Delft University of Technology

Simulation, Verification and Validation
AE3212-II

Group A12

Simulation, Verification and Validation
for the analysis of a Fokker 100 aileron under critical loading conditions

Authors:

P. Meseguer Berroy (4530934)

L. Chin A Foeng (4609972)

M. Enting (4659147)

E. Daugulis (4662350)

G. Mettepenningen (4682343)

P. González Martínez (4582675)

January 28, 2022

Contents

Introduction

As explained in the simulation plan report [1], in recent years, many technical disciplines have been revolutionised by the rapid improvements in the design and capabilities of finite element methods (FEM) and other numerical simulation software [2]. The emergence of this software, in part driven by significant developments in computer hardware, has allowed professionals to perform highly complex calculations to aid in the problem-solving process for a broad range of applications. The aerospace industry has been in the forefront of the development and adoption of this technology for many years. The development of aerospace structures is a highly complex design process where the cost of failure is simply too high. As engineers continue to rely on numerical simulation software to guide their design choices and streamline the design process, it is vital to ensure that the software used is verified and validated in a rigorous manner.

This document presents the development, verification, and validation of a numerical model used to conduct structural analysis simulations of the aileron of a Fokker 100 aircraft under critical loading conditions. The aileron is at its maximum deflection angle, suffering from an actuator malfunction, and subjected to multi-axial loading. The goal of the project is to model the stresses and deflections of an aileron under a limit load by creating a numerical model. After the numerical model is made, verification methods are created and the model is verified. Finally, it is validated to compare the results with the validation data and find the discrepancies.

The structure of this report is as follows. Chapter 1 presents a description of the problem, including the loading scenario and the geometry of the aileron under consideration. Chapter 2 describes the numerical model itself, including the spatial discretisation of the aileron structure and the numerical schemes used for interpolation and integration of the provided aerodynamic loading data. The chapter also gives the results of the simulation of the aileron. Chapter 3 presents the verification process for the numerical model, including a description of the verification model and the relevant verification tests. Lastly, Chapter 4 describes how the numerical model is validated using a finite element model of the aileron of a Boeing 737.

1 Problem Description: Load Case

In this section the loading case is explained based on the given information in [3]. The loads acting on the aileron are shown below in the free body diagram of the structure (Figure 1). A coordinate frame is established as shown in Figure 1 and Figure 2. The z-axis is aligned with the symmetry axis of the aileron and the x-axis is in the spanwise direction. A right handed coordinate frame is used and it rotates together with the aileron. Throughout the rest of the report, the x,y,z reference frame will always be aligned with the aileron deflection and no static reference frame will be used. The aileron is deflected in its maximum upward position making an angle θ with the wing. In Figure 2, y_E and z_E represent the y and z coordinates along the Earth frame of reference.

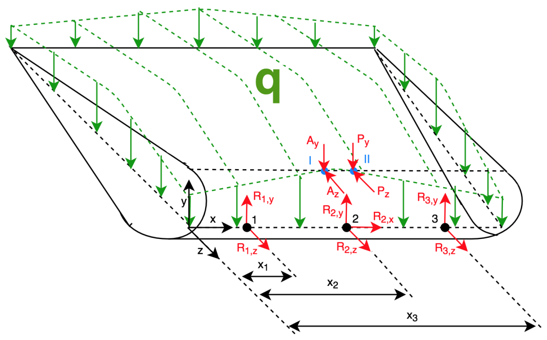


Figure 1: Free Body Diagram of the Aileron [1]

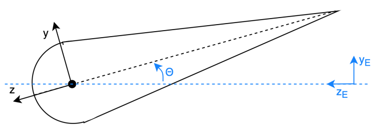


Figure 2: Reference Frame with Respect to Earth Frame Coordinate System [1]

2 Numerical Model

The following section will outline the required assumptions and steps to take to create a working numerical model of the aileron. It will take in the given values and produce the deflection of the hinge line, the twist of the aileron as well as the maximum stress experienced by the aileron.

2.1 Assumptions

In order to set up a numerical method, all the loads acting on the aileron should be taken into account. However, the magnitude of these loads is sometimes very small and can be neglected. Thus, as mentioned in [1], the following assumptions have been made to simplify the computational load of the numerical method.

- **AS-NM-01:** The aileron can be modelled as a thin-walled structure
  The dimensions of the skin thickness of the aileron are more than ten times smaller than those of the cross-sectional area of the aileron. Thus, the effects caused by the skin thickness can be neglected. This will lead to an averaged value of the shear stress running through the skin thickness, leading to a slight variation of the total shear stress around the aileron cross-section.
- **AS-NM-02:** The aileron can be modelled as a beam
  By modelling the aileron as a beam, buckling and other deformations that occur inside the structure, will not be taken into account. This is done for simplification purposes, as the stress due to this deformations is negligible compared to the one caused by torsion or the distributed aerodynamic load. This will have an impact on the value of the total stress acting on the structure, which will be underestimated.
- **AS-NM-03:** The aileron has a constant cross-section in y-z plane
  The dimensions of the aileron on the z-direction do not vary throughout the x-direction. In other words, the taper ratio of the aileron is equal to 1. Moreover, the three hinge spaces present in the aileron are neglected. These assumptions will lead to possible moment of inertia variations as well as shear flow variations. Also, the stress concentration will be lower than in reality as the stress induced by the hinge spaces will not be taken into account.
- **AS-NM-04:** The reaction forces acting on the hinges and actuators can be modelled as point loads
  Since the hinge space length and the actuator diameter are relatively small compared to the whole length of the aileron, the effect of a distributed load could be approximated by a point load. However, this will mean that in some points where there is a distributed load acting, the effect of it will not be taken into account. Thus, the total value of the stress, shear and other loads may slightly differ from reality.
- **AS-NM-05:** The effect of stiffener attachments to aileron skin is neglected
  The manufacturing technique used to attach the stiffeners to the aileron skin is not known. Most of these techniques make the stresses pass through the stiffeners without taking any of the stress load. Thus, it is assumed that one of these techniques will be used. In case the technique used uses a material that takes up part of the stress present in the skin, it will lead to a stress overestimation.
- **AS-NM-06:** Actuator II can be modelled as a rigid beam
  Only the forces acting on the aileron are of interest. Therefore it is assumed that actuator II does not deform under its loads and translates them directly to the aileron. This means the compression or elongation of the actuator is ignored, thus the z-coordinate of the point of application of the actuator could slightly differ from the given values.
- **AS-NM-07:** The weight of the aileron including the stiffeners is neglected
  The contribution of the weight (67.48N) is negligible compared to the aerodynamic load q and the discrete load P (in the order of kNs). Neglecting the weight underestimates the shear, but this difference is very small compared to the shear induced by the aerodynamic load and the torsion.
- **AS-NM-08:** The stiffeners are thin walled
  Again, the skin thickness of the stiffeners (1.1mm) is less than a tenth of the geometry, where the smallest measure is a height of 1.3cm. The moment of inertia will then be underestimated, leading to an overestimation of the normal and shear stresses.
- **AS-NM-09:** The material is homogeneous and contains no imperfections
  The mass within the material is distributed evenly throughout the material. The effect of this is that imperfections are not taken into account in the model. Imperfections would only make the material properties slightly worse than what was taken into account in the assumptions, which could potentially make the values of the deflection, and thus the total torque, slightly differ from reality.
- **AS-NM-10:** The aileron stiffeners are idealised, meaning they are assumed to be point areas
  The stiffener structure is very small compared to the airfoil structure. The moment of inertia differs, because only the Steiner terms are taken into account. This influences bending stress and shear stress calculations.

- **AS-NM-11:** Stresses are linear so they can be superimposed
  This assumption can be made as the material properties of the aileron are constant, the deformation of the structure is small relative to the deformation of the wing as a whole and buckling is not a concern (recall assumption **AS-NM-02**). This will lead to a stress overestimation in the structure.

- **AS-NM-12:** Exclusion of stiffeners in torsional stiffness analysis
  The torsion in a structure is by far mostly taken up by the closed sections, in this case the skin. The stiffeners are open sections that only take up a small part of the total torsion. Also, the contribution of the skin to the shear flow ($\int tyds$) is significantly larger than the contribution of the stiffeners ($\Sigma A_{st} d_{y,st,i}$). Therefore, their contribution to the torsional stiffness can be neglected. The effect of this assumption is that torsional stiffness J is slightly underestimated.

2.2 Spatial Discretisation

In order to compute the stress distribution in the aileron for the given loading conditions, the structure must be discretised spatially. This section describes the 2D spatial discretisation of the aileron surface, in addition to the computation of the x and z locations of the nodes corresponding to the provided aerodynamic data. This can also be found in the simulation plan report [1].

The spatial discretisation of the structure is performed as follows. The three-dimensional aileron structure is first split into a collection of two-dimensional slices taken in the spanwise direction (along the x-axis). Then, each of those cross sections is discretised in the chordwise direction (along the z-axis). Figure 3 shows a schematic representation of the discretisation.

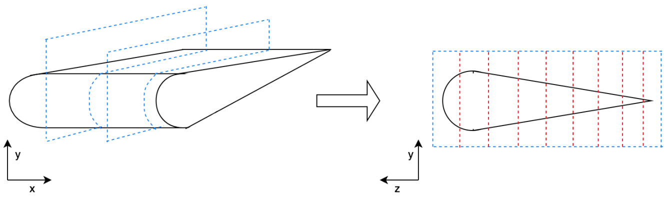


Figure 3: Schematic representation of the spatial discretisation of the aileron structure. [1]

This discretisation approach arises naturally from the provided aerodynamic loading data. The aerodynamic loading on the aileron structure is given as a discrete data set in the form of a `.dat` file. The data set can be interpreted as an 81 × 41 matrix, where each of the 81 rows corresponds to a chordwise station, and each of the 41 columns to a spanwise station. At each station, the loading is provided in units of $\frac{\text{kN}}{\text{m}^2}$, or kPa. The formulas for x- and z-coordinates of the ith chordwise station can be found in [3].

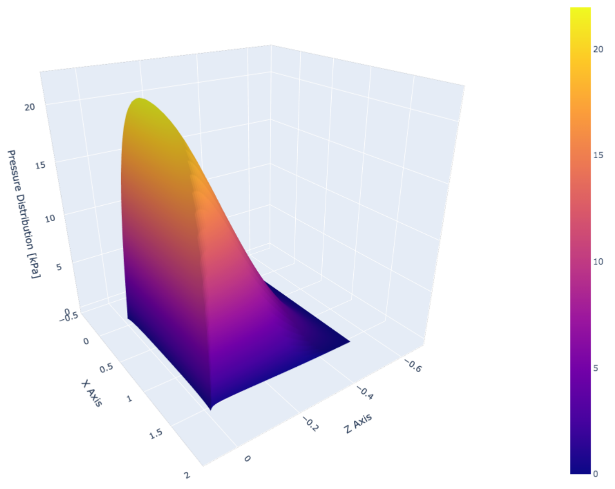


Figure 4: Surface plot of the provided aerodynamic data (81 × 41 grid). Note that the positive z-axis points towards the leading edge of the aileron. [1]

Note that the aerodynamic loading on the aileron, which in reality is a continuous function of $h(x, y, z)$[1], is provided as a set of discrete, two-dimensional data in the x-z plane. Thus, it is clear to see how the format of the provided data naturally leads to a discretisation of the aileron structure in the x and z directions, as described above. A plot of the aerodynamic loading is shown below in Figure 4. Although Figure 4 appears to show a continuous loading distribution, the used data is discrete.

2.3 Interpolation of aerodynamic data

In order to obtain reasonably accurate estimations of the stresses in the structure, it is likely that the grid spacing needs to be decreased in both the spanwise and chordwise directions. This requires the use of an interpolation scheme to convert the provided aerodynamic data, given in a discrete grid with 81×41 stations in the z and x directions respectively, to one with a higher resolution. The type of interpolation and the interpolation procedure used in the numerical model are explained hereinafter.

Univariate interpolation using cubic splines The chosen interpolation scheme performs univariate interpolation using cubic splines. Univariate interpolation is chosen over bivariate interpolation for simplicity, which ensures that (a) the code will be less prone to errors, and (b) the run time of the numerical model will be shorter, improving its efficiency. Splines are used in order to overcome some of the limitations of standard polynomial interpolation, which produces a single polynomial interpolant of degree $\leq n$ when interpolating through $n+1$ data points. The resulting interpolant can be unstable for large n, and the solution computationally expensive [4]. Furthermore, standard polynomial interpolation suffers from high frequency oscillations near the boundaries of the grid (Runge phenomenon) [4]. These limitations are overcome by using spline interpolation, which uses a *set* of low-degree polynomials to interpolate the data. Lastly, it is worthy to note that another advantage of spline interpolation is that it allows for the interpolation error to be made arbitrarily small by decreasing the grid spacing [4].

More specifically, a cubic polynomial (or spline) is used to interpolate the data at each interval. In contrast with linear splines, cubic splines have the advantage of being in $C^2([a, b])$ (their first and second derivatives are continuous on the interval $[a, b]$). This allows one to impose continuity constraints on the resulting piecewise interpolant to improve its smoothness over the entire interpolation domain [4]. The smoothness of the interpolant is of vital importance, since the aim is for it to approximate the type of aerodynamic loading distributions that occur in reality, which tend to be smooth and do not contain high frequency oscillations.

Using cubic splines, the univariate interpolation problem is equivalent to finding the coefficients of the cubic polynomials:

$$s_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, \quad i = 0, 1, ..., n - 1 \tag{1}$$

where x denotes the direction of interpolation, and n denotes the number of intervals (thus $n + 1$ is the number of data points). Following the method presented in [4, pp. 41-44], one can use interpolation constraints at the $n + 1$ data points, and continuity constraints at the $n - 1$ internal nodes to compute the coefficients of the cubic splines. Let $M_i := s''(x_i)$, i.e. the value of the second derivative of the interpolant at the node x_i. In order to obtain the coefficients, one must first solve the following system of $n - 1$ equations for the $n + 1$ unknowns $M_0, M_1, ..., M_n$:

$$\frac{h_{i-1}}{6} M_{i-1} + \frac{h_i + h_{i-1}}{3} M_i + \frac{h_i}{6} M_{i+1} = \frac{f_{i+1} - f_i}{h_i} - \frac{f_i - f_{i-1}}{h_{i-1}} \tag{2}$$

where $h_i := x_{i+1} - x_i$ and f_i is the value of the aerodynamic load at the node x_i. Since there are two more unknowns than equations, two additional constraints are needed. For simplicity, the so-called natural cubic splines are used. These set the following two constraints to complete the system of equations, namely $M_0 = M_n = 0$.

The resulting system of equations is written in the form $A\overrightarrow{m} = \overrightarrow{f} \implies \overrightarrow{m} = A^{-1}\overrightarrow{f}$ where $\overrightarrow{m} = [M_0, M_1, ..., M_n]^T$. The values of $\overrightarrow{m}$ can then be used to compute the coefficients of the cubic splines (see [4, p. 42]).

As stated above, the goal of the interpolation scheme is to transform the given aerodynamic data to a finer mesh. In order to do so, the interpolation scheme must first discretize the provided mesh to the desired resolution, and then compute the aerodynamic loading at the new nodes. A detailed explanation of how the interpolation scheme performs these steps is provided below.

[1]Note that this function would only be defined at the points which lie on the surface of the aileron.

Discretization of mesh The first step in the interpolation procedure is to determine the location of the new nodes based on the required spanwise and chordwise resolutions. The distribution of the nodes is inherited from the provided aerodynamic data, which makes use of a non-uniform node distribution with more nodes concentrated towards the boundaries of the grid in both the x and z directions.

The desired resolutions in the spanwise and chordwise directions (r_{span} and r_{chord}, with units [mm]) are provided by the user as inputs to the program. Then, the required number of points *per spline* in both the x (n_x) and z (n_z) directions are computed by ensuring that the resolution requirement is met at the most critical interval, i.e. that with the largest spacing. By doing so, the actual resolution of the interpolated grid (which varies due to the non-uniform distribution of the nodes) will always be *greater or equal* than the resolution provided by the user. The x-coordinates of the new nodes can then be computed by looping through all the splines in the x-direction and generating n_x evenly spaced points at each of them. The z-coordinates of the new nodes are computed in the same way, but using n_z.

Computation of aerodynamic loading The next step is to compute the aerodynamic loading at each of the newly created nodes. Let n_{total_x} and n_{total_z} denote the *total* number of nodes along the x and z axes, respectively. The script first loops through the existing 41 spanwise cross sections. At each cross section, interpolation is performed in the chordwise (z) direction. This results in an intermidiate grid of dimension ($n_{\text{total}_z} \times 41$). Then, the script loops through the newly created n_{total_z} chordwise cross sections. At each of those, interpolation is performed in the spanwise (x) direction. This results in the final interpolated grid, with dimension ($n_{\text{total}_x} \times n_{\text{total}_z}$). In summary, the interpolation of the provided 2D gridded data is performed by performing univariate cubic spline interpolation twice, first along the chord direction and then along the span direction.

Absolute interpolation error Let $\|R(f;x)\|_\infty$ denote the maximum absolute value of the interpolation error. It is known that this error is bounded by [4, p. 44]:

$$\|R(f;x)\|_\infty \leq Ch^4 \|f^{(4)}\|_\infty \tag{3}$$

where C is a constant and h is the node spacing. One might therefore expect the error associated with a cubic spline to behave like h^4. However, in the case of a *natural* cubic spline, the error actually behaves like h^2 as $h \to 0$. Although this is not the optimal error that could be achieved by an interpolation scheme, it is deemed to be acceptable [4]. This is specially the case when considering that due to the design of the interpolation scheme, which implements the possibility of easily altering the mesh spacing, the resolution of the mesh can be made arbitrarily small. In practice, it has been found that resolutions smaller than 0.75 mm significantly slow down the computations in the numerical model. Thus, the resolutions used in the computation are $r_{\text{chord}} = 1\,\text{mm}$ and $r_{\text{chord}} = 2\,\text{mm}$. The spanwise resolution is chosen to be larger since the gradient of the aerodynamic loading are less severe along that direction (see Figure 4), and it significantly speeds up the run time of the numerical model.

2.4 Numerical integration scheme

After interpolation is conducted on the points given, integration can now commence. Integration is required within the code to calculate the shear center, the shear flows and the integrals required by the Macaulay step functions. Integration is necessary in order to properly assess the components of the numerical model stated above as the shear center. Shear flow calculations require the summation of forces over given surfaces, while the Macaulay step functions require forces acting over the entire aileron, which requires multiple integrations of the aerodynamic loading of the aileron. Also, at all cross-sections, integration must be used to both find the total aerodynamic load q as well as find the relevant deflections, bending moments and twist. The aerodynamic load is found for every cross-section in y-z plane over the entire span. A point load can then replace the distributed aerodynamic load for each cross section.

One-Dimensional Closed Newton-Cotes Method (s=2) The one-dimensional Newton-Cotes method (s=2) was used to simplify the calculations required to integrate the distributed aerodynamic load across the aileron, which, in turn lowers the amount of time to calculate the forces acting in locations across the entire aileron since integration is required for the most time-intensive sections of the code such as the shear center and shear flow calculations. This method is applied between the given points, whether it be along the span-wise or chord-wise directions. This trapezoid method is calculated as such:

$$\sum_{i=1}^{n} (x_i - x_{i-1}) \frac{[f(x_{i-1}) + f(x_i)]}{2} \tag{4}$$

Error of the One-Dimensional Closed Newton-Cotes Method (s=2) The error of the one-dimensional trapezoid rule would be $E(f:a,b) = O(h^3)$ [4]. This method of integration would only work for single and double integrals, so it could only replace those analytical integrations in case they do not provide viable solutions.

2.5 Loading and Stress Computations

This section presents the procedure used to compute all the necessary loads and stresses in the structure. These methods were already addressed in [1]. The various loading cases that must be taken into account are the bending due to the boundary conditions, the bending generated by the aerodynamic loading, and the torsion generated by the actuator force P and relevant reaction forces. Note the reaction forces generate internal torques, shear forces, and moments that propagate throughout the structure; these must also be taken into account.

2.5.1 Geometric Properties

Before the loading on the aileron can be analysed, relevant section properties must be computed. First, the structure is idealised, as described by assumption AS-NM-10 (see section 2.1). Only the stiffeners are assumed to be point areas, but the skin still carries stresses. A schematic representation of the idealised structure is shown below in Figure 5:

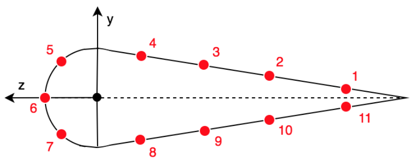


Figure 5: Idealised Cross-Section

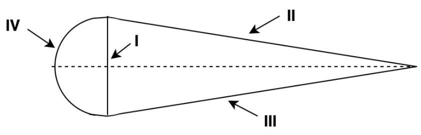


Figure 6: Cross-section divided into 4 parts to simplify moment of inertia calculations. Part I is the spar, parts II and III are the diagonal parts of the skin (a) and part IV is the semicircular part of the cross-section

First of all, the coordinates of the centroid of the cross-section must be determined. As the cross-section is symmetric along the z-axis, the centroid will lie along that axis and thus the y-coordinate is equal to 0. The z-coordinate can be calculated by using $\bar{z} = \frac{\sum A_i \cdot d_{z_i}}{\sum A_i}$ with i being each of the four parts (see Figure 6) plus the stiffeners. This yields a value equal to -0.123 m, approximately.

When accounting for the moment of inertia contribution by the stiffeners, the spacing between them must first be determined. As the stiffeners are equally spaced along the cross-section, the perimeter must be divided into 11 parts.For stiffeners 1-4 and 8-11 (see Figure 5), the y and z coordinates can be obtained by applying the sine and cosine rule. Recalling assumption AS-NM-10, the stiffeners are assumed to be point areas and thus, their moment of inertia contribution comes directly from the Steiner term (Equations 5 and 6).

$$I_{zz,st} = \sum A_{st} \cdot d^2_{y,st,i} \qquad (5) \qquad\qquad I_{yy,st} = \sum A_{st} \cdot d^2_{z,st,i} \qquad (6)$$

For the cross-section's moment of inertia calculation, the cross-section will be divided into four parts (see Figure 6). The moment of inertia contributions by part I are easily computed by applying the formulae for thin-walled beams (Equations 7 and 8). Moreover, the Steiner term that translates the moment of inertia from the spar's centroid to the cross-section's centroid must also be taken into account.

$$I_{zz,I} = \frac{1}{12} \cdot t_{sk} \cdot h_a^3 + A_I \cdot d^2_{z,I} \qquad (7) \qquad\qquad I_{yy,I} = \frac{1}{12} \cdot h_a \cdot t_{sk}^3 \qquad (8)$$

The moments of inertia for part II will yield the same values as for part III, as the distances from the centroid of a to the centroid of the cross-section will be squared. The value can be obtained by using the standard formula for thin-walled beams at an angle α. Moreover, the Steiner term must be added both for $I_{zz,II-III}$ and $I_{yy,II-III}$.

$$I_{zz,II-III} = \frac{1}{12} \cdot t_{sk} \cdot a^3 \cdot sin^2(\alpha) + t_{sk} \cdot a \cdot (\frac{a}{2} \cdot sin(\alpha))^2 \qquad I_{yy,II-III} = \frac{1}{12} \cdot t_{sk} \cdot a^3 \cdot cos^2(\alpha) + t_{sk} \cdot a \cdot (\frac{a}{2} \cdot cos(\alpha))^2$$

a and α can be calculated by using the following equations:

$$a = \sqrt{\left(\frac{h_a}{2}\right)^2 + \left(C_a - \frac{h_a}{2}\right)^2} \qquad \text{and} \qquad \alpha = arctan\left(\frac{h_a/2}{C_a - h_a/2}\right)$$

Finally, part IV must be integrated from 0 to π in order to get the moments of inertia about the hinge line. However, as the Steiner term accounts for the translation from the centroid of part IV to the centroid of the cross-section, the Steiner term from the hinge line to the centroid of the thin-walled semicircle must be substracted from the integral when calculating $I_{yy,IV}$. Once the moment of inertia about the centroid of the semicircle is obtained, the Steiner term that translates it to the centroid of the cross-section can be added. This can be clearly seen in Equations 9 and 10.

$$I_{zz,IV} = \int_0^\pi t \cdot (r \cdot cos\theta)^2 \cdot r d\theta \tag{9}$$

$$I_{yy,IV} = \int_0^\pi t \cdot (r \cdot sin\theta)^2 \cdot r d\theta - A_{IV} \cdot \left(\frac{2 \cdot (r - t_{sk}}{\pi}\right)^2 + A_{IV} \cdot \left(\bar{z} - \frac{2 \cdot (r - t_{sk})}{\pi}\right)^2 \tag{10}$$

The total moments of inertia can then be calculated by:

$$I_{zz} = I_{zz,st} + I_{zz,I} + 2 \cdot I_{zz,II-III} + I_{zz,IV} \qquad I_{yy} = I_{yy,st} + I_{yy,I} + 2 \cdot I_{yy,II-III} + I_{yy,IV}$$

The final values for the moments of inertia are $I_{zz} \approx 4.75385 \cdot 10^{-6}$ m^4 and $I_{yy} \approx 4.58957 \cdot 10^{-5}$ m^4.

The shear centre of the cross-section will be located on the symmetry line. Therefore, its y coordinate $\hat{y}$ is equal to zero. The z-coordinate of the shear centre, $\hat{z}$, can be obtained by calculating the sum of moments about the centroid of the structure. The equation used to obtain the location of the shear centre along the z-axis can be seen in equation 11, where ρ is the radius of rotation and q_{bi} is the open section shear flow of each wall.

$$-V_y \cdot \xi = \sum_{i=1}^{n} \oint \rho \cdot q_{bi} ds + \sum_{i=1}^{n} 2 \cdot A_i \cdot q_{s0_i} [5] \quad q_b(s) = -\frac{1}{I_{zz}}(\int_0^s tyds + \sum A_{st} \cdot d_{y,st,i}) + q_{b0} \tag{11}$$

A_{st} is the point area of the stiffener and $d_{y,st,i}$ is its y-coordinate. q_{s0i} in equation 11 is the correction shear flow of each cell of the beam and can be calculated using equation $q_{s0} = \frac{\oint q_b/(G \cdot t) ds}{\oint ds/(G \cdot t)}$ [5].

Another geometrical property to be found is the torsional stiffness J. The torsional stiffness of the aileron's cross-section can be calculated using the system of equations. The cross-section can be modelled as a multi-cell structure consisting of two enclosed areas (semi-circular and triangular area denoted by A_{circ} and A_{tr}, respectively). If the unit torque T acts at the cross section, there are two shear flows in each cell - q_{s0} and q_{s1}. Then the total torque can be expressed using equation 12. Secondly, the twist $\frac{d\theta}{dz}$ should be the same in both cells, which is expressed in equation 13.

$$T = 2A_{circ}q_{s0} + 2A_{tr}q_{s1} \tag{12}$$

$$\frac{1}{2A_{circ}} \oint \frac{q}{t} ds = \frac{1}{2A_{tr}} \oint \frac{q}{t} ds \tag{13}$$

For this analysis, the stiffeners are excluded, since they have small effect on torsional stiffness as described in assumptions. The integral over the length of the different sections give the second equation of the system.

$$\frac{1}{2A_{circ}}(q_{s0}\frac{\pi h_a 2}{t_{sk}} + (q_{s0} - q_{s1})\frac{h_a}{t_{sp}}) = \frac{1}{2A_{tr}}(q_{s1}\frac{2\sqrt{(C_a - 0.5h_a)^2 + (0.5h_a^2)}}{t_{sk}} + (q_{s1} - q_{s0})\frac{h_a}{t_{sp}}) \tag{14}$$

By applying the unit torque (T=1) the system of two equations (Equation 12 and 14) can be solved for q_{s0} and q_{s1}.

Finally, J can be found by $J = \frac{T}{G\frac{d\theta}{dz}}$, which uses the twist calculated in either of two cells using the equation 14. Torque is set to a unit value of 1. The numerical value found for J is $7.7485 \cdot 10^{-6} m^4$.

2.5.2 Loading Distribution

In order to find the reaction forces, deflections and corresponding stresses of a statically indeterminate structure, it is needed to find the loading distribution. The aileron is subject to bending moment around the z axis (M_z) and moment around the y axis (M_y). There is also a torque acting around the x axis, and thus the cross-section of an aileron in y-z plane will be subject to torque at every part of the structure.

At first the force equilibrium must be obtained. Then, by analysing the free body diagram, the moments can be found as a function of x. The formulae for this calculation as well as for shear force calculations can be found in [1], section 2.3.2. A numerical method will be created to solve it numerically by splitting the structure in spanwise direction.

Finally, there is a torque around the x axis, which varies along the x axis. The torque acts around the shear centre of the aileron's cross-section in the y-z plane. To find the torque on the cross-section, the distributed torque (units Nm/m) due to the distributed aerodynamic loading is calculated by integrating along the z axis as shown in [1], section 2.3.2.

Reaction forces R_{1_z}, R_{2_z} and R_{3_z} act along the symmetry plane of the aileron, so they will not cause torsion around the x axis. Their computation is described in section 2.5.4.

Using the shear forces, the shear flows can be found at the y-z plane of the aileron. Resultant shear flow q_s can be found afterwards, which has contributions due to both torque acting around the x axis and shear forces S_y and S_z acting on the structure. The torque and shear forces are known at the y-z plane as a function of x. Thus, shear flows can be found at each cross-section. An effective method to do this is to cut the structure at the neutral line, which also is the symmetry line of the cross section. The aileron's cross-section is a multi-cell thin walled structure, which needs to be taken into account when analysing shear flows. Consequently, shear stress on the structure can be found since the skin and spar thicknesses are known.

2.5.3 Moment Curvature Relationships

The moment and torque equations have been obtained and explained in subsection 2.5.2. Moment curvature relationships help finding deflections and twist angle at different parts of the aileron and to solve the statically indeterminate system by applying the boundary conditions. As the wing is bent, aileron is also deflected along its spanwise direction. The deflection in y direction due to M_z and the deflection in z direction due to M_y can be found using the equations discussed in [1], section 2.3.3.

On the hinge line, at $x = x_1$, $x = x_2$ and $x = x_3$ there is no deflection in z direction due to the assumption that hinges resist movement along z axis. Again, there is a small contribution to these displacements due to twist.

The torque around x axis is causing twist. To calculate the angular twist, $\frac{d\theta}{dx} = \frac{T}{GJ}$ is used which relates twist to torque. Hence, the twist can be found as a function of x using the equation for θ in [1], section 2.3.3.

2.5.4 Solving for Reaction Forces

As mentioned in the simulation plan report, the system has 8 unknowns (reaction forces acting on the structure as shown in Figure 1, knowing that $R_{2,x} = 0$ after performing force equilibrium along the x direction). However, the moment curvature equations have introduced 5 more unknowns as integration constants. Hence, 13 equations are needed to solve the system [1]. For the moment and shear equations the boundary conditions are the following:
$M_y(L_a) = 0$, $M_z(L_a) = 0$, $T(L_a) = 0$, $S_y(L_a) = 0$ and $S_z(L_a) = 0$.

On top of these 5 equations, the moment curvature relations add additional 7 equations by introducing 7 boundary conditions. The vertical displacements in y direction at $x = x_1$, $x = x_2$ and $x = x_3$ are correspondingly $d_1 cos(\theta_{max})$, 0 and $d_3 cos(\theta_{max})$. The displacements in z direction at $x = x_1$, $x = x_2$ and $x = x_3$ are all 0 in wing coordinate frame, but in the established aileron co-rotated frame they are $d_1 sin(\theta_{max})$, 0 and $d_3 sin(\theta_{max})$. Finally, the deflection in z direction in static wing coordinate system is 0 at the point where the actuator I is connected to the aileron at $x = x_2 - \frac{xa}{2}$, since the actuator I is jammed. These displacements in y and z directions are due to both the displacement due to bending moment and also due to twist, since the hinge line is not on the shear centre.

The 7 boundary conditions then are:

- $\nu(x_1) + \theta(x_1)(\hat{z} - 0) = d_1 cos(\theta_{max})$
- $\nu(x_2) + \theta(x_2)(\hat{z} - 0) = 0$
- $\nu(x_3) + \theta(x_3)(\hat{z} - 0) = d_3 cos(\theta_{max})$
- $w(x_1) + \theta(x_1)(\frac{h_a}{2}) = -d_1 sin(\theta_{max})$
- $w(x_2) + \theta(x_2)(\frac{h_a}{2}) = 0$
- $w(x_3) + \theta(x_3)(\frac{h_a}{2}) = -d_3 sin(\theta_{max})$
- $[w(x_2 - \frac{x_a}{2} + \theta(x_2 - \frac{x_a}{2})\frac{h_a}{2}]cos(theta_{max}) + [v(x_2 - \frac{x_a}{2}) - \theta(x_2 - \frac{x_a}{2})(0 - \hat{z} + \frac{h_a}{2}]sin(\theta_{max}) = 0$

The small angle approximation is used to obtain these boundary conditions. The term $\hat{z}$-0 is the distance from the hinge line to the shear centre along z axis and $\hat{z}$ is negative.

Another equation, which is added to matrix equation is Equation 15. It is known that force at actuator I acts parallel to force P, however P acts along z axis in the wing frame, hence it makes an angle θ_{max} with the z axis in the co-rotated aileron coordinate frame.

$$A_y = tan(\theta_{max})A_z \tag{15}$$

Therefore, 13 equations have been obtained to solve for the 13 unknowns to find the reaction forces acting on the aileron. Therefore, the system can be solved using a matrix equation.

2.5.5 Stress Computations

The final step in the computation process is to determine the stresses in the structure. Firstly, the found shear flows are divided by the thickness to obtain the shear stress distributions. Then, the direct stress distribution is computed by calculating the stresses generated by M_y and M_z using the following formula:

$$\sigma_{xx} = M_y \frac{z - \hat{z}}{I_{yy}} + M_z \frac{y}{I_{zz}} \tag{16}$$

The remaining stresses, σ_{yy} and σ_{zz}, will be computed by modifying the equation above to work for the respective axes. Lastly, the Von Mises stress is used to calculate the maximum stress in any cross section, and thus the maximum stress at any point in the aileron. The Von Mises stress is a helpful concept that allows the computation of an equivalent stress for a structure that is subjected to multiaxial loading, such as the aileron under consideration. The Von Mises stress is computed according to Equation 17 [6]:

$$\sigma_{\text{vm}} = \sqrt{\frac{(\sigma_{xx} - \sigma_{yy})^2 + (\sigma_{zz} - \sigma_{xx})^2 + (\sigma_{zz} - \sigma_{xx})^2 + 6(\tau_{xy}^2 + \tau_{yz}^2 + \tau_{xz}^2)}{2}} \tag{17}$$

2.6 Results

This section describes the results of the numerical model. The outputs of the simulation are the deflection of the hingeline, twist of the aileron and the maximum stress experienced in the structure, based on the distribution of the Von Mises stress.

The deflection of the hingeline can be seen in figures 7 and 8. At the root of the aileron (x=0) the deflection in y-direction is 4.9mm, it becomes zero at hinge 2 and at the tip the deflection is 1.29cm. In z-direction, the deflection goes from −2.8mm at the root to −7.4mm at the tip, with a peak of $7e - 5$mm at $x = 0.57$m. The twist as a function of x is visualised in Figure 9. At the root the twist is nearly constant. There are two kinks in the curve at both actuators.

The maximum stress in the aileron is 0.565GPa. It is experienced at a span location of 49.9cm and a chordwise location of 42.5cm. These results will be discussed further and verified in section 3.

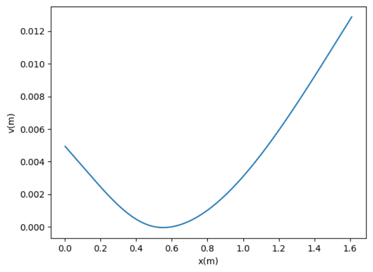


Figure 7: Hinge line deflection in y direction as a function of x

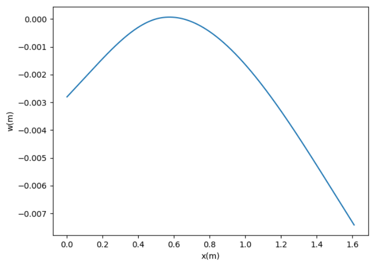


Figure 8: Hinge line deflection in z direction as a function of x

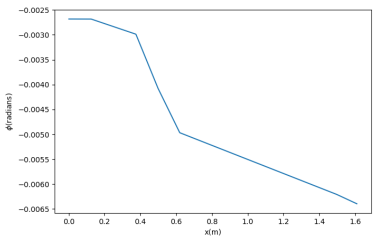


Figure 9: Twist of the aileron as a function of x

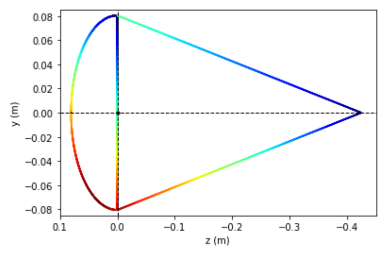


Figure 10: Numerical Model Results: Direct Stress Distribution Along the Cross-Section at x = 0.5.

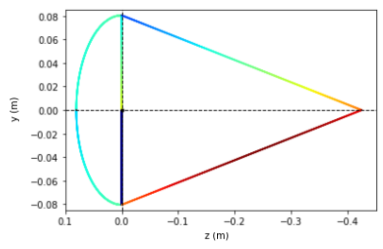


Figure 11: Numerical Model Results: Shear Flow Distribution Along the Cross-Section at x = 0.5.

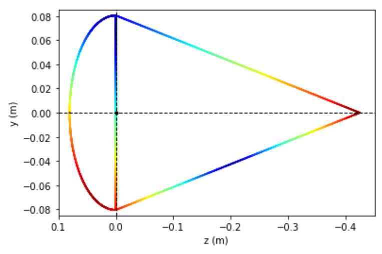


Figure 12: Numerical Model Results: Von Mises Stress Distribution Along the Cross-Section at x = 0.5.

3 Verification

In simple terms, verification is a process through which unexpected bugs or errors in a piece of software can be identified. Within the context of the present document, verification can be thought of as the process through which the numerical model is compared to other simulations in order to ensure the code is performing its intended function. Although verifying the code does not ensure that its results will be an accurate representation of real-life phenomena (see Chapter 4), it is a vital step in the design process of any software-based simulation.

This Chapter is structured as follows. Section 3.1 describes the model used to verify the numerical simulation presented in Chapter 2. Then, verification is performed at two increasing levels of complexity: Section 3.2 presents the verification tests performed at the unit level, while Section 3.3 contains the verification tests performed at the system level. In addition to a description of the used tests both sections include the results obtained from the tests, and (when applicable) a discussion on potential sources for the discrepancies found between the numerical and the verification model.

3.1 Model Description and Assumptions

The provided verification model consists of four parts, as mentioned in [1]. The first part takes care of the bending stiffness calculations. It calculates the centroid and the moment of inertia around the y and z axes. The second part computes the torsional stiffness, by computing the shear centre using the concept of shear flow distributions. In the third part, the vertical (y-axis) deflection of the aileron is calculated using the Rayleigh-Ritz method in combination with the appropriate boundary conditions. This method is based on the principle of stationary potential energy contained in a structure. The deflection of the aileron is computed by finding the deflections at each of the discretised points such that the total potential energy of the structure, Π, is minimised. A minimum value of Π represents that the structure is in equilibrium. The Rayleigh-Ritz method makes use of a linear combination of a set of basis function to compute Π. This is then differentiated with respect to the weights of the basis functions and set equal to zero in order to find the minimum. The equations used in order to do so can be found in [7].

The method sums the total strain energy and work potential of the structure, including the contributions of torque distribution and shear lateral deflection.

The last part of the verification model computes all the relevant stresses. First, the shear stress distribution is computed by diving the computer shear flow distributions by the skin thickness. Then, the direct stress distribution caused by bending around the y and z axes is computed. Lastly, these two stress distributions are used to compute the Von Mises stress [7].

The use of the model described above for verification of the numerical model presented in Chapter 2 is justified for various reasons. These are summarised below:

- The two airfoils have approximately the same geometry. They are symmetric and are made of a semi-circular leading edge, a spar and a triangular trailing edge.
- The ailerons are subjected to the same loading conditions. They both experience bending, torsion and shear.
- The models use similar assumptions, for example the idealisation of the stiffeners as point masses, beam theory and linear stresses (see next section). Similar assumptions introduce similar errors, under- or

overestimating stresses, thus the models should also give similar results.

- The verification model makes use of an energy method in order to compute the deflection of the aileron, namely the Rayleigh-Ritz method. Since the numerical model presented in this document does not make use of such method, agreement between the two models would be strong evidence that the presented model in this document is functioning as intended.

Lastly, the assumptions made in the verification model are described below:

- **AS-VE-01:** The model assumes every part of the structure to be thin-walled. Consequently, terms of order $\mathcal{O}(\mathrm{t}^2)$ or higher are neglected in the calculations.
- **AS-VE-02:** The model idealises the cross sections of the aileron. Consequently, the stringers are represented by point areas in the form of booms. Furthermore, the stringers and spar caps carry only direct stresses, while the skin carries all the shear flows (in addition to direct stresses).
- **AS-VE-03:** The model assumes the resultant vertical shear force to act at a fixed distance from the middle of the spar.
- **AS-VE-04:** The aileron can be modelled as a beam, so beam theory applies and its formulas can be used.
- **AS-VE-05:** Lastly, the stresses are assumed to be linear. Thus they can be superimposed.

3.2 Verification: Unit Tests

This section presents the results obtained from verification testing at the unit level. For each unit test a description of the test is provided, in addition to the obtained results and potential sources of discrepancies between the numerical and verification models.

In the context of software verification, a unit is defined as any function or small, closed computation loop that can be tested individually. Testing at the unit level must be performed before more complex tests (such as system tests) in order to ensure that each individual piece of code is performing its intended function correctly. Doing this allows one to identify potentially problematic units of the code in a controlled manner, without having to consider the complex interactions between the different units in the overall system.

Note that, when applicable, the presented unit tests are performed in *both* the numerical model and the verification model in order to then compare the results. However, certain unit tests (such as that for the interpolation scheme) can only be performed in the numerical model. In those instances, the verification of the particular unit under consideration is achieved by critically inspecting the results obtained from the unit test in the numerical model *only*, rather than by comparing the numerical model to the verification model.

Lastly, note that the unit tests presented below are structured according to the flowchart of the numerical model, which can be found in Appendix B. Each block in the flowchart contains an identifier code; for instance, **[AL 1]** represents the first block of the Aerodynamic Loading module of the flowchart. Each unit test presented below specifies, using the assigned identifier codes, which blocks have been verified by that particular test.

3.2.1 Unit I — Centroid (Block: SP1)

Test 1 The stiffener locations are verified both graphically and numerically. By plotting the cross-section both on the numerical model and on the verification model, the point areas are found to be located at the same place. Also, the stiffener coordinates on the verification model coincide with the ones from the numerical model. The error between the values computed by the two different models differ by 10^{-7}% to 10^{-5}%. These very small differences occur because in the verification model, the z and y coordinates are approximated to six decimals, while in the numerical model the values go until the 18^{th} decimal point. Although the accuracy of this test is not the highest, as the error is practically equal to zero, these two tests will suffice to verify the stiffener locations.

Test 2 The coordinates for the centroid of the structure obtained in the numerical model are numerically verified by comparing them to the ones in the verification model. The values differ by 0.66165% and this value is assumed to be negligible. Moreover, it is logical that the centroid is located slightly to the left in the cross-section as more area is concentrated on the left part.

3.2.2 Unit 2 — Moments of Inertia and Torsional Stiffness(Block: SP2)

Test 1 The moments of inertia are numerically verified. I_{zz} is exactly the same in both models with an error of 0%. However, I_{yy} has an error of 0.10405%, which can be explained by the slight difference on the z-coordinate of the centroid. As this value is used when calculating the steiner term about the y-axis and this distance is squared, it might lead to small variations of the total I_{yy} value.

Test 2 The verification model gives access to intermediate results of its calculations, for example the torsional stiffness J. The torsional stiffness value given by the numerical model is identical to the one in the verification model (meaning an accuracy of 100%). This is due to similar calculation methods used in both models. Both models exclude stiffeners in torsional stiffness analysis.

3.2.3 Unit 3 — Shear Centre (Block: SP3)

Test 1 To calculate the shear centre the shear flow distributions needed to be calculated. These were plotted and tested for consistency. This was done by analysing the direction of the shear flows. As well as the start and end point shear flows of each section, which should be the same for adjacent sections.

Test 2 The shear centre found has an error of 1.3% compared to the verification model. This can be explained by the methods used. The shear centre was found fully numerically whereas the base shear flows in the verification model were found analytically. The error accompanied with the numerical integration method is the same order as the total error found in the integration scheme.

3.2.4 Unit 4 — Interpolation Scheme (Blocks: AL1, AL2.1, AL2.2, AL3 and AL5)

This unit is composed on the numerical interpolation scheme. An explanation of the used interpolation scheme can be found in section 2.3.

Test 1 The first unit test for the interpolation scheme consists of a visual inspection of the provided aerodynamic data and the resulting interpolants. Two cross sections (one along the span and one along the chord) are selected at random. At each of those cross sections, the provided aerodynamic data and the interpolants of that data are plotted. Figure 13 depicts the cross section at $z = 0.05\,\text{m}$, while Figure 14 depicts the cross section at $x = 1.11\,\text{m}$. In both figures, the original aerodynamic data is shown with red dots, while the computed interpolants (i.e. cubic splines) are shown as blue lines. It can be seen in both figures that the interpolants are well-behaved: they interpolate the required points, and they show no signs of suffering from high-frequency oscillations near the boundaries (Runge phenomenon) or in between any other nodes like is characteristic for standard, single-polynomial interpolation for large n. Thus, this unit test confirms that the interpolations scheme is capable of accurately interpolating the provided aerodynamic data.

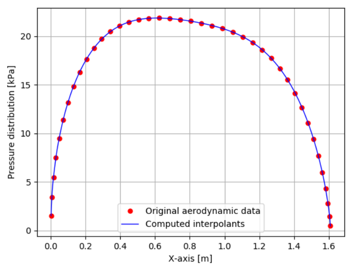


Figure 13: Plot of the provided aerodynamic data and the resulting interpolants for a chordwise cross section ($z = 0.05\,\text{m}$)

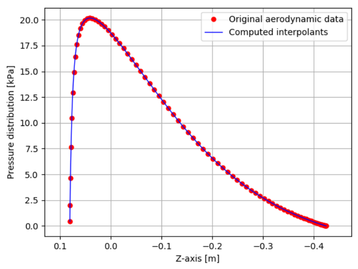


Figure 14: Plot of the provided aerodynamic data and the resulting interpolants for a spanwise cross section ($x = 1.11\,\text{m}$)

Test 2 The second unit test for the interpolation scheme is another visual comparison of the interpolation results. However, this time instead of interpolating the provided aerodynamic data, the interpolation scheme was used to interpolate sampled data from the Heaviside step function, also known as the unit step function. The same interpolation was performed twice, each time with a different value for the number of nodes per spline. This unit test was conducted for the following reasons:

- It is useful to check how the interpolation scheme performs when provided with data that is different than the aerodynamic loading data. By doing this one can verify that the interpolation scheme works in the general case, and not only when interpolating the provided data.
- The Heaviside step function has an infinitely high gradient at the point $x = 0$. By interpolating the Heaviside function on the interval $[-2, 2]$, this unit test ensures that the interpolation scheme is well behaved even in locations where the gradient of provided data is extremely high.
- Lastly, by performing the same interpolation with two different values of the number of nodes per spline, the effect of varying the mesh spacing can be observed.

Numpy's function *np.heaviside()* was used to compute the value of the Heaviside step function at 9 evenly spaced nodes in the interval $[-2, 2]$. Such a small number of nodes is chosen so that the behavior of the interpolant in between nodes is visible in the graphs. Figure 15 shows the interpolation with $n_{\text{spline}} = 3$ points per spline, Figure 16 with $n_{\text{spline}} = 50$. From the Figures it can be concluded that (a) the interpolation scheme is capable of accurately interpolating data that is different from the provided aerodynamic data, (b) the interpolation scheme is well-behaved in regions with high gradients, and (c) as the mesh spacing is refined (i.e. the number of points per spline is increased), the resultant interpolants become smoother. While this is to be expected, this unit test provides proof that the interpolation scheme produces a global interpolant that is smooth. This is of vital importance, since the interpolation scheme is used to approximate the underlying aerodynamic distribution, which tends to be smooth in reality.

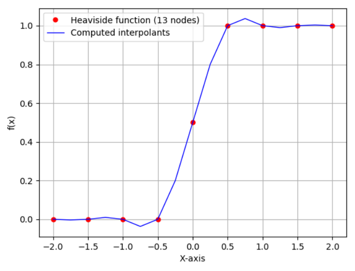


Figure 15: Interpolation of the Heaviside step function on the interval $[-2, 2]$ with 3 points per spline.

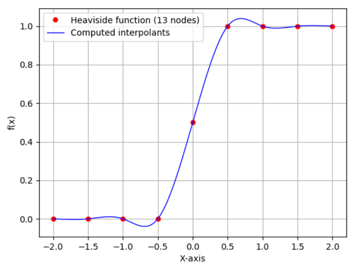


Figure 16: Interpolation of the Heaviside step function on the interval $[-2, 2]$ with 50 points per spline.

3.2.5 Unit 5 — Calculation of point loads from pressure distributions (Blocks: AL4.1 and AL4.2)

This unit deals with the conversion of the provided aerodynamic data (pressure distribution at all the nodes in [Pa]) to point loads at all the nodes (in [N]). The numerical model achieves this by multiplying the value of the provided pressure distribution at each point by the corresponding steps in the x and z directions). Multiplying each value by $\delta_{\text{A}} := \delta_{\text{x}}\delta_{\text{z}}$ converts the value to a force.

- check graphs – should match aero loading - sanity check – order of magnitude of total resultant force (sum of aeroloads) should match A*np.mean()

Test 1 The first test that can be conducted is a sanity check for the order of magnitude of the point loads. Summing all the point loads yields an estimate for the total load across the entire aileron surface. This can be compared to the value obtained by taking the average of the provided aerodynamic data (p_{av}) and multiply that by the area of the aileron surface, namely $\text{A}_{\text{aileron}} = \text{C}_a \cdot \text{L}_a$. The obtained results are summarized below:

$$\text{p}_{\text{av}} \cdot \text{A}_{\text{aileron}} = 4.747\,\text{kN} \qquad \sum_{i,j} f_{i,j} = 6.149\,\text{kN}$$

As expected, both results have the same order of magnitude. Thus, the result of this unit test indicates that the aerodynamic pressure distribution has been correctly converted to point loads.

Test 2 Another way of ensuring that the aerodynamic point loads have been calculated correctly is via visual inspection. Figures 17 and 18 show the summation of the calculated point forces along the spanwise and chordwise directions, respectively. At each node, the value shown in the figures is computed by summing all of the point loads at that cross section.

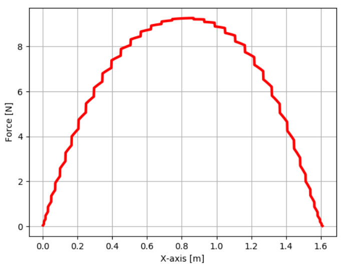


Figure 17: Spanwise distribution of the sum of point forces (along the z-axis) at each node in the x direction.

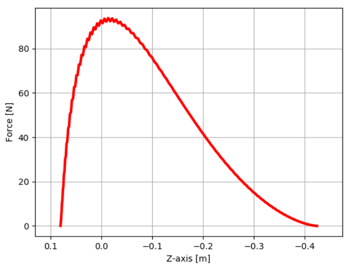


Figure 18: Chordwise distribution of the sum of point forces (along the x-axis) at each node in the z direction.

The magnitudes are correct in comparison with a graph showing the distribution of pressure across the aileron shown in 4.

It can be seen that Figures 17 and 18 very closely resemble the provided aerodynamic pressure distribution data, which can be seen in Figures 13 and 14. This is yet another indication that the point loads have been computed correctly.

3.2.6 Unit 6 — Integration Scheme (Blocks: RL1)

Test 1 Verifying the integration scheme can be done analytically. For example, the integral of $sin(x)$ between 0 and $\pi/2$ is known as 1. With the trapezoidal rule implemented in the model, using 100 trapezoids the numerical answer is 0.99979. This method has high accuracy because it is known that the analytical solution is for 100% correct.

Test 2 Since the trapezoidal rule approximates the function linearly to calculate the underlying area, the rule should approximate the integral of linear functions exactly. For example the integral of x from 0 to 1 is 1/2, which is indeed exact.

Test 3 The order of convergence for a decreasing step size (increasing number of trapezoids) for the trapezoidal rule is $O(\Delta x^2)$. The error can be plotted with respect to the step size. The error reduces with a factor 2 for decreasing step size. This gives an order of convergence of 2, what was expected.

3.2.7 Unit 7 — Reaction Forces (Block: RL 2)

Test 1 To verify the moment and shear force equations, it is helpful to model a simplified load case and evaluate if the equations are still producing the correct results. The aileron is analysed as a beam structure. Therefore, to test the moment and shear equations a simplified load case is made regarding beam structure by changing the inputs of the problem. To analyse the equations, the beam should be in static equilibrium, therefore the simplified load case is set to be symmetrical, where the hinges are located in equal distances from each other. E.g. - the length of the beam is 10m, the hinge one is at $x = 2.5m$, hinge two at $x = 5m$ and hinge 3 at $x = 7.5m$ and $x_a = 1m$. The reaction forces are symmetrical around hinge 2, their values can be given as $R_{1y}, R_{3y} = 10N$, $R_{2y} = 5N$, $A_y, P_y = 12.5N$ (reaction forces at hinges are acting in positive y direction, A_z and P_z in the negative y direction. The values in z direction can be kept the same for the respective reaction forces, when analysing moment around y axis and shear along z axis. In the matrix equation, the boundary conditions suggest that at $x = L$, both moments around z and y axes and shear forces along y and z axes are 0. In the simplified loading case this should be the case as well. The simplified load case verifies these conditions, however - this excludes verification of the contribution from the aerodynamic loading.

Test 2 The aileron is in equilibrium, this means the sum of forces (visualised in the free body diagram, figure 1) should be zero for each axis. For this model the sum of forces in x is 0, in y 1.82e-12 and in z 2.91e-11. These lie very close to zero, which means the calculation matrix functions correctly. The slight offset can be explained by the interpolation and integration of the aerodynamic loads.

3.2.8 Unit 8 —Internal Loading Distribution (Block: RL3)

Test 1 Invalid inputs can be given to functions and they should give an error. For example Macaulay functions do not work for a negative value of x. This way during the verification process, it was found that Macaulay step function was not showing the desired results at situation when it resulted in value 0 to the power of 0. Mathematically in Python the result of this operation is 1, but the desired output is 0 since the Macaulay step function is disabled when it's value is smaller than 0, it must not be set to 1.

Test 3 The solution matrix consisting of boundary conditions should not be singular in order to solve the system. It can be checked by looking at the rank of the matrix or making sure that it's determinant is not 0. None of the columns can be dependant.

Test 4 To verify the deflections of the aileron, the numerical model needs to be verified step by step. Therefore, M_z and S_y are compared with the verification model. The results of the numerical model are plotted in Figures 19 and 20.

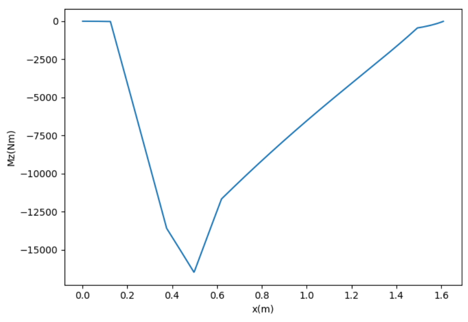


Figure 19: Moment around z axis as a function of x

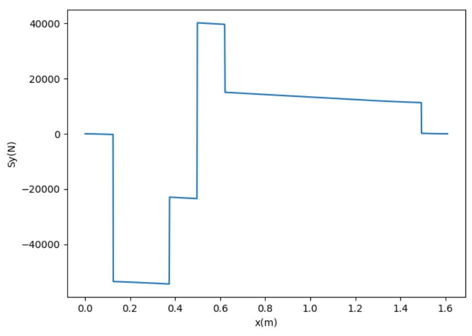


Figure 20: Shear force in y direction as a function of x.

The results plotted can be compared with the verification results shown in Figure 37. As can be seen, the moment around z axis is zero at the edges of the aileron in x direction at it reaches its peak at hinge 2 for both models. A notable difference is in the smoothness of the curves shown. The numerical model uses Macaulay step functions, which is the reason for the sharp peaks at the points, where the point forces are applied. The verification model uses energy method and it includes integrating continuous functions to find the deflections, therefore the sharp peaks do not appear in the verification model.

The largest negative moment for the numerical model is 16468 Nm, while for the verification model 14618 Nm, resulting in 12.5% difference. This is a significant difference which subsequently will be visible in the deflection in y direction. For the shear force in y direction, the maximum negative value is at actuator I. It is 54365N for numerical model, which is 2.5% higher comparing to verification model. Verification model includes fluctuations in the graph, which are not apparent in the numerical model using Macaulay step functions, the slope of the shear graph is only due to the distributed aerodynamic loading applied to the aileron. Notably, the numerical model's shear force has a sudden peak at hinge 2, this might be the result of the different methods used as discussed previously, the sharp peak in relatively short distance is smoother in energy method analysis, while for Macaulay step functions the value changes instantly due to discontinuous functions.

Next, the moment around y axis and shear in z axis are analysed, the results are shown in Figures 21 and 22.

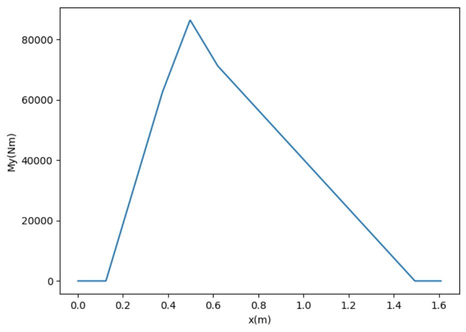


Figure 21: Moment around y axis as a function of x

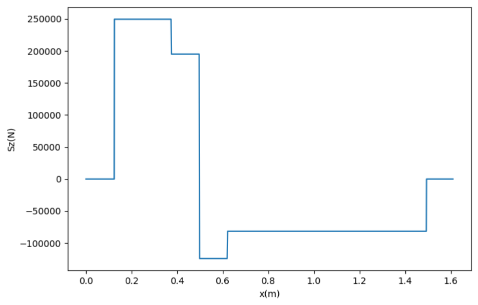


Figure 22: Shear force in z direction as a function of x.

As can be see by comparing these results to the results of the verification model in Figure 38, moment around y axis again is zero at both edges of the aileron in the x direction, reaching its maximum value at hinge 2. Correspondingly, also the shear force in z direction changes its sign at hinge 2 from negative to positive, which is reasonable since shear in z is a derivative of moment around y function. The maximum value for the moment around y axis is lower comparing to the verification model. It reaches a value of 86260 Nm, while the verification model provides the value of 82538 Nm, which is 4.5% lower. This discrepancy might result from the different method used. The discontinuous Macaulay step function generates a peak at the point where the moment is the highest. Furthermore, when increasing the number of basis functions (N) used in Rayleigh-Ritz method in verification model, the differences between the highest values gets smaller. Furthermore, the highest value is reached at lower x coordinate. If N is increased from N=20 to N=40, the difference between maximum moments decreases to 3.1%. The shear force changes its sign at location where the moment is maximum, since the shear force is derivative of the moment equation. The negative value from x=0.5m to x=1.494 at hinge 3 is -81500N for numerical model, which matches with the verification model. Verification model uses the energy method, which introduces fluctuations around a specific value, if the number of basis functions is increased, then the fluctuations get lower in magnitude as accuracy increases.

Finally, before arriving at deflection results, the torque around x axis is analysed as a function of x. The results of the numerical model are shown in Figure 23.

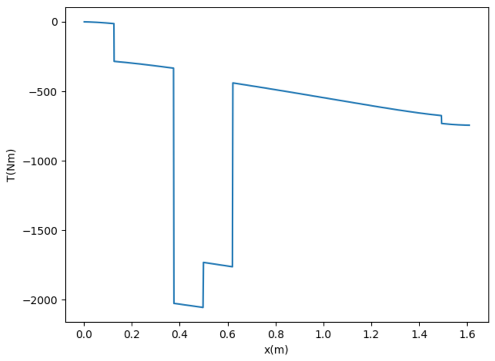


Figure 23: Torque as a function of x

The torque as a function of x shows similar characteristics compared to verification model in Figure 27. As can be seen, in the regions without peaks the absolute value of the torque in the numerical model is slightly increasing due to the distributed torque due to aerodynamic loading. This is the general trend also in the verification model, where the torque s gradually increasing from x=0m to x=0.3m and from x=0.7m. However, due to characteristics of the energy method, the verification model's graph has more fluctuations and the peak is slightly wider around x=0.5m due to continuous functions used in the model. The peak at x=0.5m is of value T=1850Nm for the verification model. At this location the numerical model shows the torque of value 2056N, which is approximately 11% higher. This discrepancy might be due to the discontinuity of Macaulay step function comparing to energy method and due to the discrepancy in the obtained shear centre location. The highest torque is modelled at hinge 2, since at this point the discontinuous Macaulay step function contributes to the torque equation - this is the case also for the verification model if the number of basis functions used is high enough (verified with N=60).

3.3 Verification: System Tests

Lastly, verification will be performed at the system level. This is the highest level of complexity, where the interactions between the modules themselves are to be tested. For this, the results from the verification model will be compared to the results obtained in the numerical model (section 2.6). A total of 15 plots will be compared and the similarities and discrepancies between them will be explained. Moreover, another system test will be to compare the values for the three final outputs.

3.3.1 Results of the Verification Model

As it can be seen in Figure 37, the deflection in the y-direction has a parabolic shape. There is no deflection at approximately x = 0.6 m, which is where actuator II is located. This makes sense as force P is acting through actuator II and thus, no deflection is allowed at this point. Also, the maximum deflection is located at x = 1.6 m, which also makes sense as this is the tip of the aileron. The curve of the slope is the derivative of the deflection and the change in sign might be due to forces A_y and P_y going in different directions. The minimum bending moment about the z-axis is located at x = 0.5 m, which is where hinge 2 is located. The reason for this is that hinge 2 is fixed. Finally, the shear force in the y-direction varies irregularly along the length of the aileron. There is a peak at approximately x = 0.6 m, probably due to the point load P acting on actuator II.

In Figure 38, the same plots as in Figure 37 are presented but in the z-direction instead of y-direction. Again, there is no deflection at the point where actuator II is located. The reasoning for this is the same as in Figure 37. Also, the change in the slope of the deflection is due to reaction forces A_z and P_z having different signs. The bending moment about the y-axis is equal to 0 on the sides of the aileron and it attains a maximum value at the location of hinge 2. Finally, the shear force in the z-direction behaves similarly to the shear force in Figure 37: it varies irregularly along the aileron and there is a peak where at the hinge 2 location with a sudden drop, probably due to force P.

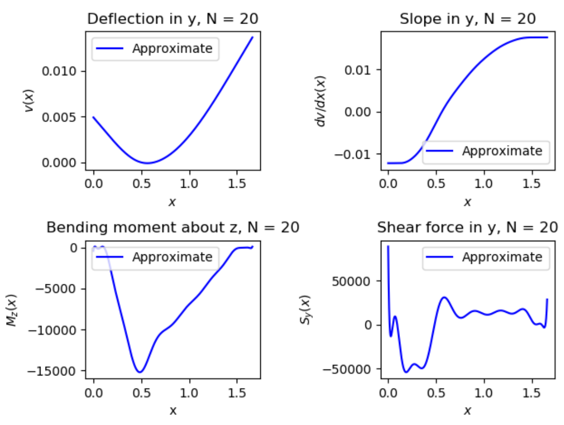


Figure 24: Verification Model Results Using 20 Basis Functions: Deflection, Slope and Shear Force Along the y-Axis and Bending Moment About the z-Axis

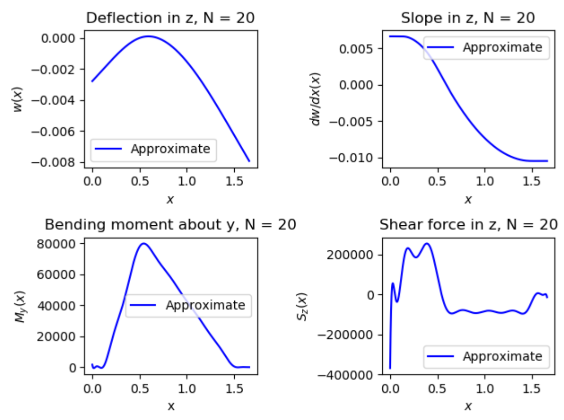


Figure 25: Verification Model Results Using 20 Basis Functions: Deflection, Slope and Shear Force Along the z-Axis and Bending Moment About the y-Axis

The three last plots of loads acting along the aileron can be found in Figure 39. The twist gradually increases throughout the x axis of the aileron. The minimum value is attained at hinge 2, as at this point the aileron is fixed and thus it is restrained to twist. However, the magnitude varies in units of 10^{-3} from the root to the tip of the aileron. Thus, the effect of the twist on the aileron is very small compared to the other loads acting on it. The distributed torque at hinge 2 is equal to zero but the value for the maximum total torque is maximum at this point. Both statements are due to the fact that the aileron is fixed at that point.

The shear flow distribution along each cross-section is plotted in Figure 27. The cross-section is symmetric along the z-axis, and so is the value of the shear flow q. The maximum value of shear flow is obtained on the skin near the spar. This is due to the fact that at this point, the three different parts of the cross-section unite.

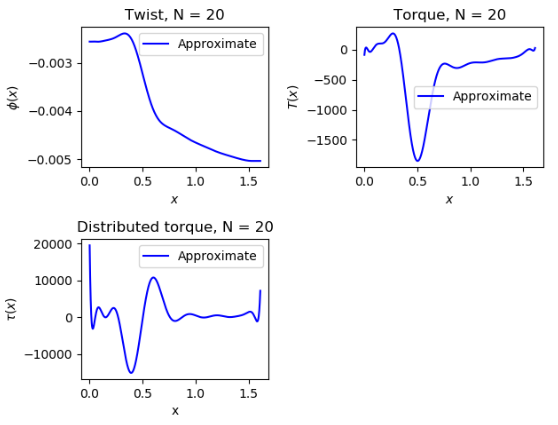


Figure 26: Verification Model Results Using 20 Basis Functions: Twist, Torque and Distributed Torque

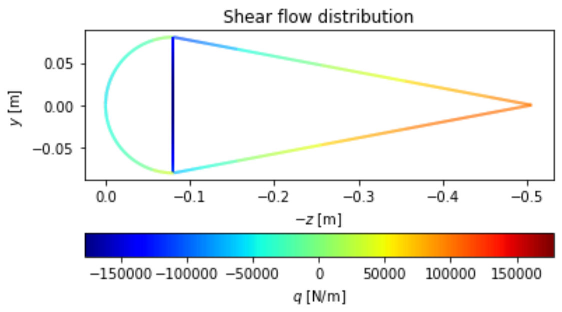


Figure 27: Verification Model Results: Shear Flow Distribution Along the Cross-Section

The distribution of the direct stress along the cross-section is plotted in Figure 28. Positive values for the direct stress are obtained in the leading edge, meaning this part of the aileron is in tension. On the contrary, the trailing edge is in compression and this can be seen in Figure 28 as the values for direct stress turn negative as it approaches the trailing edge.

Finally, the distribution of the Von Mises stress can be found in Figure 29. The values of stress are always positive throughout the cross-section. On the upper part, the maximum stresses occur on the trailing edge, while on the lower part the location of the maximum stress is on the leading edge.

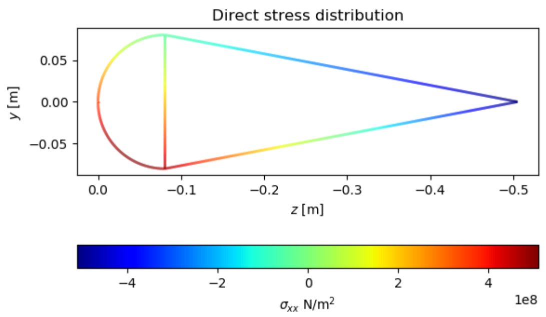


Figure 28: Verification Model Results: Direct Stress Distribution Along the Cross-Section

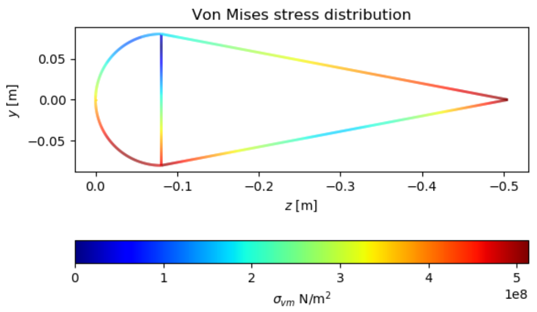


Figure 29: Verification Model Results: Von Mises Stress Distribution Along the Cross-Section

3.3.2 System Tests Results

In this subsection the results between the numerical method (section 2.6) and the verification model (subsection 3.3.1) will be compared. Reasons for the discrepancies between the errors will be also presented. When looking at the different plots, it can be seen that the plots in the verification model (Figures 37, 38 and 39) are usually smoother than the ones in the numerical model (Figures 7, 8 and 9). This is mainly due to the fact that in the numerical model discontinuous Macaulay step functions are used to simulate the behaviour of the different loading cases. However, the verification model is done by implementing energy methods, which integrates continuous functions over the dimensions of the aileron. Moreover, the fluctuations are introduced to to basis functions used. As the number of basis functions is increased, the accuracy increases and the fluctuations get smaller in magnitude approaching a certain value. The location of global maximum in the graphs can get slightly shifted, when the number of basis functions is increased.

Furthermore, it is apparent that twist graph still has peaks and sharp corners, but deflection graphs are smoother. This is due to the fact, that twist is obtained by integrating torque along the x axis, which results in the linear functions of x. However, the slopes of deflections and deflections are obtained by integrating the moment equation once and twice correspondingly. Hence, the slope is a quadratic function of x and deflection is a cubic function of x, this way eliminating the sharp peaks.

Deflection of the Hingeline and its Slope $(v_y, v_y^{'}, w_y, w_y^{'})$
One of the system levels test is to compare the outputs of the numerical model with the verification model. The deflection in y direction as a function of x is plotted in Figure 7. As compared to the verification model's result shown in Figure 37, it can be seen that the deflection in y direction at $x = L_a$ is 0.01287m, which is 0.86% smaller than in the verification model. Both of the models show that the deflection reaches a small negative value at around x=0.6m, therefore reaching a value of zero twice. However, a larger discrepancy can be seen at x=0, where the numerical model's value for deflection is 5% higher comparing to the verification model,which is a difference of 0.24mm. As Unit Test 8 suggests, there is a discrepancy in evaluation of moment around z axis, which also leads to discrepancies in the deflection graph. In overall, the numerical model can compute the deflection in y direction with a reasonable accuracy, however the inaccuracies might come from the incorrectly obtained integration constants and validation might suggest further discrepancies from the modelled loading case.

The deflection in z direction shows results as shown in Figure 8. For this deflection the results by numerical model are much closer to the verification results, since also the moment around y axis was obtained more accurately. The deflection at x=0 is estimated to be -0.00281m, which results in 0.02% difference with the verification model. Moving along the span the deflection reaches zero at $x = 0.498m$, then it is positive and is again zero at $x = 0.66m$, which is also a characteristic of the verification model. Further on, the deflection in z direction follows the same curve as in verification model, its smallest value at the tip is -0.0074m, which is 0.02% higher comparing to the verification model. The small discrepancies can result in different integration schemes used in two models.

A useful system test is to compare the integration constants with the verification model. In the moment curvature relations ([1]) integration constants C_i can be found. Their values should match the respective deflections in the verification model at locations $x = 0$:

- C_1 = -0.01256 (-15.72% difference in $\frac{dv}{dx}$ at x=0)
- C_2 = 0.00495 (5.14% difference in v_y at x=0)
- C_3 = 0.00693 (0.02% difference in $\frac{dw}{dx}$ at x=0)
- C_4 = -0.00281 (0.02% difference in w_z at x=0)
- C_5 = -0.00268 (4.78% difference in $\theta(x)$ at x=0)

The value C_2 shows deflection in y direction at x=0. Similarly, C_4 shows the deflection in z direction at x=0 and C_1 and C_3 show the slope of the deflections in the y and z directions, respectively, at x=0. Finally, C_5 shows the value of the twist at x=0.

As can be seen, the constants C3 and C4 have been obtained accurately. The differences in C2 and C5 are considerable, but the constant C2 suggests that there might be a flaw in the numerical model. The matrix does not provide the correct result for the output C1, because 15% between the numerical and verification model is too high to result from the fact that different methods are used. Since all of the constants are outputs of the matrix equation, the discrepancies of C1 can lead to discrepancies also in twist and deflection in y functions. Supposedly, the mistake might lie in the matrix used to solve the system or the wrongly implemented boundary condition.

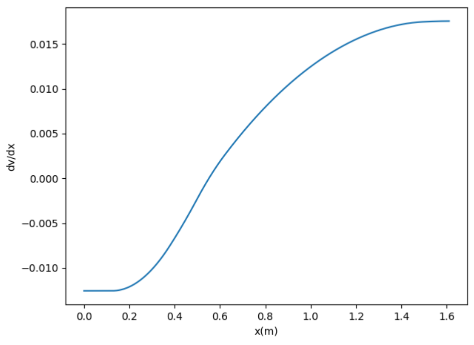


Figure 30: Slope of the deflection in y direction as a function of x

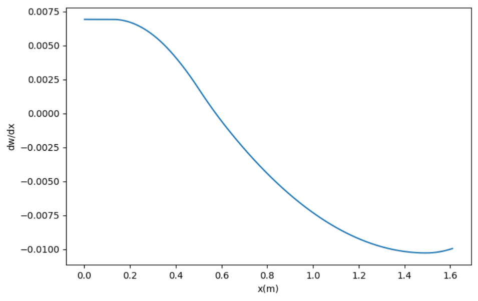


Figure 31: Slope of the deflection in z direction as a function of x.

The slopes of the deflections in y and z directions are shown in the Figures 30 and 31 correspondingly. As mentioned previously, the slope of deflection in y direction at x=0 is considerably different, comparing to the verification model due to the value of C1. This slope reaches the value of zero at x=0.55m, while for the verification model it reaches value of zero at x=0.62m, this corresponds to the deflection graph - this is the point where deflection reaches its smallest numerical value. Subsequently, the last value of the slope of deflection in y direction is around 9% lower than the value in the verification model. These discrepancies mostly result from the fact that integration constants C1 and C2 are not obtained accurately.

The discrepancies are lower for the slope of the deflection in z direction. The integration constant C3 shows the slope at x=0, which has 0.02% difference from the verification model. The slope reaches the value of 0 at x=0.575m, which is the case also for the verification model. At x=0.575 deflection in z has its highest numerical value, which therefore corresponds to the slope of zero. The verification model shows small fluctuations in slope of deflection in z direction at x>1.5m due to the energy method used with 60 basis functions, and it approaches a value of -0.01027m. The value given by numerical model, too. The numerical model predicts the deflection in z direction as it is supposed to do, the verification shows that only small discrepancies can be found, due to different methods and integration schemes used in calculations.

Twist of the Aileron

The twist as shown in Figure 9 is compared to the twist of the verification model shown in Figure 39. Both the numerical model and the verification model suggest that the absolute value of the twist is gradually increasing as moving along the span. After reaching the point of application of actuator II at x=0.6105m, the rate of twist decreases slightly - this can be seen in both models. The numerical model has a sharper peak at that point due to the discontinuous Macaulay step functions. At x=0, the absolute value of twist is equal to 0.00268 radians, which is 4.8% higher than the absolute value in verification model. However, the absolute value of twist in the numerical model gradually increases, while for the verification model at x=0.34m the value is decreasing. Both models suggest that the rate of twist is increasing after that point Furthermore, the maximum twist is considerably different - the largest absolute value of twist in the numerical model at $x = L_a$ is 0.00639 radians, which is 27% higher compared to verification model. These discrepancies mainly are due to the wrongly obtained integration constants as discussed previously. The constant C5 contributing to twist equation is approximately 5% off. Furthermore, there can be small discrepancies due to the differences in obtained shear centre location as described in subsection 3.2.3..

Verification of stress computations

Test 1 To verify the signs of the normal stresses at the cross section, the following method can be used. When the moments around y and z axes are computed, the aileron's cross-section in y-z plane can be analysed (axes established as in figure 2). The origin of the coordinate frame can be moved to the centroid location of the aileron's cross-section. Then the positive moment around z axis will cause a tension (positive normal stress) in the region where y coordinate is smaller than 0. Similarly, a positive y moment will cause the tension in the region where z coordinate is larger than 0. Hence, for positive My and Mz, the normal stress should be positive in the quadrant where y<0 and z>0 and negative in the quadrant where y>0 and z<0. The centre of the coordinate frame is moved to the centroid location for this analysis.

Test 2 To verify the shear stresses caused by the shear forces, the internal shear can be analysed. The total contribution of the shear stresses multiplied by the area it encompasses must equal the shear force. This simple test will test if the shear flows plotted are logical and correct. The test is performed for both the forces in the y and z direction. This resulted in an error of about 0.02% which can be explained by the error caused by the integration scheme.

Test 3 The stresses can be plotted along the cross section of the aileron as done in the verification model. This was done at x=0.5. The results of which can be seen in figure 10, 11 and 12. As can be seen in figure 10 the direct stresses closely resemble the stresses in the verification model. The maximum stress in the model is $4.6 \cdot 10^8$ compared to the $4.8 \cdot 10^8$ in the verification model. This an error of about 5%. This can be explained from the error in the moment calculations. The shear stresses can be seen in figure 11, the stresses are off by quite a large margin, they are off by as much as 34%. This means that there is a large computational error in the shear stress calculations. Lastly, the Von Mises stress distribution closely resembles the verification model. The error is 0.3% meaning that the Von Mises stresses can be assumed correct.

Maximum Stress Experienced by the Aileron The stress distributions are plotted for both the verification model (figures 27-29) as for the numerical model (figures 10-12). The last output of the numerical model is the maximum stress in the aileron, which is $0.5669GPa$. For the verification model the maximum stress is $0.5131GPa$, meaning a difference of 9%. It is calculated from the Von Mises stresses, which are a composition of direct and shear stresses. The difference in value can therefore be explained as the effect of the difference in direct and shear stresses combined. The exact coordinates of the point where maximum stress is experienced is not known, but the region is known. This the lower side of the trailing edge, which is the same region as where the maximum stress for the numerical model was found.

4 Validation

Validation is the last step in the process of checking the accuracy of the numerical model (described in [1]). A finite element model (FEM) created using ABAQUS CAE is given for a Boeing 737. The boundary conditions and force inputs of this model will be applied to the developed numerical model to create results that will subsequently be compared to the validation data to ensure the accuracy of the numerical model. The results will then be checked for errors and the discrepancies between the two models will be explained. This validation model will check the stresses and displacements at specific locations along the length of the aileron using the coordinates provided. This validation model will be taking a select number of coordinates provided (those along the four ribs as well as along the trailing edge and leading edge) as opposed to taking the entire 10,000 point data set and comparing it to 10,000 points produced by the numerical model. This is done to lower the amount of time required to analyse the outputs of the numerical model.

The following three load cases are included in the FEM analysis of the validation model:

- **Load Case 1 — *Bent*:** The aileron is bent with no other exterior forces. In other words, the point load P and the aerodynamic loading q are neglected.
- **Load Case 2 — *Jam_Bent*:** The aileron is bent and has a jammed actuator, whilst the exterior aerodynamic loading is still applied. This results in the same loading case as in the numerical model.
- **Load Case 3 — *Jam_Straight*:** The aileron is straight and has a jammed actuator, whilst the exterior aerodynamic loading is still applied. Thus, deflections d_1 and d_3 are neglected.

4.1 Validation Tests

The following validation tests are implemented to validate the numerical model:

- **Von Mises Stress Computations:** The data output by the numerical model is used to create a plot of the distribution of Von Mises stress at the location of one the ribs in the validation model. These stresses produced by the new model will be compared to the matching validation set of Von Mises stresses. The differences between the stress outputs by the numerical model and the validation model should be uniform. If the differences are not uniform, it could signify an issue within a specific part of the code, as certain forces have a larger effect on the Von Mises stresses than others. This test is used for each of the three load cases.
- **Shear Stress Computations:** Similarly, the shear stress distribution at the same rib is computed and plotted by the numerical model. This is compared to the stress distribution found by the validation model. These plots are used to visually validate the loading distribution across the rib.
- **Deflections Along the Aileron:** Lastly, the deflections along the y and z axes are computed for both the validation model and the numerical model for a specific load case. These are visually compared to validate the numerical model.

4.2 Validation Results and Discrepancies

Data pre-processing At first, the validation data was formatted into pandas dataframes to be more easily interpreted and compared to the numerical model. In order to ensure that the provided validation data had been transformed correctly, visual inspection of such data was performed. Figures 32 and 33 show scatter plots of the found deformation for load cases 1 and 3, respectively. A similar plot was constructed for the second load case; this is not included in the present document for the sake of brevity.

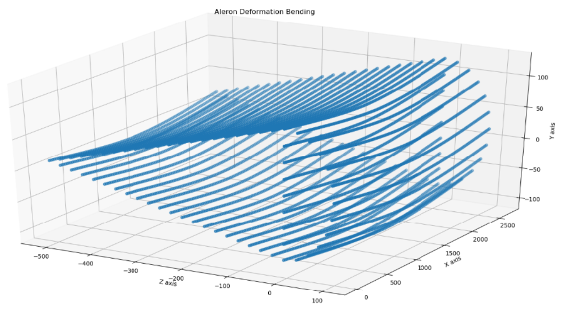


Figure 32: The aileron after the deformations are applied for the bending load case.

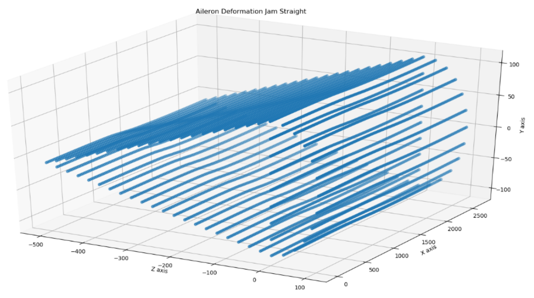


Figure 33: The deformed aileron for the jam straight load case.

Test 1 For the first validation test, the points were plotted for both the shear and Von Mises stresses, and through the use of a colour scale, these stresses could also be shown in the plot as well. This process was conducted for each load case. This can be seen below.

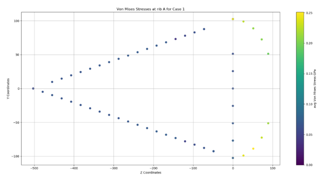


Figure 34: Validation VM stresses for Load Case 1

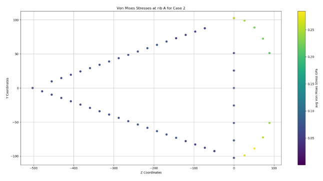


Figure 35: Validation VM stresses for Load Case 2

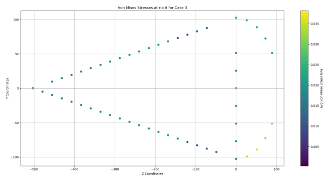
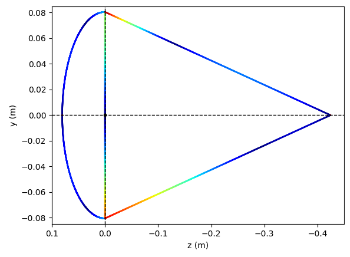
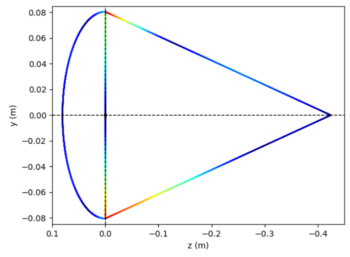


Figure 36: Validation VM stresses for Load Case 3





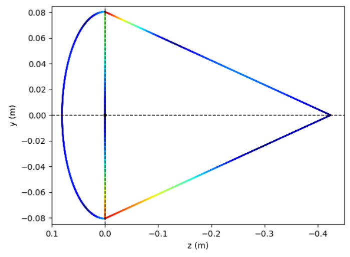


Figure 37: Numerical VM stresses for Load Case 1

Figure 38: Numerical VM stresses for Load Case 2

Figure 39: Numerical VM stresses for Load Case 3

Test 2 The results of the second test, namely the comparison of the shear stresses at Rib A of the verification model, are shown below in the figures below.

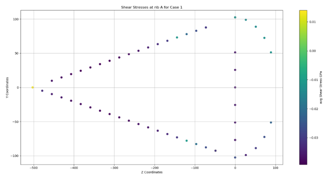


Figure 40: Validation Shear Stress Plots for Load Case 1

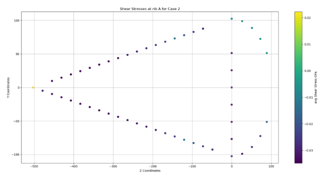


Figure 41: Validation Shear Stress Plots for Load Case 2

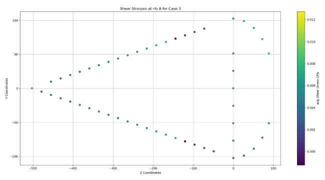


Figure 42: Validation Shear Stress Plots for Load Case 3

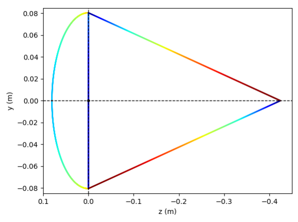


Figure 43: Numerical Shear Stress Plots for Load Case 1

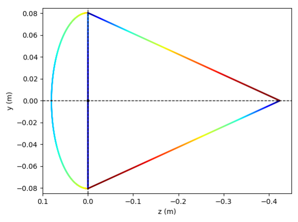


Figure 44: Numerical Shear Stress Plots for Load Case 2

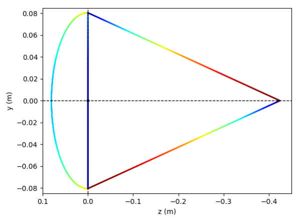


Figure 45: Numerical Shear Stress Plots for Load Case 3

Test 3 The results of the third validation tests are shown in the figure above. It can be seen that the deflections computed by the numerical model in both the y and z directions very closely match the ones computed by the validation model.

Reasoning for Discrepancies The results of the numerical model and the validation model do not match exactly. Although this was to be expected, it is vital to understand the reasons why this is the case. Important discrepancies between the numerical and validation models that are likely to have caused the difference in results are explained below:

- An assumption made by the validation model is that the stiffeners are removed and their properties are essentially introduced into the skin. This is not taken into account by the numerical model, which will affect the moments of inertia of the skin, as well as the stress distribution across the skin.
- The numerical model takes a thin-walled approach, which influences the moments of inertia. Also, the stresses across the skin are averaged, leading to a different value of the total stress across the cross-section.
- The loading application is different in both models. In the FEM, the pressure distribution is given at 11 different cross-sections located towards the tip of the aileron. In the numerical model, the pressure distribution is obtained at each grid point along the aileron. Thus, for the FEM a total number of 11 pressure values is given whilst for the numerical model a total of 41x81 pressure values can be obtained.
- The coordinates of the data points do not match exactly. Thus, the locations in which the stress and the loads are compared slightly differ, leading to small variations in the results obtained in both models.

These discrepancies must be taken into account in the redesign of the numerical model in order to produce a more accurate model. If there are to be large discrepancies, there might have perhaps been an error in the numerical model's code that may not have been seen by the verification process. There may have also been an issue in the numerical code used to compare the numerical model and the validation model. Such an issue would explain any large numerical discrepancies. This could be fixed by reviewing a single calculation of the 10,000 calculations necessary by hand, and making changes accordingly.

In summary, the results of the performed tests indicate that the numerical model has been validated correctly: the results of the numerical model have been shown to very closely resemble those generated by the validation model, and sources for the discrepancies have been identified and analyzed.

Appendix A Task Division

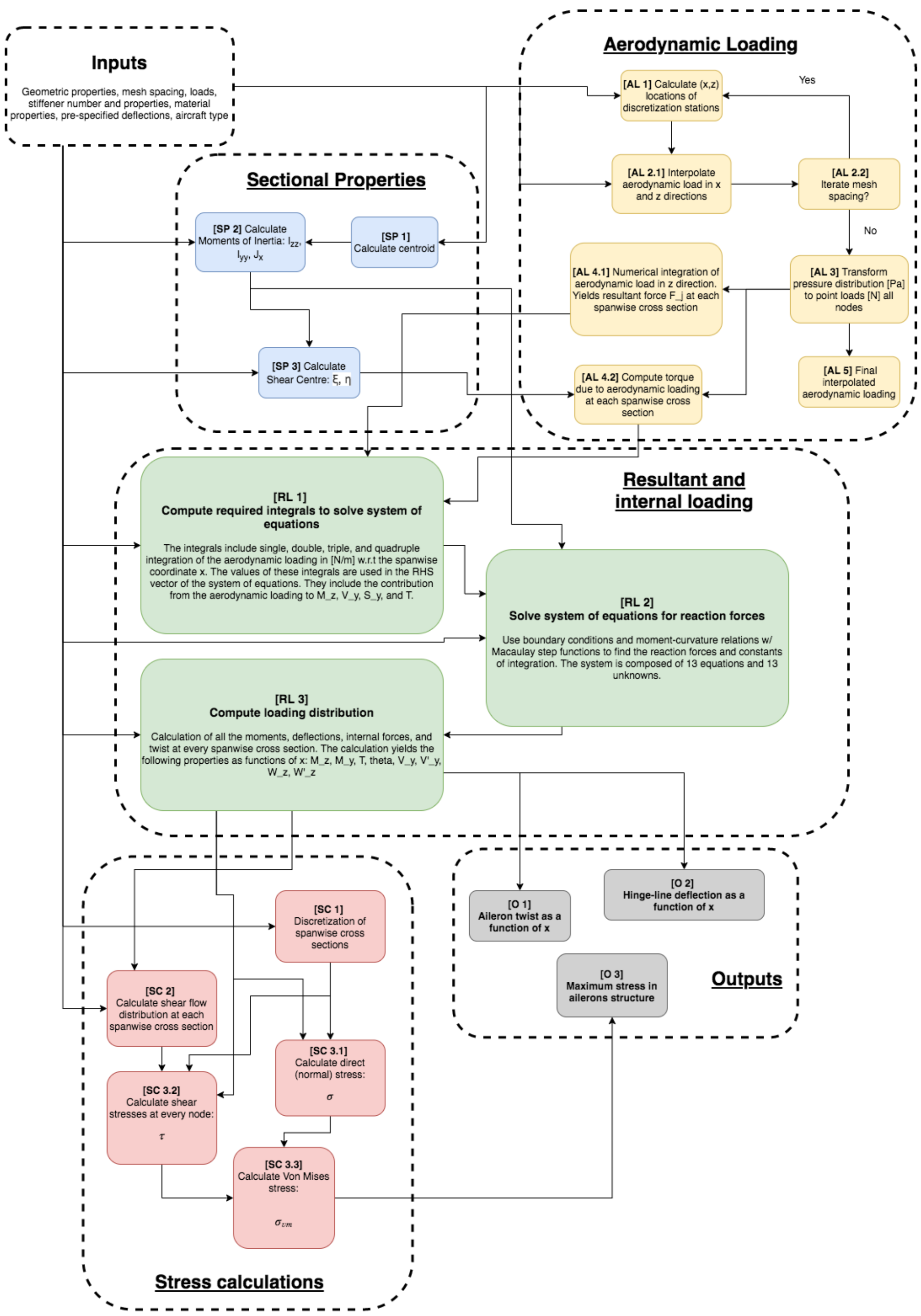
The number of hours spent by each student in each of the sections of the present document is shown below.

Student Name	P. Meseguer Berroy	L. Chin A Foeng	M. Enting
Student Number	4530934	4609972	4659147
Numerical Model	40	20	30
Verification	20	8	38
Validation	5	40	0
Total	**65**	**68**	**68**

Student Name	E. Daugulis	G. Mettepenningen	P. González Martínez
Student Number	4662350	4682343	4582675
Numerical Model	30	35	50
Verification	35	22	12
Validation	0	3	13
Total	**65**	**60**	**75**

Appendix B Flowchart

Figure 46 graphically shows the different modules that are found in the numerical model as well as the links and relations between them.



Inputs
Geometric properties, mesh spacing, loads, stiffener number and properties, material properties, pre-specified deflections, aircraft type
Aerodynamic Loading
[AL 1] Calculate (x,z) locations of discretization stations
Yes
[AL 2.1] Interpolate aerodynamic load in x and z directions
[AL 2.2] Iterate mesh spacing?
No
[AL 4.1] Numerical integration of aerodynamic load in z direction. Yields resultant force F_j at each spanwise cross section
[AL 3] Transform pressure distribution [Pa] to point loads [N] all nodes
[AL 4.2] Compute torque due to aerodynamic loading at each spanwise cross section
[AL 5] Final interpolated aerodynamic loading
Sectional Properties
[SP 2] Calculate Moments of Inertia: I_zz, I_yy, J_x
[SP 1] Calculate centroid
[SP 3] Calculate Shear Centre: ξ, η
Resultant and internal loading
[RL 1] Compute required integrals to solve system of equations
The integrals include single, double, triple, and quadruple integration of the aerodynamic loading in [N/m] w.r.t the spanwise coordinate x. The values of these integrals are used in the RHS vector of the system of equations. They include the contribution from the aerodynamic loading to M_z, V_y, S_y, and T.
[RL 2] Solve system of equations for reaction forces
Use boundary conditions and moment-curvature relations w/ Macaulay step functions to find the reaction forces and constants of integration. The system is composed of 13 equations and 13 unknowns.
[RL 3] Compute loading distribution
Calculation of all the moments, deflections, internal forces, and twist at every spanwise cross section. The calculation yields the following properties as functions of x: M_z, M_y, T, theta, V_y, V'_y, W_z, W'_z
[SC 1] Discretization of spanwise cross sections
[SC 2] Calculate shear flow distribution at each spanwise cross section
[SC 3.1] Calculate direct (normal) stress: σ
[SC 3.2] Calculate shear stresses at every node: τ
[SC 3.3] Calculate Von Mises stress: σ_vm
Stress calculations
[O 1] Aileron twist as a function of x
[O 2] Hinge-line deflection as a function of x
[O 3] Maximum stress in ailerons structure
Outputs

Figure 46: Flowchart of the Proposed Numerical Model

References

M. Enting E. Daugulis G. Mettepenningen P. González Martínez P. Meseguer Berroy, L. Chin A Foeng. *Simulation Plan. for the analysis of a Fokker 100 aileron under critical loading conditions*. Delft University of Technology, 2020.

Daryl L. Logan. *A First Course in the Finite Element Method*. CL Engineering, jan 2011. ISBN 0495668257. URL https://www.xarg.org/ref/a/0495668257/.

Ae3212-ii svv structures assignment 2020, 2020.

R.P. Dwight R. Klees. *Applied Numerical Analysis (AE2220-I) Lecture Notes*. Delft University of Technology, February 2018.

C.Rans J.Melkert. Shear of thin-walled sections. Structural Analysis Slides Lecture 5, 2019.

E. F. Bruhn. *Analysis and Design of Flight Vehicle Structures*, volume 650. Jacobs Pub, 1973. ISBN 978-0961523404.

W. van der Wal S.J. van Elsoo, J.M.J.F. van Campen. *Verification Model, Description of the Verification Model Used in the AE3212-II Project*. 2020.

```
from math import *
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import pandas as pd




def get_data(aircraft, Ca, La):

    '''
    Reads the provided aerodynamic data and returns it in the shape of a
        numpy array.

    Required inputs:
        - AIRCRAFT: Must be one of the following: A320, CRJ700, Do228, F100
        - Ca: Chordwise length, in [m]
        - La: Chrodwise length, in [m]

    Outputs:
        - AERODATA = Matrix containing the provided aerodynamic data.
                  Numpy matrix of dimension 81 by 41.
        - COORD_X = X-cordinates of data. Numpy array of
                    length 41.
        - COORD_Z = Z-cordinates of data. Numpy array of
                    length 81.

    '''

    aircraft = str(aircraft).lower()

    # Reading aerodata -- write as 81x41 matrix
    ref_file = open("aerodynamicload"+aircraft+".dat", "r")
    lines = ref_file.readlines()
    ref_file.close()

    aerodata = np.mat(np.zeros((81, 41)))


    for line in lines:
        idx = lines.index(line)
        line = line.replace("\n","")
        values = line.split(',')
        values = np.matrix(values)

        aerodata[idx,:] = values


    # Calculating x and z coordinates of data points
    theta_x = []  # List of theta_z_i angles
    theta_z = []  # List of theta_x_i angles
    coord_x = []  # List of x coordinates
    coord_z = []  # List of z coordinates


    N_x = 41
    N_z = 81
    for i in range(1,N_x+2):
        theta_i = (i-1)*np.pi/N_x
        theta_x.append(theta_i)


    for i in range(1,N_x+1):
        x_i = -0.5*(   0.5*La*(1-np.cos(theta_x[i-1])) + 0.5*La*(1-np.cos
            (theta_x[i]))  )
        coord_x.append(x_i)


    for i in range(1,N_z+2):
        theta_i = (i-1)*np.pi/N_z
        theta_z.append(theta_i)


    for i in range(1,N_z+1):
        z_i = -0.5*(   0.5*Ca*(1-np.cos(theta_z[i-1])) + 0.5*Ca*(1-np.cos
            (theta_z[i]))  )
        coord_z.append(z_i)


    # Make all x-coordinates positive (positive x-axis starting from root
        towards tip )
    coord_x = np.abs(coord_x)
    coord_z = np.abs(coord_z)

    return aerodata,coord_x,coord_z
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 20 14:53:49 2020

@author: paula
"""
from numpy import cos,sin,pi,sqrt, arctan, arange

ha = 0.161
ca = 0.505
t_st = 0.0012
w_st = 0.017
h_st = 0.013
A_st = t_st * (w_st + h_st)
t_sk = 0.0011
t_sp = 0.0024
nstiff = 11

#Stiffener Spacing in the Triangle
r = ha/2
a = sqrt(r**2+(ca-r)**2)
P = 2*a + pi*r
space_st = P/nstiff

# Coordinates for Stiffeners
zlist = [r]
ylist = [0.0]

# Coordinates for Triangle Stiffeners (Positive)
alpha = arctan(r/(ca-r))
for i in range(4):
    ztriangle = - ca + r + space_st * (1/2 + i) * cos(alpha)
    ytriangle = space_st * (1/2 + i) * sin(alpha)
    zlist.append(ztriangle)
    ylist.append(ytriangle)

# Coordinates for Circle Stiffeners
i = -1
beta = space_st/r
print(beta)
while beta < pi/2:
    zcircle = r * cos(i*beta)
    ycircle = r * sin(i*beta)

    beta = beta + space_st/r

    zlist.append(zcircle)
    ylist.append(ycircle)
    zlist.append(zcircle)
    ylist.append(-ycircle)

    i = i + 1
    print(i)

# Coordinates for Triangle Stiffeners (Negative)
for i in range(4):
    ztriangle = - ca + r + space_st * cos(alpha) * (1/2 + i)
    ytriangle = - space_st * (1/2 + i) * sin(alpha)
    zlist.append(ztriangle)
```

```
        ylist.append(ytriangle)

    # Centroid of the Cross-Section
    A1 = t_sp * ha
    A23 = t_sk * a
    A4 = pi * r * t_sk
    Atot = A1 + 2 * A23 + A4 + nstiff * A_st
    zlistcentr = []
    for i in zlist:
        zcoordst = A_st * (i)
        zlistcentr.append(zcoordst)
    zcoord = (2 * A23 * (-a/2*cos(alpha)) + A4 * ((2*(r-t_sk))/pi)+ sum
        (zlistcentr))/Atot

    # Moment of Inertia Contribution from Stiffeners
    Izz_stlist = []
    for i in ylist:
        Izz_st = A_st * i**2
        Izz_stlist.append(Izz_st)

    Iyy_stlist = []
    for j in zlist:
        Iyy_st = A_st * (zcoord - j)**2
        Iyy_stlist.append(Iyy_st)

    Izz_stot = sum(Izz_stlist)
    Iyy_stot = sum(Iyy_stlist)

    # Moment of Inertia Contribution from Thin-Walled Cross-Section
    Izz_1 = (1/12) * t_sp * ha**3
    Iyy_1 = (1/12) * ha * t_sp**3 + A1 * (zcoord)**2
    Izz_23 = (1/12) * t_sk * a**3 * (sin(alpha))**2 + A23 * (a/2*sin(alpha))**2
    Iyy_23 = (1/12) * t_sk * a**3 * (cos(alpha))**2 + A23 * (-a/2*cos(alpha)-
        zcoord)**2

    def fz(theta):
        fz = t_sk * r**3 * (cos(theta))**2
        return fz

    def fy(theta):
        fy = t_sk * r**3 * (sin(theta))**2
        return fy

    flist_z = []
    flist_y = []
    h = pi/180
    angles = arange(0,pi+h,h)
    for i in range(len(angles)-1):
        area_z = h * fz((angles[i] + angles[i+1])/2)
        flist_z.append(area_z)
        area_y = h * fy((angles[i] + angles[i+1])/2)
        flist_y.append(area_y)

    Izz_4 = sum(flist_z)
    Iyy_4 = sum(flist_y) - A4 * ((2*(r-t_sk))/pi)**2 + A4 * (zcoord - (2*(r-t_sk
        ))/pi)**2

    Izz_cs = Izz_1 + 2 * Izz_23 + Izz_4
    Iyy_cs = Iyy_1 + 2 * Iyy_23 + Iyy_4
```

```
# Total Moment of Inertia
Izz = Izz_stot + Izz_cs # in m^4
Iyy = Iyy_stot + Iyy_cs # in m^4
print(zlist)
print(ylist)
print(zcoord)
print(Izz,Iyy)
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue Feb 25 16:11:48 2020

@author: Marni
"""
import numpy as np
from math import *

r = 0.0805
ca = 0.505
t_skin = 0.0011
t_spar = 0.0024
Vy = 1
Vz = 1
Izz = 4.753851442684436e-06
Iyy = 4.5895702148629556e-05
s1 = sqrt(r**2 + (ca-r)**2)

#--------------------------------------------------------
# Integrator Circular Part

def f_circ(s):
    r = 0.0805
    return -t_skin*Vy/Izz*(r**2*sin(s))

def integral_circ(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_circ(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_circ(a,b):
    ds = (b - a)/(1e3+1)
    r = 0.0805
    s  = a + ds
    int_2 = [f_circ(s)*ds*ds*r]
    s  = a + ds
    while s < b:
        int_2.append(int_2[-1] + f_circ(s)*ds*ds*r)
        s += ds
    return sum(int_2)


#--------------------------------------------------------
# Integrator Triangular Part top

def f_triang(s):
    s1 = sqrt(r**2 + (ca-r)**2)
    return -t_skin*Vy/Izz*(r-(r/s1)*s)

def integral_triang(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_triang(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_triang(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s = a + ds
    int_2 = [f_triang(s)*ds*ds]
    s  = a + 2*ds
    while s < b:
        int_2.append(int_2[-1]+ (f_triang(s)*ds**2))
        s += ds
    return sum(int_2)

#---------------------------------------------------------
# Integrator Triangular Part bottom

def f_triang2(s):
    s1 = sqrt(r**2 + (ca-r)**2)
    return -t_skin*Vy/Izz*-(r/s1)*s

def integral_triang2(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_triang2(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_triang2(a, b):
    ds = (b - a)/(1e3+1)
    s  = a + ds
    int_2 = [f_triang2(s)*ds**2]
    s  = a + 2*ds
    while s < b:
        int_2.append(int_2[-1]+ (f_triang2(s)*ds**2))
        s += ds
    return sum(int_2)

#-----------------------------------------------------------
# Spar Integrator top

def f_spar(s):
    return -t_spar*Vy/Izz*s

def integral_spar(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_spar(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_spar(a, b):
    ds = (b - a)/(1e3+1)
    s  = a + ds
    int_2 = [f_spar(s)*ds**2]
    s  = a + 2*ds
    while s < b:
        int_2.append(int_2[-1]+ (f_spar(s)*ds**2))
        s += ds
    return sum(int_2)

#-----------------------------------------------------------
# Spar Integrator bottom

def f_spar2(s):
    return -t_spar*Vy/Izz*(-r+s)

def integral_spar2(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_spar2(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_spar2(a, b):
    ds = (b - a)/(1e3+1)
    s  = a + ds
    int_2 = [(f_spar2(s)*ds)*ds]
    s  = a + 2*ds
    while s < b:
        int_2.append(int_2[-1]+ (f_spar2(s)*ds**2))
        s += ds
    return sum(int_2)


##############################################################
#
# Integrals for z direction
#
#############################################################


#----------------------------------------------------------
# Integrator Circular Part Z

def f_circ_z(s):
    r = 0.0805
    return -t_skin*Vz/Iyy*(r**2*cos(s))

def integral_circ_z(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_circ_z(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_circ_z(a,b):
    ds = (b - a)/(1e3+1)
    r = 0.0805
    s  = a + ds
    int_2 = [f_circ_z(s)*ds*ds*r]
    s  = a + ds
```

```
        while s < b:
            int_2.append(int_2[-1] + f_circ_z(s)*ds*ds*r)
            s += ds
        return sum(int_2)


    #---------------------------------------------------------
    # Integrator Triangular Part top z

    def f_triang_z(s):
        return -t_skin*Vz/Iyy*-(ca-r)/s1*s

    def integral_triang_z(a, b):
        ds = (b - a)/(1e3+1)
        int_1 = []
        s  = a + ds
        while s < b:
            int_1.append(f_triang_z(s)*ds)
            s += ds
        return sum(int_1)

    def second_integral_triang_z(a, b):
        ds = (b - a)/(1e3+1)
        int_1 = []
        s = a + ds
        int_2 = [f_triang_z(s)*ds*ds]
        s  = a + 2*ds
        while s < b:
            int_2.append(int_2[-1]+ (f_triang_z(s)*ds**2))
            s += ds
        return sum(int_2)

    #---------------------------------------------------------
    # Integrator Triangular Part bottom z

    def f_triang2_z(s):
        s1 = sqrt(r**2 + (ca-r)**2)
        return -t_skin*Vz/Iyy*(-(ca-r) + (ca-r)/s1*s)

    def integral_triang2_z(a, b):
        ds = (b - a)/(1e3+1)
        int_1 = []
        s  = a + ds
        while s < b:
            int_1.append(f_triang2_z(s)*ds)
            s += ds
        return sum(int_1)

    def second_integral_triang2_z(a, b):
        ds = (b - a)/(1e3+1)
        s  = a + ds
        int_2 = [f_triang2_z(s)*ds**2]
        s  = a + 2*ds
        while s < b:
            int_2.append(int_2[-1]+ (f_triang2_z(s)*ds**2))
            s += ds
        return sum(int_2)

    #-----------------------------------------------------------
    # Spar Integrator top z
```

```
def f_spar_z(s):
    return -t_spar*Vz/Iyy

def integral_spar_z(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_spar_z(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_spar_z(a, b):
    ds = (b - a)/(1e3+1)
    s  = a + ds
    int_2 = [f_spar_z(s)*ds**2]
    s  = a + 2*ds
    while s < b:
        int_2.append(int_2[-1]+ (f_spar_z(s)*ds**2))
        s += ds
    return sum(int_2)

#----------------------------------------------------------
# Spar Integrator bottom z

def f_spar2_z(s):
    return -t_spar*Vz/Iyy

def integral_spar2_z(a, b):
    ds = (b - a)/(1e3+1)
    int_1 = []
    s  = a + ds
    while s < b:
        int_1.append(f_spar2_z(s)*ds)
        s += ds
    return sum(int_1)

def second_integral_spar2_z(a, b):
    ds = (b - a)/(1e3+1)
    s  = a + ds
    int_2 = [f_spar2_z(s)*ds**2]
    s  = a + 2*ds
    while s < b:
        int_2.append(int_2[-1]+ (f_spar2_z(s)*ds**2))
        s += ds
    return sum(int_2)
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 21 10:35:57 2020

@author: pablo
"""


from numpy import cos,sin,pi,sqrt, arctan, arange,matrix,pi,linspace,array,
    append
from numpy.linalg import solve
from math import *
import matplotlib as plt
import ShearIntegrator as itg



def get_MoI(ha,ca,t_st,w_st,h_st,A_st,t_sk,t_sp,nstiff):

    #Stiffener Spacing in the Triangle
    r = ha/2
    a = sqrt(r**2+(ca-r)**2)
    P = 2*a + pi*r
    space_st = P/nstiff

    # Coordinates for Stiffeners
    zlist = [r]
    ylist = [0.0]

    # Coordinates for Triangle Stiffeners (Positive)
    alpha = arctan(r/(ca-r))

    for i in range(4):
        ztriangle = - ca + r + space_st * (1/2 + i) * cos(alpha)
        ytriangle = space_st * (1/2 + i) * sin(alpha)
        zlist.append(ztriangle)
        ylist.append(ytriangle)

    # Coordinates for Circle Stiffeners
    beta = space_st/r
    while beta < pi/2:
        zcircle = r * cos(beta)
        ycircle = r * sin(beta)

        beta = beta + space_st/r

        zlist.append(zcircle)
        ylist.append(ycircle)
        zlist.append(zcircle)
        ylist.append(-ycircle)

    # Coordinates for Triangle Stiffeners (Negative)
    for i in range(4):
        ztriangle = - ca + r + space_st * cos(alpha) * (1/2 + i)
        ytriangle = - space_st * (1/2 + i) * sin(alpha)
        zlist.append(ztriangle)
        ylist.append(ytriangle)

    # Centroid of the Cross-Section
    A1 = t_sp * ha
    A23 = t_sk * a
    A4 = pi * r * t_sk
    Atot = A1 + 2 * A23 + A4 + 11 * A_st
    zlistcentr = []
    for i in zlist:
        zcoordst = A_st * (i)
        zlistcentr.append(zcoordst)
    zcoord = (2 * A23 * (-a/2*cos(alpha)) + A4 * ((2*(r-t_sk))/pi)+ sum
        (zlistcentr))/Atot

    # Moment of Inertia Contribution from Stiffeners
    Izz_stlist = []
    for i in ylist:
        Izz_st = A_st * i**2
        Izz_stlist.append(Izz_st)

    Iyy_stlist = []
    for j in zlist:
        Iyy_st = A_st * (zcoord - j)**2
        Iyy_stlist.append(Iyy_st)

    Izz_stot = sum(Izz_stlist)
    Iyy_stot = sum(Iyy_stlist)

    # Moment of Inertia Contribution from Thin-Walled Cross-Section
    Izz_1 = (1/12) * t_sp * ha**3
    Iyy_1 = (1/12) * ha * t_sp**3 + A1 * (zcoord)**2
    Izz_23 = (1/12) * t_sk * a**3 * (sin(alpha))**2 + A23 * (a/2*sin(alpha))
        **2
    Iyy_23 = (1/12) * t_sk * a**3 * (cos(alpha))**2 + A23 * (-a/2*cos(alpha)
        -zcoord)**2

    def fz(theta):
        fz = t_sk * r**3 * (cos(theta))**2
        return fz

    def fy(theta):
        fy = t_sk * r**3 * (sin(theta))**2
        return fy

    flist_z = []
    flist_y = []
    h = pi/180
    angles = arange(0,pi+h,h)
    for i in range(len(angles)-1):
        area_z = h * fz((angles[i] + angles[i+1])/2)
        flist_z.append(area_z)
        area_y = h * fy((angles[i] + angles[i+1])/2)
        flist_y.append(area_y)

    Izz_4 = sum(flist_z)
    Iyy_4 = sum(flist_y) - A4 * ((2*(r-t_sk))/pi)**2 + A4 * (zcoord - (2*(r-
        t_sk))/pi)**2

    Izz_cs = Izz_1 + 2 * Izz_23 + Izz_4
    Iyy_cs = Iyy_1 + 2 * Iyy_23 + Iyy_4

    # Total Moment of Inertia
    Izz = Izz_stot + Izz_cs # in m^4
```

```
        Iyy = Iyy_stot + Iyy_cs # in m^4


        return Izz,Iyy,zcoord

    def get_J(G,ha,t_sk,w_st,t_sp,t_st,ca):
        #calculation of J by evaluating shear flows
        #assumption - stiffeners included in analysis when integrating length
            over thickness
        #stiffeners assumed as horizontal stiffeners, vertical part is
            disregarded
        A1=(ha/2)**2*pi/2
        A2=sqrt((ha/2)**2 + (ca-ha/2)**2)*ha/2
        randtorque=1
        Y=[randtorque,
        0]
        AA=matrix([[2*A1, 2*A2],
                     [1/(2*A1*G)*((pi*ha/2)/t_sk) + ha/(2*A2*G*t_sp) + ha/(2*A1*
                         G*t_sp), - 1/(2*A2*G)*((2*sqrt((ha/2)**2+(ca-ha/2)**2))
                         /t_sk) - ha/(2*A2*G*t_sp) - ha/(2*A1*G*t_sp)]])
        shears=solve(AA,Y)
        q1=shears[0]
        q2=shears[1]
        dodz =  1/(2*A1)*(q1/G*((pi*ha/2 )/t_sk) + (q1-q2)/G*(ha)/t_sp)
        dodz2=1/(2*A2)*(q2/G*((2*sqrt((ha/2)**2+(ca-ha/2)**2))/t_sk) + (q2-q1)/G
            *(ha/t_sp))
        J=randtorque/(G*dodz)
        return J


    def get_SC(t_skin,t_spar,ha,ca,Izz):

        r = ha/2
        dz = 1e-5
        Vy = 1

        s1 = sqrt(r**2 + (ca-r)**2)
        s2 = 2*r
        ds = s1/1e3
        dtheta = pi/2/1e3
        dr = r/1e3
        A = pi*r**2/2
        x_s = (ca-r)/s1

        A_circ = r**2*pi/2
        A_triang = s1*r



        #Shear flow calculation y force, open section cut at LE and hingeline

        def qb1(theta):     # Shear Flow open section top circular
            return itg.integral_circ(0, theta)

        def qb2(s):     # Spar Top
            return itg.integral_spar(0, s)

        def qb3(s):     # Triangular part top
            return  itg.integral_triang(0, s) + qb1(pi/2) + qb2(r)
```

```
def qb4(s):     # Triangular part bottom
    return  itg.integral_triang2(0, s) + qb3(s1)

def qb5(s):     # Spar bottom
    return  itg.integral_spar2(0, r) + qb2(r)

def qb6(theta):     # Circular part bottom
    return  itg.integral_circ(-pi/2, theta) + qb1(pi/2)

"""
shears = []
s_tot = []
s = 0
while s<=s1:
    shears.append(qb4(s))
    s_tot.append(s)
    s += ds
plt.pyplot.plot(s_tot, shears)
"""
qs0_circ = -((itg.second_integral_circ(0, pi/2)/t_skin - itg.
    second_integral_spar(0, r)/t_spar + itg.second_integral_circ(-pi/2,
    0)/t_skin) - itg.second_integral_spar2(0, r)/t_spar - qb2(r)*r/
    t_spar +qb1(pi/2)*pi*r/2/t_skin)*(pi*r/t_skin + 2*r/t_spar)**-1
qs0_triang = -(itg.second_integral_triang(0, s1) + itg.
    second_integral_triang2(0, s1) + itg.second_integral_spar(0, r) +
    itg.second_integral_spar2(0, r) + (qb1(pi/2) + qb2(r))*s1/t_skin +
    qb3(s1)*s1/t_skin + qb2(r)*r/t_spar)*(2*s1/t_skin + 2*r/t_spar)**-1



def qs1(s):
    return qb1(s) + qs0_circ

def qs2(s):
    return qb2(s) - qs0_circ + qs0_triang

def qs3(s):
    return qb3(s) + qs0_triang

def qs4(s):
    return qb4(s) + qs0_triang

def qs5(s):
    return qb5(s) - qs0_circ + qs0_triang

def qs6(s):
    return qb6(s) + qs0_circ


shears1 = []
thetas1 = []
theta = 0
while theta<=(pi/2):
    shears1.append(qs1(theta))
    thetas1.append(theta)
    theta += dtheta

shears6 = []
thetas6 = []
theta = -pi/2
```

```
        while theta<=0:
            shears6.append(qs6(theta))
            thetas6.append(theta + pi/2)
            theta += dtheta

        shears2 = []
        thetas2 = []
        s = 0
        while s<=r:
            shears2.append(qs2(s))
            thetas2.append(s)
            s += dr

        shears5 = []
        thetas5 = []
        s = 0
        while s<=r:
            shears5.append(qs5(s))
            thetas5.append(s)
            s += dr

        shears3 = []
        thetas3 = []
        s = 0
        while s<=s1:
            shears3.append(qs3(s))
            thetas3.append(s)
            s += ds

        shears4 = []
        thetas4 = []
        s = 0
        while s<=s1:
            shears4.append(qs4(s))
            thetas4.append(s)
            s += ds

        #plt.pyplot.plot(thetas1, shears1)
        #plt.pyplot.plot(thetas6, shears6)
        #plt.pyplot.plot(thetas2, shears2)
        #plt.pyplot.plot(thetas5, shears5)
        #plt.pyplot.plot(thetas3, shears3)
        #plt.pyplot.plot(thetas4, shears4)

        ksi = 0
        ksi += r*(ca-r)/s1*sum(shears3)/len(shears3)*s1
        ksi += r*(ca-r)/-s1*sum(shears4)/len(shears4)*s1
        ksi += r*sum(shears1)/len(shears1)*pi*r
        return ksi


    def discretize_crosssection(n):

        # Input n is the number of points PER SECTION of the cross section
        # E.g. if n=1000, the semicircular part of the crossection will have
            1000 points

        ha = 0.161 #[m]
        Ca = 0.505  #[m]
        circle_1_y = array([])
        circle_1_z = array([])

        circle_2_y = array([])
        circle_2_z = array([])


        spar_1_y = array([])
        spar_1_z = array([])

        spar_2_y = array([])
        spar_2_z = array([])

        topline_y = array([])
        topline_z = array([])

        bottomline_y = array([])
        bottomline_z = array([])

        # Generate circle coords
        r = 0.5*ha
        theta_range_1 = linspace(pi,pi/2,n)
        theta_range_2 = linspace(3*pi/2,pi,n)

        for i in range(len(theta_range_1)):
            theta = theta_range_1[i]
            z = - r*cos(theta)
            y = r*sin(theta)

            circle_1_y = append(circle_1_y,y)
            circle_1_z = append(circle_1_z,z)

        for i in range(len(theta_range_2)):
            theta = theta_range_2[i]
            z = - r*cos(theta)
            y = r*sin(theta)

            circle_2_y = append(circle_2_y,y)
            circle_2_z = append(circle_2_z,z)


        # Generate spar coords
        spar_range_1 = linspace(0,r,n)
        spar_range_2 = linspace(-r,0,n)

        for i in range(len(spar_range_1)):
            y = spar_range_1[i]
            z = 0

            spar_1_y = append(spar_1_y,y)
            spar_1_z = append(spar_1_z,z)


        for i in range(len(spar_range_2)):
            y = spar_range_2[i]
            z = 0
            spar_2_y = append(spar_2_y,y)
            spar_2_z = append(spar_2_z,z)


        # Generate coords for both lines

        run = -(Ca-r)
        rise_top = -r
        rise_bottom = r

        m_top = rise_top/run
        m_bottom = rise_bottom/run

        b_top = r
        b_bottom = -r

        line_range = linspace(0,-(Ca-r),n)

        for i in range(len(line_range)):

            z = line_range[i]

            top_y_coord = m_top*z + b_top
            bottom_y_coord = m_bottom*z + b_bottom

            topline_y = append(topline_y,top_y_coord)
            topline_z = append(topline_z,z)

            bottomline_y = append(bottomline_y,bottom_y_coord)
            bottomline_z = append(bottomline_z,z)


        circle_1_coords = array([circle_1_z,circle_1_y])
        circle_2_coords = array([circle_2_z,circle_2_y])
        spar_1_coords = array([spar_1_z,spar_1_y])
        spar_2_coords = array([spar_2_z,spar_2_y])
        topline_coords = array([topline_z,topline_y])
        bottomline_coords = array([bottomline_z,bottomline_y])


        return circle_1_coords,circle_2_coords,spar_1_coords,spar_2_coords,
            topline_coords,bottomline_coords


    def get_shearflows(Fy, Fz):

        Izz = 4.753851442684436e-06
        t_skin =  0.0011
        t_spar = 0.0024
        ca = 0.505
        ha = 0.161
        r = ha/2
        dz = 1e-5
        Vy = 1
        s1 = sqrt(r**2 + (ca-r)**2)
        s2 = 2*r
        ds = s1/1e3
        dtheta = pi/2/1e3
        dr = r/1e3
        A = pi*r**2/2
```

```
x_s = (ca-r)/s1
angle = (0.505-r)/s1
Vz = 1
Iyy = 4.5895702148629556e-05

A_circ = r**2*pi/2
A_triang = (ca-r)*r



def qb1(theta):     # Shear Flow open section top circular
    return Fy*itg.integral_circ(0, theta)

def qb2(s):     # Spar Top
    return Fy*itg.integral_spar(0, s)

def qb3(s):     # Triangular part top
    return  Fy*itg.integral_triang(0, s) + qb1(pi/2) + qb2(r)

def qb4(s):     # Triangular part bottom
    return  Fy*itg.integral_triang2(0, s) + qb3(s1)

def qb5(s):     # Spar bottom
    return  Fy*itg.integral_spar2(0, r) + qb2(r)

def qb6(theta):     # Circular part bottom
    return  Fy*itg.integral_circ(-pi/2, theta) + qb1(pi/2)

"""
shears = []
s_tot = []
s = 0
while s<=s1:
    shears.append(qb4(s))
    s_tot.append(s)
    s += ds
plt.pyplot.plot(s_tot, shears)
"""
qs0_circ = -((itg.second_integral_circ(0, pi/2)/t_skin - itg.
    second_integral_spar(0, r)/t_spar + itg.second_integral_circ(-pi/2,
    0)/t_skin) - itg.second_integral_spar2(0, r)/t_spar - qb2(r)*r/
    t_spar +qb1(pi/2)*pi*r/2/t_skin)*(pi*r/t_skin + 2*r/t_spar)**-1
qs0_triang = -(itg.second_integral_triang(0, s1) + itg.
    second_integral_triang2(0, s1) + itg.second_integral_spar(0, r) +
    itg.second_integral_spar2(0, r) + (qb1(pi/2) + qb2(r))*s1/t_skin +
    qb3(s1)*s1/t_skin + qb2(r)*r/t_spar)*(2*s1/t_skin + 2*r/t_spar)**-1



def qs1(s):
    return qb1(s) + qs0_circ

def qs2(s):
    return qb2(s) - qs0_circ + qs0_triang

def qs3(s):
    return qb3(s) + qs0_triang

def qs4(s):
    return qb4(s) + qs0_triang
```

```
def qs5(s):
    return qb5(s) - qs0_circ + qs0_triang

def qs6(s):
    return qb6(s) + qs0_circ



#----------------------------------------------
# Shear Flows Z - force

def qb1_z(theta):     # Shear Flow open section top circular
    return Fz*itg.integral_circ_z(0, theta)

def qb2_z(s):     # Spar Top
    return Fz*itg.integral_spar_z(0, s)

def qb3_z(s):     # Triangular part top
    return  Fz*itg.integral_triang_z(0, s) + qb1_z(pi/2) + qb2_z(r)

def qb4_z(s):     # Triangular part bottom
    return  Fz*itg.integral_triang2_z(0, s) + qb3_z(s1)

def qb5_z(s):     # Spar bottom
    return  Fz*itg.integral_spar2_z(0, r) + qb2_z(r)

def qb6_z(theta):     # Circular part bottom
    return  Fz*itg.integral_circ_z(-pi/2, theta) + qb1_z(pi/2)

"""
shears = []
s_tot = []
s = 0
while s<=s1:
    shears.append(qb4(s))
    s_tot.append(s)
    s += ds
plt.pyplot.plot(s_tot, shears)
"""
qs0_circ_z = -((itg.second_integral_circ_z(0, pi/2)/t_skin - itg.
    second_integral_spar_z(0, r)/t_spar + itg.second_integral_circ_z(-pi
    /2, 0)/t_skin) - itg.second_integral_spar2_z(0, r)/t_spar - qb2_z(r)
    *r/t_spar +qb1_z(pi/2)*pi*r/2/t_skin)*(pi*r/t_skin + 2*r/t_spar)**-1
qs0_triang_z = -(itg.second_integral_triang_z(0, s1) + itg.
    second_integral_triang2_z(0, s1) + itg.second_integral_spar_z(0, r)
    + itg.second_integral_spar2_z(0, r) + (qb1_z(pi/2) + qb2_z(r))*s1/
    t_skin + qb3_z(s1)*s1/t_skin + qb2_z(r)*r/t_spar)*(2*s1/t_skin + 2*r
    /t_spar)**-1



def qs1_z(s):
    return qb1_z(s) + qs0_circ_z

def qs2_z(s):
    return qb2_z(s) - qs0_circ_z + qs0_triang_z

def qs3_z(s):
    return qb3_z(s) + qs0_triang_z
```

```
def qs4_z(s):
    return qb4_z(s) + qs0_triang_z

def qs5_z(s):
    return qb5_z(s) - qs0_circ_z + qs0_triang_z

def qs6_z(s):
    return qb6_z(s) + qs0_circ_z

'''
shears = []
s_tot = []
s = -pi/2
while s<=0:
    shears.append(qs6_z(s))
    s_tot.append(s)
    s += ds
plt.pyplot.plot(s_tot, shears)
'''


qsc1 = []
qsc2 = []
qsp1 = []
qsp2 = []
qtop = []
qbot = []

s = 0
while s<=pi/2-dtheta:
    qsc1.append(qs1_z(s) + qs1(s))
    s += dtheta

s = 0

while s<=r-dr:
    qsp1.append(qs2_z(s) + qs2(s))
    s += dr

s = 0

while s<=s1-ds:
    qtop.append(qs3_z(s) + qs3(s))
    s += ds

s = 0

while s<=s1-ds:
    qbot.append(qs4_z(s) + qs4(s))
    s += ds

s = 0

while s<=r-dr:
    qsp2.append(qs5_z(s) + qs5(s))
    s += dr

s = -pi/2
```

```
        while s<=0-dtheta:
            qsc2.append(qs6_z(s) + qs6(s))
            s += dtheta


        return qsc1,qsp1,qtop,qbot,qsp2,qsc2



    '''
    Vy = 0
    n = 0
    Izz = 4.753851442684436e-06
    t_skin =  0.0011
    t_spar = 0.0024
    ca = 0.505
    ha = 0.161
    r = ha/2
    dz = 1e-5
    Vy = 1
    s1 = sqrt(r**2 + (ca-r)**2)
    s2 = 2*r
    ds = s1/1e3
    dtheta = pi/2/1e3
    dr = r/1e3
    A = pi*r**2/2
    x_s = (ca-r)/s1
    angle = (0.505-r)/s1
    Vz = 1
    Iyy = 4.5895702148629556e-05

    A_circ = r**2*pi/2
    A_triang = (ca-r)*r

    for i in test_1:
        n+=1
        Vy += -i*dtheta*r*t_skin*sin(dtheta*n)
    n = 0
    for i in test_6:
        Vy += i*dtheta*r*t_skin*sin(dtheta*n)

    for i in test_3:
        Vy += -i*ds*t_skin*r/s1

    for i in test_4:
        Vy += i*ds*t_skin*r/s1

    for i in test_2:
        Vy += -i*dr*t_spar*r/s1

    print(Vy)
    '''
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 17 16:15:16 2020

@author: pablo


"""


from math import *
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm


'''

UNIVARIATE INTERPOLATION FUNCTIONS.

Direction of interpolation is denoted by x.

Number of data points = n+1
Number of intervals = n
Note that the indexing starts at zero, i.e. the first node is located at x_0


'''




def find_interpolants(nodes,data):

    '''
    Finds the cubic splines at each of the n nodes. Each cubic spline has
        the format:

        a_i(x-x_i)^3 + b_i(x-x_i)^2 + c_i(x-x_i) + d_i


    Required inputs:
        - NODES: X positions of the (n+1) nodes where the function values
            are known.
                 Numpy array - length of n+1.
        - DATA: Function values at the specified nodes.
                Numpy array - length of n+1.

    Outputs:
        - INTERPOLANTS = Matrix containing cubic spline parameters a_i,b_i,
            c_i,d_i for
                      each of the n intervals. Numpy matrix of dimension
                          n by 4.

    '''

    n = len(nodes)-1

    # Function values at nodes
```

```
        f = data


        # Compute spacings h where h_i = x_i+1 - x_1
        h = np.diff(nodes)


        # Initializing matrix
        A = np.zeros((n+1,n+1))
        shape = A.shape


        # Construct interior rows (all but the 0th and the nth row)
        for i in range(1,n):
            A[i,i-1] = h[i-1]/6
            A[i,i] = (h[i-1]+h[i])/3
            A[i,i+1] = 1/6*h[i]


        # Enforcing natural boundary conditions on matrix
        A[0,0]=1
        A[n,n]=1


        # Initializing RHS vector
        b = np.zeros(n+1)

        # Constructing interior values of RHS vector
        for i in range(1,n):
            b[i] = (f[i+1]-f[i])/h[i] - (f[i]-f[i-1])/h[i-1]

        # Enforcing natural boundary conditions on RHS vector
        b[0] = 0
        b[n] = 0

        # Solve system of equations for M_i's
        # Ax = b --> find x, vector of M_i's (denoted by M)
        M = np.linalg.tensorsolve(A,b)

        # Initialize results matrix, which contains a_i,b_i,c_i,d_i for
        # i={0,1,....,n-2,n-1} -- total of n interpolants
        interpolants = np.zeros((n,4))


        for i in range(0,n):
            a_i = (M[i+1]-M[i])/(6*h[i])
            b_i = M[i]/2
            c_i = (f[i+1]-f[i])/h[i] - h[i]*M[i]/3 - h[i]*M[i+1]/6
            d_i = f[i]
            row = np.array([a_i,b_i,c_i,d_i])
            interpolants[i,:] = row

        return interpolants



    def get_number_of_points(nodes,resolution):

        '''
        Computes the number of points per spline in order to obtain the
        necessary resolution at the most critical (i.e. largest) discretization
        step. By using this number of points per spline,
        the actual obtained resolution will always be equal or GREATER than
        the provided resolution.


        Required inputs:

            - NODES: X positions of the (n+1) nodes where the function values
                are known.
                     Numpy array - length of n+1.
            - RESOLUTION: Minimum resolution of the new discretized grid.
                          Must be given in mm.

        Outputs:
            - N_POINTS: Resulting number of points per spline. (Integer)
        '''


        n = len(nodes)-1

        # Convert resolution from mm to m
        resolution = resolution/1000

        # Get max spacing
        h = np.diff(nodes)
        max_spacing = np.amax(h)

        # Required number of points PER SPLINE (!) to obtain the given
        # resolution at the most critical (i.e. largest) discretization
        # step. By using this number of points per spline,
        # the actual obtained resolution will always be equal or GREATER than
        # the provided resolution.
        n_points = np.ceil(max_spacing/resolution) + 1

        n_internal = n_points - 1

        return int(n_points)



    def new_loading(nodes, interpolants, n_points):

        '''
        Computes the aerodynamic loading at the new resolution
        using the interpolants from find_interpolants().


        Required inputs:
            - INTERPOLANTS: Result from find_interpolants().
                            Numpy array - dimension n x 4
            - NODES: X positions of the (n+1) nodes where the function values
                are known.
                     Numpy array - length of n+1.
            - N_POINTS: Number of points per spline. Obtained by calling
                get_number_of_points()
                        using the same NODES and a RESOLUTION value.

        Outputs:
            - NEW_NODES = Numpy array containing the new nodes. Length = (n+1)+
                (n)(n_internal)
                          Thus, the number of new nodes depends on the
                              RESOLUTION.
            - NEW_LOADING = Array containing the loading at each of the
                        discretized stations.
                        Numpy array of length n_x
        '''


        # Ensure n is appropiately defined in order for the
        # for loops below to work properly.
        n = len(nodes)-1


        # Number of internal nodes per spline, for reference. (NOT USED IN
            FUNCTION)
        # Can be used to perform a check on the number of total points.
        n_internal = n_points - 2


        new_nodes = np.array([])
        new_loading = np.array([])

        # Loop through all splines
        for i in range(0,n):

            # Get interpolant coefficients
            a_i = interpolants[i,0]
            b_i = interpolants[i,1]
            c_i = interpolants[i,2]
            d_i = interpolants[i,3]


            # Generate new nodes
            x_b = nodes[i]
            x_f = nodes[i+1]

            interval_nodes = np.linspace(x_b,x_f,num=int(n_points))

            if i != 0:
                interval_nodes = interval_nodes[1:int(n_points)]

            new_nodes = np.append(new_nodes,interval_nodes)


            # Compute new loads at all discretized points
            for i in range(len(interval_nodes)):
                load = a_i*(interval_nodes[i]-x_b)**3 + b_i*(interval_nodes[i]-
                    x_b)**2 + c_i*(interval_nodes[i]-x_b) + d_i
                new_loading = np.append(new_loading,load)


        return new_nodes,new_loading
```

```
from math import *
import numpy as np
#This should be used along a chord of a given airfoil
#a,b,c,d should be passed as arrays and n (number of points) as an int
#c_length is the chord length
'''
This class is to be used to analytically integrate between points
INPUTS: matrix: a matrix of a,b,c,d values for the cubic splines between
    each of the pairs of the coords also given.
                OR a matrix of a,b,c,d,e values for the quartic splines
                    between each of the pairs of the coords given.
                the double_integrator and quad_integrator functions
                    integrate the cubic spline 2 times or 4 times
        coords: an array of the coordinates at which the aerodynamic data
            was taken for the a,b,c,d cubic splines(array)
OUTPUTS: The sum of the integrated portions for the entire length of the
    span or the chord over which you have integrated(array)
        OR the sum of the integrated portions for the entire integral that
            you wish
        The output of the double integrator function and the quad integrator
            function both output a single float that
        is the integrated value

'''

class Analytical_Int_1D:
    def __init__(self,matrix,coords):
        self.matrix = matrix
        if len(matrix[0]) == 4:

            self.a = np.array([arr[0] for arr in matrix])
            self.b = np.array([arr[1] for arr in matrix])
            self.c = np.array([arr[2] for arr in matrix])
            self.d = np.array([arr[3] for arr in matrix])

        if len(matrix[0]) == 5:
            self.a = np.array([arr[0] for arr in matrix])
            self.b = np.array([arr[1] for arr in matrix])
            self.c = np.array([arr[2] for arr in matrix])
            self.d = np.array([arr[3] for arr in matrix])
            self.e = np.array([arr[4] for arr in matrix])
        coords+=0.0805
        self.coords = np.array(coords)

    #x_0 and x_1 are the two x values on the chord between which you are
        integrating
    def integrator(self):
        x_1 = (self.coords[1:])
        x_0 = (self.coords[:-1])
        if len(self.matrix[0]) == 4:
            return sum(((self.a*(x_1)**4)/4 + (self.b*(x_1)**3)/3 + (self.c*
                (x_1)**2)/2 + self.d*(x_1)) - ((self.a*(x_0)**4)/4 + (self.b
                *(x_0)**3)/3 + (self.c*(x_0)**2)/2 + self.d*(x_0)))
        if len(self.matrix[0]) == 5:
            return sum((self.a*(x_1)**5)/5 + (self.b*(x_1)**4)/4 + (self.c*
                (x_1)**3)/3 + (self.d*(x_1)**2)/2 + self.e(x_1)-(self.a*(x_0
                )**5)/5 + (self.b*(x_0)**4)/4 + (self.c*(x_0)**3)/3 + (self.
                d*(x_0)**2)/2 + self.e(x_0))

    def double_integrator(self):
        x_1 = (self.coords[1:])
        x_0 = (self.coords[:-1])
        self.e = (self.a*x_0**3) + (self.b*x_0**2) + (self.c*x_0) + self.d
        return sum((self.a*.05*(x_1)**5) + ((self.b/12)*(x_1)**4) + ((self.c
            /6)*(x_1)**3) + ((self.d*0.5)*(x_1)**2) - (self.a*.05*(x_0)**5)
            + ((self.b/12)*(x_0)**4) + ((self.c/6)*(x_0)**3) + ((self.d*0.5)
            *(x_0)**2))
    def quad_integrator(self):
        x_1 = (self.coords[1:])
        x_0 = (self.coords[:-1])
        x_d = x_1-x_0
        self.e = (self.a*x_0**3) + (self.b*x_0**2) + (self.c*x_0) + self.d
        self.f = (self.a/4)*x_0**4 + (self.b/3)*x_0**3 + (self.c/2)*x_0**2 +
            self.d*x_0 + self.e
        self.g = (self.a/20)*x_0**5 + (self.b/12)*x_0**4 + (self.c/6)*x_0**3
            + (self.d/2)*x_0**2 + self.e*x_0 + self.f
        self.h = (self.a/120)*x_0**6  + (self.b/60)*x_0**5 + (self.c/24)*x_0
            **4 + (self.d/6)*x_0**3 + (self.e/2)*x_0**2 + self.f*x_0 + self.
            g

        return sum(((self.a/840)*(x_1)**7 + (self.b/360)*x_1**6 + (self.c/
            120)*x_1**5 + (self.d/24)*x_1**4) - ((self.a/840)*(x_0)**7 +
            (self.b/360)*x_0**6 + (self.c/120)*x_0**5 + (self.d/24)*x_0**4))
    #+ (self.d/24)*x_d**4 + (self.e / 6) * x_d ** 3 + (self.f / 2) * x_d **
        2 + self.g * x_d)
```

```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 17 14:56:32 2020
@author: edgar
"""

from math import *
import numpy as np
from matplotlib import pyplot as plt



"Tools to implement Macauley functions"
def Macaulay(y):
    if y<0:
        mac=0
    else:
        mac=y
    return mac

def Macauley(y):
    if y<0:
        mac=0
    else:
        mac=y
    return mac

def Macaulay_power_0(y):
    if y<0:
        mac=0
    else:
        mac=1
    return mac

def matrix_solver(c,J,L,x1,x2,x3,xa,theta,t_st,t_sk,t_sp,w_st,ha,Py,Pz,d1,d2
    ,d3,G,E,Izz,Iyy,zshear, S,D, Td, DTd1,DTd2,DTd3,DTd4, FI1, FI2, FI3, FI4
    ):
    "UNKNOWNS - to be put in a matrix equation, this is the unknown vector -
        a solution (variable 'RES'). It is a vector of size 13x1"
    #R1y=1
    #R2y=1
    #R3y=1
    #R1z=1
    #R2z=1
    #R3z=1
    #Ay=1
    #Az=1
    #C1=1
    #C2=1
    #C3=1
    #C4=1
    #C5=1

    #NECESSARY INPUTS TO SOLVE THE SYSTEM:
        #ZSHEAR, Izz, Iyy, J
        #AND THE FOLLOWING 5 INTEGRALS
    #D=100
    #Td=100
    "Td = integral from 0 to chord | q(x,z)(z- zshear) dz. Done for each
        section by calculating the ersultant torque from aerodynamic load
        (acts through centroid - causes moment aroiund shear centre"
    "D=  double integral of ||q dx dx - performed by double integrator"
    #S=100
    "S=integral from 0 to x |q(x)dx. Integration to get force in Y
        direction"
    #DTd=100
    "DTd = double integral 0tox ||Td dxdx, which is just torque at each
        cross-section times distance x"
    #q_int=100; #quadruple integral over dx
    #FI=100;  #done by quadruple integrator
    #FI=fourth integral
    #Q=100  #not needed probably
    #Q is total aerodynamic loading sumsum(q(i,j))

    "unknown vector X=[R1y,R2y,R3y,R1z,R2z,R3z,Ay,Az,C1,C2,C3,C4,C5]"
    '''RHS vector Y with BCs'''

    Y=[D+Py*Macaulay(L-(x2+xa/2)),  #Mz(L)=0
       Pz*Macaulay(L-(x2+xa/2)),
           #My(L)=0
       -Td-(0-zshear+ha/2)*Py*Macaulay_power_0(L-(x2+xa/2))+(ha/2)*Pz*
           Macaulay_power_0(L-(x2+xa/2)),    #T(L)=0
       S+Py*Macaulay_power_0(L-(x2+xa/2)),      #Sy(L)=0
       Pz*Macaulay_power_0(L-(x2+xa/2)),       #Sz(L)=0
       d1*cos(theta) -FI1/(Izz*E) - Py/(E*Izz)*1/6*Macauley(x1-(x2+xa/2))**3
           -abs(zshear-0)*DTd1/(G*J) - abs(zshear-0)*(0-zshear+ha/2)/(G*J)*
           Py/6*Macauley(x1-(x2+xa/2))**3 ,      #v(x1) - twist(x1)*(0-
           zshear) = d1*cos(theta)
       -FI2/(Izz*E) - Py/(E*Izz)*1/6*Macauley(x2-(x2+xa/2))**3 -abs(zshear-0
           )*DTd2/(G*J) ,                   #v(x2) - twist(x2)*(0-zshear) =
           0
       d3*cos(theta) -FI3/(Izz*E) - Py/(E*Izz)*1/6*Macauley(x3-(x2+xa/2))**3
           -abs(zshear-0)*DTd3/(G*J) - abs(zshear-0)*(0-zshear+ha/2)/(G*J)*
           Py/6*Macauley(x3-(x2+xa/2))**3 + abs(zshear-0)*ha/2*1/(G*J)*Pz/6*
           Macauley(x3-(x2+xa/2))**3,      #v(x3) - twist(x3)*(0-zshear) =
           d3*cos(theta)
       -d1*sin(theta), #- DTd1/(G*J)*(0) - (0-zshear+ha/2)/
           (G*J)*(0)*Py*Macauley(x1-(x2+xa/2))**1 + ha/
           (2*G*J)*0*Pz*Macauley(x1-(x2+xa/2))**1,
           #w(x1) = -d1*sin(theta)
       0,  #- DTd2/(G*J)*(0) - (0-zshear+ha/2)/(G*J)*(0)*Py*Macauley(x2-
           (x2+xa/2))**1 + ha/(2*G*J)*0*Pz*Macauley(x2-(x2+xa/2))**1,
           #w(x2) = 0
       -d3*sin(theta)+Pz/(E*Iyy)*1/6*Macauley(x3-(x2+xa/2))**3, #- DTd3/
           (G*J)*(0) - (0-zshear+ha/2)/(G*J)*(0)*Py*Macauley(x3-(x2+xa/
           2))**1 + ha/(2*G*J)*0*Pz*Macauley(x3-(x2+xa/2))**1,
           #w(x3) = -d3*sin(theta)
       #Pz/(E*Iyy)*1/6*Macauley(x2-xa/2-(x2+xa/2))**3 - DTd4/(G*J)*(ha/2) -
           (0-zshear)/(G*J)*(ha/2)*Py*Macauley(x2-xa/2-(x2+xa/2))**1 + ha/
           (2*G*J)*ha/2*Pz*Macauley(x2-xa/2-(x2+xa/2))**1,
           #[w(x2-xa/2)+twist(x2-xa/2)*ha/2]cos(theta) + [v(x2-xa/2) + (0-
           zshear+ha/2)*twist(x2-xa/2)]*sin(theta)=0                at
           ACTUATOR 1
       #cos(theta)*(-DTd4/(G*J)*(ha/2)) + (-FI4/(Izz*E)  +(0-zshear+ha/
           2)*DTd4/(G*J))*sin(theta),
       cos(theta)*(-DTd4/(G*J)*(ha/2)) + (FI4/(Izz*E)  -(0-zshear+ha/2)*DTd4
           /(G*J))*sin(theta),
       0]          #Ay - Aztan(30) = 0
#This Y vector includes all boundary conditions - RHS
#The equation becomes XX*RES = Y
```

```
XX=np.matrix([[Macaulay(L-x1),Macaulay(L-x2),Macaulay(L-x3),0,0,0,-
    Macaulay(L-(x2-xa/2)),0,0,0,0,0,0],
              [0,0,0,Macaulay(L-x1),Macaulay(L-x2),Macaulay(L-x3),0,-
                  Macaulay(L-(x2-xa/2)),0,0,0,0,0],
              [ -(0-zshear)*Macaulay_power_0(L-x1), -(0-zshear)*
                  Macaulay_power_0(L-x2), -(0-zshear)*Macaulay_power_0(L
                  -x3),0,0,0,(0-zshear+ha/2)*Macaulay_power_0(L-(x2-xa/2
                  )), -(ha/2)*Macaulay_power_0(L-(x2-xa/2)),0,0,0,0,0],
              [Macaulay_power_0(L-x1),Macaulay_power_0(L-x2),
                  Macaulay_power_0(L-x3),0,0,0,-Macaulay_power_0(L-(x2-
                  xa/2)),0,0,0,0,0,0],
              [0,0,0,Macaulay(L-x1)**0, Macaulay(L-x2)**0, Macaulay(L-x3
                  )**0,0,-Macaulay(L-(x2-xa/2))**0, 0,0,0,0,0],
              [0,0,0,  0,0,0, 0,0,x1,1,0,0,-(0-zshear)],
              [-1/(E*Izz)*1/6*(Macauley(x2-x1)**3)+abs(zshear-0)*1/(G*J)
                  *(0-zshear)*Macaulay(x2-x1)**1, -1/(E*Izz)*1/6*
                  (Macauley(x2-x2)**3)+abs(zshear-0)*1/(G*J)*(0-zshear)*
                  Macaulay(x2-x2)**1,0, 0,0,0, 1/(E*Izz)*1/6*Macauley(x2
                  -(x2-xa/2))**3 -abs(zshear-0)/(G*J)*(0-zshear+ha/2)*
                  Macaulay(x2-(x2-xa/2))**1,+abs(zshear-0)/(G*J)*(ha/2)*
                  Macaulay(x2-(x2-xa/2))**1,x2,1,0,0,-(0-zshear)],
              [-1/(E*Izz)*1/6*(Macauley(x3-x1)**3)+abs(zshear-0)*1/(G*J)
                  *(0-zshear)*Macaulay(x3-x1)**1, -1/(E*Izz)*1/6*
                  (Macauley(x3-x2)**3)+ abs(zshear-0)*1/(G*J)*(0-zshear)
                  *Macaulay(x3-x2)**1, -1/(E*Izz)*1/6*(Macauley(x3-x3)**
                  3)+abs(zshear-0)*1/(G*J)*(0-zshear)*Macaulay(x3-x3)**1
                  , 0,0,0, 1/(E*Izz)*1/6*Macauley(x3-(x2-xa/2))**3 -abs
                  (zshear-0)/(G*J)*(0-zshear+ha/2)*Macaulay(x3-(x2-xa/2)
                  )**1,+abs(zshear-0)/(G*J)*(ha/2)*Macaulay(x3-(x2-xa/2)
                  )**1,x3,1,0,0,-(0-zshear)],
              [0, 0, 0,  1/(E*Iyy)*1/6*(Macauley(x1-x1)**3), 1/(E*Iyy)*1
                  /6*(Macauley(x1-x2)**3), 1/(E*Iyy)*1/6*(Macauley(x1-x3
                  )**3),    0, -1/(E*Iyy)*1/6*Macauley(x1-(x2-xa/2))**3,
                  0,0,x1,1,0  ],
              [0, 0, 0,  1/(E*Iyy)*1/6*(Macauley(x2-x1)**3), 1/(E*Iyy)*1
                  /6*(Macauley(x2-x2)**3), 1/(E*Iyy)*1/6*(Macauley(x2-x3
                  )**3),    0, -1/(E*Iyy)*1/6*Macauley(x2-(x2-xa/2))**3,
                  0,0,x2,1,0  ],
              [0, 0, 0,  1/(E*Iyy)*1/6*(Macauley(x3-x1)**3), 1/(E*Iyy)*1
                  /6*(Macauley(x3-x2)**3), 1/(E*Iyy)*1/6*(Macauley(x3-x3
                  )**3),    0, -1/(E*Iyy)*1/6*Macauley(x3-(x2-xa/2))**3,
                  0,0,x3,1,0],
              #+[sin(theta)*(-1/(E*Izz)*1/6*(Macauley((x2-xa/2)-x1)**3)+
                  (0-zshear+ha/2)*1/(G*J)*(0-zshear)*Macaulay(x2-xa/2-
                  x1)**1) +cos(theta)*(-ha/(2*G*J)*(0-
                  zshear)*Macaulay(x2-xa/2-x1)**1), 0, 0,cos(theta)*(1/
                  (E*Iyy)*1/6*(Macauley(x2-xa/2-x1)**3)),0, 0, 0, 0,
                  sin(theta)*(x2-xa/2),sin(theta),cos(theta)*(x2-xa/
                  2),cos(theta),-(0-zshear+ha/2)*sin(theta)+ha/
                  2*cos(theta)],
              [sin(theta)*(1/(E*Izz)*1/6*(Macauley((x2-xa/2)-x1)**3)-(0-
                  zshear+ha/2)*1/(G*J)*(0-zshear)*Macaulay(x2-xa/2-x1)**
                  1) +cos(theta)*(-ha/(2*G*J)*(0-zshear)*Macaulay(x2-xa/
                  2-x1)**1), 0, 0,cos(theta)*(-1/(E*Iyy)*1/6*(Macauley
                  (x2-xa/2-x1)**3)),0, 0, 0, 0, -sin(theta)*(x2-xa/2),-
                  sin(theta),cos(theta)*(x2-xa/2),cos(theta),(0-zshear+
                  ha/2)*sin(theta)+ha/2*cos(theta)],
              [0,0,0, 0,0,0, 1,-tan(theta), 0,0,0,0,0]])
#RES=[R1y,R2y,R3y,R1z,R2z,R3z,Ay,Az,C1,C2,C3,C4,C5]
RES=np.linalg.solve(XX,Y)
```

```
    return RES

"Moment around z,y axis as a function of x, as well as torque, shear forces
    and twist angle as functions of x"



def moment_z(x,R1y,R2y,R3y,Ay,Py,D,x1,x2,x3,xa):
    Mz = -D + R1y*Macaulay(x-x1) + R2y*Macaulay(x-x2) - Py*Macaulay(x-(x2+xa
        /2)) - Ay*Macaulay(x-(x2-xa/2)) + R3y*Macaulay(x-x3)
    return(Mz)
def moment_y(x,R1z,R2z,R3z,Az,Pz,x1,x2,x3,xa):
    My = -R1z*Macaulay(x-x1) + Pz*Macaulay(x-(x2+xa/2)) - R2z*Macaulay(x-x2)
        - R3z*Macaulay(x-x3)+ Az*Macaulay(x-(x2-xa/2))
    return(My)
def torque_x(x,Td,zshear,R1y,R2y,R3y,Ay,Az,Py,Pz,x1,x2,x3,xa,ha):
    T = -(-Td - (0-zshear)*R1y*Macaulay_power_0(x-x1) - (0-zshear)*R2y*
        Macaulay_power_0(x-x2) - (0-zshear)*R3y*Macaulay_power_0(x-x3) + (0-
        zshear+ha/2)*Py*Macaulay_power_0(x-(x2+xa/2)) - (ha/2)*Pz*
        Macaulay_power_0(x-(x2+xa/2)) - (ha/2)*Az*Macaulay_power_0(x-(x2-xa/
        2)) + (0-zshear+ha/2)*Ay*Macaulay_power_0(x-(x2-xa/2)))
    return(T)
def shear_y(x,R1y,R2y,R3y,Ay,Py,S,x1,x2,x3,xa):
    Sy = -S + R1y*Macaulay_power_0(x-x1) + R2y*Macaulay_power_0(x-x2) - Py*
        Macaulay_power_0(x-(x2+xa/2)) - Ay*Macaulay_power_0(x-(x2-xa/2)) +
        R3y*Macaulay_power_0(x-x3)
    return(Sy)
def shear_z(x,R1z,R2z,R3z,Az,Pz,x1,x2,x3,xa):
    Sz = -R1z*Macaulay_power_0(x-x1) - R2z*Macaulay_power_0(x-x2) + Pz*
        Macaulay_power_0(x-(x2+xa/2)) + Az*Macaulay_power_0(x-(x2-xa/2)) -
        R3z*Macaulay_power_0(x-x3)
    return(Sz)
def twist(x,G,J,DTd,R1y,R2y,R3y,Py,Pz,Ay,Az,C5,x1,x2,x3,xa,ha,zshear):
    twist = -(1/(G*J)*(DTd - (0-zshear)*R1y*Macaulay(x-x1)**1 - (0-zshear)*
        R2y*Macaulay(x-x2)**1 - (0-zshear)*R3y*Macaulay(x-x3)**1 + (0-zshear
        +ha/2)*Py*Macaulay(x-(x2+xa/2))**1 - (ha/2)*Pz*Macaulay(x-(x2+xa/2))
        **1 - (ha/2)*Az*Macaulay(x-(x2-xa/2))**1 + (0-zshear+ha/2)*Ay*
        Macaulay(x-(x2-xa/2))**1 ) + C5)
    return(twist)
    #deflection functions as functions of x in both y and z directions (v in
        y direction, w in z direction)


def v_prime(x,E,Izz,q_int_3,R1y,R2y,R3y,Py,Ay,C1,x1,x2,x3,xa):
    v_prim = -1/(E*Izz)*(-q_int_3 + R1y/2*(Macauley(x-x1)**2) + R2y/2*
        (Macauley(x-x2)**2) - Py/2*Macauley(x-(x2+xa/2))**2 - Ay/2*Macauley
        (x-(x2-xa/2))**2 + R3y/2*(Macauley(x-x3)**2)) + C1
    return v_prim

def w_prime(x,E,Iyy,R1z,R2z,R3z,Pz,Az,C3,x1,x2,x3,xa):
    w_prim = -1/(E*Iyy)*(-R1z/2*(Macauley(x-x1)**2) - R2z/2*(Macauley(x-x2)*
        *2) + Pz/2*Macauley(x-(x2+xa/2))**2 + Az/2*Macauley(x-(x2-xa/2))**2
        + R3z/2*(Macauley(x-x3)**2)) + C3
    return w_prim

def v(x,E,Izz,q_int_4,R1y,R2y,R3y,Py,Ay, C1,C2,x1,x2,x3,xa):
    v = -1/(E*Izz)*(-q_int_4 + R1y/6*(Macauley(x-x1)**3) + R2y/6*(Macauley(x
        -x2)**3) - Py/6*Macauley(x-(x2+xa/2))**3 - Ay/6*Macauley(x-(x2-xa/2)
        )**3 + R3y/6*(Macauley(x-x3)**3)) + C1*x + C2
    return v
```

```
def w(x,E,Iyy,R1z,R2z,R3z,Pz,Az,C3,C4,x1,x2,x3,xa):
    w = -1/(E*Iyy)*(-R1z/6*(Macauley(x-x1)**3) - R2z/6*(Macauley(x-x2)**3) +
        Pz/6*Macauley(x-(x2+xa/2))**3 +Az/6*Macauley(x-(x2-xa/2))**3 -R3z/6*
        (Macauley(x-x3)**3)) + C3*x + C4
    return w

#Stress
```

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 19 09:01:46 2020

@author: Marni
"""
from math import *
import matplotlib as plt

#Pseudo code to calculate shear stresses in the cross section
#arrays need to match in order to implement in the main

q_circle1,q_spar1,q_top,q_bottom,q_spar2,q_circle2 = section.get_shearflows
    ()

normal_stresses = np.array([])
shear_stresses = np.array([])
von_Mises = np.array([])
for i in range(new_aerodata.shape[1]):

    normal_stresses_crosssection = np.array([])
    shear_stresses_crosssection = np.array([])
    von_Mises_crosssection = np.array([])

    V_y = V_y[i]
    W_z = W_z[i]
    T_x = T_x[i]
    #solving for shear flows due to torque
    #effect of stiffeners excluded in calculating qs_0 due to torque
    Y=[T_x[i],
       0]
    A=np.matrix([[2*A1, 2*A2],
                 [1/(2*A1*G)*((pi*ha/2 - 3*w_st)/t_sk + 3*w_st/(t_sk+t_st))
                     + ha/(2*A2*G*t_sp) + ha/(2*A1*G*t_sp), - 1/(2*A2*G)*((2
                     *sqrt((ha/2)**2+(c-ha/2)**2) - 8*w_st)/t_sk + 8*w_st/
                     (t_sk+t_st)) - ha/(2*A2*G*t_sp) - ha/(2*A1*G*t_sp)]])
    shears=np.linalg.solve(A,Y)

    q_circle0=shears[0]
    q_triangle0=shears[1]

    for j in range(len(crosssection_z)):

        z = crosssection_z[j]
        y = crosssection_y[j]

        sigma_xx = M_z[i]*y/Izz + M_y[i]*(z-z_centroid)/Iyy

        normal_stresses_crosssection = np.append
            (normal_stresses_crosssection,sigma_xx)
        #condition (j<5000 etc.) to make sure that calculations are made in
            the correct part of the aileron
        #shear flow array must match with a coordinate (yz plane) arrays -
            z = crosssection_z[j] and y = crosssection_y[j]
        if j<5000:
            tau = (q_circle1[j] + q_circle0)/tsk   #section 1
        if j>5000 and j<10000:
            tau = (q_spar1[j] - q_circle0 + q_triangle0)/tsp   #section 2
        if j>1000:
            tau = (q_top[j] + q_triangle0)/tsk   #section3
        if j>10000:
            tau = (q_bottom[j] + q_triangle0)/tsk   #section 4
        if j>10000:
            tau = (q_spar2[j - q_circle0 + q_triangle0)/tsp   #section 5
        if j>100000:
            tau = (q_circle2 + q_circle0)/tsk                   #section 6
        #shear stress is due to both shear forces Vy and Vz, which are
            included in q_circle and q_spar etc. arrays
        #..and due to torque, which adds q_circle0 and/or q_triangle0
            contributions

        shear_stresses_crosssection = np.append(shear_stresses_crosssection,
            tau)
        von_Mises_crosssection=np.append(von_Mises_crosssection,sqrt
            (sigma_xx**2 + 3*tau**2))







ha = 0.161
ca = 0.505
z = 0
z_hat = 0.25
Mx = 1
My = 2
Mz = 3
x = 1

Ixx = 1
Iyy = 1
Izz = 1

def y(z):
    if z<ha/2:
        ys = ((ha/2)**2-(ha/2-z)**2)**0.5
    if z>=ha/2:
        ys = ha/2-(z-ha/2)*(ha/2)/(ca-ha/2)
    return ys

y_tot = []
z_tot = []

#while z<=ca:
#    y = profile_y(z)
#    z+=0.001
 #   y_tot.append(y)
 #   z_tot.append(z)

#plt.pyplot.plot(z_tot,y_tot)

def stress_xx(z):
    s = My*(z-z_hat)/Iyy - Mz*y(z)/Izz
    return s

def stress_yy(z):
    s = -Mx*z/Ixx + Mz*x/Izz
    return s

def stress_zz(z):
    s = -My*(x)/Iyy + Mx*y(z)/Ixx
    return s

def stress_vm(x,y,z):
    s_vm = (((sigma_xx(z) - sigma_yy(z))**2 + (sigma_zz - sigma_xx)**2 + 6*
        (Txy**2 + Tyz**2 + Txz**2))/2)**0.5
    return s_vm

s_xx = []
s_yy = []
s_zz = []
zs = []
z = 0

while z<ca:
    s_xx.append(stress_xx(z))
    s_yy.append(stress_yy(z))
    s_zz.append(stress_zz(z))
    zs.append(z)
    z+=0.001

#plt.pyplot.plot(zs,s_xx)
#plt.pyplot.plot(zs,s_yy)
#plt.pyplot.plot(zs,s_zz)
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 24 19:49:50 2020

@author: pablo
"""


#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 20 15:01:21 2020

@author: pablo
"""



from math import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
import time

import Loading
import Interpolation as inter
import Read
#import final_integration as integration
import SectionalProperties as section

# Start timer to measure runtime
start_time = time.time()


# ----------------- PARAMETERS -----------------
Ca = 0.505 #[m]
La = 1.611 #[m]
ha = 0.161 #[m]
z_hingeline = ha/2 #[m]
t_st = 0.0012 #[m]
w_st = 0.017 #[m]
h_st = 0.013 #[m]
A_st = t_st * (w_st + h_st) #[m^2]
t_sk = 0.0011 #[m]
t_sp = 0.0024 #[m]
nstiff = 11 #[-]

x_1 = 0.125
x_2 = 0.498
x_3 = 1.494 #m
x_a = 0.245    #m
x_4 = x_2-0.5*x_a
theta=radians(30)   #degrees

r = 0.5*ha
A_circ = r**2*pi/2
A_triang = (Ca-r)*r
```

```
Py = 49200*sin(theta)
Pz = 49200*cos(theta)   #N
d1=0.00389   #m
d2=0
d3=0.01245    #m
G = 28*10**9    #in Pa
E=73.1*10**9     #in Pa

aircraft = 'F100'




# ----------------- SECTIONAL PROPERTIES  -----------------

Izz, Iyy, z_centroid = section.get_MoI(ha,Ca,t_st,w_st,h_st,A_st,t_sk,t_sp,
    nstiff) # Moments of Inertia
#z_sc = section.get_SC(t_sk, t_sp, ha, Ca, Izz) # Z-location of shear center
z_sc = -0.005104143830014916
J = section.get_J(G, ha, t_sk, w_st, t_sp, t_st, Ca)


# ----------------- AERODYNAMIC DATA -----------------

print('\n')
print('\n')
print('Extracting aerodynamic data from .txt file...')
print('\n')
print('\n')

# Get original aerodynamic data (before interpolation)
original_aerodata,x_coord,z_coord = Read.get_data(aircraft, Ca, La)
original_aerodata = original_aerodata*1000 #Transform aerodynamic data from
    kPa to Pa
original_shape = original_aerodata.shape # Shape of data in the form (81,41)



# ----------------- INTERPOLATION STEP 0 -----------------
# Obtain preliminary values for interpolation

# Desired resolutions, in [mm]
r_span = 2
r_chord = 5

# Get value of n for both chord and span (= number of nodes - 1)
n_chord = original_shape[0] - 1 # Should be 80
n_span = original_shape[1] - 1 # Should be 40


# Get number of points PER SPLINE in both directions from resolutions
n_points_chord = inter.get_number_of_points(z_coord, r_chord)
n_points_span = inter.get_number_of_points(x_coord, r_span)

n_internal_chord = n_points_chord -2
n_internal_span = n_points_span - 2

total_points_chord = (n_chord + 1) + (n_internal_chord)*n_chord
total_points_span = (n_span + 1) + (n_internal_span)*n_span
```

```
    # Initialize matrix with interpolated nodes
    new_nodes_x = np.array([])
    new_nodes_z = np.array([])


    # ------------------ INTERPOLATION STEP 1 ------------------
    # (Go from 81x41 to 500x41)
    print('\n')
    print('\n')

    # Initialize matrix with intermidiate aerodynamic data (1/2 interpolation
        steps)
    intermidiate_aerodata = np.zeros((total_points_chord,original_shape[1]))


    #Loop through spanwise cross-sections
    for i in range(n_span+1):

        percentage=round(50*i/n_span,2)

        if percentage%5==0:
            print('Interpolating aerodynamic data... '+str(percentage)+'%'+'\n')


        # Select individual spanwise cross section from original aerodymic data
        cross_sectional_loading = original_aerodata[:,i]

        # Interpolate in chordwise direction
        cross_sectional_interpolants = inter.find_interpolants(z_coord,
            cross_sectional_loading)


        # New cross sectional values and nodes after interpolation
        new_cross_sectional_nodes, new_cross_sectional_loads = inter.new_loading
            (z_coord,cross_sectional_interpolants,n_points_chord)

        # Keep record of new z coordinates only once
        if i==0:
             new_nodes_z = new_cross_sectional_nodes

        # Append to matrix
        intermidiate_aerodata[:,i] = new_cross_sectional_loads




    # ------------------ INTERPOLATION STEP 2 ------------------
    # (Go from 500x41 to 500x100)

    # Initialize matrix with final aerodynamic data (2/2 interpolation steps)
    new_aerodata = np.zeros((total_points_chord,total_points_span))


    #Loop through chordwise cross-sections
    for i in range(intermidiate_aerodata.shape[0]):

        percentage = 50 + 50*i/(intermidiate_aerodata.shape[0]-1)
        if percentage%5==0:
            print('Interpolating aerodynamic data... '+str(round(percentage,2))+
                '%'+'\n')


        # Select individual chordwise cross section from intermidiate
            aerodynamic data
        cross_sectional_loading = intermidiate_aerodata[i,:]

        # Interpolate in spanwise direction
        cross_sectional_interpolants = inter.find_interpolants(x_coord,
            cross_sectional_loading)

        # New cross sectional values and nodes after interpolation
        new_cross_sectional_nodes, new_cross_sectional_loads = inter.new_loading
            (x_coord,cross_sectional_interpolants,n_points_span)

        # Keep record of new x coordinates only once
        if i==0:
             new_nodes_x = new_cross_sectional_nodes

        # Append to matrix
        new_aerodata[i,:] = new_cross_sectional_loads



    # ----------------- COORD. SYSTEM ADJUSTMENTS -----------------
    # Flip z-axis direction + translate to match chosen coordinate system
    # By doing this, the positive z-axis points towards the LE and starts
    # at the hinge line.

    new_nodes_z = -1*new_nodes_z + z_hingeline



    # ----------------- RESULTANT LOAD CALCULATIONS -----------------
    # Loop through spanwise cross sections, calculate resultant force and
        centroid.

    # Z-coordinates of centroids of each spanwise cross section
    centroids_spanwise = np.array([])

    # Resultant forces PER UNIT SPAN at each spanwise cross section
    resultant_forces = np.array([])

    # Torque at each spanwise crosssection
    torques = np.array([])
    #torques2=np.array([])

    # Aerodynamic force generated at each point
    aero_loads = np.zeros(new_aerodata.shape)

    # Spanwise spacing -- steps in x direction
    delta_x = np.array([])

    print('\n')
    print('\n')
    # Loop through cross sections
    for i in range(new_aerodata.shape[1]):
        percentage = 100*i/(new_aerodata.shape[1]-1)

        if percentage%5==0:
            print('Computing resultant forces at grid nodes... '+str(round
                (percentage,2))+'%\n')


        q = new_aerodata[:,i] # Pressure distribution

        # ADD -- Convert q to force by multiplying by d_x,d_z (area in between
            spanwise crossections)
        if i ==0:
            delta_span = new_nodes_x[2]-new_nodes_x[1]
            delta_x = np.append(delta_x,delta_span)
        else:
            delta_span = new_nodes_x[i] - new_nodes_x[i-1]
            delta_x = np.append(delta_x,delta_span)

        torque = 0
        resultant = 0

        for j in range(new_aerodata.shape[0]):

            if j == 0:
                delta_chord = abs(new_nodes_z[j]-z_hingeline)
            else:
                delta_chord = abs(new_nodes_z[j] - new_nodes_z[j-1])

            area = delta_span*delta_chord

            load = q[j]*area
            aero_loads[j,i] = load

            torque_i = load*(new_nodes_z[j]-z_sc)
            torque += torque_i

            resultant += load

        centroid = np.sum(np.multiply(aero_loads[:,i],new_nodes_z))/resultant
        centroids_spanwise = np.append(centroids_spanwise,centroid)
        #torques2=np.append(torques2, resultant*centroid)
        resultant_forces = np.append(resultant_forces,resultant)
        torques = np.append(torques,torque)

    #-----------------------------------------

    '''

    # ----------------- PLOT COUNTOUR MAP AND 3D SURFACE OF INTERPOLATED DATA --
        ---------------

    X, Z = np.meshgrid(new_nodes_x,new_nodes_z)
    Y = new_aerodata/1000

    # 2D Countour Plot
    plt.figure()
    countour_plot = plt.contourf(X,Z,Y)
    plt.colorbar(countour_plot)

    plt.title('Contour plot of interpolated aerodynamic loading distribution
        [kPa]',pad=10)
    plt.xlabel('X axis (Spanwise direction)')
    plt.ylabel('Z axis (Chordwise direction)')
    plt.show()


    # 3D Surface Plot
    ax = plt.axes(projection='3d')
    ax.plot_surface(X,Z,Y,cmap='plasma')
    ax.set_title('Interpolated aerodynamic distribution [kPa]', pad=20)

    #fontname='AppleGothic'
    plt.xlabel('X-axis (Spanwise direction)', labelpad=10)
    plt.ylabel('Z-axis (Chordwise direction)', labelpad=10)

    plt.show()









    '''

    # ----------------- COMPUTING INTEGRALS REQUIRED TO SOLVE S.O.E.
        -----------------

    # S -- First integral of q w.r.t x
    total_resultants = np.cumsum(resultant_forces)
    total_resultant_force = total_resultants[-1]

    # D -- Second integral of q w.r.t x
    # [F_0 * x_0, ..., F_n * x_n]
    aero_moment_z_contributions = np.multiply(resultant_forces,new_nodes_x)
    aero_moment_z = np.cumsum(aero_moment_z_contributions)
    #D = integration.integrate(new_nodes_x,resultant_forces,'double')

    #Td is the last torque
    Td_contributions = torques
    Td = np.cumsum(Td_contributions)

    #Indicies for DTd1, DTd2, DTd3
    index_x1 = np.where(new_nodes_x>x_1)[0][0]
    index_x2 = np.where(new_nodes_x>x_2)[0][0]
    index_x3 = np.where(new_nodes_x>x_3)[0][0]
    index_x4 = np.where(new_nodes_x>x_4)[0][0]

    x1_nodes = new_nodes_x[0:index_x1]
    x2_nodes = new_nodes_x[0:index_x2]
    x3_nodes = new_nodes_x[0:index_x3]
    x4_nodes = new_nodes_x[0:index_x4]

    # DTd -- First integral of T w.r.t x
    DTd_contributions = np.multiply(Td_contributions,delta_x)
    DTd = np.cumsum(DTd_contributions)
```

```
    # DTd1, DTd2, DTd3
    DTd1 = DTd[index_x1]
    DTd2 = DTd[index_x2]
    DTd3 = DTd[index_x3]
    DTd4 = DTd[index_x4]

    # Third integral of q w.r.t x
    aero_deflection_slope_contributions =  np.multiply(aero_moment_z,delta_x)
    aero_deflection_slope = np.cumsum(aero_deflection_slope_contributions)


    # FI1  -- Quadruple integral of q w.r.t x
    aero_deflection_contributions = np.multiply(aero_deflection_slope,delta_x)
    aero_deflection = np.cumsum(aero_deflection_contributions)


    FI1 = aero_deflection[index_x1]
    FI2 = aero_deflection[index_x2]
    FI3 = aero_deflection[index_x3]
    FI4 = aero_deflection[index_x4]


    # ----------------- SOLVING S.O.E. FOR REACTION FORCES AND INTEGRATION
        CONSTANTS -----------------



    #RES=[R1y,R2y,R3y,R1z,R2z,R3z,Ay,Az,C1,C2,C3,C4,C5]
    RES = Loading.matrix_solver(Ca,J,La,x_1,x_2,x_3,x_a,theta,t_st,t_sk,t_sp,
        w_st,ha,Py,Pz,d1,d2,d3,G,E,Izz,Iyy,z_sc,total_resultant_force,
        aero_moment_z[-1],Td[-1],DTd1,DTd2,DTd3,DTd4,FI1,FI2,FI3,FI4)

    R1y = RES[0]
    R2y = RES[1]
    R3y = RES[2]
    R1z = RES[3]
    R2z= RES[4]
    R3z= RES[5]
    Ay = RES[6]
    Az= RES[7]
    C1 = RES[8]
    C2= RES[9]
    C3= RES[10]
    C4 = RES[11]
    C5 = RES[12]


    #print([sin(theta)*(-1/(E*Izz)*1/6*(Macauley((x2-xa/2)-
        x1)**3)+abs(z_sc-0)*1/(G*J)*(0-z_sc)*Macaulay(x2-xa/2-x1)**1)
        +cos(theta)*(-ha/(2*G*J)*(0-z_sc)*Macaulay(x2-xa/2-x1)**1),
        sin(theta)*(-1/(E*Izz)*1/6*(Macauley(x2-xa/2-x2)**3)+ abs(z_sc-0)*1/
        (G*J)*(0-z_sc)*Macaulay(x2-xa/2-x2)**1)+cos(theta)*(-ha/(2*G*J)*(0-
        z_sc)*Macaulay(x2-xa/2-x2)**1), sin(theta)*(-1/(E*Izz)*1/6*(Macauley(x2-
        xa/2-x3)**3)+abs(z_sc-0)*1/(G*J)*(0-z_sc)*Macaulay(x2-xa/2-x3)**1)+
        cos(theta)*(-ha/(2*G*J)*(0-z_sc)*Macaulay(x2-xa/2-
        x3)**1),cos(theta)*(-1/(E*Iyy)*1/6*(Macauley(x2-xa/2-x1)**3)),
        cos(theta)*(-1/(E*Iyy)*1/6*(Macauley(x2-xa/2-x2)**3)), cos(theta)*(-1/
        (E*Iyy)*1/6*(Macauley(x2-xa/2-x3)**3)), sin(theta)*(1/
        (E*Izz)*1/6*Macauley(x2-xa/2-(x2-xa/2))**3 -abs(z_sc-0)/(G*J)*(0-
        z_sc+ha/2)*Macaulay(x2-xa/2-(x2-xa/2))**1) + cos(theta)*( ha/(2*G*J)*(0-
        z_sc+ha/2)*Macaulay(x2-xa/2-(x2-xa/2))**1), sin(theta)*(+abs(z_sc-0)/
        (G*J)*(ha/2)*Macaulay(x2-xa/2-(x2-xa/2))**1) +cos(theta)*(1/
        (E*Iyy)*1/6*Macauley(x2-xa/2-(x2-xa/2))**3-ha/(2*G*J)*(ha/
        2)*Macaulay(x2-xa/2-(x2-xa/2))**1), sin(theta)*(x2-xa/
        2),sin(theta),cos(theta)*(x2-xa/2),cos(theta),sin(theta)+cos(theta)])


    # ----------------- COMPUTE MOMENTS AND DEFLECTIONS AS A 'FUNCTION' OF X
        -----------------
    print('\n')
    print('\n')
    M_y = np.array([])
    M_z = np.array([])
    T_x = np.array([]) #Note that this torque includes the torque due to aero
        loading and that due to reaction forces
    S_y = np.array([])
    S_z = np.array([])
    V_y = np.array([])
    V_y_prime = np.array([])
    W_z = np.array([])
    W_z_prime = np.array([])
    twist = np.array([])

    # Loop through spanwise cross sections
    for i in range(new_aerodata.shape[1]):

        percentage = 100*i/(new_aerodata.shape[1]-1)

        if percentage%5==0:
            print('Computing shear forces, moments, displacements, and twist...
                '+str(round(percentage,2))+'%\n')


        #Calculate required integrals
        D_i = aero_moment_z[i]
        Td_i = Td[i]
        DTd_i = DTd[i]
        S_i = total_resultants[i]
        aero_deflection_slope_i = aero_deflection_slope[i]
        FI_i = aero_deflection[i]

        # Moments and torques
        M_z_i = Loading.moment_z(new_nodes_x[i], R1y,  R2y, R3y, Ay, Py, D_i,x_1
            ,x_2,x_3,x_a)
        M_y_i = Loading.moment_y(new_nodes_x[i], R1z, R2z, R3z, Az, Pz,x_1,x_2,
            x_3,x_a)
        T_x_i = Loading.torque_x(new_nodes_x[i], Td_i, z_sc, R1y, R2y, R3y, Ay,
            Az, Py, Pz,x_1,x_2,x_3,x_a,ha)

        # Shear forces
        S_y_i = Loading.shear_y(new_nodes_x[i], R1y, R2y, R3y, Ay, Py, S_i,x_1,
            x_2,x_3,x_a)
        S_z_i = Loading.shear_z(new_nodes_x[i], R1z, R2z, R3z, Az, Pz, x_1,x_2,
            x_3,x_a)

        # Deflections
        V_y_i = Loading.v(new_nodes_x[i],E,Izz,FI_i,R1y,R2y,R3y,Py,Ay,C1,C2,x_1,
            x_2,x_3,x_a)
        V_y_prime_i = Loading.v_prime(new_nodes_x[i], E, Izz,
            aero_deflection_slope_i, R1y, R2y, R3y, Py, Ay, C1,x_1,x_2,x_3,x_a)
        W_z_i = Loading.w(new_nodes_x[i],E,Iyy,R1z,R2z,R3z,Pz,Az,C3,C4,x_1,x_2,
            x_3,x_a)
        W_z_prime_i = Loading.w_prime(new_nodes_x[i], E, Iyy, R1z, R2z, R3z, Pz,
            Az, C3,x_1,x_2,x_3,x_a)

        # Twist

        twist_i = Loading.twist(new_nodes_x[i], G, J, DTd_i, R1y, R2y, R3y, Py,
            Pz, Ay, Az, C5,x_1,x_2,x_3,x_a,ha,z_sc)


        # Appending to arrays
        M_z = np.append(M_z,M_z_i)
        M_y = np.append(M_y,M_y_i)
        T_x = np.append(T_x,T_x_i)
        S_y = np.append(S_y,S_y_i)
        S_z = np.append(S_z,S_z_i)
        V_y = np.append(V_y,V_y_i)
        V_y_prime = np.append(V_y_prime,V_y_prime_i)
        W_z = np.append(W_z,W_z_i)
        W_z_prime = np.append(W_z_prime,W_z_prime_i)
        twist = np.append(twist,twist_i)




    '''
    # ----------------- GENERATE DEFLECTION PLOTS -----------------
    print(np.max(twist))
    plt.figure()
    plt.plot(new_nodes_x,T_x)
    plt.figure()
    plt.plot(new_nodes_x,twist)
    plt.show()

    '''


    # ----------------- STRESS CALCULATIONS -----------------

    # Get discretization of cross section -- (z,y) coordiantes of points
    circle1,circle2,spar1,spar2,top,bottom = section.discretize_crosssection
        (1000)

    circle_1_z = circle1[0]
    circle_1_y = circle1[1]

    circle_2_z = circle2[0]
    circle_2_y = circle2[1]

    spar_1_z = spar1[0]
    spar_1_y = spar1[1]

    spar_2_z = spar2[0]
    spar_2_y = spar2[1]

    top_z = top[0]
    top_y = top[1]
    bottom_z = bottom[0]
    bottom_y = bottom[1]

    crosssection_z = np.concatenate((circle_1_z,spar_1_z,top_z,bottom_z,spar_2_z
        ,circle_2_z))
    crosssection_y = np.concatenate((circle_1_y,spar_1_y,top_y,bottom_y,spar_2_y
        ,circle_2_y))


    # Get shear flows due to shear forcesat each points
    q_circle1,q_spar1,q_top,q_bottom,q_spar2,q_circle2 = section.get_shearflows
        (V_y[384], W_z[384])


    # Array containing one subarray per cross section. Each subrray contains
        60000 points
    normal_stresses = np.array([])
    shear_stresses = np.array([])
    vm_stresses = np.array([])

    # Loop through spanwise crossections
    for i in range(new_aerodata.shape[1]):

        percentage = 100*i/(new_aerodata.shape[1]-1)

        if percentage%5==0:
            print('Calculating shear flow and Von Mises stress distributions...
                '+str(round(percentage,2))+'%\n')


        normal_stresses_crosssection = np.array([])
        shear_stresses_crosssection = np.array([])
        vm_stresses_crosssection = np.array([])

        #V_y = V_y[i]
        #W_z = W_z[i]
        #T_x = T_x[i]


        # ------- SHEAR STRESSES -------

        # Compute shear flows due to torque
        # Effect of stiffeners excluded in calculation
        Y=[T_x[i],0]
        A=np.matrix([[2*A_circ, 2*A_triang],[1/(2*A_circ*G)*((pi*ha/2)/t_sk) +
            ha/(2*A_triang*G*t_sp) + ha/(2*A_circ*G*t_sp), - 1/(2*A_triang*G)*
            ((2*sqrt((ha/2)**2+(Ca-ha/2)**2))/t_sk) - ha/(2*A_triang*G*t_sp) -
            ha/(2*A_circ*G*t_sp)]])
        shears=np.linalg.solve(A,Y)

        q_circle0=shears[0]
        q_triangle0=shears[1]

        # ORDER: Circle 1, Spar 1, Top, Bottom, Spar 2, Circle 2

        # Circle 1
        for j in range(len(circle_1_z)):
            z = circle_1_z[j]
            y = circle_1_y[j]
            tau = (q_circle1[j]+q_circle0)/t_sk
            shear_stresses_crosssection = np.append(shear_stresses_crosssection,
                tau)

        # Spar 1
        for j in range(len(spar_1_z)):
            z = spar_1_z[j]
            y = spar_1_y[j]

            tau = (q_spar1[j]-q_circle0+q_triangle0)/t_sp
            shear_stresses_crosssection = np.append(shear_stresses_crosssection,
                tau)

        # Top
        for j in range(len(top_z)):
            z = top_z[j]
            y = top_y[j]

            tau = (q_top[j]+q_triangle0)/t_sk
            shear_stresses_crosssection = np.append(shear_stresses_crosssection,
                tau)

        # Bottom
        for j in range(len(bottom_z)):
            z = bottom_z[j]
            y = bottom_y[j]

            tau = (q_bottom[j]+q_triangle0)/t_sk
            shear_stresses_crosssection = np.append(shear_stresses_crosssection,
                tau)

         # Spar 2
        for j in range(len(spar_2_z)):
            z = spar_2_z[j]
            y = spar_2_y[j]

            tau = (q_spar2[j]-q_circle0+q_triangle0)/t_sp
            shear_stresses_crosssection = np.append(shear_stresses_crosssection,
                tau)

        # Circle 2
        for j in range(len(circle_2_z)):
            z = circle_2_z[j]
            y = circle_2_y[j]

            tau = (q_circle2[j]+q_circle0)/t_sk
            shear_stresses_crosssection = np.append(shear_stresses_crosssection,
                tau)


        # ------- NORMAL STRESSES -------

        for j in range(len(crosssection_z)):

            z = crosssection_z[j]
            y = crosssection_y[j]

            sigma_xx = M_z[i]*y/Izz + M_y[i]*(z-z_centroid)/Iyy

            normal_stresses_crosssection = np.append
                (normal_stresses_crosssection,sigma_xx)


        # ------- VON MISES STRESSES -------

        for j in range(len(crosssection_z)):
            vm = sqrt((normal_stresses_crosssection[j] ** 2) * (3*
                (shear_stresses_crosssection[j] ** 2)))
            vm_stresses_crosssection = np.append(vm_stresses_crosssection,vm)


        shear_stresses = np.append(shear_stresses,shear_stresses_crosssection)
        normal_stresses = np.append(normal_stresses,normal_stresses_crosssection
            )
        vm_stresses = np.append(vm_stresses,vm_stresses_crosssection)



    vm_GPa = 1e-9*vm_stresses


    # Print runtime
    print('\n')
    print('\n')
    print('Complete.'+'\n\n'+'Runtime: %f seconds\n' % (time.time()-start_time))

    '''

    # ----------------- PLOT CROSS-SECTION -----------------

    shear_stresses[0+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])

    shear_stresses[999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])

    shear_stresses[1000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])

    shear_stresses[2000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])

    shear_stresses[3000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])

    shear_stresses[4000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])

    shear_stresses[5000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])

    shear_stresses[1999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])

    shear_stresses[2999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])

    shear_stresses[3999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])

    shear_stresses[4999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])

    shear_stresses[5999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])


    # ----------------- PLOT CROSS-SECTION -----------------
    # Cross section
    shear_stresses[0+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])
    shear_stresses[999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])
    shear_stresses[1000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])
```

```
shear_stresses[2000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])
shear_stresses[3000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])
shear_stresses[4000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])
shear_stresses[5000+ 463*6000] = min(shear_stresses[463*6000:464*6000+1])
shear_stresses[1999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])
shear_stresses[2999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])
shear_stresses[3999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])
shear_stresses[4999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])
shear_stresses[5999+ 463*6000] = max(shear_stresses[463*6000:464*6000+1])
# ----------------- PLOT CROSS-SECTION -----------------
# Cross section
plt.figure(1)
plt.scatter(circle_1_z,circle_1_y, s = 0.7, c = (shear_stresses[463*6000:
    1000+ 463*6000]), cmap = 'jet')
plt.scatter(circle_2_z,circle_2_y, s = 0.7,c = (shear_stresses[5000+ 463*
    6000:6000+ 463*6000]), cmap = 'jet')
plt.scatter(spar_1_z,spar_1_y, s = 0.7,c = (shear_stresses[1000+ 463*6000:
    2000+ 463*6000]), cmap = 'jet')
plt.scatter(spar_2_z,spar_2_y, s = 0.7,c = (shear_stresses[4000+ 463*6000:
    5000+ 463*6000]), cmap = 'jet')
plt.scatter(top_z,top_y, s = 0.7,c = (shear_stresses[2000+ 463*6000:3000+
    463*6000]), cmap = 'jet')
plt.scatter(bottom_z,bottom_y, s = 0.7,c = (shear_stresses[3000+ 463*6000:
    4000+ 463*6000]), cmap = 'jet')
# Plot origin
plt.plot(0,0,'k.')
# Plot axes`
plt.axhline(0,linestyle='dashed',color='black',linewidth=1) # x = 0
plt.axvline(0,linestyle='dashed',color='black',linewidth=1) # y = 0
# Plot heatmap of von mises stress
# Set ranges for axes and show plot
plt.xlim(0.1,-0.45)
plt.ylim(-0.085,0.085)
plt.xlabel('z (m)')
plt.ylabel('y (m)')
plt.show()


normal_stresses[0+ 463*6000] = 0
normal_stresses[999+ 463*6000] = max(normal_stresses[463*6000:464*6000+1])
normal_stresses[1000+ 463*6000] = 0
normal_stresses[2000+ 463*6000] = 0
normal_stresses[3000+ 463*6000] = 0
normal_stresses[4000+ 463*6000] = 0
normal_stresses[5000+ 463*6000] = 0
normal_stresses[1999+ 463*6000] = max(normal_stresses[463*6000:464*6000+1])
normal_stresses[2999+ 463*6000] = max(normal_stresses[463*6000:464*6000+1])
normal_stresses[3999+ 463*6000] = max(normal_stresses[463*6000:464*6000+1])
normal_stresses[4999+ 463*6000] = max(normal_stresses[463*6000:464*6000+1])
normal_stresses[5999+ 463*6000] = max(normal_stresses[463*6000:464*6000+1])
plt.figure(2)
plt.scatter(circle_1_z,circle_1_y, s = 0.7, c = (normal_stresses[463*6000:
    1000+ 463*6000]), cmap = 'jet')
plt.scatter(circle_2_z,circle_2_y, s = 0.7,c = (normal_stresses[5000+ 463*
    6000:6000+ 463*6000]), cmap = 'jet')
plt.scatter(spar_1_z,spar_1_y, s = 0.7,c = (normal_stresses[1000+ 463*6000:
    2000+ 463*6000]), cmap = 'jet')
plt.scatter(spar_2_z,spar_2_y, s = 0.7,c = (normal_stresses[4000+ 463*6000:
    5000+ 463*6000]), cmap = 'jet')
plt.scatter(top_z,top_y, s = 0.7,c = (normal_stresses[2000+ 463*6000:3000+
    463*6000]), cmap = 'jet')
plt.scatter(bottom_z,bottom_y, s = 0.7,c = (normal_stresses[3000+ 463*6000:
    4000+ 463*6000]), cmap = 'jet')
# Plot origin
plt.plot(0,0,'k.')
# Plot axes
plt.axhline(0,linestyle='dashed',color='black',linewidth=1) # x = 0
plt.axvline(0,linestyle='dashed',color='black',linewidth=1) # y = 0
# Plot heatmap of von mises stress
# Set ranges for axes and show plot
plt.xlim(0.1,-0.45)
plt.ylim(-0.085,0.085)
plt.xlabel('z (m)')
plt.ylabel('y (m)')
plt.show()

'''
ab, abc, abcd, abcde, abdef, abdefg = section.get_shearflows(S_y[463],S_z
    [463])
print(max(ab))
print(max(abc))
print(max(abcd))
print(max(abcde))
print(max(abdef))
print(max(abdefg))
'''

plt.figure(3)
vm_stresses[0+ 463*6000] = min(vm_stresses[463*6000:464*6000+1])
vm_stresses[999+ 463*6000] = max(vm_stresses[463*6000:464*6000+1])
vm_stresses[1000+ 463*6000] = min(vm_stresses[463*6000:464*6000+1])
vm_stresses[2000+ 463*6000] = min(vm_stresses[463*6000:464*6000+1])
vm_stresses[3000+ 463*6000] = min(vm_stresses[463*6000:464*6000+1])
vm_stresses[4000+ 463*6000] = min(vm_stresses[463*6000:464*6000+1])
vm_stresses[5000+ 463*6000] = min(vm_stresses[463*6000:464*6000+1])
vm_stresses[1999+ 463*6000] = max(vm_stresses[463*6000:464*6000+1])
vm_stresses[2999+ 463*6000] = max(vm_stresses[463*6000:464*6000+1])
vm_stresses[3999+ 463*6000] = max(vm_stresses[463*6000:464*6000+1])
vm_stresses[4999+ 463*6000] = max(vm_stresses[463*6000:464*6000+1])
vm_stresses[5999+ 463*6000] = max(vm_stresses[463*6000:464*6000+1])
plt.figure(2)
plt.scatter(circle_1_z,circle_1_y, s = 0.7, c = (vm_stresses[463*6000:1000+
    463*6000]), cmap = 'jet')
plt.scatter(circle_2_z,circle_2_y, s = 0.7,c = (vm_stresses[5000+ 463*6000:
    6000+ 463*6000]), cmap = 'jet')
plt.scatter(spar_1_z,spar_1_y, s = 0.7,c = (vm_stresses[1000+ 463*6000:2000+
    463*6000]), cmap = 'jet')
plt.scatter(spar_2_z,spar_2_y, s = 0.7,c = (vm_stresses[4000+ 463*6000:5000+
    463*6000]), cmap = 'jet')
plt.scatter(top_z,top_y, s = 0.7,c = (vm_stresses[2000+ 463*6000:3000+ 463*
    6000]), cmap = 'jet')
plt.scatter(bottom_z,bottom_y, s = 0.7,c = (vm_stresses[3000+ 463*6000:4000+
    463*6000]), cmap = 'jet')
# Plot origin
plt.plot(0,0,'k.')
# Plot axes
plt.axhline(0,linestyle='dashed',color='black',linewidth=1) # x = 0
plt.axvline(0,linestyle='dashed',color='black',linewidth=1) # y = 0
# Plot heatmap of von mises stress
# Set ranges for axes and show plot
plt.xlim(0.1,-0.45)
plt.ylim(-0.085,0.085)
plt.xlabel('z (m)')
plt.ylabel('y (m)')
plt.show()
'''
```