the limits and scope of this work. Finally, it describes the motivation behind this project.

1.1 Problem Statement

Aeroelasticity is a branch of physics and engineering which studies the response of elastic bodies exposed to a fluid flow. The forces involved in this interaction are inertial, elastic and aerodynamic. Aeroelastic problems in engineering can be classified into two broad categories: static aeroelasticity which deals with the steady state response, and dynamic aeroelasticity dealing mainly with the body's vibrational response.

The most common aeroelastic effects encountered by aircraft are:

- *Aerodynamic Divergence*, where the deflection of lifting surfaces of an aircraft leads to additional lift that, in turn, leads to further deflection in the same direction, resulting in excessive stress or even leading to structural failure
- *Aeroelastic Control reversal,* where the forces generated by the control aileron (responsible for the roll control of an aircraft) are sufficient to twist the wing itself to such an extent that it changes the lift characteristics of the wing which makes the control surfaces ineffective or even produces the opposite of the expected result
- *Aeroelastic Flutter* is a dynamic instability of a structure that occurs due to the interaction of the fluid flow with the eigenmodes of the structure

In this thesis the flutter characteristics of a lifting surface will be explored and tailored to specific requirements using optimization techniques.

1.2 Objective

The objective of this project is to develop an understanding of the flutter characteristics of a lifting surface made of a laminate composite material and the methods used to computationally predict the flutter region using Altair's Optistruct solver. Thereafter the effect of several structural parameters is studied to determine their effect on the flutter characteristics and the effectiveness of several optimization techniques is tested to tailor the flutter characteristics to a set of requirements using Python.

1.3 Scope and Limitations

The project will include:

- The study of composite materials and their implementation in the Optistruct solver
- The study of the Vortex Lattice panel Method and how it is used during the Aerodynamic Flutter Analysis
- The study of Aeroelastic Flutter Analysis on a theoretical basis by means of the governing equations of motion and on a computational basis by studying the implementation of theory into commercial solvers.
- Investigation of the effect of composite material properties on the flutter characteristics
- Investigation of different optimization techniques to manipulate the flutter characteristics including line search methods, genetic algorithms and Neural Networks

1.4 Motivation

In the aerospace industry minimization of weight of structures is of utmost importance, since it allows the increase of useful payload, improves efficiency maneuverability and other control characteristics. A consequence of structural weight minimization is the reduction of safety margins in comparison to other engineering fields, thus the need for more precise calculations arises to ensure safety.

As aerospace prototype testing costs are elevated computer simulations are used more and more and become ever more advanced with the ability to study a wider range of phenomena. This project would help students and researchers understand the basics of wing flutter instability and provide insight into the optimization process for such structures. The code developed in this project could also be extended and modified by anyone tackling a similar project.

2 Theoretical Background

In this next chapter the basic theory which will be used during this project is presented. There are four main sections in this chapter. The first three sections are involved in solving for a lifting surface's flutter characteristics while the fourth section presents the various optimization techniques that will be used.

2.1 Composite Finite Elements

The theory behind the composite structural elements is developed according to E. Oñate, Structural Analysis with the Finite Element Method [1] which provides the basic equations and assumptions needed to develop 2D shell finite elements that take into account the effect of multilayer laminate composite materials.

2.1.1 Displacement Field

When studying composite laminated plate elements, the main problem that arises is that in contrast to homogeneous materials, points that belong on the middle plane of the element can be displaced "in plane" this results in axial forces that are not possible within a homogeneous material. To account for this fact, we introduce two axial displacements $u_0(x,y)$ and $v_0(x,y)$ and thus the displacement of any point within the plate can be calculated as follows:

$$\begin{aligned} u(x,y,z) &= u_{0(x,y)} - z \cdot \theta_x(x,y) \\ v(x,y,z) &= v_{0(x,y)} - z \cdot \theta_y(x,y) \\ w(x,y,z) &= w_0(x,y) \end{aligned} \tag{2.1}$$

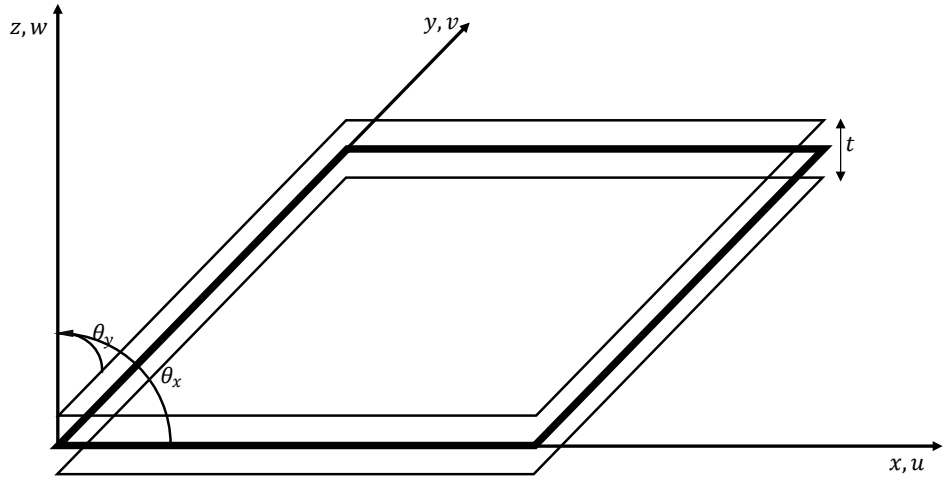


Figure 2-1Composite element coordinate system

2.1.2 Strain vectors

The strain within the plate can be calculated as follows:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \\ \frac{\partial u}{\partial z} + \frac{\partial w}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial u_0}{\partial x} \\ \frac{\partial v_0}{\partial y} \\ \frac{\partial u_0}{\partial y} + \frac{\partial v_0}{\partial x} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -z\frac{\partial \theta_x}{\partial x} \\ -z\frac{\partial \theta_y}{\partial y} \\ -z\left(\frac{\partial \theta_x}{\partial y} + \frac{\partial \theta_y}{\partial x}\right) \\ \frac{\partial w_0}{\partial x} - \theta_x \\ \frac{\partial w_0}{\partial y} - \theta_y \end{bmatrix} = \begin{bmatrix} \hat{\epsilon}_m \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} -z \cdot \hat{\boldsymbol{\epsilon}}_b \\ \hat{\boldsymbol{\epsilon}}_s \end{bmatrix} = [\boldsymbol{S}] \cdot \hat{\boldsymbol{\epsilon}} \quad (2.2)$$

Where:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \hat{\boldsymbol{\epsilon}}_m \\ \hat{\boldsymbol{\epsilon}}_b \\ \hat{\boldsymbol{\epsilon}}_s \end{bmatrix}, \quad \hat{\epsilon}_m = \begin{bmatrix} \frac{\partial u_0}{\partial x} \\ \frac{\partial v_0}{\partial y} \\ \frac{\partial u_0}{\partial y} + \frac{\partial v_0}{\partial x} \end{bmatrix}, \quad \hat{\epsilon}_b = \begin{bmatrix} \frac{\partial \theta_x}{\partial x} \\ \frac{\partial \theta_y}{\partial y} \\ \left(\frac{\partial \theta_x}{\partial y} + \frac{\partial \theta_y}{\partial x}\right) \end{bmatrix}, \qquad \hat{\boldsymbol{\epsilon}}_s = \begin{bmatrix} \frac{\partial w_0}{\partial x} - \theta_x \\ \frac{\partial w_0}{\partial y} - \theta_y \end{bmatrix}$$

Are the generalized Straub vectors due to membrane (m), bending (b) and transverse (s) shear deformation effects.

$$S = \begin{bmatrix} I_3 & -zI_3 & 0_2 \\ O_3^T & 0_2 & I_2 \end{bmatrix}$$

$$I_n \; is \; the \; n \times n \; identity \; matrix$$

$$O_2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \qquad 0_3 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

From the preceding relationship we can extract the following useful expressions:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \boldsymbol{\epsilon_p} \\ \boldsymbol{\epsilon_s} \end{bmatrix}$$

Where: $\boldsymbol{\epsilon_p} = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \end{bmatrix} = \hat{\boldsymbol{\epsilon}}_{\boldsymbol{m}} - z\hat{\boldsymbol{\epsilon}}_{\boldsymbol{b}}$ and $\boldsymbol{\epsilon_s} = \begin{bmatrix} \gamma_{xy} \\ \gamma_{yz} \end{bmatrix} = \hat{\boldsymbol{\epsilon}}_{\boldsymbol{s}}$

2.1.3 Stress – Strain relationship

The stress strain relationship in a composite laminated plate will now be derived.

We consider a composite laminated plate formed by piling n_l orthotropic layers called plies with orthotropy axes L,T,z and isotropy in the L axis (the Tz plane). The L axis is parallel to the direction of the longitudinal fibers of the composite material.

It will be assumed that:

- Each layer (k) is defined by the plane $z = z_k$ and $z = z_{k+1}$ with $z_k \leq z \leq z_{k+1}$
- The Orthotropy axes L and T can vary for each layer and are represented by angle β_i defined to be the angle between the global x axis and the L_i axis of the ith layer
- Each layer satisfies the plane stress assumption, namely $\sigma_z = 0$ and that the z axis is the orthotropy axis common for all layers
- The displacement field is continuous between the layers and satisfies 2.1

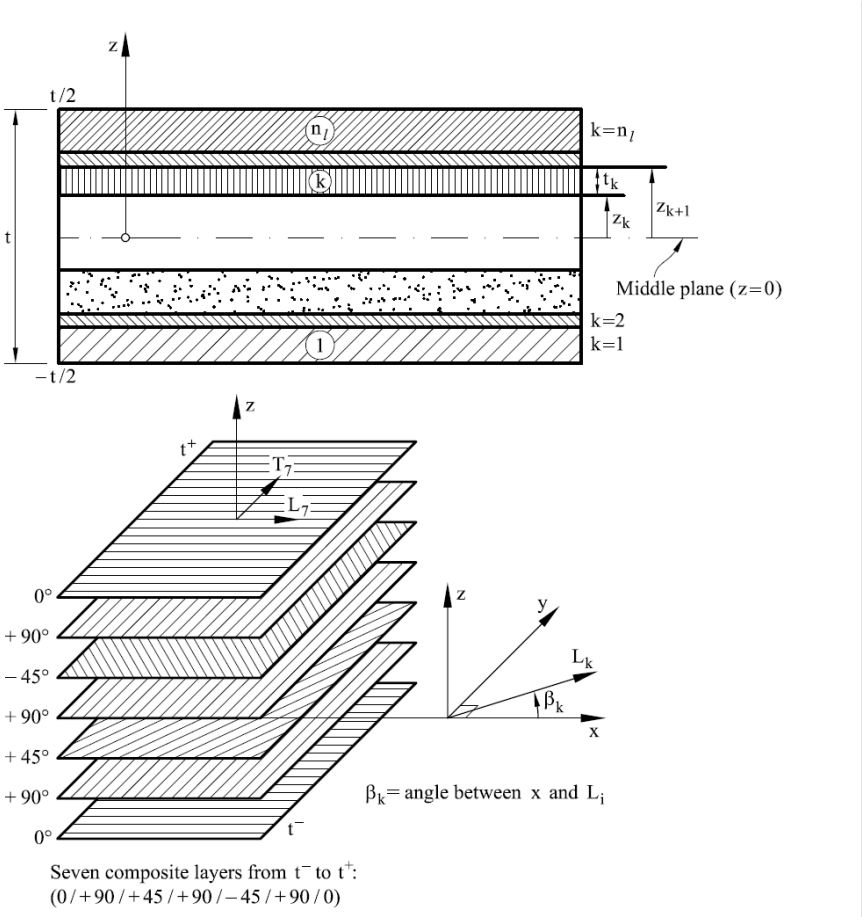


Figure 2-2 Definition of layers in a composite laminated plate [1]

The assumptions stated above allow us to express the relationship between the in-plane stresses $\sigma_x, \sigma_y, \tau_{xy}$ and the transverse shear strains τ_{xz}, τ_{yz} with their corresponding strains for each layer k as follows:

$$\boldsymbol{\sigma_p} = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \boldsymbol{D_p} \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{bmatrix} = \boldsymbol{D_p} \epsilon_p \tag{2.3}$$

$$\boldsymbol{\sigma_s} = \begin{bmatrix} \tau_{xz} \\ \tau_{yz} \end{bmatrix} = \boldsymbol{D_s} \begin{bmatrix} \gamma_{xz} \\ \gamma_{yz} \end{bmatrix} = \boldsymbol{D_s} \epsilon_s \tag{2.4}$$

And finally

$$\boldsymbol{\sigma} = \begin{bmatrix} \boldsymbol{\sigma_p} \\ \boldsymbol{\sigma_s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{D_p} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D_s} \end{bmatrix} \cdot \begin{bmatrix} \epsilon_p \\ \boldsymbol{\epsilon_s} \end{bmatrix} = D\epsilon \tag{2.5}$$

The constitutive matrices $D_p,\ D_s$ are symmetrical and their terms are a function of the five independent material properties as well as the angle β_k. The calculation of the aforementioned matrices begins by expressing the stress – strain relationships in the orthotropy axes L, T, z

$$\sigma_1 = \boldsymbol{D_1} \epsilon_1, \qquad \sigma_2 = \boldsymbol{D_2} \epsilon_2 \tag{2.6}$$

Where:

$$\sigma_1 = \begin{bmatrix} \sigma_L \\ \sigma_T \\ \tau_{LT} \end{bmatrix}, \qquad \epsilon_1 = \begin{bmatrix} \epsilon_L \\ \epsilon_T \\ \gamma_{LT} \end{bmatrix}, \qquad \boldsymbol{D_1} = \begin{bmatrix} D_{LL} & D_{LT} & 0 \\ & D_{TT} & 0 \\ Sym & & G_{LT} \end{bmatrix}$$

$$\sigma_2 = \begin{bmatrix} \tau_{Lz} \\ \tau_{Tz} \end{bmatrix}, \qquad \epsilon_2 = \begin{bmatrix} \gamma_{Lz} \\ \gamma_{Tz} \end{bmatrix}, \qquad \boldsymbol{D_2} = \begin{bmatrix} G_{Lz} & 0 \\ 0 & G_{Tz} \end{bmatrix}$$

$$D_{LL} = \frac{E_L}{a}, \qquad D_{TT} = \frac{E_T}{a}, \qquad a = 1 - \nu_{LT}\nu_{TL}, \qquad D_{LT} = \frac{E_T\,\nu_{LT}}{a}$$

The most commonly used five independent material parameters are:

$$E_L, \qquad E_T, \qquad \nu_{LT}\left(or\ \nu_{TL} = \frac{E_T}{E_L}\nu_{LT}\right), \qquad G_{Lz} = G_{LT}, \qquad G_{Tz}$$

Finally, the relationship between matrices $D_1,\ D_2$ and $D_p,\ D_s$ can be expressed as a simple matrix product with transformation matrices $T_1\ and\ T_2$ as follows

$$D_p = T_1^T \cdot D_1 \cdot T_1, \qquad D_s = T_2^T \cdot D_2 \cdot T_2$$

Where:

$$T_1 = \begin{bmatrix} \cos^2(\beta_k) & \sin^2(\beta_k) & cos(\beta_k)\sin(\beta_k) \\ \sin^2(\beta_k) & \cos^2(\beta_k) & -cos(\beta_k)\sin(\beta_k) \\ -2cos(\beta_k)\,sin(\beta_k) & 2cos(\beta_k)\,sin(\beta_k) & \cos^2(\beta_k) - \sin^2(\beta_k) \end{bmatrix}$$

$$T_2 = \begin{bmatrix} cos(\beta_k) & sin(\beta_k) \\ -sin(\beta_k) & cos(\beta_k) \end{bmatrix}$$

2.1.4 Generalized Constitutive Matrix

$$\hat{\boldsymbol{\sigma}}_m = \begin{bmatrix} N_x \\ N_y \\ N_{xy} \end{bmatrix} = \int_{-\frac{t}{2}}^{\frac{t}{2}} \boldsymbol{\sigma_p} dz\,, \qquad membrane\ forces$$

$$\hat{\boldsymbol{\sigma}}_b = \begin{bmatrix} M \\ M \\ M_{xy} \end{bmatrix} = -\int_{-\frac{t}{2}}^{\frac{t}{2}} z\boldsymbol{\sigma_p} dz\,, \qquad Bending\ moments$$

$$\hat{\boldsymbol{\sigma}}_s = \begin{bmatrix} Q \\ Q \end{bmatrix} = \int_{-\frac{t}{2}}^{\frac{t}{2}} z\boldsymbol{\sigma_s} dz\,, \qquad transverse\ shear\ forces$$

$$\hat{\boldsymbol{\sigma}}_m = \hat{\boldsymbol{D}}_m \hat{\epsilon}_m + \hat{\boldsymbol{D}}_{mb} \hat{\epsilon}_b$$
$$\hat{\boldsymbol{\sigma}}_b = \hat{\boldsymbol{D}}_{mb} \hat{\epsilon}_m + \hat{\boldsymbol{D}}_b \hat{\epsilon}_b$$
$$\hat{\boldsymbol{\sigma}}_s = \hat{\boldsymbol{D}}_s \hat{\epsilon}_s$$

Where:

$$\hat{\boldsymbol{D}}_m = \int_{-t/2}^{t/2} \boldsymbol{D}_p dz \tag{2.7}$$

$$\hat{\boldsymbol{D}}_{mb} = \int_{-t/2}^{t/2} z\boldsymbol{D}_p dz \tag{2.8}$$

$$\hat{\boldsymbol{D}}_b = \int_{-t/2}^{t/2} z^2\boldsymbol{D}_p dz \tag{2.9}$$

$$\hat{\boldsymbol{D}}_s = \begin{bmatrix} k_{11}\overline{D}_{s_{11}} & k_{12}\overline{D}_{s_{12}} \\ Sym & k_{22}\overline{D}_{s_{22}} \end{bmatrix}, \qquad with\ \overline{D}_{s_{ij}} = \int_{-\frac{t}{2}}^{\frac{t}{2}} D_{s_{ij}} dz \tag{2.10}$$

In equation 2.10 the k_{ij} terms are called the shear correction factors

The integrals in the equations above can be expressed as a finite sum when the composite laminate plate consists of n_l orthotropic layers. For composite plates where x and y are orthotropy axes for all the layers the $\hat{\boldsymbol{D}}_s$,matrix is diagonal and only k_{11} and k_{22} need to be computed.

The simplest method of computing the shear stress factors is by assuming cylindrical bending. This means that

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xz}}{\partial z} = 0 \tag{2.11}$$

$$\frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} = 0 \tag{2.12}$$

Further assuming a constant distribution of the transverse shear stress in the thickness direction and after some manipulation it is finally deduced

$$k_{11} = \hat{D}_{b_{11}}^2 \left[\bar{G}_{xz} \int_{-t/2}^{t/2} \frac{g_1^2(z)}{G_{xz}} dz \right]^{-1} \tag{2.13}$$

$$k_{22} = \hat{D}_{b_{22}}^2 \left[\bar{G}_{yz} \int_{-t/2}^{t/2} \frac{g_2^2(z)}{G_{yz}} dz \right]^{-1} \tag{2.14}$$

Where:

- $g_{1(z)} = \int_{-t/2}^{z} z D_{p_{11}} dz$
- $g_{2(z)} = \int_{-t/2}^{z} z D_{p_{22}} dz$
- $\bar{G}_{xz} = \int_{-t/2}^{t/2} G_{xz}\, dz$
- $\bar{G}_{yz} = \int_{-t/2}^{t/2} G_{yz}\, dz$

Equations 2.7 through 2.10 can be expressed as a finite sum when the laminate composite is made of n_l orthotropic layers within each of which the material properties are constant

$$\hat{\boldsymbol{D}}_m = \sum_{k=1}^{n_l} t_k \boldsymbol{D}_{pk} \tag{2.15}$$

$$\hat{\boldsymbol{D}}_{mb} = -\sum_{k=1}^{n_l} t_k \overline{z_k} \boldsymbol{D}_{pk} \tag{2.16}$$

$$\hat{\boldsymbol{D}}_b = \sum_{k=1}^{n_l} \frac{1}{3} (\bar{z}_{k+1}^{\boldsymbol{3}} - z_k^{\boldsymbol{3}}) \boldsymbol{D}_{pk} \tag{2.17}$$

$$\hat{\boldsymbol{D}}_{\boldsymbol{s}} = \begin{bmatrix} \tilde{k}_{11} \sum_{k=1}^{n_l} t_k \boldsymbol{D}_{sk_{11}} & 0 \\ 0 & \tilde{k}_{22} \sum_{k=1}^{n_l} t_k \boldsymbol{D}_{sk_{22}} \end{bmatrix} \tag{2.18}$$

Where:

$$t_k = z_{k+1} - z_k, \qquad \bar{z}_k = \frac{1}{2}(z_{k+1} + z_k)$$

$$\tilde{k}_{ii} = \widehat{\boldsymbol{D}}_{b_{ii}} \cdot \left[\sum_{k=1}^{n_l} t_k G_{xz} \cdot \sum_{k=1}^{n_l} \left(\frac{\sum_{l=1}^{k} \left(t_k \bar{z}_k \boldsymbol{D}_{p_{ii_l}} \right)}{G_{xz_k}} t_k \right) \right], \qquad i = 1, 2$$

2.1.5 Discretized stress and strain - Shape functions

The displacements within a four-node quadrilateral composite element can be expressed through shape functions.

$$\boldsymbol{u} = \begin{bmatrix} u \\ v \\ w \\ \theta_x \\ \theta_y \end{bmatrix} = \sum_{i=1}^{4} N_i a_i = \boldsymbol{N} \cdot \vec{\boldsymbol{a}} \tag{2.19}$$

Where:

$$\boldsymbol{N} = \begin{bmatrix} N_1 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 & 0 & 0 & N_3 & 0 & 0 & 0 & 0 & N_4 & 0 & 0 & 0 & 0 \\ 0 & N_1 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 & 0 & 0 & N_3 & 0 & 0 & 0 & 0 & N_4 & 0 & 0 & 0 \\ 0 & 0 & N_1 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 & 0 & 0 & N_3 & 0 & 0 & 0 & 0 & N_4 & 0 & 0 \\ 0 & 0 & 0 & N_1 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 & 0 & 0 & N_3 & 0 & 0 & 0 & 0 & N_4 & 0 \\ 0 & 0 & 0 & 0 & N_1 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 & 0 & 0 & N_3 & 0 & 0 & 0 & 0 & N_4 \end{bmatrix}$$

$$\vec{\boldsymbol{a}} = \begin{bmatrix} \vec{\boldsymbol{a}}_1 \\ \vec{\boldsymbol{a}}_2 \\ \vec{\boldsymbol{a}}_3 \\ \vec{\boldsymbol{a}}_4 \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ w_1 \\ \theta_{x_1} \\ \theta_{y_1} \\ u_2 \\ v_2 \\ w_2 \\ \theta_{x_2} \\ \theta_{y_2} \\ u_3 \\ v_3 \\ w_3 \\ \theta_{x_3} \\ \theta_{y_3} \\ u_4 \\ v_4 \\ w_4 \\ \theta_{x_4} \\ \theta_{y_4} \end{bmatrix}$$

The stress strain relationship using equation 2.2 can now be expressed using the shape functions:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \hat{\boldsymbol{\epsilon}}_m \\ \hat{\boldsymbol{\epsilon}}_b \\ \hat{\boldsymbol{\epsilon}}_s \end{bmatrix}, \quad \hat{\epsilon}_m = \begin{bmatrix} \frac{\partial u_0}{\partial x} \\ \frac{\partial v_0}{\partial y} \\ \frac{\partial u_0}{\partial y} + \frac{\partial v_0}{\partial x} \\ \frac{\partial \theta_x}{\partial x} \\ \frac{\partial \theta_y}{\partial y} \\ \frac{\partial \theta_x}{\partial y} + \frac{\partial \theta_y}{\partial x} \\ \frac{\partial w_0}{\partial x} - \theta_x \\ \frac{\partial w_0}{\partial y} - \theta_y \end{bmatrix} = \sum_{i=4}^{4} \begin{bmatrix} \frac{\partial N_i}{\partial x} u_i \\ \frac{\partial N_i}{\partial x} v_i \\ \frac{\partial N_i}{\partial y} u_i + \frac{\partial N_i}{\partial x} v_i \\ \frac{\partial N_i}{\partial x} \theta_{x_i} \\ \frac{\partial N_i}{\partial y} \theta_{y_i} \\ \frac{\partial N_i}{\partial y} \theta_{x_i} + \frac{\partial N_i}{\partial x} \theta_{y_i} \\ \frac{\partial N_i}{\partial x} w_i - N_i \theta_{x_i} \\ \frac{\partial N_i}{\partial y} w_i - N_i \theta_{y_i} \end{bmatrix} = [\boldsymbol{B_1}, \boldsymbol{B_2}, \boldsymbol{B_3}, \boldsymbol{B_4}] \cdot \vec{\boldsymbol{a}} = \boldsymbol{B} \cdot \vec{\boldsymbol{a}} \quad (2.20)$$

Where:

$$\boldsymbol{B_i} = \begin{bmatrix} \boldsymbol{B_{m_i}} \\ \boldsymbol{B_{b_i}} \\ \boldsymbol{B_{s_i}} \end{bmatrix}$$

$$\boldsymbol{B_{m_i}} = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial N_i}{\partial y} & 0 & 0 & 0 \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{B_{b_i}} = \begin{bmatrix} 0 & 0 & 0 & -\frac{\partial N_i}{\partial x} & 0 \\ 0 & 0 & 0 & 0 & -\frac{\partial N_i}{\partial y} \\ 0 & 0 & 0 & -\frac{\partial N_i}{\partial y} & -\frac{\partial N_i}{\partial x} \end{bmatrix},$$

$$\boldsymbol{B_{s_i}} = \begin{bmatrix} 0 & 0 & \frac{\partial N_i}{\partial x} & -N_i & 0 \\ 0 & 0 & \frac{\partial N_i}{\partial y} & 0 & -N_i \end{bmatrix}$$

The shape functions used can vary but the simplest is the bilinear quadrilateral element which uses linear shape functions for interpolation. All calculations described in the preceding chapter are carried out in a transformed coordinate space where every element is a perfect square of side length two length units. This transformed space is called the "natural" space. The shape functions are:

$$N_i(\xi,\eta) = \frac{1}{4}(1 + a_i \cdot \xi)(1 + b_i \cdot \eta) \tag{2.21}$$

$$\frac{\partial N_i(\xi,\eta)}{\partial \xi} = \frac{1}{4} a_i (1 + b_i \eta) \tag{2.22}$$

$$\frac{\partial N_i(\xi,\eta)}{\partial \eta} = \frac{1}{4} b_i (1 + a_i \xi) \tag{2.23}$$

Table 1 shape function coefficients

i	1	2	3	4
a_i	-1	1	1	-1
b_i	-1	-1	1	1

The Jacobian of this transformation from physical to natural space is defined as:

$$\mathcal{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \tag{2.24}$$

The derivatives of the shape functions in the physical space can then be calculated as:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} = \mathcal{J}^{-1} \cdot \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} \tag{2.25}$$

Finally, another property of the Jacobian is that its determinant represents the scale of the transformation.

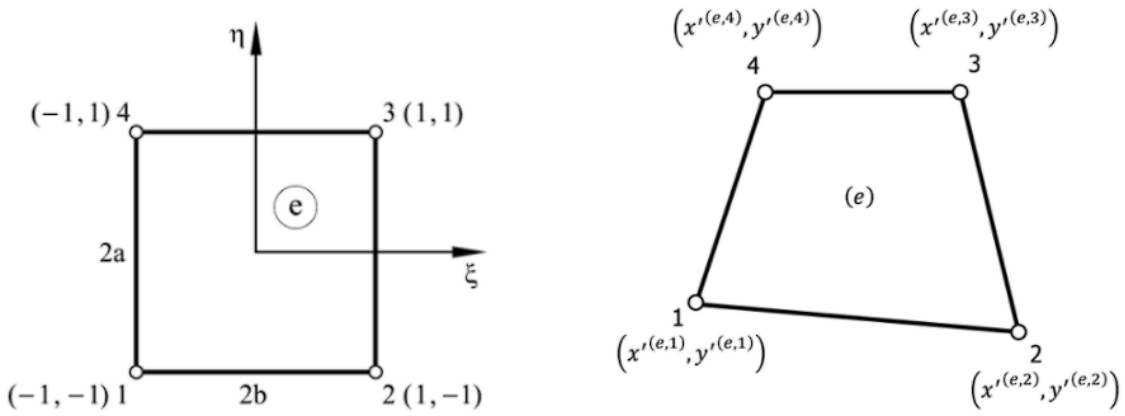


Figure 2-3 natural and physical coordinate space of quadrilateral plate element [2]

2.1.6 Stiffness matrix

The final step in the computation of the finite elements is the calculation and assembly of the stiffness matrix. Using the Principle of Virtual Work in the standard manner the local stiffness matrices can be written:

$$\boldsymbol{K}_{m_{ij}[20\times20]} = \iint_{Area} \boldsymbol{B}_{m,i}^{T} \widehat{\boldsymbol{D}}_{m} \boldsymbol{B}_{m,j}\, dA\,, \qquad membrane\ stiffness \tag{2.26}$$

$$\boldsymbol{K}_{b_{ij}[20\times20]} = \iint_{Area} \boldsymbol{B}_{b,i}^{T} \widehat{\boldsymbol{D}}_{b} \boldsymbol{B}_{b,j}\, dA\,, \qquad bending\ stiffness \tag{2.27}$$

$$\boldsymbol{K}_{s_{ij}[20\times20]} = \iint_{Area} \boldsymbol{B}_{s,i}^{T} \widehat{\boldsymbol{D}}_{s} \boldsymbol{B}_{s,j}\, dA\,, \qquad shear\ stiffness \tag{2.28}$$

$$\boldsymbol{K}_{mb_{ij}[20\times20]} = \iint_{Area} \left[\boldsymbol{B}_{m,i}^{T} \widehat{\boldsymbol{D}}_{mb} \boldsymbol{B}_{b,j} + \boldsymbol{B}_{b,i}^{T} \widehat{\boldsymbol{D}}_{mb} \boldsymbol{B}_{m,j}\right] dA\,, \qquad membrane - bending\ stiffness \tag{2.29}$$

All the integrals in equations 2.26 through 2.29 are computed using the Gauss quadrature. The full Gauss quadrature for the 4-node plate element developed involves four integration points while the reduced integration form of these elements require only one Gauss point.

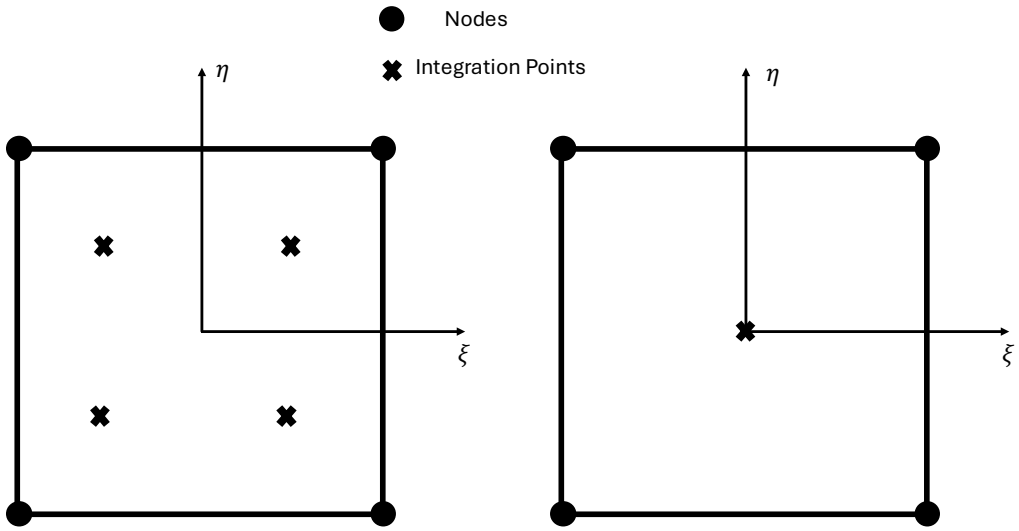


Figure 2-4Gauss points for full and reduced Integration in 4 node elements

The Gaussian integration for two dimensional domains using n Gauss points states that:

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)dxdy \approx \sum_{j=1}^{n}\sum_{i=1}^{n} w_i w_j f(x_i, y_i), for\ every\ i,j \leq n \qquad (2.30)$$

Since the plate element has four nodes only two integrations are possible:

- Using one Gauss point resulting in the reduced Integration scheme
- Using four Gauss points resulting in the full integration scheme.

Using full integration results in greater computational time but improves accuracy, on the other hand reduced integration make computation faster requiring only a fourth as many computations, but results in less accurate results and introduces the so called zero energy modes which are deformed states of the element which have zero strain energy. This is physically impossible and reduces the stiffness of the structure as to achieve these modes of deformation no energy is needed. This phenomenon is also known as the hourglass effect in FEM.

The table that follows exposes the Gauss point coordinates and the weights that shall be used when performing one or four Gauss point integration

Table 2 Gauss points weights and coordinates for one and four gauss point integration

N_G	k	w_k	ξ_k	η_k
1	1	2	0	0
4	1	1	$-1/\sqrt{3}$	$-1/\sqrt{3}$
	2	1	$+1/\sqrt{3}$	$-1/\sqrt{3}$
	3	1	$+1/\sqrt{3}$	$+1/\sqrt{3}$
	4	1	$-1/\sqrt{3}$	$+1/\sqrt{3}$

Using the Gauss quadrature the local stiffness matrix of the plate element can be calculated as the sum of all the component stiffness matrices:

$$\boldsymbol{K}_{local} = \boldsymbol{K}_m + \boldsymbol{K}_b + \boldsymbol{K}_s + \boldsymbol{K}_{mb} \tag{2.31}$$

The final step to obtain the global stiffness matrix is to transform the local coordinate system back to the global coordinate system. This is done using the rotation matrix and the determinant of the Jacobian as the scale as follows:

$$\boldsymbol{K}_{global} = \det(\mathcal{J}) \cdot \boldsymbol{R}^{\boldsymbol{T}} \cdot \boldsymbol{K}_{local} \cdot \boldsymbol{R} \tag{2.32}$$

Where:

$$\underbrace{\boldsymbol{R}}_{5n\times 6n} = \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \ddots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{L}_n \end{bmatrix}$$

$$\boldsymbol{L_i} = \begin{bmatrix} \lambda_{x'x} & \lambda_{x'y} & \lambda_{x'z} & 0 & 0 & 0 \\ \lambda_{y'x} & \lambda_{y'y} & \lambda_{y'z} & 0 & 0 & 0 \\ \lambda_{z'x} & \lambda_{z'y} & \lambda_{z'z} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\lambda_{y'x} & -\lambda_{y'y} & -\lambda_{y'z} \\ 0 & 0 & 0 & \lambda_{x'x} & \lambda_{x'y} & \lambda_{x'z} \end{bmatrix}$$

With $n = 4$ and $\lambda_{x'x}$ being the cosine of the angle formed by axes x' and x etc.

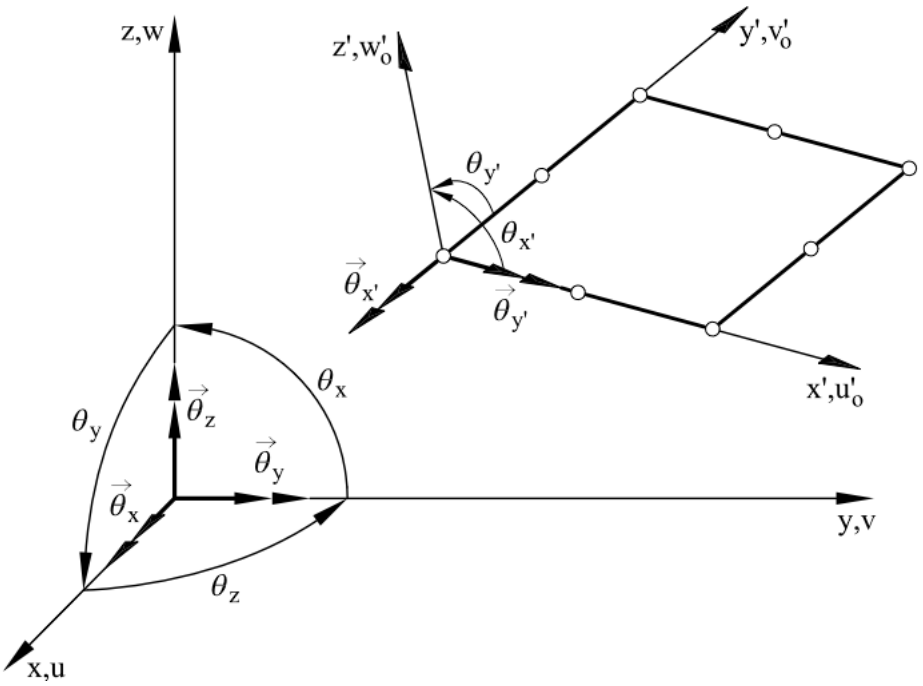


Figure 2-5 local and global axes definition [1]

The plate elements developed here only have 2 degrees of freedom per node. In order to obtain the full six degrees of freedom per node a stiffness component in the rotation about the z axis ϑ_z needs to be added. This is a fictitious stiffness term that can be added to bring the stiffness matrix to its full size of 24×24. This technique is often used to avoid potential singularities in the stiffness matrix, and it is more intuitive for every node to have six degrees of freedom.

A common technique used to avoid shear locking phenomenon is to integrate the component stiffness matrices using different Gauss quadrature.

2.2 Aerodynamic Theory – Vortex Lattice Method (VLM)

The vortex lattice method theory developed in this chapter follows the conventions of the book A. P. Joseph Katz, Low-Speed Aerodynamics [3]

2.2.1 The Vortex Filament – Biot Savart Law

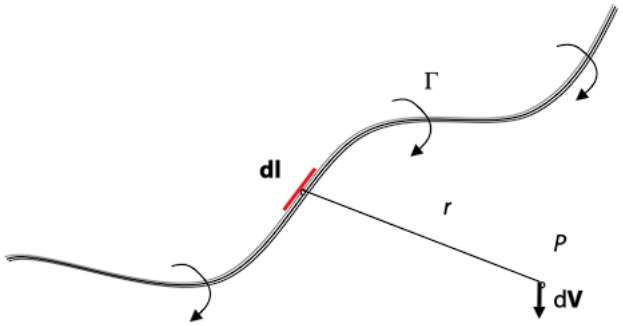


Figure 2-6 Curved Three-dimensional vortex filament of strength Γ [4]

The continuity equation for an incompressible fluid is:

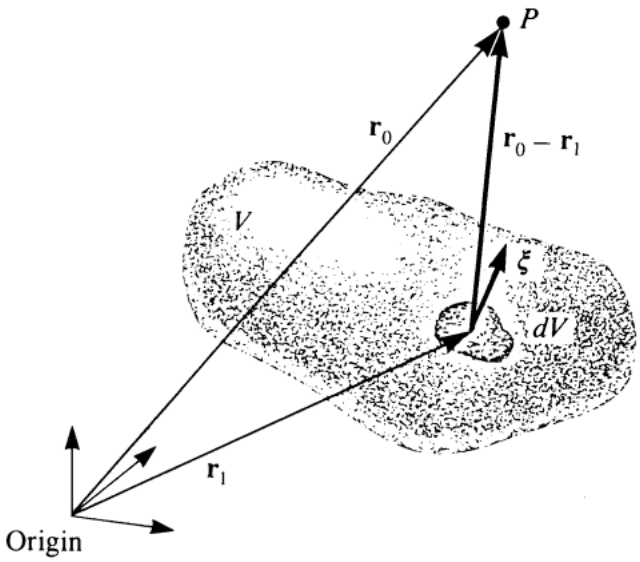
$$\nabla \cdot \vec{V} = 0 \tag{2.33}$$

The vector potential of the velocity field is a vector field $\vec{B}$ which is defined by:

$$\nabla \times \vec{B} = \vec{V}$$

The divergence of the vector potential $\vec{B}$ is zero and thus the vorticity can be expressed as:

$$\zeta = \nabla \times \vec{V} = \nabla \times \left(\nabla \times \vec{B}\right) = \cancel{\nabla\left(\nabla \cdot \vec{B}\right)} - \nabla^2 \vec{B} = -\nabla^2 \vec{B} \tag{2.34}$$



The general solution to this equation using greens theorem is:

$$\vec{B} = \frac{1}{4\pi}\int_V \frac{\vec{\zeta}}{|\vec{r}|}dV \tag{2.35}$$

$$\vec{V} = \frac{1}{4\pi}\int_V \nabla \times \frac{\vec{\zeta}}{|\vec{r}|}dV \tag{2.36}$$

Where: $\vec{r} = \vec{r}_0 - \vec{r}_1$

Considering an infinitesimal piece of vorticity filament ζ so that $d\boldsymbol{l} = \frac{\zeta}{\zeta}dl$, $\Gamma = \zeta \cdot dS$ and $dV = dS \cdot dl$

$$\nabla \times \frac{\zeta}{|\vec{r}|}d\vec{V} = \nabla \times \Gamma\frac{d\boldsymbol{l}}{|\vec{r}|} = \Gamma\frac{d\boldsymbol{l} \times \vec{r}}{|\vec{r}|^3} \tag{2.37}$$

Substitution of equation 2.37 into equation 2.36 leads to

$$\vec{V} = \frac{1}{4\pi}\int_V \frac{d\boldsymbol{l} \times \vec{r}}{|\vec{r}|^3}dV \tag{2.38}$$

2.2.2 Straight Vortex Segment

The vortex segment is placed at an arbitrary orientation with constant circulation and finite length, as shown in the figure below. As is shown, the induced velocity has only a tangential component q_θ.

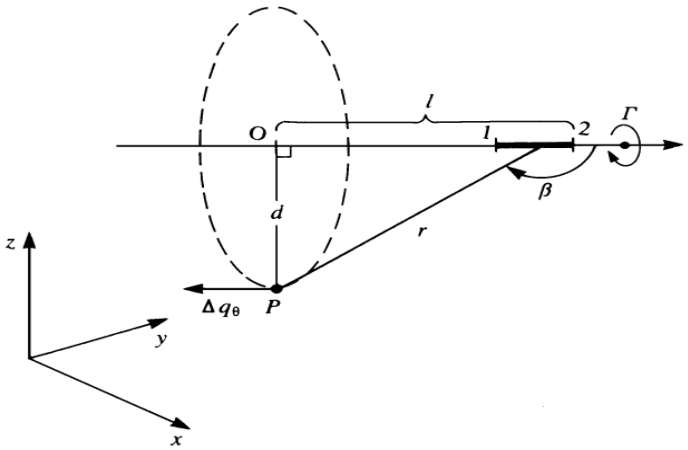


Figure 2-7 Induced Velocity from straigh Vortex Segment [3]

From the analysis of the infinitesimal vortex filament, it has been shown that the induced velocity is:

$$\Delta\boldsymbol{V} = \frac{\Gamma}{4\pi}\frac{d\boldsymbol{l} \times \vec{r}}{r^3} = \frac{\Gamma}{4\pi}\frac{\sin(\beta)}{r^2}dl\,\widehat{e_\theta} \tag{2.39}$$

Note that:

$$d = rsin(\beta)\ and \tan(pi - \beta) = \frac{d}{l}$$

Therefore:

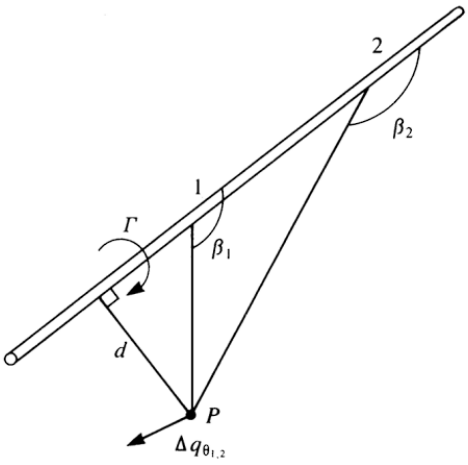
$$l = -\frac{d}{\tan(\beta)}\ and\ dl = \frac{d}{\sin^2(\beta)} d\beta$$

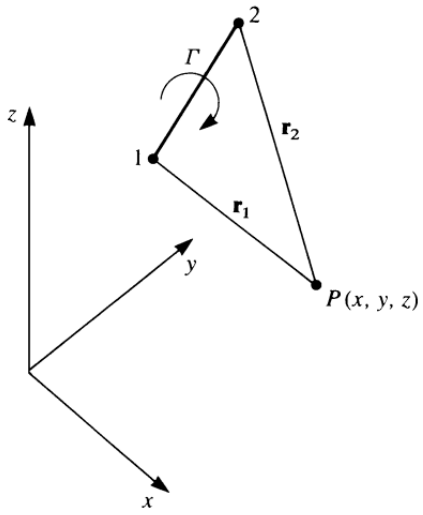
Substituting these terms in equation 2.39 we get:

$$\Delta V_\theta = \frac{\Gamma}{4\pi d} \sin(\beta)\, d\beta \tag{2.40}$$

This equation can be integrated over the straight cortex segment resulting in:

$$V_{\theta_{1\to2}} = \frac{\Gamma}{4\pi d} \int_{\beta_1}^{\beta_2} \sin(\beta)\, d\beta = \frac{\Gamma}{4\pi d} \left(cos(\beta_1) - cos(\beta_2)\right) \tag{2.41}$$





Finally noting that

- $d = \frac{|\vec{r}_1 \times \vec{r}_2|}{|\vec{r}_0|}$
- $cos\beta_1 = \frac{(\vec{r}_0 \cdot \vec{r}_1)}{|\vec{r}_0||\vec{r}_1|}$
- $cos\beta_2 = \frac{(\vec{r}_0 \cdot \vec{r}_2)}{|\vec{r}_0||\vec{r}_2|}$
- $\hat{e}_\theta = \frac{\vec{r}_1 \times \vec{r}_2}{|\vec{r}_1 \times \vec{r}_2|}$

The induced velocity becomes:

$$\vec{V}_{1\to2} = \frac{\Gamma}{4\pi} \frac{\vec{r}_1 \times \vec{r}_2}{|\vec{r}_1 \times \vec{r}_2|} (\vec{r}_1 - \vec{r}_2) \cdot \left(\frac{\vec{r}_1}{|\vec{r}_1|} - \frac{\vec{r}_2}{|\vec{r}_2|}\right) \tag{2.42}$$

2.2.3 Lifting Surface Computational Solution by Vortex Ring Elements

The goal of this section is the calculation of the influence coefficient matrix which is necessary to predict the lift of an oscillating wing surface.

The main boundary condition that must be satisfied is the zero normal flow across the wing's solid surface which can be expressed using the velocity potential $\nabla\Phi = \vec{V}$ as:

$$\nabla(\Phi + \Phi_\infty) \cdot \hat{n} = 0 \tag{2.43}$$

The numerical method begins by defining the type of singularity element that will be used. In the Vortex Lattice method, a Vortex ring is used. The vortex ring is quadrilateral element which has a straight vortex line at every edge. The leading vortex is placed at the quarter chord of the panel while the collocation point is at the center of the three-quarter chord line. The normal vector of the panel is calculated on the collocation point. A positive circulation Γ is defined according to the right-hand rule. By placing the leading-edge vortex at the quarter-chord line the 2-dimension Kutta condition is satisfied along the chord.

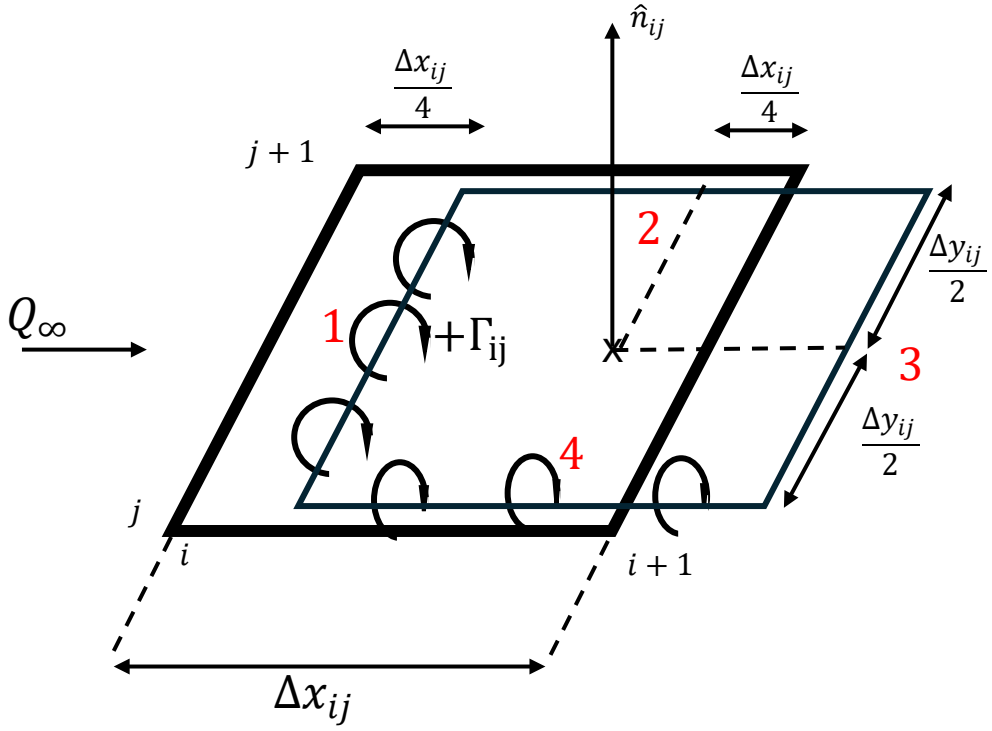


Figure 2-8 Vortex Ring Element

The numbers 1 through 4 on Figure 2-8 Vortex Ring Element represent the four finite straight vortex segments that make up the element. The induced velocity of the element can be calculated using equation 2.42 on each segment separately. For an arbitrary point in space $P(x, y, z)$ the induced velocity is:

$$\vec{V}_P = \vec{V}_1 + \vec{V}_2 + \vec{V}_3 + \vec{V}_4 \tag{2.44}$$

Where:

$$\vec{V}_i = \frac{\Gamma}{4\pi} \frac{\vec{r}_{i,1} \times \vec{r}_{i,2}}{|\vec{r}_{i,1} \times \vec{r}_{i,2}|} (\vec{r}_{i,1} - \vec{r}_{i,2}) \cdot \left(\frac{\vec{r}_{i,1}}{|\vec{r}_{i,1}|} - \frac{\vec{r}_{i,2}}{|\vec{r}_{i,2}|} \right) \tag{2.45}$$

These elements are sorted in a two-dimensional grid to cover the lifting surface shape. To satisfy the trailing edge condition (Kutta condition) the trailing vortex of the last panel row must be cancelled. Therefor the last row of panels as well as all the wake panels have the same circulation Γ.

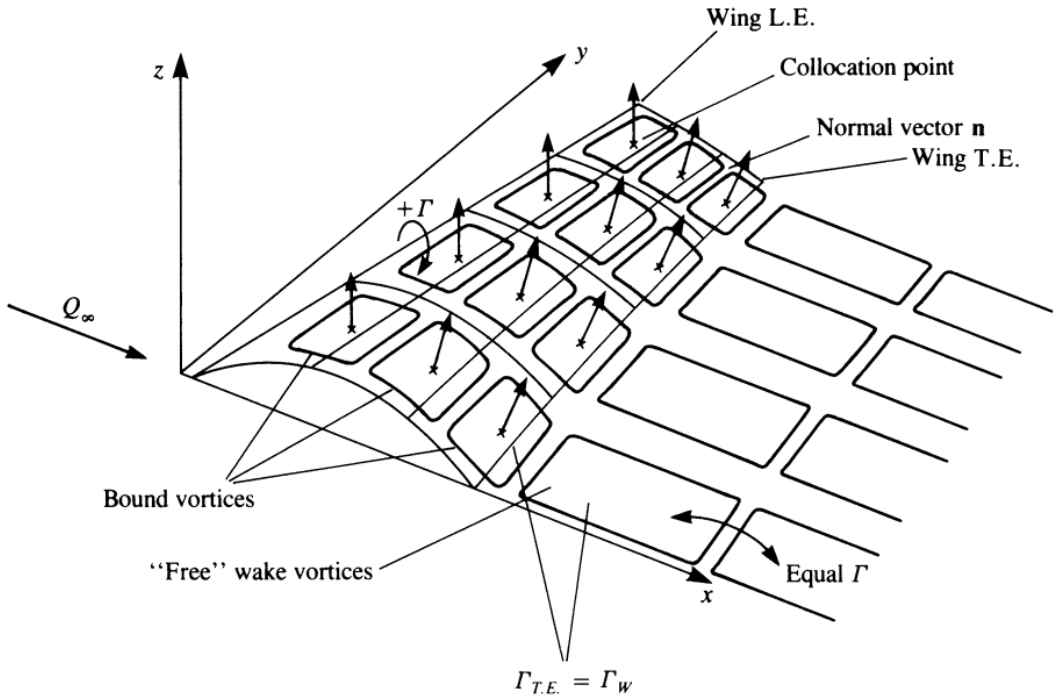


Figure 2-9 vortex ring elements in a grid [3]

The influence coefficient α_{ij} is essentially the induced velocity of the i-th vortex ring element with unitary circulation at the j-th collocation point. The influence coefficients can easily be calculated using equation 2.45 . Iterating over all panels and all collocation points results in a matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{bmatrix} \tag{2.46}$$

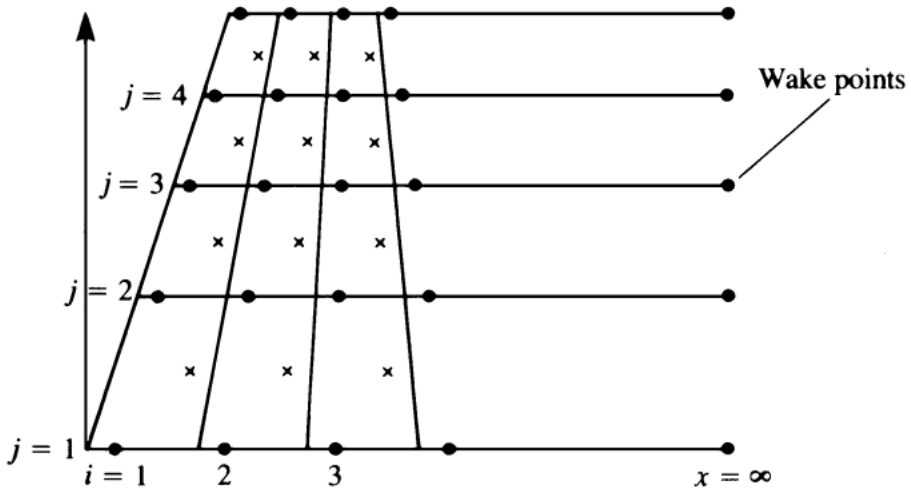


Figure 2-10 Array of wing and wake panel corner points (dots) and of collocation points(× symbols) [3]

For the sake of completeness, the linear set of equations that has to be solved in order to calculate the circulation intensity of each element given a certain angle of attack for each panel is:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{bmatrix} \begin{bmatrix} \Gamma_1 \\ \Gamma_2 \\ \vdots \\ \Gamma_m \end{bmatrix} = \begin{bmatrix} -\vec{Q}_\infty \cdot \hat{n}_1 \\ -\vec{Q}_\infty \cdot \hat{n}_2 \\ \vdots \\ -\vec{Q}_\infty \cdot \hat{n}_m \end{bmatrix} \tag{2.47}$$

The downwash induced at each vortex ring element can be calculated by another set of linear equations

$$\begin{bmatrix} \mathrm{w}_{\mathrm{ind}_1} \\ \mathrm{w}_{\mathrm{ind}_2} \\ \vdots \\ \mathrm{w}_{\mathrm{ind}_m} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mm} \end{bmatrix} \begin{bmatrix} \Gamma_1 \\ \Gamma_2 \\ \vdots \\ \Gamma_m \end{bmatrix} \tag{2.48}$$

The lift and drag can then be calculated by using the relationships:

$$L = \sum_{i=1}^{M} \sum_{\mathrm{j}=1}^{\mathrm{N}} \Delta L_{ij}\,, \qquad where\ \Delta L_{ij} = \begin{cases} \rho Q_\infty (\Gamma_{i,j} - \Gamma_{i-1,j}) \Delta y_{ij}, & i > 1 \\ \rho Q_\infty \Gamma_{ij} \Delta y_{ij}, & i = 1 \end{cases} \tag{2.49}$$

$$D = \sum_{i=1}^{M} \sum_{\mathrm{j}=1}^{\mathrm{N}} \Delta D_{ij}\,, \qquad where\ \Delta D_{ij} = \begin{cases} \rho w_{ind_{ij}} (\Gamma_{i,j} - \Gamma_{i-1,j}) \Delta y_{ij}, & i > 1 \\ \rho w_{ind_{ij}} \Gamma_{ij} \Delta y_{ij}, & i = 1 \end{cases} \tag{2.50}$$

2.3 Flutter Analysis Equations

The general form of the aeroelastic equation of motion is

$$[A]\ddot{q} + (\rho V[B] + D)\dot{q} + (\rho V^2[C] + [E])q = [0]$$

Where:

- A, C, and E, are the structural matrices corresponding to the M, C and K matrices which is the classical notation used in structural analysis
- B and C are the aerodynamic matrices which depend on the Mach number and the reduced frequency. The way these matrices are computed can vary depending on the aerodynamic theory used.
- q are the generalized modal coordinates.

2.3.1 Interconnection of the Structure with Aerodynamics – Infinite Plate splines

The structural and aerodynamic degrees of freedom are connected by interpolation. This allows the independent selection of grid points (nodes) of the structural and aerodynamic matrices thus decoupling the two discretization of geometry. This interpolation method I s called splining. The Infinite plate spline which is used here is based on the structural deformation of a theoretically infinite plate subject to point loads at given points. The splines are used for two distinct purposed: as a force interpolator to compute the structurally equivalent force distribution on the structure given a force distribution on the aerodynamic mesh and as a displacement interpolator to compute a set of aerodynamic displacements given as a set of structural displacements. Mathematically this means that:

$$F_s = [G_{s\to a}]F_a, \qquad U_a = [G_{a\to s}]U_s \tag{2.51}$$

Where:

- F_s and F_a are the structural and aerodynamic force vectors
- U_a and U_s are the aerodynamic and structural displacement vectors
- $G_{s\to a}$ and $G_{a\to s}$ are the spline matrices connecting the structural to aerodynamic and vice versa. It has been proven by the virtual work principle that the two spline

matrices are the transform of one another i.e. $[G_{s\to a}] = [G_{a\to s}]^T$

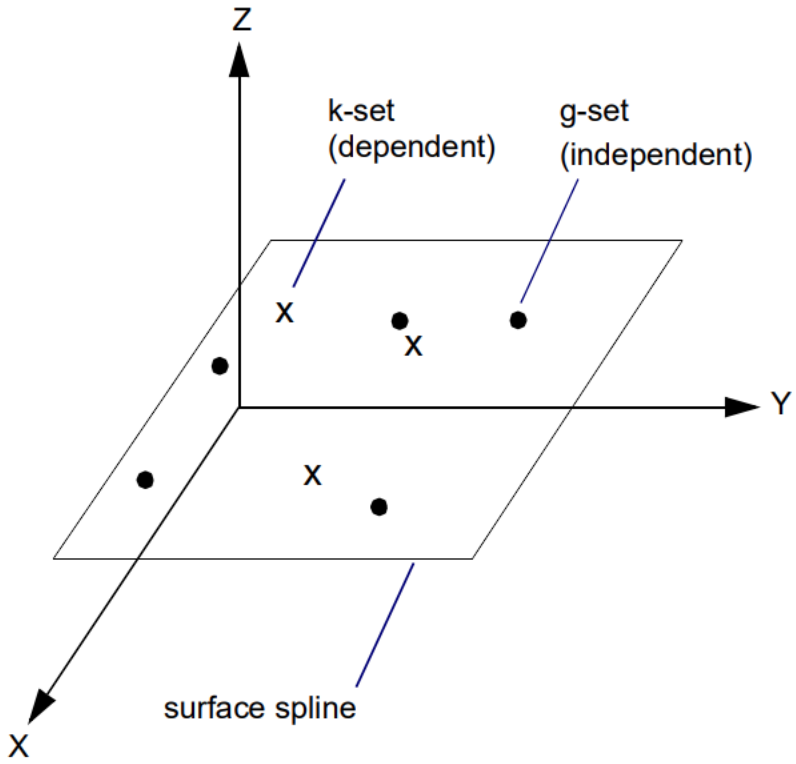


Figure 2-11 Surface Spline coordinate system [5]

The Infinite plate spline and all other surface splines are used to find a surface in the form $w(x, y)$ for all points (x, y) when w is known at a discrete set of points $w_i = w(x_i, y_i)$.

The Infinite plate spline mimics the behavior of an infinite plate under point load. The deflection of such a plate due to a single concentrated load at $(x_i = 0, y_i = 0)$ is called the fundamental solution. The governing equation is:

$$D\nabla^4(w) = D\frac{1}{r}\frac{d}{dr}\left\{r\frac{d}{dr}\left[\frac{1}{r}\frac{d}{dr}\left(r\frac{dw}{dr}\right)\right]\right\} = q \qquad (2.52)$$

The general solution to the homogeneous form of is:

$$w = C_1 + C_1 r^2 + C_2 \ln(r) + C_3 r^2 ln(r) (2.53)$$

Setting $C_2 = 0$ to keep solution finite at $r = 0$ then integrating from $r = 0$ to $r = \epsilon\ (small\ number)$ leads to $C_3 = \frac{P}{8\pi D}$. Thus, the fundamental solution for a concentrated load is:

$$w = A + Br^2 + \frac{P}{16\pi D} r^2 \ln(r^2) \qquad (2.54)$$

The fundamental solution can be superimposed to solve the entire plate problem:

$$w(x,y) = \alpha_0 + \alpha_1 x + \alpha_2 y + \sum_{i=1}^{N} K_i(x,y) P_i \qquad (2.55)$$

Where:

- $K_i(x,y) = \frac{1}{16\pi D} r_i^2 \ln(r_i^2)$
- $r_i^2 = (x - x_i)^2 + (y - y_i)^2$
- P_i concentrated load at (x_i, y_i)

The N+3 unknowns of are determined from the N+3 equations

$$\sum Pi = 0$$

$$\sum x_i P_i = 0$$

$$\sum y_i P_i = 0$$

$$w_j = a_0 + a_1 x_j + a_2 y_2 + \sum_{i=1}^{N} K_{ij} P_i, \qquad (j = 1, \dots, N)$$

Where $K_{ij} = K_i(x_j, y_j)$ and $K_{ij} = K_{ji},\ K_{ij} = 0\ when\ i = j$

These N+3 equations can be expressed into a matrix form:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & \cdots & 1 \\ 0 & 0 & 0 & x_1 & \cdots & x_N \\ 0 & 0 & 0 & y_1 & \cdots & y_N \\ \\ 1 & x_1 & y_1 & 0 & \cdots & K_{1N} \\ 1 & x_2 & y_2 & K_{21} & \cdots & K_{2N} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_N & y_N & K_{N1} & \cdots & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \\ P_1 \\ P_2 \\ \vdots \\ P_N \end{bmatrix} = [C][P] \qquad (2.56)$$

Now that the a_i and P_i any quantity can be interpolated at any point in the (x, y) plane using the following matrix equation:

$$\{w\} = \begin{bmatrix} 1 & x_1 & y_1 & K_{1,1} & K_{1,2} & \cdots & K_{1,n} \\ 1 & x_2 & y_2 & K_{2,1} & K_{2,2} & \cdots & K_{2.m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & y_n & K_{n,1} & K_{n,2} & \cdots & K_{n,n} \end{bmatrix} \cdot [C]^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \tag{2.57}$$

2.3.2 The PK Method of Flutter Solution

The fundamental equation for modal flutter analysis by the PK method is:

$$\left[M_{hh}p^2 + \left(B_{hh} - \frac{1}{4}\rho cV\frac{Q_{hh}^I}{k} \right) p + \left(K_{hh} - \frac{1}{2}\rho V^2 Q_{hh}^R \right) \right] \cdot \{u_h\} = \underline{0} \tag{2.58}$$

Where:

- M_{hh} = modal mass matrix, usually diagonal
- B_{hh} = modal damping matrix
- K_{hh} = modal stiffness matrix, usually diagonal, may be complex if actual damping is used
- $Q_{hh}(M,k)$ = aerodynamic force matrix which is a function of mach number and reduced frequency $k = \omega\bar{c}/2V$
- Q_{hh}^R, Q_{hh}^I are the real and imaginary parts of the aerodynamic force matrix Q_{hh} also called aerodynamic stiffness and aerodynamic damping matrices respectively
- $k = \omega\bar{c}/2V$ reduced frequency
- M = Mach number
- V = Velocity
- ρ = fluid density
- $\{u_h\}$ = modal amplitude vector (modal participation vector
- $p = \omega(\gamma \pm i)$ eigenvalue

The equation is rewritten in the state-space form with order reduction

$$[A - pI]\{\bar{u}_h\} = 0 \tag{2.59}$$

Where $[A]$ is the real matrix:

$$[A] = -\begin{bmatrix} [\mathbf{0}] & [\boldsymbol{I}] \\ -M_{hh}^{-1}\left(K_{hh} - \frac{1}{2}\rho V^2 Q_{hh}^R \right) & -M_{hh}^{-1}\left(B_{hh} - \frac{1}{4}\rho cV\frac{Q_{hh}^I}{k} \right) \end{bmatrix} \tag{2.60}$$

And $\bar{u}_h = \begin{bmatrix} u_h \\ \dot{u}_h \end{bmatrix}$

The basic algorithm for the pk method involves the following steps:

1. Make an initial guess of the frequency for the mode
2. Calculate the corresponding reduced frequency k and airspeed V
3. Determine (by interpolation) the aerodynamic stiffness and damping matrices Q_{hh}^R, Q_{hh}^I
4. Determine the frequencies for the system at this flight condition using Equation 2.59
5. Take the frequency solution that was closest to the initial guess and repeat steps 1-5 until the frequency converges
6. Store the final modal frequency and modal damping
7. Consider the next mode of interest using the same procedure until all modes of interest have been investigated

2.4 Optimization techniques

2.4.1 Brent's – Dekker Line search method

This algorithm is intended on finding the minimum of a function of one variable without using its derivatives. This algorithm is commonly used when finding the minimum of a function $g(\boldsymbol{x})$ of several variables. To this end the following function often needs to be minimized

$$\gamma(\lambda) = f(\boldsymbol{x}_0 + \lambda \boldsymbol{s}) \tag{2.61}$$

where $\boldsymbol{x}_0$ and $\boldsymbol{s}$ are fixed and λ is a scalar variable. This problem essentially describes a one-dimensional search beginning form $\boldsymbol{x}_0$ in the direction of $\boldsymbol{s}$.

The algorithm finds an approximation to the minimum of a function f defined on the interval $[a, b]$. There are six significant points $a, b, u, v, w \; and \; x$ not all distinct. These points are initialized as follows:

$$v = w = x = a + \left(\frac{3 - \sqrt{5}}{2}\right)(b - a) \tag{2.62}$$

The number $\left(\frac{3-\sqrt{5}}{2}\right)$ comes from the golden section search algorithm and is somewhat arbitrary. The points defined in 2.62 serve a specific purpose:

- Points a and b define the interval within which a local minimum lies
- x: of all the points at which f has been evaluated, x is the one with the least value of f
- w is the point with the next lowest value of f
- v is the previous value of w
- u is the last point at which f has been evaluated

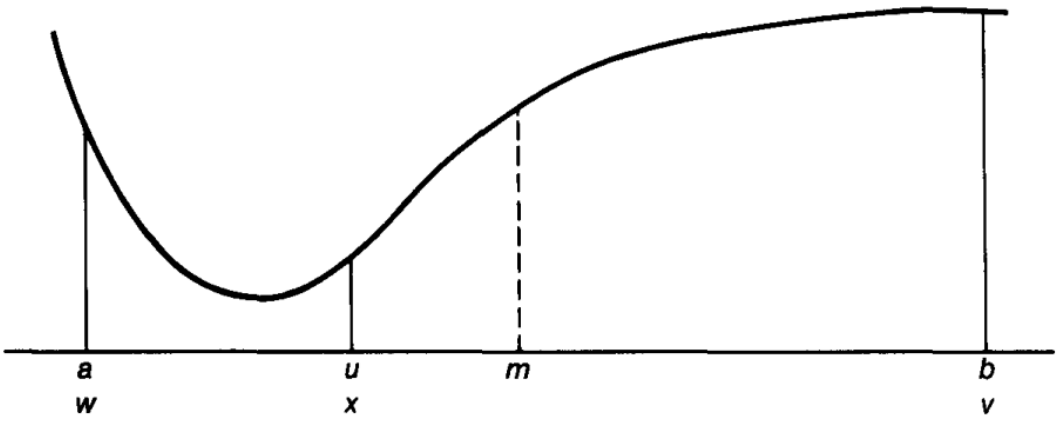


Figure 2-12A possible configuration of points [6]

A typical iteration of the algorithm evolves as follows:

1. Let $m = 1/2(a + b)$ be the midpoint of the interval
   a. If $|x - m| \leq 2 \cdot tol - \frac{1}{2}(b - a)$ then the algorithm terminates with x as the minimum
   b. Otherwise, the numbers p and q are calculated such that $x + \frac{p}{q}$ is the minimum of the parabola passing through points $(v, f(v)),\ (w, f(w)),\ (x, f(x))$
2. Let e be the value of p/q
   a. If $|e| \leq tol, q = 0, x + p/q \notin [a, b]$ or $|p/q| \geq 1/2|e|$ then the golden ration step is performed:

$$u = \begin{cases} \frac{\sqrt{5}-1}{2}x + \frac{3-\sqrt{5}}{2}a, & if\ x \geq m \\ \frac{\sqrt{5}-1}{2}x + \frac{3-\sqrt{5}}{2}b, & if\ x < m \end{cases} \quad (2.63)$$

   b. Otherwise $u = x + p/q$ (the distances $|u - x|,\ u - a,\ b - u$ must be at least tol
3. f is evaluated at the new point u the points $a, b, v, w\ and\ x$ are updated and the cucle repeats

The algorithm typically terminates when $x = b - tol$ or $x = a + tol$ after a parabolic interpolation has been performed where the condition that $|u - x| \geq tol$ has been enforced. The next parabolic interpolation point lies close to x and b so u is forced to be $x - tol$.

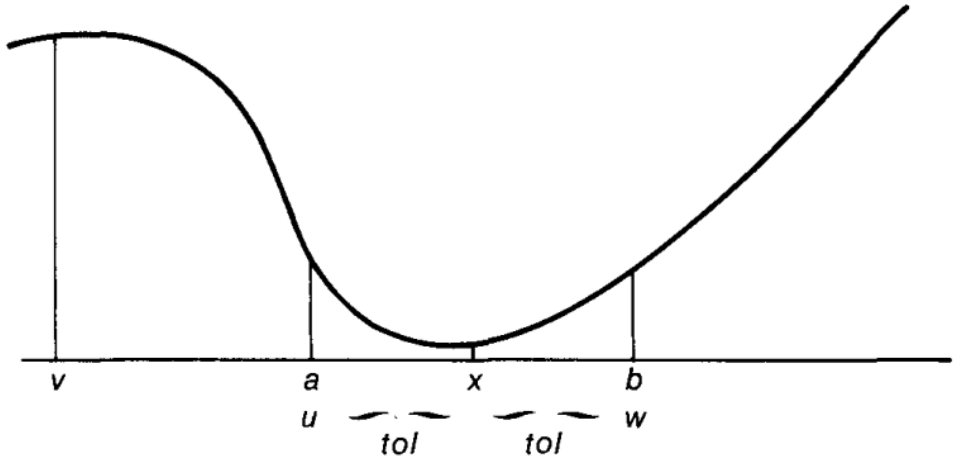


Figure 2-13 typical terminal configuration of important points [6]

2.4.2 Powell's Method

Powell's method is a modified cyclic coordinate search. Both methods aim to minimize a multivariate function without using any of its derivatives. That's why they are called zero-order methods.

The Cyclic coordinate search method is in essence a series of line-search optimization performed cyclically over all inputs of the function. The search starts from an initial $x^{(0)}$ and optimizes the first input:

$$\boldsymbol{x}^{(1)} = \arg_{x_1} \min f\left(x_1, x_2^{(0)}, \ldots, x_n^{(0)}\right) \quad (2.64)$$

Having solved this, it optimizes the next coordinate:

$$\boldsymbol{x}^{(2)} = \arg_{x_2} \min f\left(x_1^{(1)}, x_2, \dots, x_n^{(1)}\right) \tag{2.65}$$

This can also be expressed with the help of equation 2.61if $\boldsymbol{s}$ the i[th] basis vector for $i = 1, \dots, n$

Any line search algorithm can be used but Brent's Algorithm has seen wide usage in optimization libraries including SciPy's implementation of the Powell method.

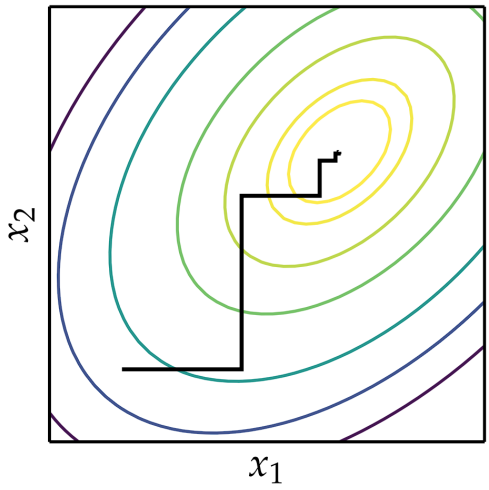


Figure 2-14Cyclic coordinate search [7]

As can be seen in Figure 2-14 a weakness of this method is the slow traversal of diagonal valleys where repeated small steps are taken in every direction. M.J.D. Powell's idea was to expand the cyclic coordinate search method so that it can search in directions that are not orthogonal to each other and are also not the basis vectors. In order to achieve this new capability, Powell algorithm maintains a list of search directions $\left[\boldsymbol{u}^{(0)}, \dots \boldsymbol{u}^{(n)}\right]$ which are initially the coordinate basis vectors $\boldsymbol{u}^{(i)} = \boldsymbol{e}^{(i)}$ for all i.

Starting at $\boldsymbol{x}^{(0)}$ Powell's method conducts a line search as before for each search direction in succession:

$$\boldsymbol{x}^{(i+1)} \leftarrow linesearch\left(f, \boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)}\right) for\ i\ in\ [0, \dots, n] \tag{2.66}$$

Next the all the search directions are shifted down by one index dropping the oldest search direction,$\boldsymbol{u}^{(0)}$

$$\boldsymbol{u}^{(i)} \leftarrow \boldsymbol{u}^{(i+1)}\ for\ i\ in\ [0, \dots, n-1] \tag{2.67}$$

The last search direction is replaced with the direction from $\boldsymbol{x}^{(0)}$ to $\boldsymbol{x}^{(n+1)}$, which is the overall direction of progress over the last n line searches:

$$\boldsymbol{u}^{(n)} \leftarrow \boldsymbol{x}^{(n+1)} - \boldsymbol{x}^{(0)} \tag{2.68}$$

This process is repeated until convergence.

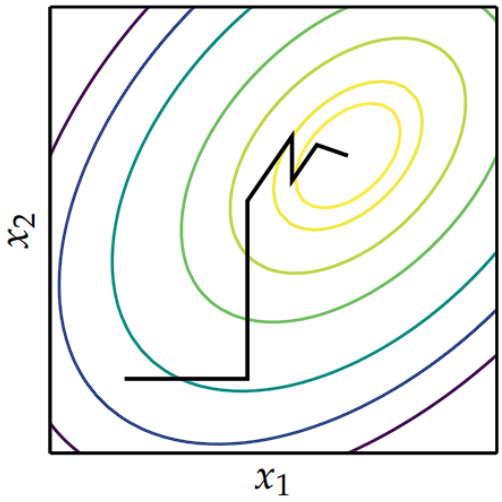


Figure 2-15Powell's method [7]

As can be seen in Figure 2-15 Powell's method begins the same as cyclic coordinate search but gradually learns conjugate direction.

2.4.3 Genetic Algorithm

Genetic algorithms mimic the logic behind Darwinian evolution, where the fitter individuals are more likely to pass on their genes to the next generation. The chromosomes of each individual define a single design point. At each generation the chromosomes of the fitter individuals are passed on to the next generation after undergoing the genetic operation of *crossover* and *mutation*

Chromosomes are often represented by a binary array but in this case an array of real values describes the design space better.

Initialization

The Genetic Algorithm begins by initializing a population with random chromosomes

Parent Selection

The selection process is the process of choosing which chromosomes will be selected to create offspring for the next generation after being subjected to the crossover and mutation genetic processes. There are many methods of selecting which individuals get to mate and combine their genes some of the most popular include:

- Roulette Wheel selection: Where each individual is assigned an area of an imaginary roulette proportional to their fitness. Then the roulette is spun until the desired number of parents is reached. This method is simple and effective in most cases. It can become overwhelmed by individuals who are much fitter than the rest of the population

- Tournament selection: A subset of individuals is selected are random and then the best from this subject is selected as a parent. The process is repeated until the desired number of parents is reached
- Steady state Selection: A steady portion of the population is replaced at each generation. Only the weakest individuals are replaced by offsprings, this method insures that the fittest individual is maintained most of the time. A disadvantage of this method is that it is slow to adapt as only a small portion of the population is replaced at every generation

Crossover

Crossover is the way the chromosomes of the parents are combined to create an offspring. There are several crossover schemes. The most commonly used are :

- Single point crossover
  Iin this scheme the first portion of parent A forms the first portion fo the child chromosome and the latter portion of parent B's chromosome forms the latter part of the child chromosome.

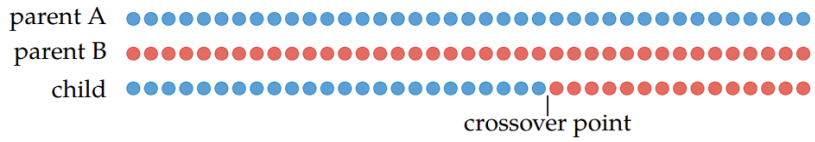


Figure 2-16 Single Point Crossover [7]

- Two-point crossover
  Same as before but a second crossover point is added.

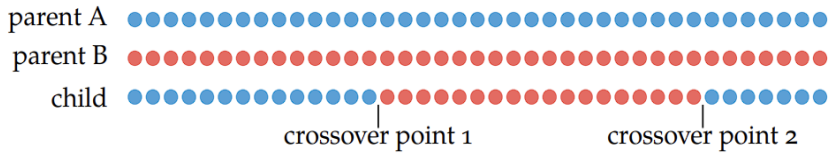


Figure 2-17 Two-point Crossover [7]

- Uniform crossover
  In this scheme every chromosome has a 50% chance of coming from parent A or parent B.

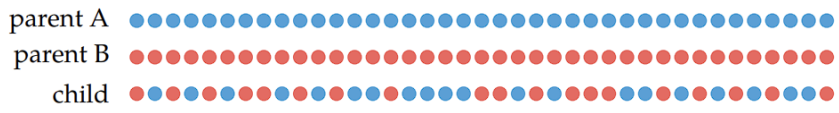


Figure 2-18 Uniform Crossover [7]

Mutation

Mutation is a necessary step of the genetic algorithm since it allows new traits to develop which were not present in the initial population. In essence a random value is added to a

random chromosome of a randomly selected offspring. The chance of a mutation occurring to an individual is called the *mutation rate*

Elitism

Elitism is a strategy that ensures that a fixed number of the most fit individuals survive to the next generation

Genetic algorithms have a very wide spectrum of application and are very adaptable to many problems. But there is a catch. All those parameters and strategies affect the results greatly and there exists no standard method of selecting every parameter. So, it is the responsibility of the researcher too understand the problem at hand and through experimentation and model hyperparameter tuning techniques determine a good enough set of parameters.

2.4.4 Neural Networks

What are Artificial Neural Networks (ANN's)

An artificial neural network is a computational model which draws inspiration from the way a biological brain works. The basic structure of an ANN consists of interconnected nodes (or neurons) organized in layers.

The base unit of this structure is a neuron. A neuron has a very simple structure. It accepts an arbitrary number of inputs along with a bias and only has one output. The output can be binary or scalar. Within the neuron a summation is performed using the weight of each input and the bias. The result of each summation is further processed through an "activation function" and the output is reached. The activation function allows for nonlinear behavior of the neural networks. It can be any function, but some functions have prevailed.

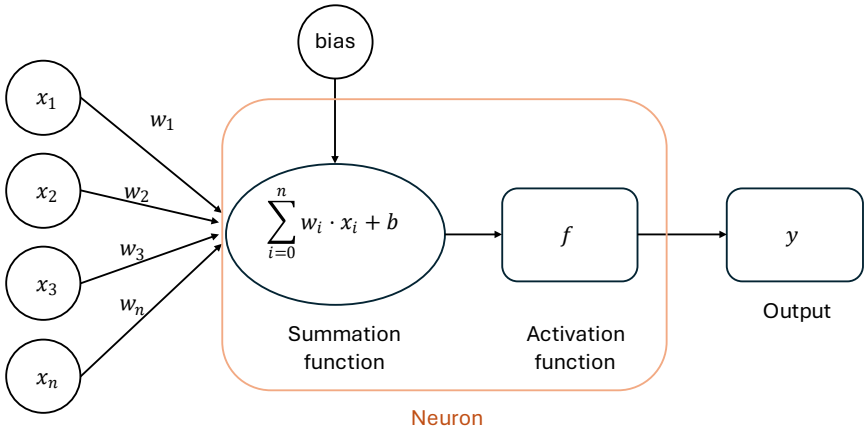


Figure 2-19 The Structure of a Neuron

The basic architecture of a neural network consists of:

1. An input layer which is responsible for receiving the initial data. Each neuron in the input layer depicts a specific feature or attribute of the input data

2. Hidden layers, which are intermediate layers between the input and output layers which are responsible for complex feature recognition.
3. The weights and biases which are to be used in the neural network, and which are the parameters to be trained in order for the neural network to make meaningful predictions.
4. An output layer which generated the final predictions. The output can be binary for classification problems or scalar for regression problems.

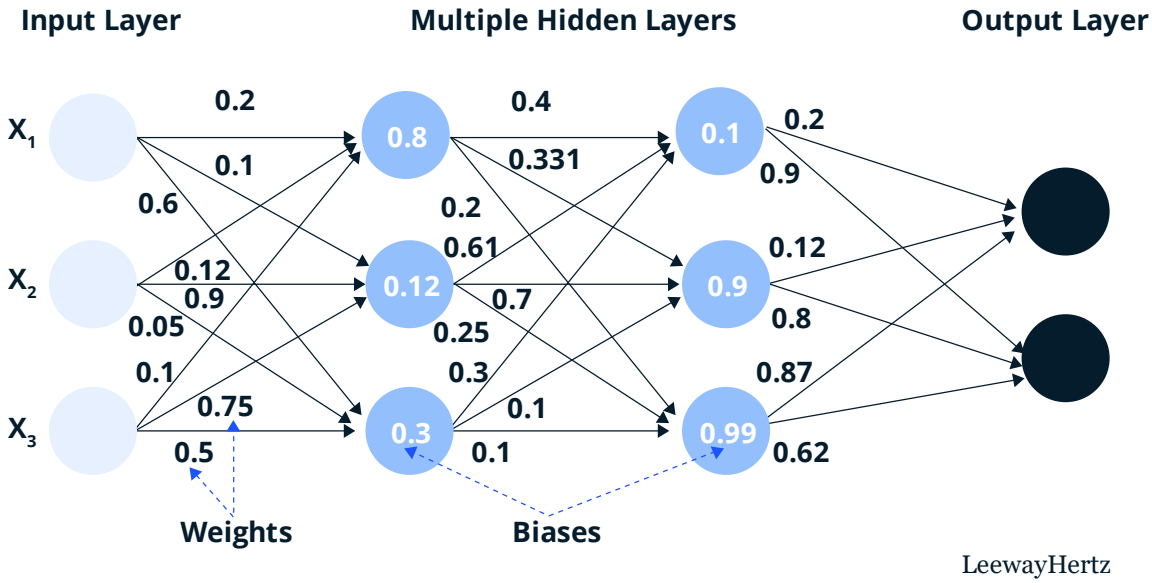


Figure 2-20 Structure of a Neural Network [8]

The number of Hidden layers in a neural network can vary. Neural networks which do not have any hidden layers are called “shallow”, on the other hand Neural Networks with hidden layers are called Deep Neural Networks.

Artificial Neural Network Training

As stated previously, for a neural network to be useful it has to be trained. Training involves gathering data similar to the ones that the NN is supposed to predict and then performing an optimization of the weights and biases within this network to minimize some loss function. A loss function is a measure of the amount of error. There are many loss functions commonly used for Neural networks, but when a Neural network is trained on a regression problem the loss function is usually one of the following:

1. Mean Squared Error (MSE)

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2 \qquad (2.69)$$

2. Root Mean Squared Error

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2} \quad (2.70)$$

3. Mean Average Error (MAE)

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}| \quad (2.71)$$

Where: $\hat{y}\ is\ the\ predicted\ value\ of\ y$

Minimizing any of these functions on test data that the network hasn't seen before is the goal of training.

To achieve this goal first the available data which contain both the input and the wanted output, are randomly split between training data (those that will be used for the adjustment of weights and biases) and test data (those which will be used to judge the performance of the neural network. Then through a process called backpropagation (the exact explanation of which is beyond the scope of this thesis) on batches of the training data the weights are updated in such a way so as to reduce the value of the loss function for this particular set of data points. Once all the batches of data have been run through the network and the weights have been adjusted accordingly, we say that the network has completed one pass through the training data This procedure estimates the negative gradient of the loss function and so the network works its way down the slope of the loss function. The number of times this procedure is repeated is called epochs. The number of epochs should be such that an optimum solution is reached without overfitting the network to the test data.

Hyperparameter Tuning

As we have seen, in order to define a neural network many parameters have to be chosen in advance. Those parameters include the number of hidden layers, the number of neurons in each layer along with the activation function of each layer to name a few. The best set of parameters is not known a priori and there is no method of selecting the optimal set.

The process of optimization of a model's hyperparameters is called *hyperparameter tuning*. There are many optimization algorithms that achieve this. One of the more interesting ones and the one that will be employed for this application is the HyperBand algorithm [9]. The algorithm is an improvement of the Successive Halving algorithm.

The Idea behind the Successive halving algorithm is quite simple. It uniformly allocates a budget of resources (computational time) to a set of hyperparameters evaluates the performance of each test, reject the bottom 50% and repeat until one configuration remains. The problem with this algorithm is that it is not known whether testing a larger number of configurations with limited budget each or a smaller number of configurations with more budget will result in a better outcome.

This is the problem that the HyperBand algorithm was made to tackle. It does this by performing a grid search over feasible values of n and r, where n is the number of configurations to be tested and r is the minimum resource that is allocated to all configurations before they are allowed to be discarded.

3 Methodology

In this chapter the theoretical background developed in chapter 0 will be put to practice to solve an actual example problem. This chapter will explain the problem that is to be solved. It will explain how the model is set up for aeroelastic analysis as well as the process used to optimize the design.

3.1 Problem Introduction

The problem that has been chosen as an example application of all the theory developed in the previous chapter is the flutter characteristics study of the ASW 28 glider main wing and the subsequent material design optimization to tailor the flutter characteristics to specific requirements. This choice of this specific aircraft is because it has an extremely large aspect ratio thus making it also very flexible. Moreover, the main goal of a glider is to achieve the best glide angle so reduction of weight is of utmost importance and so these vehicles are ideal for this application as they combine the high likelihood of flutter occurring with the need to optimize for the lowest weight possible.

“The ASW 28 is Schleicher’s high-performance glider for the FAI-Standard Class with 15m span.” [10] The technical data of this aircraft are summarized in the following table:

Table 3 Technical Data of ASW 28 glider [10]

Technical Data

Span	15 m	49.2 ft
Wing area	10.5 m²	113 sqft
Wing aspect ratio	21.4	
Winglet height	0.5 m	1.64 ft
Fuselage length	6.585 m	21.6 ft
Cockpit width	0.64 m	2.1 ft
Cockpit height	0.8 m	2.62 ft
Height at tail	1.3 m	4.26 ft
Empty mass	235 kg	518 lbs
Max. take-off mass	525 kg	1157 lbs
Min. wing loading	29 kg/m²	5.93 lbs/sqft
Max. wing loading	50 kg/m²	10.2 lbs/sqft
Useful load, max.	130 kg	286 lbs
Water ballast wing	180 l	397 lbs
Max. speed	285 km/h	154 kts
Min. sink	0.55 m/s	108 ft/min
Best glide ratio	45	

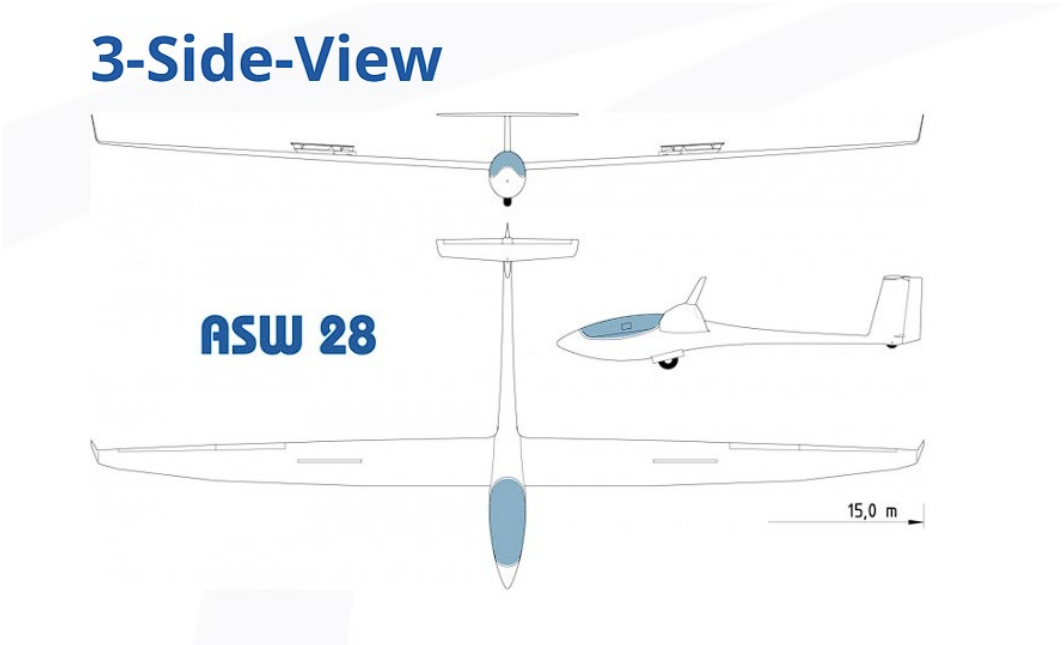


Figure 3-1 Front, side and top view of ASW 28 glider [10]

The analysis of the main Wing of the ASW 28 will include the following steps:

1. Model set up in Altair HyperMesh for modal analysis and flutter analysis
2. Creation of a Python script to read Flutter results from Optistruct solver
3. Definition of Optimization problem parameters using different optimization techniques and implementation using Python.

3.2 ASW 28 main Composite Wing Model

3.2.1 Wing Geometry & Discretization

The external geometry of the Wing was imported to HyperMesh from a CAD file. The geometry was cleaned up so as to contain only the main wing of the aircraft and because of some inaccuracies and because of the non-metric length units used a scale transformation had to be used.

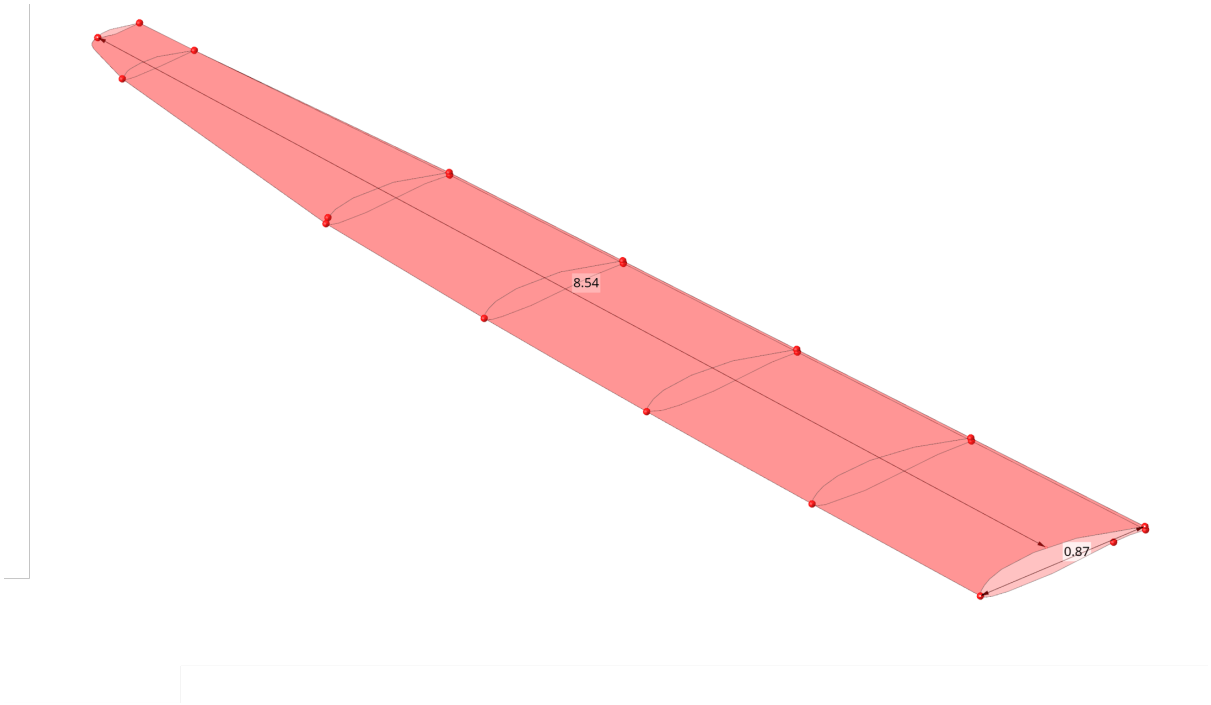


Figure 3-2 ASW 28 Wing external geometry (length in meters)

After the creation of the external geometry the internal structure of the Wing had to be created from scratch. Since the detailed drawings of the internal structure are not available, the internal structure geometry was improvised based on some pictures of the disassembled wing. The main features of most modern wings' internal geometry are the spars and ribs. Spars are the main structural member of the wing which run spanwise along the length of the wing and are responsible for carrying aerodynamic loads to the main fuselage. Ribs are structural members of the wing that run chordwise (perpendicular to the spars) and their purpose is to support the wing's skin so that it maintains the proper airfoil profile.

Modern glider wings usually have only one main spar and several ribs. For this application a single main spar of rectangular cross section which tapers towards the wing's tip and 6 ribs are used.

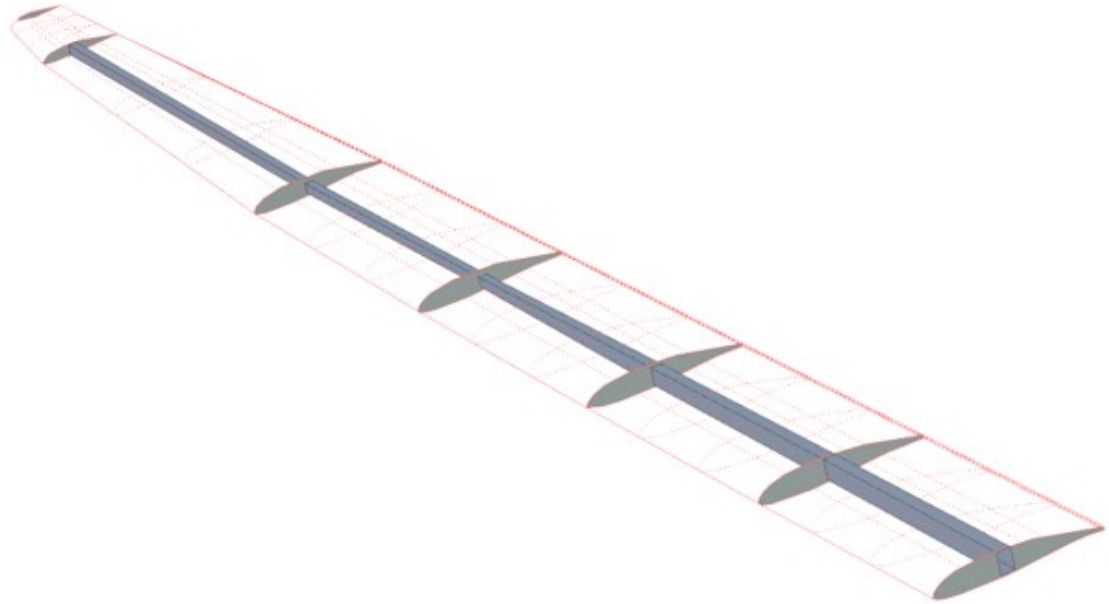


Figure 3-3 ASW 28 Wing Internal geometry

The discretization of the geometry was achieved using shell elements arranged in a mesh which was created with the help of the "Panel Mesh" utility of HyperMesh for the skin of the wing since this function is especially well suited for this type of panel-like geometry, and the "General 2D Mesh" utility for the internal geometry. The average element size has a side length of $0.02\ [m]$.

The discretization does not need to be very accurate since for the flutter analysis we only need to capture enough detail to describe the first few eigenmodes accurately and we do not care about the exact stresses at specific points within the structure.

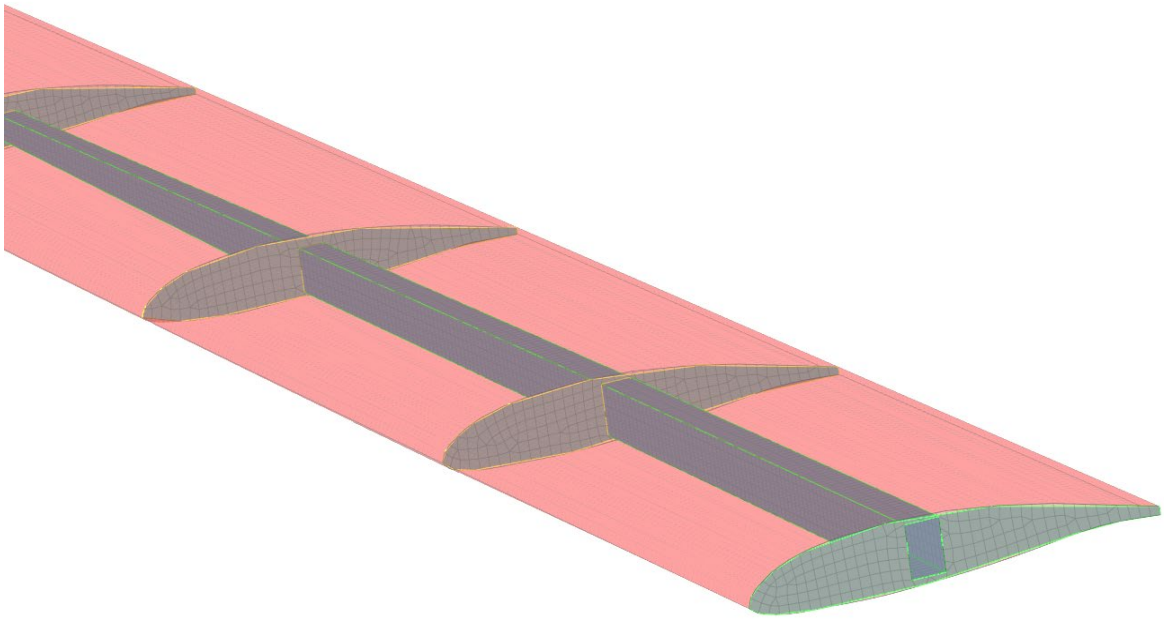


Figure 3-4ASW 28 Wing Internal mesh

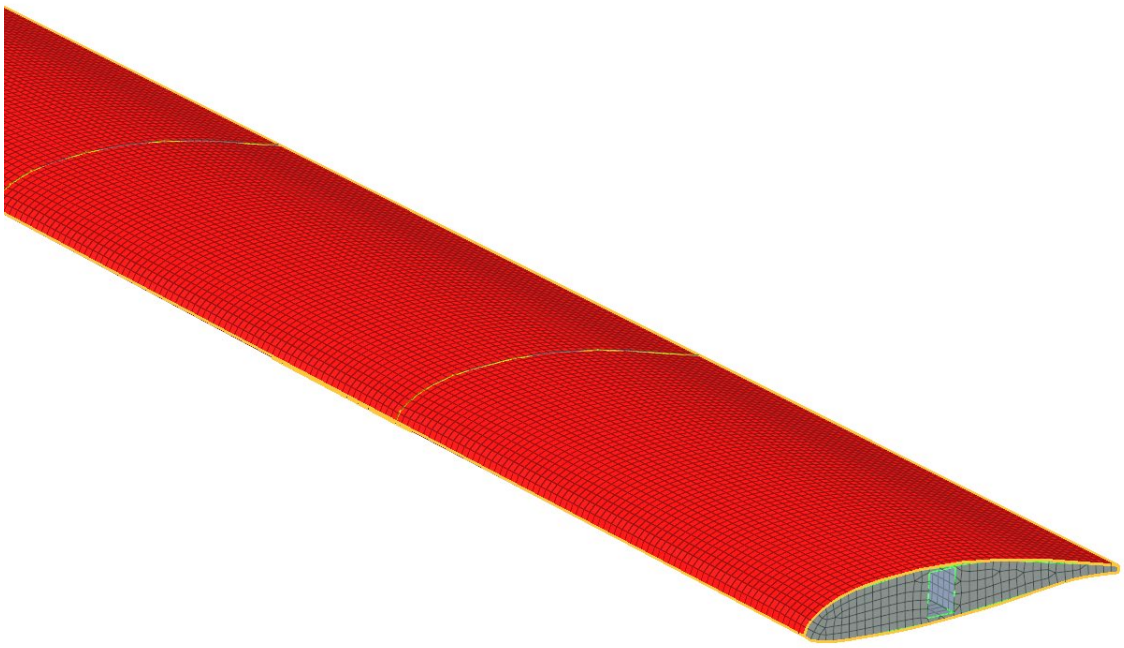


Figure 3-5 ASW 28 Wing skin mesh

3.2.2 Material properties Definition.

The material properties assigned to different components of the wing structure are somewhat arbitrary as this information is not publicly available. This should not pose any problems to the goal of this thesis since during the optimization, material properties are subject to change anyway.

A common composite material for the skin of high-performance gliders is fiberglass or carbon fiber, because these materials offer high strength, low weight and a smooth aerodynamic surface. For this application a carbon fiber / Epoxy composite was chosen. For the Optistruct solver this kind of composite material can be defined using the MAT8 material card with the following properties [11]:

- $E_1 = 125\ GPa$
- $E_2 = 8.41 GPa$
- $\nu_{12} = 0.35$
- $G_{12} = 4.23\ GPa$
- $\rho = 1517\ kg/m^3$

This material is a laminated composite and although the material properties have been sufficiently defined the laminate structure hasn't. To define the laminate configuration a PCOMP property card is needed. This card is essentially a list containing the properties of each laminate layer. These properties are:

1. The material of the laminate layer defined as a composite material card, in this case a MAT8 card
2. The thickness of each laminate layer defined as a float
3. The angle in degrees at which the primary axis (axis 1) of the laminate layer is oriented

For this application a six-layer laminate composite will be created using the carbon fiber epoxy orthotropic material with angles alternating between +45 and -45 degrees and a thickness of $0.5\ mm$ for each layer.

For the internal structure of the wing an aviation grade aluminum is used. More specifically aluminum 6061 is used, which has the following properties:

- $E = 69\ GPa$
- $\nu = 0.33$
- $\rho = 2700\ kg/m^3$

This is an isotropic material and is defined within Optistruct with a MAT1 card which essentially lists these material parameters in a specific manner.

Similarly, the material definition alone is not enough as we are working with shell elements, a property which defines the thickness of the material need to be present. In Optistruct this property is defined using a PSHELL property card which only needs to hold a material reference and a thickness value. In this chase the thickness is uniform everywhere and equal to $t = 2mm$

3.2.3 Boundary Conditions

The boundary conditions defined for this problem are similar to that of a fixed free cantilever beam. The nodes at the root of the wing are fixed along every degree of freedom with an SPC (Single Point Constraint) entry. These boundary conditions assume that the wing's root is completely fixed on its support which works well for a detached wing which is placed on a

test bench but are only an approximation of the real flight situation where the wing is fixed to the fuselage which is not static during flight.

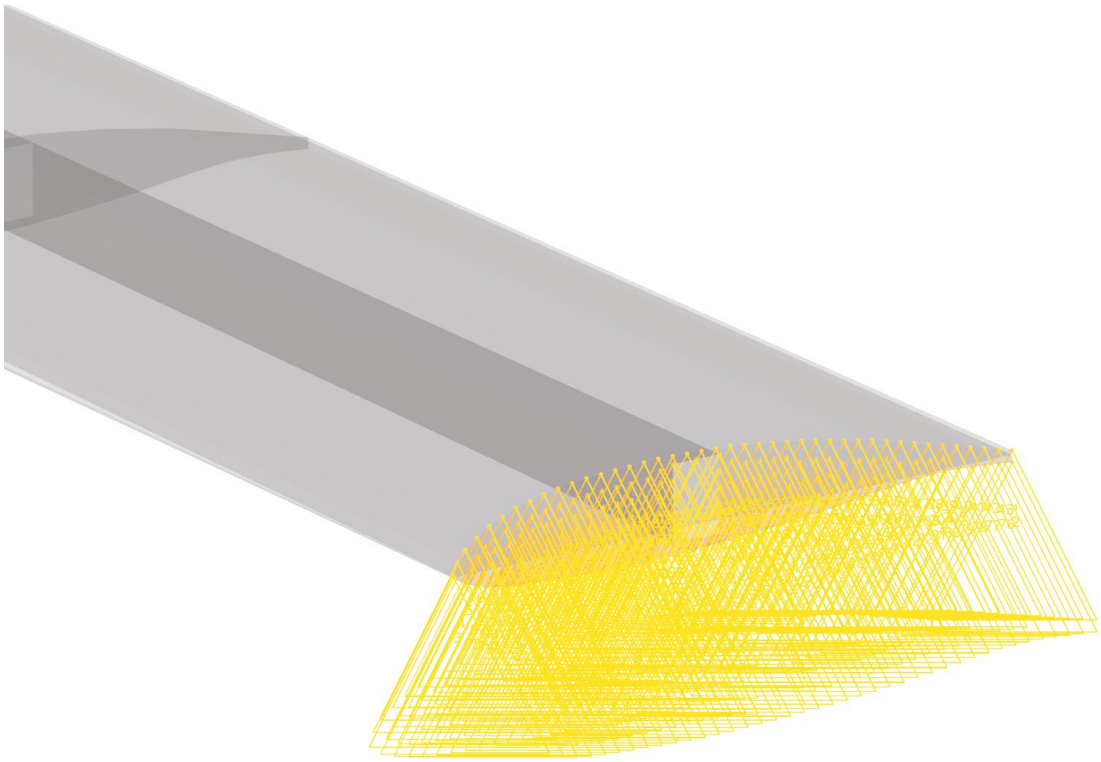


Figure 3-6ASW 28 Wing Boundary conditions

3.2.4 Aerodynamic Grid

To create the vortex-lattice the lifting surface of the wing needs to be discretized. The way this is modeled in Optistruct is through the CAERO1 card which defines an “aerodynamic macro element” with a simple two-dimensional quadrilateral geometry which is split into a predefined number of boxes in the chordwise and spanwise directions. To define the CAERO1 card one must first define the four corner points in a specified order:

- The first point is on the leading edge and on the root of the wing
- The second point has the same y coordinate as the first but lies on the trailing edge of the wing
- The third point lies on the tip of the wing and at the trailing edge
- The Fourth point has the same Y coordinate as the third but lies on the leading edge of the tip of the wing

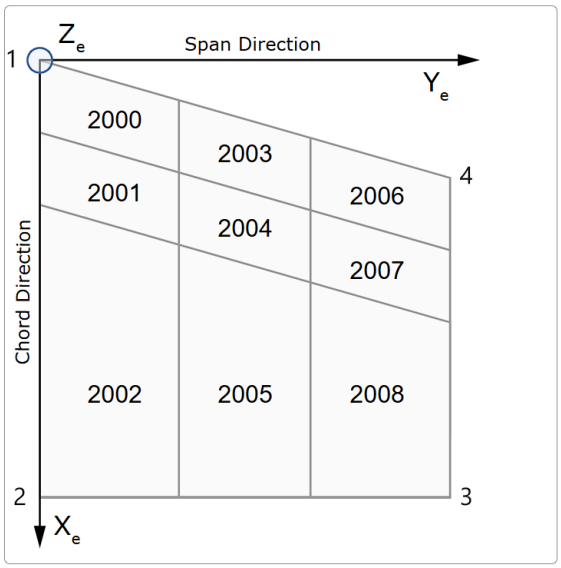


Figure 3-7 Coordinate System of CAERO1 Aerodynamic panel

This allows for the modelling of only simple trapezoidal geometries. Since the projection of the ASW28 glider wing on the XY plane does not fit well within a single trapezoidal shape two of these macro elements were used to capture the variable taper ration of this wing. These panels can be seen in the following image of the model.

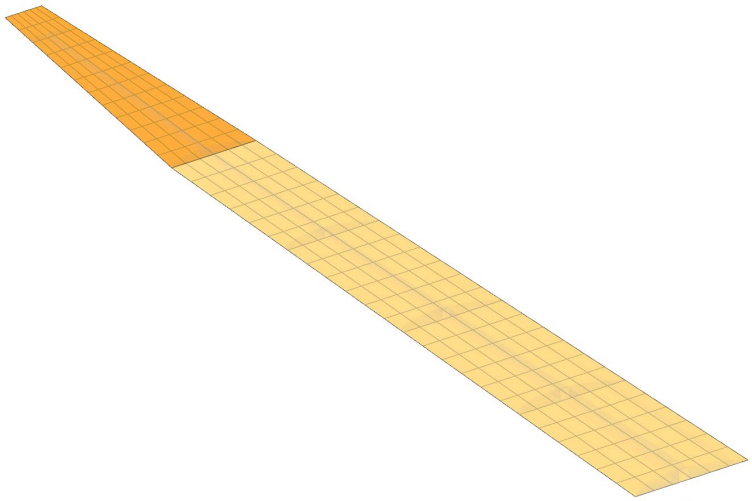


Figure 3-8 CAERO1 macro elements of the ASW 28 Wing Model

The next step of defining the CAERO1 elements is defining the discretization into aerodynamic boxes through two integer values NSAPN and NCHORD which define the number of spanwise and chordwise boxes respectively.

For the Inner CAERO1 macro element:

$$NSPAN = 24, \qquad NCHORD = 6$$

For the Outer CAERO1 macro element:

$$NSPAN = 12, \qquad NCHORD = 6$$

This discretization is chosen so that the aspect ratio of the boxes is less than about three and the chordwise length of each box is less than $\Delta x < 0.08 V_{max} / f_{max} \approx 0.65m$. These two suggestions for the discretization of the aerodynamic panels are specified in [5]

3.2.5 The Spline

In Flutter analysis the SPLINE entry is used to couple the structural and aeroelastic domains. For this application a SPLINE1 entry is used which defines a surface spline (linear splines are also available but do not apply in this case.) To define the SPLINE1 the following entries are needed:

- The CAERO Id which was defined in the previous step
- The Id's of the first and last aerodynamic boxes to be included (All the aerodynamic panels are selected for this analysis)
- A set of nodes from the structural grid

The selection of the structural nodes is important. Typically, not all the nodes of the structure are selected. Only a subset of the nodes on the bottom or upper surface of the wing are selected. These nodes need to be under the area covered by the CAERO entry and the aerodynamic boxes that were selected. Ideally each aerodynamic grid point has one corresponding structural node directly above or below it, although this is not feasible in most cases

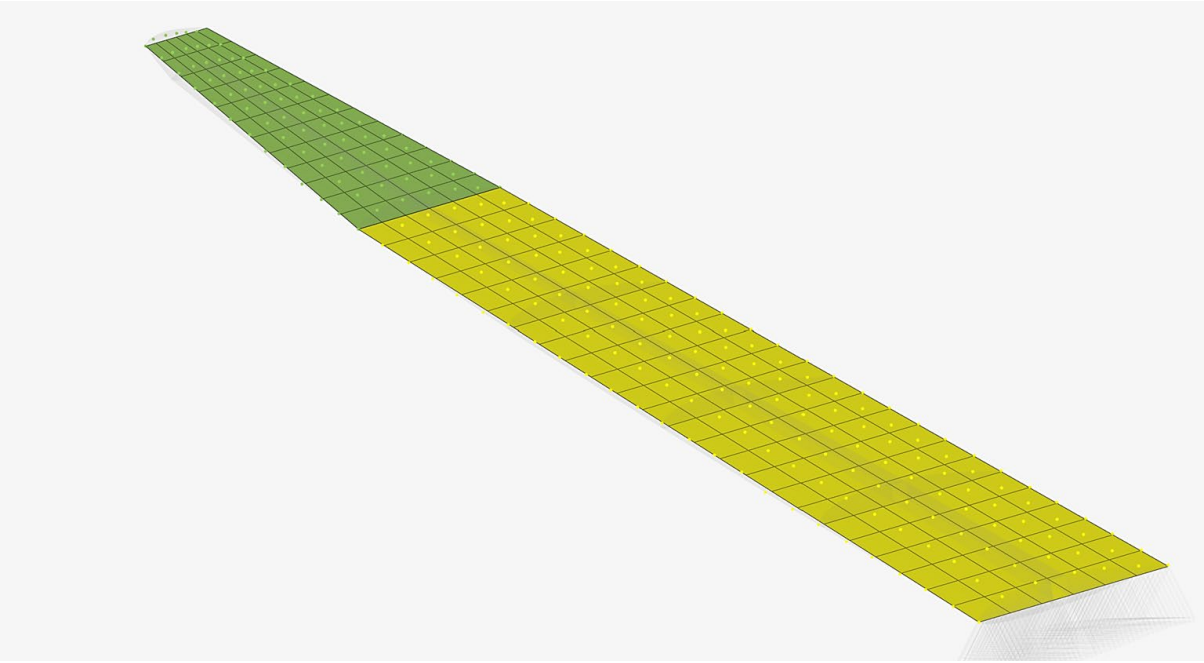


Figure 3-9 SPLINE1 entries of the ASW 28 Wing model

As can be seen in Figure 3-9 two separate SPLINE1 entries were made in this model one for each CAERO entry defined previously. In this case nodes from the top surface of the wing are selected. The nodes are in series of along the span of the wing at various chord percentages, so as to closely match the aerodynamic grid points.

3.2.6 Aeroelastic Problem Setup

To set up the Aeroelastic Flutter analysis several parameters need to be defined.

The AERO card:

The AERO bulk data entry card defines basic parameters for dynamic aeroelasticity regarding mainly the conditions of flight. The entries of this card are summarized as follows:

- VELOCITY: has no effect for flutter analysis since the velocity is varies during the analysis and defined elsewhere but cannot be left blank so unity is entered
- REFC**:** Reference chord length used for the calculation of reduced frequency and lift and drag coefficients if requested $REFC = 0.92m$
- RHOREF**:** Reference density $RHOREF = 1.225\ kg/m^3$ which is the density of air at sea level
- SYMXZ**:** which defines symmetry for the XZ plane and can have three values
  - -1**:** for antisymmetry
  - 0: for no symmetry
  - 1: for symmetry

  for this application $SYMZX = 1$
- **SYMXY:** defines symmetry for the XY plane in a similar way. In this analysis $SYMXY = 0$

The MKAERO1 card:

The MKAERO1 card is a bulk data entry card which is used to input a table of Mach Numbers and Reduced Frequencies for which the aerodynamic matrices are calculated.

The format of the MKAERO1 card has two column entries with a maximum of eight elements each. One column is for the Mach number entry while the other for the Reduced Frequencies. The aerodynamic matrices are computed at every pair of values of reduced frequency and Mach number.

There are no concrete recommendations for the range of reduced frequencies that must be covered by this entry, but since the aerodynamic matrices are interpolated for the actual resultant reduced frequency logic dictates that the range of reduced frequencies must be at least greater than the resultant range of reduced frequencies. Of course, this cannot be known a priori since the resultant reduced frequencies are only made known after the analysis is run. For this analysis quite a wide range of reduced frequencies was used after consulting many examples of this type of analysis.

In case more than eight values are required for reduced frequency or Mach number a second MKAERO1 entry can be made.

The values used are as follows:

$$\vec{M} = 0.0, \qquad \vec{K} = \begin{bmatrix} 0.2 \\ 0.4 \\ 0.8 \\ 1.6 \\ 3.2 \\ 6.4 \\ 10 \\ 14 \end{bmatrix}$$

As can be seen there is only one Mach number $M = 0$ which means that incompressible flow is assumed. The aerodynamic matrices are calculated at every point (M_i, K_j)

The FLFACT card:

The FLFACT bulk data entry card is a card that specifies a series of aerodynamic factors. These factors are used to define:

1. Density ratios
2. Mach Numbers
3. Reduced Frequencies or Velocities (PK Method only).

These factors can be defined using two different formats.

1. In Format 1 a series of values is directly entered into the card
2. In Format 2 the so-called THRU format is used defining a series of values using:
   a. F1: The first factor
   b. FNF: the final factor
   c. NF: The Number of factors (integer)
   d. FMID: The intermediate aerodynamic factor

   The actual series of values produced when using the THRU format are calculated using the following formula:

$$\frac{F1(FNF - FMID)(NF - i) + FNF(FMID - F1)(i - 1)}{(FNF - FMID)(NF - i) + (FMID - F1)(i - 1)}, \qquad where\ i = 1,2 \dots, NF \quad (3.1)$$

Note that when $FMID = \frac{F1+FNF}{2}$ the factors are equally distributed between F1 and FNF

For this Analysis three FLFACT Entries are needed:

- FLFACT 1: Is the density factor(s) and has a value of 1. This factor is multiplier of the Reference Density define in the AERO card and indicates that the analysis is to be performed at $1 \times RHOREF$
- FLFACT 2: Is the Mach Number(s) and has a Value of 0. This factor is the Mach Number at which the analysis is to be performed.

- FLFACT 3: Is the Velocity/ties at which the analysis is to be performed. This FLFACT is defined using the THRU Format with factors:
  - $F1 = 20m/s$
  - $FNF = 320m/s$
  - $NF = 30$
  - $FMID = 160$

The analysis is performed for every combination of combination of density Mach number and velocity in the FLFACT entries

The Flutter card:

The flutter bulk data entry card specifies the method and parameters of aeroelastic flutter analysis:

The most important fields of this card are

1. METHOD the method can be one of K, PK, PKNL, KE for this analysis the PK method is used.
2. DENS: a reference to the FLFACT Bulk Data entry which specifies the density multipliers
3. MACH: a reference to the FLFACT Bulk Data entry which specifies the Mach number
4. VEL: a reference to the FLFACT Bulk Data entry which specifies the velocities
5. IMETH the interpolation method for the aerodynamic matrix which can be either L or S for linear or surface interpolation respectively. The default value of L is retained for this analysis

The EIGRL card:

The EIGRL bulk data entry card defines the data required to perform real eigenvalue analysis with the Lanczos method. The main fields of this card are:

1. V1, V2 Frequency range of analysis
2. ND Number of desired eigenfrequencies

For this analysis, the Values $V1 = 0.0\ Hz$ and $ND = 8$ are used. These values mean that the first eight eigenfrequencies starting from zero Hertz will be calculated.

Subcase Definition:

For the subcase definition the “Aerodynamic Flutter” analysis type is selected and then

- The FMETHOD field requires a reference to the Flutter card
- The SPC Field requires a reference to the SPC load collector
- The METHOD Field requires a reference to the EIGRL
- The CMETHOD Field requires a reference to an EIGC card but can be left blank if no complex eigenvalues need to be computed (it is blank in this case since no structural damping is considered)

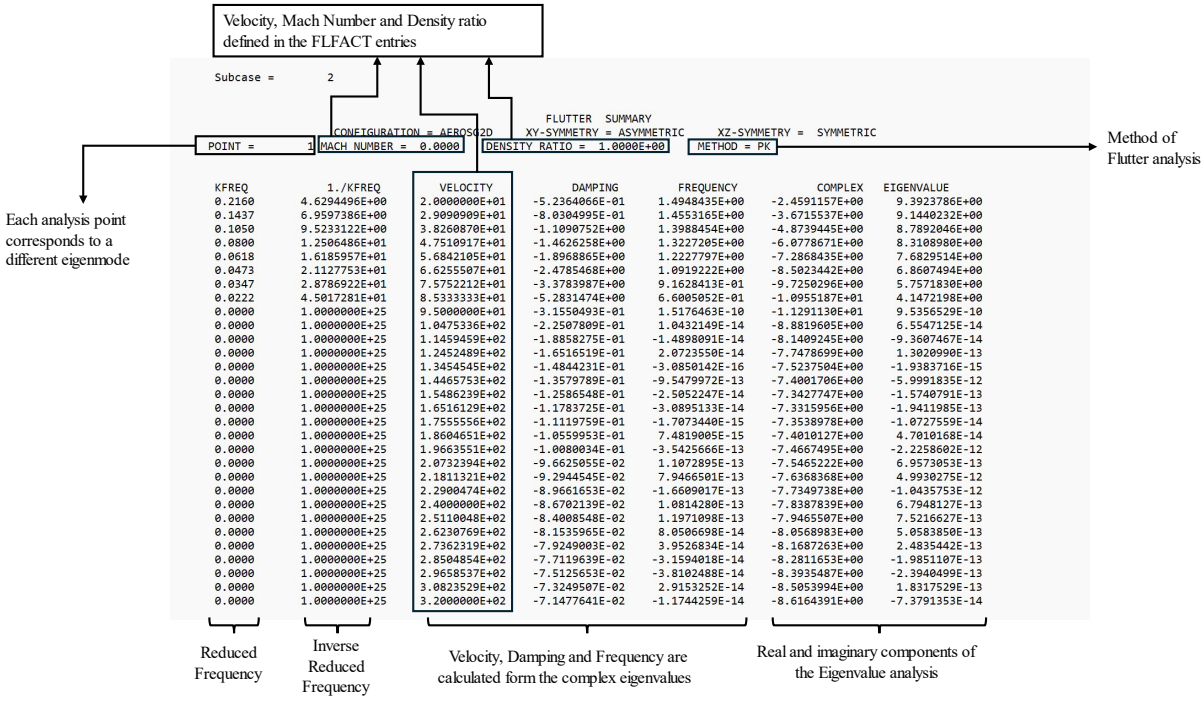
- The SMETHOD Field requires a reference to a damping curve TDMP but can be left blank if structural damping is not considered. (it is blank in this case since no structural damping is considered)

The results of the aeroelastic flutter analysis are presented in chapter 4.2

3.3 Optistruct – Python Interface

3.3.1 Results of Flutter Analysis & Python

The output of the analysis defined in the previous section is a .flt text file in addition to the typical Optistruct .out output file. It has a very specific format. It is organized in blocks of information each corresponding to a different combination of Mach number, Density (defined in the FLFACT entries) and Eigenmode number. The most important aspects of the .flt file are outlined below.



The typical .out file, which is also output contains information about whether the solver encountered any exceptions or warning and most importantly information about the mass of the structure.

From the data contained in the .flt file two very useful plots can be produced.

1. The V-g plot plots the damping of each eigenmode (y-axis) against Velocity (x-axis)
2. The V-f plot plots the frequency of each eigenmode (y-axis) against Velocity (x-axis)

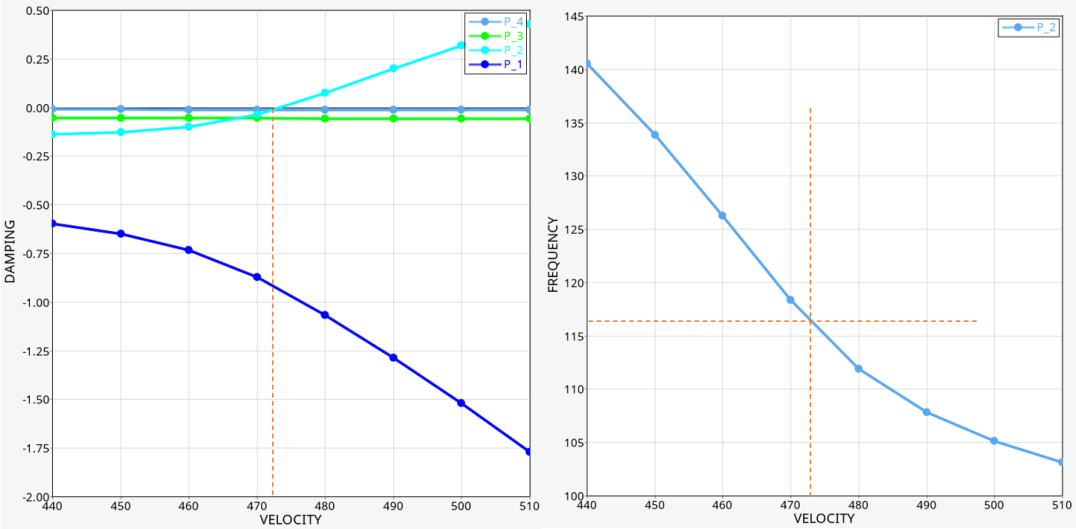


Figure 3-10 Example of V-g and V-f plot containing four eigenmodes [12]

From these data one can determine the Flutter speed by examining the V-g plot. More specifically a mode diverges when the damping of that particular mode changes from negative to positive. Note that many modes can diverge, but the flutter speed of the structure is determined by the mode that diverges at the lowest speed.

The data blocks contained in the .flt file are read by a python script and stored as pandas data frames. To determine the flutter velocity the sign changes of each eigenmode are monitored until a negative to positive change is detected. Then all the points where there is such a sign change are stored and the flutter speed is determined by the minimum speed at which such a sign change occurs.

A separate function is responsible for reading the .out file and determining if the solver completed the analysis successfully as well as recording the mass of the structure.

3.3.2 Modifying Optistruct's input using python

In order to optimize the composite material of the structure one needs to be able to modify the composite material's property programmatically so that one can try many different variations and arrive at an optimum. Optistruct makes this quite easy because it uses a text input file that encodes all the information of the model. This file is called a .fem file and can be output from HyperMesh as a comma separated text file. To modify the composite material's property it is necessary to locate the correct part of this file and decode it. According to Optistruct's documentation the composite material's property is encoded in the following way.

Table 4 PCOMP encoding

1	2	3	4	5	6	7	8	9	10
PCOMP	PID	Z0	NSM	SB	FT	TREF	GE	LAM	+
	MID1	T1	THETA1	SOUT1	MID2	T2	THETA2	SOUT2	+
	MID3	T3	THETA3	SOUT3	Etc.				

Where:

- PCOMP: is the keyword indicating that a composite material property definition follows (string)
- PID: is the ID of the property (integer)
- NSM: is the non-structural mass per unit area (float)
- SB: allowable inter lamina shear stress (default 0.0) (float)
- FT: Failure Theory selection
- TREF: Reference stress free temperature (float)
- GE: Damping coefficient (float)
- LAM: Laminate option different ways to define the laminate. In the default case all plies must be defined one by one
- MIDi: The material ID of ply i (integer)
- Ti: The thickness of ply i
- THETAi: The angle of ply i
- SOUTi: Stress, Strain output request default NO (bool)

The quantities to modify are mainly the Ti and THETAi entries to cover the decision variables that will be needed.

3.4 Optimization Problem

In this chapter the implementation of the optimization algorithms discussed in the theory section 2.4 will be discussed.

3.4.1 Applying Powell's method

This algorithm is implemented using SciPy's [13] minimization function by selecting the *method = "Powell"* optional argument. In order to define the optimization problem some other important parameters need to be defined.

Decision Variables:

The first step is the definition of the decision variables. These variables consist of the thickness and angle of each layer in the composite material. Since the composite laminate consists of six layers that would mean that 12 decision variables would have to be used.

Because of the computational time required though, some assumptions have to be made in order to reduce the complexity of the optimization problem. Theses assumptions are:

- The layers are antisymmetric.

  This means that for every layer at height z_k with ply angle θ_k above the middle surface there exists an identical layer with the opposite ply angle $-\theta_k$ at height $-z_k$ from the middle surface

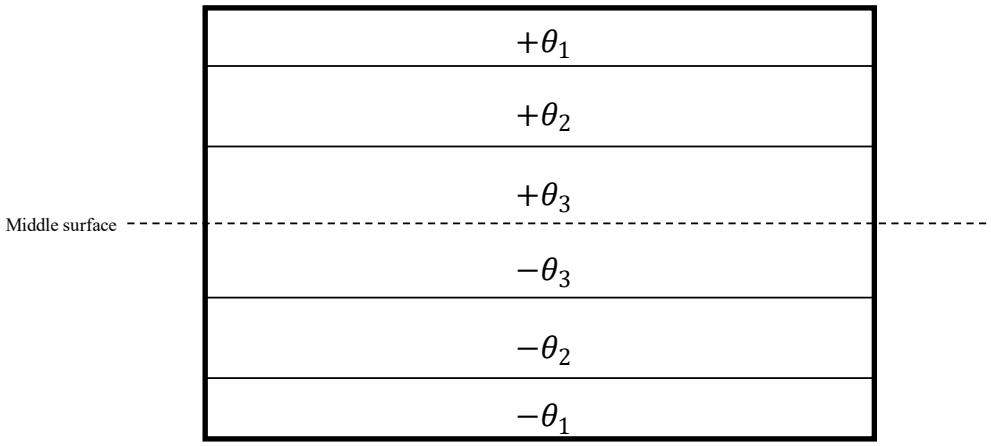


Figure 3-11 Antisymmetric layer configuration

This assumption reduces the number of decision variables for the ply angles in half since from the original six independent variables only three need to be defined for the six layered composite. $[\theta_1,\ \theta_2,\ \theta_3]$

This assumption is reasonable since most composites are either symmetric or anti symmetric in order to reduce the membrane - bending coupling effects.

- Each layer has the same thickness.

This assumption reduces the number of independent variables from six to just one, since only one thickness $[t]$ needs to be defined.

This assumption is also quite reasonable since during manufacturing of composite parts every layer originates from the same spool of carbon fiber which has an even thickness in its entirety.

These two assumptions lead to the final decision variables being:

$$\vec{\boldsymbol{x}} = [t, \theta_1, \theta_2, \theta_3]^T \tag{3.2}$$

Objective function:

Next the objective function is defined. Two different scenarios were considered for the objective function.

- Scenario 1: The objective function considers the mass as well as the flutter velocity. The concept behind the formulation of this objective function is to be able to minimize the mass of the structure while maintaining a sufficient flutter speed. This would normally fall under the category of multi-objective optimization, but Powell's method doesn't allow for that, so a work around is used. The penalty method has to be employed, thus resulting in the following formulation.

$$f_{obj}(\vec{\boldsymbol{x}}) = \begin{cases} M, & V_{flutter} > V_{limit} \\ M + P \cdot \left(V_{limit} - V_{flutter}\right), & V_{flutter} < V_{limit} \end{cases} \tag{3.3}$$

  Where:

  - M: is the mass
  - P: is a large constant called the penalty
  - V_{limit}: is the limit below which the flutter speed of the wing is deemed unacceptably low
  - $V_{flutter}$: is the resultant flutter speed from the Optistruct solver

  This definition allows for the minimization of mass while keeping the flutter speed above a certain limit. If the flutter speed drops below the specified limit a penalty term proportional to the amount by which the constraint is violated is added to the objective function so that the optimization algorithm is forced to return to a region where the constraint is not violated any more.

- Scenario 2: The simpler scenario is an objective function where the input is the decision variable vector from equation 3.2 but excluding the thickness t, and the output is the negated flutter velocity calculated by the Optistruct solver.

$$f_{obj}(\vec{\boldsymbol{x}}) = -V_{flutter}, \qquad with\ \vec{\boldsymbol{x}} = [\theta_1, \theta_2, \theta_3]^T \tag{3.4}$$

  The negative sign on the velocity is so that minimization occurs in the direction of increasing flutter velocity. Also notice that thickness has been removed from the optimization variables. Since mass is not considered in the objective function the

optimizer could simply increase the thickness of the material and achieve a higher flutter speed this way. This behavior is of course undesirable and is the reason why thickness was removed from the optimization variables and now remains constant throughout the optimization process along with the mass of the wing.

Search space boundaries

To fully define the optimization problem, the search space needs to be fully defined. The boundary definition is quite simple in this case.

- For Scenario 1:
  Only the angles need to be constrained between -90 and +90 degrees so the constraints are:

$$\theta_1, \theta_2, \theta_3 \in [-90^o, +90^o] \quad (3.5)$$

- For Scenario 2:
  The thickness has to be restrained in addition to the angles. The boundaries for the thickness are defined to be within a reasonable range of 0.1 to 1 mm per layer.

$$t \in [10^{-4}, 10^{-3}] \;\; and \;\; \theta_1, \theta_2, \theta_3 \in [-90^o, +90^o] \quad (3.6)$$

Acceleration of the algorithm

Because of the computationally intensive nature of the objective function a way to reduce computational time was applied. Instead of the algorithm being able to choose any arbitrary floating-point number within the specified range of each optimization variable a slight compromise was made. The values of each variable are internally rounded to a specified precision given by the user. Moreover, the results of each iteration are stored in a cache so that in case the algorithm finds itself trying to use the same input, the calculations are omitted and the cached results is used instead. The caching in combination with the rounding result in far fewer calls to the Optistruct solver than would be otherwise required. In this application the angles are rounded to the nearest integer while the thickness is rounded to the nearest tenth of a millimetre.

To illustrate this concept more clearly an example will be made:

Let's assume that the initial vector is $\vec{x}_0 = [0.0005,\ 45, -45, 45]^T$

1. For the first iteration let's assume that the algorithm tries
   $$\vec{x}_1 = [0.00043, -20.3, -69.6\ , 42.14]^T$$
   this will internally get rounded to
   $$\vec{x}_1 = [0.0004, \quad 20, -70, 42]^T$$
   and the result will be cached

2. If in the second iteration the algorithm tries
   $$\vec{x}_2 = [0.00041, -20.4, -69.8\ , 42.47]^T$$

it will still get rounded to

$$\vec{x}_2 = [0.0004, \quad 20, -70, 42]^T$$

and the same cached result will be used

The results of Powell's method are presented in chapter 4.3

3.4.2 Applying the Genetic Algorithm

For the application of the genetic algorithm the PyGAD [14] library is used. In order to define the optimization problem for the genetic algorithm many parameters need to be defined. Unfortunately, there is no way of finding the optimal settings for every variable in every specific problem. Therefore, the parameters were chosen after some experimentation using a smaller number of generations which can be run faster. The parameters that are selected are most probably not the most optimal but work well enough.

1. First and foremost, the **genes** and the **gene space** must be decided.

The genes are analogous to the optimization variables of classical optimization algorithms and are chosen to be the three angles and the thickness of the layers so four genes in total.

A range of possible values needs to be defined for every gene. The range for every angle gene is:

$$\vartheta_{RANGE} = [-90, +90, step = 1]$$

and for the thickness:

$$t_{RANGE} = [0.0001, 0.001, step = 0.0001]$$

these ranges define the gene space.

2. The so-called fitness function needs to be defined. The fitness function is analogous to the objective function of classical optimization algorithms. Because of the advanced abilities of this algorithm a multi-objective optimization is carried out where the two goals are:
   - The minimization of the mass of the ASW 28 Wing structure
   - The maximization of the Flutter Velocity.To achieve those goals a function is created with the following format

$$f_{fitness}\left(\begin{bmatrix} t \\ \vartheta_1 \\ \vartheta_2 \\ \vartheta_3 \end{bmatrix}\right) = \begin{bmatrix} -Mass \\ V_{flutter} \end{bmatrix} \tag{3.7}$$

   The negative sign for the mass is because this algorithm works on maximizing the fitness function.

After those basic and mandatory inputs are defined, several other parameters which control the way the algorithm runs are defined.

- *Num_generations = 1000, controls* the number of generations in the span of which evolution will take place. This parameter is chosen so that a reasonable computational time is maintained
- *sol_per_pop = 10,* the number of solutions that will be produced (number of chromosomes) after each generation
- *parent_selection_type = steady state selection*
- *keep_elitisism = 4* which means that the four best solution of each generation are carried over to the next
- *crossover_type = "single point"*
- *crossover_probability = 0.7* The probability of selecting a parent for applying the crossover operation. For each parent, a random value between 0.0 and 1.0 is generated. If this random value is less than or equal to the value assigned to the crossover_probability parameter, then the parent is selected.
- *mutation_type = "random"*
- *mutation_probability = 0.1* The probability of selecting a gene for applying the mutation operation. For each gene in a solution, a random value between 0.0 and 1.0 is generated. If this random value is less than or equal to the value assigned to the mutation_probability parameter, then the gene is selected
- *mutation_by_replacement = True,* means replace the gene by the randomly generated value instead of adding the random value to it.

The results of the Genetic Algorithm are presented in chapter 4.4

3.4.3 Flutter Speed Prediction using Neural networks

The purpose of this section is to develop a surrogate model using Neural Networks in order to potentially accelerate the process of optimization. Surrogate models are often used in engineering when the outcome of interest is not easily measured or when the computational effort required for proper simulation is too great. The development of the Neural Network models is done in python using the Keras library from Tensorflow.

The development of neural networks is a multistep process.

1. The training data for the models have to be collected and organized in a convenient way
2. The structure of the model and its parameters have to be defined
3. The model has to be trained on the training data
4. Lastly the performance of the network has to be evaluated

Training Data

To acquire the training data the solver is run repeatedly for random material parameters within specified ranges. The parameters that vary are the same as those used for the optimization.

More specifically those parameters are:

The angle of the major axis of each layer assuming an antisymmetric construction

$$\theta_i, \quad \in [-90, +90]\, deg, \quad for\ i = 1,2,3 \tag{3.8}$$

The thickness of each layer assuming all the layers have equal thickness

$$t, \quad \in [0.1, 0.9]\, mm \tag{3.9}$$

3000 random datapoints were collected and organized in a data frame where every row represents a data point and contains information about the input variables used and the predicted flutter speed from Optistruct which is the target variable of the Neural Network. The mass of the wing was also recorded but this is not a target variable for the neural networks that are about to be developed because the mass is directly proportional to the thickness of the layers. $Mass \propto t$. The computational time required to gather all the data is approximately 23 hours.

Model Structure & parameters

The structure of the layers is a very important aspect of neural networks. There is no way to know a priori the optimal structure of the neural network so that a good prediction is made, and a reasonable training time is achieved. For this reason, many different models will be created, and their performance will be evaluated in order to find the best one.

The first layer of every Neural Network is a normalization layer. This is an important step that ensures a consistent scale between the different data types of the input. In turn this ensures that no feature dominates the learning process. Furthermore, normalization can improve convergence by keeping the weights and activations within a reasonable range. Lastly normalization can help by reducing the overfitting and making a model more robust.

For this example, the two different kinds of inputs (the angles and the thickness) have very different ranges since the angles vary between -90 and +90 while the thicknesses are four orders of magnitude smaller varying between 0.0001 and 0.001 meters.

The normalization layer in Keras shifts and scales the input data into a distribution centered around 0 and with a standard deviation of 1 by subtracting the mean of the training data and the dividing by the square root of the variance in the data.

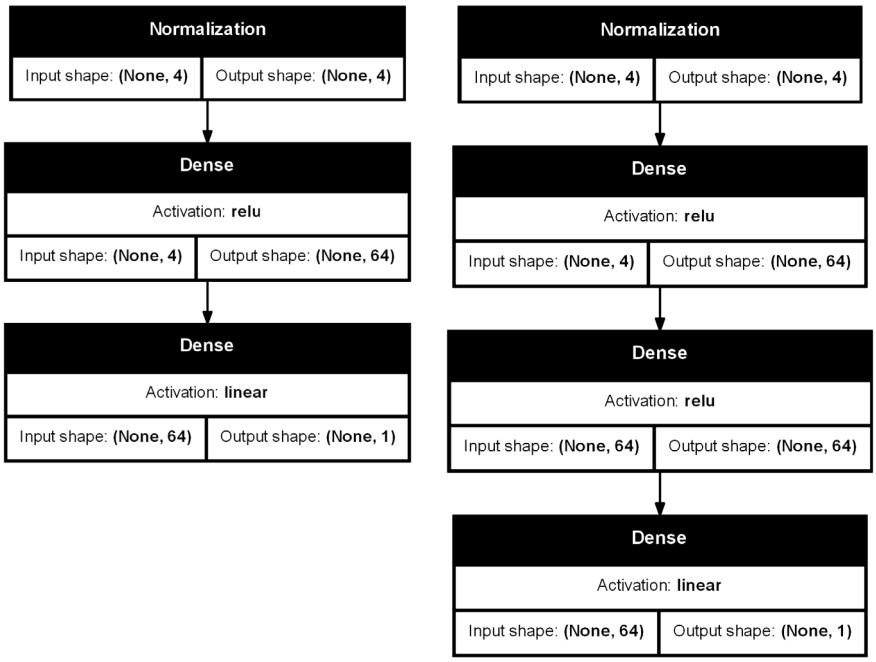
$$f_{norm}(x) = \frac{x - mean(\vec{x})}{\sqrt{var(\vec{x})}} \tag{3.10}$$

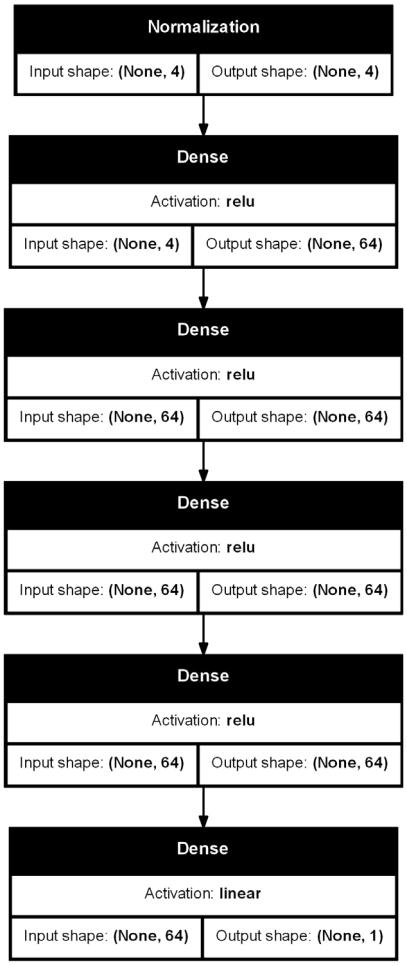
The mean and variation of the data is calculated based on the training data. If the training data is not representative of the actual data on which the network will be called to make predictions on, the normalization might fail producing a non-centered distribution with a variation much different than 1.

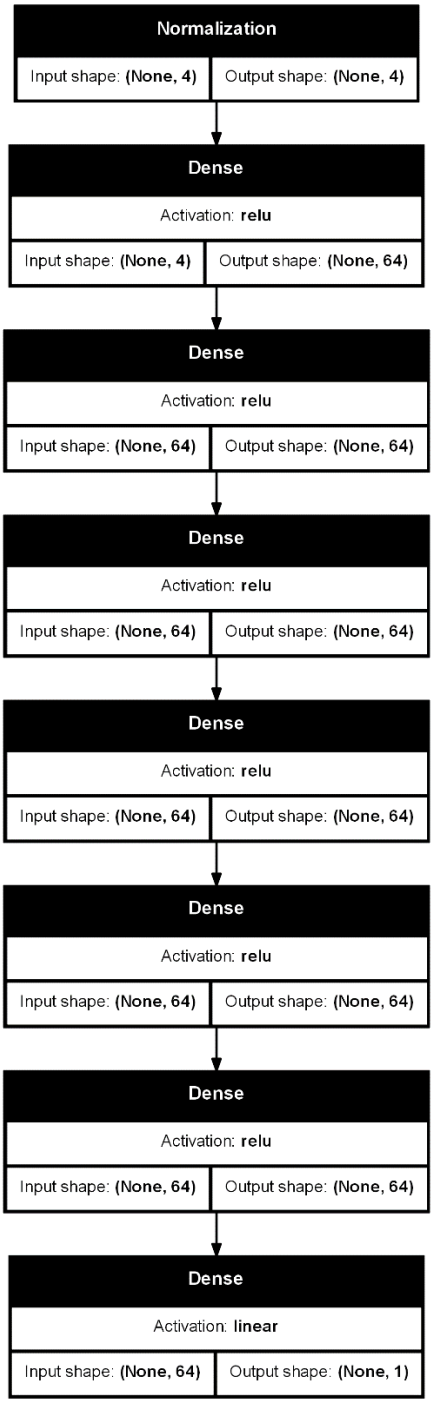
After the normalization layer the Hidden layer follow. These layers differ in number from model to model. In this work Neural Networks with 1, 2, 4, and 6 Hidden layers will be tested. Each layer has 64 Neurons and uses the RELU activation function.

Finaly the output layer is added which uses a linear activation function as this is a regression problem and only has 1 Neuron because the only prediction that these neural networks make is the flutter speed of the ASW 28 Wing model.

Table 5 Structure of Neural Networks with 1,2,4 & 6 Hidden Layers







Loss Function

The loss function of choice for this application is the Mean Average Error described in Equation 2.71 is used because it is easy to interpret as it has the same units as the output variable and is robust to outliers.

Optimizer

The optimizer of choice is Adam which is widely used and considered one of the best optimizers with a learning rate of 0.01

Hyper Parameter Tuning

Finally, a Hyper tuned model is constructed using the HyperBand Algorithm. The model is free to choose the following hyperparameters:

- The Number of Hidden layers between 1 and 10
- The number of Neurons for each layer from a discrete set of values $\{2^i\}\ with\ i = 1,2 \dots 10$
- The activation function of each layer with the choices being ReLu, tanh, Leaky ReLu
- The learning rate of the optimizer

The structure of this model is not known at this phase as it is a product of the optimization process. This is why it will be presented in the results section.

The results of the Neural networks are presented in chapter 4.5

4 Results

The Results chapter studies the dynamic flutter characteristics of the ASW28 Wing structure. Furthermore, the results of the optimization techniques that were implemented are presented here. Finally, the potential of Neural Networks to predict flutter instability is demonstrated.

4.1 Modal Analysis

To get a better understanding of the ways in which this Wing structure oscillates, a simple modal analysis is performed. The eigenmodes and eigenvectors found during this analysis are the eigenvectors and eigenvalues that would be found in a flutter analysis for zero airspeed. The first 6 modes of the Wing are shown in the figure below:

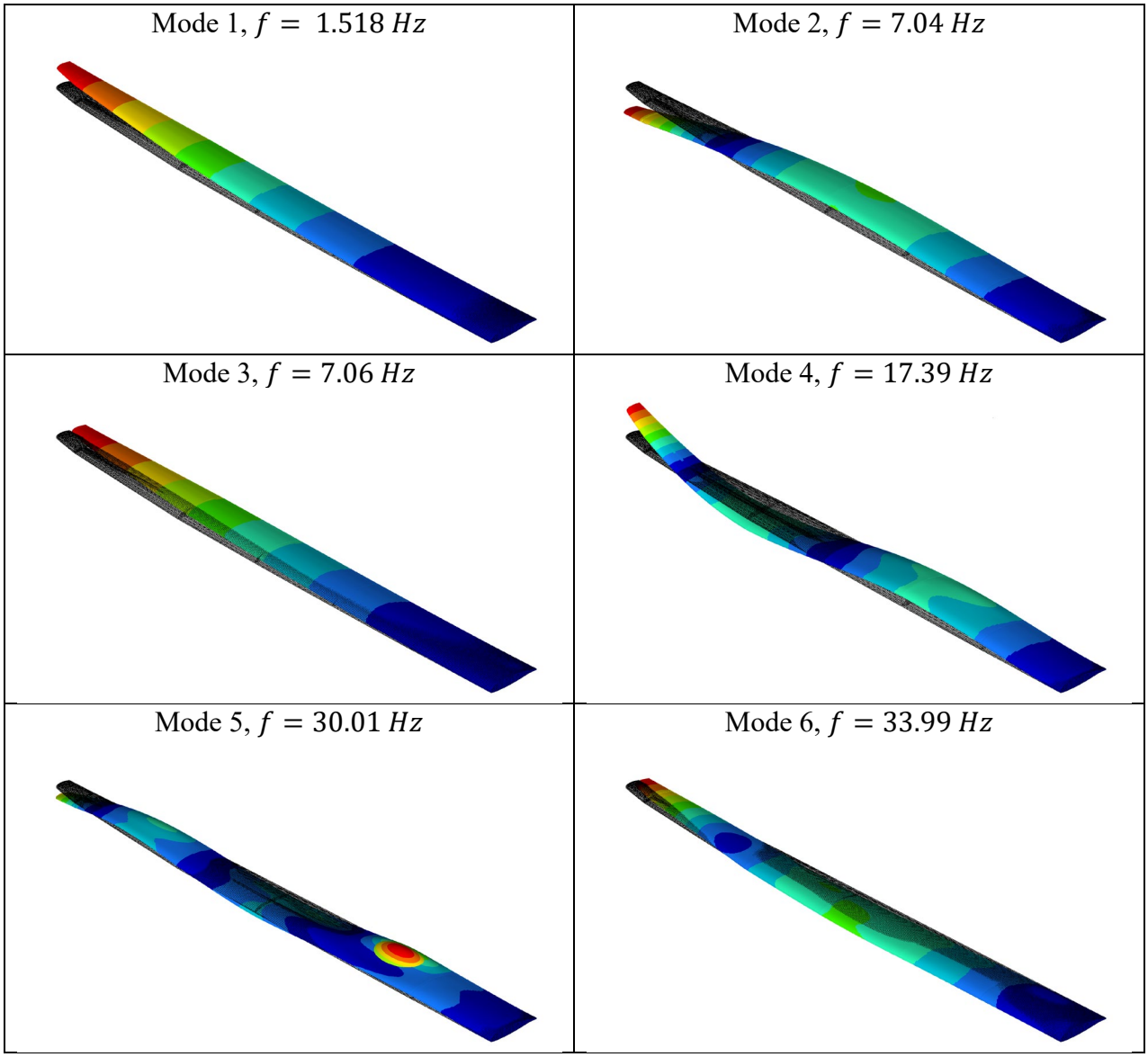


Figure 4-1First six Eigenmodes of ASW 28 Wing

From Figure 4-1 one can observe the following:

- The first eigenmode has a particularly low frequency indicating that the wing is quite flexible in that direction.
- Modes 1, 2, 4 and 5 are all bending modes about the x-axis with an increasing number of nodes (stationary points) on the Wing.
    - Mode 1 is the simplest with just one node at the rooth of the wing. As the frequency increases so does the complexity of the bending motion and the number of nodes.
    - Mode 2 has two nodes one at the root of the wing and one at about three quarters of the Wingspan
    - Mode 4 has three nodes one at the root of the wing, one in the middle and one approximately 5/6ths of the Wingspan
    - Mode 5 has four nodes spread across the Wingspan but also exhibits some signs of plate vibration on the upper surface of the Wing close to the root, where the ribs are farthest apart and there exists a large section of Wing skin which is unsupported.
- Modes 3 and 6 are bending about the z-axis. Mode 3 has only one node while mode 6 has two.

4.2 Initial Flutter Analysis

By applying the methodology developed in chapter 3.2 to the model the following results are obtained:

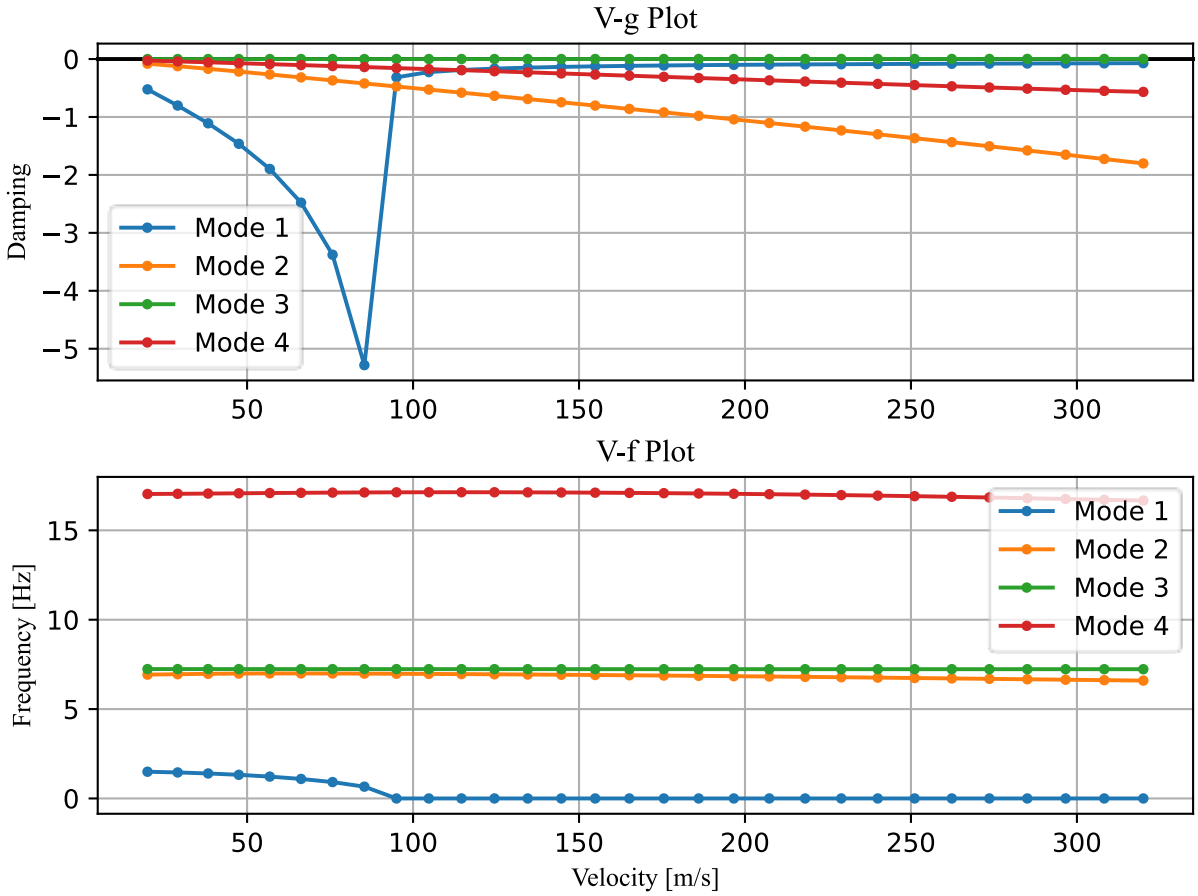


Figure 4-2 Initial Flutter plot for the first four modes.

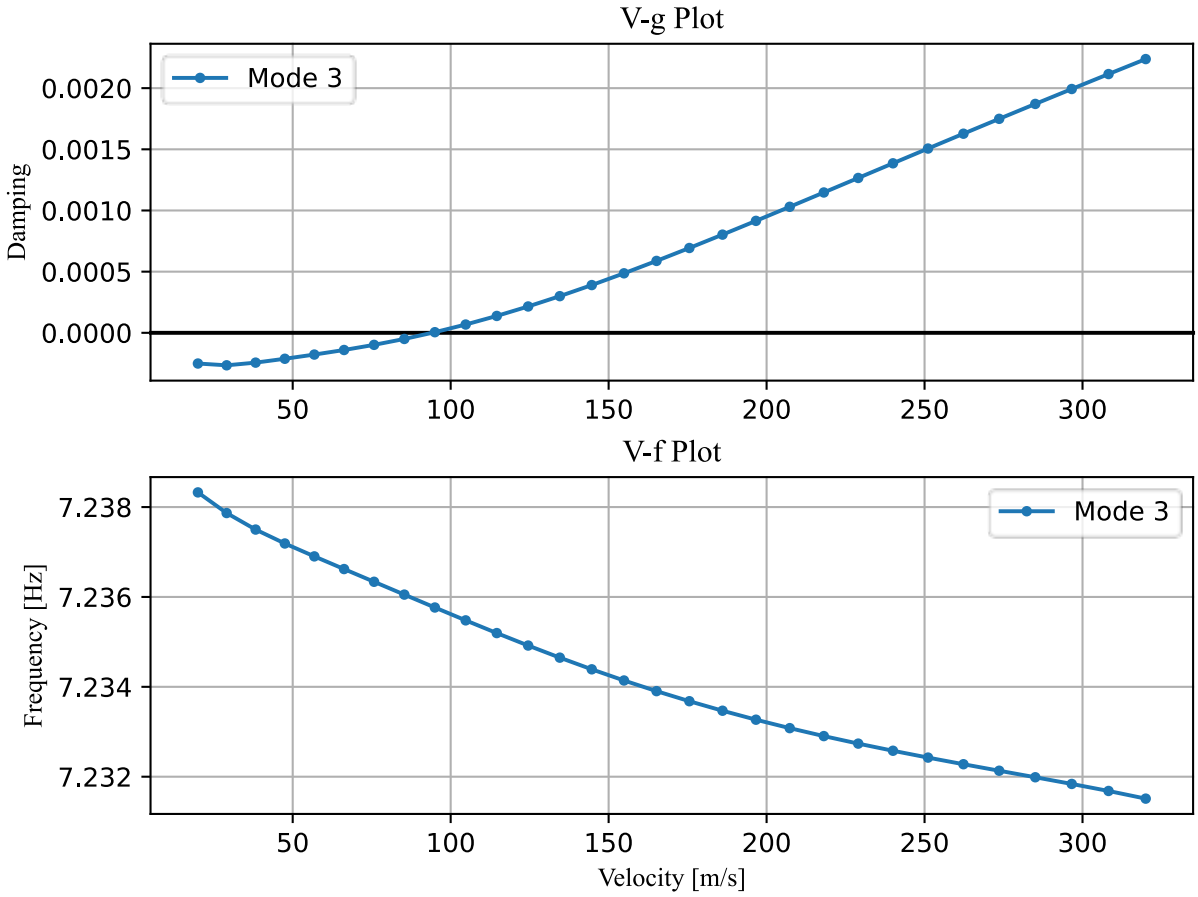


Figure 4-3 Flutter plot of the 3rd mode

From Figure 4-2 and Figure 4-3 Flutter occurs on the 3rd mode at:

$$V_{flutter} = 94.11\ m/s \tag{4.1}$$

Notice that Mode 1 is very likely to also diverge at about the same velocity even though its damping does not change sign. The abrupt increase of the damping to a value close to zero (but still negative) along with the reduction of the frequency to almost zero indicate some form of instability which is not captured by the algorithm since the sign hasn't technically changed.

4.3 Powell's Optimization Method

After applying Powell's Optimization method as described in chapter 3.4.1 the following results are obtained

Scenario 1:

This was the first objective function which was formulated in equation 3.3 as it defines a more complete optimization problem. The optimization concluded with partial success. The results can be seen below:

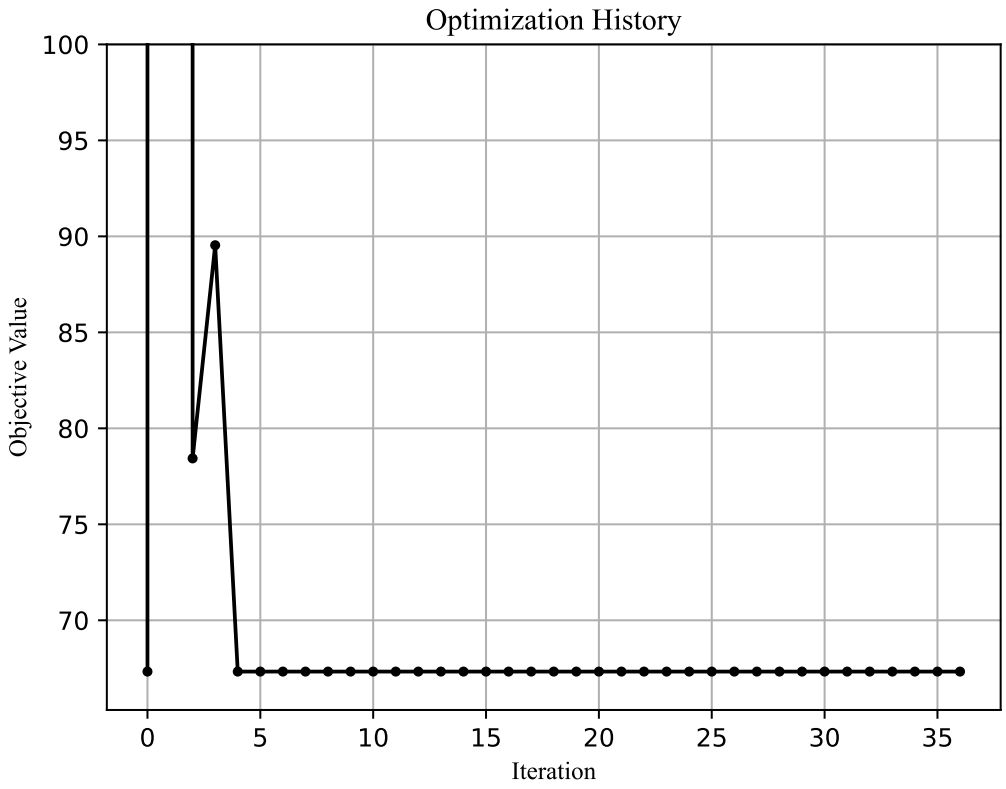


Figure 4-4 Value of objective function throughout the optimization (Scenario 1)

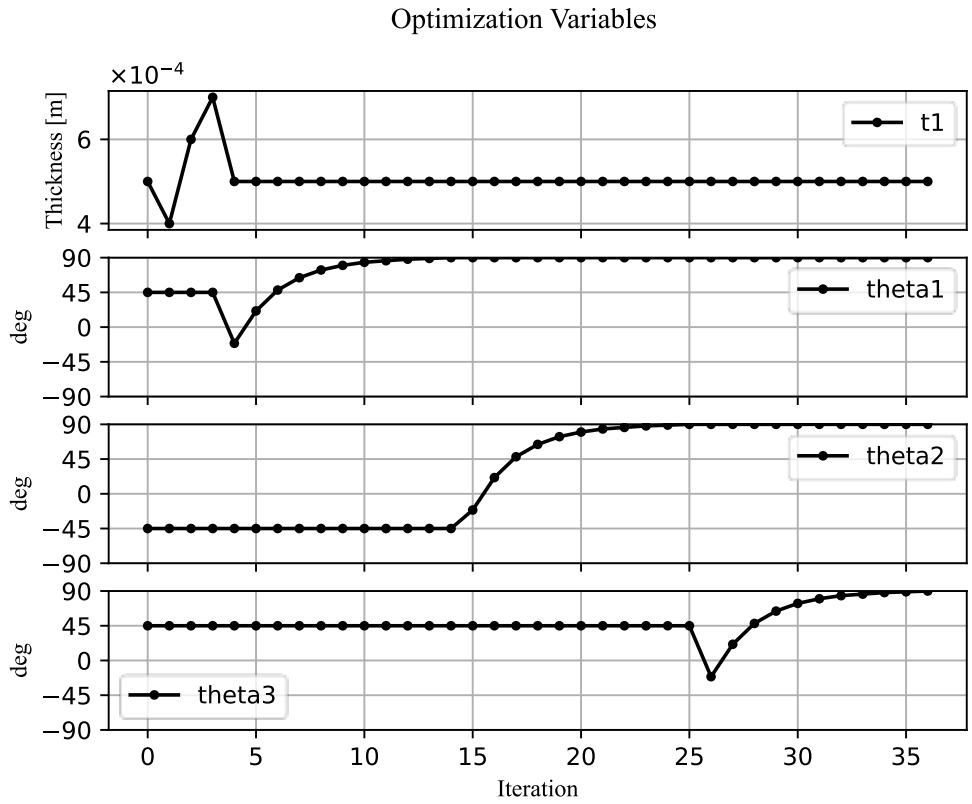


Figure 4-5 value of every optimization variable throughout the optimization process (Scenario 1)

As can be seen the algorithm fails to improve the objective function beyond the initial point. On the first iteration the algorithm tries to reduce the mass of the wing by reducing the thickness of the layers by a tenth of a millimeter, but the flutter speed decreases too much and the penalty is applied resulting in a very high objective function value.

After that point the algorithm experiments with greater values of thickness which of course result in greater mass but sufficiently high flutter speed for the penalty to not apply.

Finally, the algorithm reduces the thickness of the layers incrementally until it reaches the initial thickness.

The algorithm never figures out that by changing the ply angles first so that the flutter speed increases and then reducing the thickness the flutter speed remains high enough for the penalty to not ally and the mass is reduced. This is a complicated solution since the angles do not directly affect the mass but only the flutter speed. Had the algorithm explored the angles better while it was in the penalty zone it could have increased the flutter speed sufficiently to get out of the penalty zone and ultimately find a better solution. Unfortunately, this behavior is quite complex and is beyond the capabilities of this relatively simple algorithm.

The Flutter analysis results of the final solution of this algorithm are shown below:

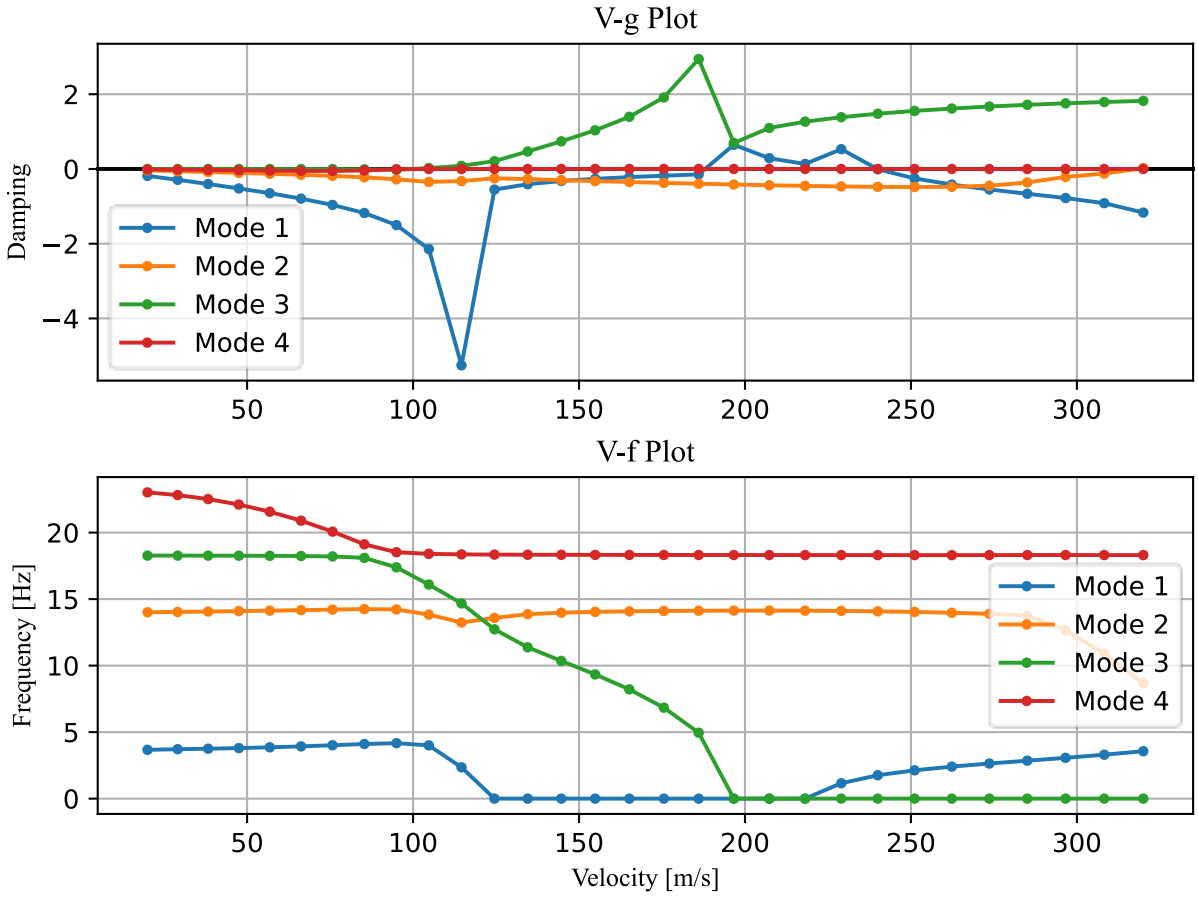


Figure 4-6 Flutter analysis plots from Powell's method (Scenario 1)

The first mode that diverges is mode three which is plotted on its own in the following figure to be able to clearly see when the damping changes sign.

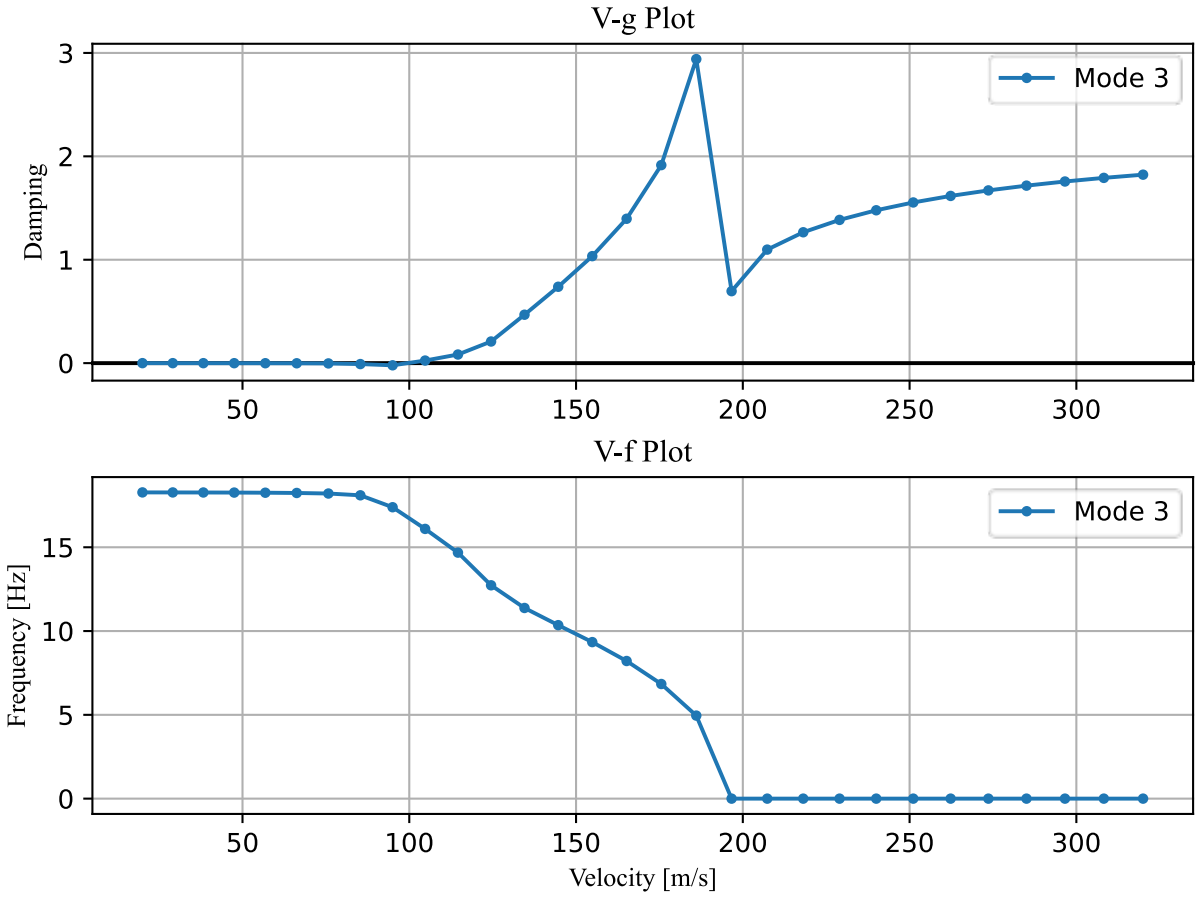


Figure 4-7 Mode 3 Flutter analysis Powell's method (Scenario 1)

The speed at which this mode first diverges is called the flutter speed and in this case:

$$V_{flutter} = 99.48\, m/s \tag{4.2}$$

Scenario 2:

To obtain a better result the simplified objective function from equation 3.4 is used. This time the algorithm explores the possibility-space with greater success. The results can be seen below:

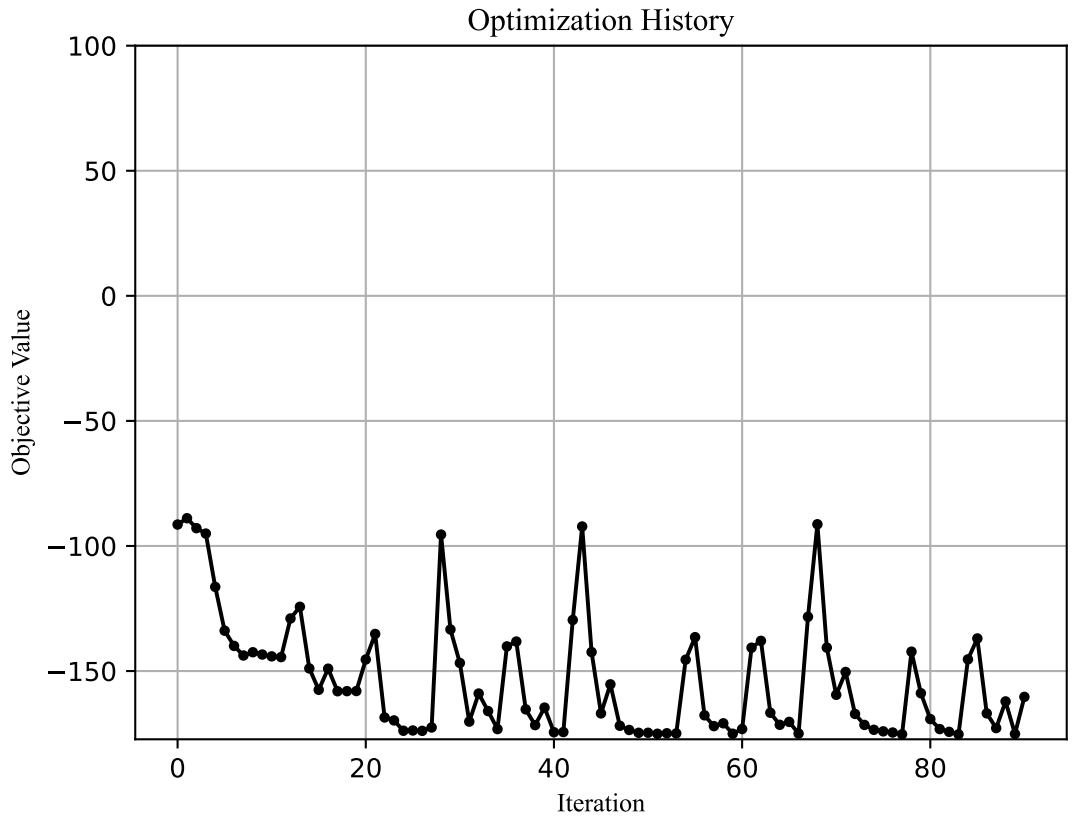


Figure 4-8 Value of objective function throughout the optimization (Scenario 2)

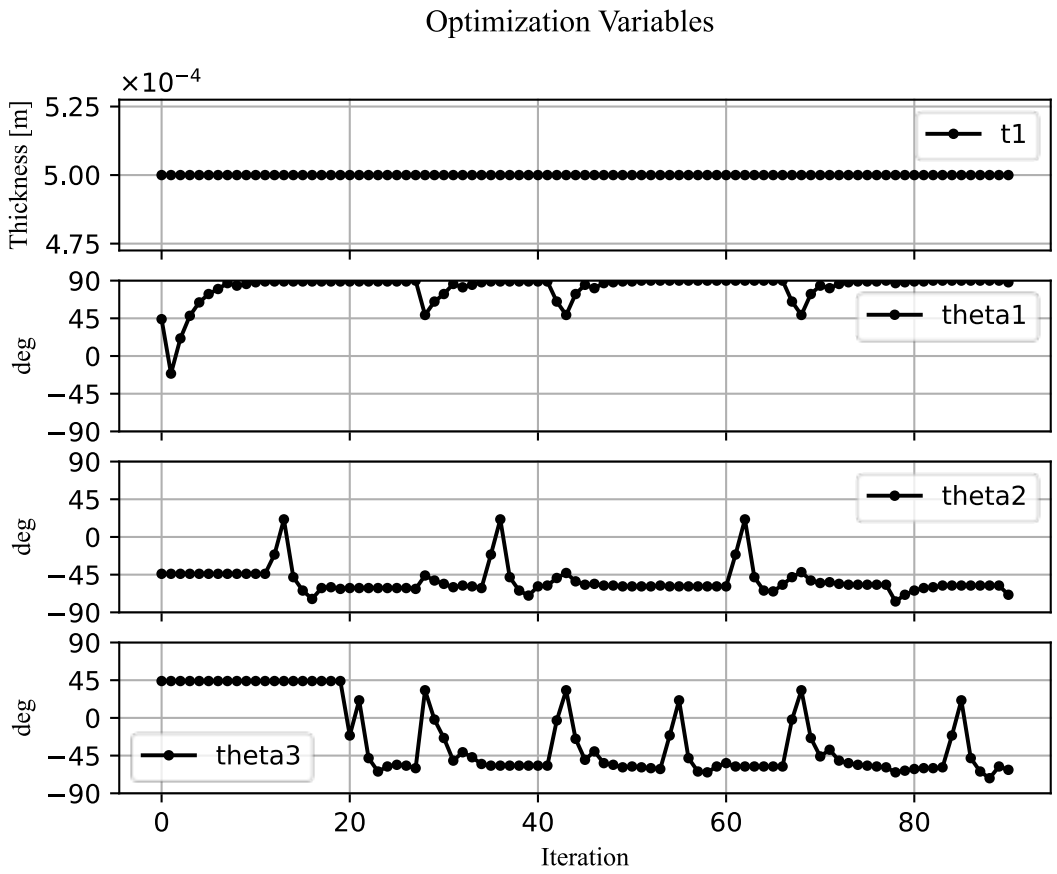


Figure 4-9 value of every optimization variable throughout the optimization process (Scenario 2)

One can observe from Figure that the objective function, which is now the minimization of the negated flutter speed, is significantly improved.

Moreover, as per the definition of this problem the thickness variable remains constant and equal to the initial value of 0.5 mm. This means that while the flutter speed increases significantly the mass remains constant.

The flutter plots for the optimal solution are now presented for this method.

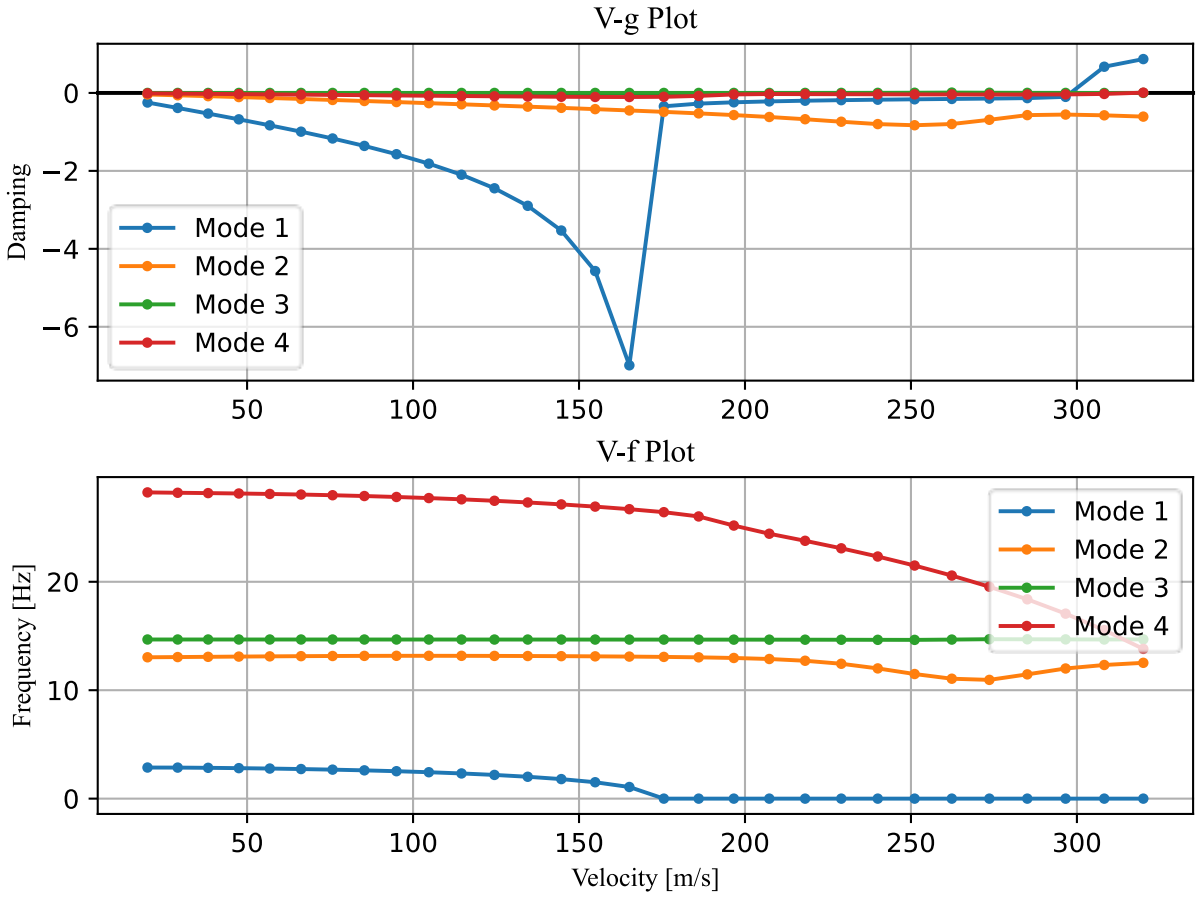


Figure 4-10 Flutter analysis plots from Powell's method (Scenario 2)

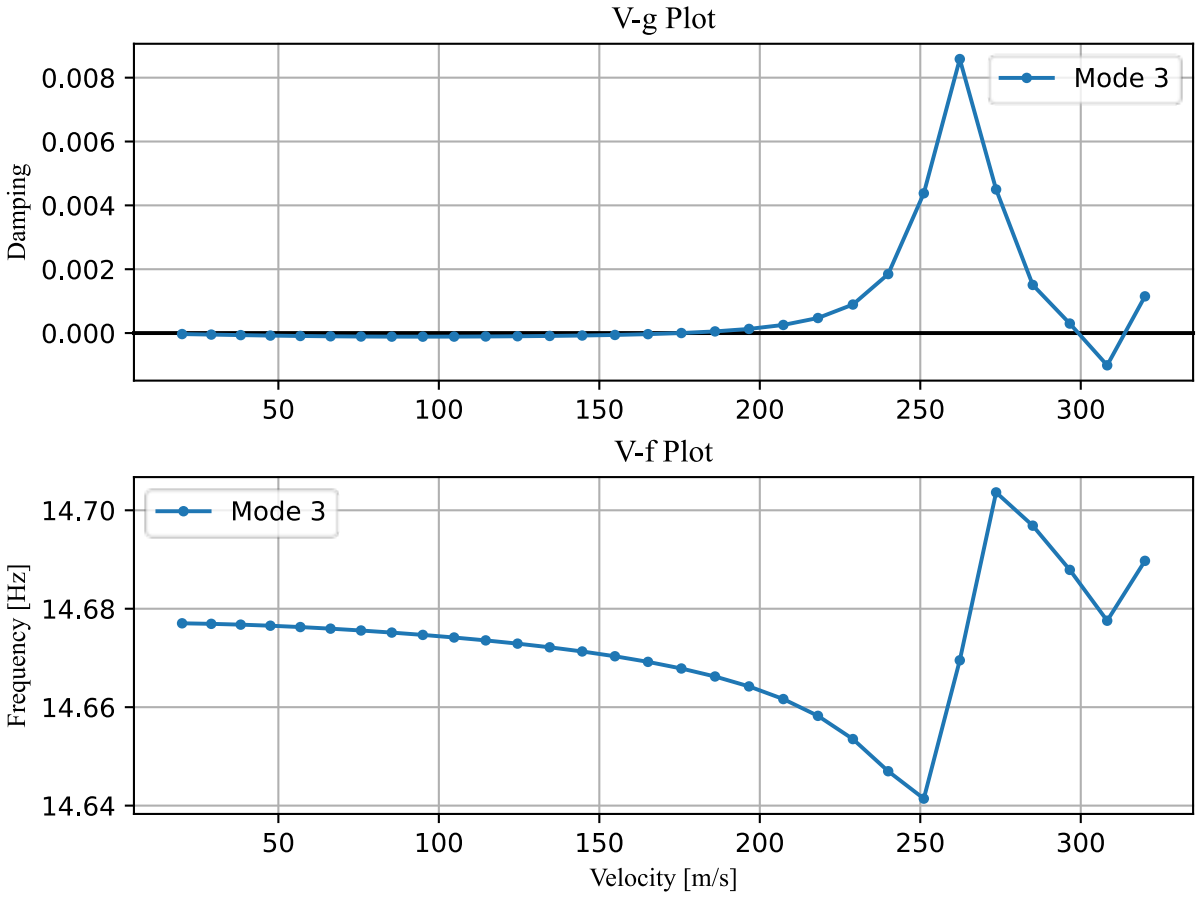


Figure 4-11 Mode 3 Flutter analysis plots from Powell's method (Scenario 2)

Again, the first mode that diverges is mode 3. This time though at a much greater speed than before.

$$V_{flutter} = 175.28\, m/s \tag{4.3}$$

Modes 1 one and four also eventually diverge at 267.09 and 177.14 m/s respectively. As is obvious this is a great improvement of the flutter speed of the Wing from the initial solution, without any extra material and thus the same mass as the original wing.

The solution vector that gives the optimum solution is:

$$\vec{\boldsymbol{x}}_{opt} = [0.0005{,}90, -58, -59]^T \tag{4.4}$$

4.4 Genetic Algorithm Optimization

After applying the genetic algorithm as described in chapter 3.4.2 the following results are obtained:

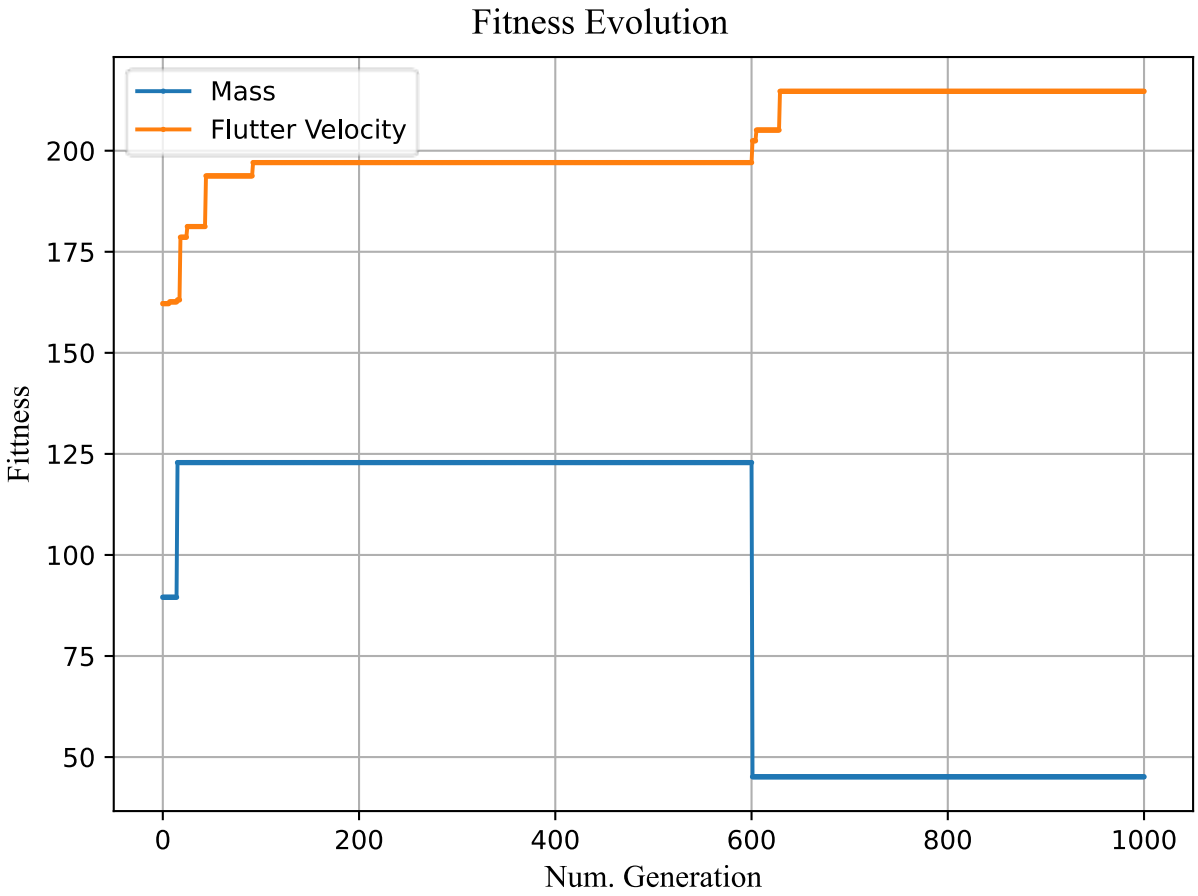


Figure 4-12 Evolution of the Fitness metrics of the best solution of every generation in the optimization process

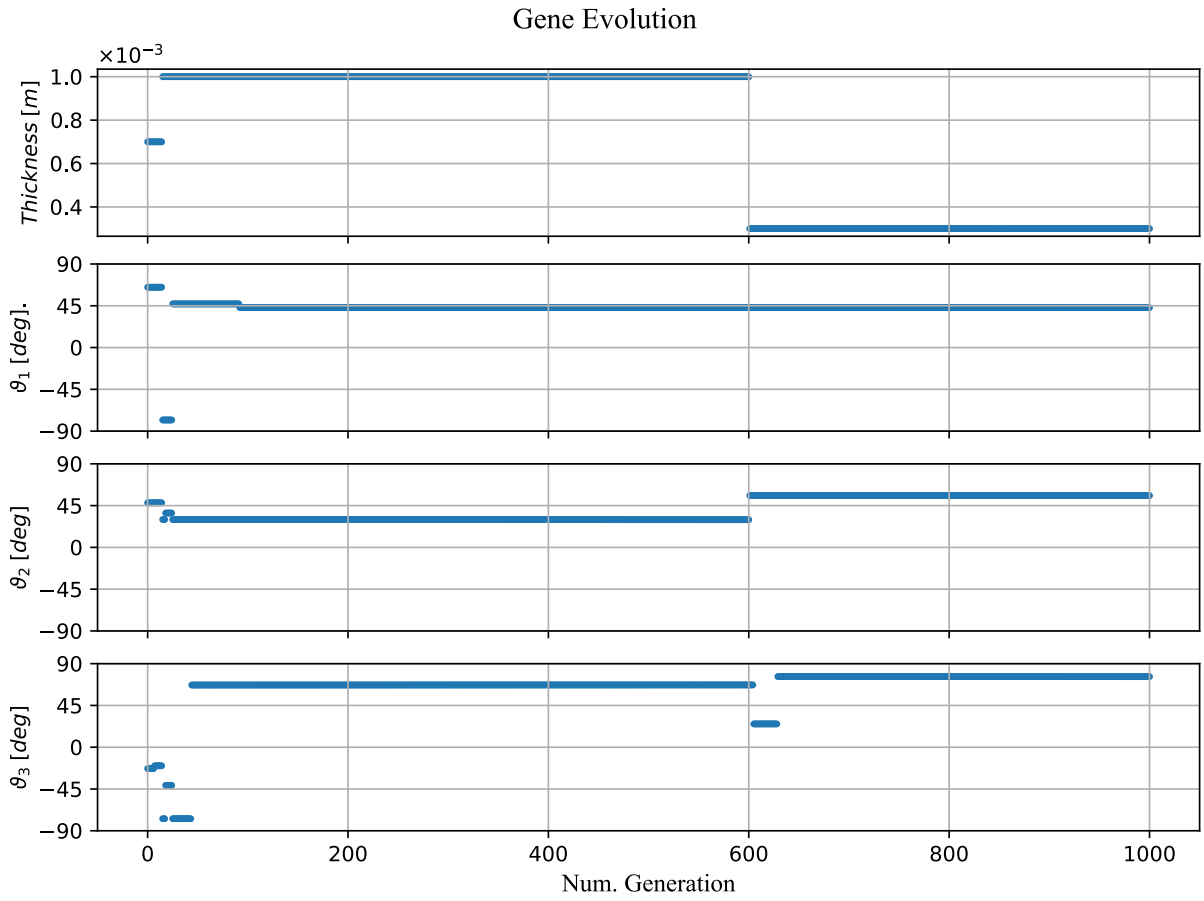


Figure 4-13 Evolution of gene Values throughout the optimization process

From Figure 4-12 where the best solution found thus far at every generation is plotted it is observed that both goals are improved by the end of the optimization process, even though those goals are contradictory to one another. From the optimization logs the best solution has the following characteristics.

$$V_{flutter} = 214.69\, m/s, \qquad Mass = 45.1\, kg \tag{4.5}$$

The gene values which result in the optimal solution are

$$t = 0.0003\, mm, \qquad \vartheta_1 = 43°, \qquad \vartheta_2 = 56°, \qquad \vartheta_3 = 76° \tag{4.6}$$

The flutter plot for this solution is presented below:

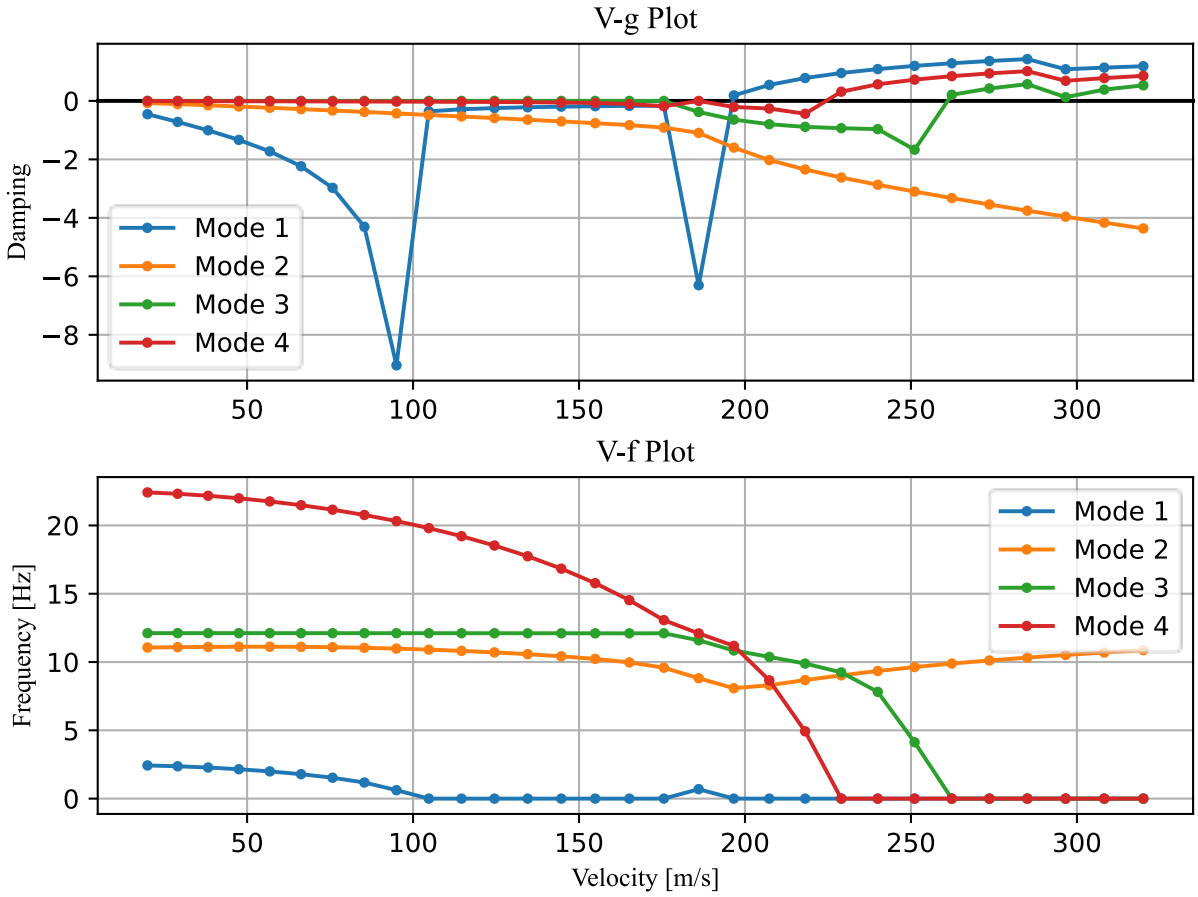


Figure 4-14 Flutter Plot from Genetic Algorithm

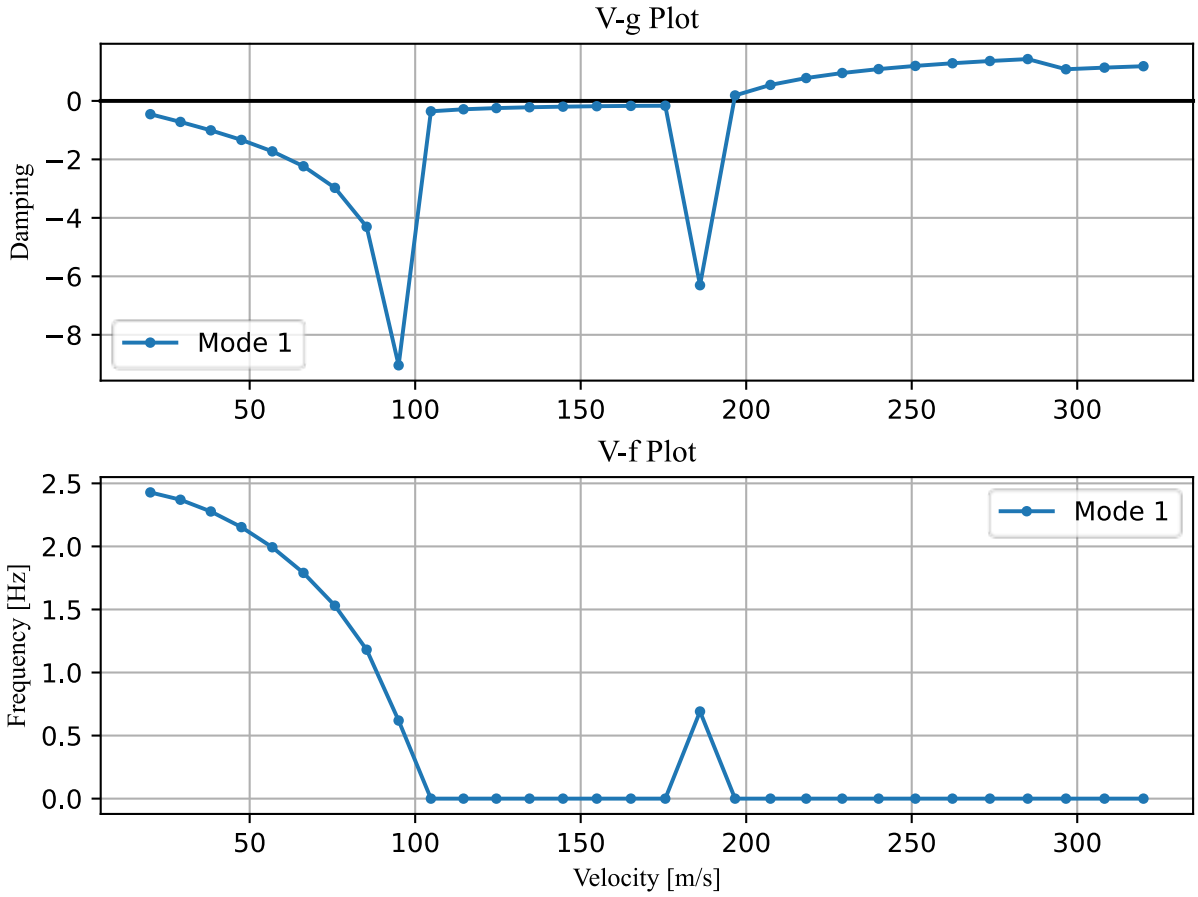


Figure 4-15Mode 1Flutter Plot from Genetic Algorithm

Even though the solution obtained by the genetic algorithm appears ideal by examining Figure 4-15 we can see that instability likely occurs earlier than the stated 214 m/s as the damping of this mode changes dramatically at 110 m/s and the frequency drops to zero which is a sign of flutter. Even if the damping remains negative in this region small changes to the modal characteristics to the wing or even modelling error could lead the real structure to flutter. This is a marginally acceptable solution that would have to be investigated further to ensure its validity

4.5 Neural Network Prediction Results

In this section the results of every Neural Network developed in section 3.4.3 will be reviewed. Three basic plots are presented to understand the performance of each network.

1. The first plot is the value of the loss function on the training and validation data at each epoch of training.
2. The second plot is used to judge the performance of the network by plotting the $y_{pred}\ vs\ y_{true}$ scatter plot. Ideally these values would match thus producing a straight line at a 45° angle.
3. The third plot plots the residuals with respect to y true $(y_{true} - y_{pred})\ vs\ y_{true}$ which makes it easier to understand the magnitude of the error as well as any trends that might exist at particular flutter speeds.

4.5.1 Training data examination

A preliminary exploration of the data is done using a pair plot, which is a matrix of graphs the enables the visualization of the relationship between each pair of variables in the dataset.

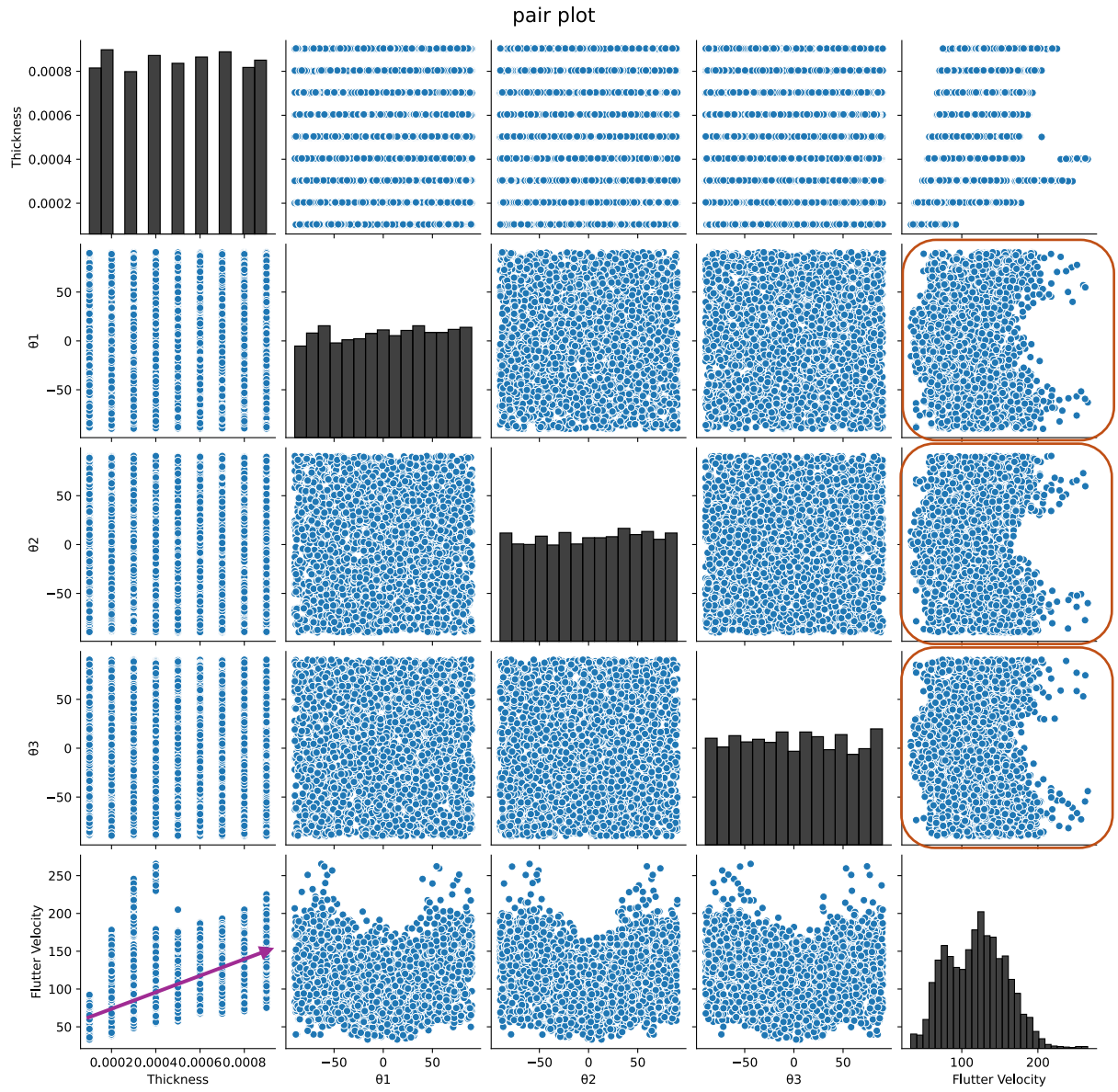


Figure 4-16 Pair Plot of training data

From Figure 4-16 we can observe the following relationships in the gathered data:

1. As thickness increases so does the predicted flutter speed.
2. The angles $\theta_1 - \theta_3$ have a strongly nonlinear correlation with flutter speed with no obvious trends
3. Along the diagonal histograms which represent the correlation of a variable with itself it can be observed that the data is randomly and evenly distributed across the range of possible values.

4.5.2 1 Hidden Layer Neural Network

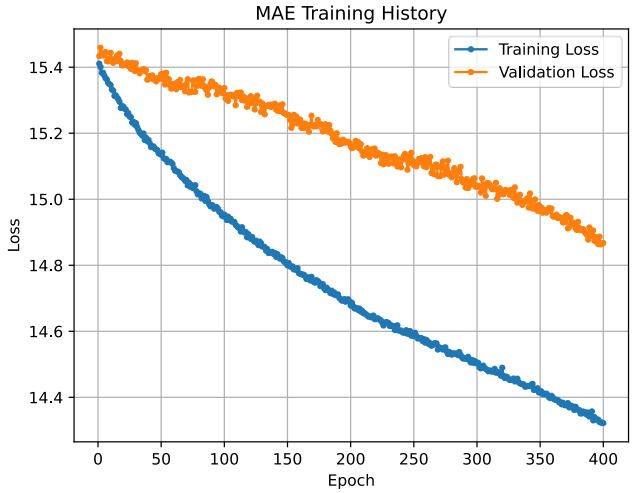


Figure 4-17 1 Hidden Layer NN Loss. Test and Training MAE vs Epochs

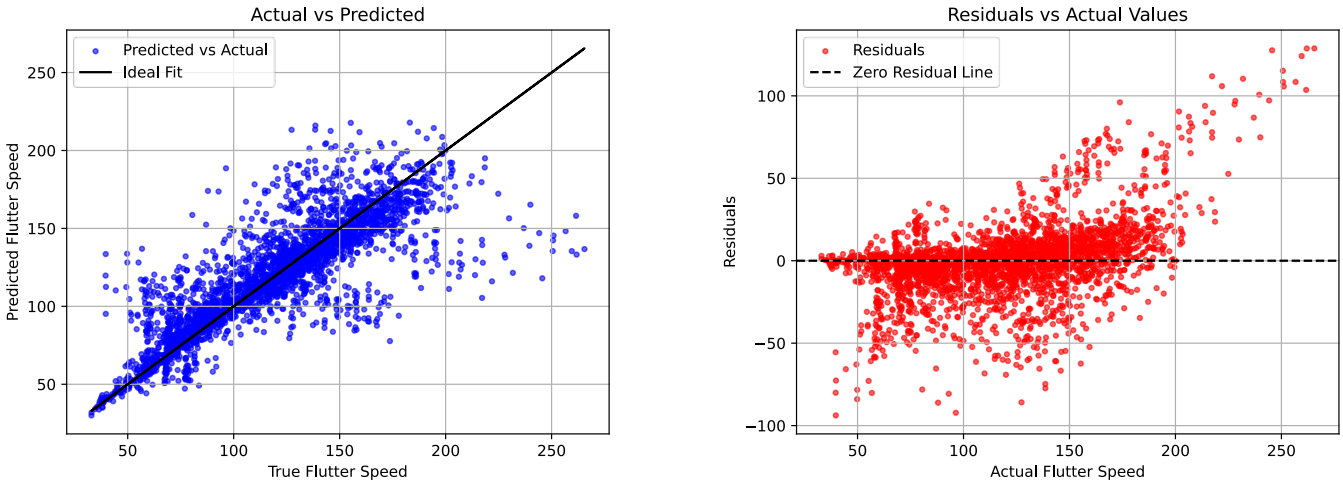


Figure 4-18 1Hidden Layer NN Performance. left: y_{true} vs y_{pred} , right: y_{true} vs $(y_{true} - y_{pred})$

From Figure 4-17 and Figure 4-18a it becomes apparent that even though this model with only one hidden layer was given 400 epochs to train which is a lot as will become evident by the following results, the validation data Mean Average Error is too great for the network to be of any use. The variance above and below the zero line in in Figure 4-18b is way too great with many points being over or underestimated by more than 50 m/s

4.5.3 2 Hidden Layer Neural Network

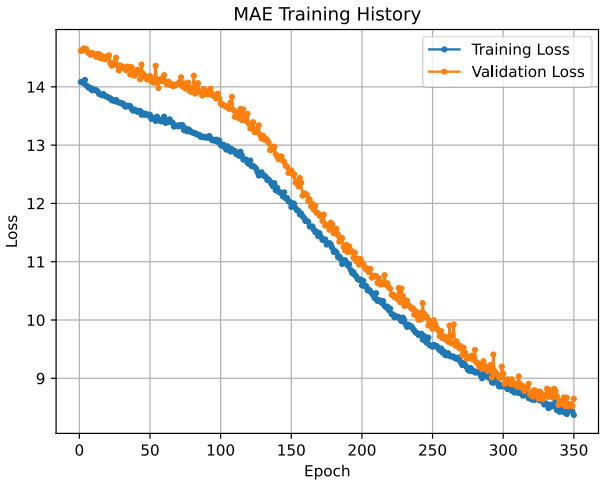


Figure 4-19 2Hidden Layer NN Loss. Test and Training MAE vs Epochs

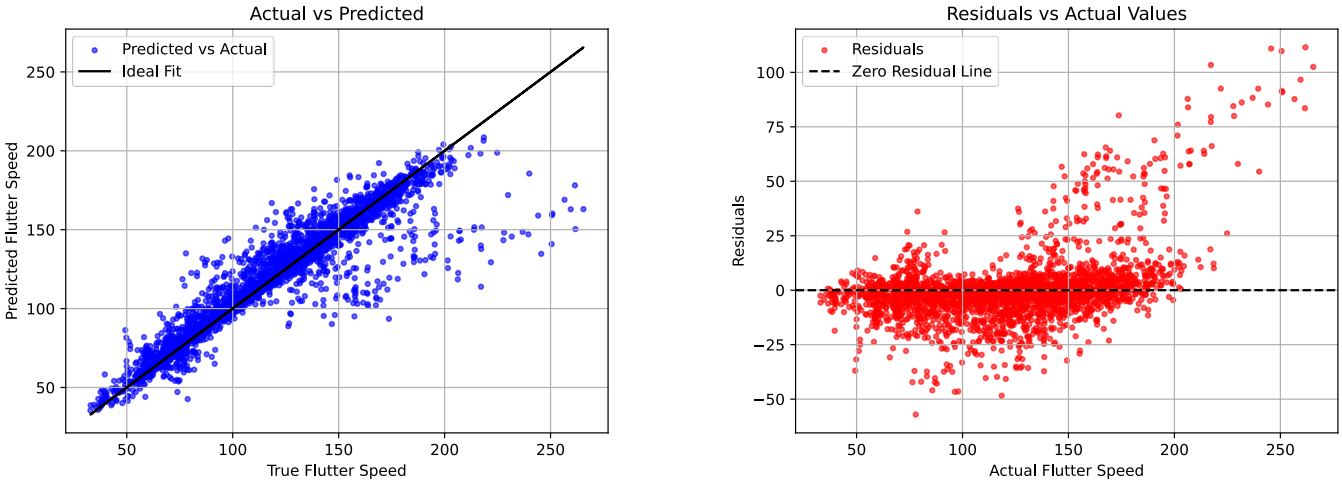


Figure 4-20 2 Hidden Layer NN Performance left: y_{true} vs y_{pred} , right: y_{true} vs $(y_{true} - y_{pred})$

This model was trained for 350 epochs. 2 Hidden layers perform much better than a single layer. The mean average error on the validation data by the end of the training is about 7 m/s. The only problem is that there is still great variance in the predictions as can be seen from Figure 4-20b.

4.5.4 4 Hidden Layer Neural Network

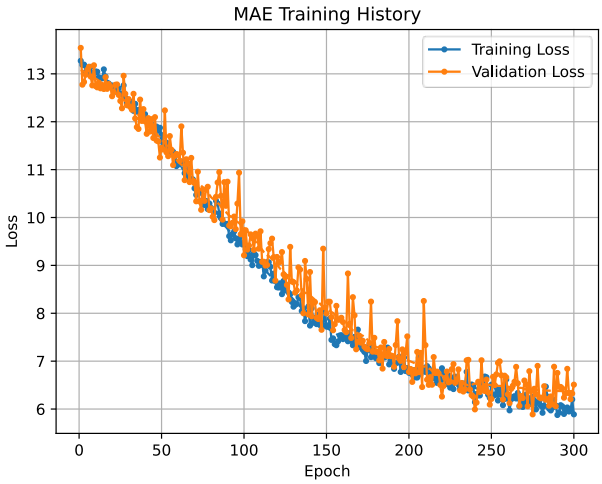


Figure 4-21 4 Hidden Layer NN Loss. Test and Training MAE vs Epochs

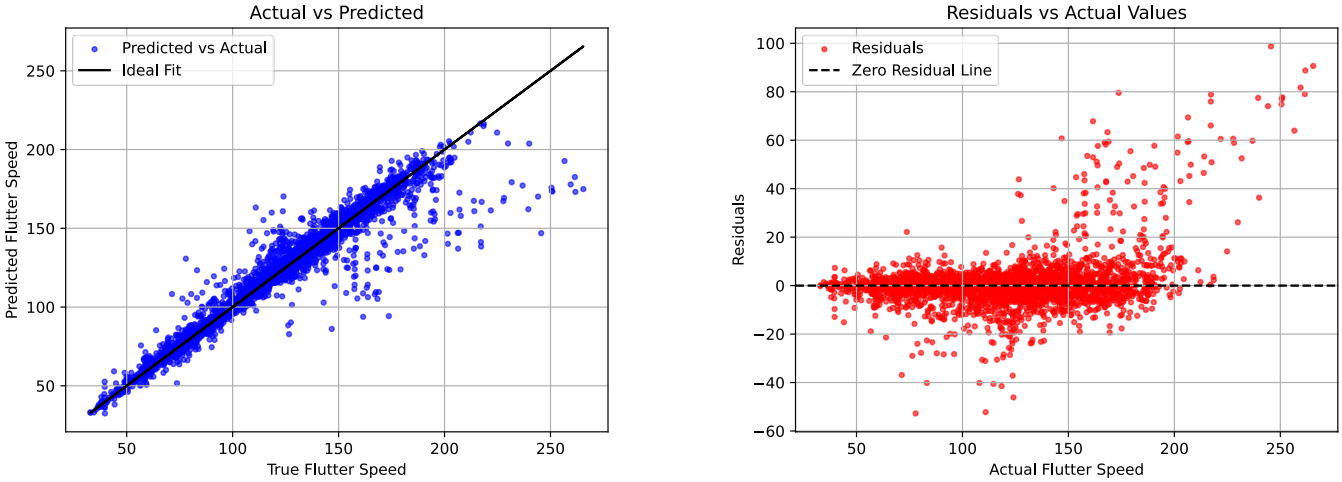


Figure 4-22 4 Hidden Layer Performance left: y_{true} vs y_{pred} , right: y_{true} vs $(y_{true} - y_{pred})$

The four hidden layer neural network was trained for 300 epochs because the validation loss starts to flatten out as can be seen from Figure 4-21. The spread of data around the zero residual line is not much tighter than the previous model.

4.5.5 6 Hidden Layer Neural Network

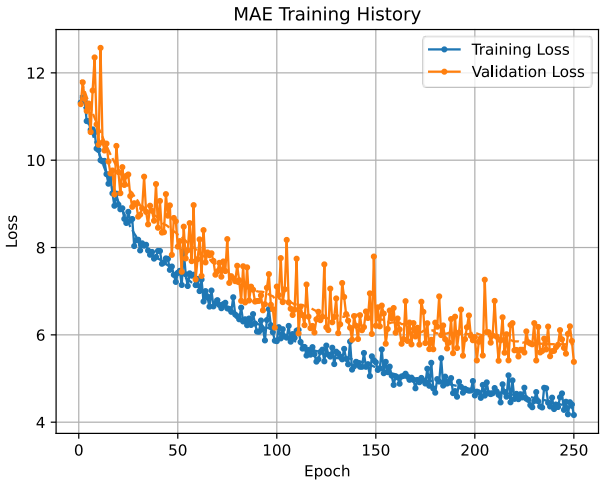


Figure 4-23 6 Hidden Layer NN Loss. Test and Training MAE vs Epochs

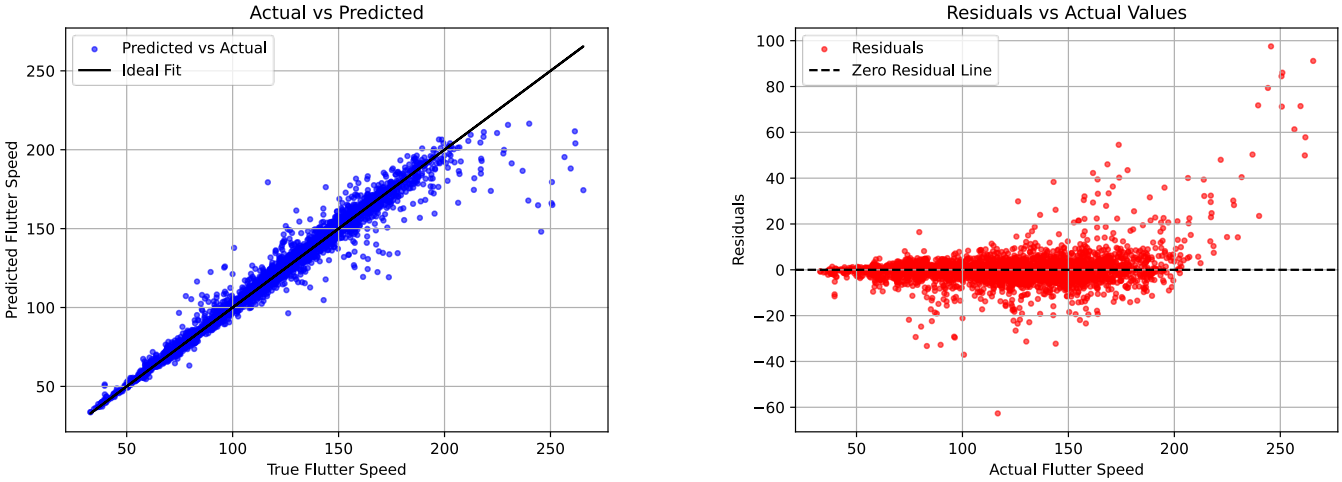


Figure 4-24 6 Hidden Layer Performance left: y_{true} vs y_{pred} , right: y_{true} vs $(y_{true} - y_{pred})$

The 6 Hidden layer neural network was trained for 250 epochs to prevent overfitting as the validation loss is stagnant after epoch 220. This can be seen in Figure 4-23

The spread of the residuals is better but still high but getting better in relation to the previous model as can be seen in Figure 4-24b.

4.5.6 Hyperparameter tuned Neural Network

The hyperparameters that were chosen from the HyperBand algorithm which was described in section 2.4.4 are as follows:

- Number of hidden layers: 5
- Number of Neurons for each layer: $1024,\ 512,\ 256,\ 512, and\ 1024$ respectively
- Activation function for every layer: ReLu
- Learning Rate of Adam Optimizer: 0.001

The Resulting structure of the model can be seen in .

Figure 4-25.

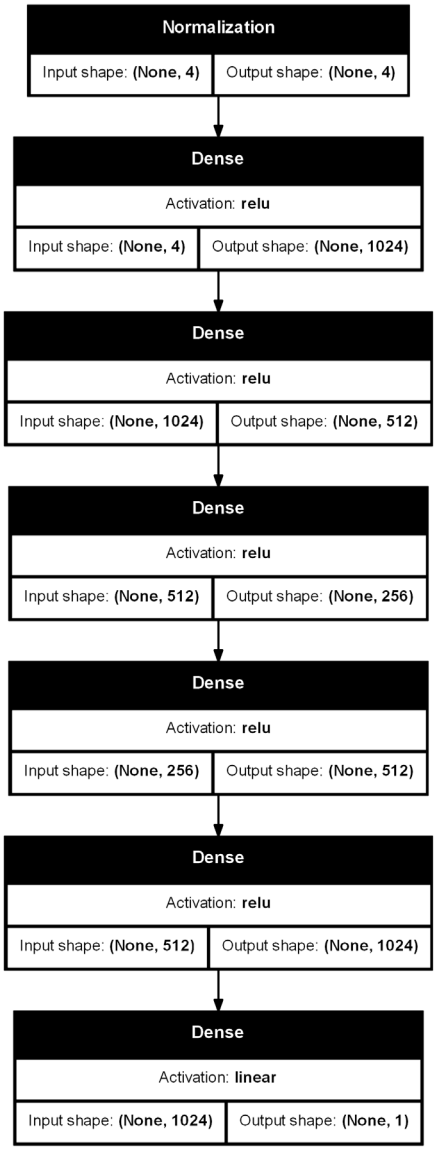


Figure 4-25 Hyper-tuned model structure

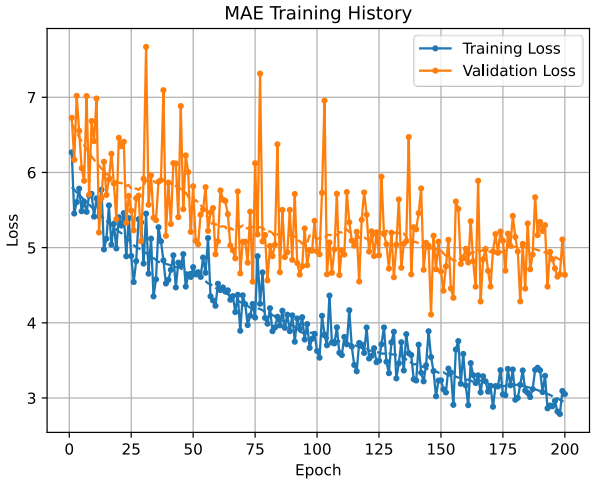


Figure 4-26 Hyper Tuned NN Loss. Test and Training MAE vs Epochs

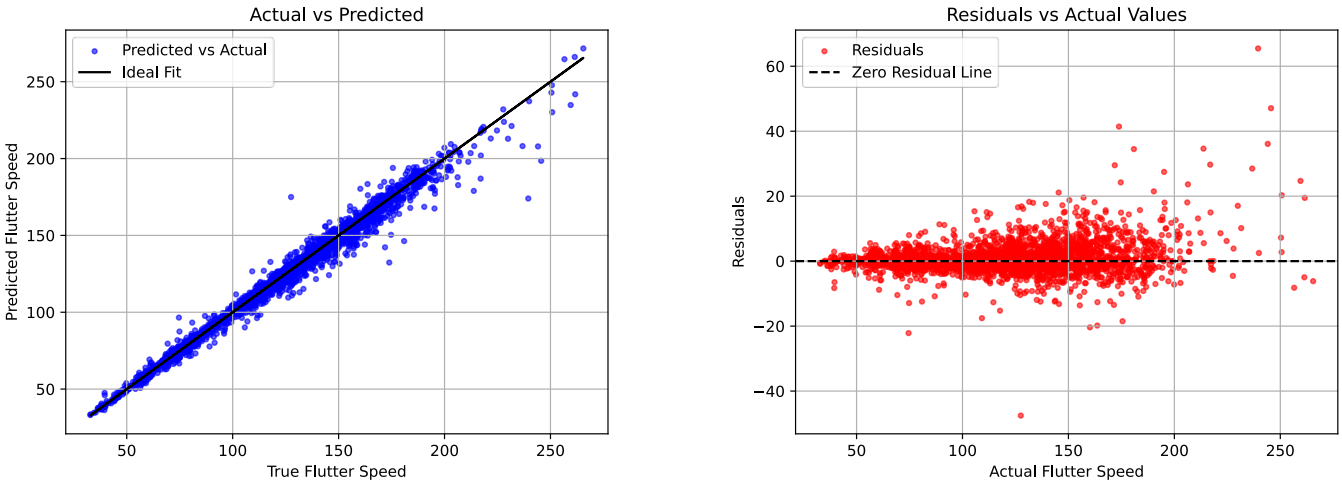


Figure 4-27 Hyperparameter Tuned NN Performance left: y_{true} vs y_{pred} , right: y_{true} vs $(y_{true} - y_{pred})$

As can be seen from Figure 4-26 this model was trained for 200 epochs because the validation MAE stop decreasing after that. This model exhibits the lowest MAE of every other model with a validation $MAE < 5$.

What is more the distribution of data around the Zero Residual Line in Figure 4-27, is the tightest of all the models thus producing the best results yet. It is also worth mentioning that this model has the highest number of trainable parameters and takes the longest to train.

5 Conclusions & Future Work

5.1 Optimization

A summary of the results produced in this work is now made and general conclusions are drawn.

Below is a summary of the results obtained with every different optimization method used in this work, as well as a table with the percentage difference from the initial flutter solution.

Table 6 Summary of Optimization methods' results

Method	Flutter Velocity [m/s]	Structural Mass [kg]	Divergent mode
Initial flutter characteristics	94.11	67.33	3
Powell's Method Scenario 1	99.48	67.33	3
Powell's Method Scenario2	175.28	67.33	3
Genetic Algorithm	214.69	45.1	1

Table 7 Comparison of different optimization methods

Method	Flutter Velocity Change	Structural Mass Change
Powell's Method Scenario 1	5.71%	0.00%
Powell's Method Scenario2	86.25%	0.00%
Genetic Algorithm	128.13%	-33.02%

What is clear from the results summary is that there is a lot of room for improvement from the initial solution. The effect of the ply angles is considerable and something that should be considered when designing a composite laminate aircraft wing. This kind of optimization is worth executing because it can potentially reduce the structural mass of the aircraft which results in enhanced flight performance, handling and fuel economy. It is worth noting though that dynamic flutter instability is not the only limiting factor of a main wing structure and there are other factors that should be considered. One of the main considerations being the wing deformation and dynamic response under different loads and angles of attack. Another consideration is static aeroelastic effects and flight control reversal.

From Table 7 it can be seen that the best result is achieved while using the genetic algorithm which manages to reduce the mass of the wing by a very substantial 33% while still managing to increase the flutter speed by 128%. This impressive result should be treated with a lot of caution, because as it has been discussed in section 4.4 the solution seems to become

unstable earlier, even though the sign of the damping of the first mode does not change until 214 m/s.

5.2 Neural Network prediction.

From the development of Neural Networks to predict the flutter speed of this structure we have concluded that quite a complex neural network is required for the best accuracy in predictions. What is more, the data required for training is very computationally expensive to acquire and, in most cases, the computational effort of generating the training data far exceeds the computation effort required for a direct optimization.

The accuracy of the predicted results is good enough in most cases but not enough for accurate flutter prediction in every case because there are some outliers in the data, where the prediction is not satisfactory.

It is worth noting that after the model has been trained, there is almost no computational effort required to make a prediction about the flutter speed. The Neural network can then be used for optimization of the structure.

5.3 Future Work

The current study provides some insight into flutter analysis and flutter tailoring techniques using optimization methods and python, but several avenues remain unexplored and could benefit from further investigation.

Future efforts should include at least some experimental validation of the simulation results. This way the influence of the ply-angles of the laminate composite material could be verified in practice

Although several optimization algorithms were employed in this study with promising results, many advanced techniques remain unexplored. Future work could investigate these more sophisticated algorithms, along with the potential application of machine learning-based methods to further enhance optimization processes.

The flutter phenomenon investigated in this thesis, modeled using Vortex Lattice aerodynamic theory coupled with a structural solver based on modal analysis, could also be simulated through a transient Fluid-Structure Interaction (FSI) analysis. This would allow for a comparison of the results between the two approaches, providing further validation and insights

Finaly, the integration of Physics-Informed Neural Networks (PINNs) in this application presents a promising avenue for future research. PINNs are an emerging computational framework that combines data-driven learning with governing physical laws, such as the equations of fluid flow and structural dynamics. By embedding the physics directly into the neural network's training process, PINNs can offer a more efficient and accurate solution to complex fluid-structure interaction problems. Their potential to bypass some of the limitations of traditional numerical methods, such as mesh generation and solver convergence issues, makes them particularly attractive for engineering systems.

6 References

[1] E. Oñate, Structural Analysis with the Finite Element Method, vol. 2, Barcelona: Springer, 2013.

[2] G. A. Bolla, "Aeroelastic Study of the Flutter Conditions of an Aircraft Wing," UNIVERSITAT POLITECNICA DE CATALUNYA, 2022.

[3] A. P. Joseph Katz, Low-Speed Aerodynamics, Second Edition, CAMBRIDGE UNIVERSITY PRESS, 2001.

[4] S. Pinzón, "Introduction to Vortex Lattice Theory," *Ciencia yPoder Aereo,* vol. 10, pp. 39-48, 2015.

[5] H. M. Software, MSC Nastran 2021.1 Aeroelastic User's Guide, 2021.

[6] R. P. Brent, Algorithms for Minimization without Derivatives, New Jersey: Prentice Hall, 1973.

[7] T. A. W. Mykel L. Kochenderfer, Algorithms for Optimization, London: The MIT Press, 2019.

[8] A. Takyar, "Neural networks: Architecture, types, working, applications, case studies, development and implementation," [Online]. Available: https://www.leewayhertz.com/what-are-neural-networks/#Insight:Whatareneuralnetworks?Howdotheywork?-Thebiologicalinspirationbehindneuralnetworks.

[9] K. J. G. D. A. R. A. T. Lisha Li, "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," *Journal of Machine Learning Research,* 2018.

[10] A. Schleicher. [Online]. Available: https://www.alexander-schleicher.de/en/flugzeuge/asw-28/.

[11] "MatWeb," [Online]. Available: https://www.matweb.com/search/datasheet.aspx?matguid=39e40851fc164b6c9bda29d798bf3726&ckck=1.

[12] P.-S. Tang, 29 August 2022. [Online]. Available: https://community.altair.com/community/en/flutter-analysis-setup-tips-tricks?id=kb_article_view&sysparm_article=KB0120069&sys_kb_id=8e92b84fdb659d50cfd5f6a4e296197b&spa=1.

[13] R. G. T. E. O. a. o. Pauli Virtanen, "Fundamental Algorithms for Scientific Computing in Python," *Nature Methods,* vol. 17, pp. 262-272, 2020.

[14] A. F. Gad, “Pygad: An intuitive genetic algorithm python library,” *Multimedia Tools and Applications,* pp. 1-14, 2023.

[15] J. E. Jan R.Wright, Introduction to AIrcraft Aeroelasticity and Loads, John Wiley & Sons Ltd., 2007.

7 Appendix

In this appendix the main code for the simplest case of optimization using Powell's method is presented. The code for all the other applications is modified but the core logic and many parts remain the same throughout. All the code including the code used to process the results and produce the graphics in this thesis is available on GitHub: vasxen/Aeroelastic_Flutter (github.com).

```
 1. import os
 2. import subprocess
 3. import pandas as pd
 4. import numpy as np
 5. from enum import Enum
 6. from scipy.optimize import minimize, Bounds
 7. from numpy.typing import NDArray
 8. from dataclasses import dataclass
 9. from typing import List, Tuple, Callable, Any, Dict
10. from matplotlib.axes import Axes
11. from matplotlib.lines import Line2D
12. from matplotlib.figure import Figure
13. from matplotlib import pyplot as plt
14. import pickle
15. # from time import time
16.
17.
18. # ---------- Classes -------------
19. @dataclass
20. class MAT8():
21.     '''A MAT8 class which contains all the information of a MAT8 orthotropic material
bulk data entry in optistruct'''
22.     LineIndex: int
23.     MID: int
24.     E1: float
25.     E2: float
26.     NU12: float
27.     G12: float
28.     G1Z: str
29.     G2Z: str
30.     RHO: float
31.
32.     def to_string(self) -> str:
33.         ''' This methods returns the information contained in the class in a string
format identical to the one found in a .fem input file'''
34.         s =
f'MAT8,{self.MID},{self.E1},{self.E2},{self.NU12},{self.G12},{self.G1Z},{self.G2Z},{self.RH
O},'
35.         return s
36.
37. @dataclass
38. class Ply():
39.     '''A Ply class which contains all the information of a Ply of an PCOMP composite
shell property bulk data entry in optistruct'''
40.     index: int
41.     MID: int
42.     Thickness: float
43.     Theta: float
44.
45.     def to_string(self, SOUT: str = 'NO') -> str:
46.         ''' This methods returns the information contained in the class in a string
format identical to the one found in a .fem input file'''
47.         s = f'{self.MID},{self.Thickness},{self.Theta},{SOUT}'
48.         return s
49.
```

```
 50. class PCOMP():
 51.     '''A PCOMP class which contains all the information of a Ply of an PCOMP composite
shell property bulk data entry in optistruct'''
 52.
 53.     def __init__(self,Id: int, Plies: List[Ply], original_txt_lines: List[str],
Indeces: Tuple[int,int]):
 54.         self._Id = Id
 55.         self._Plies = Plies
 56.         self._original_txt_lines = original_txt_lines
 57.         self._Indeces = Indeces
 58.
 59.
 60.     #=============== PROPERTIES =================
 61.
 62.     # --------- Id -----------
 63.     @property
 64.     def Id(self):
 65.         return self._Id
 66.
 67.     @Id.setter
 68.     def Id(self, val):
 69.         raise AttributeError('Id is immutable and cannot be changed')
 70.     # --------- // -----------
 71.
 72.     # -------- Plies ---------
 73.     @property
 74.     def Plies(self):
 75.         return self._Plies
 76.
 77.     @Plies.setter
 78.     def Plies(self, val: List[Ply]):
 79.         self._Plies = val
 80.     # --------- // -----------
 81.
 82.     # --------- Original_txt -----------
 83.     @property
 84.     def original_txt_lines(self):
 85.         return self._original_txt_lines
 86.
 87.     @original_txt_lines.setter
 88.     def original_txt_lines(self, val):
 89.         raise AttributeError('original_txt_lines is immutable and cannot be changed')
 90.     # --------- // -----------
 91.
 92.     # --------- Indeces -----------
 93.     @property
 94.     def Indeces(self):
 95.         return self._Indeces
 96.
 97.     @Indeces.setter
 98.     def Indeces(self, val: Tuple[int, int]):
 99.         raise AttributeError('Indeces is immutable and cannot be changed')
100.
101.     # --------- // -----------
102.
103.
104.     # --------- NumPlies -----------
105.     @property
106.     def NumPlies(self) -> int:
107.         return len(self.Plies)
108.     # --------- // -----------
109.
110.
111.     #=============== METHODS =================
112.
113.     def to_string(self) -> List[str]:
```

```
114.         ''' This methods returns the information contained in the class in a string
format identical to the one found in a .fem input file'''
115.
116.         Lines: List[str] = []
117.         Lines.append(self.original_txt_lines[0][:-1])
118.
119.         for i in range(0,self.NumPlies - 1,2):
120.             ply1 = self.Plies[i]
121.             ply2 = self.Plies[i + 1]
122.             line = f'+,{ply1.to_string()},{ply2.to_string()},'
123.             Lines.append(line)
124.
125.         if i + 1 < self.NumPlies - 1:
126.             line = f'+,{self.Plies[-1].to_string()},'
127.             Lines.append(line)
128.
129.         return Lines
130.
131. @dataclass
132. class FlutterAnalysisPoint():
133.     ModeNumber : int
134.     MachNumber : float
135.     DensityRatio: float
136.     Method: str
137.     Data: pd.DataFrame
138.
139.     def DetectFlutter(self) -> Tuple[List[float], List[Tuple[int, int]]]:
140.         Stable = self.Data['DAMPING'] < 0
141.         FlutterIndeces: List[Tuple[int, int]] = []
142.         for i in range(1, Stable.shape[0]):
143.             if (Stable[i-1]) and (not Stable[i]):
144.                 FlutterIndeces.append((i-1, i))
145.
146.         FlutterSpeed = []
147.         for ind in FlutterIndeces:
148.             D1 = self.Data['DAMPING'][ind[0]]
149.             D2 = self.Data['DAMPING'][ind[1]]
150.             V1 = self.Data['VELOCITY'][ind[0]]
151.             V2 = self.Data['VELOCITY'][ind[1]]
152.             m = (D2 - D1) / (V2 - V1)
153.             Vflutter = V1 - D1/m
154.             FlutterSpeed.append(Vflutter)
155.
156.         return FlutterSpeed, FlutterIndeces
157.
158.     def Plot(self,ax: Axes, ix: int, iy: int, label: str = '') -> Line2D:
159.         line, =  ax.plot(self.Data.iloc[:,ix], self.Data.iloc[:,iy], marker = '.',
label = label )
160.         return line
161.
162.
163.     def PlotDamping(self) -> Figure:
164.         fig, ax = plt.subplots()
165.         self.Plot(ax, 2, 3, f'MODE {self.ModeNumber}')
166.         ax.set_xlabel('VELOCITY')
167.         ax.set_ylabel('DAMPING')
168.         ax.legend()
169.         return fig
170.
171.
172.     def PlotFrequency(self) -> Figure:
173.         fig, ax = plt.subplots()
174.         self.Plot(ax, 2, 4, f'MODE {self.ModeNumber}')
175.         ax.set_xlabel('VELOCITY')
176.         ax.set_ylabel('FREQUECY')
177.         ax.legend()
```

```
178.         return fig
179.
180.
181.
182. @dataclass
183. class FlutterSubcase():
184.     SubcaseId: int
185.     XY_Symmetry: bool
186.     XZ_Symmetry: bool
187.     Method: str
188.     NumPoints: int
189.     Points: List[FlutterAnalysisPoint]
190.
191.     def FlutterInfo(self):
192.         FlutterInfo: Dict[int, float] = {}
193.         for point in self.Points:
194.             Vel, _ = point.DetectFlutter()
195.             if Vel: FlutterInfo[point.ModeNumber] = min(Vel)
196.
197.         return FlutterInfo
198.
199. class FlutterSummary():
200.
201.     @staticmethod
202.     def __split_subcases(lines: List[str]) -> List[List[str]]:
203.         subcase_id = []
204.         for item in lines:
205.             if item.startswith('Subcase'):
206.                 split = item.split()
207.                 subcase_id.append(int(split[2]))
208.
209.         subcase_id = np.array(subcase_id)
210.         Subcaseschange: List = (np.where(subcase_id[:-1] != subcase_id[1:])[0] +
1).tolist()
211.         Subcaseschange.insert(0, 0)
212.         Subcaseschange.append(len(lines))
213.         subcase_indeces = Subcaseschange
214.
215.         Subcases = []
216.         for i in range(len(subcase_indeces) - 1):
217.             Subcases.append(lines[subcase_indeces[i]:subcase_indeces[i+1]])
218.         return Subcases
219.
220.     @staticmethod
221.     def __split_points(lines: List[str]) -> List[List[str]]:
222.         point_indeces: List[int] = []
223.         for ind, item in enumerate(lines):
224.             if item.startswith('Subcase'):
225.                 point_indeces.append(ind)
226.
227.         point_indeces.append(len(lines))
228.
229.         points:List[List[str]] = []
230.         for i in range(len(point_indeces) -1 ):
231.             points.append(lines[point_indeces[i]: point_indeces[i+1]])
232.         return points
233.
234.     @staticmethod
235.     def __read_analysis_point(point: List[str]) -> FlutterAnalysisPoint :
236.         def remove_letters(input_string: str) -> str:
237.             tempresult = ''
238.             # Iterate through each character in the input string
239.             for char in input_string:
240.                 # Check if the character is not a letter
241.                 if not char.isalpha():
242.                     tempresult += char
```

```
243.
244.             return tempresult.replace('+', 'E+')
245.
246.         def read_header(header: str)-> Tuple[int, float, float, str]:
247.             split = header.strip().split('=')
248.             # assert len(split) == 12, 'length of header of analysis point incorrect'
249.             headerdata =[remove_letters(item) for item in split[1:-1]]
250.             headerdata.append(split[-1].strip())
251.             out = (int(headerdata[0]), float(headerdata[1]), float(headerdata[2]),
str(headerdata[3]))
252.             return out
253.
254.         def read_data(data: List[str]) -> pd.DataFrame:
255.                 # Initialize an empty list to store rows
256.             rows = []
257.
258.             # Iterate over each string in the input list
259.             for string in data:
260.                 # Split the string into numbers
261.                 numbers = string.split()
262.                 row = [float(num) for num in numbers]
263.                 rows.append(row)
264.
265.             # Create a DataFrame from the list of rows
266.             df = pd.DataFrame(rows, columns = ['KFREQ', '1/KFREQ', 'VELOCITY',
'DAMPING', 'FREQUENCY', 'COMPLEX', 'EIGENVALUE'])
267.             return df
268.
269.         header = point[3]
270.         data = point[5:]
271.
272.         header = read_header(header)
273.         data = read_data(data)
274.         return FlutterAnalysisPoint(*header, data)
275.
276.     @staticmethod
277.     def __read_subcase_header(subcase_header: List[str]) -> Tuple[int, bool, bool]:
278.         Id = int(subcase_header[0].split('=')[1])
279.         row3 = subcase_header[2]
280.         row3eqsplit = row3.split('=')
281.         row3split = []
282.         for s in row3eqsplit:
283.             row3split.extend(s.split())
284.
285.         XY_Symmetry: bool = False
286.         XZ_Symmetry: bool = False
287.         if row3split[2] == 'SYMMETRIC':
288.             XY_Symmetry = True
289.
290.         if row3split[5] == 'SYMMETRIC':
291.             XZ_Symmetry = True
292.         return (Id, XY_Symmetry, XZ_Symmetry)
293.
294.     def __init__(self, filepath: str):
295.         self.title: str = ''
296.         self.Filepath: str = ''
297.         self.NumSubcases: int = 0
298.         self.Subcases: List[FlutterSubcase] = []
299.
300.         assert os.path.isfile(filepath), f'File at {filepath} does not exist'
301.
302.         #Title
303.         self.title = os.path.basename(filepath).split('.')[0]
304.
305.         #Filepath
306.         self.Filepath = filepath
```

```
307.
308.         #Subcases
309.         with open(filepath, 'r') as file:
310.             lines = file.readlines()
311.         stripedlines = []
312.         for line in lines:
313.             if line.isspace():
314.                 lines.remove(line)
315.             else:
316.                 stripedlines.append(line.strip())
317.
318.         # Split on Subcase level
319.         subcases = self.__split_subcases(stripedlines)
320.         Subcases: List[FlutterSubcase] = []
321.         for subcase in subcases:
322.             subcase_header = subcase[0:3]
323.             SubcaseHeader = self.__read_subcase_header(subcase_header)
324.             points = self.__split_points(subcase)
325.             Points: List[FlutterAnalysisPoint] = []
326.             for point in points:
327.                 Points.append(self.__read_analysis_point(point))
328.
329.             SubcaseMethod = Points[0].Method
330.             Subcases.append(FlutterSubcase(*SubcaseHeader, SubcaseMethod, len(Points),
Points))
331.
332.         self.Subcases = Subcases
333.
334.         #Number of Subcases
335.         self.NumSubcases = len(Subcases)
336.
337.     def FlutterInfo(self) -> str:
338.         s = f'\n\n********* FLUTTER INFORMATION *********\n\n'
339.         for subcase in self.Subcases:
340.             s += f'|------ SUBCASE {subcase.SubcaseId} ------|\n'
341.             s += '|                          |\n'
342.             for point, vel in subcase.FlutterInfo().items():
343.                 s += f'| POINT {point} -> {round(vel, 2)} m/s |\n'
344.         return s
345.
346. class PlySymmetry(Enum):
347.     AntiSymmetric = -1
348.     NoSymmetry = 0
349.     Symmetric = 1
350.
351. class ToleranceWrapper():
352.     def __init__(self, func: Callable,
353.                  ThicknessTolerance: float,
354.                      AngleTolerance: float,
355.                        FlutterVelocityConstraint: float,
356.                          inputfile: str,
357.                            solverpath: str,
358.                              Penalty: float = 1E10,
359.                                PlySymmetry: PlySymmetry = PlySymmetry.AntiSymmetric):
360.         self.func = func
361.         self.ThicknessTolerance = ThicknessTolerance
362.         self.AngleTolerance = AngleTolerance
363.         self.FlutterVelocityConstraint = FlutterVelocityConstraint
364.         self.inputfile = inputfile
365.         self.solverpath = solverpath
366.         self.Penalty = Penalty
367.         self.PlySymmetry = PlySymmetry
368.         self.cache: Dict[Tuple[float], float] = {}
369.         self.history: Dict[Tuple[float], float] = {}
370.
371.     def __call__(self, x: NDArray[np.float64]) -> float:
```

```
372.         t = x[0]
373.         t = np.round(t / self.ThicknessTolerance) * self.ThicknessTolerance
374.
375.         a = x[1:]
376.         a = (np.round(a / self.AngleTolerance) * self.AngleTolerance).tolist()
377.         NumLayers = len(a)
378.         t = NumLayers * [t]
379.         rounded_input = (*t, *a)
380.         if rounded_input in self.cache:
381.             result = self.cache[rounded_input]
382.             self.history[rounded_input] = result
383.             return result
384.         else:
385.             result = self.func(t, a, self.inputfile, self.solverpath,
self.FlutterVelocityConstraint, self.PlySymmetry, self.Penalty)
386.             self.history[rounded_input] = result
387.             self.cache[rounded_input] = result
388.             return result
389.
390.     def __str__(self) -> str:
391.         s = f'Thickness Tolerance: {self.ThicknessTolerance}\n'
392.         s += f'Angle Tolerance: {self.AngleTolerance}\n'
393.         s += '\nCached Data:\n'
394.         for k, v in self.cache.items():
395.             s += f'x = {k} -> f(x) = {v}\n'
396.         return s
397.
398.     def savecahce(self, file: str) -> None:
399.         assert file.endswith('.xlsx'), f'The file must be an .xlsx file not a
.{file.split('.')[1]} file'
400.         Dataframe = pd.DataFrame(list(self.cache.items()), columns = ['Input Vector',
'Function Value'])
401.         Dataframe.to_excel(file)
402.         return
403.
404.     def savehistory(self, file: str) -> None:
405.         assert file.endswith('.xlsx'), f'The file must be an .xlsx file not a
.{file.split('.')[1]} file'
406.         Dataframe = pd.DataFrame(list(self.history.items()), columns = ['Input
Vector', 'Function Value'])
407.         Dataframe.to_excel(file)
408.         return
409.
410. # -------- Functions --------------
411. def ReadFem(inputfile: str) -> Tuple[List[PCOMP], List[MAT8]]:
412.     '''This funciton reads a .fem optistruct iinput file and detects any PCOMP and any
MAT8 entries. This function assumes that PCOMP refers exlusively to MAT8 material types.
413.      ### Parameters:
414.      - inputfile: a file path to a .fem file given as a string
415.      ### Returns:
416.      - A List of PCOMP objects
417.      - A list of MAT8 objects '''
418.     def readply(PlyIndex: int, input: List[str]) -> Ply:
419.         Mid = int(input[0])
420.         Thickness = float(input[1])
421.         Theta = float(input[2])
422.         return Ply(PlyIndex, Mid, Thickness, Theta)
423.
424.     def readPCOMP(StartIndex: List[int]) -> List[PCOMP]:
425.         numPCOMP = len(StartIndex)
426.         EndIndex: List[int] = []
427.         for index in StartIndex:
428.             i = 1
429.             while lines[index + i].startswith('+'):
430.                 i += 1
431.
```

```
            EndIndex.append(index + i)

        PCOMPs: List[PCOMP] = []

        for i in range(numPCOMP):
            Section = lines[StartIndex[i] : EndIndex[i]]
            Id = int(Section[0].split(',')[1])
            Plies_str: List[List[str]] = []
            for line in Section[1:]:
                SplitLine = line.split(',')
                lenSplit = len(SplitLine)
                if lenSplit == 10:
                    if SplitLine[8] == 'YES' or SplitLine[8] == 'NO':
                        Plies_str.append(SplitLine[1:5])
                        Plies_str.append(SplitLine[5:9])
                    else:
                        pass
                elif lenSplit == 6:
                    if SplitLine[4] == 'YES' or SplitLine[4] == 'NO':
                        Plies_str.append(SplitLine[1:5])

            Plies: List[Ply] = []
            for Plyindex, ply in enumerate(Plies_str):
                Plies.append(readply(Plyindex, ply))

            PCOMPs.append(PCOMP(Id, Plies, Section, (StartIndex[i], EndIndex[i]) ))
        return PCOMPs

    def readMAT8(Indeces: List[int]) -> List[MAT8]:
        MAT8s: List[MAT8] = []
        for index in Indeces:
            line = lines[index]
            split = line.split(',')[1:]
            MID = int(split[0])
            E1 = float(split[1])
            E2 = float(split[2])
            NU12 = float(split[3])
            G12 = float(split[4])
            G1Z = split[5]
            G2Z = split[6]
            RHO = float(split[7])
            M = MAT8(index, MID,E1,E2,NU12,G12,G1Z,G2Z,RHO)
            MAT8s.append(M)

        return MAT8s

    if not inputfile.endswith('.fem'):
        raise ValueError('Incorrect file type. File must be of type .fem')

    with open(inputfile, 'r') as f:
        lines = f.readlines()

    StartIndex: List[int] = []
    Mat8Indeces: List[int] = []

    for index, line in enumerate(lines):
        if line.startswith('PCOMP'):
            StartIndex.append(index)
        elif line.startswith('MAT8'):
            Mat8Indeces.append(index)

    PCOMPs = readPCOMP(StartIndex)
    MAT8s = readMAT8(Mat8Indeces)

    return PCOMPs, MAT8s

```

```
498. def WriteFem(Properties: List[PCOMP], Materials: List[MAT8], inputfile: str) -> None:
499.     '''Writes a back to a .fem file the modified PCOMP and MAT8 objects given in the
input. It is designed to be used on the same file as ReadFem function
500.     ### Parameters:
501.     - Properties: A list of PCOMP objects
502.     - Materials: A list of MAT 8 objects
503.     - inputfile: A path to a file as a string to which the objects will be returned
504.     '''
505.     def replace_lines_in_file(file_path: str, line_numbers : List[int], new_lines:
List[str]):
506.         if len(line_numbers) != len(new_lines):
507.             raise ValueError("The number of line numbers must match the number of new
lines.")
508.
509.         try:
510.             # Read all lines from the file
511.             with open(file_path, 'r') as file:
512.                 lines = file.readlines()
513.
514.             # Replace the specified lines
515.             for line_number, new_line in zip(line_numbers, new_lines):
516.                 if 0 <= line_number < len(lines):
517.                     lines[line_number] = new_line + '\n'
518.                 else:
519.                     Warning(f"Warning: Line number {line_number} is out of range.
Skipping this replacement.")
520.
521.             # Write the modified lines back to the file
522.             with open(file_path, 'w') as file:
523.                 file.writelines(lines)
524.
525.             print("Lines replaced successfully.")
526.
527.         except FileNotFoundError:
528.             print(f"Error: The file {file_path} was not found.")
529.         except Exception as e:
530.             print(f"An error occurred: {e}")
531.
532.     if Properties:
533.         for Pcomp in Properties:
534.             LineIndeces = list(range(Pcomp.Indeces[0], Pcomp.Indeces[1]))
535.             replace_lines_in_file(inputfile, LineIndeces, Pcomp.to_string())
536.
537.     if Materials:
538.         for Mat in Materials:
539.             replace_lines_in_file(inputfile, [Mat.LineIndex], [Mat.to_string()])
540.
541. def readmass(outputfile: str) -> float:
542.     ''' Reads a .out file and checks if it contans Mass infirmation if it does it
returns the mass as a float otherwise it throws a Value Error
543.     ### Parameters:
544.     - outputfile: A path as a string to a .out file'''
545.     if not outputfile.endswith('.out'):
546.         raise ValueError('Incorrect file type. File must be of type .out')
547.
548.     with open(outputfile, 'r') as f:
549.         lines = f.readlines()
550.
551.     Mass: float = -1.0
552.     for line in lines:
553.         if 'Mass' in line:
554.             MassString = line.split('Mass')[1]
555.             MassString = MassString.replace('=', '')
556.             MassString = MassString.strip()
557.             Mass = float(MassString)
558.             break
```

```
559.
560.     if Mass == -1.0: raise ValueError('Provided file does not contain any Mass
information')
561.
562.     return Mass
563.
564. def CallSolver(inputfile: str, solverpath: str, options: str) ->
subprocess.CompletedProcess:
565.     '''This function calls the optistruct solver on the input file.
566.     The function works by writing a temporary batch file the excecuting it and finally
deleting it.
567.     It returns the std out to the consoleand the completedProcess object provided by
the subprocess module
568.     ### Parameters:
569.     - inputfile: a path as a string to a .fem solver input file
570.     - solverpath: the path to the altair yperwors solver scripts as a string
571.     - options: a string of options excactly as they would be set in the command window
or in the altair compute consosle'''
572.     inputfile = f'"{inputfile.replace('/', '\\')}"'
573.     solverpath = f'"{solverpath.replace('/', '\\')}"'
574.
575.
576.     lines = ['@echo off\n',
577.              'optistruct ' + inputfile + ' ' + options]
578.
579.     with open('temp.bat', 'w') as file:
580.         file.writelines(lines)
581.
582.
583.     s = subprocess.run([f'temp.bat'])
584.     # os.remove('temp.bat')
585.     return s
586.
587. def ObjectiveFunction(thicknesses: List[float], angles: List[float], inputfile: str,
solverpath: str, FlutterVelocityConstraint: float, sym: PlySymmetry, penalty: float) ->
float:
588.     assert len(thicknesses) == len(angles), f'Thicknesses and angles must haver the
same length'
589.     assert all([e > 0 for e in thicknesses]), ' All thicknesses must be strictly
positive'
590.
591.     Properties, _ = ReadFem(inputfile)
592.     assert len(Properties) == 1, f'Function expected only one PCOMP to optimize,
{len(Properties)} found'
593.     Property = Properties[0]
594.     # Property.to_string()
595.
596.     match sym.value:
597.         case -1: #Antisymmetric
598.             assert Property.NumPlies / 2 == len(thicknesses), f'for antisymmetric
laminates the length of the inputs should be half the number of plies'
599.             thicknesses.extend(thicknesses)
600.             angles.extend([-e for e in angles])
601.
602.             for i in range(Property.NumPlies):
603.                 Property.Plies[i].Thickness = thicknesses[i]
604.                 Property.Plies[i].Theta = angles[i]
605.
606.         case 0: # No symmetry
607.             assert Property.NumPlies == len(thicknesses), f'length of the number of
inputs should be half the number of plies'
608.             for i in range(Property.NumPlies):
609.                 Property.Plies[i].Thickness = thicknesses[i]
610.                 Property.Plies[i].Theta = angles[i]
611.
612.         case 1: # Symmetric
```

```
613.             assert Property.NumPlies / 2 == len(thicknesses), f'for symmetric
laminates the length of the inputs should be half the number of plies'
614.             thicknesses.extend(thicknesses)
615.             angles.extend(angles)
616.
617.             for i in range(Property.NumPlies):
618.                 Property.Plies[i].Thickness = thicknesses[i]
619.                 Property.Plies[i].Theta = angles[i]
620.
621.
622.     WriteFem([Property], [], inputfile)
623.     CallSolver(inputfile, solverpath, '-nt 6')
624.     outfile = inputfile.replace('.fem', '.out')
625.     fltfile = inputfile.replace('.fem', '.flt')
626.
627.     Mass = readmass(outfile)
628.     Flutter = FlutterSummary(fltfile)
629.     assert len(Flutter.Subcases) == 1, f'Analysis should include only one subcase not
{len(Flutter.Subcases)}'
630.     Subcase = Flutter.Subcases[0]
631.
632.     Velocities: List[float] = []
633.     for point in Subcase.Points:
634.         Vel, _ = point.DetectFlutter()
635.         if Vel: Velocities.append(Vel[0])
636.
637.     P: float = 0
638.     if Velocities:
639.         FlutterVelocity = min(Velocities)
640.         if FlutterVelocity < FlutterVelocityConstraint:
641.             P = penalty * (FlutterVelocityConstraint - FlutterVelocity)
642.
643.     Objective = Mass + P
644.
645.     return Objective
646.
647. def DeleteUnessesaryFiles(directory: str, FileExtensions: Tuple[str, ...]) -> None:
648.     '''This function deletes al;l file with certain extension within a directory USE
CAREFULLY
649.     ### Parameters:
650.     - directory: path a string to a directory (folder)
651.     - FileExtensions: A tuple of string containing the extension that will be deleted
including the dot eg: ('.txt', '.exe') '''
652.     files = os.listdir(directory)
653.
654.     for file in files:
655.         if file.endswith(FileExtensions):
656.             os.remove(os.path.join(directory, file))
657.
658. def main():
659.     # Define optimization problem parameters
660.     inputFile = "C:/Users/vasxen/OneDrive/Thesis/code/ASW28 Wing.fem"
661.     solverpath = "C:/My_Programms/Altair/hwsolvers/scripts"
662.     x0 = np.array([0.0005, 45, -45, 45], dtype = np.float64)# initial solution vector
663.     lower_bounds = [0.0002] + 3 * [-90]                     # lower constraints of
thickness and ply angles
664.     upper_bounds = [0.0008] + 3 * [+90]                     # Upper constraints of
thickness and ply angles
665.     bounds = Bounds(lower_bounds, upper_bounds)             # type: ignore
666.     options = {'disp' : True,
667.                'maxfev' : 1000,
668.                'return_all' : True}
669.     WrappedObj = ToleranceWrapper(ObjectiveFunction, 0.0001, 1, 90, inputFile,
solverpath)
670.     Min = minimize(WrappedObj, x0 = x0,method = 'powell', bounds = bounds, options =
options)
```

```
671.     DeleteUnessesaryFiles(os.path.dirname(inputFile), FileExtensions = ('.out',
'.stat', '.mvw'))
672.     DeleteUnessesaryFiles(os.getcwd(), FileExtensions = ('.bat', ))
673.
674.
675.     WrappedObj.savecahce('FunctionEvaluations.xlsx')
676.     WrappedObj.savehistory('OptimizationHistory.xlsx')
677.     with open('minimization.pkl', 'wb') as f:
678.         pickle.dump(Min, f)
679.
680.     with open('Optimization Summary.txt', 'w') as f:
681.         print('\n\n=========== OPTIMIZATION SUMMARY ===========', file = f)
682.         print(Min, file = f)
683.         print(FlutterSummary(inputFile.replace('.fem', '.flt')).FlutterInfo(), file =
f)
684.
685.
686. if __name__ == '__main__':
687.     main()
688.
689.
```