Final Report

Simulation, Verification & Validation

by

Group A10

Janneke Blok	4300408
Klaas Burger	4471245
Laura Jou Ferrer	4456734
Lex Losch	4487656
Daniel Martini Jimenez	4448650

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Aerospace Engineering AE3212-II Simulation, Verification & Validation

at the Delft University of Technology,

Publication date : February 14, 2019

Contents

1
Introduction

This report focuses on analyzing an aileron in its critical load condition. Ailerons are one of the crucial control surfaces required to control an aircraft in flight. During a typical flight the ailerons are subjected to a variety of different loads. Due to the critical role that the aileron plays in the control of the aircraft, it is very important that it can sustain all these different loads. Therefore, it is crucial to test the aileron properly. This testing can be performed using a real-life test, but this can be highly expensive. For this reason, a common first step in the testing is the generation of a computer model to simulate the loads on the aileron.

In this report, two different models will be created, an analytical model and a numerical model. In order to ensure performance and accuracy of the models, they are verified by comparing their results. The models are also validated to make sure that they represent reality with sufficient detail and accuracy. This validation will be performed by comparing the results from the numerical model to the provided test data.

The purpose of this report is to supply a well performing and accurate model for the analysis of the Airbus A320's aileron. This will be achieved by the following steps. First, the problem at hand will be analyzed and explained in chapter 2. This chapter will also state the general assumptions that will hold throughout this report. Next, **??** will provide the analytical model for the analysis of the aileron. Afterwards, chapter 4 features the numerical approach to analyzing the aileron under critical loading. The analytical model will mainly be used for the verification of the numerical model. This verification process is explained in chapter 5. After verification has been completed and the model is deemed adequate, it is validated. The validation procedure will be elaborated upon in chapter 6. Finally, conclusions on the model and aileron's performance under critical loading will be drawn in chapter 7.

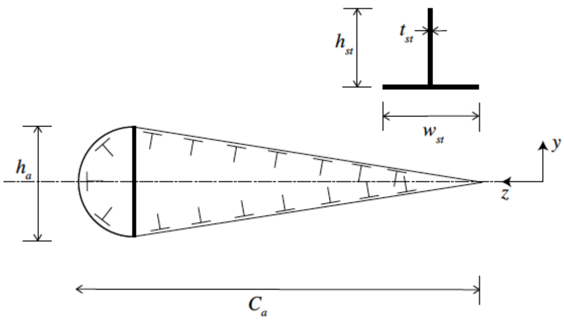
2
Problem Analysis

The first step in analyzing the aileron is stating the problem at hand. This will be done by first analyzing the cross-section and establishing a coordinate system in section 2.1. Section 2.2 will state the input variables that were used. The load case placed on the cross-section will be examined in section 2.3. In section 2.4, the general assumptions will be elaborated upon. The output variables are stated in section 2.5. Finally, the free body diagram and equilibrium analysis are presented in section 2.6. The problem analysis presented in this chapter will be very similar to the one presented the Simulation Plan [1].

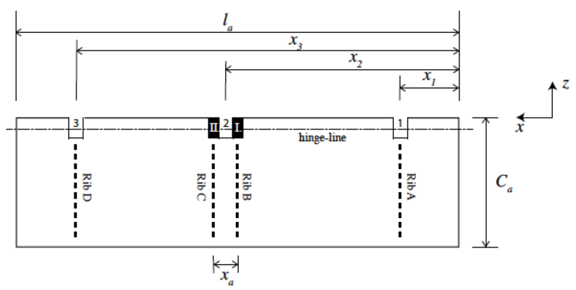
2.1. Coordinate frame and aileron cross-section

A right-handed 3D Cartesian Coordinate Frame (x,y,z) will be used. The origin of the coordinate system will be positioned at the hinge line of the cross section. The x-axis runs along the length of the aileron (l_a) and is positive in the direction from hinge 3 towards hinge 1. The y-axis is the vertical axis, positive upwards. The z-axis is defined perpendicular to the the x- and y-axes and is defined positive in the direction of the trailing edge. The axis system will not rotate with the deflection angle θ of the aileron.

The cross-section of the aileron is as shown in figure 2.1a. The cross-section contains 17 stiffeners. Seven of them are uniformly distributed along each of the inclined bars. A further three stiffeners are uniformly distributed along the semi-circular part of the cross section. The stiffeners run along the full length of the aileron. The aileron will be loaded in its maximum upward deflected position, at $\theta = 26deg$.



(a) Cross-section of the aileron



(b) Top view of the aileron

A top view of the aileron can be found in figure 2.1b. The aileron is hinged at three locations along the x-axis, with hinges 3, 2 and 1. Two actuators (I and II) are attached to the middle of the aileron, next to hinge 2. The aileron has four ribs (A, B, C and D), positioned at the same x-coordinate as hinge 1, actuator I, actuator II and hinge 3, respectively.

2.2. Input Variables

In table 2.1, one will find the input variables used during the analysis.

2.3. Load Case

The aileron that will be analyzed is under a critical loading condition. This critical loading condition is caused by different aspects during flight, as described below.

- The wing of the A320 aircraft will deform due to the aerodynamic load that it has to carry. This bending deformation will result in a bending load on the aileron. Hinge 2 is assumed to be fixed. Hinge 1 is fixed in x and z direction but moves in y-direction by a predefined amount of d1. Hinge 3 is fixed in x-direction and moved by a predefined distance d3 in y-direction.

Table 2.1: Input variables

Property	Symbol	Value	Unit
Chord length	C_a	0.547	m
Span	l_a	2.771	m
x-location of hinge 1	x_1	0.153	m
x-location of hinge 2	x_2	1.281	m
x-location of hinge 3	x_3	2.681	m
Distance between actuator 1 and 2	x_a	28.0	cm
Aileron height	h_a	22.5	cm
Skin thickness	t_{sk}	1.1	mm
Spar thickness	t_{sp}	2.9	mm
Thickness of stiffener	t_{st}	1.2	mm
Height of stiffener	h_{st}	1.5	cm
Width of stiffener	w_{st}	2.0	cm
Number of stiffeners	n_{st}	17	-
Vertical displacement hinge 1	d_1	4.34	cm
Vertical displacement hinge 3	d_3	6.46	cm
Maximum upward deflection	θ	26	deg
Load in actuator 2	P	91.7	kN
Net aerodynamic load	q	4.53	kN/m
Young's modulus	E	73.1 [2]	GPa
Shear Modulus	G	$28 \cdot 10^9$ [2]	GPa

- Actuator II introduces a force P to act in negative z-direction.
- Actuator I is jammed and therefore kept fixed in z-direction.
- A distributed load q acts on the aileron due to aerodynamic forces, this load acts in the negative y-direction, when considering the non-deflected position of the aileron.
- No other loads will be acting of the aileron.

2.4. General Assumptions

For both models, different specific assumptions will be made. Next to these specific assumptions, some general assumptions hold for the entire analysis. Some of these assumptions will have a larger impact on the final result than others. An overview of the general assumptions and their consequences can be found below. First, the assumptions with the biggest impact will be discussed, after which less significant assumptions are stated.

2.4.1. High-Impact Assumptions

The following assumptions will have a high impact on the analysis of the aileron:

- For the numerical model, the aileron is considered to be an idealized structure.
  By idealizing the structure, the stiffeners and spar will be replaced by booms that will carry all the normal stresses. The skin will not carry any direct stresses, but it will carry all the shear stresses. The moment of inertia calculation will change, since only the contribution of the booms is taken into account. All in all, the idealization of the structure will have a significant influence on the analysis: the maximum normal stress will be overestimated due to lower bending stiffness, the maximum shear stress will be underestimated. The overestimation of the normal stress will not pose a problem, it will merely add a "safety factor" of sorts. However, the underestimation of the shear stress could pose problems. In order to show that the analysis is still valid, it will be verified using the analytical model. The analytical will be calculated without structural idealizations.
- The aileron can be modelled as a beam. This means that standard beam theory can be applied. This also implies the following two assumptions that are integral of standard beam theory [3]:
  1. Cross-sections of the beam are assumed to be rigid, they will not deform in a significant

manner due to transverse or axial loads.
2. Cross-sections of the beam are assumed to remain planar and normal to the deformed axis of the beam.

- The hinges only restrict translations, but allow rotations.
- The net aerodynamic load can be modelled as a uniformly distributed load q, that will remain constant irrespective of the the deflection of the aileron.
- The problem will be analyzed as a static problem while the movement of an aileron in flight is a dynamic problem.
- The shear center is assumed to be located at the hinge line.

2.4.2. Low-Impact Assumptions

The following assumptions will have little impact on the analysis of the aileron.

- The reaction loads at the hinges and actuators are considered as point loads.
- Attachments of the stiffeners to the skin of the aileron will not be considered in this analysis.
- The material is assumed to be homogeneous.

2.5. Output Variables

The analysis of the aileron will generate several output variables, as stated in table 2.2.

Table 2.2: Output Variables

Property	**Symbol**
Maximum deflection leading edge, x-direction	$\delta_{LE,x}$
Maximum deflection leading edge, y-direction	$\delta_{LE,y}$
Maximum deflection leading edge, z-direction	$\delta_{LE,z}$
Maximum deflection trailing edge, x-direction	$\delta_{TE,x}$
Maximum deflection trailing edge, y-direction	$\delta_{TE,y}$
Maximum deflection trailing edge, z-direction	$\delta_{TE,z}$
Maximum shear stress in rib A	$\tau_{max,A}$
Maximum shear stress in rib B	$\tau_{max,B}$
Maximum shear stress in rib C	$\tau_{max,C}$
Maximum shear stress in rib D	$\tau_{max,D}$
Maximum normal stress in rib A	$\sigma_{max,A}$
Maximum normal stress in rib B	$\sigma_{max,B}$
Maximum normal stress in rib C	$\sigma_{max,C}$
Maximum normal stress in rib D	$\sigma_{max,D}$

2.6. Equilibrium analysis

The equilibrium analysis will be performed using the axis system marked x, y and z in the free body diagram below. This is the axis system as defined in section 2.1, but fixed at the initial position of the aileron. The other axis system, marked $\tilde{x}$, $\tilde{y}$ and $\tilde{z}$ in free body diagram, is as mentioned in section 2.1

As can be seen from the free body diagram (Figure 2.2), the hinges provide a total of 8 reaction forces. Hinge 1 delivers reaction forces in x- and y-direction but not in z-direction, because it is free to move in this direction. Hinges 2 and 3 provide reaction forces in x-, y- and z-direction. Additionally, force F_h is introduced by the jammed actuator II and forms the ninth unknown.
In order to analytically solve this statically indeterminate problem, the number of unknowns (9) should match the number equations. The first three equations will be provided by the normal static equilibrium in x-, y- and z-direction, these equations are stated below in Equation 2.1 - Equation 2.3. Furthermore,

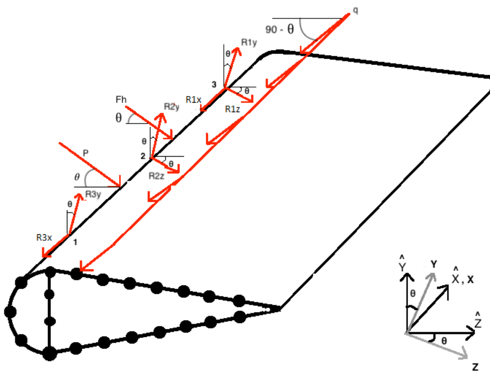


Figure 2.2: Free body diagram showing the forces acting on the wing.

it is possible to set-up a further three moment equations around hinge 2, as stated in Equation 2.4 - Equation 2.7. Since the problem is static, all of these equilibrium equations should equate to 0.

Additionally, in this problem it is possible to assume that all forces in the x-direction are negligible for the following two reasons. First of all, the forces in the x-direction have a insignificant contribution to the total bending moments, because their arms are relatively small compared to the arm of the aerodynamic load q and the vertical reaction forces. In fact, the highest value the arm of this forces will reach is d_3, which is only a 2.33% of the total span. This difference is significant enough to justify this assumption apart from the fact that even if this forces are not zero they will have a smaller magnitude with respect to the vertical forces because there is no external load applied in this direction. Secondly, the forces in x-direction will have an even smaller contribution to the moment around the y-axis because the deflection of the aileron in z-direction is very small. Since a broken connection at hinge 3 is simulated, hinge 3 is not fixed in z-direction. However, the displacement in this direction is not likely to cause a moment arm big enough to contribute to the total deflection.

$$\sum F_x = R_{x1} + R_{x2} + R_{x3} = 0 \tag{2.1}$$

$$\sum F_y = -q \cdot l_a + R_{y1} + R_{y2} + R_{y3} = 0 \tag{2.2}$$

$$\sum F_z = P + F_h + R_{z1} + R_{z2} = 0 \tag{2.3}$$

$$\sum M_z = -R_{3y} \cdot (x3 - x2) + R_{1y} \cdot (x2 - x1) + q \cdot la \cdot (la/2 - x2) = 0 \tag{2.4}$$

$$\sum M_x = F_h \cdot \frac{h_a}{2} \cdot (sin(\theta) - cos(\theta)) + P \cdot \frac{h_a}{2} \cdot (sin(\theta) - cos(\theta)) \tag{2.5}$$

$$+R_{1z} \cdot d1 + q \cdot l_a \cdot cos(\theta) \cdot (z_4 - z_h) = 0 \tag{2.6}$$

$$\sum M_y = F_h \cdot \frac{x_a}{2} + P \cdot \frac{x_a}{2} \cdot +R_{1z} \cdot (x2 - x1) = 0 \tag{2.7}$$

The choice of this reference system makes it possible to solve the forces in the z direction using three of the equilibrium equations(2.3,2.4 and 2.6) that depend only on these forces. After solving this linear system of equations the forces in z-direction give the following results: $R_{1z} = 26.14kN$, $R_{2z} = 1.06kN$ and $F_h = -118.94kN$.

3
Analytical Solution

This chapter explains the different steps which were followed to achieve the analytical solution, plus the results.

3.1. Moments of Inertia

The cross section's moments of inertia along both the y-axis and the z-axis will be determined using the equations stated in the Simulation Plan [1]. The equations will be restated below, Equation 3.2 - Equation 3.7. The cross section is symmetrical along the z-axis, therefore I_{zy} will equal zero. Both I_{yy} and I_{zz} will be calculated by dividing the cross section into three different components, the arc, the spar and the inclined bars. The contribution of the stiffeners will be included at the end of the calculation.

$$I_{zz_{arc}} = \frac{\pi \cdot r^3 \cdot t_{sk}}{2} \tag{3.1}$$

$$I_{yy_{arc}} = \frac{\pi \cdot r^3 \cdot t_{sk}}{2} - A \cdot (\tilde{z}_{arc} - z_{arc})^2 \tag{3.2}$$

$$I_{zz_{spar}} = \frac{1}{12} \cdot t_{sp} \cdot h_a^3 \tag{3.3}$$

$$I_{yy_{spar}} = \frac{1}{12} \cdot t_{sp}^3 \cdot h_a \tag{3.4}$$

$$l_{skin_{inclined}} = \sqrt{2 \cdot r^2 + C_a^2 - 2C_a \cdot r} \tag{3.5}$$

$$I_{zz_{bars}} = 2 \cdot \frac{l_{skin_{inclined}} \cdot t_{skin} \cdot (C_a - r)^2}{12} + 2 \cdot l_{skin_{inclined}} \cdot t_{skin} \cdot \left(\frac{r}{2}\right)^2 \tag{3.6}$$

The total moment of inertia around the z-axis can be calculated using equation Equation 3.7, this does not include the contribution of the stiffeners yet.

$$I_{zz} = I_{zz_{spar}} + I_{zz_{skin}} = I_{zz_{spar}} + I_{zz_{arc}} + I_{zz_{bars}} \tag{3.7}$$

The total moment of inertia around the y-axis can be calculated in a similar way.
Next, the contribution of the stiffener moments of inertia will be analyzed. The stiffeners are uniformly distributed along the conical part of the cross section. Furthermore, three stiffeners have been equally spaced along the semi-circular part of the cross section. The stiffeners moment of inertia contribution around the z-axis can be calculated using Equation 3.8 - 3.12.

$$I_{st_{zz}} = \frac{w_{st}^3 \cdot t_{st} \cdot sin^2(\gamma)}{12} + \frac{h_{st}^3 \cdot t_{st} \cdot sin^2(\pi - \gamma)}{12} + A_{st} \cdot (\tilde{y} - y_{st})^2 \tag{3.8}$$

$$I_{st_{yy}} = \frac{w_{st}^3 \cdot t_{st} \cdot cos^2(\gamma)}{12} + \frac{h_{st}^3 \cdot t_{st} \cdot cos^2(\pi - \gamma)}{12} + A_{st} \cdot (\tilde{z} - z_{st})^2 \tag{3.9}$$

Where γ is the inclination angle of the bars given in radians. Furthermore, $\tilde{y}$ and $\tilde{z}$ represent the y and z coordinate of the centroid of the stiffener respectively.

In order to account for the Steiner's Theorem contribution in the moment of inertia of the stiffeners, their exact location needs to be calculated. The stiffeners in the triangular part of the cross section are uniformly distributed along the length of the bar. In order to determine the location of the stiffeners, a linear equation has been set-up. This equation is given below by Equation 3.10. The location of the stiffeners in the circular part of the cross section will determined using Equation 3.11.

$$d_{st}(z) = h_a - \frac{z}{C_a - h_a} \cdot h_a + \tilde{y} \tag{3.10}$$

$$d_{st}(\theta) = r \cdot sin(\theta) + \tilde{y} \tag{3.11}$$

The total moment of inertia can now be determined using Equation 3.12.

$$I_{tot} = I_{skin} + I_{spar} + n_{st} \cdot I_{st} + A_{st} \cdot d_{st}^2 \tag{3.12}$$

The calculated values for the moments of inertia around the z-axis and y-axis are presented in Table 3.1. The values in the table are calculated using the input values as given in Table 2.1.

Table 3.1: Moment of Inertia

Property	**Value**	**Unit**
I_{zz}	$1.238 \cdot 10^7$	mm^4
I_{yy}	$6.590 \cdot 10^7$	mm^4

3.2. Reaction Forces

In order to correctly analyze and simulate the stresses in the cross section of the aileron, the reaction forces at the hinges need to be determined first. All reaction forces and other loads that are inflicted on the aileron in this analysis are depicted in the free body diagram (Figure 2.2), which is based on the load condition as described in section 2.3.

The reaction forces in z-direction have already been calculated. This should simplify the calculations as only the three vertical reaction forces at the hinges remain unknown.

Next to the equilibrium equations, there are three boundary conditions at the hinges. The boundary conditions come in the form of known deflection in y-direction. These boundary conditions can be used to solve the moment curvature equation. Two of these conditions will be used to solve for the integration constants that occur when integrating the moment curvature equation. The third boundary condition can be used to solve for the reaction forces together with the equilibrium equations.

In order to be consistent the following equations have been described in the chosen coordinate system, as described in section 2.1. This choice will permit to set equations with the boundary conditions of the bending deflections, as these are given in y-direction with respect to this coordinate system. In case the tilted coordinate system $\tilde{x}\tilde{y}\tilde{z}$ would have been used, it would not have been possible to set a fifth equation at hinge three because the degree of freedom in the z axis does not allow to calculate the total vertical deflection.

As the aileron is deflected by θ degrees, the loads depicted in figure 2.2 will create bending moments around the y- and z-axis. The deflection in these axes will thus be a linear combination of the two moments.

Moreover, the complication of this choice is that the equations of the deflection to find are inclined relative to the centroidal axes. The problem becomes unsymmetrical bending so the following equation holds[4] where u is the respective deflection about the z axis and v is the one abou the y axis.

$$\begin{bmatrix} u'' \\ v'' \end{bmatrix} = \frac{1}{E \cdot (I_{zz} \cdot I_{yy} - Izy^2)} \cdot \begin{bmatrix} I_{zz} & -I_{zy} \\ -I_{zy} & I_{yy} \end{bmatrix} \cdot \begin{bmatrix} M_z \\ M_y \end{bmatrix}$$

As was mentioned before, the moments and the displacements are calculated about the fixed coordinate frame at the wing, thus the moments of inertia will also be different from the previously calculated one. In fact, there will also be an I_{zy} component as the cross section of the aileron is not symmetrical anymore about this axis. In order to calculate the moment of inertia of a tilted axis with respect to the centroidal one there is a standard transformation, which is displayed in equation 3.13

$$I_{xyz} = R(x,\theta)^T \cdot I_{\tilde{x}\tilde{y}\tilde{z}} \cdot R(x,\theta) \tag{3.13}$$

In the moment functions below, the Macaulay Step Function is denoted by [....] and 'H' denotes a heaviside function. Furthermore, the x-datum starts at the left of the aileron close to hinge 3 and proceeds towards hinge one. This direction was chosen in order to be consistent with the reference frame that points in this direction as well.

$$M_z(x) = (q \cdot \frac{x^2}{2} - R_{1y} \cdot [x - x1] - R_{2y} \cdot [x - x2] - R_{3y} \cdot [x - x3] \tag{3.14}$$

$$M_y(x) = R_{1z} \cdot [x - x1] + F_h \cdot [x - x2 - xa/2] + R_{2z} \cdot [x - x2] + P \cdot [x - x2 + xa/2] \tag{3.15}$$

$$M_x(x) = R_{2z} \cdot \frac{ha}{2} \cdot H(x - x_2) + P \cdot \frac{ha}{2} \cdot (cos(\theta) - sin(\theta)) \cdot H(x - x_2 + \frac{xa}{2}) \tag{3.16}$$

$$+F_h \cdot (cos(\theta) - sin(\theta)) \cdot \frac{ha}{2} \cdot H(x - x_2 - \frac{xa}{2}) + R_{1z} \cdot d1 \cdot H(x - x_1) + q \cdot x \cdot cos(\theta) \cdot (z4 - zh) \tag{3.17}$$

With the analytical expressions for the moment distribution it is possible to solve the the curvature equation for unsymmetrical bending. The system is composed by the two equilibrium equations and from the analytical expression in the y deflection derived as in equation 3.18 it is possible to obtain other three equations by evalutating the deflection at the three hinges and then set it equal to d1, 0 and d3.

$$y = \frac{1}{E \cdot (I_{zz} \cdot I_{yy} - Izy^2)} \cdot \int_0^x \int_0^x \left(I_{yy} \cdot M_y - I_{zy} \cdot M_z\right) dx dx \tag{3.18}$$

Solving the linear system of five equations leads to the following result for the vertical forces at the hinges. The results are consistent as the forces at the two outer hinges are pointing upwards as the deflection in those respective locations, while the force at hinge 2 is pointing downwards as this point is fixed and is not allowed to move at all.

Table 3.2: Results reaction forces

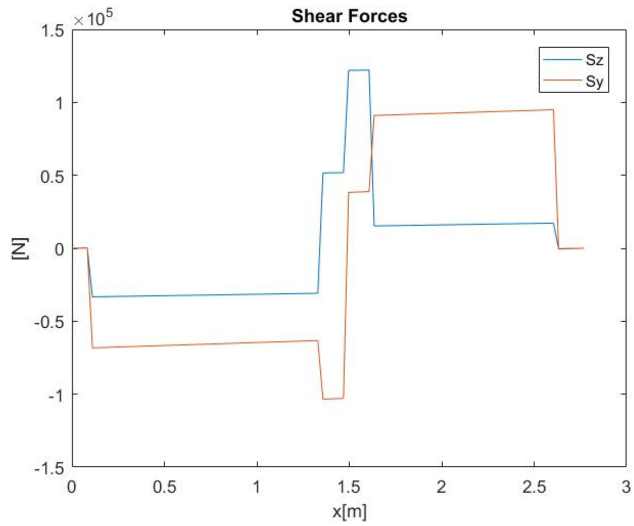
$$R_{1y} = 93.66kN$$
$$R_{2y} = -157.52kN$$
$$R_{3y} = 76.40kN$$

3.3. Load Diagrams

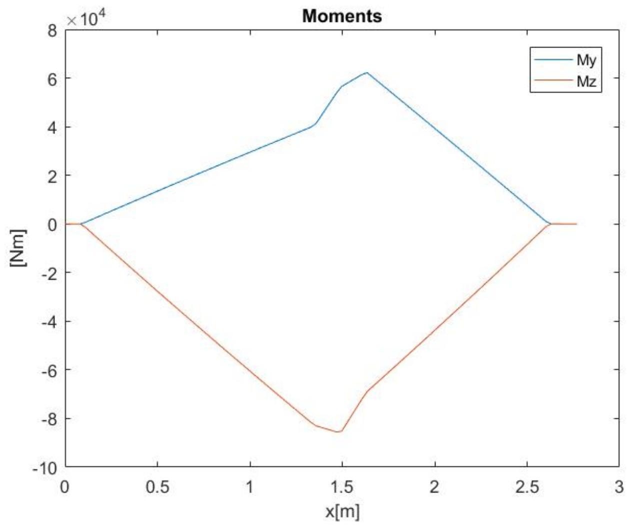
In the following section the loading case is going to be analyzed, the plots presented will give a clear idea of the locations along the aileron that are experiencing higher moments and shear forces. Now that the reaction forces have been calculated it is possible to obtain mathematical expressions for the moment by substituting the value of the force in the moment equation found in section 3.2. It is not possible to use this moments for the stress and the deflection calculation as this expressions are not given about the centroidal axes. After obtaining the expression of the moment in function of the span, the moments have been rotated in order to have expression for the loading around the symmetrical axes of the aileron and simplify all the stress calculation.

The following diagrams start at the left of the aileron close to hinge three and proceeds until the next extreme where hinge one is located. The shear diagram gives a clear idea of the location of the forces and its distribution. The aerodynamic distributed load has a smaller contribution with respect to the reaction forces in fact the diagram is mostly composed by constants segments that change their value at the location of each force. The effect of the distributed load makes the lines to have a small slope. Furthermore it is possible to notice the same in the moment diagram about the x and y axis which are linear for the same reason as in the shear diagrams. The moment around the z axis is negative as it produces compression in the top of the aileron cross section, moreover the magnitude of the moment in the z axis increases as the aileron decrease the rotation. Thus it is important to remark that the stresses due to this forces are more significant, not only due to the magnitude of the moment but also because the inertial properties of the aileron in the z axis are smaller compared to the inertia in the y axis.

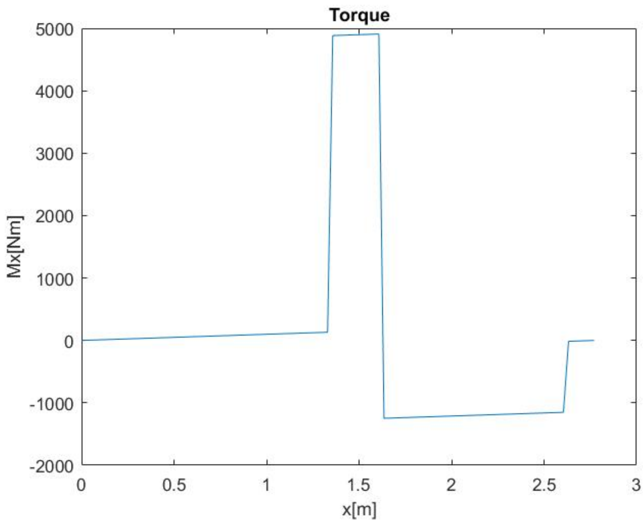
Moreover it is possible to notice that the diagrams start and end at zero thus the equilibrium requirement is met for all the axis. In case there would not be equilibrium the situation would be unrealistic



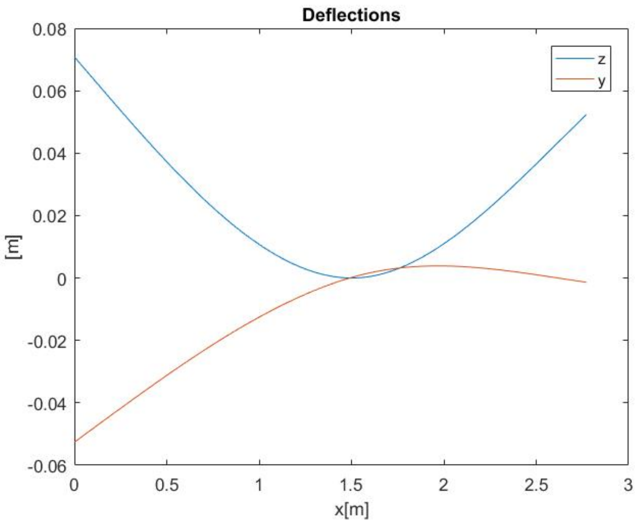
(a) Shear force along the span of the aileron



(b) Moments along the span of the aileron



(a) Torque along the span of the aileron



(b) Deflection of the hinge along the span of the aileron

as the aileron has free ends at the two extremes and it would not be structurally stable to withstand the loads so it would be moving and deflecting.

The Torque diagram has the contribution of three main loads which are the two actuators and the distributed load. Furthermore the reaction forces at the hinges also produce a torque which is not introduced locally but it increases in the span wise direction in function of the deflection. In order to simplify the distribution the contribution of the reaction forces was added at hinge 1 in order to prove that there is equilibrium in this axis as well.

3.4. Deflection

The deflection of the aileron in y-direction will be found using the internal bending moment, which can be calculated for each place along the span of the aileron by Equation 3.14. The internal bending moment at a location will be inputted into Equation 3.19. Equation 3.19 will then be integrated twice in order to find the deflection equation. The integration will add two integration constants to the equation. These constants can be solved for by applying the boundary conditions that are supplied by d_1, d_2 and d_3. The equation that is found in this way provides the deflection along the x-axis of the hinge line. In order to find the deflection at the leading and trailing edge, the twist rate of the aileron due to the torsion should be included.

The maximum deflection of the hinge line due to bending in the vertical direction is 7.1 centimeters while the displacement in the chord direction is 5.2 centimeters. The plot shows two different dimension for the two deflection but it is consistent with the signs. For instance, the deflection in z is negative because z points towards the trailing edge the aileron is deflecting forward.

$$-M_z(x) = E \cdot I_{zz} \cdot \frac{d^2y}{dx^2} \tag{3.19}$$

3.5. Normal Stresses

The normal stresses which are caused by the bending moments, can be determined with the moment diagrams. Now with the internal bending moment at each location, the normal stress at each location can be found with equation 3.20.

$$\sigma_x = \frac{I_{zz} \cdot M_y - I_{zy} \cdot M_z}{I_{zz} \cdot I_{yy} - I_{zy}^2} \cdot z + \frac{I_{yy} \cdot M_z - I_{zy} \cdot M_y}{I_{zz} \cdot I_{yy} - I_{zy}^2} \cdot y \tag{3.20}$$

Since I_{xy} is equal to zero this equation simplifies into the equation 3.21.

$$\sigma_x = \frac{M_y}{I_{yy}} \cdot z + \frac{M_z}{I_{zz}} \cdot y \tag{3.21}$$

With the moment of inertia computed in the previous section, given in table 3.1 the normal stresses can be found. As is stated in the output values in Table 2.2, the normal stresses should be determined at the rib locations. The maximum stresses at ribs A, B, C and D are presented in Table 3.3. As can be seen from this table, the maximum normal stress at the ribs is equal to 646.53 MPa. This normal stress is located at $y = 101.68mm$ and $z = -56.88mm$ in rib B.

Table 3.3: Maximum Normal Stress at the Ribs

Rib	Max normal stress	Unit	y-coordinate [mm]	z-coordinate [mm]
A	0.145	MPa	101.68	-56.88
B	646.53	MPa	101.68	-56.88
C	538.37	MPa	101.68	-56.88
D	220.87	MPa	101.68	-56.88

3.6. Shear Flow due to Pure Shear and Torsion

The shear flows in the cross section can be calculated with Equation 3.22. In this equation, q_b represents the base shear flow and $q_{s,0}$ represents the torsional component of the shear flow.

$$q_s = q_b + q_{s,0} - \frac{S_y \cdot I_{yy} - S_z \cdot I_{zy}}{I_{xx} \cdot I_{yy} - I_{zy}} \int_0^s t \cdot yds - \frac{S_z \cdot I_{zz} - S_y \cdot I_{zy}}{I_{zz} \cdot I_{yy} - I_{zy}} \int_0^s t \cdot zds + q_{s,0} \tag{3.22}$$

However, since the cross-section of the aileron is symmetrical, $I_{zy} = 0$ and Equation 3.22 can be simplified to the following equation.

$$q_s = -\frac{S_y}{I_{zz}} \int_0^s t \cdot yds - \frac{S_z}{I_{yy}} \int_0^s t \cdot zds + q_{s,0} \tag{3.23}$$

The main load-bearing structure with regard to shear stresses is the skin. The part of the stiffener cross-section that is normal to the skin will have little to no influence on the shear flows in the skin. Therefore, it has been decided to take into account the extra thickness in the cross-section only. The stiffener thickness is ”smeared” out over the separate parts. This will be done for the arc and the bars separately. Since there are no stiffeners on the spar, the spar thickness will remain the same. This will result in the following thicknesses: $t_{arc} = 1.3037mm$ and $t_{bar} = 1.4743mm$ and $t_{spar} = 2.9mm$ stays as it was before.

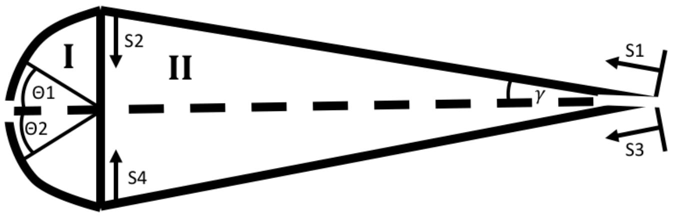


Figure 3.3: Reference coordinates shear flow calculations

In order to calculate the base shear flow in the cross-section, two cuts were made in the section. One cut is located at the leading edge and one cut is located at the trailing edge, both are on the symmetry axis. From these cuts, shear flow distributions were made for each part of the section. For the base shear flow one uses the shear forces as if they work through the shear center. Due to symmetry, the influence of the shear force in z-direction will cause a shear flow distribution that is anti-symmetric in the z-axis. Similarly, the influence of the shear force in y-direction will cause a shear flow distribution that is symmetric in the z-axis. Using these features and the coordinates as shown in Figure 3.3, the equations to be found in Table 3.4 were derived.

Table 3.4: Shear Flow Equations

Section	**Shear Flow**
Arc, top half	$q_{b,arc,top} = t_{arc}r^2\left(\frac{S_y}{I_{zz}}(cos(\theta_1)-1)+\frac{S_z}{I_{yy}}(sin\theta_1)\right)$
Bar, top	$q_{b,bar,top} = -t_{bar}s_1\left(\frac{S_y}{2I_{zz}}t_{bar}sin\gamma s_1+\frac{S_z}{I_{yy}}((C_a-r)-\frac{1}{2}s_1cos\gamma)\right)$
Spar, top half	$q_{b,spar,top} = -\frac{S_y}{I_{zz}}t_{spar}(rs_2-\frac{1}{2}s_2^2)+t_{arc}r^2\left(-\frac{S_y}{I_{zz}}+\frac{S_z}{I_{yy}}\right)-l_bt_{bar}\left(\frac{S_y}{2I_{zz}}r+\frac{S_z}{I_{yy}}(C_a-r)\right)$
Arc, bottom half	$q_{b,arc,bottom} = t_{arc}r^2\left(-\frac{S_y}{I_{zz}}(cos(\theta_2)-1)+\frac{S_z}{I_{yy}}(sin\theta_2)\right)$
Bar, bottom	$q_{b,bar,bottom} = -t_{bar}s_1\left(-\frac{S_y}{2I_{zz}}t_{bar}sin\gamma s_1+\frac{S_z}{I_{yy}}((C_a-r)-\frac{1}{2}s_3cos\gamma)\right)$
Spar, bottom half	$q_{b,spar,bottom} = \frac{S_y}{I_{zz}}t_{spar}(rs_4-\frac{1}{2}s_4^2)+t_{arc}r^2\left(\frac{S_y}{I_{zz}}+\frac{S_z}{I_{yy}}\right)-l_bt_{bar}\left(\frac{S_y}{2I_{zz}}r+\frac{S_z}{I_{yy}}(r-C_a)\right)$

In order to calculate the maximum shear flow in the ribs, the shear flows on either side of each rib will be calculated. The differential that these flows cause will be equal to the shear flow introduced in the ribs. The x-locations of the ribs A, B, C and D are $x_A = 2.618m$, $x_B = 1.630m$, $x_C = 1.350m$ and $x_D = 0.090m$, respectively.

After the open section shear flow have been determined, the cut can be closed and the shear flow q_{s0} caused by the torque can be found. The rate of twist of cell 1 can be found with Equation 3.24 and the rate of twist of cell 2 can be found with Equation 3.25.

$$\left(\frac{d\theta}{dz}\right)_I = \frac{1}{2\cdot A_1\cdot G}\cdot\left[\frac{q_{s0,1}\cdot\pi\cdot r}{t_{arc}}+\frac{2q_{s0,1}\cdot r}{t_{spar}}-\frac{2\cdot q_{s0,2}\cdot r}{t_{spar}}\right] \tag{3.24}$$

$$\left(\frac{d\theta}{dz}\right)_{II} = \frac{1}{2\cdot A_2\cdot G}\left[\frac{2\cdot q_{s0,2}\cdot l_b}{t_{bar}}+\frac{2\cdot q_{s0,2}\cdot r}{t_{spar}}-\frac{2\cdot q_{s0,1}\cdot r}{t_{spar}}\right] \tag{3.25}$$

$$\left(\frac{d\theta}{dz}\right)_I = \left(\frac{d\theta}{dz}\right)_{II} \tag{3.26}$$

In order to solve this system another relation for the constant shear flows $q_{s0,1}$ and $q_{s0,2}$ is given by Equation 3.27.

$$T = 2\cdot A_1\cdot q_{s0,1}+2\cdot A_2\cdot q_{s0,2} \tag{3.27}$$

The shear stress follows from the shear flows, by dividing the shear flow by the thickness, Equation 3.28.

$$\tau = \frac{q}{t} \tag{3.28}$$

By adding the different shear flows in an excel file the maximum shear stresses in the ribs have been found and are tabulated in Table 3.5. In this table the absolute value of the maximum shear stress at each of the 4 ribs is given. From this table it is obvious that the maximum shear stress occurs in Rib A and is equal to 395.3418 MPa.

Table 3.5: Maximum Shear Stress at the Ribs

Rib	Max shear stress	Unit
A	395.3418	MPa
B	382.8615	MPa
C	359.6083	MPa
D	319.8316	MPa

3.7. Results

In Table 3.6 the outputs of the analytical model are given. The values for the maximum deflections of the leading and trailing edge in x-direction can be neglected.

Table 3.6: Required Outputs from the Analytical Model

Property	Symbol	Value	Unit
Maximum deflection at hinge line, y-direction	δ_y	7.1	cm
Maximum deflection at hinge line, z-direction	δ_z	5.2	cm
Maximum shear stress in rib A	$\tau_{max,A}$	395.34	MPa
Maximum shear stress in rib B	$\tau_{max,B}$	382.86	MPa
Maximum shear stress in rib C	$\tau_{max,C}$	359.60	MPa
Maximum shear stress in rib D	$\tau_{max,D}$	319.83	MPa
Maximum normal stress in rib A	$\sigma_{max,A}$	0.145	MPa
Maximum normal stress in rib B	$\sigma_{max,B}$	646.53	MPa
Maximum normal stress in rib C	$\sigma_{max,C}$	538.37	MPa
Maximum normal stress in rib D	$\sigma_{max,D}$	220.87	MPa

4
Numerical Solution

4.1. Code

The numerical solution was reached using a Python script developed by the authors, which is presented in Appendix B. This script contains four different classes. The first one is Boom, which calculates the distance to the centroid, area and bending stress of each boom. An instance of the class Edge represents a skin section between two booms. This class does not contain any methods: it is only used to store values such as the thickness and length of a piece of skin, or the shear flows in a convenient way. The class Geometry is used to find geometrical properties of the cross-section, mainly the moments of inertia and the centroid. Finally, the shear flows and stresses are calculated using methods from the class DiscreteSection, which represents a slice of the aileron.

4.2. Boom Area Calculation

For the numerical solution, the structure is idealized using booms. It is assumed that the normal stress is carried only by the booms and the shear stress is carried only by the skin between the booms.

In this section the area of the booms is computed using the approach as explained in the Simulation Plan [1].The boom areas will be proportional to the stiffener area and the skin area that can be included in the boom. The boom areas will be calculated using Equation 4.1. In this equation t_{sk} represent the skin thickness, l_{skin} represents the length of the skin between the two booms and $\frac{\sigma_j}{\sigma_i}$ represents the ratio in stresses between the boom being calculated and the booms surrounding it. This ratio is equal to the ratio of the distances from booms to the neutral axis, represented by d_{st_j} and d_{st_i} respectively. In this case the neutral axis is assumed to be on the line of symmetry, afterwards with the help of the Python code the boom area will be found.

As can be seen from Figure 4.1, the A320 aileron will have 43 booms with 49 pieces of skin between them. The cross section has 17 stiffeners that will be represented by booms. Additionally, the spar will be modelled with four booms. Moreover, in the middle of each skin piece that connects two stiffeners, another boom will be added. The booms that are not located at the location of a stiffener will only take the skin area and stress ratio into account. A further two booms will be located at the intersections of the skin and the spar. Additionally, in the calculation of the boom areas, the neutral line is assumed to be located at the line of symmetry. This assumption is valid since the moment around the z-axis is significantly bigger than the moment around the y-axis.

$$B_i = \frac{t_{sk} \cdot l_{skin}}{6} \cdot \left(2 + \frac{\sigma_j}{\sigma_i}\right) + A_{stringers} = \frac{t_{sk} \cdot l_{skin}}{6} \cdot \left(2 + \frac{d_{st_j}}{d_{st_i}}\right) + A_{stringers} \tag{4.1}$$

The 32 booms lying on the inclined bars are spaced uniformly. The same holds for the seven booms on the arc and the four booms on the spar. By calculating the length of the bar, the spar and the arc and subsequently dividing this by 16, 5 and 8 respectively, the length of the skin part between the booms can be determined l_{skin}.
Boom 19 is located on the line of symmetry and therefore on the neutral axis. Because of this, the boom area calculation will only consider the area of the stiffener and neglect the part containing the stress ratio in Equation 4.1. This is caused by the fact that it is not possible to calculate the boom area using the stress ratio on the neutral axis, as this would imply that in the distance ratio would lead to a division by zero.

The final results from the boom area calculations are presented in **??**.

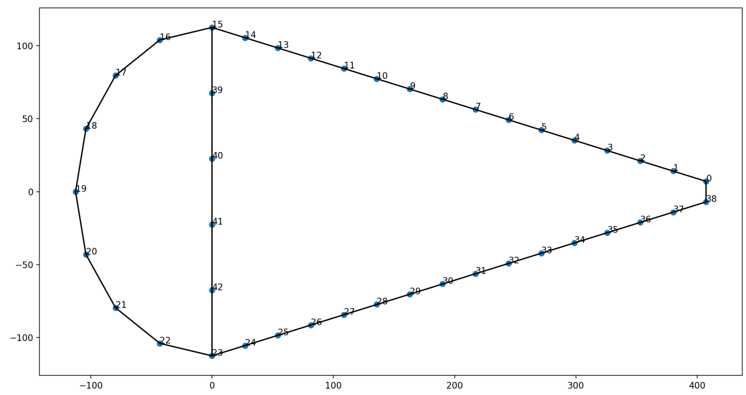


Figure 4.1: Idealized cross section.

4.3. Moment of Inertia

Now that the boom areas are known, the moments of inertia of the cross section can be calculated. For this calculation, structural idealization is used once more. In the numerical model, the moment of inertia is calculated using two different classes in the Python code, the Boom class and the Geometry class.

4.3.1. Boom Class

The Boom class is used to store the geometrical properties of the booms. It holds the boom areas, presented in **??**, the booms y- and z- coordinates with respect to the centroid and the calculation of the normal stresses. The calculation of the normal stresses will be discussed later in section 4.5.

4.3.2. Geometry Class

The Geometry class is mainly used to perform the actual calculations. Furthermore, the class also holds a list of areas from all the different parts that make up the cross section, namely the arc, the spar, the inclined bars and the stiffeners. The class starts by calculating the centroid location using the boom areas and there respective locations. The centroid is calculated to be located at $(y, z) = (0; 112.26)$ in mm from the hinge line. In this class, the moments of inertia, I_{zz} and I_{yy} are calculated using the moment of inertia formulas for idealized cross sections, as presented in Equation 4.2 and Equation 4.3. The class first calculates the moment of inertia contribution of a single boom and then iterates the calculation for all 43 booms. Finally, the moment of inertia contribution for all booms is summed together, providing a final I_{zz} of $12.173 \cdot 10^6 mm^4$ and a final I_{yy} of $60.758 \cdot 10^6 mm^4$.

$$I_{zz} = \sum A_{boom,i} \cdot z_{boom,i}^2 \tag{4.2}$$

$$I_{yy} = \sum A_{boom,i} \cdot y_{boom,i}^2 \tag{4.3}$$

4.4. Reaction forces

The reaction forces for the numerical model have not been calculated with a different model because the geometrical properties of the aileron were assumed not to change significantly due to the twist. Therefore, a numerical model would not have altered the forces significantly because the iteration of the bending curvature equation due to the change in inertia would have produced the same results as the analytic expression. The reaction forces have been calculated using the same approach as for the mathematical method. The only difference in this case is the moment of inertia of the aileron geometry which is structurally idealized and kept constant for the cross-section along the span. The new values have a small difference with the analytic method because the structural idealization has values for the moment of inertia that are similar to the non- idealized structure. The new reaction forces in y are displayed in the following table Furthermore it was decided not to use energy methods as in the

Table 4.1: Results reaction forces

$$R_{1y} = 91.89kN$$
$$R_{2y} = -154.32kN$$
$$R_{3y} = 74.98kN$$

simulation plan because the fact that the energy due to torsion and the axial forces would be neglected creates significant errors in the energy methods calculation thus the reaction forces calculated would not allow for the natural process of deformation where the complementary energy is minimized and the corresponding values would not have been as realistic as possible. Moreover using the bending curvature equation it assured that the deflection meets the boundary conditions imposed.

4.5. Normal stresses

The method used to calculate the normal stresses in the numerical model is comparable to the method explained in section 3.5. Equation 3.20 also holds for the numerical analysis. Furthermore, in the numerical approach I_{zy} can still be taken to be zero, because the cross section is symmetric. Therefore, the normal stress equation can, once again, be simplified to Equation 3.21.

The difference between the method used for the analytical model and the numerical model is that the numerical model only calculates the normal stresses in the booms. This is a result of the structural idealization. When applying structural idealization, one assumes that all normal stresses are carried by the booms and that the skin merely carries the shear stresses. Hence, the normal stresses in the skin are assumed to be zero.

The normal stresses are calculated in the Boom class of the numerical model code. In this class, the normal stresses is calculated using the moments of inertia calculated in section 4.3, the boom locations stored in the Boom class and the internal bending moment distribution as provided from section 3.3. The maximum normal stresses at each of the ribs are presented in Table 4.2.

4.6. Shear Stresses

In order to determine the shear stresses in the aileron, the problem will be split in two different parts. First, the shear stress purely caused by the shear forces on the aileron will be analyzed. Then, the shear stresses due the torque acting the aileron will be analyzed. In the end, both contributions will be summed for every node in the aileron in order to find the total shear stresses at these nodes. Furthermore, as mentioned in section 2.4, two highly important assumptions hold in this calculation. First, the shear center is assumed to be located at the hinge line. Secondly, due to structural idealization being applied, the skin is assumed to carry all the shear stresses while the booms carry none.

4.6.1. Shear Stress due to Shear Forces

This subsection uses the shear force distribution as presented in section 4.4. The aileron's cross section is a multi-cell structure, therefore the shear flows for cell 1 and cell 2 will be determined separately. The cells will be defined in the same way as is presented in Figure 3.3 for the analytical model. In order to determine the shear flow, a virtual cut has been made in each cell. The cut in cell 1 is located in the panel between boom 19 and 20 and the cut in cell 2 is located between booms 38 and 0. Then the shear flow is calculated using the Equation 4.4.

$$q_B = -\left(\frac{S_z I_{zz} - S_y I_{zy}}{I_{zz} I_{yy} - I_{zy}^2}\right) B_r z_r - \left(\frac{S_y I_{yy} - S_z I_{zy}}{I_{zz} I_{yy} - I_{zy}^2}\right) B_r y_r \tag{4.4}$$

Afterwards, the closed section shear flow is determined. Since, it is assumed that the shear forces are acting through the shear center, the twist rate is equal for both cells and has a value of zero.

$$\left(\frac{d\theta}{dx}\right)_I = \left(\frac{d\theta}{dx}\right)_{II} = 0 \tag{4.5}$$

The rate of twist for each cell is given by Equation 4.6. Here δ represents the length of a wall divided by its thickness. δ_I refers to the sum of all walls on cell I divided by their respective thickness, and $\delta_{I,II}$ refers to the wall that is shared by cell I and cell II. G is the shear modulus of the material and A is the area of the cell indicated by its subscript.

$$\left(\frac{d\theta}{dx}\right)_I = \frac{1}{2 \cdot A_I \cdot G}\left(q_{S,0,I}\delta_I - q_{S,0,II}\delta_{I,II} + \oint q_B \frac{ds}{t}\right) \tag{4.6}$$

The twist rate for cell II can be found similarly to Equation 4.6. This leads to a system of two equations with two unknowns which allows finding the closed section shear flows for each cell, $q_{S,0,I}$ and $q_{S,0,II}$.

Finally, the total shear flow on each piece of skin in the cross-section can be found by adding the open section shear flow q_B at that wall and the corresponding closed section shear flow for that wall.

4.6.2. Shear Stress due to Torsion

This subsection is focused on finding the twist rate introduced by the shear forces that are not applied through the shear center. Firstly, the torque M_x that is applied at a particular location in the span must be found, which is explained in section 3.3. This torque is divided between the two sections as described by Equation 4.7.

$$T = T_I + T_{II} \tag{4.7}$$

The torque on each cell can be found using Equation 4.8. There are two equations like this one, one for each cell, which introduces two more unknowns: $q_{T,I}$ and $q_{T,II}$.

$$T_n = 2 \cdot A_n \cdot q_{T,n} \tag{4.8}$$

Since the twist rate must be equal on both cells, Equation 4.9 can be used in combination with Equation 4.6 to complete a system of four equations with four unknowns.

$$\left(\frac{d\theta}{dx}\right)_I = \left(\frac{d\theta}{dx}\right)_{II} \tag{4.9}$$

With this system it is possible to find the values of T_I, T_{II}, $q_{T,I}$ and $q_{T,II}$. The last step is substituting these values in Equation 4.6 to find the twist rate at a given point in the span. This also allows finding the total shear stress acting on each skin section of the structure, which is the sum of the shear flow due to pure shear q_S and the one due to torsion q_T. Using Equation 4.10, it is then possible to find the shear stress.

$$\tau = \frac{q}{t} \tag{4.10}$$

4.7. Deflection

The total deflection of both the leading and trailing edge of the aileron has two contributors. The first is the deflection due to the internal bending moment. Secondly, the twist due to torsion on the aileron contributes to the maximum total deflection.

4.7.1. Deflection due to Torsion

The torsion applied on the cross section creates a twist on the structure around the hinge point. The explanations for the computation of the twist rate are given in subsection 4.6.2, and the angle of rotation is calculated using finite differences:

$$(\theta)_i = \left(\frac{d\theta}{dx}\right)_i \cdot \Delta x + (\theta)_{i-1} \tag{4.11}$$

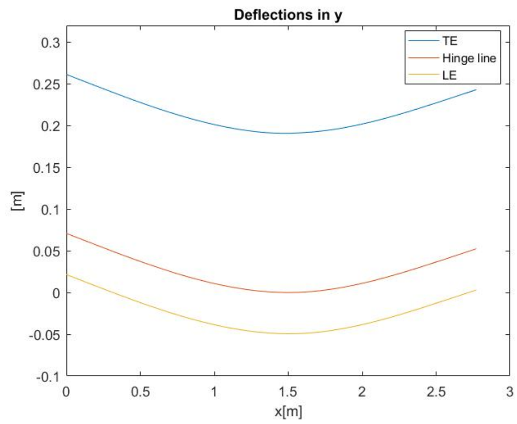
This rotation contributes to the deflection, which is calculated at the trailing and leading edge using trigonometric relations and the twist angle θ. The twist angle has been found to be small.

4.7.2. Twist influence on the deflections at the LE and TE

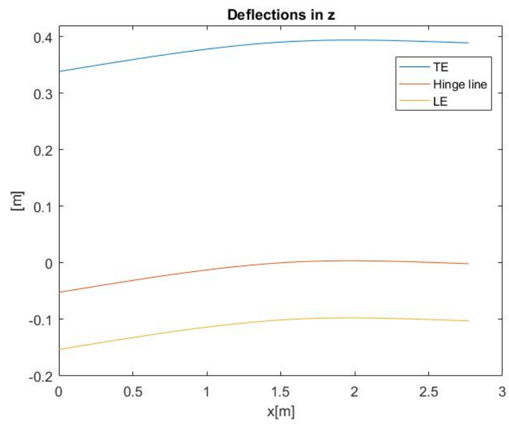
The deflection due to internal bending bending moment is calculated using the bending moment distributions found in section 3.3. The deformations at the leading and trailing edge are going to be different than at the hinge line because these extremes are rotated by θ degrees plus the twist angle which has a maximum value of slightly less than one degree. The plots of the deformations shown in Figure **??** are with respect to the hinge line, where the coordinate frame starts. Thus the value of the deformation is taken from this point. The actual deformation is almost the same for the leading and trailing edge, as the twist does not change the shape of the curve significantly. In fact, if the three lines would be shifted to be superimposed, the deformation curves would coincide.

The absolute maximum deformation with respect to the initial position of each edge always occurs at the left most end before hinge three for both the bending deformation and forward deformation (z direction). The leading edge is deflected up to 7.095 cm in the y axis and 5.595 cm in the z axis. The trailing edge has a deformation of 7.052 cm in the vertical direction and 5.567 cm in the z axis.

In the following plots it is possible to see the deformation of the leading edge, the trailing edge and the hinge line with respect to each other. the hinge line is not rotated as both the twist and the aileron inclination(θ) are about this line. The leading edge is at the bottom as it will move downwards when rotated while the trailing edge which is at the opposite extreme is above the hinge line because, the rotation is counterclockwise meaning it will move it upwards.



(a) Deflection of the aileron in the y direction along span



(b) Deflection of the aileron in the y direction along span

4.8. Results

The numerical model is required to provide the output values that were requested in Table 2.2. These results are presented in Table 4.2 in this section. The values for the maximum deflections of the leading and trailing edge in x-direction can be neglected.

Table 4.2: Required Outputs from the Numerical Model

Property	**Symbol**	**Value**	**Unit**
Maximum deflection leading edge, y-direction	$\delta_{LE,y}$	7.095	cm
Maximum deflection leading edge, z-direction	$\delta_{LE,z}$	5.595	cm
Maximum deflection trailing edge, y-direction	$\delta_{TE,y}$	7.052	cm
Maximum deflection trailing edge, z-direction	$\delta_{TE,z}$	5.567	cm
Maximum shear stress in rib A	$\tau_{max,A}$	0.638	MPa
Maximum shear stress in rib B	$\tau_{max,B}$	98.978	MPa
Maximum shear stress in rib C	$\tau_{max,C}$	166.793	MPa
Maximum shear stress in rib D	$\tau_{max,D}$	171.689	MPa
Maximum normal stress in rib A	$\sigma_{max,A}$	0.143	MPa
Maximum normal stress in rib B	$\sigma_{max,B}$	690.842	MPa
Maximum normal stress in rib C	$\sigma_{max,C}$	577.237	MPa
Maximum normal stress in rib D	$\sigma_{max,D}$	229.902	MPa

5
Verification

Verification is a crucial step in the simulation of the A320 aileron as it ensures the model's performance and accuracy. For the purpose of this report the verification has been split into two separate parts. First, unit testing will be conducted. In unit testing the each block of code that serves a specific function is tested separately. Afterwards, system testing will be performed. During the system testing, the complete code for the numerical model is run and the results will be compared to the results produced by the analytical model. This chapter will follow the same structure, unit testing will be discussed in section 5.1 and system testing will be explained in section 5.2.

5.1. Unit Testing

In this section the unit testing of the numerical model is discussed. The unit tests are performed on all individual modules in the computer code. Most of the tests have been conducted by applying the module to a different problem of which the correct result was known. The testing problems were mostly taken from [4], the specific problems will be stated in the respective subsections. Furthermore, the solution results presented in [4] were assumed to be correct. If the discrepancy between the results is less than 10%, the module is considered correct and verified. This maximum error limit has been chosen because, it does allow for some discrepancy to be present between the results, but also limits this discrepancy from getting too large. Errors that adhere to these limits can easily be accounted for by applying a maximum safety factor of 1.1, which is still reasonable.

Reaction forces Testing

The reaction forces are calculated only once with the equilibrium and the bending curvature equations, thus there is no numerical approach because the inertial properties were assumed to be constant along the span as the twist deformation will not affect this values . In fact the twist is predicted to have a maximum value of 1 degree. In order to validate the forces it is needed to check that the equations used are correct. The first test that was performed consists in comparing the forces when the deflections at the the three hinges are doubled. When performing this test the forces double their value as well this result is consistent since the forces are linearly dependent on the distance due to the moment they have to counteract. Furthermore when applying a virtual load of any magnitude at the hinges, the moment distribution does not change as the reaction forces will increase proportionally with the virtual load counteracting the contribution of it.

Syntax Error Testing

The verification of the numerical model will start with syntax error testing. The checking for syntax error is mainly performed by the python interpreter. The compiler checks the code of each module and also the complete code before it is run. It will return error messages when errors are detected. Since no error messages were displayed prior to running the modules or the complete code, the code is concluded to be free of syntax errors.

Boom Area Testing

The first module to be tested is the module that calculates the boom areas. The calculation of the boom areas is based on Equation 4.1. This module will be verified using problem 20.1 from [4]. The discrepancy in the results from this problem and the results calculated by the boom area module of the numerical model equated to 0.0%. This error is clearly within the limit given above. Therefore, this module has passed unit testing.

Moment of Inertia Testing

The module for the calculation of the moments of inertia is crucial to the correct operation of the rest of the model. Therefore, it is important to strictly verify this module. The module calculates the moments of inertia of the idealized cross section. For the verification of the moments of inertia calculation, example 20.2 from [4] was used. The error in the results from the example and the results from the numerical module is less the 5%, which is within the limit. Therefore, this module has passed unit testing and is considered accurate enough.

Additionally, the boom area module and the moment of inertia module were tested together using the problem presented on slide 43 in [5]. In this case the error in I_{zz} was equal to 0.0% and error in I_{yy} equated to 0.64%, both of which are clearly within the limit of 10%.

Furthermore, the moment of inertia in the numerical model is calculated using the principal of structural idealization. Applying structural idealization for calculating the moment inertia simplifies the geometry by assuming the area is concentrated in booms. In this report, 21 booms were used for the idealization of the cross section. However, idealizing the structure will also deteriorate the accuracy of the model. In order to check whether the loss in accuracy is acceptable, the moments of inertia calculated by the numerical model will be compared the moments of inertia calculated using the analytical model. The analytical model does not use structural idealization. Once again, the error between the values of the moments of inertia should not differ more than 10%. From the comparison, it has become evident that the difference is equal to 1.7% for I_{zz} and 8.48% for I_{yy}. Therefore, it has been verified that the assumption of structural idealization as valid and is not too detrimental to the accuracy of the model's moment of inertia calculation.

Since the errors in all three of the tests are within the limits as stated above, the boom area and moment of inertia module are considered correct and accurate enough.

Shear Flow due to Pure Shear Testing

The next module to be test is the module that calculates the shear flow due to pure shear. This module is tested using problem 23.6 from [4]. The results from the numerical model and the solution manual presented in [4] have once again been compared. The error between these two solutions equated to 0.013%. Therefore, it is concluded that this module has passed unit testing.

Shear Flow due to Torsion Testing

The shear flow due to torsion module has been tested in a similar way to shear flow due to pure shear. The module has been tested using that was designed specifically to test this module. The answer to this problem has been produced by hand calculations and checked by two individuals. The error between the results from the numerical model and hand-calculated solution equals 0.069%. Consequently, this module has been approved and is sufficiently accurate.

Normal Stress Testing

The module that calculates the normal stress distribution in the aileron is considered next. The module is also verified by using a problem from [6]. The specific problem used is problem 26 from [6]. Results are once again compared and the error between them comes to 0.01%. Therefore, this module is approved and considered accurate enough.

Shear Center Assumption Testing

In section 2.4 it is stated that the calculation is this report have been conducted under the assumption that the shear is locate at the hinge line. Extensive hand calculations have been conducted in order to verify the validity of the this assumption. From the hand calculations it can be concluded that the shear center is located $1.53mm$ to the right of the hinge line. Realizing that the chord length of the entire aileron is $547mm$, it can be concluded that the discrepancy between the assumption and the hand calculations is negligible. Meaning that, the assumption will not significantly influence the outcome of the simulation. Therefore, the validity of the assumption is considered verified and the assumption will be used throughout the simulation.

5.2. System Testing

Once unit testing has been completed and all module of the numerical model code have been verified, the entire model can be tested. This testing of the full code is often referred to as system testing. During system testing, several tests are conducted to show that the model is fully functional and that its performance and accuracy are up to the desired level. For the purpose of this report, the main part of the system testing will be conducted be comparing the results from the analytical model to the results of the numerical model. Several tests that will performed in order to get the full numerical model verified will be explained below.

Syntax Error Testing

First of all, syntax error tests will be conducted. The approach for syntax error testing is exactly the same as the approach described in section 5.1, which is used for the syntax error testing of the individual modules.

Functionality Testing

During functionality testing, it is tested whether the model fulfills all of its functional requirements. In short, this tests whether the model does what it is supposed to do. In this report the most important output that the model should deliver is the maximum deflection of the leading and trailing edge and the maximum shear flow in the ribs, as is mentioned in Table 2.2. Furthermore, it should be able to properly ingest the input data and load case as given in section 2.2 and section 2.3 respectively.

Functionality is mainly verified by running the program several times, using several different sets of input values. It should then be confirmed that all the desired output data is present and within a reasonable range.

Limit Testing

In limit testing several extreme values will be used as inputs for the code. These extreme values should lead to results that are easy calculate by hand, but the program might struggle with them. In this testing the results from numerical model have been compared to the results computed by hand.

The model has been tested at its zero-limit, which means that all inputted loads and forces were set to zero. The expected outcome that the model should provide is in this case $0MPa$ for the normal and shear stresses and no deflection. The model did indeed provide these outcomes. It is therefore concluded that the model works at its zero-limit.

Further limit testing is unfortunately not possible because, the reaction will then have to recalculated analytically, as is explained in **??**.

Repeatability Testing

Repeatability testing is used to confirm that the model is consistent in the output values that are returned. In order to test this, a set of input values should be ingested by the model. The program should then be run several times, each time the output data should be the same. This test can then be repeated with multiple sets of input data. For the numerical model in this report, three different sets of input data have been used. The test was then run five times for each input data set. It has been concluded that the model provides consistent outcomes. The answers of the five runs did not vary at all. Therefore, the numerical model has passed the repeatability test.

Comparison to the Analytical Model

Finally, the numerical model will be compared to the analytical model. This is the most important part of system testing as it verifies that the numerical is in agreement with the analytical model. Mainly, the structural idealization assumptions will be tested, because the analytical model does not use structural idealization. Additionally, the proper operation of the numerical model code as a whole can be verified.

In this test, the stresses and deflection calculated by the numerical model and the analytical model will be compared directly. The discrepancy between these two values should not exceed 10%. The discrepancy limit of 10% has been chosen because it does allow some difference between the values due to rounding and assumption differences between the two models, while the accuracy of the numerical model is still being safeguarded. Furthermore, a safety factor of 1.1 to be applied to the numerical results is still acceptable.

As the numerical model has already undergone repeatability testing and the limit test has proved that the input values from Table 2.1 are within the operating limits, the model will only be tested against the analytical model once.

A comparison was made between the maximum shear stresses and the maximum normal stresses at each rib. An error of 10 to 20 % was found for the maximum normal stresses in the ribs. The errors for the maximum shear stresses produce much larger errors, ranging from 48.7 % to 99.8 %.

While these errors are quite large, the maximum normal stresses show the same distribution, with the smallest normal stresses in rib A and the largest stresses in rib B.

Part of the error can be explained with the assumptions, the error in the maximum normal stresses is reasonable. However, the error in the maximum shear stresses is exceedingly big. Considering that all unit and system tests showed that the errors in the numerical model are small, it was concluded that these discrepancies are caused by a calculation error in the verification model. This will result in a thorough check of the verification model in the assumptions made, the values that were inputted and the underlying theory. This check, while vital, is outside the scope of this project.

From the verification, it can be concluded that the the preliminary decision can be made to use the numerical model, after it has gone through thorough validation. However, due to the large discrepancies both models will be rechecked, starting with the analytical model.

6
Validation

In this chapter numerical model from chapter 4 will be validated. Validation of the model means that the models results are compared to actual test data. This should test whether the model represents reality with sufficient accuracy. The numerical results will be compared to the provided validation data. When comparing the results, discrepancies will be discussed and recommendations will be made to improve the numerical model.

6.1. Validation with reference data

The validation data that has been provided featured the Von Mises stress at each of the nodes on the cross section on both the inner and outer side of the skin. In order to find one maximum value for the Von Mises stress that can be compared the results of the numerical model, the two given Von Mises stresses were averaged for each node.

Validating the numerical results with the given Von Mises stresses posed a significant problem. Since the numerical model uses structural idealization, it is not possible to calculate Von Mises stresses. Von Mises stresses provide a value for the stress at a certain point taking into account both the shear stresses and the normal stresses, using Equation 6.1. However, when structural idealization is applied, it is assumed the booms carry only normal stresses and the skin solely carries shear stresses. Therefore, it is not possible to calculate a stress value in a node that takes both the shear stress and normal stress into account.

In order to still provide some level of validation for the results provided by the numerical model, Table 6.1 provides an overview of the maximum normal and shear stresses found at the ribs by the numerical model and maximum Von Mises stresses found from the validation data. Furthermore, it has been assumed that in order to have the validation data match the numerical results more closely, it was assumed that the validation Von Mises stresses were given in GPa.

As can be seen from the data in Table 6.1, the maximum value of the validation data Von Misses stress is quite close to the maximum normal stress that was calculated by the numerical model, the error is 15.13%. Furthermore, the locations where these maximum values occur is also close. Due to the restrictions mentioned above, it is unfortunately not possible to make any valid claims on the maximum shear that was calculated by the numerical model.

Validation data was provided on the deflection of the aileron. Maximum values for the deflection of the leading and trailing edges as presented by both the numerical model and validation data are presented in Table 6.2.

$$Y = \sqrt{\frac{1}{2} \cdot [(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2] + 3\tau_{xy}^2 + 3\tau_{yx}^2 + \tau_{xz}^2} \tag{6.1}$$

Table 6.1: Overview of numerical solutions and validation data

Output Variable	Value	Location(x;y;z)	Unit
Maximum Von Misses stress	814	1281 ; -112.5 ; 0	MPa
Maximum normal stress numerical model	690.842	Rib B	MPa
Maximum shear stress numerical model	164.126	Rib D	MPa

Output variable	Numerical solution	Validation data	Unit
Maximum deflection LE	90.36	71.68	mm
Maximum deflection TE	89.85	267.3	mm

Table 6.2: Maximum and minimum deflections

6.2. Discussion and recommendation for improvement

As mentioned above, the assumptions induced by the use of structural idealization in the numerical do not allow proper validation of the stress calculations to be performed. Therefore, the results will be discussed but it is not possible to draw any valid conclusions from the stress calculation validation process. On the other hand, the validation of maximum deflections does allow for proper discussion and concluding of the validity of the numerical model.

As can be seen from Table 6.1, the maximum Von Mises stress taken from the validation data and the maximum normal stress from the numerical model are quite close. In fact, as mentioned above, the discrepancy between them is 15.13%. This might give an indication of the proper working of the numerical model. However, the conclusion can not be drawn since Von Mises stress is compared to normal stress. Furthermore, the locations where these maximum normal and Von Mises occur are also close, once again hinting that the numerical model functions well. The maximum shear stress indicated in Table 6.1 does show a large discrepancy with the validation data, both in value and location. This might indicate a mistake in the numerical model. However, since the locations of the maximum shear stress and the maximum Von Mises stress a far apart, it is not possible to draw a valid conclusion on the proper functioning of the model. Taking all of this into account, it is not possible to make any recommendations based on this validation test due to the structural idealization being applied.

From the results presented in Table 6.2, it can be concluded that the numerical model is somewhat accurate in the determination of the maximum deflection of the leading edge. The error between the numerical result and the validation data for the leading edge is 20.67%. In contrast, the error between for the maximum trailing edge deflection equates to 66.35%, which that it is highly inaccurate. Some of the discrepancy between the values can be attributed to the structural idealization applied in the numerical model. However, structural idealization should not return errors of this magnitude if implemented correctly. Therefore, it should be concluded that there is an error in the numerical model. This error specifically influences the trailing edge maximum deflection calculation significantly. Consequently, a revision of this part of the numerical model is strongly advised and the model is at this moment not deemed sufficiently accurate in the determination of the leading and trailing edge deflections.

7 Conclusion

Proper testing of ailerons is crucial, but expensive. To avoid unnecessary expenses, simulations are often made for aileron designs. This report shows an overview of the simulation, verification and validation process of a numerical model that describes how well the aileron will sustain a critical load condition. An explanation of the problem can be found in chapter 2.

During simulation, two models were made: a numerical model and an analytical model. The analytical model was produced for verification purposes only. More information on the numerical model can be found in chapter 4, chapter 3 elaborates on the analytical model.

Verification was done by performing unit tests and system tests, as can be found in chapter 5. From verification, it was found that all unit tests and all regular system tests provide small errors. The only large discrepancy occurred when comparing the results of the numerical model to the results of the analytical model. It was concluded that these discrepancies were most likely caused by an error in the analytical model. This is cause for a recheck of both models, starting with the analytical model. However, this is outside the scope of this project.

During validation, the numerical model was compared to experimental results. This comparison can be found in chapter 6. The experimental results were assumed to be the correct. During the comparison, there were a few indicators to the proper working of the model. However, due to structural idealization it was not possible to draw any valid conclusions from the experimental data.

The verification and validation were not entirely conclusive on the proper working of the numerical model. However, the unit and system tests showed that the underlying methods are correct. Also, the deflections, the maximum stresses and the locations where they occur are within reason and as expected.

Bibliography

[1] J. Blok, K. Burger, L. J. Ferrer, L. Losch, and D. M. Jimenez, *Simulation Plan, Simulation, Verification and Validation*, Tech. Rep. (Delft University of Technology, 2018).

[2] A. A. S. M. Inc., *Aluminum 2024-t3,* http://asm.matweb.com/search/SpecificMaterial.asp?bassnum=ma2024t3, retrieved:28-02-2018.

[3] MIT, *Module notes simple beam theory,* http://web.mit.edu/16.20/homepage/7_SimpleBeamTheory/SimpleBeamTheory_files/module_7_no_solutions.pdf (2016), retrieved: 15-02-2018.

[4] T. Megson, *Aircraft structures for engineering students*, edited by Butterworth-Heinemann (2012).

[5] M. Damghani, *Lec6-aircraft structural idealisation 1,* https://www.slideshare.net/scemd3/lec6aircraft-structural-idealisation-1 (2017), retrieved:27-02-2018.

[6] L. Noels, *Aircraft structures: Aircraft component - part 1,* , retrieved:01-03-2018.

A
Appendix A: Work division

Table A.1: My caption

Task	Total time	Janneke Blok	Laura Jou	Lex Losch	Daniel Martini	Klaas Burger
Programming numerical model	150		80	10	60	
Producing analytical model	120	60		30		30
Debugging	40		20	10	10	
Verification	25	5				20
Validation	35			20	5	10
Reporting	90	20	5	20	15	30
Checking of the report	15	7	2	2	2	2
Total	475	92	107	92	92	92

B
Code listing

```
syms x R1y R2y R3y R1x R2x R3x R1z R2z Fh K1 K2 C1 C2
format long
%Input data
P=9.17e4;
ha= 22.5e-2;
q=4.53e3;
theta=26*pi/180;
la=2.771;
steps=100;
step_size=la/steps;
Ca=0.547;
z4=Ca*0.25;
zh=ha/2;
x1=la-0.153;
x2=la-1.281;
x3=la-2.681;
xa=28e-2;
E=73.1e9;
Ixx=5;
Izz=12377899.7e-12;%12173000e-12;%
Iyy=65896885.75e-12;%60758000e-12;%
Icentroid=[Ixx,0,0;0,Iyy,0;0,0,Izz];
Rot=[1,0,0;0,cos(theta),sin(theta);0 -sin(theta) cos(theta)];
Inew=Rot'*Icentroid*Rot;
Izz_new=Inew(3,3);
Iyy_new=Inew(2,2);
Izy_new=Inew(2,3);

%Transform this displacements in new axis
delta1=11.03e-2/2.54;
delta2=0;
delta3=16.42e-2/2.54;

%Force equilibrium
eqn1= R1x + R2x +R3x==0;
eqn2= R1y + R2y +R3y-q*la ==0;
eqn3=R1z+ R2z +P+Fh==0;
%Moment equilibrium
%Mz
eqn4=-R3y*abs(x3-x2)+R1y*abs(x2-x1)+q*la*abs(la/2-x2)==0;
%Mx
eqn5 =
    ↪ Fh*ha/2*(cos(theta)-sin(theta))+R1z*delta1+q*la*cos(theta)*(z4-zh)+P*ha/2*(cos(theta)-sin(theta))==0;
%My
eqn6=R1z*abs(x2-x1)+Fh*xa/2-P*xa/2==0;

%V(x)
Vy=(-q*x+R1y*heaviside(x-x1)+R2y*heaviside(x-x2)+R3y*heaviside(x-x3))*-1;
```

```
Vz=R1z*heaviside(x-x1)+Fh*heaviside(x-x2-xa/2)+R2z*heaviside(x-x2)+P*heaviside(x-x2+xa/2);
%M(x)
Mz=(-q*x^2/2+R1y*(x-x1)*heaviside(x-x1)+R2y*(x-x2)*heaviside(x-x2)+R3y*(x-x3)*heaviside(x-x3))*-1;
My=R1z*(x-x1)*heaviside(x-x1)+Fh*(x-x2-xa/2)*heaviside(x-x2-xa/2)+R2z*(x-x2)*heaviside(x-x2)+
    P*(x-x2+xa/2)*heaviside(x-x2+xa/2);
Mx=
    P*(ha/2)*-1*(sin(theta)-cos(theta))*heaviside(x-x2+xa/2)+Fh*(ha/2)*(cos(theta)-sin(theta))*heaviside(x-
   R1z*(delta1)*heaviside(x-x1)+q*x*cos(theta)*(z4-zh);
%Start moment curvature integration
dslopes=-1/(E*(Izz_new*Iyy_new-Izy_new^2))*[-Izy_new,Izz_new;Iyy_new,-Izy_new]*[Mz;My];
%Solve equation for y
d2y=dslopes(2);
slopey=int(d2y,x);
deflectiony=int(slopey,x);
%Set Bcs
eqn7=simplify(subs(deflectiony, [x], [x1]))+K1*x1+K2==delta1;
eqn8=simplify(subs(deflectiony, [x], [x2]))+K1*x2+K2==delta2;
eqn9=simplify(subs(deflectiony, [x], [x3]))+K1*x3+K2==delta3;
%Solve system of equations
[A,B] = equationsToMatrix([eqn2,eqn3,eqn4, eqn5,eqn6,eqn7,eqn8,eqn9], [R1y R2y R3y R1z R2z
    Fh K1 K2]);
X = linsolve(A,B);

%Solve equation for z
d2z=dslopes(1);
slopez=int(d2z,x);
deflectionz=int(slopez,x);
%Set Bcs
eqn10=simplify(subs(deflectionz, [x], [x1]))+C1*x1+C2==0;
eqn11=simplify(subs(deflectionz, [x], [x2]))+C1*x2+C2==0;
%Solve system of equations
[C,D] = equationsToMatrix([eqn10,eqn11], [C1 C2]);
T = linsolve(C,D);

%Get expression for moments and shear diagrams
Momy=subs(My, [R1z R2z Fh], [X(4) X(5) X(6)]);
Momz=subs(Mz, [R1y R2y R3y], [X(1) X(2) X(3)]);
Momx=subs(Mx, [R1y R2y R3y R1z R2z Fh], [X(1) X(2) X(3) X(4) X(5) X(6)]);
Sz=subs(Vz, [R1z R2z Fh], [X(4) X(5) X(6)]);
Sy=subs(Vy, [R1y R2y R3y], [X(1) X(2) X(3)]);
Sx=0;

%Rotate forces and moments
Mom=rotx(theta*180/pi)*[Momx;Momy;Momz];
S=rotx(theta*180/pi)*[Sx;Sy;Sz];

%Get expression for deflection (x)
y=vpa(subs(deflectiony, [R1y R2y R3y R1z R2z Fh], [X(1) X(2) X(3) X(4) X(5)
    X(6)])+X(7)*x+X(8));
deflectionzz=deflectionz+T(1)*x+T(2);
z=vpa(subs(deflectionzz, [R1y R2y R3y R1z R2z Fh], [X(1) X(2) X(3) X(4) X(5) X(6)]));

%Print forces
R1y=vpa(X(1))/1000
R2y =vpa(X(2))/1000
R3y =vpa(X(3))/1000
R1z=vpa(X(4))/1000
R2z=vpa(X(5))/1000
```

```
Fh=vpa(X(6))/1000

%Make plots
t=0:step_size:2.771;
%Diagrams
Mx_a=subs(Mom(1), [x], [t]);
My_a=subs(Mom(2), [x], [t]);
Mz_a=subs(Mom(3), [x], [t]);
Sy_a=subs(S(2), [x], [t]);
Sz_a=subs(S(3), [x], [t]);
figure(1);
ax1 = subplot(1,1,1);
plot(ax1,t,My_a);title(ax1,'Moments');xlabel('x[m]');ylabel('[Nm]');
hold on
plot(ax1,t,Mz_a);%title(ax2,'Mz');xlabel('x[m]');ylabel('[Nm]');
legend('My','Mz')

figure(2);
ax2 = subplot(1,1,1);
plot(ax2,t,Mx_a);title(ax2,'Torque');xlabel('x[m]');ylabel('Mx[Nm]');

figure(3);
ax3 = subplot(1,1,1);
plot(ax3,t,Sz_a);title(ax3,'Shear Forces');xlabel('x[m]');ylabel('[N]');
hold on
plot(ax3,t,Sy_a);%title(ax5,'Sy');xlabel('x[m]');ylabel('Sy[N]');
legend('Sz','Sy')


%Deflections plots
figure(4);
ax4 = subplot(1,1,1);
yplot=subs(y, [x], [t]);
plot(ax4,t,yplot);title(ax4,'Deflections');xlabel('x[m]');ylabel('[m]');
hold on
zplot=subs(z, [x], [t]);
plot(ax4,t,zplot);
%plot3(ax4,t,zplot);%title(ax7,'Deflection z');xlabel('x[m]');ylabel('z[m]');
legend('z','y')


%Write text files for python communication
fileID = fopen('Deflections.txt','w');
fprintf(fileID,'%6s %6s %6s\n','x_i','z_i','y_i');
for i=1:length(zplot)
fprintf(fileID,'%1.2f %1.5f %1.5f\n',t(i),zplot(i),yplot(i));
end
fclose(fileID);

fileID = fopen('Loads.txt','w');
fprintf(fileID,'%6s %6s %6s %6s %6s %6s\n','x_i','Mx_i','My_i','Mz_i','Sy_i','Sz_i');
for i=1:length(zplot)
fprintf(fileID,'%1.2f %1.5f %1.5f %1.5f %1.5f %1.5f
  ↪  \n',t(i),Mx_a(i),My_a(i),Mz_a(i),Sy_a(i),Sz_a(i));
end
fclose(fileID);
```

```
My_a=subs(Mom(2), [x], [t]);
Mz_a=subs(Mom(3), [x], [t]);
sloc =
  ↪  [[1,-223.64,0];[2,-167.39,79.55];[3,-56.82,101.68];[4,-2.515,87.16];[5,51.80,72.63];...

      ↪  [6,106.11,58.10];[7,160.42,43.58];[8,214.74,29.05];[9,269.05,14.53];[10,269.05,-14.53];...
    [11,214.74,-29.05];[12,160.42,-43.58];[13,106.11,-58.10];[14,51.80,-72.63];...
    [15,-2.515,-87.16];[16,-56.83,-101.68];[17,-167.39,-79.55]];
stress=[];
riba=2.681;
ribb=1.421;
ribc=1.141;
ribd=0.513;
ribs=[riba,ribb,ribc,ribd];
for i=1:length(sloc)
  stressy=subs(Mom(2), [x], [ribd])*sloc(i,2)/Iyy;
  stressz=subs(Mom(3), [x], [ribd])*sloc(i,3)/Izz;
  stress(i)=(stressy+stressz)/1000;
end
max_stress=vpa(max(stress(:)))/1000000;
idx=find(stress == max(stress(:)));
sloc_maxstress=sloc(3,:);
```

```
import numpy as np
import math
import helpers
import edges

class Boom():
    def __init__(self, number, coordinates, stringer_area, neutral_axis):
        """
        Initialise instance of boom for structural idealisation.
        :param coordinates: coordinates (z, y) of boom location. Origin is taken at hinge
  ↪  point.
        :param adjacents: list of adjacent booms. List of number_booms length where each
  ↪  element is a list that contains
        details about each adjacent boom. Each element contains [boom number, thickness of
  ↪  edge, length of edge].
        :param stringer_area: Area of the stringers. If there are no stringers in the place
  ↪  of the boom, set to 0.0.
        :param neutral_axis: line of the neutral axis. Format: (A, B, C) where the neutral
  ↪  axis: Ax + By + C = 0
        """
        self.neutral_axis = neutral_axis
        self.coordinates = coordinates
        self.adjacents = []
        self.stringer_area = stringer_area
        self.dist_neutral_axis = 0.0
        self.area = 0.0
        self.z_dist = 0.0
        self.y_dist = 0.0
        self.number = number
        self.dist_origin_coordinates = 0.0
        self.bending_stress = None

```

```
    def calc_distance_neutral_axis(self):
        """
        calculate and update distance from boom to neutral axis
        """
        self.dist_neutral_axis = helpers.distance_point_line(self.coordinates,
          ↪ self.neutral_axis)

    def calc_dist_origin_coordinates(self):
        """
        calculate the distance from the boom to the origin of coordinates. This is useful to
    ↪  fid the new coordinates
        after a rotation
        update value for each boom
        """
        self.dist_origin_coordinates = (self.coordinates[0] ** 2 + self.coordinates[1] ** 2)
          ↪ **0.5

    def update_coordinates(self, theta):
        rotation_matrix = np.array([[np.cos(theta), -np.sin(theta)],
                                    [np.sin(theta), np.cos(theta)]])
        new_coords = np.dot(rotation_matrix, np.asarray(self.coordinates))
        self.coordinates = new_coords

    def calc_y_dist(self, aileron_geometry):
        """
        :param aileron_geometry: geometry of the cross-section, we need the centroid from it
        update distance from boom to centroid in y-direction
        """
        self.y_dist = self.coordinates[1] - aileron_geometry.centroid[1]

    def calc_z_dist(self, aileron_geometry):
        """
        :param aileron_geometry: geometry of the cross-section, we need the centroid from it
        update distance from boom to centroid in z-direction
        """
        self.z_dist = self.coordinates[0] - aileron_geometry.centroid[0]

    def calculate_area(self, aileron_geometry):
        """
        Calculate area of boom following formula 20.1 of Megson
        :param aileron_geometry: instance of class Geometry describing the geometrical
    ↪  properties of the cross-section
        update area of boom
        """
        boom_area = self.stringer_area
        self.calc_distance_neutral_axis()
        for adjacent_edge in self.adjacents:
            if adjacent_edge.booms[0] != self.number:
                boom = adjacent_edge.booms[0]
            else:
                boom = adjacent_edge.booms[1]
            boom_obj = aileron_geometry.booms[boom]
            boom_obj.calc_distance_neutral_axis()
            t = adjacent_edge.thickness     # thickness of link
            l = adjacent_edge.length        # length of link
            if boom_obj.coordinates[0] == self.coordinates[0] and boom_obj.coordinates[1] ==
              ↪ - self.coordinates[1]:
                ratio = -1
```

```
            else:
                if abs(self.coordinates[1]) < 0.001:
                    continue
                else:
                    ratio = boom_obj.dist_neutral_axis / self.dist_neutral_axis
            boom_area += (t * l)/6.0 * (2 + ratio)
        self.area = boom_area

    def calc_bending_stress(self, Mz, My, aileron_geometry):
        """
        Calculates bending stresses at given point (z, y) in the particular section of the
        ↪ aileron
        :param Mz: Moment distribution at given point in x
        :param My: Moment distribution at given point in x
        update Bending stress at given point in the cross-section at given point in x
        ↪ direction
        """
        moment_contribution = (Mz * self.y_dist) / aileron_geometry.Izz + (My * self.z_dist)
            ↪ / aileron_geometry.Iyy
        self.bending_stress = moment_contribution
```

```
class Edge:
    def __init__(self, booms, thickness, length):
        """

        :param booms: list of booms that the edge is uniting
        :param thickness: thickness of the skin section the edge represents
        :param length: length of the skin section the edge represents
        """
        self.thickness = thickness
        self.length = length
        self.booms = booms
        self.q_B = 0.0
        self.q_0 = 0.0
        self.q_total = 0.0
        self.q_T = 0.0
        self.shear_stress = 0.0
```

```
import numpy as np
import math
from helpers import *


class Geometry:
    def __init__(self, number_booms, booms, edges, cells_area, G):
        """
        :param number_booms: Number of booms in the cross section
        :param booms: list of Boom class instances containing all booms in the cross section
        :param edges: list of Edge class instances containing all edges in the cross section
        :param cells_area: list containing the areas of the cells (for multicell problems)
        :param G: shear modulus of the material
        """
```

```
        self.number_booms = number_booms
        self.booms = booms
        self.edges = edges
        self.cells = []
        self.boom_areas = np.zeros(self.number_booms)
        self.centroid = np.zeros(2)
        self.neutral_axis = ()  # in the form A, B, C : neutral axis line Az + By + C = 0
        self.z_dists = np.zeros(self.number_booms)
        self.y_dists = np.zeros(self.number_booms)
        self.Iyy = 0.0
        self.Izz = 0.0
        self.Izy = 0.0
        self.shear_center = 0.0
        self.cells_area = cells_area
        self.G = G


    def construct_geometry(self):
        """
        modify all Boom objects. For each object, modify their attribute "adjacents" to include a list of all the edges
        that contain the boom.
        """
        for element in self.booms:
            for edge in self.edges:
                if element.number in edge.booms:
                    element.adjacents.append(edge)

    def get_areas(self):
        """
        Update values in self.boom_areas to include the areas of the booms.
        """
        for i, boom in enumerate(self.booms):
            self.boom_areas[i] = boom.area

    def calc_centroid(self):
        """
        Calculate centroid position (z, y) taking as origin of coordinates the hinge point.
        Set self.centroid to calculated coordinates.
        """
        sum_y = 0.0
        sum_z = 0.0
        for boom in self.booms:
            sum_y += boom.area * boom.coordinates[1]
            sum_z += boom.area * boom.coordinates[0]

        self.centroid[1] = sum_y/sum(self.boom_areas)
        self.centroid[0] = sum_z/sum(self.boom_areas)

    def calc_y_dists(self):
        """
        Calculates the distance in y-direction from each boom to the centroid and store in a list y_dists.
        Modifies self.y_dists[]
        """
        for i, boom in enumerate(self.booms):
            self.y_dists[i] = boom.coordinates[1] - self.centroid[1]

```

```
    def calc_z_dists(self):
        """
        Calculates the distance in z-direction from each boom to the centroid and store in a
        ↪ list z_dists.
        Modifies self.z_dists[]
        """
        for i, boom in enumerate(self.booms):
            self.z_dists[i] = boom.coordinates[0] - self.centroid[0]

    def moment_inertia_Izz(self):
        """
        Calculates moment of inertia in z using Izz = Sigma(Bi * yi^2)
        Updates self.Izz to moment of inertia.
        """
        self.calc_y_dists()
        for i, area in enumerate(self.boom_areas):
            self.Izz += area * self.y_dists[i] ** 2

    def moment_inertia_Iyy(self):
        """
        Calculates moment of inertia in y using Izz = Sigma(Bi * zi^2)
        Updates self.Iyy to moment of inertia.
        """
        self.calc_z_dists()
        for n, area in enumerate(self.boom_areas):
            self.Iyy += area * self.z_dists[n] ** 2

    def plot_edges(self):
        """
        Plot the booms (numbered) and the edges uniting them.
        This plot is used to verify that the booms and edges created correspond to the
        ↪ correct geometry.
        """
        coordinates = []
        for element in self.booms:
            coordinates.append(element.coordinates)
        zs = []
        ys = []
        n = range(len(coordinates))
        for boom_coord in coordinates:
            zs.append(boom_coord[0])
            ys.append(boom_coord[1])
        for wall in self.edges:
            z_positions = [self.booms[wall.booms[0]].coordinates[0],
              ↪ self.booms[wall.booms[1]].coordinates[0]]
            y_positions = [self.booms[wall.booms[0]].coordinates[1],
              ↪ self.booms[wall.booms[1]].coordinates[1]]
            plt.plot(z_positions, y_positions, color='black')
        plt.scatter(zs, ys)
        for i, txt in enumerate(n):
            plt.annotate(txt, (zs[i], ys[i]))
        plt.show()
```

```
import numpy as np
import math
```

```python
from helpers import *
from sympy import nsolve


C_a = 0.547
l_a = 2.771
x_1 = 0.153
x_2 = 1.281
x_3 = 2.681
x_a = 0.28
h_a = 0.225
t_sk = 0.0011
t_sp = 0.0029
t_st = 0.0012
h_st = 0.015
w_st = 0.02
n_st = 17
d_1 = 0.1103
d_3 = 0.1642
theta = 26
P = 9170
q = 4530


class DiscreteSection:
    def __init__(self, neutral_axis, aileron_geometry):
        """
        :param neutral_axis: neutral axis of the cross section
        :param aileron_geometry: instance of the class Geometry containing information on
        ↪ the cross section's geometry
        """
        self.neutral_axis = neutral_axis
        self.bending_stress = None
        self.bending_deflection = None
        self.aileron_geometry = aileron_geometry
        self.twist_rate = None
        self.T_1 = None
        self.T_0 = None
        self.q_T_array = None

    def calc_shear_flow_q_B(self, Sz, Sy, wall_common):
        """
        Calculate the open section shear flow q_B by the traditional method: make an
        ↪ imaginary cut.
        :param Sz: Shear force in z
        :param Sy: Shear force in y
        :param wall_common: wall that both cells have in common
        Modifies the value of q_B of each wall to the correct value.
        """
        Izz = self.aileron_geometry.Izz
        Iyy = self.aileron_geometry.Iyy
        Izy = self.aileron_geometry.Izy

        inertia_term_z = - (Sz * Izz - Sy * Izy) / (Izz * Iyy - Izy ** 2)
        inertia_term_y = - (Sy * Iyy - Sz * Izy) / (Izz * Iyy - Izy ** 2)
        number_cells = len(self.aileron_geometry.cells)
        for num_cell in range(number_cells):
            # make the cut at the first wall of each cell
```

```python
            wall_cut = self.aileron_geometry.cells[num_cell][0]
            wall_cut.q_B = 0.0
            accumulation_y, accumulation_z = 0.0, 0.0
            for n, wall in enumerate(self.aileron_geometry.cells[num_cell]):
                # only calculate the qB if it is not the wall that you have already cut or the common wall
                if wall == wall_cut:
                    continue
                if wall == wall_common and num_cell > 0:
                    accumulation_y += self.aileron_geometry.booms[wall.booms[1]].area * \
                                  self.aileron_geometry.booms[wall.booms[1]].y_dist
                    accumulation_z += self.aileron_geometry.booms[wall.booms[1]].area * \
                                      self.aileron_geometry.booms[wall.booms[1]].z_dist
                    continue
                accumulation_y += self.aileron_geometry.booms[wall.booms[0]].area * \
                                  self.aileron_geometry.booms[wall.booms[0]].y_dist
                accumulation_z += self.aileron_geometry.booms[wall.booms[0]].area * \
                                  self.aileron_geometry.booms[wall.booms[0]].z_dist
                if wall == wall_common:
                    continue
                wall.q_B = (inertia_term_z * accumulation_z + inertia_term_y * accumulation_y)

        # find shear flow on web
        contribution = 0.0
        for adjacent_to_common_wall in self.aileron_geometry.booms[wall_common.booms[0]].adjacents:
            if adjacent_to_common_wall != wall_common:
                if adjacent_to_common_wall in self.aileron_geometry.cells[0]:
                    contribution += adjacent_to_common_wall.q_B
                else:
                    contribution += -adjacent_to_common_wall.q_B
        inertia_contribution_z = inertia_term_z * self.aileron_geometry.booms[wall_common.booms[0]].area *\
                                 self.aileron_geometry.booms[wall_common.booms[0]].z_dist
        inertia_contribution_y = inertia_term_y * self.aileron_geometry.booms[wall_common.booms[0]].area *\
                                 self.aileron_geometry.booms[wall_common.booms[0]].y_dist
        wall_common.q_B = contribution + inertia_contribution_y + inertia_contribution_z

    def calc_closed_section_pure_shear_flow_q_0(self, wall_common):
        """
        Find closed section shear flow due to pure shear q_0
        Modify the q_0 value of each wall to the corresponding q_0
        """
        integral_term_list = np.zeros(2)
        delta_total_list = []
        for num_cell, cell in enumerate(self.aileron_geometry.cells):
            integral_term = 0
            delta_total_term = 0
            for wall in cell:
                delta_total_term += wall.length/wall.thickness
                if wall == wall_common and num_cell > 0:
                    integral_term += - wall.q_B * wall.length/wall.thickness
                    continue
                integral_term += wall.q_B * wall.length/wall.thickness
            integral_term_list[num_cell] = integral_term
            delta_total_list.append(delta_total_term)
```

```python
        delta_common = wall_common.length/wall_common.thickness
        # create system of equations that when solved gives you q_01 and q_02
        matrix_1 = np.array([[-delta_common, delta_total_list[0]], [delta_total_list[1], -delta_common]])
        matrix_2 = integral_term_list
        q_s_0_array = np.linalg.solve(matrix_1, matrix_2)
        for number_cell, cell_list in enumerate(self.aileron_geometry.cells):
            for wall in cell_list:
                if wall != wall_common:
                    wall.q_0 = q_s_0_array[number_cell]
        wall_common.q_0 = q_s_0_array[0] - q_s_0_array[1]

    def calc_torsion_shear_flow(self, T, common_wall):
        """
        Find the shear due to pure torsion q_T
        :param T: Torque applied on the structure
        Modify the value of q_T on each wall to the correct value
        Compute the twist rate at this point and modify the attribute twist_rate of the section
        """
        integral_cell_0, integral_cell_1 = 0.0, 0.0
        for i, wall in enumerate(self.aileron_geometry.cells[0]):
            integral_cell_0 += (wall.length/(wall.thickness * self.aileron_geometry.G))/\
                               (2 * self.aileron_geometry.cells_area[0])
        for n, skin in enumerate(self.aileron_geometry.cells[1]):
            integral_cell_1 += (skin.length/(skin.thickness * self.aileron_geometry.G)) * 1/\
                               (2 * self.aileron_geometry.cells_area[1])
        common_term_0 = common_wall.length / \
                        (2 * self.aileron_geometry.cells_area[0] * common_wall.thickness * self.aileron_geometry.G)
        common_term_1 = common_wall.length / \
                        (2 * self.aileron_geometry.cells_area[1] * common_wall.thickness * self.aileron_geometry.G)

        A = np.array([[1, 1, 0, 0],
                      [-1, 0, 2 * self.aileron_geometry.cells_area[0], 0],
                      [0, -1, 0, 2 * self.aileron_geometry.cells_area[1]],
                      [0, 0, integral_cell_0 + common_term_0, - integral_cell_1 - common_term_1]])

        B = np.array([[T],
                      [0],
                      [0],
                      [0]])
        # solve the system
        solutions = np.linalg.solve(A, B)
        # insert values in attributes
        self.T_0, self.T_1 = solutions[0], solutions[1]
        self.q_T_array = np.array([solutions[2], solutions[3]])
        for number_cell, cell_list in enumerate(self.aileron_geometry.cells):
            for wall in cell_list:
                wall.q_T = self.q_T_array[number_cell]
        self.twist_rate = (self.q_T_array[0] * integral_cell_0 - self.q_T_array[1] * common_term_0)

    def calc_total_shear_flow(self, Sz, Sy, T, wall_common):
        """
```

```
        Calculate total shear flow including due to pure shear and due to pure torsion
        :param Sz: Shear force in z
        :param Sy: Shear force in y
        :param T: Total torque around the shear center
        :param wall_common: wall the two cells have in common
        Modify the attribute q_total on each wall to the correct value
        """
        self.calc_shear_flow_q_B(Sz, Sy, wall_common)
        self.calc_closed_section_pure_shear_flow_q_0(wall_common)
        self.calc_torsion_shear_flow(T, wall_common)

        for edge in self.aileron_geometry.edges:
            edge.q_total = (edge.q_0 + edge.q_B + edge.q_T)/0.1

    def calc_shear_stress(self):
        """
        Calculate shear stress on each wall
        IMPORTANT: this function must be called AFTER self.calc_total_shear_flow()
        Modify attribute shear_stress on each wall to the correct value
        """
        for wall in self.aileron_geometry.edges:
            wall.shear_stress = wall.q_total / wall.thickness
```

```
import unittest
import geometry
import helpers
import numpy as np
import math
import boom
import edges
import DiscreteSection
import matplotlib.pyplot as plt

def initialise_problem():
    stringer_area = 42*10**(-6)
    neutral_axis = (0, 1, 0)

    # CREATE LIST OF COORDINATES FOR BOOMS
    # give the coordinates of the booms with respect to the hinge point
    # for the first eight, they are on a straight line
    coordinates = []
    for n in range(16):
        coordinates.append([((43.45 - 5.43125/2 * (n + 1)) * 10)*10**(-3), ((1.40625/2 * (n
        ↪  + 1)) * 10)*10**(-3)])
    # booms 8, 9 and 10 are along a semi-circle
    coordinates.append([-112.5 * math.sin(math.pi / 8)*10**(-3), 112.5 * math.cos(math.pi /
      ↪  8)*10**(-3)])
    coordinates.append([-112.5 * math.sin(math.pi / 4)*10**(-3), 112.5 * math.cos(math.pi /
      ↪  4)*10**(-3)])
    coordinates.append([-112.5 * math.sin(3 * math.pi / 8)*10**(-3), 112.5 * math.cos(3 *
      ↪  math.pi / 8)*10**(-3)])
    coordinates.append([-112.5*10**(-3), 0.0])

    # the last 8 are symmetric to the first eight wrt the z-axis
    for i in range(18, -1, -1):
```

```
            coords = coordinates[i]
            coordinates.append([coords[0], -coords[1]])
        # the ones on the spar are always at z=0 and distributed equally along the height of the
          ↪  spar
        coordinates.append([0.0, (22.5 + 45)*10**(-3)])
        coordinates.append([0.0, 22.5*10**(-3)])
        coordinates.append([0.0, -22.5*10**(-3)])
        coordinates.append([0.0, (-22.5 - 45)*10**(-3)])


        # CREATE BOOM INSTANCES AND INSERT THEM IN GEOMETRY
        booms = []
        boom0 = boom.Boom(0, coordinates[0], 0.0, neutral_axis)
        booms.append(boom0)
        boom1 = boom.Boom(1, coordinates[1], stringer_area, neutral_axis)
        booms.append(boom1)
        boom2 = boom.Boom(2, coordinates[2], 0.0, neutral_axis)
        booms.append(boom2)
        boom3 = boom.Boom(3, coordinates[3], stringer_area, neutral_axis)
        booms.append(boom3)
        boom4 = boom.Boom(4, coordinates[4], 0.0, neutral_axis)
        booms.append(boom4)
        boom5 = boom.Boom(5, coordinates[5], stringer_area, neutral_axis)
        booms.append(boom5)
        boom6 = boom.Boom(6, coordinates[6], 0.0, neutral_axis)
        booms.append(boom6)
        boom7 = boom.Boom(7, coordinates[7], stringer_area, neutral_axis)
        booms.append(boom7)
        boom8 = boom.Boom(8, coordinates[8], 0.0, neutral_axis)
        booms.append(boom8)
        boom9 = boom.Boom(9, coordinates[9], stringer_area, neutral_axis)
        booms.append(boom9)
        boom10 = boom.Boom(10, coordinates[10], 0.0, neutral_axis)
        booms.append(boom10)
        boom11 = boom.Boom(11, coordinates[11], stringer_area, neutral_axis)
        booms.append(boom11)
        boom12 = boom.Boom(12, coordinates[12], 0.0, neutral_axis)
        booms.append(boom12)
        boom13 = boom.Boom(13, coordinates[13], stringer_area, neutral_axis)
        booms.append(boom13)
        boom14 = boom.Boom(14, coordinates[14], 0.0, neutral_axis)
        booms.append(boom14)
        boom15 = boom.Boom(15, coordinates[15], 0.0, neutral_axis)
        booms.append(boom15)

        # semi circle booms
        boom16 = boom.Boom(16, coordinates[16], 0.0, neutral_axis)
        booms.append(boom16)
        boom17 = boom.Boom(17, coordinates[17], stringer_area, neutral_axis)
        booms.append(boom17)
        boom18 = boom.Boom(18, coordinates[18], 0.0, neutral_axis)
        booms.append(boom18)
        boom19 = boom.Boom(19, coordinates[19], stringer_area, neutral_axis)
        booms.append(boom19)
        boom20 = boom.Boom(20, coordinates[20], 0.0, neutral_axis)
        booms.append(boom20)
        boom21 = boom.Boom(21, coordinates[21], stringer_area, neutral_axis)
        booms.append(boom21)
```

```
    boom22 = boom.Boom(22, coordinates[22], 0.0, neutral_axis)
    booms.append(boom22)

    # lower straight line booms
    boom23 = boom.Boom(23, coordinates[23], 0.0, neutral_axis)
    booms.append(boom23)
    boom24 = boom.Boom(24, coordinates[24], 0.0, neutral_axis)
    booms.append(boom24)
    boom25 = boom.Boom(25, coordinates[25], stringer_area, neutral_axis)
    booms.append(boom25)
    boom26 = boom.Boom(26, coordinates[26], 0.0, neutral_axis)
    booms.append(boom26)
    boom27 = boom.Boom(27, coordinates[27], stringer_area, neutral_axis)
    booms.append(boom27)
    boom28 = boom.Boom(28, coordinates[28], 0.0, neutral_axis)
    booms.append(boom28)
    boom29 = boom.Boom(29, coordinates[29], stringer_area, neutral_axis)
    booms.append(boom29)
    boom30 = boom.Boom(30, coordinates[30], 0.0, neutral_axis)
    booms.append(boom30)
    boom31 = boom.Boom(31, coordinates[31], stringer_area, neutral_axis)
    booms.append(boom31)
    boom32 = boom.Boom(32, coordinates[32], 0.0, neutral_axis)
    booms.append(boom32)
    boom33 = boom.Boom(33, coordinates[33], stringer_area, neutral_axis)
    booms.append(boom33)
    boom34 = boom.Boom(34, coordinates[34], 0.0, neutral_axis)
    booms.append(boom34)
    boom35 = boom.Boom(35, coordinates[35], stringer_area, neutral_axis)
    booms.append(boom35)
    boom36 = boom.Boom(36, coordinates[36], 0.0, neutral_axis)
    booms.append(boom36)
    boom37 = boom.Boom(37, coordinates[37], stringer_area, neutral_axis)
    booms.append(boom37)
    boom38 = boom.Boom(38, coordinates[38], 0.0, neutral_axis)
    booms.append(boom38)

    # booms on the spar
    boom39 = boom.Boom(39, coordinates[39], 0.0, neutral_axis)
    booms.append(boom39)
    boom40 = boom.Boom(40, coordinates[40], 0.0, neutral_axis)
    booms.append(boom40)
    boom41 = boom.Boom(41, coordinates[41], 0.0, neutral_axis)
    booms.append(boom41)
    boom42 = boom.Boom(42, coordinates[42], 0.0, neutral_axis)
    booms.append(boom42)


    # CREATE EDGES INSTANCES AND PUT THEM IN A LIST
    edge_list = []
    edge10 = edges.Edge([1, 0], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge10)
    edge21 = edges.Edge([2, 1], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge21)
    edge32 = edges.Edge([3, 2], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge32)
    edge43 = edges.Edge([4, 3], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge43)
```

```
    edge54 = edges.Edge([5, 4], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge54)
    edge65 = edges.Edge([6, 5], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge65)
    edge76 = edges.Edge([7, 6], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge76)
    edge87 = edges.Edge([8, 7], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge87)
    edge98 = edges.Edge([9, 8], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge98)
    edge109 = edges.Edge([10, 9], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge109)
    edge1110 = edges.Edge([11, 10], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge1110)
    edge1211 = edges.Edge([12, 11], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge1211)
    edge1312 = edges.Edge([13, 12], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge1312)
    edge1413 = edges.Edge([14, 13], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge1413)
    edge1514 = edges.Edge([15, 14], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge1514)

    # booms on semicircle
    edge1615 = edges.Edge([16, 15], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge1615)
    edge1716 = edges.Edge([17, 16], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge1716)
    edge1817 = edges.Edge([18, 17], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge1817)
    edge1918 = edges.Edge([19, 18], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge1918)
    edge2019 = edges.Edge([20, 19], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge2019)
    edge2120 = edges.Edge([21, 20], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge2120)
    edge2221 = edges.Edge([22, 21], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge2221)
    edge2322 = edges.Edge([23, 22], 1.1*10**(-3), 44.179*10**(-3))
    edge_list.append(edge2322)

    # booms on lower spar
    edge2423 = edges.Edge([24, 23], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge2423)
    edge2524 = edges.Edge([25, 24], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge2524)
    edge2625 = edges.Edge([26, 25], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge2625)
    edge2726 = edges.Edge([27, 26], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge2726)
    edge2827 = edges.Edge([28, 27], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge2827)
    edge2928 = edges.Edge([29, 28], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge2928)
    edge3029 = edges.Edge([30, 29], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3029)
    edge3130 = edges.Edge([31, 30], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3130)
```

```
    edge3231 = edges.Edge([32, 31], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3231)
    edge3332 = edges.Edge([33, 32], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3332)
    edge3433 = edges.Edge([34, 33], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3433)
    edge3534 = edges.Edge([35, 34], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3534)
    edge3635 = edges.Edge([36, 35], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3635)
    edge3736 = edges.Edge([37, 36], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3736)
    edge3837 = edges.Edge([38, 37], 1.1*10**(-3), 56.103 * 0.5*10**(-3))
    edge_list.append(edge3837)
    edge038 = edges.Edge([0, 38], 1.1*10**(-3), 56.103*10**(-3))
    edge_list.append(edge038)

    # booms on the vertical spar
    edge3915 = edges.Edge([39, 15], 2.9*10**(-3), 45*10**(-3))
    edge_list.append(edge3915)
    edge4039 = edges.Edge([40, 39], 2.9*10**(-3), 45*10**(-3))
    edge_list.append(edge4039)
    edge4140 = edges.Edge([41, 40], 2.9*10**(-3), 45*10**(-3))
    edge_list.append(edge4140)
    edge4241 = edges.Edge([42, 41], 2.9*10**(-3), 45*10**(-3))
    edge_list.append(edge4241)
    edge2342 = edges.Edge([23, 42], 2.9*10**(-3), 45*10**(-3))
    edge_list.append(edge2342)

    # CREATE INSTANCE OF AILERON GEOMETRY WITH ALL THE BOOMS
    aileron_geometry = geometry.Geometry(43, booms, edge_list, [19880.391*10**(-6),
      ↪ 36225*10**(-6)], 28 * 10**9)
    aileron_geometry.construct_geometry()
    aileron_geometry.cells = [[edge038, edge3837, edge3736, edge3635, edge3534, edge3433,
      ↪ edge3332, edge3231, edge3130,
                              edge3029, edge2928, edge2827, edge2726, edge2625, edge2524,
                                ↪ edge2423, edge2342, edge4241,
                              edge4140, edge4039, edge3915, edge1514, edge1413, edge1312,
                                ↪ edge1211, edge1110, edge109,
                              edge98, edge87, edge76, edge65, edge54, edge43, edge32,
                                ↪ edge21, edge10],
                             [edge2019, edge1918, edge1817, edge1716, edge1615, edge3915,
                               ↪ edge4039, edge4140, edge4241,
                              edge2342, edge2322, edge2221, edge2120]]
    # calculate areas of all booms
    for element in booms:
        element.calculate_area(aileron_geometry)

    # insert them in aileron_geometry object
    aileron_geometry.get_areas()

    # calculate centroid position
    aileron_geometry.calc_centroid()
    for boom_element in booms:
        boom_element.calc_y_dist(aileron_geometry)
        boom_element.calc_z_dist(aileron_geometry)

    # calculate moments of inertia
```

```python
    aileron_geometry.moment_inertia_Izz()
    aileron_geometry.moment_inertia_Iyy()

    # PLOT AND PRINT GEOMETRICAL PROPERTIES FOR VERIFICATION
    aileron_geometry.plot_edges()
    for it, el in enumerate(booms):
            print('area of boom ', it, ' : ', aileron_geometry.boom_areas[it], '[mm^2]')
    print('centroid position : ', aileron_geometry.centroid)
    print('z moment of inertia : ', aileron_geometry.Izz, ' [mm^4]')
    print('y moment of inertia : ', aileron_geometry.Iyy, ' [mm^4]')
    print('zy moment of inertia : ', aileron_geometry.Izy, '[mm^4]')

    # GET THE LIST OF FORCES AND MOMENTS
    file_name = "Loads.txt"
    x_i_array = helpers.get_array_x_i(file_name)
    Mx_array = helpers.get_array_Mx_i(file_name)
    My_array = helpers.get_array_My_i(file_name)
    Mz_array = helpers.get_array_Mz_i(file_name)
    Sz_array = helpers.get_array_Sz_i(file_name)
    Sy_array = helpers.get_array_Sy_i(file_name)

    # create a matrix of stresses
    stress_matrix = np.zeros((43, 101))
    for j, location in enumerate(x_i_array):
        for i, boom_member in enumerate(aileron_geometry.booms):
            boom_member.calc_bending_stress(Mz_array[j], My_array[j], aileron_geometry)
            stress_matrix[i][j] = boom_member.bending_stress

    # find maximum stress
    max_stress_matrix = np.amax(stress_matrix, axis=1)
    # set up matrix of shear stresses
    stress_matrix_shear = np.zeros((len(aileron_geometry.edges), 101))
    # set up lists
    twist_rate_list = []
    thetas_list = []
    section_numbers = 100
    step = 2.771 / section_numbers
    thetas_list.append(0.453786)
    file = open("thetas_list.txt", "w")

    for i, x_i in enumerate(x_i_array):
        # create new instance of section with new location
        aileron_section = DiscreteSection.DiscreteSection(neutral_axis, aileron_geometry)
        # calculate shear flows due to pure shear and torque
        aileron_section.calc_total_shear_flow(Sz_array[i], Sy_array[i], Mx_array[i],
            ↪ edge2342)
        # calculate shear stress due to total shear flows and insert in the shear stress
            ↪ matrix
        aileron_section.calc_shear_stress()
        for n1, edge_ex in enumerate(aileron_geometry.edges):
            stress_matrix_shear[n1][i] = edge_ex.shear_stress
        # append the twist rate (computed at the same time as torque shear flow) in the
            ↪ twist rate list
        twist_rate_list.append(aileron_section.twist_rate)
        # calculate theta with finite differences, append to the list and copy to the txt
            ↪ file
        theta = twist_rate_list[i - 1] * step + thetas_list[i - 1]
        thetas_list.append(theta)
```

```
            file.write(str(float(theta)) + '\n')

        # find the maximum shear stress on each rib
        print('the maximum shear stress in rib A : ', np.max(stress_matrix_shear[:, 97]))
        print('the maximum shear stress in rib B : ', np.max(stress_matrix_shear[:, 51]))
        print('the maximum shear stress in rib C : ', np.max(stress_matrix_shear[:, 41]))
        print('the maximum shear stress in rib D : ', np.max(stress_matrix_shear[:, 18]))
        # find the maximum normal stress on each rib
        print('the maximum normal stress in rib A : ', np.max(stress_matrix[:, 97]))
        print('the maximum normal stress in rib A : ', np.max(stress_matrix[:, 51]))
        print('the maximum normal stress in rib A : ', np.max(stress_matrix[:, 41]))
        print('the maximum normal stress in rib A : ', np.max(stress_matrix[:, 18]))

initialise_problem()
```

```
import unittest
import geometry
import helpers
import boom
import edges
import numpy as np
import DiscreteSection

class TestGeometry(unittest.TestCase):
    def test_inertia0(self):
        # test moment of inertia calculation comparing it to results of example 20.2 in
          ↪ Megson
        # IMPORTANT: this only passes the test if on the moment of inertia calculator the
          ↪ line where the list of
        # distances is calculated is COMMENTED OUT. This is because the list of distances
          ↪ that should be used is given
        # directly on the example so they should NOT be recalculated.
        example_20_2 = geometry.Geometry(16, [0], [], [], 0.0)
        example_20_2.boom_areas = [640, 600, 600, 600, 620, 640, 640, 850, 640, 600, 600,
          ↪ 600, 620, 640, 640, 850]
        example_20_2.y_dists = np.array([660, 600, 420, 228, 25, -204, -396, -502, -540,
          ↪ 600, 420, 228, 25, -204, -396, -502])
        example_20_2.centroid = (0, 0)
        example_20_2.moment_inertia_Izz()
        # 1 is a good enough error because in Megson they round the contribution of each
          ↪ boom, so they accumulate error
        # from the 16 components
        self.assertTrue(abs(example_20_2.Izz*10**(-6) - 1854) < 1)

    def test_areas_centroid_inertia(self):
        # following the example on slide 43 of
          ↪ https://www.slideshare.net/scemd3/lec6aircraft-structural-idealisation-1
        # set up boom architecture
        neutral_axis = (0, 1, 0)
        boom0 = boom.Boom(0, [-250, 150], 1000, neutral_axis)
        boom1 = boom.Boom(1, [250, 150], 640, neutral_axis)
        boom2 = boom.Boom(2, [250, -150], 640, neutral_axis)
        boom3 = boom.Boom(3, [-250, -150], 1000, neutral_axis)
        edge01 = edges.Edge([0, 1], 10, 500)
        edge03 = edges.Edge([0, 3], 10, 300)
```

```
        edge12 = edges.Edge([1, 2], 8, 300)
        edge23 = edges.Edge([2, 3], 10, 500)

        # calculate area for each boom
        example_43 = geometry.Geometry(4, [boom0, boom1, boom2, boom3], [edge01, edge03, edge12, edge23], [0.0], 0.)
        example_43.construct_geometry()
        for element in [boom0, boom1, boom2, boom3]:
            element.calculate_area(example_43)

        # test the boom areas
        example_43.get_areas()
        self.assertTrue(abs(example_43.boom_areas[0] - example_43.boom_areas[3]) < 0.01 and
                        abs(example_43.boom_areas[0] - 4000) < 0.01)
        self.assertTrue(abs(example_43.boom_areas[1] - example_43.boom_areas[1]) < 0.01 and
                        abs(example_43.boom_areas[1] - 3540) < 0.01)

        # test centroid
        example_43.calc_centroid()
        self.assertTrue(abs(abs(example_43.centroid[0]) - 15.25) < 0.1)
        self.assertTrue(abs(abs(example_43.centroid[1]) - 0.0) < 0.1)

        # test moments of inertia
        example_43.moment_inertia_Izz()
        example_43.moment_inertia_Iyy()
        self.assertTrue(abs(example_43.Izz - 339300000) < 1)
        self.assertTrue(abs(example_43.Iyy - 938992042.5) < 1)
        self.assertTrue(abs(example_43.Izy) < 1)


    def test_boom_areas(self):
        # problem 20.1 taken from Megson. Solution is given on the book
        neutral_axis = (0, 1, 0)
        # set up boom architecture
        boom0 = boom.Boom(0, [0, 150], 1000, neutral_axis)
        boom1 = boom.Boom(1, [500, 150], 50 * 8 + 30 * 8, neutral_axis)
        boom2 = boom.Boom(2, [500, -150], 50 * 8 + 30 * 8, neutral_axis)
        boom3 = boom.Boom(3, [0, -150], 1000, neutral_axis)
        edge01 = edges.Edge([0, 1], 10, 500)
        edge03 = edges.Edge([0, 3], 10, 300)
        edge12 = edges.Edge([1, 2], 8, 300)
        edge23 = edges.Edge([2, 3], 10, 500)

        booms = [boom0, boom1, boom2, boom3]
        edge_list = [edge01, edge03, edge12, edge23]
        problem_20_1 = geometry.Geometry(4, booms, edge_list, [0.], 0.)
        problem_20_1.construct_geometry()

        # calculate boom area for each boom
        for element in booms:
            element.calculate_area(problem_20_1)
        self.assertTrue(abs(boom0.area - 4000) < 1)
        self.assertTrue(abs(boom0.area - boom3.area) < 0.01)
        self.assertTrue(abs(boom1.area - 3540) < 1)
        self.assertTrue(abs(boom1.area - boom2.area) < 0.01)

    def test_shear_flow_pure_shear0(self):
        # following the problem 23.6 in Megson
```

```
        # set up booms and edges
        neutral_axis = (0, 1, 0)
        boom0 = boom.Boom(0, [1092, 153], 0.0, neutral_axis)
        boom1 = boom.Boom(1, [736, 153], 0.0, neutral_axis)
        boom2 = boom.Boom(2, [380, 153], 0.0, neutral_axis)
        boom3 = boom.Boom(3, [0, 153], 0.0, neutral_axis)
        boom4 = boom.Boom(4, [0, -153], 0.0, neutral_axis)
        boom5 = boom.Boom(5, [380, -153], 0.0, neutral_axis)
        boom6 = boom.Boom(6, [736, -153], 0.0, neutral_axis)
        boom7 = boom.Boom(7, [1092, -153], 0.0, neutral_axis)
        boom_list = [boom0, boom1, boom2, boom3, boom4, boom5, boom6, boom7]
        edge10 = edges.Edge([1, 0], 0.915, 356)
        edge07 = edges.Edge([0, 7], 1.250, 306)
        edge21 = edges.Edge([2, 1], 0.915, 356)
        edge32 = edges.Edge([3, 2], 0.783, 380)
        edge52 = edges.Edge([5, 2], 1.250, 306)
        edge34 = edges.Edge([3, 4], 1.250, 610)
        edge54 = edges.Edge([5, 4], 0.783, 380)
        edge65 = edges.Edge([6, 5], 0.915, 356)
        edge76 = edges.Edge([7, 6], 0.915, 356)
        edge_list = [edge52, edge21, edge10, edge07, edge76, edge65, edge32, edge34, edge54]
        problem_23_6 = geometry.Geometry(8, boom_list, edge_list, [217872, 167780],
          ↪  24.2*10**9)
        problem_23_6.cells = [[edge10, edge07, edge76, edge65, edge52, edge21], [edge34,
          ↪  edge32, edge52, edge54]]
        boom0.area = 1290
        boom1.area = 645
        boom2.area = 1290
        boom3.area = 645
        boom4.area = 645
        boom5.area = 1290
        boom6.area = 645
        boom7.area = 1290
        # calculate geometrical properties
        problem_23_6.get_areas()
        problem_23_6.construct_geometry()
        problem_23_6.calc_centroid()
        problem_23_6.moment_inertia_Iyy()
        problem_23_6.moment_inertia_Izz()
        for boom_element in boom_list:
            boom_element.calc_y_dist(problem_23_6)
            boom_element.calc_z_dist(problem_23_6)
        # test moment of inertia
        self.assertTrue(abs(problem_23_6.Izz * 10**(-6) - 181.2) < 1)
        # calculate shear flow due to shear forces
        problem_23_6_section = DiscreteSection.DiscreteSection(neutral_axis, problem_23_6)
        problem_23_6_section.calc_shear_flow_q_B(0, 66750, edge52)
        problem_23_6_section.calc_closed_section_pure_shear_flow_q_0(edge52)
        # test open section shear flow
        self.assertTrue(abs(edge10.q_B - 0.0) < 1)
        self.assertTrue(abs(edge07.q_B - (-72.6)) < 1)
        self.assertTrue(abs(edge32.q_B - (-36.2)) < 1)
        self.assertTrue(abs(edge21.q_B - 36.2) < 1)
        self.assertTrue(abs(edge34.q_B) < 0.1)
        self.assertTrue(abs(edge54.q_B - (-36.3)) < 1)
        self.assertTrue(abs(edge52.q_B - 145.3) < 1)
        self.assertTrue(abs(edge65.q_B - 36.3) < 1)
        self.assertTrue(abs(edge76.q_B) < 1)
```

```
        # test closed section shear flow
        self.assertTrue(abs(edge21.q_0 - (-39.2)) < 1 and abs(edge10.q_0 - (-39.2)) < 1 and abs(edge07.q_0 - (-39.2))
                        < 1 and abs(edge76.q_0 - (-39.2)) < 1 and abs(edge65.q_0 - (-39.2)) < 1)
        self.assertTrue(abs(edge32.q_0 - 17.8) < 1 and abs(edge34.q_0 - 17.8) < 1 and abs(edge54.q_0 - 17.8) < 1)
        self.assertTrue(abs(edge52.q_0 - (-57)) < 1)

    def test_shear_flow_pure_shear1(self):
        # using problem 23.5 in Megson. Some sign conventions are switched for consistency.
        # initialise booms
        neutral_axis = (0, 1, 0)
        boom0 = boom.Boom(0, [-635, -127], 0.0, neutral_axis)
        boom1 = boom.Boom(1, [0, -203], 0.0, neutral_axis)
        boom2 = boom.Boom(2, [763, -101], 0.0, neutral_axis)
        boom3 = boom.Boom(3, [763, 101], 0.0, neutral_axis)
        boom4 = boom.Boom(4, [0, 203], 0.0, neutral_axis)
        boom5 = boom.Boom(5, [-635, 127], 0.0, neutral_axis)
        boom_list = [boom0, boom1, boom2, boom3, boom4, boom5]
        # initialise edges
        edge45 = edges.Edge([4, 5], 0.915, 647)
        edge14 = edges.Edge([1, 4], 2.032, 406)
        edge10 = edges.Edge([1, 0], 0.915, 647)
        edge05 = edges.Edge([0, 5], 1.625, 254)
        edge43 = edges.Edge([4, 3], 0.559, 775)
        edge32 = edges.Edge([3, 2], 1.220, 202)
        edge21 = edges.Edge([2, 1], 0.559, 775)
        edge_list = [edge45, edge14, edge10, edge05, edge43, edge32, edge21]
        # initialise Geometry
        problem_23_5 = geometry.Geometry(6, boom_list, edge_list, [232000, 258000], 1.0)
        problem_23_5.cells = [[edge43, edge32, edge21, edge14], [edge45, edge14, edge10, edge05]]
        # set boom areas to given values
        boom0.area = 1290
        boom1.area = 1936
        boom2.area = 645
        boom3.area = 645
        boom4.area = 1936
        boom5.area = 1290
        problem_23_5.get_areas()

        # calculate and verify geometrical properties
        problem_23_5.construct_geometry()
        problem_23_5.calc_centroid()
        problem_23_5.moment_inertia_Iyy()
        problem_23_5.moment_inertia_Izz()
        self.assertTrue(abs(problem_23_5.Izy < 1))
        self.assertTrue(abs(problem_23_5.Izz * 10**(-6) - 214.3) < 1)
        for boom_element in boom_list:
            boom_element.calc_y_dist(problem_23_5)
            boom_element.calc_z_dist(problem_23_5)

        # calculate shear flows
        problem_23_5_section = DiscreteSection.DiscreteSection(neutral_axis, problem_23_5)
        problem_23_5_section.calc_shear_flow_q_B(0, 44500, edge14)
        # test open section shear flows
        self.assertTrue(abs(edge45.q_B) < 0.1 and abs(edge21.q_B) < 0.1)
```

```
        self.assertTrue(abs(edge43.q_B) < 0.1 and abs(edge10.q_B) < 0.1)
        self.assertTrue(abs(edge32.q_B - (-13.6)) < 1)
        self.assertTrue(abs(edge14.q_B) - 81.7 < 1)
        self.assertTrue(abs(edge05.q_B) - 34.07 < 1)

        # calculate closed section shear flows
        problem_23_5_section.calc_closed_section_pure_shear_flow_q_0(edge14)
        # test open section shear flows
        self.assertTrue(abs(edge45.q_0 - 4.12) < 1 and abs(edge05.q_0 - 4.12) < 1 and abs(edge10.q_0 - 4.12) < 1)
        self.assertTrue(abs(edge43.q_0 - (-5.74)) < 1 and abs(edge21.q_0 - (-5.74)) < 1 and abs(edge32.q_0 - (-5.74)) < 1)
        self.assertTrue(abs(edge14.q_0 - (-9.85)) < 1)

    def test_helper(self):
        self.assertTrue(abs(helpers.distance((-4, 6.5), (-7, 17)) - 10.920164833920778) < 0.001)
        self.assertTrue(abs(helpers.distance((50.67, -4.006), (-3.345, 36.98)) - 67.80466371128169) < 0.001)

    def test_helpers_point_line(self):
        self.assertTrue(abs(helpers.distance_point_line((5, 6), (-2, 3, 4)) - 3.328) < 0.001)
        self.assertTrue(abs(helpers.distance_point_line((-3, 7), (6, -5, 10)) - 5.506) < 0.001)

    def test_boom_normal_stress(self):
        # taken from http://www.ltas-cm3.ulg.ac.be/MECA0028-1/StructAeroAircraftComp.pdf
        # exercise on slide 26
        # set up boom list and areas
        boom_list = []
        neutral_axis = (0, 1, 0)
        boom0 = boom.Boom(0, [300, - 600], 0.0, neutral_axis)
        boom_list.append(boom0)
        boom0.area = 900
        boom1 = boom.Boom(1, [300, 0], 0.0, neutral_axis)
        boom_list.append(boom1)
        boom1.area = 1200
        boom2 = boom.Boom(2, [300, 600], 0.0, neutral_axis)
        boom_list.append(boom2)
        boom2.area = 900
        boom3 = boom.Boom(3, [- 300, 600], 0.0, neutral_axis)
        boom_list.append(boom3)
        boom3.area = 900
        boom4 = boom.Boom(4, [- 300, 0], 0.0, neutral_axis)
        boom_list.append(boom4)
        boom4.area = 1200
        boom5 = boom.Boom(5, [- 300, - 600], 0.0, neutral_axis)
        boom_list.append(boom5)
        boom5.area = 900

        # initialise Geometry instance anc calculate geometrical properties
        example_liege = geometry.Geometry(6, boom_list, [], [], 0.0)
        example_liege.get_areas()
        example_liege.calc_centroid()
        example_liege.moment_inertia_Iyy()
        example_liege.moment_inertia_Izz()

```

```
        # calculate and test normal stresses
        for element in example_liege.booms:
            element.calc_z_dist(example_liege)
            element.calc_y_dist(example_liege)
            element.calc_bending_stress(0, -200*10**2, example_liege)
            self.assertTrue(abs(abs(element.bending_stress) - 0.0111) < 0.01)

    def test_shear_flow_pure_torsion(self):
        # this problem is a generic pure torsion problem
        # initialise booms
        neutral_axis = (0, 1, 0)
        boom_list = []
        boom0 = boom.Boom(0, [900, 250], 0.0, neutral_axis)
        boom_list.append(boom0)
        boom1 = boom.Boom(1, [900, -250], 0.0, neutral_axis)
        boom_list.append(boom1)
        boom2 = boom.Boom(2, [400, -250], 0.0, neutral_axis)
        boom_list.append(boom2)
        boom3 = boom.Boom(3, [0, -250], 0.0, neutral_axis)
        boom_list.append(boom3)
        boom4 = boom.Boom(4, [0, 250], 0.0, neutral_axis)
        boom_list.append(boom4)
        boom5 = boom.Boom(5, [400, 250], 0.0, neutral_axis)
        boom_list.append(boom5)

        # initialise edges
        edge_list = []
        edge01 = edges.Edge([0, 1], 4, 500)
        edge_list.append(edge01)
        edge12 = edges.Edge([1, 2], 4, 500)
        edge_list.append(edge12)
        edge23 = edges.Edge([2, 3], 2, 400)
        edge_list.append(edge23)
        edge34 = edges.Edge([3, 4], 2, 500)
        edge_list.append(edge34)
        edge45 = edges.Edge([4, 5], 2, 400)
        edge_list.append(edge45)
        edge50 = edges.Edge([5, 0], 4, 500)
        edge_list.append(edge50)
        edge52 = edges.Edge([5, 2], 3, 500)
        edge_list.append(edge52)

        # initialise geometry
        problem_torsion = geometry.Geometry(6, boom_list, edge_list, [400*500, 500**2], 27 *
          ↪  10**3)
        problem_torsion.cells = [[edge50, edge01, edge12, edge52], [edge45, edge52, edge34,
          ↪  edge23]]
        problem_torsion.construct_geometry()

        # calculate torsion shear flows
        problem_torsion_section = DiscreteSection.DiscreteSection(neutral_axis,
          ↪  problem_torsion)
        problem_torsion_section.calc_torsion_shear_flow(2.0329 * 10**9, edge52)

        # verify twist rate
        self.assertTrue(abs(problem_torsion_section.twist_rate * 10**5 - 8.73) < 1)

if __name__ == '__main__':
```

```
    tester = TestGeometry()
  #  tester.test_inertia0()  this is commented out for the reasons explained in its
    ↪  definition
    tester.test_areas_centroid_inertia()
    tester.test_boom_areas()
    tester.test_shear_flow_pure_shear0()
    tester.test_shear_flow_pure_shear1()
    tester.test_boom_normal_stress()
    tester.test_shear_flow_pure_torsion()
```

```
import math
import matplotlib.pyplot as plt
import boom
import geometry
import numpy as np

def distance(point1, point2):
    z_1, y_1 = point1[0], point1[1]
    z_2, y_2 = point2[0], point2[1]
    return math.sqrt((y_1 - y_2)**2 + (z_1 - z_2)**2)

def distance_point_line(point, line):
    """
    :param point: tuple (z, y) containing point coordinates
    :param line: tuple (A, B, C) containing line such that Az + By + C = 0
    :return: euclidean distance between line and point
    """
    z, y = point[0], point[1]
    A, B, C = line[0], line[1], line[2]
    return abs(A * z + B * y + C) / math.sqrt(A ** 2 + B ** 2)

def plot_boom_coordinates(coordinates):
    """
    Plot coordinates to verify visually that they are correct.
    :param coordinates: list of lists [z, y] containing the coordinates of each boom.
    """
    zs = []
    ys = []
    n = range(len(coordinates))
    for boom_coord in coordinates:
        zs.append(boom_coord[0])
        ys.append(boom_coord[1])
    fig, ax = plt.subplots()
    ax.scatter(zs, ys)
    plt.axhline(0, color='black')
    for i, txt in enumerate(n):
        ax.annotate(txt, (zs[i], ys[i]))
    plt.show()


def get_list(booms, parameter_position):
    list = []
    for element in booms:
        num = len(element.adjacents)
        adjacent_booms = []
```

```
            for num in range(num):
                adjacent_booms.append(element.adjacents[num][parameter_position])
            list.append(adjacent_booms)
        return list

    def get_thickness_list(booms):
        get_list(booms, 1)

    def get_adjacents_list(booms):
        get_list(booms, 0)

    def get_lengths_list(booms):
        get_list(booms, 2)

    def get_common_wall(cells):
        for wall1 in cells[0]:
            if wall1 in cells[1]:
                return wall1

    def get_array_x_i(file_name):
        data = np.genfromtxt(file_name, skip_header=1)[:, 0]
        return data
    def get_array_Mx_i(file_name):
        data = np.genfromtxt(file_name, skip_header=1)[:, 1]
        return data
    def get_array_My_i(file_name):
        data = np.genfromtxt(file_name, skip_header=1)[:, 2]
        return data
    def get_array_Mz_i(file_name):
        data = np.genfromtxt(file_name, skip_header=1)[:, 3]
        return data
    def get_array_Sy_i(file_name):
        data = np.genfromtxt(file_name, skip_header=1)[:, 4]
        return data
    def get_array_Sz_i(file_name):
        data = np.genfromtxt(file_name, skip_header=1)[:, 5]
        return data
```

```
%Leading and trailing edge deformation
fileID = fopen('thetas_list.txt','r');
theta = [fscanf(fileID,'%f')]';
fclose(fileID);
la=2.771;
steps=1000;
step_size=la/steps;
Ca=0.547;
z4=Ca*0.25;
zh=ha/2;
yle=yplot-zh*sin(theta);
yte=yplot+sin(theta)*(Ca-zh);
zte=zplot+cos(theta)*(Ca-zh);
zle=zplot-cos(theta)*(zh);

figure(1);
ax6 = subplot(1,1,1);
```

```
plot(ax6,t,yte);title(ax6,'Deflections in
    y');xlabel('x[m]');ylabel('[m]');ylim([-0.1,0.32]);
hold on
plot(t,yplot,'-');
plot(t,yle);
legend('TE','Hinge line','LE');

figure(2);
ax7 = subplot(1,1,1);
plot(ax7,t,zte);title(ax7,'Deflections in
    z');xlabel('x[m]');ylabel('[m]');;ylim([-0.2,0.42])
hold on
plot(t,zplot);
plot(t,zle);
legend('TE','Hinge line','LE');
```