POLITECNICO
DI TORINO

POLITECNICO DI TORINO
DEPARTMENT OF AEROSPACE ENGINEERING
THESIS

NONLINEAR BEAM OPTIMIZATION TOOL FOR INTEGRATION INTO A GRADIENT-BASED AERO-STRUCTURAL OPTIMIZATION FRAMEWORK

Marco Tramontano

TORINO, TO
2020

POLITECNICO
DI TORINO

POLITECNICO DI TORINO
DEPARTMENT OF AEROSPACE ENGINEERING
THESIS

NONLINEAR BEAM OPTIMIZATION TOOL FOR INTEGRATION INTO A GRADIENT-BASED AERO-STRUCTURAL OPTIMIZATION FRAMEWORK

Marco Tramontano

PoliTo Supervisor:
Prof. Erasmo Carrera

UC3M Supervisor:
Prof.Rauno Cavallaro

UC3M Co-supervisor:
Rocco Bombardieri

TORINO, TO
2020

To all my family

Without them non of this would have been possible

Contents

List of Figures

List of Tables

Listings

Abstract

The study conducted in this thesis regarded the development and extension of a tool for structural sizing and optimization of slender wings featuring the classic wing box reinforced with stiffeners.

The tool is composed of two main module. The core module, in-house developed in C++, implements a geometrically nonlinear finite element solver based on Euler-Bernoulli beams. This module was already employed as structural solver within a framework performing outer mold line optimization of flexible wings. Within this thesis, a preliminary capability for evaluation of the integrity of the structure has been developed, thanks to a pre-processing layer connecting the wing box geometry, that can be defined by the user, to the inertial properties of the section, and a post-processing layer enabling weight of the structure, stress retrieving and buckling assessment of the reinforced panel; in such layer it also possible to aggregate stresses or buckling with the Kreisselmeier-Steinhauser function in order to describe the integrity state of the beam with a reduced number of responses. The whole capability has been developed considering integration with the algorithmic differentiation library CoDiPack, for evaluating the sensitivities of the responses with respect to the design variables, i.e., wing box geometric parameters.

An external module, written in Python, wraps the core solver in order to promote the easy management of inputs and outputs; moreover, it provides an interface for allowing calls to the basic functions called within a gradient-based optimization process, such as, call to evaluate responses (objective function and constraints) and their sensitivities.

Capabilities of the tool have been then demonstrated, performing, within the OpenMDAO, an open-source Multidisciplinary design optimization (MDO) framework, several structural optimizations. Validity of the optimization has been verified for some cases checking for the Karush-Kuhn-Tucker condition with the support of Matlab symbolic tool.

The tool developed within this thesis has proven to be suitable for integration into the original algorithmic-differentiation based coupled aero-structural optimization framework, extending its original capability, limited to aerodynamic objective functions and constraints, to include also structural design variables, objective functions and constraints.

Keywords : *Structural optimization Open-source framework PyBeam Kreisselmeier-Steinhauser Karush–Kuhn–Tucker OpenMDAO*

Chapter 1

Introduction

Currently, the main tendency in multidisciplinary design optimization (MDO) , is to hold preliminary studies using high-fidelity prediction tool. MDO is a branch of engineering focused on the numerical optimization , involving different disciplines or subsystems.Fluid Structure interaction (FSI) draws attention, cause of its complexity and utility. Aeroelasticity in particular , involves the concept of flexible structure and the numerical study of this subject is to be considered in the development of an Aeronautical project; suffice it to say that all aircraft must be certified under Airworthiness regulations in order make sure that they are safe from aeroelastic instability and do not exceed design loads throughout the entire desired flight envelope. The whole certification process , includes numerical simulations validated by experimental test such as Ground Vibration Test (**GVT**) and Flight Vibration Test (**FVT**). Aeroelastic numerical simulations are really expensive in terms of computational cost , for this reason stick model is used : it is a simplified model of the aircraft made of beam elements (figure 1.1). Moreover, aeroelastic optimization turns to be really important : it consists in finding an earlier structural layout and sizes , avoiding cost of redesign and corrections; therefore, an appropriate sizing of the structure implies an increasing in aircraft performances, in terms of Aerodynamic (minimum drag) and Structure (minimum weight). Main issue of Aeroelasticity is to consider the flexibility of the structure , which results in a complex aero-structure coupling; this effects must be considered from the beginning to the end of design phase.

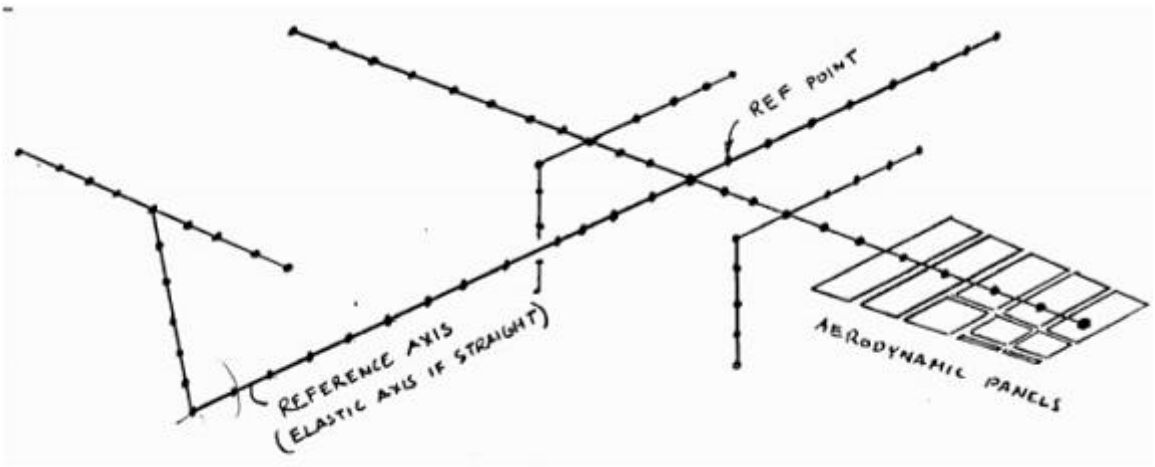


FIGURE 1.1. Example of stick model for aeroelastic analysis

Several research have been conducted about aircraft aeroelastic optimization, and different tool have been developed for Coupled aeroelastic analysis such as FUN3D [5] from NASA,TAU3 [6] from DLR or ElsA4 [7] from ONERA. The main purpose of this thesis , is to develop a tool capable to perform structural design and optimization of beams, exploiting a homemade non linear structural solver, named pyBeam [1], and an open-source framework using properly for the optimization, named openMDAO [3]. Originally , pyBeam was able to calculate the displacements of a flexible structure , specifying in input the inertial properties of the beams that make it up (moments of inertia); doing that, a wing model can be built using longitudinal flexible beams, that state for the wing semi-span, and traversal rigid beams that simulate the airfoils.

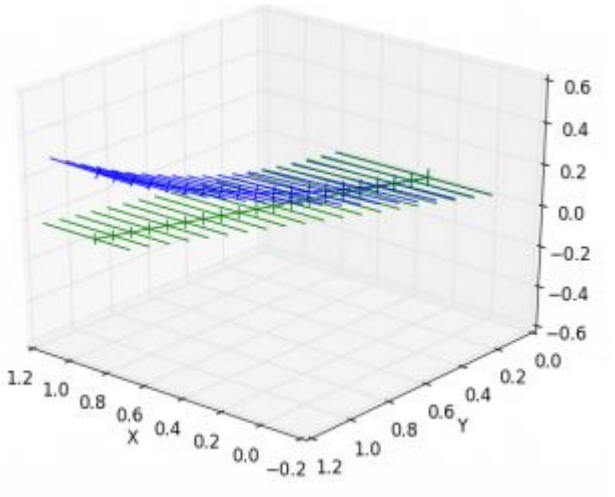


FIGURE 1.2. Stick model. Deformed configuration of a wing , using previous version of PyBeam [1]

PyBeam can be modified to reduce further the complexity of the model : in this way, the wing is represented just by one beam (figure 1.3) which is credited with airfoils core characteristics such as wingbox chord and height , area of stringers and caps , thickness of skin and spar (Chapter 3) ; these parameters represent the actual design variables.

Than other changes have been done to obtain as output the values of the function to minimize, the weight in this case , and the value of constraint must be satisfied ; for the latter , based on the reinforced shell theory, has been applied Von Mises criterion, finally to aggregate all constraints Kreisselmeier-Steinhauser function has been adopted [8].

Once do that , using openMDAO , optimization has been implemented and finally validated by a simple testcase.

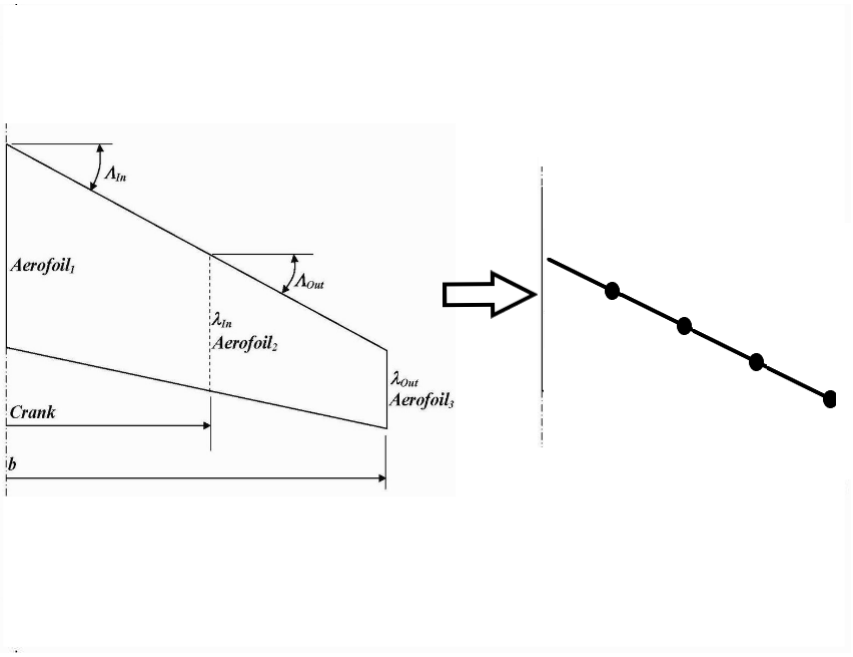


FIGURE 1.3. Transition from wing to a beam

Chapter 2

A beam finite element solver

The aim of this chapter is to give an overview on pyBeam structural solver and its theoretical background. PyBeam is a geometrically nonlinear FE structural solver which has been interfaced to the CFD solver SU2 for the calculation of aerostructural coupled sensitivities, key step for gradient-based optimization of slender wings. PyBeam is an UC3M in-house open source FE solver which employs Algorithmic Differentiation (AD) to solve the adjoint method an evaluate Jacobians and gradients. AD relies on the application of the chain rule of differentiation to each operation in the program flow (chapter 4). The derivatives given by the chain rule can be propagated forward (forward mode) or backwards (reverse mode). This last option is the natural one to solve the adjoint equations and evaluate sensitivities.

PyBeam FE structural solver is based on a 6-dof non linear beam model following the Euler-Bernoulli beam kinematic assumption. The classic solid mechanics statement of the elastostatic problem is:

$$\mathcal{S}(\mathbf{u}) = 0 \longrightarrow \begin{cases} \nabla \cdot \sigma + \mathbf{F_s} = 0 \\ \varepsilon = \varepsilon(\mathbf{u}) \\ \sigma = \mathcal{K} : \varepsilon \end{cases} \tag{2.1}$$

where

- σ = *Cauchy stress tensor*,
- $\mathbf{F_S}$ = *structural body forces per unit volume*,
- ε = *strain tensor*,

- $\mathbf{u}$ = *displacement vector*,
- $\mathcal{K}$ = *stiffness tensor*.

the first equation is the classical forces equilibrium, the second represents the strain-displacement equation featuring the geometrical nonlinearities and the last one the constitutive equation for a linear elastic material.

2.1 Nonlinear analysis

Let the equations of equilibrium be expressed as follows:

$$\mathbf{N^T F} = \mathbf{P} \tag{2.2}$$

where $\mathbf{F}$ are the member forces or stresses and $\mathbf{P}$ are the external applied loads; the operator $\mathbf{N^T}$ describes the system equilibrium. When a perturbation occurs, the system response is describe by

$$\mathbf{dN^T F} + \mathbf{N^T dF} = \mathbf{dP} \tag{2.3}$$

the first term on the left-hand side (LHS), is representative of geometric nonlinearities, the second one returns the linear theory. From equation 2.3 it can be noticed that the action of geometric nonlinearities do not allow the system to reach the equilibrium, obtained just in the deformed configuration. For a discrete system , equation 2.3 can be expressed as:

$$(\mathbf{K_E} + \mathbf{K_G})\delta = \mathbf{dP} \tag{2.4}$$

where

- $\mathbf{K_E}$ = Elastic stiffness Matrix
- $\mathbf{K_G}$ = Geometric stiffness Matrix
- δ = System displacements

PyBeam solver employs beam elements characterized by 6-dof (represented in figure 2.1): every node has three rotational and three translational dof. Hence, for each finite element the generealized nodal displacement vectors (δ) and nodal forces ($\mathbf{P}$) are vectors of 12 components. In the following, a brief overview on geometric nonlinearities is given, and equation 2.4 and its solution discussed.

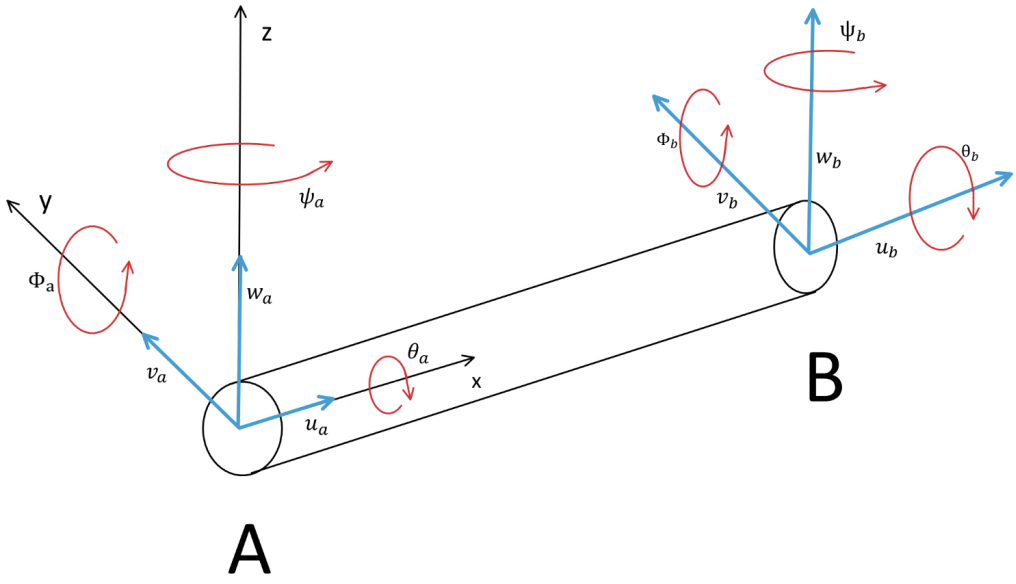


FIGURE 2.1. 3D Beam element.

2.1.1 Geometric nonlinearities

Different types of nonlinearities exist, however, only geometric ones are the ones considered in pyBeam. These nonlinearities are related to large displacements of the deformed structure that invalidate, for example, the paradigm of formulating the equilibrium on the undeformed configuration. It should be pointed out, however, that

even if the displacement are large, the strains are small, hence, the material continues behaving as a linear elastic one.

One aspect relative to large displacement is the need for properly account for large/finite rotations as opposed to infinitesimal rotations. Considering a structural element subjected to a displacement field relative to an infinitesimal rigid rotation, its length does change even though the rotation is rigid. This is not happening with an appropriate description of the displacement fields with a finite rotation. This is clearly shown in figures 2.2 and 2.3, obtained by a linear and nonlinear description of a L-shaped system of two beams, in which a torsional moment is applied on one beam in such a way that the other beam experiences a rigid rotation. It is apparent how, even if unloaded, in the linear case the external beam changes its length. If equilibrium has to be enforced in the actual configuration, such length variation would introduce parasitic forces that affect the whole solution.

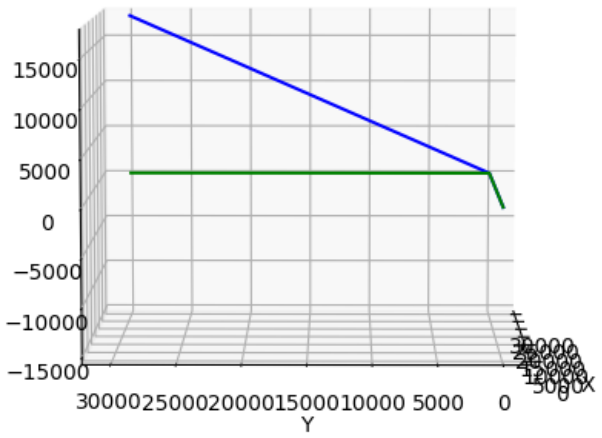


FIGURE 2.2. Torsion spring : linear analysis

A further consequence of geometric nonlinearities is the so called stress stiffening: according to this phenomenon, a prestressed structure increases its stiffness.

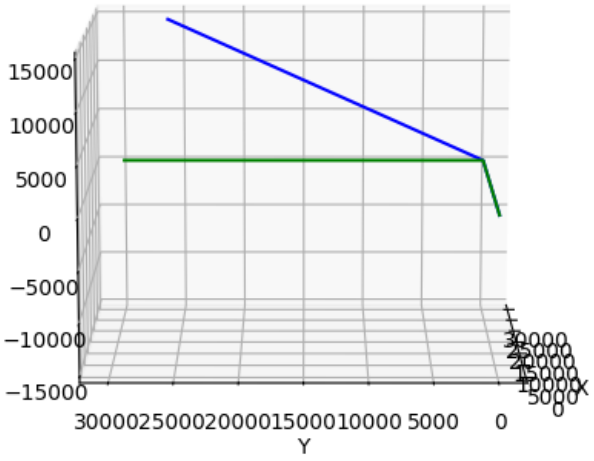


FIGURE 2.3. Torsion spring : nonlinear analysis

Hence, as the structure deforms, the internal stresses contribute in changing the response of the system, something that does not happen for linear structures. More details about geometrical nonlinearities are presented in [9].

After this brief introduction to some phenomenological aspects of geometric nonlinearities, let's go back to the solution of equation 2.4. Next sections will analyze the terms on the LHS of this equation.

2.1.2 Elastic stiffness matrix derivation ($\mathbf{K_e}$)

$\mathbf{K_e}$ is the elastic matrix and can be derived bringing up the linear theory and the "node method" which allows us to analyze the stresses and displacements system of a

member, using the following equations :

$$\mathbf{N^T F = P} \quad \longrightarrow \quad node\ equilibrium \tag{2.5}$$

$$\mathbf{F = K}\delta \quad \longrightarrow \quad constitutive\ equation \tag{2.6}$$

$$\mathbf{\Delta = N}\delta \quad \longrightarrow \quad member\ node\ displacement\ equation \tag{2.7}$$

where $\mathbf{\Delta}$ is the member displacements matrix , σ the joint displacements matrix and K is the primitive stiffness matrix.For a beam, K is a 12 x 12 matrix.
Considering these three equations , the equilibrium can be written as

$$\mathbf{N^T K N}\delta = \mathbf{P} \quad \Rightarrow \quad \mathbf{K_e}\delta = \mathbf{P} \tag{2.8}$$

N is a matrix whose components are unitary vectors that describe the beam slope :

$$N_{ij} = \begin{cases} n_i^T & \Rightarrow \ j\ positive\ end\ of\ i \\ -n_i^T & \Rightarrow \ j\ negative\ end\ of\ i \\ 0 \end{cases} \tag{2.9}$$

where index "i" designs the current element, instead index "j" represents the node.
For a beam element, $\mathbf{n_i}$ is the 3 x 1 column matrix of unit vector components:

$$\mathbf{n_i} = \{(n_i)_x, (n_i)_y, (n_i)_z\}^T \tag{2.10}$$

each component of $\mathbf{n_i}$ depends on the current length of the beam. For the 3D beam in figure 2.1, $\mathbf{n_i}$ is organized in two kinematic matrices N_a and N_b, 6 x 6 elements each.The 12 x 12 primitive stiffness matrix is known from the theory, so the 12 x 12 Elastic stiffness matrix $\mathbf{K_e}$ is :

$$\mathbf{K_e} = \begin{bmatrix} [\mathbf{N_a}]^T \\ [\mathbf{N_b}]^T \end{bmatrix} \big[\mathbf{K}\big] \ \big[[\mathbf{N_a}] \quad [\mathbf{N_b}]\big]$$

2.1.3 Geometric stiffness matrix derivation ($\mathbf{K_g}$)

For the calculation of this matrix has to be looked at the perturbation equation 2.3: as the term $\mathbf{N^T dF} \Rightarrow \mathbf{K_e}\delta$, the component $\mathbf{dN^T F} \Rightarrow \mathbf{K_g}$ matrix.

N matrix is composed by n_i terms that can be written in nodes coordinates :

$$\begin{aligned}(n_i)_x &= \frac{x_b - x_a}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}} = \frac{x_b - x_a}{L_i} \\ (n_i)_y &= \frac{y_b - y_a}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}} = \frac{y_b - y_a}{L_i} \\ (n_i)_z &= \frac{z_b - z_a}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}} = \frac{z_b - z_a}{L_i}\end{aligned} \tag{2.11}$$

with L_i the length of "i" element , which can vary when a perturbation occurs.

Since N is a function of $A \equiv [x_a, y_a, z_a]$ and $B \equiv [x_b, y_b, z_b]$, its gradient is obtained using the chain rule of differentiation:

$$\begin{aligned}d(\mathbf{N}_i^T)F_i = (\nabla \mathbf{N}_i^T \cdot \delta)F_i = \\ \left(\frac{\partial \mathbf{N_i}^T}{\partial x_A}(\delta_a)_x + \frac{\partial \mathbf{N_i}^T}{\partial y_A}(\delta_a)_y + \frac{\partial \mathbf{N_i}^T}{\partial z_A}(\delta_a)_z + \right. \\ \left. + \frac{\partial \mathbf{N_i}^T}{\partial x_b}(\delta_b)_x + \frac{\partial \mathbf{N_i}^T}{\partial y_b}(\delta_b)_y + \frac{\partial \mathbf{N_i}^T}{\partial z_b}(\delta_b)_z \right)\end{aligned} \tag{2.12}$$

The shape of geometric matrix becomes :

$$\mathbf{K_g} = (\nabla \mathbf{N}_i^T)F_i = \begin{bmatrix} [\nabla \mathbf{N_i}]^{AA} & [\nabla \mathbf{N_i}]^{AB} \\ [\nabla \mathbf{N_i}]^{BA} & [\nabla \mathbf{N_i}]^{BB} \end{bmatrix} \cdot F_i$$

Where :

- AA is the variation of normal in A w.r.t. the displacement in A
- AB is the variation of normal in A w.r.t. the displacement in B
- BB is the variation of normal in B w.r.t. the displacement in B
- BA is the variation of normal in B w.r.t. the displacement in A

Knowing the components of $\mathbf{N_i}$, is possible to calculate $\nabla \mathbf{N_i}$ by deriving 2.11 . By following the algebraic steps reported in [9], $\mathbf{K_G}$ is computed :

$$\mathbf{K_g} = \frac{F_i}{L_i} \begin{bmatrix} [\mathbf{I} - \mathbf{n_i}\mathbf{n_i^T}] & -[\mathbf{I} - \mathbf{n_i}\mathbf{n_i^T}] \\ -[\mathbf{I} - \mathbf{n_i}\mathbf{n_i^T}] & [\mathbf{I} - \mathbf{n_i}\mathbf{n_i^T}] \end{bmatrix}$$

Where $\mathbf{n_i}$ components are reported in 2.11.
$\mathbf{K_G}$ is arranged in a 12 x 12 matrix so that is possible to sum it with the elastic matrix $\mathbf{K_e}$.

2.1.4 Solution of perturbation equation

To analyze the final deformed configuration , should be solved iteratively the perturbation equation 2.4 . From the previous section , can be noticed that $\mathbf{K_G}$ depends on $\mathbf{F_i}$; Since $\mathbf{dP} = \mathbf{P} - \mathbf{N^T F_i}$ (no equilibrium in undeformed configuration), expression 2.4 is a non-linear equation.

$$(\mathbf{K_e} + \mathbf{Kg})\delta = \mathbf{P} - \mathbf{N^T F} \tag{2.13}$$

Letting $\mathbf{x} = \mathbf{\Delta}$, $\mathbf{dx} = \delta$ and $\mathbf{dP} = \mathbf{R(x)}$, the iterative process to be applied is the Newton-Raphson method : it consists in finding a value of $\mathbf{x}^*$ that neglect the Residual $\mathbf{R(x)}$, starting from an initial guess. Newton's method of solving a nonlinear system, implies the linearization of the expression $\mathbf{R(x)}$ round $\mathbf{x^n}$

$$\mathbf{R(x^n)} + (\nabla_{\mathbf{x}}\mathbf{R})\Big|_{\mathbf{x^n}} \cdot \mathbf{dx} = 0 \;\; \Rightarrow \;\; \mathbf{dx} = -[\nabla_{\mathbf{x}}\mathbf{R}]^{-1}\Big|_{\mathbf{x^n}} \cdot \mathbf{R(x^n)} \tag{2.14}$$

Where

- $[\nabla_{\mathbf{x}}\mathbf{R(x)}]^{-1}$ = $\mathbf{K_e} + \mathbf{K_G(x)}$ is the Tangent Matrix , whose calculation relies on the evaluation of elastic and geometric stiffness matrices. Actually , when tangent matrix is built for the whole structure , it needs to be rotated,through a rotational matrix $\mathbf{R_{rot}}$, for passing from local to global reference frame.

- $\mathbf{R(x)}$ = $\mathbf{P} - \mathbf{N^T F(x)}$ is the residual which is calculated using the constitutive equation

The iterative process for the nonlinear analysis of a beam , takes the following steps :

1. Give an initial guess $\mathbf{x^n} = \mathbf{x^0}$.
2. Substitute the initial guess in the linearized expression 2.14 and solve for $\mathbf{dx}$
3. update the initial guess with $\mathbf{dx}$ finding the new displacement : $\mathbf{x^1} = \mathbf{x^0} + \mathbf{dx}$
4. With the calculated displacement , evaluate $\mathbf{R}(\mathbf{x^1})$: if Residual becomes zero , it has been reached the stationary , if not return to point 2) using as initial guess $\mathbf{x^1}$

When the process converges , the values of $\mathbf{x}^*$ represent the deformed beam displacements :

$$\mathbf{x}^* = \begin{bmatrix} u_a \\ v_a \\ w_a \\ \theta_a \\ \Psi_a \\ \phi_a \\ u_b \\ v_b \\ w_b \\ \theta_b \\ \Psi_b \\ \phi_b \end{bmatrix} \Rightarrow \mathbf{R}(\mathbf{x}^*) = \mathbf{0}$$

2.2 PyBeam architecture

This section is devoted to investigate the pyBeam architecture. It is an open-source software whose core is coded in C++ but it is wrapped in Python, in order to manage better input files and to favour the interaction with other software, like SU2, for coupled aerostructural analysis: A more in-dept explanation of the interaction between the C++ core and Python wrapping will be given in Chapter 5. In this section is shown just the C++ core structure for the solution of structural nonlinear analysis

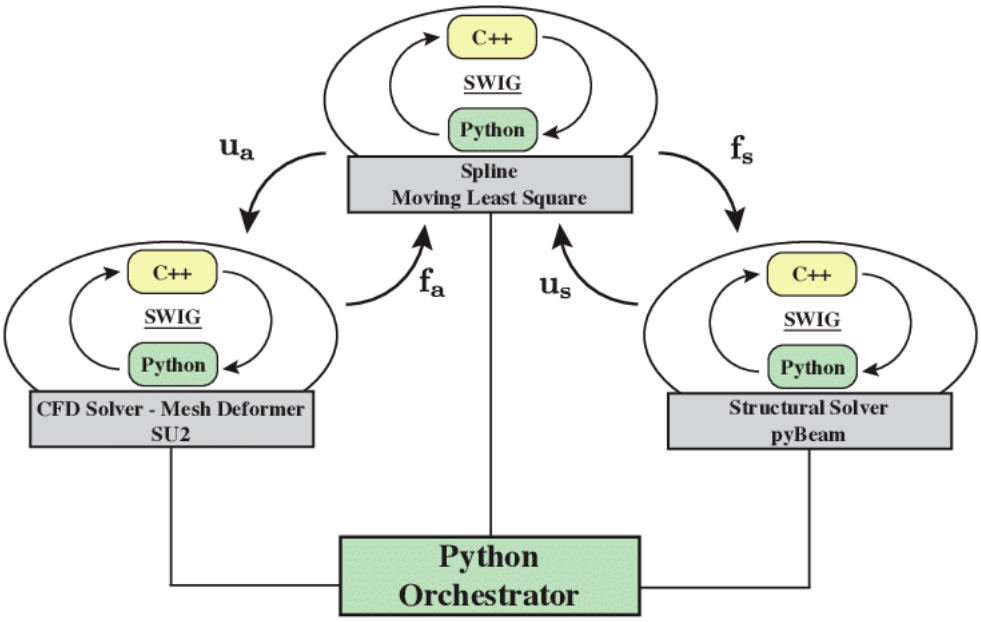


FIGURE 2.4. Aerostructural framework layout [1]

and the calculation of responses that are used as objective function and constraint in the structural optimization problem (Chapter 4).

PyBeam core consists in header files (.h), where typically classes and relative methods (functions) and member variables are declared, and ".cpp" files in which such member variable and functions are defined, see Figure 2.5.

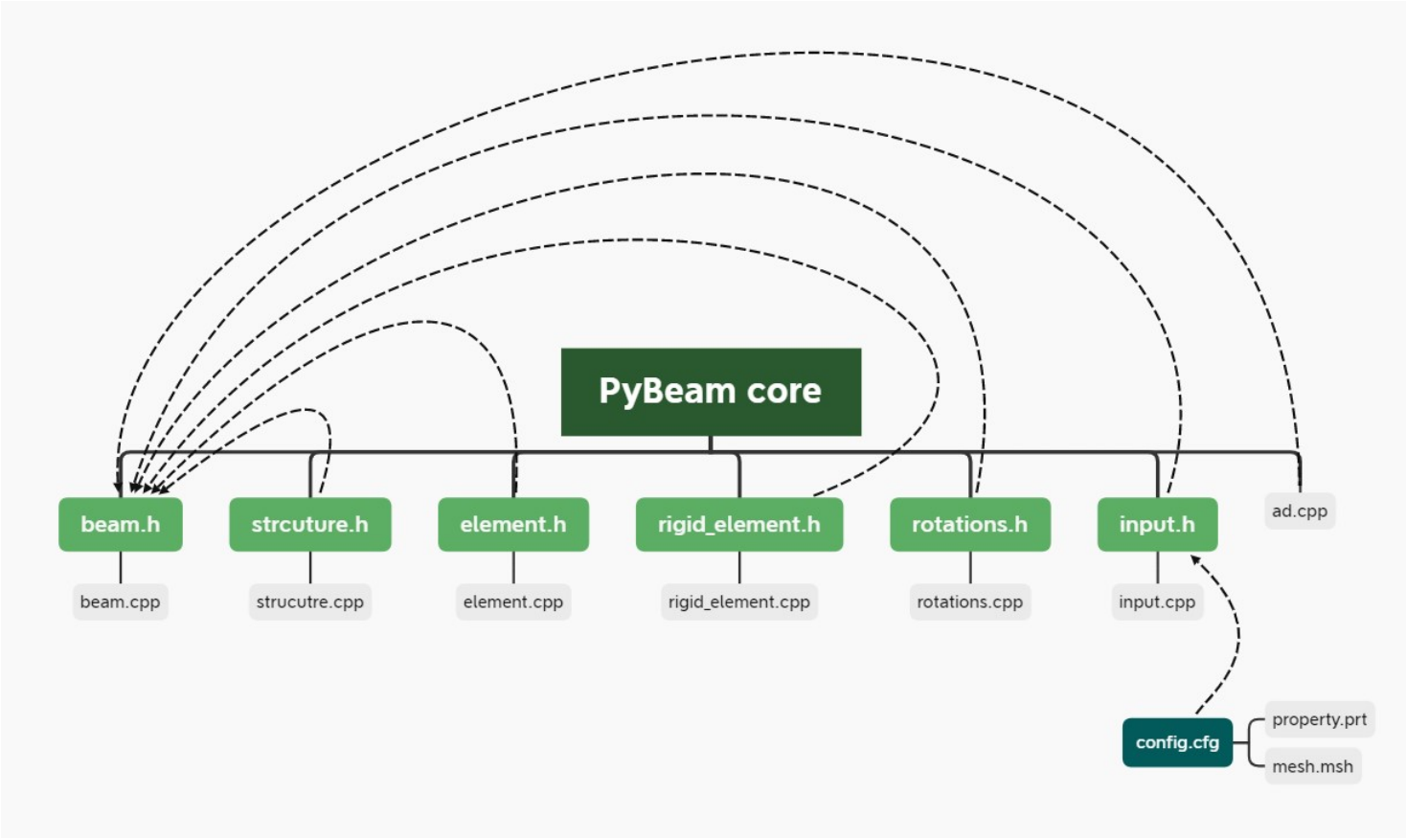


FIGURE 2.5. PyBeam C++ Core architecture

The main classes are listed below.

beam: here are recalled functions from other classes, to be used for the implementation of the actual solver, located in the current class. We also find functions that return values of objective function , constraints and their gradients , useful for the optimization process .

Structure: functions written in this class are devoted to build the global model starting from local components evaluated in other classes. Moreover, are provided each terms necessary for the implementation of Newton method (2.14). For instances, here are evaluated the global stiffness matrix and the the residual and the eq. 2.14 is solved.

element : the features of each finite element are defined in this class. For example,are set the FE length, position and properties; moreover, the FEM matrices are evaluated.

rigid_element: a structure can includes also rigid beams, whose mathematical formulation is reported in the current class.

rotations: in this class is computed the rotational matrix , $\mathbf{R_{rot}}$

input: it reads input provided to pybeam from the outside. External input files are :

- property.prt : here are reported the section properties (design variables),
- mesh.msh : here are specified the number of elements , their length , their property , the constrained degree of freedom and the number of rigid elements,
- config.cfg : here are recalled property and mesh files , moreover are specified several important parameters as the Young Modulus , Density, Poisson Ratio , number of iteration and loadstep (dp) for the nonlinear analysis, the tolerance of the solver and the kind of linear solver used.

ad: here is set the package CODIPACK , useful for the calculation of functions gradient using Adjoint method.

Chapter 3

PyBeam development and extension

The main goal of this Chapter is to describe the extensions and addition of features to pyBeam code carried out in this thesis; these enhanced capabilities were necessary to allow the employment of pyBeam within a structural gradient-based optimization process (Chapter 4). The added capabilities are listed below:

1. Instead of defining as input the values of moments of inertia of the beam section (I_y, I_z, J, A), it is now possible to specify the cross-sectional wingbox geometric parameters, which are then treated as DVs in the optimization problem. Within pre-processing layer takes care of transforming the combination of these DVs the inertial properties of the section.

2. In the post-processing layer, once displacements and internal forces are available, the stress state at the central section of each finite element is retrieved, based on the classic theory of reinforced shell .

3. With the aid of Von Mises criterion, the shear are transformed to equivalent stresses, and a reserve factor measuring the distance from the maximum allowable stress is built. All these factors are aggregated according to the Kreisselmeier–Steinhauser method in a single response measuring the integrity of the structure in terms of allowable stress.

4. Buckling is taken into account employing the theory of local instability of stiffeners. Through the semi-empiric Vallat formula, the reserve factor with respect to the buckling occurrence is evaluated for each FE. As above, all these factors are aggregated into a single integrity response by means of the Kreisselmeier–Steinhauser function.

5. In the post-processing layer, a function which calculates the weight of the structure has been implemented. Such response is relevant as it will be treated as objective function in hte structural optimization problem.

3.1 Beam inertial properties

The slender wing have been modeled as a series of flexible Euler-Bernoulli beams, whose cross sections are represented by the classic wing box (figure 3.1 - 3.2)

- skin $\rightarrow$ horizontal panels
- spar $\rightarrow$ vertical panels
- stringers $\rightarrow$ represented in grey
- caps $\rightarrow$ represented in blue

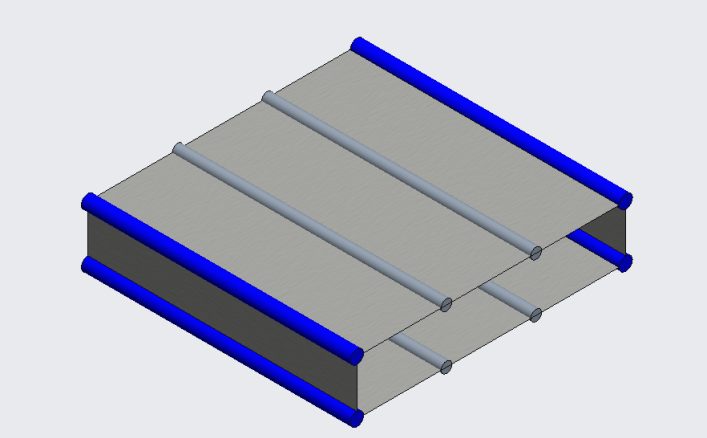


FIGURE 3.1. Isometric view of a wing box

First step is to evaluate, from the geometric properties of the rectangular and doubly-symmetric wing box (figure 3.1), the inertial properties of the wing section. These inertial properties drive the response of the system when external loads are applied. According to the employed reference frame, the **y**-axis is aligned with the chord-wise direction, **x**-axis aligned with the beam, leaving the section, and the **z**-axis perpendicular to x- and y-axes , forming a right-handed coordinates system. The reference frame

is centered in the section center of gravity and the axes are principal of inertia because the wing box is symmetric. The main purpose is to determine the moment of inertia with respect to the vertical axis, **Iz**, and to the horizontal one, **Iy**, and finally the Torsional moment of inertia **Jt** of the section.

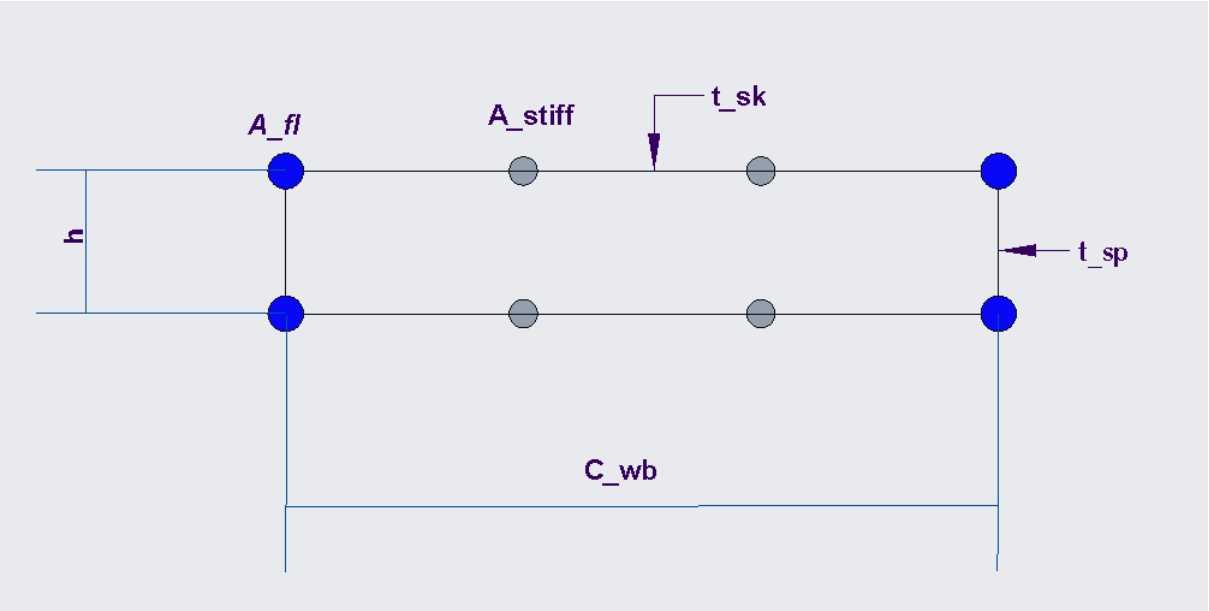


FIGURE 3.2. wing box section

As shown in figure (3.2), section properties that can be defined by the user, and can be selected as DVs in the optimization problem, are the following ones:

- A_{stiff} = stringers area
- A_{fl} = caps area
- C_{wb} = Box chord
- h = Box height
- n_{stiff} = Number of stringers
- t_{sk} = Skin thickness
- t_{sp} = Spar thickness

The model analyzed consists in a rectangular doubly-symmetric section with the spar caps concentrated in the corners of the box; moreover stringer have been considered uniformly distributed along the edges parallels to the y axis. An even or odd number of stringers per edges can be selected.
I_z and I_y of the whole structure are calculated with the theorem of Huygens-Steiner (eq. 3.1): they are equal to the summation of each single component moment of inertia with respect to their own center of gravity axis, plus the moment of transport of each component with respect to the the wing box center of gravity.

$$I = \sum_{i=1}^{n} I_{cmi} + A_i {d_{cm}}_i^2 \tag{3.1}$$

with i=1,...,n (number of components).

The skin and spar web contributions to the inertia can be evaluated, accordingly to figure 3.3, considering that:

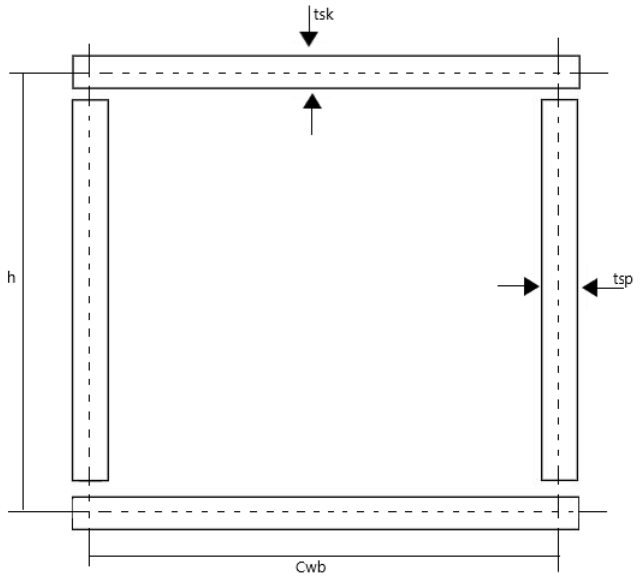


FIGURE 3.3. wing box panels (skin+spar)

- 2 horizontal rectangles with dimensions $(\mathbf{C_{wb}} + \mathbf{t_{sp}})$ **x** $\mathbf{t_{sk}}$
- 2 vertical rectangles with dimensions $(\mathbf{h} - \mathbf{t_{sk}})$ **x** $\mathbf{t_{sp}}$

Moreover, the stringers and four caps, are modeled as concentrated area.

3.1.1 Area

First parameter to calculate is the section total area. Lets trivially sum the contributions given by the skin, the spar and the system of concentrated areas. It should be taken into account that the core is composed by rectangles

$$A_{sk} = (\mathbf{C_{wb}} + \mathbf{t_{sp}})\mathbf{t_{sk}} \tag{3.2}$$

$$A_{sp} = (\mathbf{h} - \mathbf{t_{sk}})\mathbf{t_{sp}} \tag{3.3}$$

Finally , the total Area obtained is :

$$A_{tot} = 2A_{sk} + 2A_{sp} + n_{stiff}A_{stiff} + 4A_{fl} \tag{3.4}$$

3.1.2 Moments of Inertia (I_y and I_z)

When a load is applied along z or y axis the moments of inertia I_z and I_y play important role in the final deformation. Lets take as example, a classical cantilever beam operating in linear consitions: in this case $P = R_A$, moreover, a bending moment

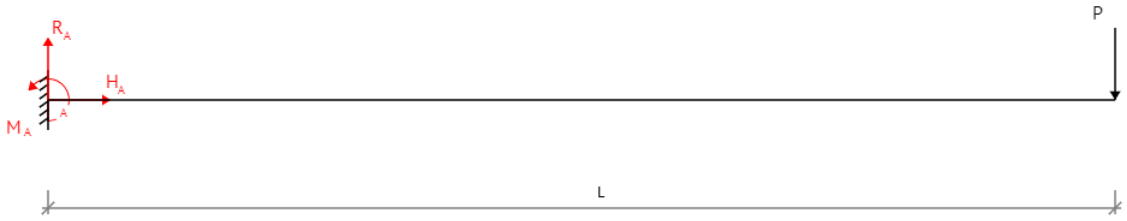


FIGURE 3.4. Cantilever beam with force at its end

is generated which vary with a linear law along the beam : $M_y = P \cdot x$, reaching its maximum value of $M_y = P \cdot L$ at the root. From the theory, the elastic line differential equation is

$$\frac{d^2\delta(x)}{dx^2} = \frac{Px}{EI_y} \tag{3.5}$$

where x represents the coordinate along the beam axis direction . The solution of 3.5 depends on some known parameters as :

- Young's Modulus (E).

- Length of the beam (L).
- Concentrated load (P).

On the other hand, there is one term that is not known: the moment of inertia of the cross section with respect to the horizontal axis. The same line of reasoning can be done for a load applied in y direction , in this case the unknown is I_z. For this reason, analytic expressions have been implemented in pyBeam , for the calculation of I_y and I_z.

The above example refers to a linear beam, whereas pyBeam treats also geometric nonlinearities. However, within a finite element, the same line of reasoning holds.

The contributions for spar web to the inertia is:

$$I_{y_{spar}} = \frac{t_{sp}h^3}{12} \tag{3.6}$$

$$I_{z_{spar}} = \frac{t_{sp}^3 h}{12} + A_{sp}\left(\frac{C_{wb}}{2}\right)^2 \tag{3.7}$$

Instead, for the upper/lower skin we obtain:

$$I_{y_{skin}} = \frac{t_{sk}^3 C_{wb}}{12} + A_{sk}\left(\frac{h}{2}\right)^2 \tag{3.8}$$

$$I_{z_{skin}} = \frac{t_{sp}C_{wb}^3}{12} \tag{3.9}$$

$I_{z_{spar}}$ and $I_{y_{skin}}$ depend on two terms: the first represents the inertia with respect to their own barycentric axis, the second is the moments of transport with respect to the barycentric axis of the whole section.
Iy_{spar} and Iz_{skin} have not moments of transport because the horizontal axis of the spar and the vertical axis of the skin matches the wing box axis.
For the stringers and caps, we have :

$$Iy_{booms} = n_{stiff}A_{stiff}\left(\frac{h}{2}\right)^2 + 4A_{fl}\left(\frac{h}{2}\right)^2 \tag{3.10}$$

$$Iz_{booms} = n_{stiff}A_{stiff}{z_i}^2 + 4A_{fl}\left(\frac{C_{wb}}{2}\right)^2 \tag{3.11}$$

where z_i represents the distance of each stringers from the z axis.
To calculate the inertia of the section we should sum the contributions of each compo-

nent as follows

$$I_y = 2I_{y_{spar}} + 2I_{y_{skin}} + I_{y_{booms}} \tag{3.12}$$

$$I_z = 2I_{z_{spar}} + 2I_{z_{skin}} + I_{z_{booms}} \tag{3.13}$$

3.1.3 Torsional moment of inertia (Jt)

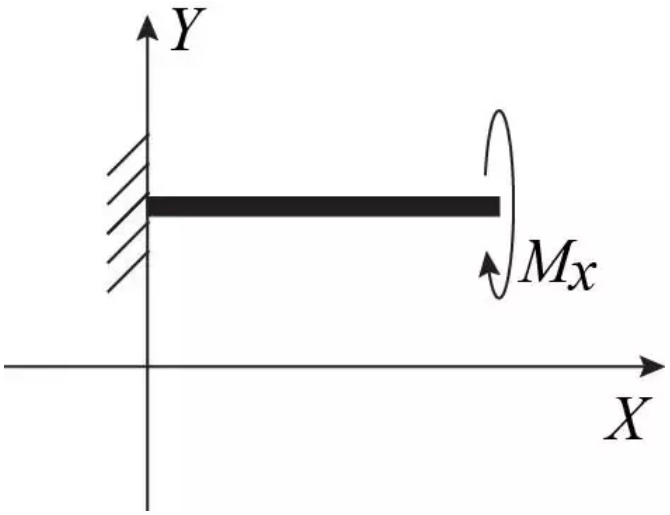


FIGURE 3.5. Torsion of a beam due to the application of an external torque

When a torsion is applied along the beam (figure 3.5), the angular deformations are given by the Saint Venant's torsion theory. Below, the hypothesis of the theory:

- Each cross-section rotates as a rigid body (no distortion of cross-section shape).
- Rate of twist is equal to a constant.
- Cross-sections are free to warp in x-direction.

Similarly to the equation 3.5, from the theory, the following differential equation with the twist angle as unknown, is obtained:

$$\frac{d\phi}{dx} = \frac{M_x}{GJ_t} \tag{3.14}$$

where ϕ = *twist angle* , M_x = *torque*, G = *the shear modulus of the material* and J_t = *Torsional moment of inertia*. If we integrate along the beam the expression 3.14, the twist angle due to the applied torque is obtained.

It is again brought to the attention the fact that this theory is applied, inside the overall geometrically nonlinear FE method, only within a single finite element.

To solve 3.14, is necessary an analytical expression for J_t. The second Bredt law for a closed thin section can be considered:

$$J_t = \frac{4A^2}{\oint_A \frac{ds}{t(s)}} \tag{3.15}$$

where s=*curvilinear coordinate along the mean line* , A = *Area surrounded by the mean line*, t = *local thickness*. Equation 3.15, can be adapted to the section in exam (figure 3.6),

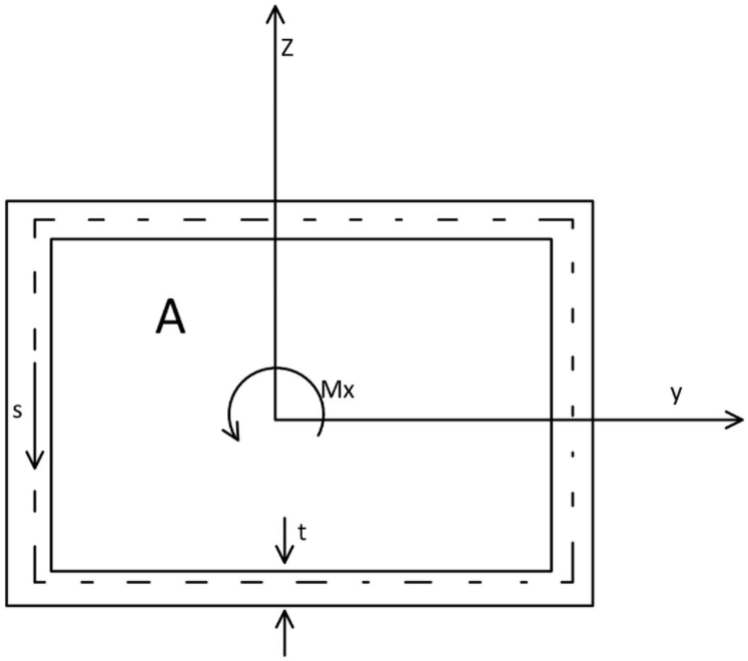


FIGURE 3.6. Torque for a wing box

obtaining the following expression :

$$J_t = \frac{2 \cdot t_{sp} \cdot t_{sk} \cdot C_{wb}^2 \cdot h^2}{C_{wb} \cdot t_{sp} + h \cdot t_{sk}} \tag{3.16}$$

3.1.4 Inertial properties validation

Previous formulae for the inertia calculations have been implemented in pyBeam, as shown in appendix A.1. In order to validate the code, an online moment of

inertia calculator has been used to cross-check results given by pyBeam. The test case analyzed is the following:

- n_{stiff}=0
- A_{sp}= 200 mm^2
- h= 500 mm
- C_{wb}= 3000 mm
- t_{sk} = 3 mm
- t_{sp} = 5 mm

As mentioned above, using the Huygens-Steiner relations, the section moment of inertia can be calculated as the sum of the inertia due to the spar and web (the core), and the inertia due to the system of concentrated areas (stringers + 4 caps). From the

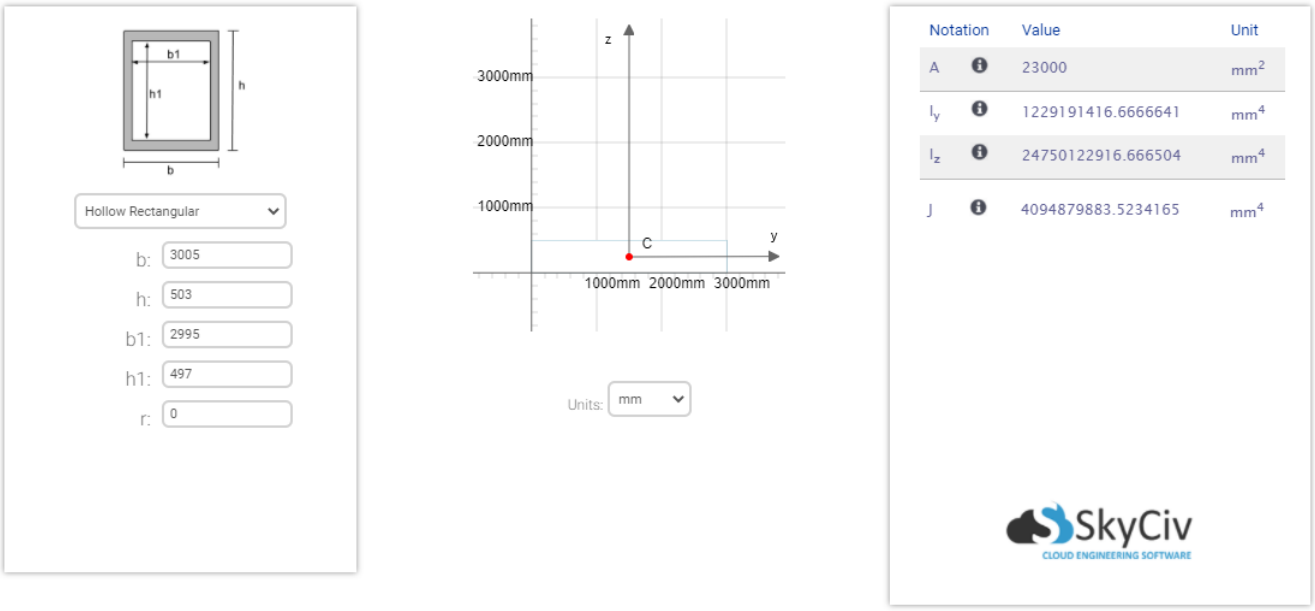


FIGURE 3.7. Validation of inertial properties calculation with an online calculator

figure 3.7, the calculator retrieves the values of I_z, I_y and J_t of the section.

The inertia of the booms is the following :

$$I_{y_{booms}} = 4 \cdot A_{fl} \cdot (\frac{h}{2})^2 = 4 \cdot 200 \cdot (250)^2 = 50000000 \quad mm^4$$
$$I_{z_{booms}} = 4 \cdot A_{fl} \cdot (\frac{C_{wb}}{2})^2 = 4 \cdot 200 \cdot (3000)^2 = 1800000000 \quad mm^4$$

by summing the moment of inertia calculated with the online calculator and the moment of inertia of the concentrated areas system, it is obtained:

$$A_{tot} = A_{core} + 4 \cdot A_{fl} = 2300 + 800 = 23800 \quad mm^2$$
$$I_{ytot} = Iy_{booms} + Iy_{panels} = 50000000 + 1229191416{,}6666641 = 1{,}2792e+09 \quad mm^4$$
$$I_{ztot} = Iz_{booms} + Iz_{panels} = 1800000000 + 24750122916{,}666504 = 2{,}6550e+10 \quad mm^4$$
$$J_t = 4094879883{,}5234165 \quad mm^4$$

When comparing these values with the ones provided pyBeam implementation, a perfect match is found.

3.2 Stress retrieving: theory and implementation in pyBeam

For structural analysis and optimization purposes, it turns to be interesting to calculate the stress state of a particular section at a certain x-station, knowing the value of the external loads. This can be done considering a single finite element beam, and the relative forces acting on the edge nodes. For a linear analysis, or a converged nonlinear analysis, the nodal forces are easily retrieved by the finite element code. Within each finite element, the beam is modeled as a linear behaving one.

Let us consider the case of figure 3.8, representing a cantilever beam (of length L), loaded at the tip with a vertical (P_z) and an horizontal (P_y) load. Calculation of constraint reaction forces is trivially obtained, as represented in the figure 3.9:

$$\begin{cases} H_A = P_y \\ V_A = P_z \\ M_{Ay} = P_z \cdot L \quad \rightarrow with respect\, to\, the\, fixed\, end \\ M_{Az} = P_y \cdot L \quad \rightarrow with respect\, to\, the\, fixed\, end \end{cases}$$

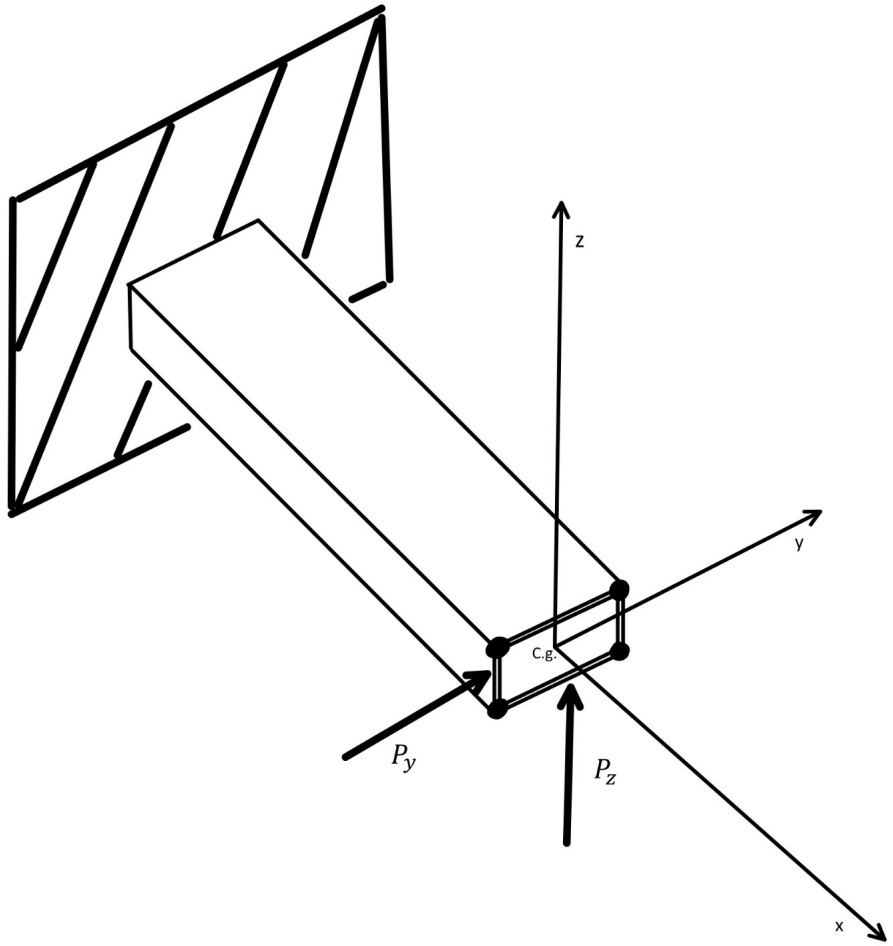


FIGURE 3.8. 3D cantilever beam loaded at the tip

Next step consists in drawing the diagrams of moment and shear using the rule of "Beam Segment": it is a convention to identify signs for normal stress, shear stresses and bending moments. In figure 3.10 the diagrams of shear and moments are plotted with an example of beam convention.
Shear stresses are constant along the axis:

$$P_z(x) = P_z \tag{3.17}$$

$$P_y(x) = P_y \tag{3.18}$$

For the moments, their trend changes linearly with the x-direction (from the fixed end) :

$$M_y(x) = P_z(L - x) \tag{3.19}$$

$$M_z(x) = -P_y(L - x) \tag{3.20}$$

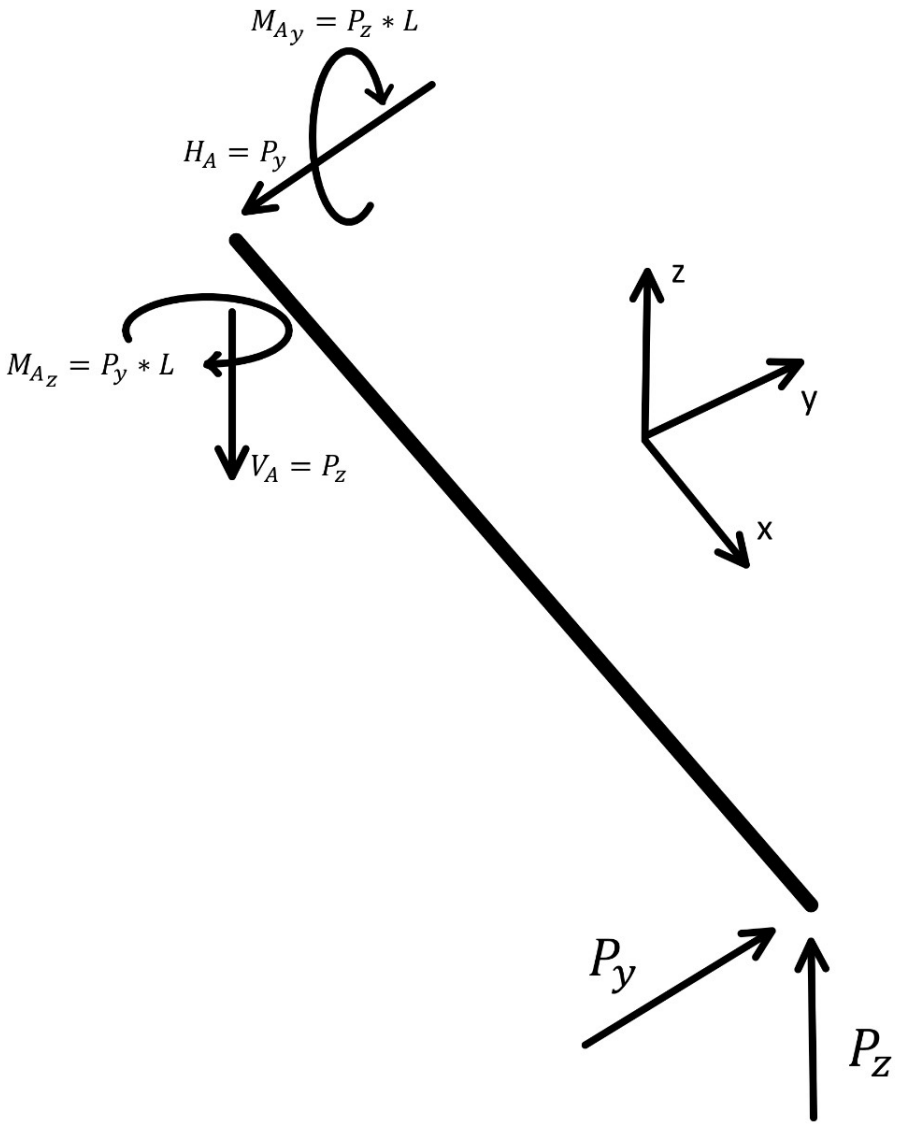


FIGURE 3.9. Balance of forces

Recall that, such example can easily translated to a single finite element, which experiences forces only at edges, whose axis is considered as straight for equilibrium purposes.

3.2.1 Reinforced shell theory (Navier Formula)

Since we have found loads and moments in a specific section, let's pass to determine the normal stress, absorbed by the booms, and the shear stress absorbed by the skin and spars. The theory used is the "reinforced shell theory": in the actual structure , each component can withstand every type of load but some more than other;

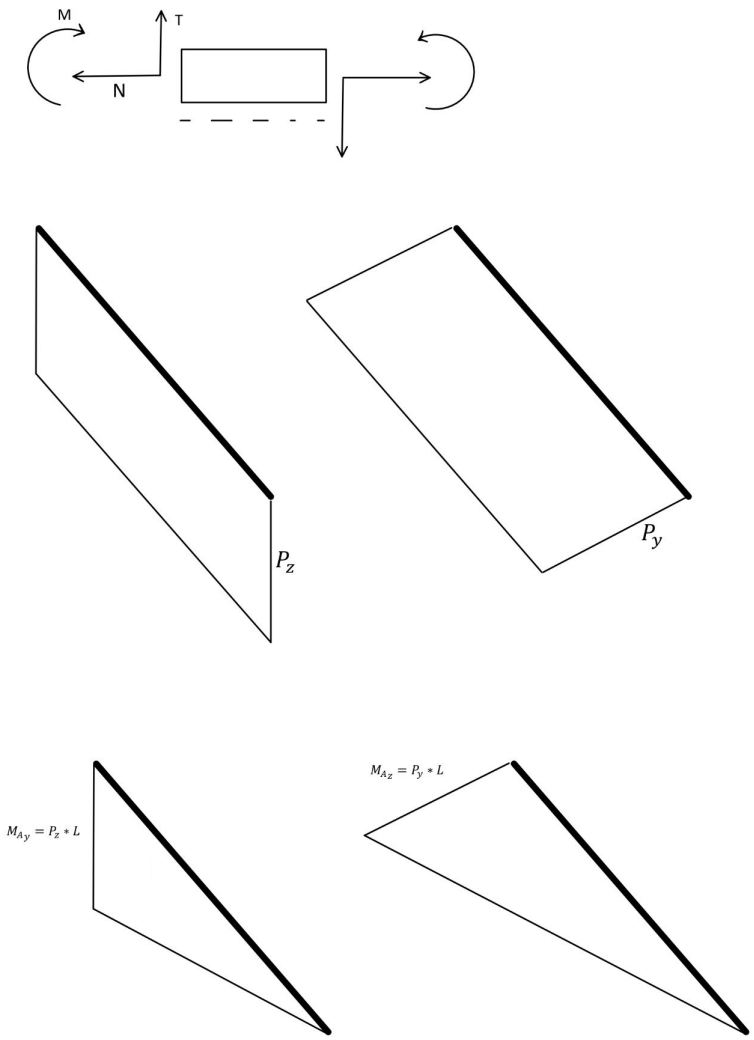


FIGURE 3.10. Shear and Moments diagrams with an example of beam convention

in general spar and skin absorb mainly the shear stresses , instead stringers and caps just the normal ones. Nevertheless, for the compressed panels, a small strip attached to the stringers and caps can withstand compression (effective width), instead in case of stretched panels, they can bear normal stress well; using this theory is possible to study separately normal and shear stress, considering the real structure as the sum of two systems :

1. System of booms: concentrated areas that can just withstand compression; it is composed by stringers and caps whose areas are enlarged with the corresponding effective-width.

2. Panels: skin and spars that can just support shear.

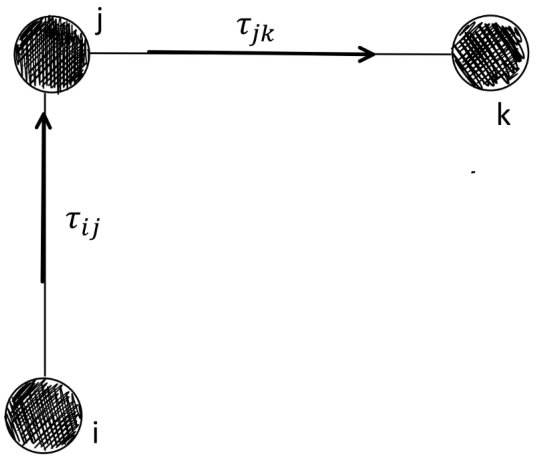


FIGURE 3.11. Fluxes balance in a generic j-boom

In order to determine normal stress, Navier Formula has been applied to the system of booms:

$$\sigma^i_{xx} = \frac{N}{A} + \frac{M_z}{I_z} Y_i + \frac{M_y}{I_y} Z_i \tag{3.21}$$

where Y , Z = *coordinates of the concentrates areas (marked by the index "i")*, N= *Normal load*, σ_{xx}=*Normal stress*. For the calculation of shear stress, according to the theory, the sum of shear fluxes converging to a stiffener is equal to the gradient of the normal stress in the stiffener itself. So, once calculated the gradients

$$\frac{dN^i}{dx} = A_i \frac{d\sigma_{xx}}{dx}^i = A_i \frac{P_y}{I_z} Y_i + A_i \frac{P_z}{I_y} Z_i \tag{3.22}$$

a system of equations can be written considering the equilibrium to the translation in correspondence of each booms, remembering that positive flux enters in the node instead the negative one comes out from it. For the generic boom in figure 3.11, the balance is the following :

$$\frac{dN^j}{dx} + \tau_{ij} - \tau_{jk} = 0 \tag{3.23}$$

Writing this balance for each concentrated area, is obtained a system of $n_{booms} - 1$ linearly independent equations in n_{booms} unknown (τ_{ij}). To close the system, it is necessary the equilibrium of moments in one of the booms, this can be done using the first Bredt Formula:

$$M_t = 2\tau \cdot \Omega \cdot t \tag{3.24}$$

where M_t=*Torque*, τ= *Shear flux*, Ω= *area enclosed in the section mean line* , t=*local thickness*. In our case:

$$M_t = P_z \cdot a + P_y \cdot b \tag{3.25}$$

where a, b represent the arms of the external forces with respect to a chosen boom. Now it is possible to solve the system in the unknown τ_{ij}. Finally, by dividing the fluxes for the local thickness, are obtained the corresponding shear stresses.

3.2.2 Stresses retrieving implementation

The function shown in appendix A.2 allows to calculate the stress state each FE. Three generic examples of cross sections that can be treated are shown in figures 3.12, 3.13 and 3.14. Lets analyze the case shown in figure 3.8, considering the wing box of

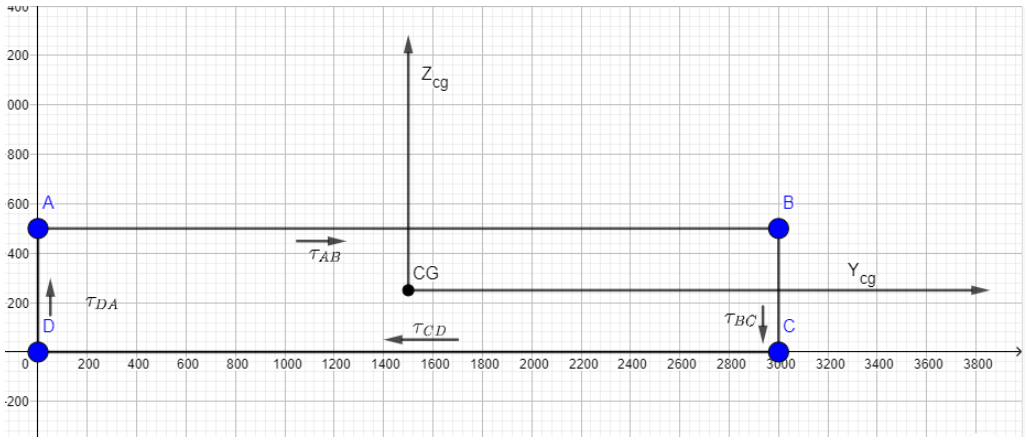


FIGURE 3.12. wing box with no stringers

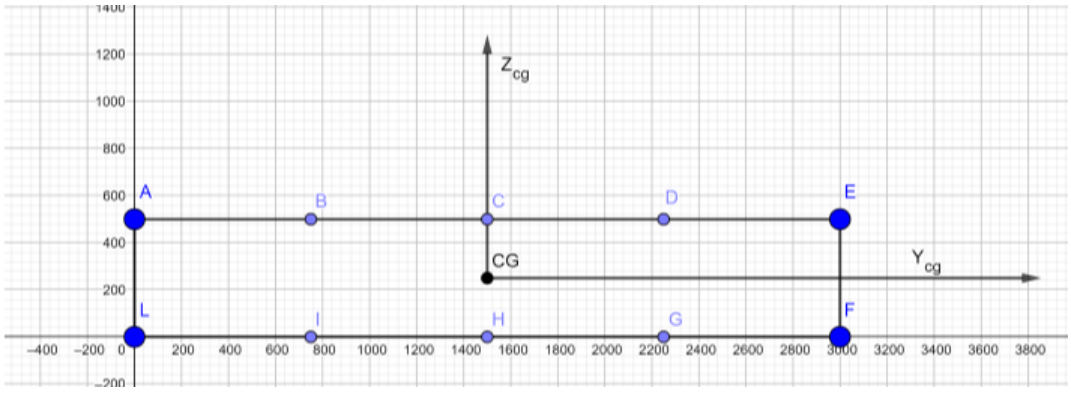


FIGURE 3.13. wing box with a odd number of stringers per skin

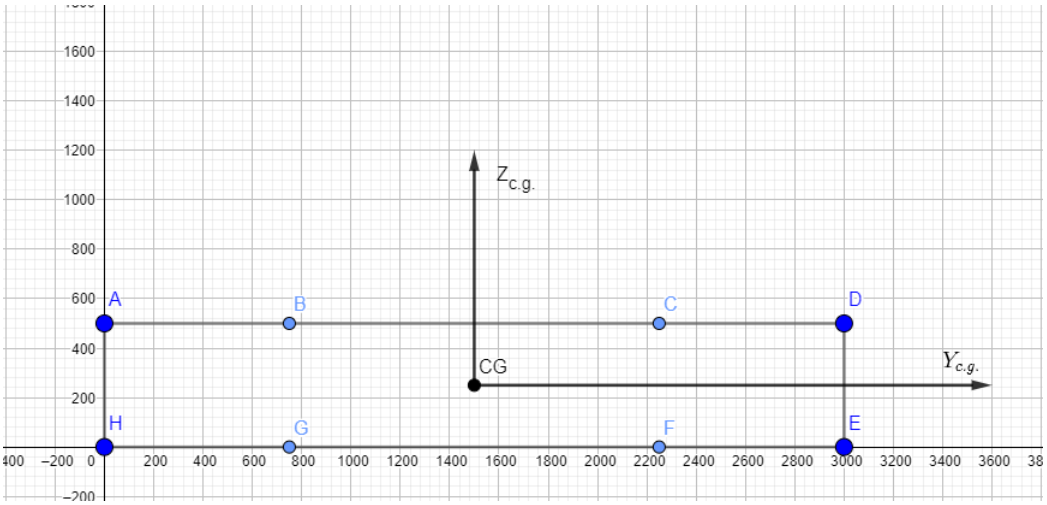


FIGURE 3.14. wing box with an even number of stringers per skin

figure 3.12. Knowing the inertial properties (section 3.1), the section geometric parameters ($C_{wb}, h, t_{sk}, t_{sp}, n_{stiff}, A_{fl}, A_{stiff}$), and the concentrated loads, the equation 3.21 has been implemented:

$$\sigma_{xx}^{A} = \frac{N}{A} + \frac{M_z}{I_z}Y^A + \frac{M_y}{I_y}Z^A$$
$$\sigma_{xx}^{B} = \frac{N}{A} + \frac{M_z}{I_z}Y^B + \frac{M_y}{I_y}Z^B$$
$$\sigma_{xx}^{C} = \frac{N}{A} + \frac{M_z}{I_z}Y^C + \frac{M_y}{I_y}Z^C$$
$$\sigma_{xx}^{D} = \frac{N}{A} + \frac{M_z}{I_z}Y^D + \frac{M_y}{I_y}Z^D$$

with:

$$\begin{array}{cc} Y^A = \frac{-C_{wb}}{2} & Z^A = \frac{h}{2} \\ Y^B = -Y^A & Z^B = Z^A \\ Y^C = -Y^A & Z^C = -Z^A \\ Y^D = Y^A & Z^D = -Z^A \end{array}$$

Then , from equation 3.22 the gradients of normal stress in the nodes can be computed:

$$\frac{dN^A}{dx} = A_{fl}\frac{P_y}{I_z}Y^A + A_{fl}\frac{P_z}{I_y}Z^A$$
$$\frac{dN^B}{dx} = A_{fl}\frac{P_y}{I_z}Y^B + A_{fl}\frac{P_z}{I_y}Z^B$$
$$\frac{dN^C}{dx} = A_{fl}\frac{P_y}{I_z}Y^C + A_{fl}\frac{P_z}{I_y}Z^C$$

Finally, using the expression 3.23 and from the equilibrium of Moments (with pole in D) equation 3.24, is written a system of four equations in four unknown:

$$\begin{cases} \dfrac{dN^A}{dx} + \tau_{DA} - \tau_{AB} = 0 \\ \dfrac{dN^B}{dx} - \tau_{BC} + \tau_{AB} = 0 \\ \dfrac{dN^C}{dx} + \tau_{BC} - \tau_{CD} = 0 \\ P_y a + P_z b = 2\tau_{AB}\Omega_{ABD} + 2\tau_{BC}\Omega_{BDC} \end{cases} \tag{3.26}$$

Ω_{ABD} and Ω_{BDC} are the upper and lower triangular areas that form the rectangular section.
For the cases in figure 3.13 and 3.14, the way of proceeding is exactly the same; the only differences are the values of inertial properties and the size of the system.

3.3 Constraints

For most optimization problems we are going to deal with, design constraints are necessary and must be handled. The number of constraints considered can affects the convergence of the optimization iterative algorithm and its computational cost. For this reason, an approach should be used able to minimize these effects in order to obtain the optimum in a reasonable time with adequate accuracy. As will be explained in chapter 4, an optimization consists in minimizing an objective function, given a number of design variables and constraints.
The standard approach to an optimization constrained problem is:

$$\begin{aligned} & min \qquad f(x) \\ & wrt \qquad x \in \mathbb{R}^n \\ st \qquad & g_j(x) \leq 0 \qquad j = 1, \ldots, N_c \end{aligned} \tag{3.27}$$

where x=*design variables*, g_j=*individual constraint* and N_c=*number of constraints*.
For problems of our interest, the number of constraints would be very high if considering the stresses on each finite element, so it turns out to be prohibitively expensive in terms of computational costs as the particular method employed for sensitivity

evaluation, i.e., the adjoint method, gives an advantage when number of constraints is smaller than number of design variables. To fix this issue, constraint aggregation method is used. The implemented one is the Kreisselmeier–Steinhauser function (KS). Constraints of our interest are two: the first expresses the condition that the equivalent stresses that must not overcome an allowable value (appxA.5), and the second which ensures that the stringers are free from Buckling (appendix A.6).

3.3.1 Individual constraints calculation : Von Mises criterion

For a weight optimization of a structure made of "j" beam elements, the problems is

$$\begin{aligned} &min \quad Weight(x_i) \\ &wrt \quad x_i && with \quad i = 1, \dots, n \\ &st \quad g_j = \frac{\sigma_j(x)}{\sigma_{allow}} - 1 \leq 0 && with \quad j = 1, \dots, N_c \end{aligned} \tag{3.28}$$

The objective function is *Weight(x_i)*, the design variables x_i are : A_{stiff}, A_{fl}, t_{sk}, t_{sp}, h and C_{wb}; the inequality constraint g_j dictates that the maximum stress in each element section $\sigma_j(x)$, can not overcome the allowable stress σ_{allow}; if that happens , the failure of the structure shows up. It has been decided to analyze the middle section of each elements to determine $\sigma_j(x)$, instead for σ_{allow} has been considered the yielding stress of the material in question.

Individual constraints are determined using the function presented in appx. A.2 : for the case of figure 3.12 are retrieved four normal stresses and four shear stresses. Then , can be computed the system of constraints :

$$g_j = \left(\frac{\sigma_{ej}}{\sigma_{all}}\right) - 1 \qquad j = 1, \dots, Nc \tag{3.29}$$

Where σ_e is the equivalent stress. For Von mises (Energetic criterion), in case of general state of tension :

$$\sigma_e = \sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2 - (\sigma_x\sigma_y + \sigma_x\sigma_z + \sigma_z\sigma_y) + 3(\tau_{xy}^2 + \tau_{zx}^2 + \tau_{yz}^2)} \tag{3.30}$$

In the concentrated areas , no shear stress is present and only the normal principal stress along x-axis is considered:

$$\sigma_e = \sigma_x \tag{3.31}$$

For Spar and Skin the state of tension is purely of shear:

$$\sigma_e = \sqrt{3}\tau_{xy} \qquad skin \tag{3.32}$$

$$\sigma_e = \sqrt{3}\tau_{xz} \qquad spar \tag{3.33}$$

Here τ is the shear stress obtained dividing the shear fluxes for the thickness. σ_{all} is the allowable stress in terms of Yelding stress($\sigma_{\mathbf{y}}$) scaled by a safety factor(**SF**):

$$\sigma_{all} = \frac{\sigma_y}{SF} \tag{3.34}$$

The value of SF is in general in between 1.3 and 2 but it is imposed by regulations or choose by the designer , based on the following requirements:

- the uncertainty about the loads magnitude or how they are applied,
- the uncertainty about material properties,
- the inaccuracy of the mathematical model for the stress calculation
- costs

For our purpose a **SF** = 1.5 and Aluminum 7075,$\sigma_{\mathbf{y}}$ = 468.5 Mpa , have been selected.

3.3.2 Individual constraints calculation : Buckling

A Nonlinear phenomena which can lead the wing box to fail is the caps and stringers local Buckling. Is considered a "L" section stiffeners with equals perpendicular edges and fixed thickness , figure 3.15

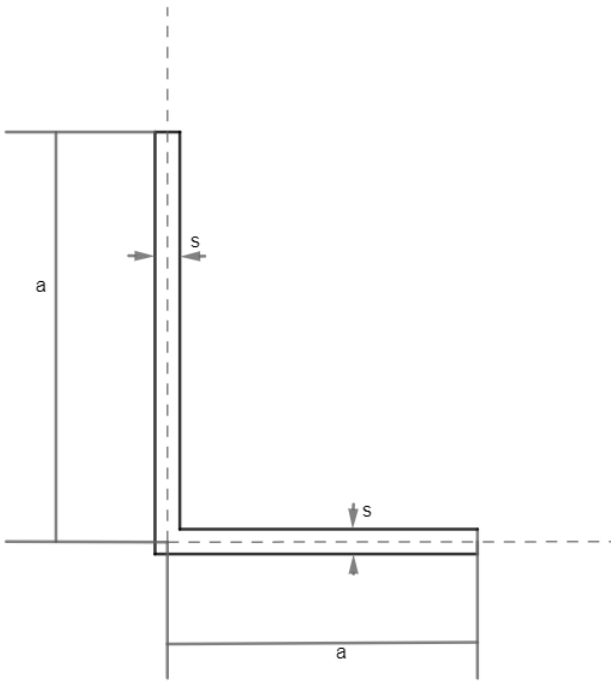


FIGURE 3.15. Caption

Vallat formula is applied to give an expression for the σ_{cr} which must not be exceeded to avoid the buckling:

$$\sigma_{cr} = \frac{\sigma_y}{1 + \beta \cdot k \cdot \frac{\sigma_s}{E}} \tag{3.35}$$

- **E** = Young module
- k = 8.5 for open section
- $\beta = \frac{a}{s}$

Knowing the booms area , fixing the stiffeners section thickness **s** , can be calculated the edge **a**. For each finite element is calculated the respective constraint.

$$g_j = \left(\frac{\sigma_j}{\sigma_{cr}}\right) - 1 \leq 0 \qquad j = 1, \ldots, Nc \tag{3.36}$$

Where N_c represents the number of compressed booms.

3.3.3 Aggregation constraint calculation : Kreisselmeier–Steinhauser function

For our purpose , a gradient-based optimization has been used; this class of algorithms allows to solve optimization problems, relying on the calculation of the sensitivities, these represent the gradient of objective function and constraints w.r.t. design variables. Moreover, an effective way of estimating gradients is the Adjoint Method(Chapter 4). By using constraint aggregation, individual constraints are lumped into a single composite function. With only a single or few composite constraints, the Adjoint method can compute the sensitivities efficiently.

The KS function was first presented by G. Kreisselmeier and R. Steinhauser. It produces an envelope surface that states for an estimation of the maximum among a set of functions. KS definition is

$$KS(g_j(x)) = \frac{1}{\rho} \cdot ln\left[\sum_{j=1}^{Nc} e^{\rho g_j}\right] \tag{3.37}$$

where ρ is the *Aggregation Parameter*.
The function 3.37 has the following properties [8] :

- KS (x, ρ) $\geq$ max($g_j(x)$) for $\rho > 0$
- $\lim_{\rho\to\infty} KS(x,\rho) = max(g_j(x))$ for j = 1, . . . , Nc
- KS (x, ρ_1) $\geq$ KS (x, ρ_2) for $\rho_1 > \rho_2 > 0$
- KS (x, ρ) is convex, if and only if all constraints are convex

Cause of the magnitude of aggregation parameter , the calculation of equation 3.37 can induce numerical overflow, so to reduce this issue , an alternative form is used :

$$KS = g_{max} + \frac{1}{\rho} \cdot ln\left[\sum_{j=1}^{Nc} e^{\rho(g_j - g_{max})}\right] \tag{3.38}$$

where g_{max} is the maximum of the constraints at the current design point, x. The value of KS function is bounded from above and below as follows:

$$g_{max} < KS < g_{max} + \frac{\ln N_c}{\rho} \tag{3.39}$$

From this expression is possible to understand the importance of the "Draw-Down" factor, as it approach infinity, the KS function became the maximum of all constraints (g_{max}); it is possible to calculate the error, in fact from the expression 3.39, the lower boundary is the most accurate and, for approaching it, ρ should be as higher as possible; nevertheless, higher is ρ higher is the possibility of running into numerical problems. In fact, estimating the Hessian of the KS function at such points, leads to an ill-conditioned matrix when ρ is too large, which causes numerical difficulties . Based on what we said , could be implemented two type of KS function : the adaptive and the no adaptive form.

Cause of the simplicity of models we are going to use the no-adaptive form. The main difference between the two versions, reside in the definition of the "draw-down" factor ρ: for the no-adaptive case , ρ is fixed and this , as it will be demonstrated , can cause a loss of accuracy for problems with high number of individual constraints; the adaptive form consists in considering an initial value of ρ which changes by increasing it as needed according to the sensitivity of KS with respect to the aggregation parameter,$\frac{dKS}{d\rho}$, at the current design point. The adaptive form has been implemented in pyBeam functions appx. A.5 - A.6.

For the properties of KS function , the higher is ρ value , the lower is the error commit to aggregate the constraints. To prove this feature , it has been run the function shown in appx. A.5 using the same input data exploited for the validation of inertial properties (section 3.1.4) but varying the value of ρ and the number of constraints N_c (0 and 6 stringers with A_{stiff} = 50 mm^2). Processing in Matlab the output obtained from pyBeam, lets get the trend of the error $Error = |\frac{Ks - g_{max}}{g_{max}}|$ versus the variation of ρ and N_c.

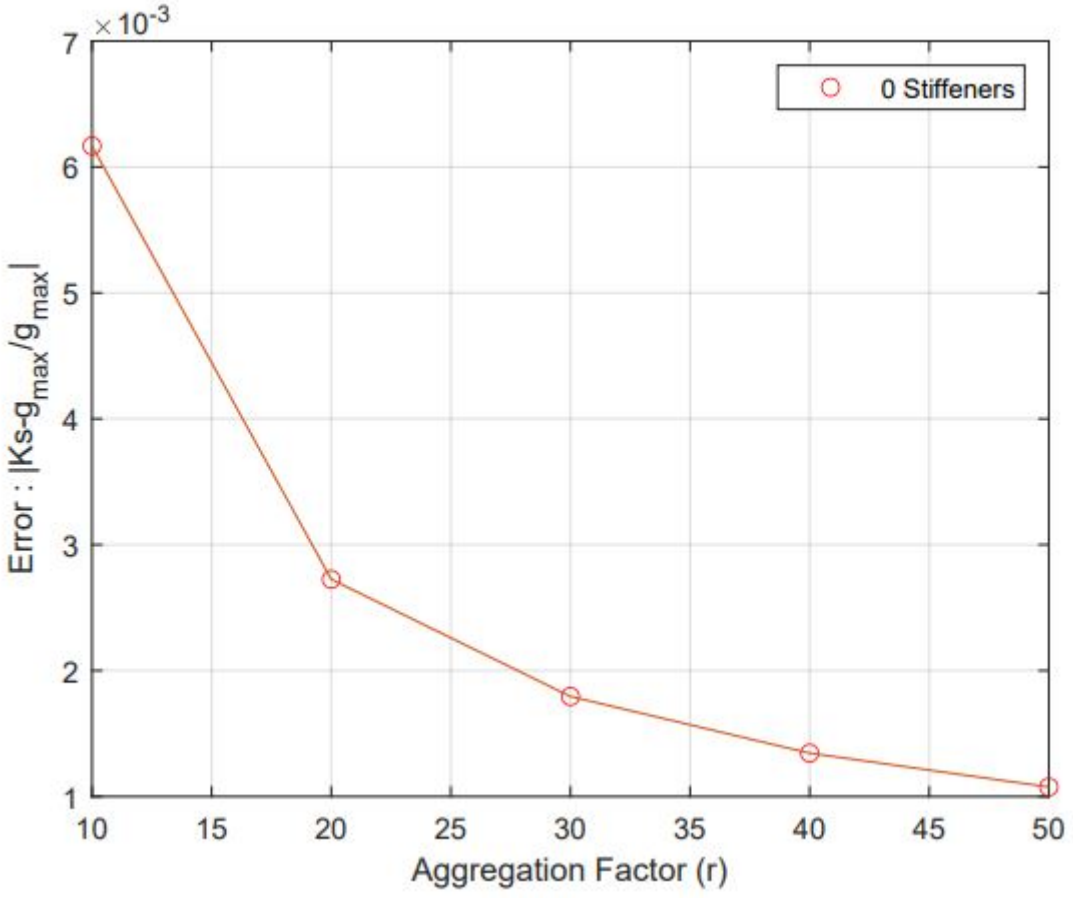


FIGURE 3.16. KS error for increasing r

Increasing the "draw-down" factor from a value of 10 to 50 , can be seen a decreasing error. Anyway for the case in exam (no many constraints), the order of the error is of 10^{-3}.

In case of just four caps , and 60 finite elements , the number of constraints is equal to 480; if is considered the wing box with 6 stringers , the number of active constraints rise to 1200. According to the theory , the higher is the number of constraints the higher is the error commit, this result is clearly visible below 3.17. The aggregation number is set to a value of 50 :

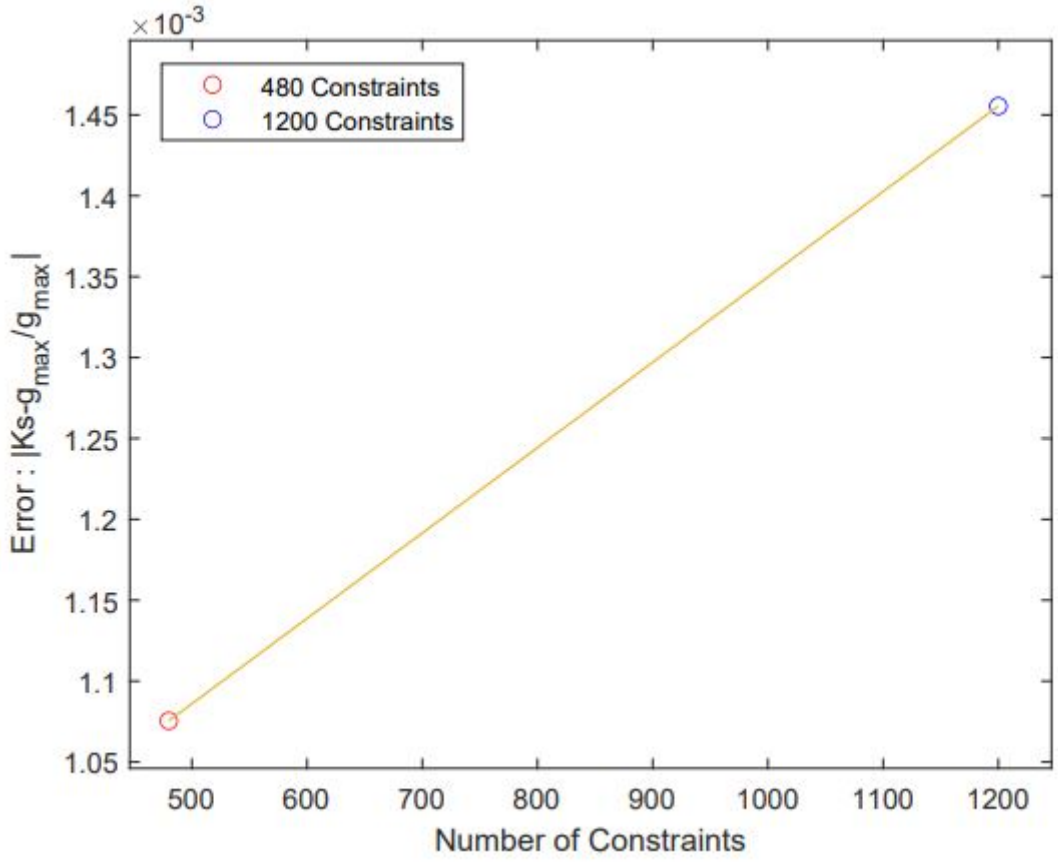


FIGURE 3.17. Relative error using traditional KS function with r = 50 for increasing constraint

3.4 Objective function calculation

The weight of the whole structure , is calculated summing the weight of each finite element :

$$W = \sum_{i=1}^{n_{fem}} (\rho l_i A_i) \tag{3.40}$$

$\mathbf{n_{fem}}$ is the number of elements , ρ is the density, $\mathbf{A_i}$ is the area of the element section and $\mathbf{l_i}$ is the current length of the element. Function implemented in pyBeam which returns the total weight is presented in appx.A.7.

Chapter 4

Structural optimization: theory and implementation

Numerical optimization consists in finding the best solutions to mathematically defined problems in different fields. As anticipated in the previous chapter, the mathematical formulation of an optimization problem is the following :

$$minimize\ f(\mathbf{x})\ wrt\ \mathbf{x},\ with\ \mathbf{x} = [x_1, x_2, \ldots, x_n]^T\ \in\ R^n \tag{4.1}$$

Subjected to the constraint :

$$\begin{aligned} g_j(\mathbf{x}) \leq & 0,\ with\ j = 1, \ldots, m \\ h_j(\mathbf{x}) = & 0,\ with\ j = 1, \ldots, r \end{aligned} \tag{4.2}$$

- $f(\mathbf{x})$: *objective function*,
- **x**: *column of design variables* ,
- $g_j(\mathbf{x})$: *inequality constraint*,
- $h_j(\mathbf{x})$: *equality constraint*.

The optimum vector **x**, denoted as **x***, represents the solution of problem 4.1.

4.1 Minimization problem

Before moving on to structural optimization (more than two design variables), is important to visualize solutions displayed on the chartesian plane , for this reason it is going to be analyzed the case of one and two dimensional minimization.

4.1.1 Equality constrained one-dimensional problem

If the objective function depends on just one variable x, the minimum of the function can be easily calculated by setting the first derivative to zero and the second derivative bigger than zero :

$$\begin{cases} f'(x) = \dfrac{df(x)}{dx} = & 0 \\ f''(x) = \dfrac{df(x)^2}{d^2x} \geq & 0 \end{cases} \rightarrow x^* | \; f(x^*) \quad minimum\ value$$

for simple cases in which the objective function is not complex ,like a quadratic form (figure4.1), the solution can be retrieved analytically; instead, for a more general form of "f(x)", the solution can be calculated numerically. The solution may be obtained numerically via *Newton-Raphson algorithm:*

$$x^{i+1} = x^i - \frac{f'(x^i)}{f''(x^i)}, \quad i = 1, 2, \ldots, n_{iter} \tag{4.3}$$

this method relies on the calculation of the first and second derivatives of the objective function. Giving an initial value x^0, after several number of iterations , x should approach the optimum design variable x* :

$$\lim_{x \to +\infty} x^i = x^* \tag{4.4}$$

if the problem is constrained can be defined an inequality ($g(x) \leq 0$) and an equality($h(x) = 0$) constraints . In this case , a closed-form analytical solution is not simple to get and particular constrained optimization techniques, such as the method

of Lagrange multipliers, must be applied to solve the constrained problem analytically. Let's show a qualitative example of a simple quadratic function minimization , with a linear equality constraint : function depicted in figure 4.1 is a simple parabola :

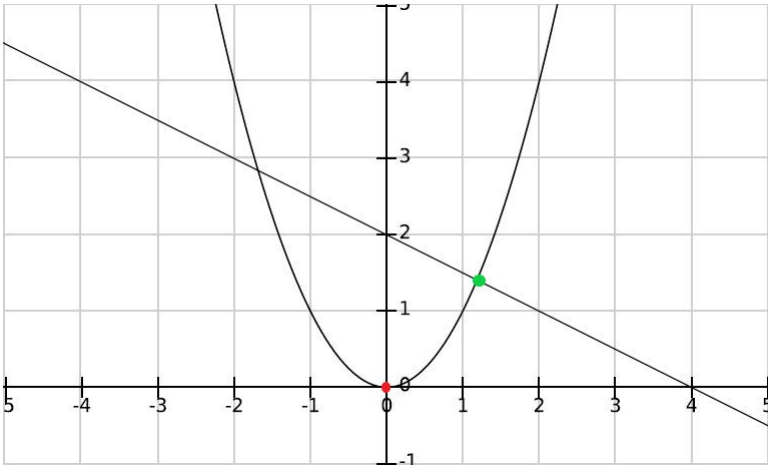


FIGURE 4.1. Parabola : f(x)=x^2 and equality constraint

$f(x) = x^2$ with a linear constraint : $h(x) = -0{,}5 \cdot x + 2$; The red circle represents the unconstrained solution , instead the real optimum is signed by the green one.

4.1.2 Inequality constrained two-dimensional problem

For a two dimension objective function, the solution can be viewed in form of contour. In figure 4.2 is sketched a contour representation of a general paraboloid , considering an inequality constraint : g(x) < 0; g(x) = 0 divides the plane into a feasible region and an infeasible region.

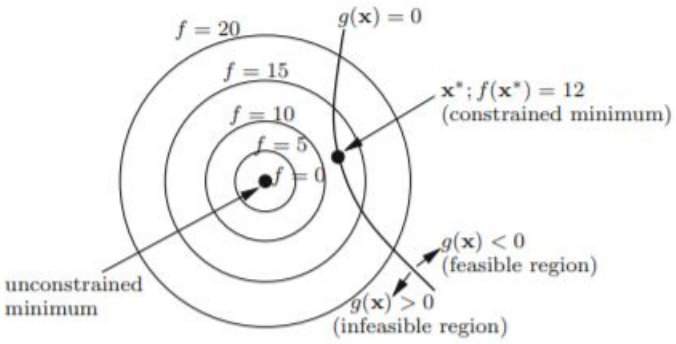


FIGURE 4.2. Contour representation of inequality constrained problem [2]

4.2 N-dimensional problem and gradient based second order optimizer(SLSQP)

If the objective function depends on several design variables, numerically algorithms can be applied for the resolution of problem 4.1. for our purpose , has been chosen the "Sequential Least Square Programming" (SLSQP), a local algorithm that uses a second order approximation of the objective function based on the knowledge of the gradient. The advantage of this method is the small number of calculation of the objective function, reducing the computational cost; its disadvantages regards the convergence of the solution to a local minimum and the evaluation of merit function gradient which, for an higher number of design variables, can represent a problem. To repair the latter problem, has been taken into account the Adjoint Method for the calculation of $\frac{df(\mathbf{x})}{d\mathbf{x}}$, it is particularly useful for problems in which has to be computed the gradient of a small number of functions respect to high number of design variables. Considering the generic n-dimensional problem 4.1,in order to include the constraint , is used the Lagrange Multipliers method, so the extended function to minimize becomes :

$$\hat{f}(\mathbf{x}) = f(\mathbf{x}) - \sum_{j=1}^{N_c} \lambda_j g_j(\mathbf{x}) \tag{4.5}$$

starting from an initial guess $\mathbf{x}_n$, a second order optimization method reckons the value of $\mathbf{x}^*$ at which $\hat{f}(x)$ is minimum. It is an iterative procedures similar to the Newton method (expression 4.3) but in n-variables. With the hypothesis of twice-differentiable, is possible to approximate the function 4.5 close to a fixed point x_n with a second order

Taylor expansion :

$$\hat{f}(\mathbf{x}) \approx \hat{f}(\mathbf{x_n}) + \Delta\mathbf{x}^T\mathbf{J}_n + \frac{1}{2}\mathbf{\Delta x}^T\mathbf{H_n}\mathbf{\Delta x} \tag{4.6}$$

where $\mathbf{J}_n$ and $\mathbf{H}_n$ are respectively the Jacobian and the Hessian of $\hat{f}(\mathbf{x})$.The function we have to minimize is $\hat{f}(\mathbf{x})$, so should be neglected the first its first derivative

$$0 = \mathbf{J_n} + \frac{1}{2}\mathbf{H_n}\mathbf{\Delta x} \tag{4.7}$$

Inverting $\mathbf{H_n}$ can be computed $\mathbf{\Delta x}$,

$$\mathbf{\Delta x} = -\mathbf{H}_n^{-1}\mathbf{J}_n \tag{4.8}$$

After updated the design point , $\mathbf{x_n}$: $\mathbf{x_{n+1}} = \mathbf{x_n} + \delta\mathbf{x}$, must be check if it neglect the function gradient. If not , you have to keep going with the iterations , if yes has been found the design variable vector which minimize $\hat{f}(x)$.
The calculation of the Hessian matrix depends on the Jacobian evaluation , in fact , using the secant method, is possible to give an expression to the Hessian :

$$\mathbf{H_n} = \frac{\mathbf{J_n} - \mathbf{J_{n-1}}}{\mathbf{x_n} - \mathbf{x_{n-1}}} \tag{4.9}$$

Since the value of $\mathbf{H_n}$ relies on $\mathbf{J_n}$, is important to find how to calculate the gradient. How previously announce, for structural optimization, is particularly profitable to use the Adjoint method.

4.3 Adjoint method

Adjoint algorithmic differentiation is a mathematical technique used to significantly speed up the calculation of sensitivities and of the total variation of a quantity of interest , which can be the objective function or the constraint in our optimization problem. The method is based on the application of the chain rule of differentiation to each operation in the program flow. The derivatives given by the chain rule can be propagated in order to finally calculate the total derivative of quantity of our interest.

$$Q(\mathbf{x}, \mathbf{u}(\mathbf{x})) \longrightarrow \ quantity\ of\ our\ interest$$
$$R((\mathbf{x}, \mathbf{u}(\mathbf{x})) = 0 \longrightarrow \ Residual\ of\ primal\ problem$$

With **x**=*design variables* and **u(x)**=*solution of structural problem (displacements)*.

Introducing a number of Lagrangian functions ($\tilde{\mathbf{q}}$) equal to the number of functions of our interest, we can join together Q and R in one expression :

$$\mathcal{L}(\mathbf{x}, \mathbf{u}(\mathbf{x}), \tilde{\mathbf{q}}) = \mathbf{Q}(\mathbf{x}, \mathbf{u}(\mathbf{x})) - \tilde{\mathbf{q}}^{\mathbf{T}}\mathbf{R}(\mathbf{x}, \mathbf{u}(\mathbf{x})) \tag{4.10}$$

The gradient of augmented function 4.10 w.r.t. design variables is :

$$\frac{d\mathcal{L}_i}{dx_j} = \frac{dQ_i}{dx_j} + \tilde{\mathbf{q}}_\mathbf{i}\frac{d\mathbf{R}}{dx_j} \tag{4.11}$$

Actually $\frac{d\mathbf{R}}{dx_j}$ = 0 , so the gradient of Q and the gradient of the augmented function are the same.
Using the chain rule can be written :

$$\frac{dQ_i}{dx_j} = \frac{\partial Q_i}{\partial x_j} + \frac{\partial Q_i}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial x_j} \tag{4.12}$$

$$\frac{d\mathbf{R}}{dx_j} = \frac{\partial \mathbf{R}}{\partial x_j} + \frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial x_j} = 0 \tag{4.13}$$

If we substitute expressions 4.12-4.13 in 4.11 obtaining :

$$\frac{dQ_i}{dx_j} = \frac{d\mathcal{L}_i}{dx_j} = \frac{\partial Q_i}{\partial x_j} - \tilde{\mathbf{q}}_\mathbf{i}^T\frac{\partial \mathbf{R}}{\partial x_j} + \left(\frac{\partial Q_i}{\partial \mathbf{u}} + \tilde{\mathbf{q}}_\mathbf{i}^T\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right)\frac{\partial \mathbf{u}}{\partial x_j} \tag{4.14}$$

Can be neglected the dependence of 4.14 on the term $\frac{\partial \mathbf{u}}{\partial x_j}$, that is highly cost to determine, putting to zero expression the term in bracket retrieving the dual Adjoint problem :

$$\begin{aligned} \frac{dQ_i}{dx_j} = \frac{d\mathcal{L}_i}{dx_j} = \frac{\partial Q_i}{\partial x_j} - \tilde{\mathbf{q}}_\mathbf{i}^T \frac{\partial \mathbf{R}}{\partial x_j} \\ \frac{\partial Q_i}{\partial \mathbf{u}} + \tilde{\mathbf{q}}_\mathbf{i}^T \frac{\partial \mathbf{R}}{\partial \mathbf{u}} = 0 \end{aligned} \tag{4.15}$$

To evaluate the gradient of the vector Q with respect to the DVs onlyN_{obj} large linear systems(second of 4.15) have to be solved, in structural shape optimization only few functional require gradient evaluation(weight and constraint) so this approach is preferred.

4.4 Structural optimization

The type of optimization we are going to deal with , is a structural optimization. The problems is the following :

$$\begin{aligned} &min\, Weight(\mathbf{DVs}) \\ &wrt\, \mathbf{DVs} \\ &st\, KS(\mathbf{DVs}) \leq 0 \end{aligned} \tag{4.16}$$

Structural optimization has been performed used openMDAO: an open-source framework for multidisciplinary design, analysis and optimization. It is primarily designed for gradient-based optimization; its most useful and unique features relate to the efficient and accurate computation of the model derivatives. OpenMDAO was developed using Python languages because it provides many options for interfacing to compiled languages like SWIG and Cython for C and C++ and it is an open-source language. For this reasons have been implemented a series of wrapping functions that allows pyBeam, whose core is coded in C++, to interact with openMDAO.

The problem is : *solve the optimization problem 4.16 in openMDAO,using the optimizer SLSQP (sec 4.2), exploiting the functions written in pyBeam (chapter 3) for the calculation of constraint and objective function, given in input the design variables*

We should imagine openMDAO as an orchestror which from one side recall functions that return values of objective function, constraint and their gradient; to the other hand it recalls, through "Scipy" Python library, the SLSQP optimizer; putting all together in the "main" module of openMDAO , this framework can returns us values of the minimum objective function $Weight(\mathbf{DVs})$ subjected to the aggregation constraint $KS(\mathbf{DVs})$. Implemented functions and their interaction will be explained in chapter 5.

4.5 OpenMDAO architecture

OpenMDAO framework allows for troubleshooting significantly larger and complex multidisciplinary design optimization (MDO) problems. It uses an object-oriented programming paradigm and an object composition design pattern. In this section,it is going to be described the framework architecture based on four most fundamental types of classes : Component, Group, Driver, and Problem. Let's describe, from the bottom to the top, classes hierarchy :

- **Component** :Instances of the Component class, give the lowest-level functionality representing basic calculations. Each component instance charts of input values to output values via some calculation. A component could be a simple explicit function or a implicit one; it can also calls external functions, in fact, for our optimization problem, the two important explicit components are functions that return objective function and constraint, recalled by the optimizer every iteration of its recursive calculation.

- **Group** : Instances of the Group are Components ,Groups or a mixture. Connections between instances corresponds to a hierarchical tree . This class is fundamental to better organize the namespaces thanks to the packaging of components sets together. The connections tree, forms the Group Model.

- **Driver**: This class is located at the same level of the Group Model; its instances call the Model using some algorithms (SLSPQ in our case): the design variables are a subset of the Model inputs, the objective and constraint functions and their derivatives are a subset of the Model outputs.

- **Problem**: this class stands for a top-level container taking in, all other objects. A problem instance contains both the groups and components that constitute the model hierarchy and also contains a single driver instance. Moreover, a Problem also provides the user interface for Model setup and execution.

The architecture is depicted below

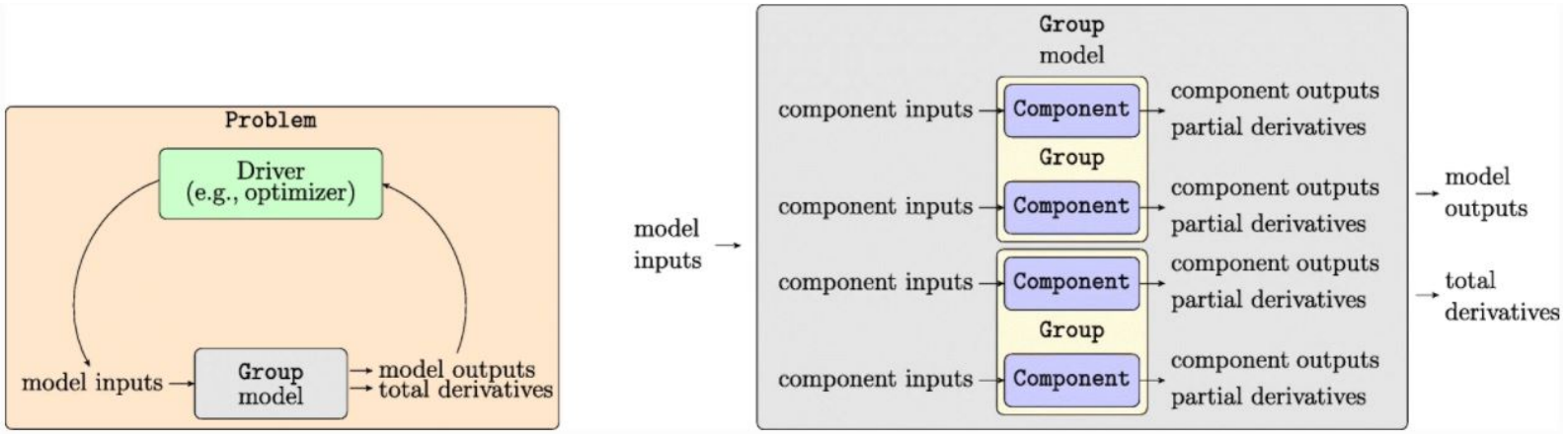


FIGURE 4.3. OpenMDAO architecture [3]

Chapter 5

Implemented codes

This chapter is focused on the description of each coded functions and on how they interact for solving structural optimization problem 4.16. The whole optimization framework consists of the new version of PyBeam, described in chapters 2-3, interfaced with openMDAO. The interface is represented by pyBeam wrapping functions that allows the core outputs to be passed as input to openMDAO.

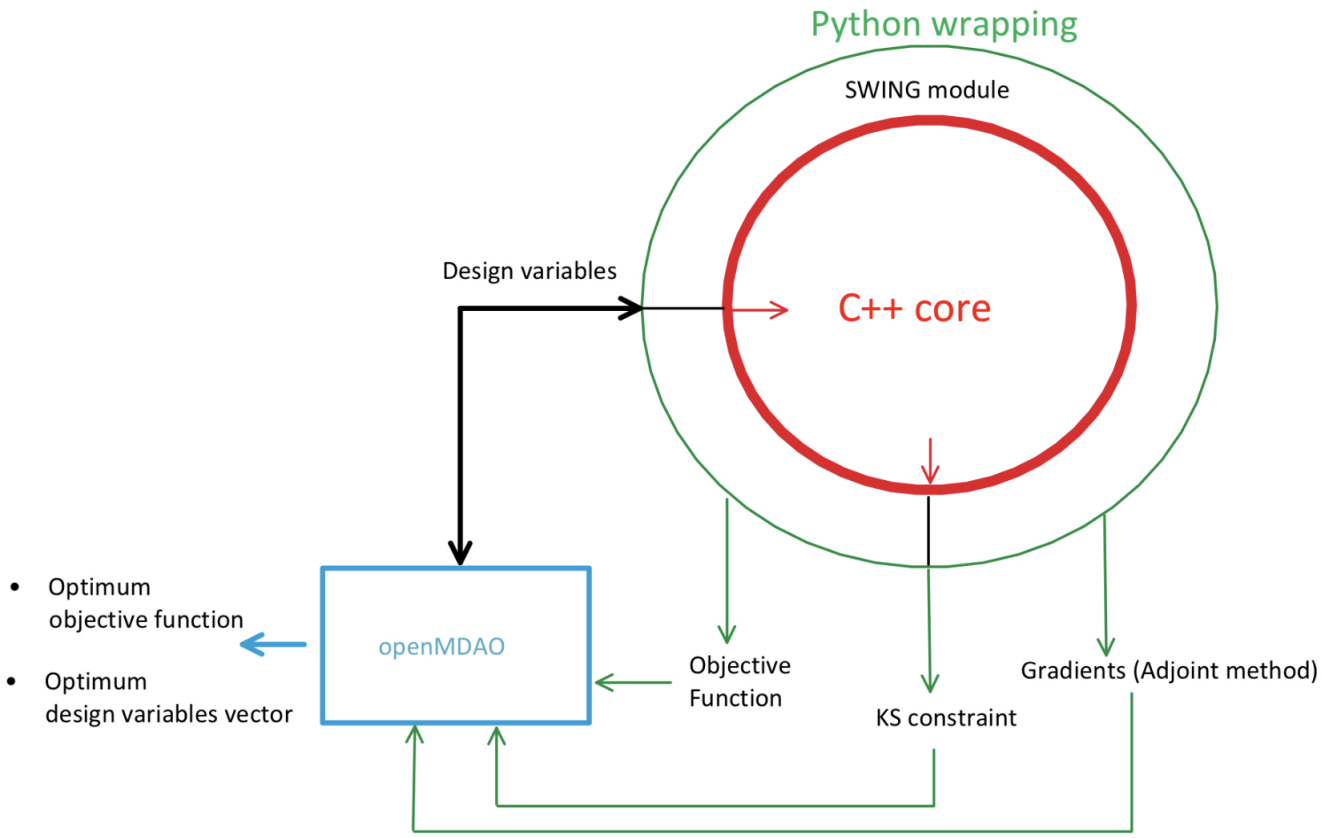


FIGURE 5.1. Optimization tool layout

5.1 C++ core functions

Lets start from the internal functions added to pyBeam based on the theory of chapter 3. To understand their location , can be helpful to take a look at figure 2.5. Core functions are presented in appx. A

1. The functions written for the calculation of section inertia, receive in input the properties from an external inputs file . They are allocated in the class "property" and described in appx. A.1. The output are the following :

   Iyy,Izz,Jt : inertia of the whole section.

   Iyy_b,Izz_b : inertia of concentrated areas.

   A : Area of the whole section.

2. For the stress state retrieving, the written functions is shown in appx. A.2. It is placed in the class "element" and returns

   - τ **: shear stress absorbed by panels**
   - σ_{booms} **: normal stress absorbed by concentrated areas**

3. Output of function A.2 are used in A.3. The latter is necessary for the calculation of **individual stresses constraint**, using the Von Mises criterion. It is defined as member function of class "element".

4. In appx A.4 is reported the function for the calculation of **buckling constraint** to impose on each FE. It is a member of class "element" too.

5. **KS constraint for the stresses** is calculated in the function reported in appx. A.5, exploiting the outputs of A.3. It is defined as a function of class "structure" .

6. **KS constraint for the buckling** is returned by the function in appx. A.6, making use of A.4 output. It is allocated as member function of class "structure" .

7. In appx. A.7 is shown the function for the calculation of **weight** : it uses design variables from input file, total section area from A.1, finite elements length and their number. It belongs to the class "structure" .

5.1.1 Core "main" file

Main C++ class is named "beam", here are allocated the most important functions to pass to the Python wrapping.

```
1 void Solve(int FSIIter);
2 void SolveLin(int FSIIter);
3 passivedouble EvalWeight();
4 passivedouble EvalKSStress();
5 passivedouble EvalKSBuckling();
6 void ComputeAdjointWeight(void);
7 void ComputeAdjointKSstresses(void);
8 void ComputeAdjointKSbuckling(void);
9 void ReadRestart();
10 void RunRestart();
11 void RunRestart_lin();
```

LISTING 5.1. Several member functions of the class "beam"

Functions above are interesting for our purpose :

- "Solve" and "SolveLin" run non-linear and linear solver,
- "EvalWeight" , "EvalKSStress" and "EvalKSBuckling" simply recall respectively functions in appx. A.7 , A.5 and A.6
- "WriteRestart" writes in a text file the solution of the primal problems (displacements) .
- "ReadRestar" is able to reads the solution text file, passing it to the functions "RunRestart" and "RunRestart_lin".

- "RunRestart" and "RunRestart_lin" , restart the solving sequence (linear or non-linear) starting from the solution of the primal. Restart sequence is necessary Adjoint calculation.

- "ComputeAdjointWeight" , "ComputeAdjointKSstresses" and "ComputeAdjointKSbuckling" compute Objective function and constraint gradients with Adjoint method, using a solver implemented in the Automatic Differentiation (AD) library CodiPack(Code Differentiation Package).

5.2 Python functions

- **Python wrapping functions** :

  Through SWING is possible to import header file from pyBeam C++ core to a Python Library. Here Wrapping functions have been implemented as members of :

```
1 class pyBeamSolver:
2 class pyBeamSolverAD:
```

LISTING 5.2. "Wrapping Class"

  They are described in Appx. B.1.

- **OpenMDAO functions**:
  To facilitate the interaction with openMDAO have been written functions, belonging to "Class PyBeamOpt", and reported in appx. B.2. These relies on that ones described in appx. B.1 and are able to returns the values of objective function and constraints for a linear and nonlinear analysis.

- **Main optimization script:**
  Finally we have the main openMDAO script, which should be run from terminal with Python 3 , in order to get the results of optimization (appx. B.3). Has been following precise logic, figure 4.3, exploiting functions written in opnemdao.api library. We have proceeded by taking following steps :

1. Build the Problem .
2. Initialize class "PyBeamOpt".
3. Add to the Problem the Components, exploiting the functions in appx. B.2.
4. Setup the optimization, specifying the Optimizer, SLSQP, and parameters should be displayed each iteration.
5. Set the Recorder , to keep track of the values of objective function , constraint and their gradients.
6. Attribute the initial value of the design variables.
7. set the function which execute the Driver.
8. Get the graphic and save values of constraint , design variables , objective functions and derivatives in folders created each iteration.

Chapter 6

Results and Validations

This final chapter is intended to analyze results obtained from linear and non linear structural optimization of different test cases; moreover, in order to validate what has been implemented, analytical calculations has been performed. Firstly, nonlinearities are not taken into account; in this phase a simple example is analyzed (table 6.1) , then the optimizer is validated showing the adherence to the KKT (Karush-Kuhn-Tucker) conditions , namely , the condition necessary and sufficient for a global minimum. Finally ONERA M6 wing is investigated.

Remembering that the problem to solve is 4.16, for validating the optimization , the following testcase will be treated, launching just the linear analysis and without considering the KS Buckling constraint :

LONG BEAM (150 m)						
property:						
$C_{wb}[mm]$	$h[mm]$	$t_{sk}[mm]$	$t_{sp}[mm]$	$A_{fl}[mm^2]$	n_{stiff}	$A_{stiff}[mm^2]$
3000	500	3	5	200	0/8	50
loads:						
N [N]	T_y [N]	T_z [N]				
100	5000	5000				
mesh:						
Nº of nodes	Nº of element	constrained nodes	Nº of rigid element			
61	60	first node (clamped)	0			
config Linear :						
E [Mpa]	ν	ρ [Kg/mm^3]	s			
70000	0.3	2700e-9				
config NonLinear :						
E [Mpa]	ν	ρ [Kg/mm^3]	Nº of Struct. iter.	Nº of Load steps		
70000	0.3	2700e-9	10	15		

TABLE 6.1. Long beam. Input Data for a wing box with 0/8 stringers

Initial values , obtained from the structural linear analysis are the following :

LONG BEAM (150 m)		
Linear :		
	Weight $[Kg]$	$KS - stress$
0 stiff.	9.639	12.90460884787548
8 stiff.	9.801	8.380501388537933

TABLE 6.2. Long beam. Initial values for a wing box with $0/8$ stringers (linear)

Once validated the optimizer with the previous example , it is helpful to analyze a real case scenario:

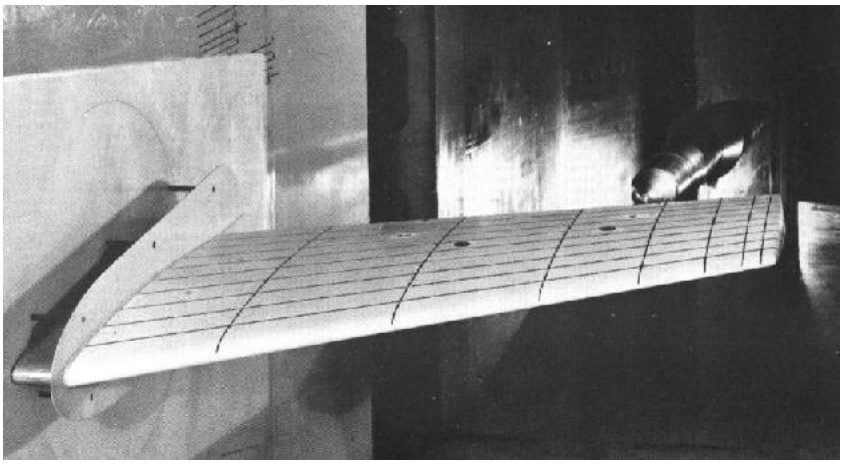


FIGURE 6.1. ONERA M6 [4]

The ONERA M6 wing is a swept, semi-span wing with no twist. The airfoil and its wing box properties are reported below

ONERA M6		
(SA) Swept angle [rad]	$\frac{\pi}{6}$	
	Tip	Root
C [mm]	453.33	805.9
T_C	0.10	0.10
xFS	0.15	0.15
xRS	0.65	0.65
zS	0.85	0.85
t_{sk} [mm]	1	1
t_{sp} [mm]	1	1
A_{fl} $[mm^2]$	100	100

TABLE 6.3. Airfoil and wing box geometry.

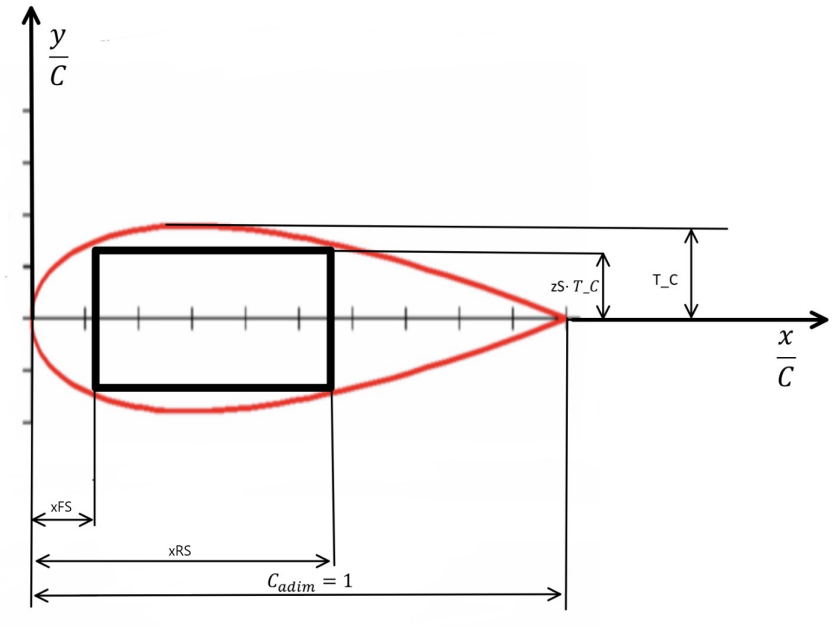


FIGURE 6.2. Airfoil and wing box geometry

Knowing the wing properties from the previous table , is possible to build a beam model , considering as design variables the ones given by the mean between tip and root characteristics :

$$C_{wb} = \left(\frac{C_{tip} + C_{root}}{2}\right)(xRS - xFS)cos30$$
$$h = Zs \cdot T\_C \cdot \left(\frac{C_{tip} + C_{root}}{2}\right)$$

Afterwards, is useful to analyze the same ONERA M6 testcase but considering more than one property since an actual wing is characterized by different sections along the span. A linear interpolation has been performed using Matlab, starting from the data in 6.3

$$C = \frac{X_{spanwise}}{L} \cdot (Ctip - Croot) + Croot)$$
$$Cwb = C \cdot (xRS - xFS) \cdot cos(SA)$$
$$h = C \cdot zS \cdot T_C$$

ONERAM M6							
property:							
	$C_{wb}[mm]$	$h[mm]$	$t_{sk}[mm]$	$t_{sp}[mm]$	$A_{fl}[mm^2]$	n_{stiff}	$A_{stiff}[mm^2]$
	272.5598	53.517	1	1	100	0/8	25
loads:							
	N [N]	T_y [N]	T_z [N]				
	1000	50000	50000				
mesh:							
	Nº of nodes	Nº of element	constrained nodes	Nº of rigid element			
	20	19	first node (clamped)	0			
config Linear :							
	E [Mpa]	ν	ρ [Kg/mm^3]				
	70000	0.3	2700e-9				
config NonLinear :							
	E [Mpa]	ν	ρ [Kg/mm^3]	Nº of Struct. iter.	Nº of Load steps		
	70000	0.3	2700e-9	10	15		
Initial Objective function and Constraint :							
	Weight $[Kg]$	$KSstress_{lin}$	$KSstress_{nonlin}$				
0 stiff.	1.5616098952142794	8.590784838904824	8.522129451466077				
8 stiff.	1.85841009521428	5.493365398603632	5.460027276767425				

TABLE 6.4. ONERAM M6. Initial design parameters (no buckling)

```
1
2  Cwb,h,tsk,tsp,Afl,nstiff,Astiff
3
4 346.432371330250  68.0043685773346  1 1 100 8 25
5 341.367331503554  67.0101057320038  1 1 100 8 25
6 336.303167072611  66.0160147260863  1 1 100 8 25
7 331.239002641668  65.0219237201688  1 1 100 8 25
8 326.173962814972  64.0276608748380  1 1 100 8 25
9 321.108922988276  63.0333980295072  1 1 100 8 25
10 316.044758557333  62.0393070235897  1 1 100 8 25
11 310.980594126390  61.0452160176722  1 1 100 8 25
12 305.915554299695  60.0509531723414  1 1 100 8 25
13 300.850514472999  59.0566903270106  1 1 100 8 25
14 295.785474646303  58.0624274816798  1 1 100 8 25
15 290.721310215360  57.0683364757623  1 1 100 8 25
16 285.657145784417  56.0742454698448  1 1 100 8 25
17 280.592105957721  55.0799826245140  1 1 100 8 25
18 275.527066131025  54.0857197791833  1 1 100 8 25
19 270.462026304329  53.0914569338525  1 1 100 8 25
20 265.397861873386  52.0973659279350  1 1 100 8 25
21 260.333697442443  51.1032749220175  1 1 100 8 25
22 255.268657615747  50.1090120766867  1 1 100 8 25
23 ----------------------------------------------------------------
24 N    Ty   Tz
25
26 100 5000 5000
27 ----------------------------------------------------------------
28 Weight [Kg]          KSstress-nonlin              KSbuckl-nonlin
29
30 1.858409666        -0.38923213272330087        0.31830662458817005
```

LISTING 6.1. ONERA M6. Initial design parameters considering 19 different sections

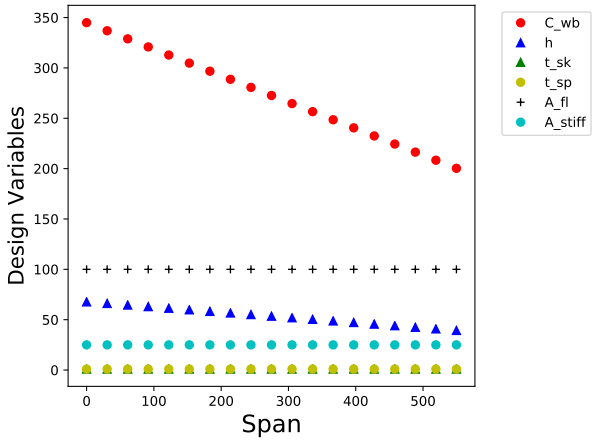


FIGURE 6.3. initial design variables distribution along the span

in this case , buckling is taken into account but external loads should be modified : if they are too high numerical problems, for the calculation of KS-Buckling constraint, can occur. Just nonlinear analysis is performed.

First of all , calculation of gradient through Adjoint should be validated. Running the primal problem , using functions described in appx. B.2, is retrieved the gradient of KS constraints and Weight w.r.t. design variables, using Adjoint. Than , they have been compared to the results obtained using central finite differences

$$\begin{aligned} \frac{\partial f(\mathbf{x})}{\partial x_1} &= \frac{f(x_1+\delta,[x_2,\ldots,x_n]) - f(x_1-\delta,[x_2,\ldots,x_n])}{2||\delta||} \\ \vdots & \qquad\qquad \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n}) &= \frac{f(x_n+\delta,[x_1,\ldots,x_{n-1}]) - f(x_n-\delta,[x_1,\ldots,x_{n-1}])}{2||\delta||} \end{aligned} \tag{6.1}$$

An example is given below, here the results for the validation of KS-stress sensitivities in linear and nonlinear case for the testacase in table 6.2.

```
----------  LINEAR CASE   --------------------
####VALIDATION of KS-stress without stringers

AD =  -0.0006614637979221289    FD =  -0.0006614448281716534
diff% =  -0.0028678440959441056

AD =  -0.02381269667492503    FD =  -0.023808012766401987
diff% =  -0.019669794593132194

AD =  1.1727090632918934e-09    FD =  -1.0139306283463156e-08
diff% =  111.56596941158134

```

```
AD =  -1.5630725881176689e-10    FD =  -1.639509861206534e-10
diff% =  4.662202704448146

AD =  -0.06945569976422845    FD =  -0.06945154852147084
diff% =  -0.005976820868129058

AD =  0.0    FD =  0.0    diff% =  0.0


####VALIDATION of KS-stress with 8 stringers

AD =  -0.00024052952643572467    FD =  -0.0002406446393443673
diff% =  0.04783522664633569

AD =  -0.01587494875839842    FD =  -0.015871808921197328
diff% =  -0.019778565895724937

AD =  8.147822218637321e-10    FD =  -4.28111857075919e-09
diff% =  119.03199382116725

AD =  -9.461272455957214e-13    FD =  -6.164373012040869e-10
diff% =  99.84651687304651

AD =  -0.029737306681525525    FD =  -0.029736933213353467
diff% =  -0.0012558910464145192

AD =  -0.05422669641359451    FD =  -0.054223943994191615
diff% =  -0.005075764494121074

----------  NONLINEAR CASE   --------------------

####VALIDATION of KS-stress with 8 stringers (10 iteratioions adjoint)
AD =  0.00014362938153240 91    FD =  0.00014362336063555858
diff% =  0.004191967399893084

AD =  -0.009207870332321035    FD =  -0.009207915128488509
diff% =  0.00048649631157827257

AD =  0.36294781454246583    FD =  0.3629355015899449
diff% =  0.003392485648778898

AD =  0.020040657484314738    FD =  0.020040754247219184
diff% =  -0.0004828306522416301

AD =  -0.026832883424874106    FD =  -0.02683195336823374
diff% =  -0.0034661077068725544

AD =  -0.048865242859052846    FD =  -0.04886204429261909
diff% =  -0.006545688195969668
```

LISTING 6.2. Validation Ks sensitivities using Adjoint method

6.1 Long beam.One property wing box with 0 and 8 stringers (linear analysis) : results,considerations and validation

6.1.1 Long beam. wing box 0 stringers

It is studied the testcase whose data are reported in table 6.1. Since exist different combinations of design variables from which you can obtain a certain weight reduction, some limitations should be introduced. If this addition is not considered , can be returned no sense results , such as a wing box height bigger than the chord. For this reason, an extra constraint is given :

$$g1 = h - 0{,}2 \cdot Cwb \tag{6.2}$$

this means that the height should be always the 20 % of wing box chord.

- **Finite Differences**:

```
1
2 Optimization Problem -- Optimization using pyOpt_sparse
3 ================================================
4     Objective Function: _objfunc
5
6     Solution:
7 ------------------------------------------------
8     Total Time:                    18.0996
9        User Objective Time :        5.1004
10       User Sensitivity Time :     12.9085
11       Interface Time :             0.0850
12       Opt Solver Time:             0.0057
13    Calls to Objective Function :     101
14    Calls to Sens Function :           31
15
16
17   Objectives
18      Index  Name                          Value          Optimum
19          0  weight_comp.Obj_f     6.793805E+03     0.000000E+00
20
21   Variables (c - continuous, i - integer, d - discrete)
22      Index  Name     Type      Lower Bound            Value
23          0  C_wb_0      c     1.000000E+00     3.477780E+03
24          1  h_0         c     1.000000E+00     6.955559E+02
25          2  t_sk_0      c     1.000000E+00     9.995473E-01
26          3  t_sp_0      c     1.000000E+00     9.998855E-01
27          4  A_fl_0      c     1.000000E+00     2.107866E+03
```

```
28
29    Constraints (i - inequality, e - equality)
30       Index  Name              Type          Lower           Value
31           0  Cwb_h_comp.g1        e   0.000000E+00    0.000000E+00
32           1  KS_comp.Const_KS     i  -1.000000E+30   -1.459895E-06
33
34 ---------------------------------------------------
```

LISTING 6.3. Long beam. Results of a wing box optimization with 0 stringers using finite differences

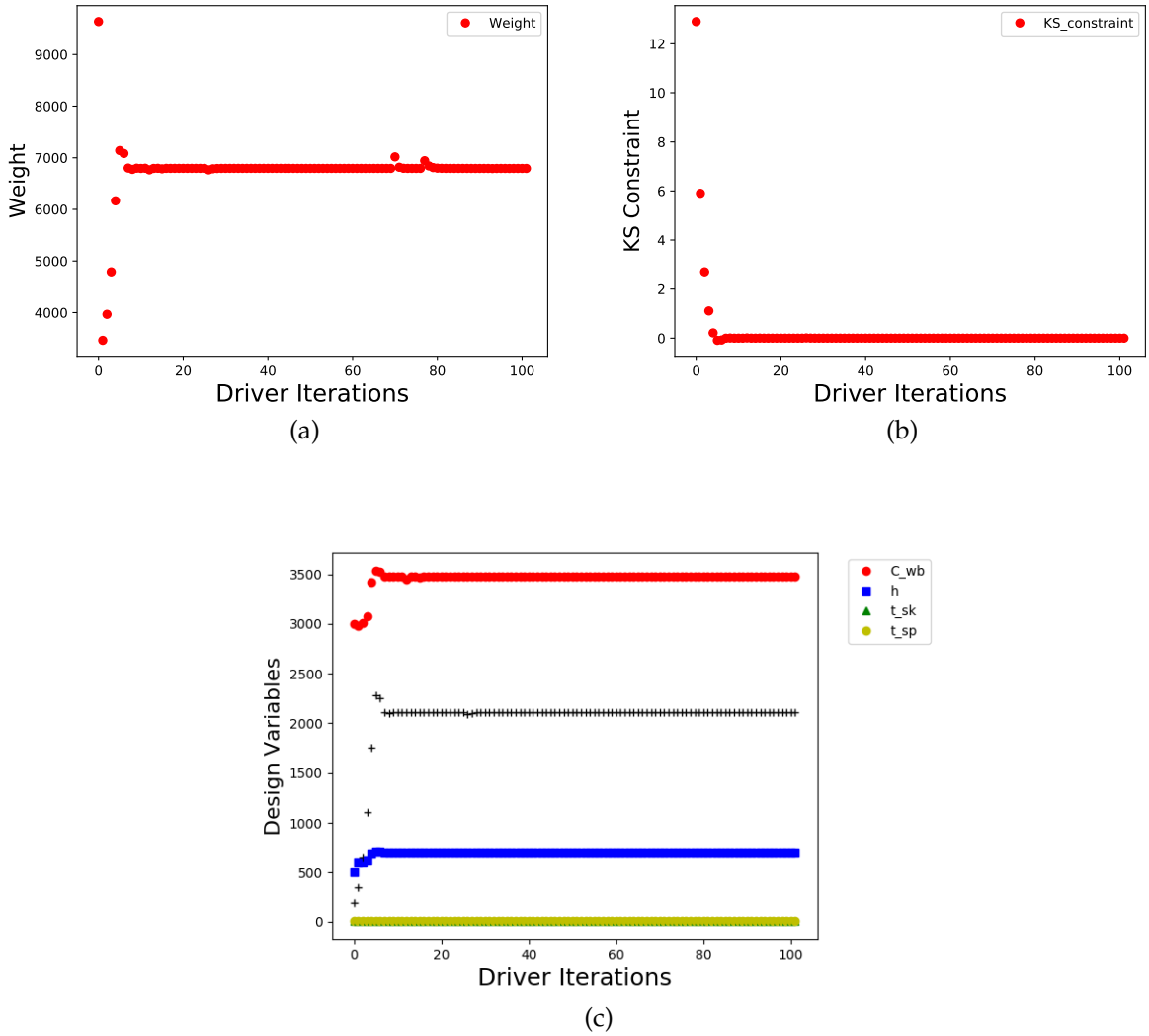


FIGURE 6.4. Long beam. wing box , with 0 stringers, linear optimization : design variables, KS-stress constraint and Weight using finite differences

- **Adjoint method**

```
1
2 Optimization Problem -- Optimization using pyOpt_sparse
3 ================================================
4     Objective Function: _objfunc
5
6     Solution:
7 ------------------------------------------------
8     Total Time:                    2.0110
9        User Objective Time :       0.7442
10       User Sensitivity Time :     1.2405
11       Interface Time :            0.0242
12       Opt Solver Time:            0.0021
13    Calls to Objective Function :      16
14    Calls to Sens Function :           16
15
16
17   Objectives
18      Index  Name                          Value          Optimum
19          0  weight_comp.Obj_f     6.795050E+03     0.000000E+00
20
21   Variables (c - continuous, i - integer, d - discrete)
22      Index  Name     Type      Lower Bound            Value
23          0  C_wb_0      c     1.000000E+00     3.495399E+03
24          1  h_0         c     1.000000E+00     6.990798E+02
25          2  t_sk_0      c     1.000000E+00     9.999987E-01
26          3  t_sp_0      c     1.000000E+00     9.999997E-01
27          4  A_fl_0      c     1.000000E+00     2.097238E+03
28
29   Constraints (i - inequality, e - equality)
30      Index  Name             Type          Lower            Value
31          0  Cwb_h_comp.g1       e   0.000000E+00    1.136868E-13
32          1  KS_comp.Const_KS    i  -1.000000E+30   -2.616142E-07
33
34 ------------------------------------------------------------
```

LISTING 6.4. Long beam. Results of a wing box optimization with 0 stringers using Adjoint

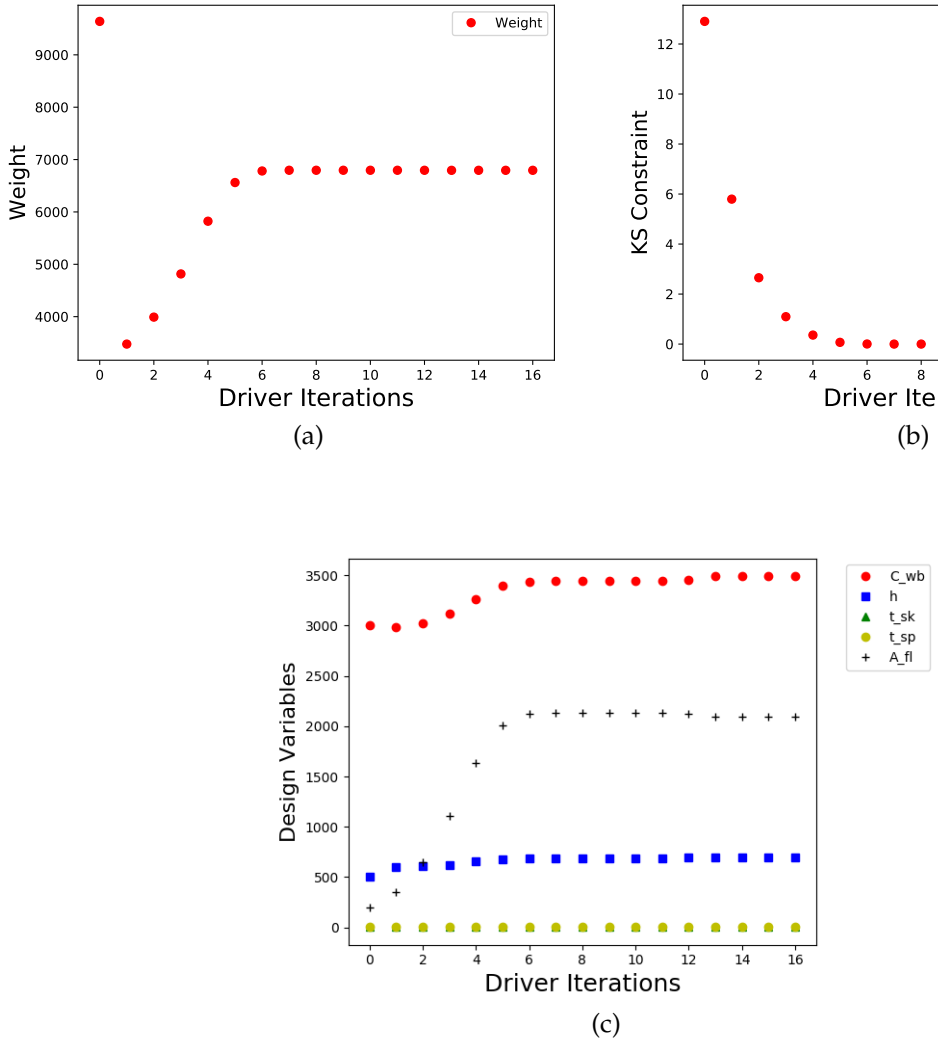


FIGURE 6.5. Long beam. wing box , with 0 stringers, linear optimization : design variables, KS-stress constraint and Weight using Adjoint method

6.1.2 Long beam.wing box 8 stringers

In this case , if boundaries are no imposed , it happens that for keeping adherence to KS-stress constraint, just A_{fl} increases dizzily reaching very high value , while A_{stiff} approach zero value since all work is done by A_{fl}.But this does not make sense. For this reason another constraint has been taken into account, in addition to the one considered before :

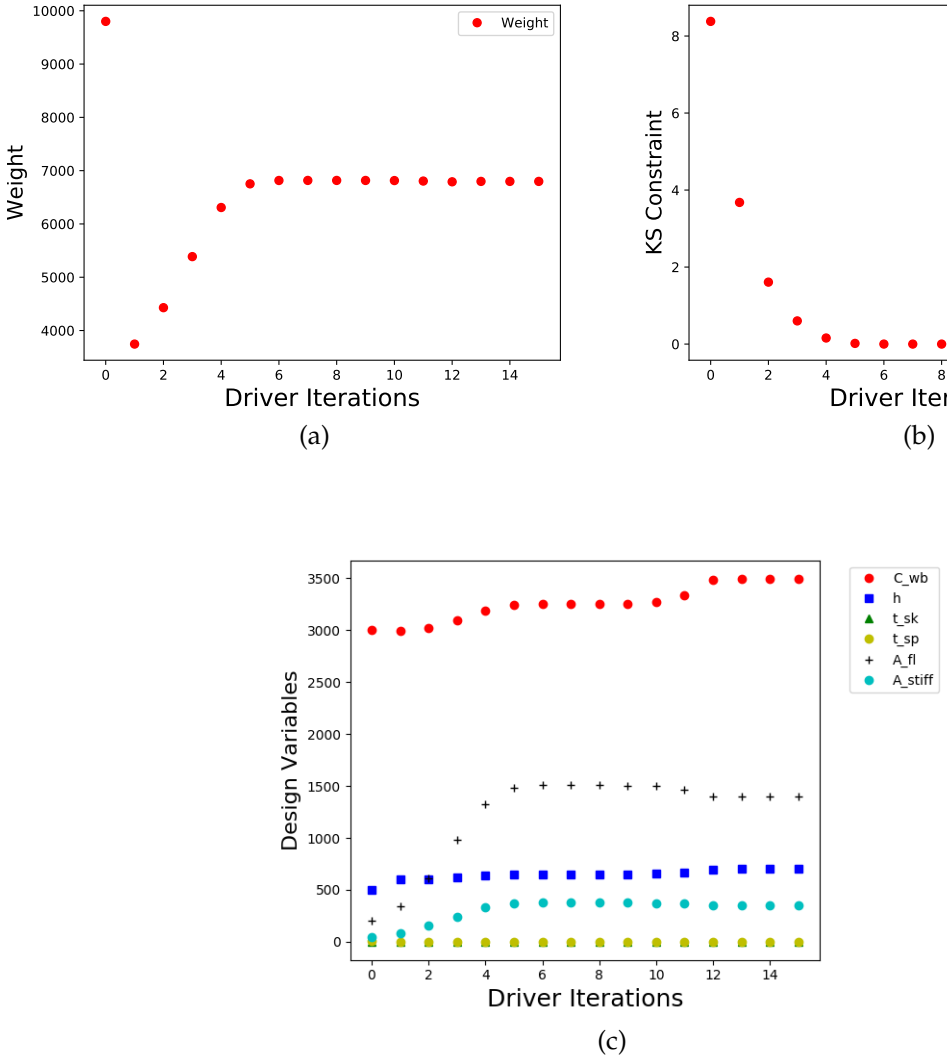
$$g2 = A_{stiff} - 0{,}25 \cdot A_{fl} \tag{6.3}$$

the stringers area must be the 25 % of the caps one.

Since the calculation using finite differences starts to become long and inaccurate, just the analysis with Adjoint method is performed .

```
1
2 Optimization Problem -- Optimization using pyOpt_sparse
3 ================================================
4     Objective Function: _objfunc
5
6     Solution:
7 ------------------------------------------------
8     Total Time:                    2.3916
9        User Objective Time :       0.9140
10       User Sensitivity Time :     1.4436
11       Interface Time :            0.0318
12       Opt Solver Time:            0.0022
13    Calls to Objective Function :      15
14    Calls to Sens Function :           15
15
16
17   Objectives
18      Index  Name                         Value          Optimum
19          0  weight_comp.Obj_f     6.798290E+03     0.000000E+00
20
21   Variables (c - continuous, i - integer, d - discrete)
22      Index  Name        Type      Lower Bound            Value
23          0  C_wb_0         c     1.000000E+00     3.496963E+03
24          1  h_0            c     1.000000E+00     6.993926E+02
25          2  t_sk_0         c     1.000000E+00     9.999994E-01
26          3  t_sp_0         c     1.000000E+00     9.999999E-01
27          4  A_fl_0         c     1.000000E+00     1.398866E+03
28          5  A_stiff_0      c     1.000000E+00     3.497164E+02
29
30   Constraints (i - inequality, e - equality)
31      Index  Name               Type          Lower           Value
32          0  Cwb_h_comp.g1         e   0.000000E+00    0.000000E+00
33          1  Astiff_Afl_comp.g2    e   0.000000E+00    0.000000E+00
34          2  KS_comp.Const_KS      i  -1.000000E+30    1.234835E-08
```

LISTING 6.5. Long beam. Results of a wing box optimization with 8 stringers using Adjoint



(a) (b) (c)

FIGURE 6.6. Long beam. wing box , with 8 stringers, linear optimization : design variables, KS-stress constraint and Weight using Adjoint method

6.1.3 Considerations (Long beam)

Referring to initial data in tab. 6.1 , results of previous simulations, are summarized in the tables below :

Linear optimization results (o stiff.)								
	Cwb [mm]	h	t_{sk}[mm]	t_{sp} [mm]	$A_{fl}[mm^2]$	Weight [Kg]	KS-stress	tot time [s]
AD	3.495399E+03	6.990798E+02	1	1	2.097238E+03	6.795050E+03	-2.616142E-07	2.0110
FD	3.477780E+03	6.955559E+02	1	1	2.107866E+03	6.793805E+03	-1.459895E-06	18.0996
diff	3.5 %	3.57 %	0	0	3.45 %	0.00756%		

TABLE 6.5. Long beam . Summary of optimization results for a wing box with 0 stringers , using finite differences and Adjoint method

A first consideration can be done comparing the two methods : both give similar results in terms of optimum weight, design variables and KS-stress final values. Nevertheless, the computational cost using finite differences is huge , in fact are done more objective function evaluations , 101, respect to the Adjoint method, 16 . It can be deduced also by the Total time to finalize the optimization: 2.0110 s for the Adjoint versus 18.0996 s . Moreover,for finite differences the progress of weight , design variables and KS-stress constraint in respect to the optimizer iterations is affected by numerical oscillation, while the Adjoint is more accurate.

Generally , looking at the design variables trend , can be appreciate how the caps area tends to increase to keep adherence to the KS-stress constraint, which is reaching its minimum value. Thicknesses approach the lower fixed bound of 1 , cause the constraint acts more on normal stresses at which concentrated areas must withstand , than on shear stresses, in particular for a long beam and for not to high external loads. Since normal stresses does not depend on t_{sk} and t_{sp}, the constraint is not active on these design variables.

For 0 stringers wing box, **weight has been reduced approximately of a 30 %**

Linear optimization results (8 stiff.)									
	Cwb [mm]	h	t_{sk}[mm]	t_{sp} [mm]	$A_{fl}[mm^2]$	$A_{stiff}[mm^2]$	Weight [Kg]	KS-stress	tot time [s]
AD	3.496963E+03	6.993926E+02	1	1	1.398866E+03	3.497164E+02	6.798290E+03	7.302541E-08	2.3916

TABLE 6.6. Long beam. Summary of optimization results of a wing box with 8 stringers , using Adjoint method

If wing box has 8 stringers , both A_{fl} and A_{stiff} contribute to the absorption of normal stress , as can be noticed from the Navier formula. For this reason respect to the previous case , the A_{fl} is lower. **The reduction in weight is of a 30.6 %**

6.1.4 Validation

To validate the results obtained , the optimum design variables vector , should satisfy the KKT conditions. Considering the general problem :

$$\begin{aligned} min\ f(\mathbf{x}) \\ with\ respect\ to\ \mathbf{x} \\ subjected\ to\ g(\mathbf{x}) \leq 0 \end{aligned} \tag{6.4}$$

Lagrangian function to be minimized is the following :

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda^T g(\mathbf{x}) \tag{6.5}$$

Theorem below is a condition necessary and sufficient for the existence of global minimum

Theorem: *is f a convex function , with ∇ f continuous in $\mathcal{R}^n$, a point x^* is of minimum , $\iff$ $\exists\ \lambda^* \in \mathcal{R}^m$ such that are respected following conditions :*

1. $g(\mathbf{x}^*) \leq 0$
2. $\nabla L(x^*, \lambda^*) = 0$ Stationary
3. $\lambda^* \geq 0$ dual feasibility
4. $\lambda^{*T} g(\mathbf{x}^*) = 0$ complementary slackness

To validate correctness of optimization , giving the Lagrange function to minimize , has been calculated the gradient $\nabla L(x, \lambda)$ using the symbolic tool in Matlab. Then , considering the results of openMDAO ($\mathbf{x}^*$), retrace steps from 4) to 1) , starting from the calculation of λ^* and verifying the adherence to the KKT conditions.

Alternatively, a simpler problem has been treated instead of 4.16 to facilitate calculations in Matlab. Lets consider the previous Long Beam , whose section is represented by a wing box without stringers imposing extra constraints 6.2.

The problem to solve is the following :

$$\begin{aligned} & min\ Weight(\mathbf{DVs}) \\ & with\ respect\ to\ \mathbf{DVs} \\ & subjected\ to\ g_{max}(\mathbf{DVs}) = \frac{\sigma_{max}(\mathbf{DVs})}{\sigma_{all}} - 1 \leq 0 \\ & g_{tsk} = 1 - t_{sk} \leq 0 \\ & g_{tsp} = 1 - t_{sp} \leq 0 \\ & g_{tsp} = -h + 0{,}2 * Cwb = 0 \end{aligned} \tag{6.6}$$

with $\mathbf{DVs} = [Cwb, h, t_{sk}, t_{sp}, A_{fl}]$.

Conceptually 6.6 - 4.16 are similar , since KS-stress value is really close to the maximum constraint g_{max}. The point is that in the expression of g_{max} , are not taken into account the other g_i constraints imposed on each "i" element. In linear case , the maximum constraint is definitely imposed at the root element, in particular on the up-left (or up right) flange , cause for the Navier formula (eq 3.21) here is absorbed the higher normal stress (σ_{max}).

Running the optimization, openMDAO provides te following results

Adjoint :				
Cwb [mm]	h	t_{sk}[mm]	t_{sp} [mm]	$A_{fl}[mm^2]$
3400.58076048	680.1161521	1	1	1942.07691444

TABLE 6.7. Optimum Point $\mathbf{DV}^*$ using Adjoint method (linear analysis)

We want to verify $\mathbf{DVs}^*$ to be the global minimum.

```
1
2  syms tsk tsp Afl N My Mz h Cwb Lbd1 Lbd2 Lbd3 Lbd4
3
4  %Beam Property
5  rho = 2700e-9;
6  L=150000;
7
8  %external Loads : N=100, Ty =5005 , Tz=5000
9
10 Mz = 743750000.31892622; % Maximum Mz at the root
11 My=-743750000.00332570;  % Maximum My at the root
12 N=100;
13
14
15 A = 2*((Cwb+tsp)*tsk)  + 2*((tsp*(h-tsk))) +4*Afl; %Total Area
16 weight = rho*L*A;   % obj Function
17
18 sigma_amm = 312.33333333333333333333333333333; % Allowable sigma
19 sigma_max=(N/(4*Afl)) - (My*(h/2)/(4*Afl*(h/2)^2)) + (Mz*(Cwb/2)/(4*Afl*(
       Cwb/2)^2));   % sigma max (Navier Formula)
20
21 g1 = (sigma_max/sigma_amm) - 1; % constraint g<=0
22 g2 = 1 - tsk;
23 g3 = 1 - tsp;
24 g4 = -h + 0.2*Cwb;
25
26 %Lagrangian function to be minimized
27 Lagr = weight + Lbd1*g1 + Lbd2*g2 + Lbd3*g3 + Lbd4*g4;
28
29 %Gradient
30 dL_dtsk = diff(Lagr,tsk);
31 dL_dtsp = diff (Lagr,tsp);
32 dL_dAfl = diff (Lagr,Afl);
33 dL_dh = diff (Lagr,h);
34 dL_dCwb = diff (Lagr,Cwb);
35
36
37 %openMDAO results
38
39 Cwb       =   3400.58076048;
40 h         =    680.1161521 ;
41 Afl       =    1942.07691444 ;
42 tsk=1;
43 tsp=1;
44
45 % complementary slackness
46 eq1 = Lbd1*double(subs(g1))==0;
47 Solve_Lbd1 = solve(eq1,Lbd1);
48 eq4 = Lbd4*double(subs(g4))==0;
49 Solve_Lbd4 = solve(eq4,Lbd4);
50
51 % dual feasibility check for Lbd1
52 Ldbd1=double(Solve_Lbd1)
53 Ldbd4=double(Solve_Lbd4)
54
```

```
55
56 %equations to finde Lbd2 and Lbd3
57 eq2= subs(dL_dtsk)==0;
58 eq3= subs(dL_dtsp)==0;
59
60 Solve_Lbd2_Lbd3 =solve([eq2 , eq3],[Lbd2,Lbd3])
61
62 % dual feasibility check for Lbd2 and Lbd3
63
64 Lbd2 = double(subs((Solve_Lbd2_Lbd3.Lbd2)))
65 Lbd3 = double(subs((Solve_Lbd2_Lbd3.Lbd3)))
66
67 %stationary
68
69 gradL_tsk=double(subs(dL_dtsk));
70 gradL_tsp=double(subs(dL_dtsk));
71 gradL_h=double(subs(dL_dtsk));
72 gradL_Cwb=double(subs(dL_dtsk));
73 gradL_Afl=double(subs(dL_dtsk));
```

LISTING 6.6. Matlab script for validation of linear structural optimization

Running the script above , the following results are obtained :

$$\lambda^* = \begin{bmatrix} 0 \\ 2{,}7545e+03 \\ 550,8941 \\ 0 \end{bmatrix} \Longrightarrow \nabla L(\mathbf{DVs}^*, \lambda^*) = 0$$

Since all conditions are respected , $\mathbf{DVs}^*$ is a global minimum and the correctness of the optimization tool, has been proved.

6.2 ONERA M6 : results and considerations

Since the finite differences have been used just for validation purpose , lets consider for ONERA M6 testcase Adjoint method for the gradient calculation. The input data of the testcase in exam, are reported in table 6.4. Taking into account that the wing box should respects some overall dimensions , dictated by the airfoil geometry (i.e. the maximum thickness) , h and C_{wb} must be fixed.

linear and non linear analysis of ONERA M6 provide results reported below.

6.2.1 ONERA M6. 1 property wing box with 0 and 8 stringers(linear)

- **0 stringers** :

```
1
2 Optimization Problem -- Optimization using pyOpt_sparse
3 ========================================
4     Objective Function: _objfunc
5
6     Solution:
7 ----------------------------------------
8     Total Time:                    1.0561
9        User Objective Time :       0.4717
10       User Sensitivity Time :     0.5645
11       Interface Time :            0.0182
12       Opt Solver Time:            0.0018
13    Calls to Objective Function :      15
14    Calls to Sens Function :           14
15
16
17
18 Design Vars
19 {'A_fl': array([886.3597089]),
20  't_sk': array([2.57083338]),
21  't_sp': array([10.55250325])}
22
23
24 Minimum Weight = [9.40664271]
25 Constraint = [9.39241875e-08]
```

LISTING 6.7. ONERA M6 . Results of a wing box optimization with 0 stringers using Adjoint (linear)

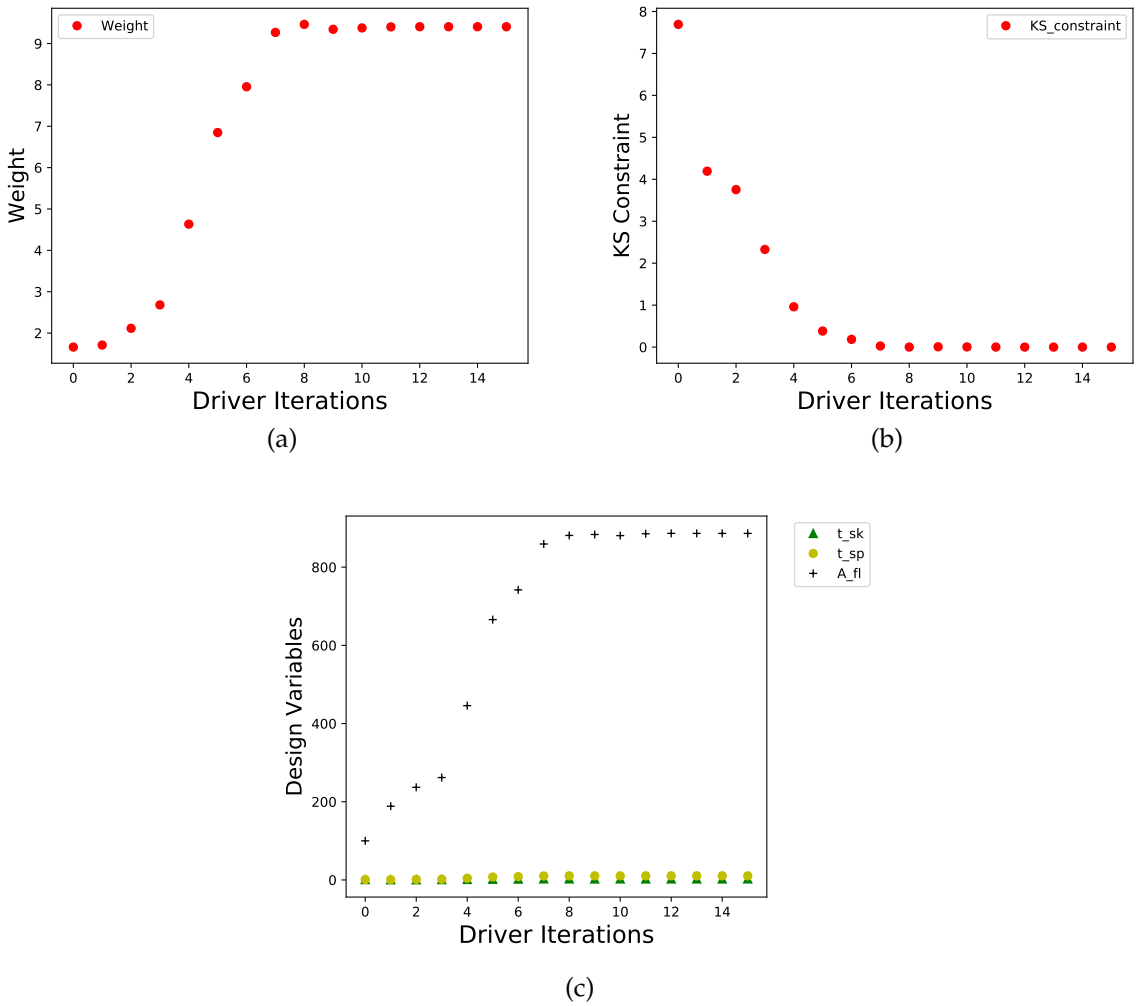


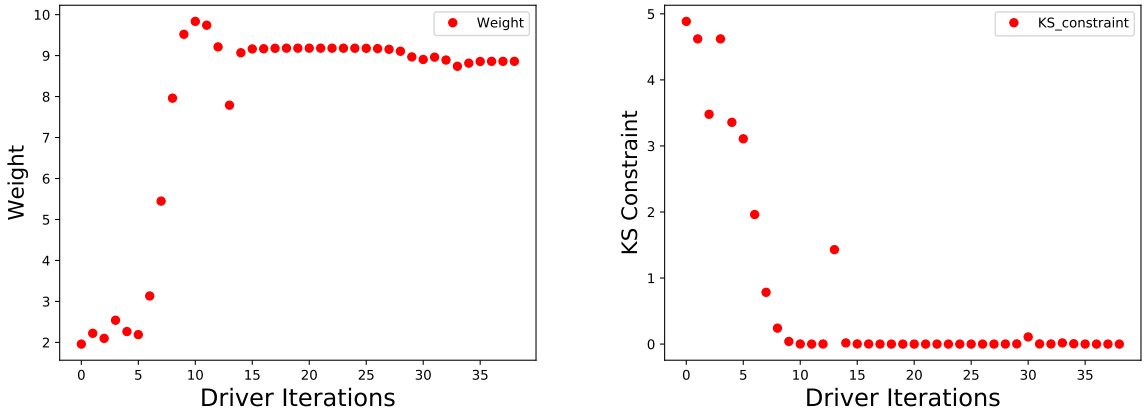
FIGURE 6.7. ONERA M6. wing box , with 0 stringers, linear optimization : design variables, KS-stress constraint and Weight using Adjoint method

- **8 stringers**

```
1
2
3  Optimization Problem -- Optimization using pyOpt_sparse
4  ==========================================
5      Objective Function: _objfunc
6
7      Solution:
8  ------------------------------------------
9      Total Time:                    2.8465
10        User Objective Time :       1.3382
11        User Sensitivity Time :     1.4497
12        Interface Time :            0.0534
13        Opt Solver Time:            0.0052
14     Calls to Objective Function :      38
15     Calls to Sens Function :           31
16
17
```

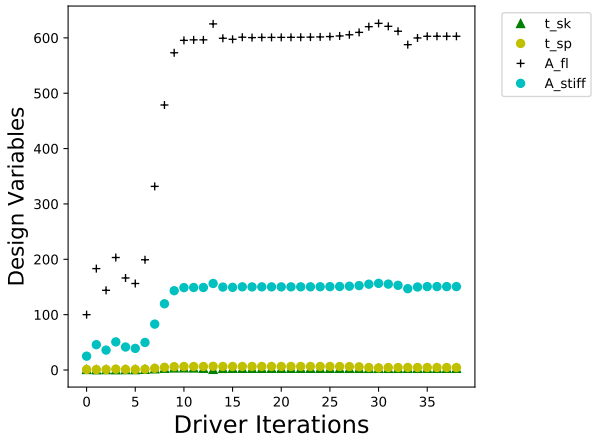
```
18
19      Total Time:                    1.9979
20         User Objective Time :       0.7789
21         User Sensitivity Time :     1.1707
22         Interface Time :            0.0452
23         Opt Solver Time:            0.0030
24      Calls to Objective Function :      26
25      Calls to Sens Function :           24
26
27
28 Design Vars
29 {'A_fl': array([602.88397845]),
30  'A_stiff': array([150.72099461]),
31  't_sk': array([3.02963283]),
32  't_sp': array([4.48725024])}
33
34
35 Minimum Weight = [8.85984415]
36 Constraint = [4.55368793e-08]
```

LISTING 6.8. ONERA M6 . Results of a wing box optimization with 8 stringers using Adjoint (linear)



(a)

(b)



(c)

FIGURE 6.8. ONERA M6. wing box , with 8 stringers, linear optimization : design variables, KS constraint and Weight using Adjoint method

6.2.2 ONERA M6. 1 property wing box with 0 and 8 stringers(Nonlinear)

- **0 stringers**

```
Optimization Problem -- Optimization using pyOpt_sparse
================================================
    Objective Function: _objfunc

    Solution:
------------------------------------------------
    Total Time:                    37.1308
       User Objective Time :       21.3338
       User Sensitivity Time :     15.7219
       Interface Time :             0.0692
       Opt Solver Time:             0.0059
    Calls to Objective Function :      60
    Calls to Sens Function :           41
```

```
14
15
16
17 Design Vars
18 {'A_fl': array([889.7984088]),
19  't_sk': array([2.58415232]),
20  't_sp': array([4.74529346])}
21
22 Minimum Weight = [8.42105775]
23 Constraint = [1.7307561e-08]
```

LISTING 6.9. ONERA M6. Results of a wing box optimization with 0 stringers (Nonlinear)

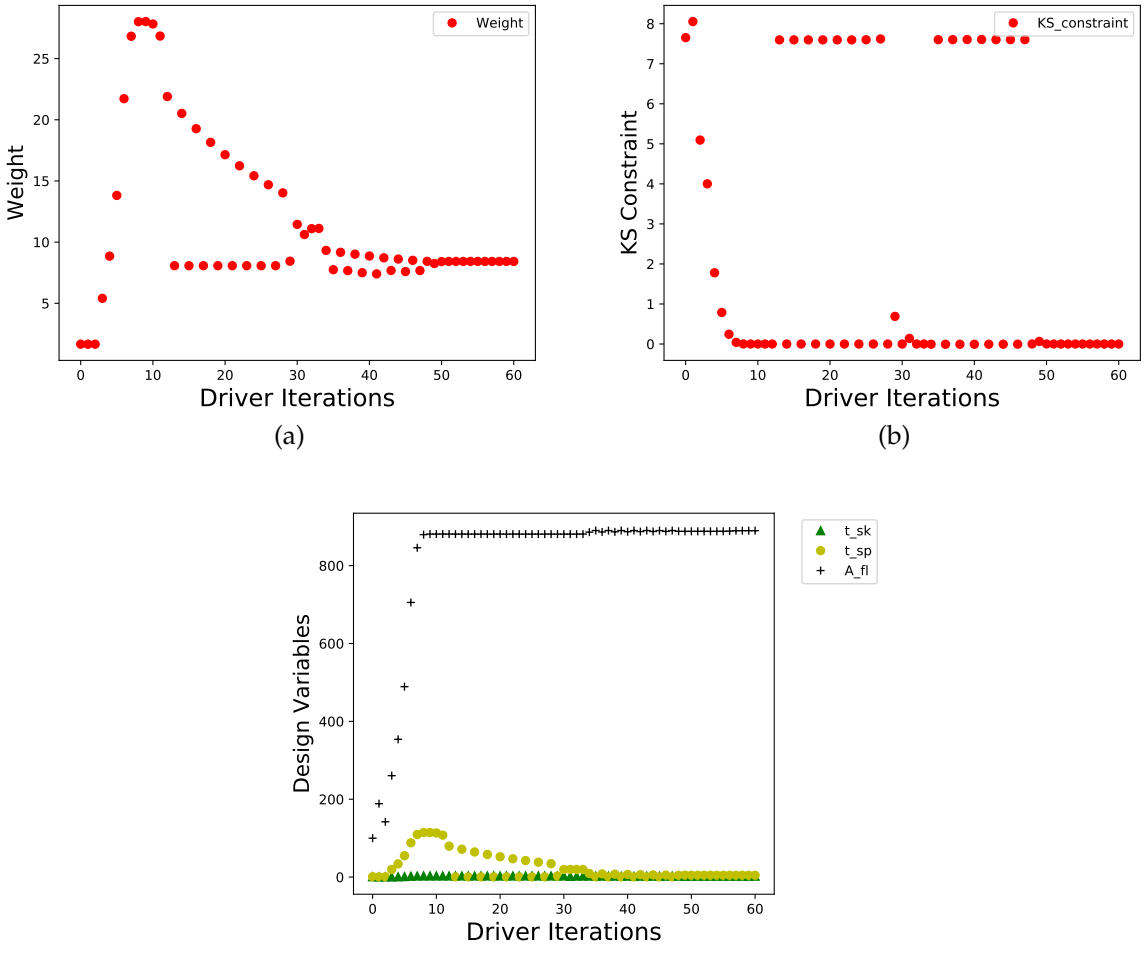


FIGURE 6.9. ONERA M6. wing box , with 0 stringers, nonlinear optimization : design variables, KS constraint and Weight using Adjoint method

- **8 stringers**

```
1
2
3 Optimization Problem -- Optimization using pyOpt_sparse
4 ============================================
5     Objective Function: _objfunc
6
7     Solution:
8 --------------------------------------------
9     Total Time:                   18.8544
10       User Objective Time :       9.6699
11       User Sensitivity Time :     9.1449
12       Interface Time :            0.0370
13       Opt Solver Time:            0.0025
14    Calls to Objective Function :      30
15    Calls to Sens Function :           26
16
17 Design Vars
18 {'A_fl': array([602.71340379]),
19  'A_stiff': array([150.67835095]),
20  't_sk': array([3.06369936]),
21  't_sp': array([4.51909071])}
22
23
24
25 Minimum Weight = [8.89432521]
26 Constraint = [1.40445641e-08]
27
28
29 ---------------------------------------------------
```

LISTING 6.10. ONERA M6. Results of a wing box optimization with 8 stringers (Nonlinear)

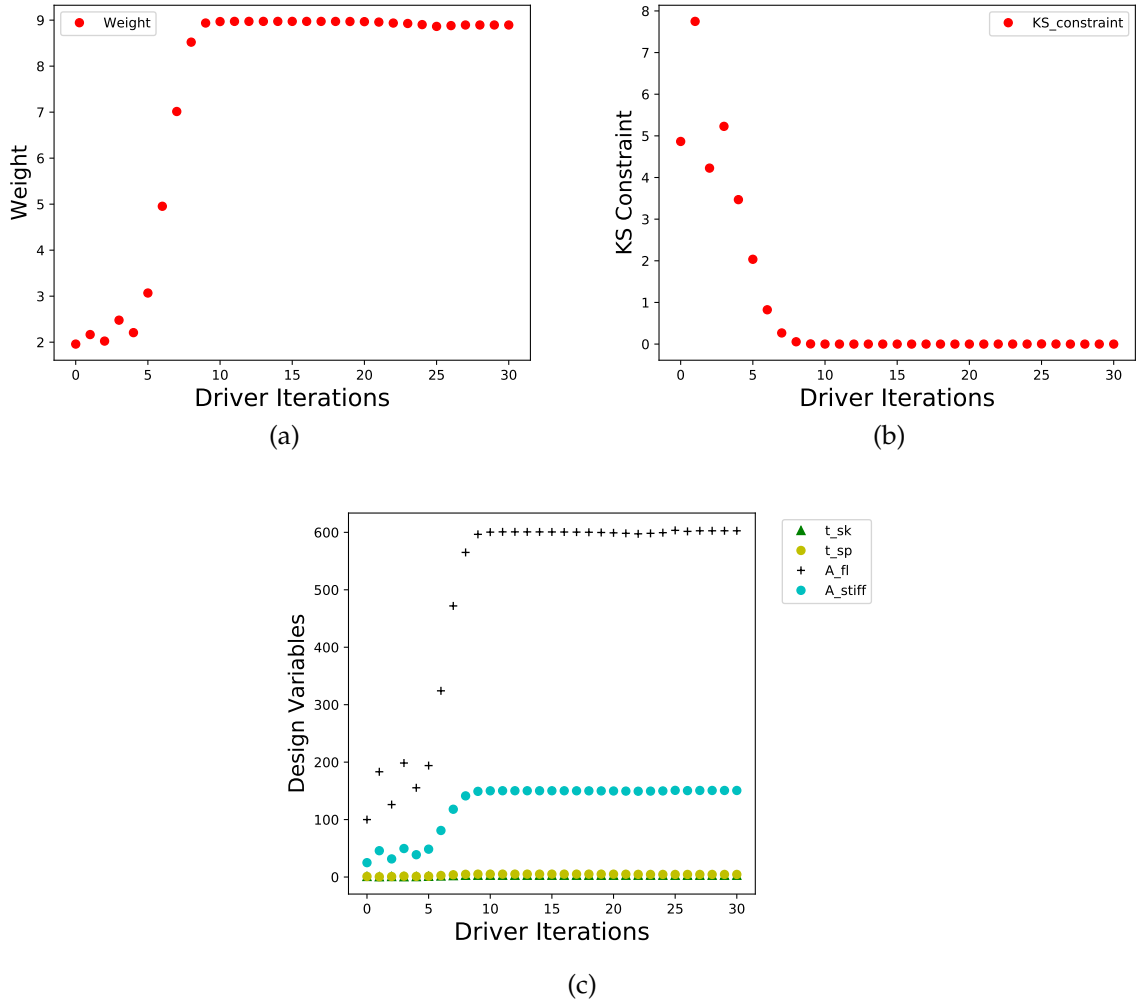


FIGURE 6.10. ONERA M6. wing box , with 8 stringers, nonlinear optimization : design variables, KS constraint and Weight using Adjoint method

1. First peculiarity is linked to the objective function value : fixing wing box height and chord, the structure can not withstand the high external loads with a lower weight than the initial one.

2. Looking at the results about design variables, can be seen how stringers,when they are present, and caps areas rise ; thicknesses grows to satisfy the KS-stress constraint. From the function A.2, can be noticed the reason for which the thicknesses rise : considering high external loads and a short beam , shear stresses become significant in the evaluation of internal stress state ;

Linear optimization results							
	t_{sk}[mm]	t_{sp} [mm]	$A_{fl}[mm^2]$	$A_{stiff}[mm^2]$	Weight [Kg]	KS	tot time [s]
0 stiff	2.57083338	10.55250325	886.3597089	/	9.40664271	9.39241875e-08	1.0561
8 stiff	3.02963283	4.48725024	602.88397845	150.72099461	8.85984415	4.55368793e-08	1.9979
Noninear optimization results							
	t_{sk}[mm]	t_{sp} [mm]	$A_{fl}[mm^2]$	$A_{stiff}[mm^2]$	Weight [Kg]	KS	tot time [s]
0 stiff	2.58415232	4.74529346	889.7984088	/	8.42105775	1.7307561e-08	37.1308
8 stiff	3.06369936	4.51909071	602.71340379	150.67835095	8.89432521	1.40445641e-08	18.8544

TABLE 6.8. ONERAM M6. Summary of optimization results for a wing box with 0/8 stringers , using Adjoint method (linear and nonlinear analysis)

1. Comparing the results considering 0 and 8 stringers , can be inferred from the data above that to accomplish the KS-stress constraint, A_{fl} rises , but for 0 stringers it reaches higher values cause the wing is less stiff than the case with 8 stringers; for the latter, the increasing of A_{stiff} allows A_{fl} to be lower since the absorption of normal stresses is distributed on more concentrated areas.

2. For nonlinear analysis , total time to compute the calculation is higher , this cause each optimizer iteration , an internal loop (Newton method) must be accomplished , for the resolution of nonlinear structural analysis which is fundamental for giving us the value of KS-constraint.

3. For nonlinear analysis , weight minimum value is lower in comparison to the one obtained from a linear analysis, this cause of geometric nonlinearities which can increase the geometrical beam stiffness, so KS-constraint is satisfied for a lower weight. This features can be noticed from the cases of a wing box without stringers : the stiffness is lower (more flexible) so the effect of nonlinearity is more evident.

4. Is important to set the number of load steps for solving the nonlinear structural analysis , because if is given the whole amount of external load, without dosing it , numerical problems can appear ; in this case, it has been proved by running several analysis at different load steps, that under the value of 15 , calculation diverge.

6.2.3 ONERA M6. 19 properties wing box with 8 stringers (nonlinear)

```
1
2 Optimization Problem -- Optimization using pyOpt_sparse
3 ==========================================
4     Objective Function: _objfunc
5
6     Solution:
7 ------------------------------------------
8     Total Time:                    42.7588
9        User Objective Time :       29.6128
10       User Sensitivity Time :     12.9489
11       Interface Time :             0.1735
12       Opt Solver Time:             0.0235
13    Calls to Objective Function :       44
14    Calls to Sens Function :            13
15
16
17
18
19
20 Design Vars
21 {'A_fl': array([98.29483754, 98.29483754, 98.29483754, 98.29483754,
      98.29483754,
22        98.29483754, 98.29483754, 98.29483754, 98.29483754, 98.29483754,
23        98.29483754, 98.29483754, 98.29483754, 98.29483754, 98.29483754,
24        98.29483754, 98.29483754, 98.29483754, 98.29483754]),
25  'A_stiff': array([53.13547581, 53.13547581, 53.13547581, 53.13547581,
      53.13547581,
26        53.13547581, 53.13547581, 53.13547581, 53.13547581, 53.13547581,
27        53.13547581, 53.13547581, 53.13547581, 53.13547581, 53.13547581,
28        53.13547581, 53.13547581, 53.13547581, 53.13547581]),
29  't_sk': array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5,
30        0.5, 0.5, 0.5, 0.5, 0.5, 0.5]),
31  't_sp': array([0.6057939, 0.6057939, 0.6057939, 0.6057939, 0.6057939,
      0.6057939,
32        0.6057939, 0.6057939, 0.6057939, 0.6057939, 0.6057939, 0.6057939,
33        0.6057939, 0.6057939, 0.6057939, 0.6057939, 0.6057939, 0.6057939,
34        0.6057939])}
35
36
37
38
39 Minimum Weight = [1.71580736]
40 Constraint_stress = [-0.00364424]
41 Constraint_buckling = [-1.56602574e-05]
```

LISTING 6.11. 19 properties ONERA M6. Wing box with 8 stringers. Non linear analysis.

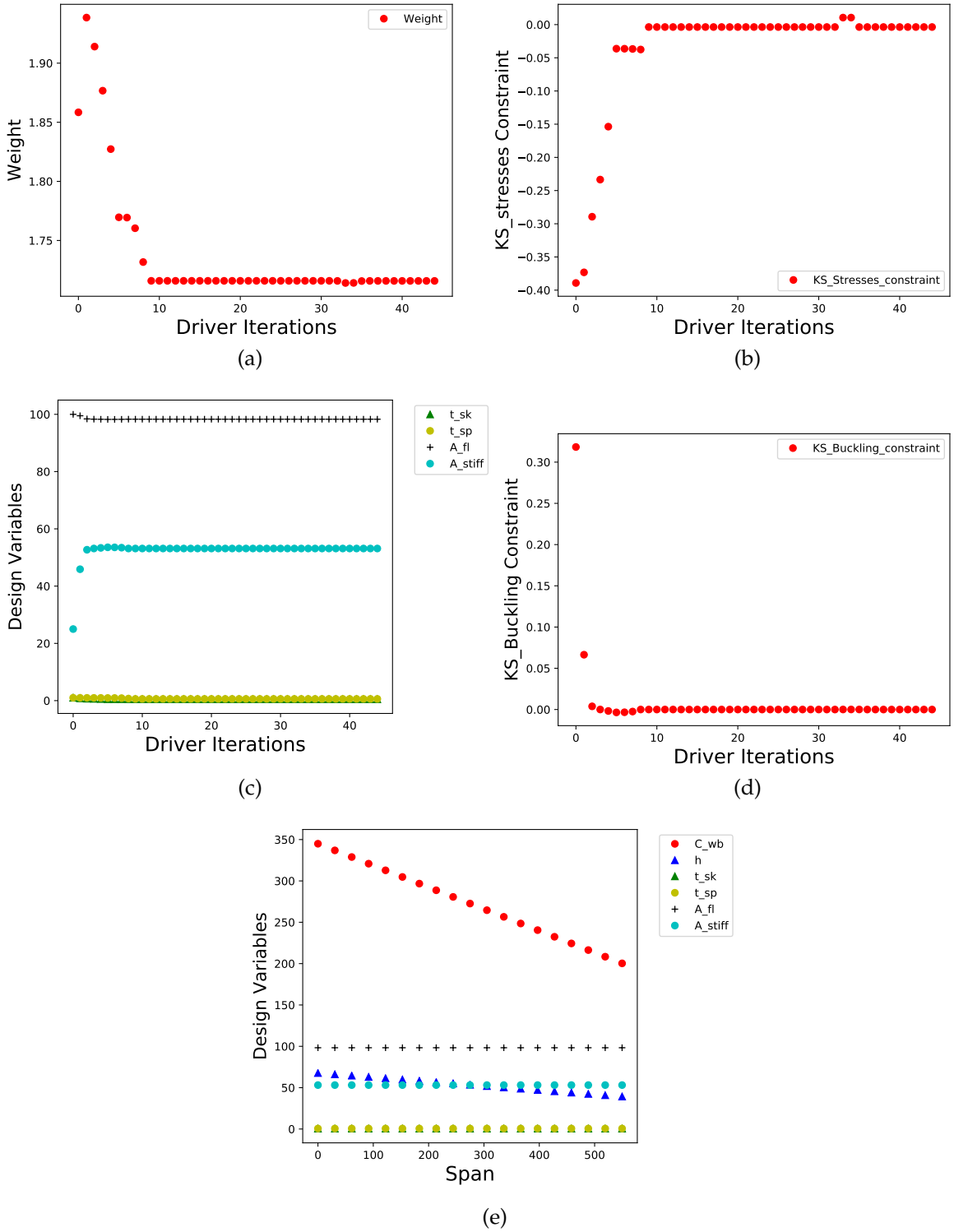


FIGURE 6.11. ONERA M6. wing box , with 8 stringers, nonlinear optimization : design variables, KS-stress constraint, KS-buckling constraint and Weight using Adjoint method

Also considering more than one section, have been obtained reasonable results. This final analysis includes the KS-buckling constraint. Buckling phenomena occurs

before the failure of the material , for this reason looking at the initial parameters (listing 6.1) KS-buckling is major than KS-stress, in our case the wing can withstand external loads but it is not free from buckling. Launching the optimization, design variables final values are such that both constraints are satisfied and **the weight reduced of a 8%**

Chapter 7

Conclusions

7.1 Conclusions

The study conducted, has provided an open-source tool able to perform linear and nonlinear optimization of flexible beams. To this end, firstly have been carried out preliminary study about nonlinear structural analysis and optimization, secondly a deep immersion in the nonlinear beam finite element solver, has been necessary to understand its source code; therefore, through the use of objected-oriented programming in C++ and Python, has been enlarged pyBeam and implemented the actual optimization framework , openMDAO. Finally, have been shown the results and, the validation of the tool in the most simple case, has been proved.
Three testcases have been analyzed : a Long beam loaded at the tip along the three axes ,in order to prove the correctness of the optimization tool by launching a linear analysis; then, a semi-span wing, named ONERA M6, has been studied running linear and nonlinear analysis, considering just one section and just the stresses constraint, finally, the same wing has been examined but considering 19 different sections and the buckling constraint. Satisfactory results have been obtained. Regarding future development, can be thought of introducing a lot more accuracy into the tool. For example, a more accurate implementation of the "reinforced shell" theory : should be introduced, in the calculation of stress state, the contribution of panels strip able to absorb the normal stresses; this addition would change the concentrated areas system center of gravity , with a consequent variation of the stress sate. If this is not considered, optimization can lead to an oversized structure. At the end, different testcases, with variable complexity, should be investigated in order to improve the tool efficiency.

Bibliography

[1] Rocco Bombardieri, Ruben Sanchez, Rauno Cavallaro, and Nicolas Gauger. Towards an open-source framework for aero-structural design and optimization within the su2 suite. 09 2019.

[2] Jan A. Snyman and Daniel N. Wilke. *Practical Mathematical Optimization - Basic Optimization Theory and Gradient Based Algorithms*. Springer, 2018.

[3] Justin S. Gray, John T. Hwang, Joaquim R. R. A. Martins, Kenneth T. Moore, and Bret A. Naylor. OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, April 2019.

[4] NASA. "NPARC Verification and Validation Web site Home Page", https://www.grc.nasa.gov/www/wind/valid/homepage.html.

[5] Carlson J.-R. Derlaga J. M. Gnoffo P. A. Hammond D. P. Jones W. T. Kleb B. Lee-Rausch E. M. Nielsen E. J. Park M. A. Rumsey C. L. Thomas J. L. Thompson K. B. Biedron, R. T. and W. A. Wood. *"FUN3D Manual: 13.5", https://fun3d.larc.nasa.gov, 2019.*

[6] Ronzheimer A. Haar-D. Abu-Zurayk M. Lummer M. Krüger W. Brezillon, J. and F. J. Natterer. *"Development and application of multi-disciplinary optimization capabilities based on high-fidelity methods," 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference 2012, 2012.*

[7] L. Cambier and M. Gazaix. "elsa: An efficient object-oriented solution to cfd complexity,". *40th AIAA Aerospace Sciences Meeting and Exhibit*, 2002.

[8] Nicholas M. K. Poon Joaquim R. R. A. Martins. On Structural Optimization Using Constraint Aggregation. *6th World Congress on Structural and Multidisciplinary Optimization*, Rio de Janeiro, 30 May - 03 June 2005, Brazil.

[9] Robert Levy and William R. Spillers. *Analysis of Geometrically Nonlinear Structures,second edition*. Springer-Science+Business Media, B.V., 2003.

[10] Luca Pustina. *Towards transonic aerodynamic shape optimization of unconventional aircraft with opensource software*. Master Thesis , Uniersità Degli Studi Roma Tre, 2018 , Roma.

[11] Javier Vela Peña. *Aeroelastic calculations on an equivalent beam-based NASA CRM*. Bechelor Thesis,Universidad Carlos III De Madrid, Madrid.

[12] Brian Moran Ted Belytschko, Wing Kam Liu. *Nonlinear Finite Elements for Continua and Structures*. John Wiley Sons, Ltd, 2000.

[13] Lars Andersen and Søren R.K. Nielsen. *Elastic Beams in Three Dimensions*. Aalborg University - Department of Civil Engineering, August 2008.

[14] Di Sciuva Marco. Course of aircraft structures, lecture notes, politecnico di torino. 2018-2019.

[15] T.H.G. Megson. *Aircraft Structures*. E. Arnold Ed, 1990.

[16] J.N.Reddy. *An Introduction to the Finite Element Method*. McGraw-HillBookCompany,Ltd., 1984.

[17] Mohammed Hjiaj Thanh-Nam Le, Jean-Marc Battini. Dynamics of 3d beam elements in a corotational context: A comparative study of established and new formulations. *Finite Elements in Analysis and Design*, 61:97–111, 2012.

[18] Mohammed Hjiaj Thanh-Nam Le, Jean-Marc Battini. A consistent 3d corotational beam element for nonlinear dynamic analysis of flexible structures. *Computer Methods in Applied Mechanics and Engineering*, 269:538–565, 2014.

[19] George J Simitses. *An introduction to the elastic stability of structures*. Englewood Cliffs, N.J. : Prentice-Hall, 1976. Includes bibliographical references and indexes.

[20] Howard D. Curtis. *Fundamentals of Aircraft Structural Analysis*. McGraw-Hill, 1997.

Appendix A

C++ core functions

A.1 "SetSectionProperty" and "FromWBtoInertias" functions

```
1
2
3 void CProperty::SetSectionProperties(passivedouble C_wb_, passivedouble h_,
      passivedouble t_sk_,
4                                 passivedouble t_sp_, passivedouble A_fl_,
5                                 int n_stiff_, passivedouble A_stiff_)
6 {
7     C_wb = C_wb_;
8     h = h_;
9     t_sk = t_sk_;
10    t_sp = t_sp_;
11    A_fl = A_fl_;
12    n_stiff = n_stiff_;
13    A_stiff = A_stiff_;
14    FromWBtoInertias();
15
16
17
18
19    isWBDV = 1;
20
21
22 }
23
24 void CProperty::FromWBtoInertias()
25 {
26     addouble b=(C_wb)/(((n_stiff+4)/2)-1);       //distance between
      stiffeners
27
28     int r= ((n_stiff)/2)%2;
29
30     addouble a=0;
31     addouble summ_ys=0;
32
```

```
33      if (n_stiff == 0 ){  //std::cout << "--> no stiffeners "<< std::endl;
34
35          summ_ys=0;}     //stiffeners moment of inertia respect vertical
    axis = 0
36
37      else if (r ==0)  {  //std::cout << "--> Even Number of stiffeners"<<
    std::endl;
38          for (int j=0;j<=(n_stiff/4)-1;j+=1){
39             a=a + pow(0.5+j,2);}   }
40
41      else  { //std::cout << "--> Odd Number of stiffeners"<< std::endl;
42
43          for (int i=1;i<=(((n_stiff/2)-1)/2);i+=1) {
44              a=a+pow(i,2);}   }
45
46     summ_ys=4*pow(b,2)*a;
47
48     addouble A_skin = (C_wb+t_sp)*t_sk;                          //  skin area
     (on one side)
49     addouble A_web   = (t_sp*(h-t_sk));                          //  spar area
     (on one side)
50
51     addouble Iyy_skin= (C_wb+t_sp)*(pow(t_sk,3)/12)+A_skin*(pow(((h/2)),2))
    ;  // Iyy of the skin
52     addouble Iyy_web= t_sp*(pow((h-t_sk),3)/12);
       // Iyy of the spar web
53     addouble Izz_skin= t_sk*(pow((C_wb-t_sp),3)/12);
       //  Izz of the skin
54     addouble Izz_web= (h-t_sk)*(pow(t_sp,3)/12)+A_web*(pow(((C_wb/2)),2));
       //  Izz of the spar web
55     addouble Izz_fl = 4*A_fl*pow((C_wb/2),2); // Izz of the spars caps
56
57     A_b = n_stiff*A_stiff + 4*A_fl;  /// Area of the boom
58
59     Iyy_b=(n_stiff)*((A_stiff)*pow((h/2),2))  + 4*A_fl*pow((h/2),2);  //
    Iyy of the booms system for ideal shell theory
60
61     Izz_b=(A_stiff*summ_ys+Izz_fl);                                  // Izz
     of the booms system for ideal shell theory
62
63     A = 2*A_skin + 2*A_web  +  n_stiff*A_stiff  +  4*A_fl;       //  total
    area
64
65     Iyy = 2*Iyy_skin  + 2*Iyy_web + (n_stiff)*((A_stiff)*pow((h/2),2))  +
    4*A_fl*pow((h/2),2);
66
67     Izz = 2*Izz_skin  +   2*Izz_web + (A_stiff*summ_ys+Izz_fl);
68
69     Jt=(2*t_sp*t_sk*pow(C_wb,2)*pow(h,2))/(C_wb*t_sp+h*t_sk);
70
71     J0=Iyy+Izz;
72 }
```

LISTING A.1. "SetSectionProperty" and "FromWBtoInertias" functions

A.2 "StressRetrieving" function

```
void  CElement::StressRetrieving()
{

    int n_tot = 4 + n_stiff;  // n_stiff + 4 spars caps

    addouble b=(C_wb)/(((n_tot)/2)-1);


    addouble L_Qxy = -h/2;
    addouble L_Qxz = -C_wb/2;
    addouble Edim =input->GetYoungModulus_dimensional();
    addouble N    =  Edim*fint(7-1);
    addouble Qxy  = Edim*fint(8-1);                    // shear y end B and A
    addouble Qxz  = Edim*fint(9-1);                    // shear z end B and A
    addouble Mt   = Edim*fint(10-1);
    addouble My   = Edim*fint(11-1)-Qxz*(l_curr/2.0);    // moment y, mid section
    addouble Mz   = Edim*fint(12-1)+Qxy*(l_curr/2.0);    // moment z, mid section



    VectorXdDiff dsigma_dx   = VectorXdDiff::Zero(n_tot);
    VectorXdDiff axial_load  = VectorXdDiff::Zero(n_tot);

    sigma_booms = VectorXdDiff::Zero(n_tot);

    /// Calculation of Normal stress absorbed by booms (Navier Formula)
    int r= ((n_stiff)/2)%2;

    if (n_stiff == 0 ){
        sigma_booms(1-1)=(N/A_b) - (Mz/Izz_b)*C_wb*0.5 +(My/Iyy_b)*(h/2.0);
        sigma_booms(2-1)=(N/A_b) + (Mz/Izz_b)*C_wb*0.5 +(My/Iyy_b)*(h/2.0);
        sigma_booms(3-1)=(N/A_b) + (Mz/Izz_b)*C_wb*0.5 -(My/Iyy_b)*(h/2.0);
        sigma_booms(4-1)=(N/A_b) - (Mz/Izz_b)*C_wb*0.5 -(My/Iyy_b)*(h/2.0);

        dsigma_dx(1-1)= -A_fl*(Qxy/Izz_b)*C_wb*0.5  + A_fl*(Qxz/Iyy_b)*(h/2.0);
        dsigma_dx(2-1)=  A_fl*(Qxy/Izz_b)*C_wb*0.5  + A_fl*(Qxz/Iyy_b)*(h/2.0);
        dsigma_dx(3-1)=  A_fl*(Qxy/Izz_b)*C_wb*0.5  - A_fl*(Qxz/Iyy_b)*(h/2.0);
        dsigma_dx(4-1)= -A_fl*(Qxy/Izz_b)*C_wb*0.5  - A_fl*(Qxz/Iyy_b)*(h/2.0);

        axial_load  = sigma_booms*A_fl;}

    else if (r==0) // Even number
    {
        for (int i=1-1  ; i<= ((n_tot)/4 - 1) ; i += 1){
```

```
45          sigma_booms(i)                    = (N/A_b) - (Mz/Izz_b)*b*((
    n_tot/4)-1-i+(1/2)) + (My/Iyy_b)*(h/2);
46          sigma_booms( ((n_tot)/4)+i )    = (N/A_b) + (Mz/Izz_b)*b*(i +
    (1/2))          + (My/Iyy_b)*(h/2);
47          sigma_booms( ((n_tot)/2)+i )    = (N/A_b) + (Mz/Izz_b)*b*((
    n_tot/4)-1-i+(1/2)) - (My/Iyy_b)*(h/2);
48          sigma_booms( (((n_tot)*3/4))+i )= (N/A_b) - (Mz/Izz_b)*b*(i +
    (1/2))          - (My/Iyy_b)*(h/2);
49
50          dsigma_dx(i)                      =  -A_stiff*(Qxy/Izz_b)*b*((
    n_tot/4)-1-i+(1/2))  + A_stiff*(Qxz/Iyy_b)*(h/2);
51          dsigma_dx( ((n_tot)/4)+i )        =  A_stiff*(Qxy/Izz_b)*b*(i
    + (1/2))            + A_stiff*(Qxz/Iyy_b)*(h/2);
52          dsigma_dx( ((n_tot)/2)+i )        =  A_stiff*(Qxy/Izz_b)*b*((
    n_tot/4)-1-i+(1/2))   - A_stiff*(Qxz/Iyy_b)*(h/2);
53          dsigma_dx( ((n_tot)*3/4)+i)       =  -A_stiff*(Qxy/Izz_b)*b*(i
     + (1/2))           - A_stiff*(Qxz/Iyy_b)*(h/2);
54      }
55      //Take into account the different Spars' Area in the corners
56      dsigma_dx(1-1)             =  dsigma_dx(1-1)*(A_fl/A_stiff);
57      dsigma_dx((n_tot/2) - 1)   =  dsigma_dx((n_tot/2) - 1)*(A_fl/
    A_stiff);
58      dsigma_dx((n_tot/2+1) - 1) =  dsigma_dx((n_tot/2+1) - 1)*(A_fl/
    A_stiff);
59      dsigma_dx(n_tot - 1)       =  dsigma_dx(n_tot - 1)*(A_fl/A_stiff);
60
61      axial_load                  = sigma_booms*A_stiff;
62      axial_load(1-1)             = axial_load (1 -1)*(A_fl/A_stiff);
63      axial_load((n_tot/2) - 1)   = axial_load((n_tot/2) - 1)*(A_fl/
    A_stiff);
64      axial_load((n_tot/2+1) - 1) = axial_load((n_tot/2+1) - 1)*(A_fl/
    A_stiff);
65      axial_load(n_tot - 1)       = axial_load(n_tot - 1)*(A_fl/A_stiff);
66  }
67  else{
68  //odd number
69      for (int i=1-1 ; i<= ((n_tot-2)/4 - 1) ; i += 1){
70          sigma_booms(i)                      = (N/A_b) - (Mz/Izz_b)*b
    *(((n_tot-2)/4)-i)  + (My/Iyy_b)*(h/2);
71          sigma_booms((((n_tot-2)/4)+1) +i)   = (N/A_b) + (Mz/Izz_b)*b*(i
     + 1)            + (My/Iyy_b)*(h/2);
72          sigma_booms((((n_tot-2)/2)+1) +i)   = (N/A_b) + (Mz/Izz_b)*b
    *(((n_tot-2)/4)-i)  - (My/Iyy_b)*(h/2);
73          sigma_booms((((n_tot-2)*3/4)+2) +i) = (N/A_b) - (Mz/Izz_b)*b* (
    i + 1)          - (My/Iyy_b)*(h/2);
74
75          dsigma_dx(i)                        =  -A_stiff*(Qxy/Izz_b)*b
    *(((n_tot-2)/4)-i)  + A_stiff*(Qxz/Iyy_b)*(h/2);
76          dsigma_dx((((n_tot-2)/4)+1) +i)     =   A_stiff*(Qxy/Izz_b)*b*(i
     + 1)            + A_stiff*(Qxz/Iyy_b)*(h/2);
77          dsigma_dx((((n_tot-2)/2)+1) +i)     =   A_stiff*(Qxy/Izz_b)*b
    *(((n_tot-2)/4)-i)  - A_stiff*(Qxz/Iyy_b)*(h/2);
78          dsigma_dx((((n_tot-2)*3/4)+2) +i)   =  -A_stiff*(Qxy/Izz_b)*b*(i
     + 1)            - A_stiff*(Qxz/Iyy_b)*(h/2);
79      }
```

```
80
81        //Take into account the different Spars' Area in the corners
82        dsigma_dx(1-1)              =  dsigma_dx(1-1)*(A_fl/A_stiff);
83        dsigma_dx((n_tot/2) - 1)    =  dsigma_dx((n_tot/2) - 1)*(A_fl/
    A_stiff);
84        dsigma_dx((n_tot/2+1) - 1) =  dsigma_dx((n_tot/2+1) - 1)*(A_fl/
    A_stiff);
85        dsigma_dx(n_tot - 1)        =  dsigma_dx(n_tot - 1)*(A_fl/A_stiff);
86
87        sigma_booms((n_tot-2)/4)           = (N/A_b) + (My/Iyy_b)*(h/2);
     //upper stiffener on Z-axis
88        sigma_booms(((n_tot-2)*3/4)+1)     = (N/A_b) - (My/Iyy_b)*(h/2);
    //lower stiffener on Z-axis
89
90        dsigma_dx((n_tot-2)/4)           =  A_stiff*(Qxz/Iyy_b)*(h/2);
    //upper stiffener on Z-axis
91        dsigma_dx(((n_tot-2)*3/4)+1)     = -A_stiff*(Qxz/Iyy_b)*(h/2);   //
    lower stiffener on Z-axis
92
93        axial_load                  = sigma_booms*A_stiff;
94        axial_load(1-1)             = axial_load (1 -1)*(A_fl/A_stiff);
95        axial_load((n_tot/2) - 1)   = axial_load((n_tot/2) - 1)*(A_fl/
    A_stiff);
96        axial_load((n_tot/2+1) - 1) = axial_load((n_tot/2+1) - 1)*(A_fl/
    A_stiff);
97        axial_load(n_tot - 1)       = axial_load(n_tot - 1)*(A_fl/A_stiff);
98       }
99
100
101     /// Shear Flux calculation
102
103     //Solve the equation :
104
105     // tau_coeff*tau + dsigma_dx=0 ---> tau= -dsigma_dx*(tau_coeff)^-1
106
107     //    % tau_coeff = ( 1    0       0     0      0 ...     -1;
          dsigma_dx=( dsigma/dx (1st )
108     //    %               -1     1      0     0      0 ...     0;
                        dsigma/dx (2nd)
109     //    %                0     -1      1     0     0         0;
                          .
110     //    %                0     0       -1    1     0         0;
                          .
111     //    %                0     0       0     -1    1         0;
                        dsigma/dx (ntot-1 )
112     //    %               bh     bh ...  bh     0     0 ...     0];
                           M_Q]
113
114     MatrixXdDiff tau_coeff= MatrixXdDiff::Zero(n_tot,n_tot);
115
116     VectorXdDiff q  = VectorXdDiff::Zero(n_tot);
117
118     addouble M_Q = Mt + Qxz*L_Qxz - Qxy*L_Qxy ;   // total Torque in the
    section due to Qxz and Qxy
119
```

```
120     dsigma_dx(n_tot-1)= M_Q;     //index start from 0
121
122
123
124     // fill tau_coeff matrix
125     for (int j=1 -1 ; j<= (n_tot-1) -1; j+=1){
126         tau_coeff(j,j)=1;}           // Diagonal
127
128
129     for (int jj=1 -1; jj<= (n_tot-2) -1; jj+=1){
130         tau_coeff(jj+1,jj)= -1;}      // sub-diagonal
131
132     for (int jjj=1 -1 ; jjj<= (n_tot/2) -1 ; jjj+=1){
133         tau_coeff(n_tot-1,jjj)= b*h;}   // last row
134
135
136     tau_coeff(1-1,(n_tot)-1)= -1;  // up right corner
137
138     // System resolution
139     q = (tau_coeff).fullPivHouseholderQr().solve(-dsigma_dx);
140
141     // Tau retrieving
142     tau = VectorXdDiff::Zero(n_tot);
143     tau.segment(1-1,n_tot/2 -1 )           = q.segment(1-1, n_tot/2 -1) /
    t_sk;
144     tau(n_tot/2 -1)                        = q(n_tot/2 -1) /t_sp;
145     tau.segment(n_tot/2+1 -1, n_tot/2 -1 ) = q.segment(n_tot/2+1 -1, n_tot
    /2 -1 )/t_sk;
146     tau(n_tot -1)                          = q(n_tot -1) /t_sp;
147
148
149     //--- Section Verification ---------
150 /*   //N resultant in the section
151     addouble N_sec =0;
152
153     for (int i = 1-1 ;i<=(n_tot) -1 ; i=i+1){
154         N_sec=N_sec+axial_load(i);}
155
156     // Tz resultant in the section
157     addouble Tz_sec= -tau( (n_tot/2)-1)*h + tau(n_tot-1)*h ;
158
159
160     //Ty resultant in the section
161     VectorXdDiff Ty_vect = VectorXdDiff::Zero((n_tot/2)-1);
162     Ty_vect.segment(1-1,(n_tot/2)-1)=tau.segment(1-1, (n_tot/2)-1)*b - tau.
    segment((n_tot/2), (n_tot/2)-1)*b;
163
164     addouble Ty_sec=0;
165     for (int iy = 1-1 ;iy<=((n_tot/2)-1) -1 ; iy=iy+1){
166         Ty_sec=Ty_sec + Ty_vect(iy);
167     }
168  */
169
170 }
```

LISTING A.2. "StressRetrieving" function

A.3 "VonMises" function

```
void CElement:: VonMises()
{
 /// Von mises criteria (skin and booms )-->(sigma_e/sigma_all) -1<=0

     // n_stiff + 4 spars caps
     int n_tot = n_stiff+4;


     //initialization element constraint
     g_element = VectorXdDiff::Zero(2*n_tot);

     // Safety factor
     SF=1.5;

     // yielding stress Alluminum 7075
     sigma_y= 468.5;

     // Allowable Stress
     addouble sigma_all = sigma_y/SF;


 // Constraints

    for (int i= 1 -1 ; i<= n_tot -1 ; i=i+1)
    {

 //Normal stress state (Booms)--->g=(sigma_x/sigma_all) -1
      g_element(i)=(fabs(sigma_booms(i))/sigma_all)-1;

// Pure shear state (Spar and skin)---> g= (sqrt(3)*tau /sigma_all) -1
      g_element((n_tot+1 - 1) +i )=(pow(3,0.5)*fabs(tau(i))/sigma_all)-1;
    }

}
```

LISTING A.3. "VonMises" function

A.4 "BoomsBuckling" function

```
void CElement:: BoomsBuckling()
{
void CElement:: BoomsBuckling(){

 //  Vallat formula for the calculation of sigma_cr  buckling for the
    concentrated areas

    // has been considered " L " section booms ,

     int n_tot = n_stiff+4;                    // n_stiff + 4 flanges
     addouble t_b =1;
     addouble h_stiff = (A_stiff + pow(t_b,2))/(2*t_b);
```

```
12      addouble h_fl =  (A_fl + pow(t_b,2))/(2*t_b);
13
14
15      addouble K = 8.5;                        // constant for open section
    stiffeners
16      addouble beta_fl = h_fl/t_b;
17      addouble beta_stiff = h_stiff/t_b;
18
19
20
21      addouble Edim =input->GetYoungModulus_dimensional();
22
23
24      sigma_y = 468.5;
25
26      addouble  sigma_buckl_fl =  sigma_y/(1 + K*beta_fl*sigma_y/Edim );
27      addouble  sigma_buckl_stiff =  sigma_y/(1 + K*beta_stiff*sigma_y/Edim
    );
28
29     // Constraints
30
31     int n_neg=0;
32
33     for (int i= 1 -1 ; i<= n_tot -1 ; i=i+1)
34     {
35       if (sigma_booms(i) < 0){
36           n_neg++ ;
37       }
38     }
39
40     g_buckl_element = VectorXdDiff::Zero(n_neg);
41
42     int j=0;
43     for (int i= 1 -1 ; i<= n_tot -1 ; i=i+1)
44     {
45       if (sigma_booms(i) < 0){
46
47          if (i == 0 or i== (n_tot/2) - 1 or i==(n_tot/2+1) - 1 or i ==n_tot
    - 1 ){
48               g_buckl_element(j++)=(fabs(sigma_booms(i)/sigma_buckl_fl))-1;
49
50          }else
51           {
52
53       g_buckl_element(j++)=(fabs(sigma_booms(i)/sigma_buckl_stiff))-1;
54
55           }
56       }
57     }
58 }
```

LISTING A.4. "BoomsBuckling" function

A.5 "Evaluate_no_AdaptiveKSstresses" function

```
addouble CStructure::Evaluate_no_AdaptiveKSstresses()
{
    // Ks calculation ----> Ks(g_element) = g_max + summ (exp(aggr_parameter*(g_element - g_max)))
    int n_stiff = 0;
    int n_tot = n_stiff+4;  // n_stiff + 4 spars caps

    r=50;  // aggregation parameter

    int id_fe;


    addouble g_max;
    addouble summ_KS=0;

    for (id_fe=1;     id_fe <= nfem ; id_fe++) {

       cout<<"element ----------------------> "<< id_fe <<endl;

       element[id_fe-1]->StressRetrieving();
       element[id_fe-1]->VonMises();

     //g_max
       g_max= element[1-1]->g_element(1-1);

       for(int i= 1-1 ; i<= n_tot;i=i+1)
     {
       if(element[id_fe-1]->g_element(i) >= g_max)
       {

        g_max= element[id_fe-1]->g_element(i);
       }

        summ_KS=summ_KS + pow(M_E,r*(element[id_fe-1]->g_element(i)));
     }
    }

    //KS
     KS=g_max+(1/ r)*log(summ_KS *pow(M_E,-r*g_max )); //contribute of g_max


}
```

LISTING A.5. "Evaluate_no_AdaptiveKSstresses" function

A.6 "Evaluate_no_AdaptiveKSbucklings" function

```
addouble CStructure::Evaluate_no_AdaptiveKSbuckling()
{
   // Ks calculation ----> Ks(g_element) = g_max + summ (exp(aggr_parameter
    *(g_element - g_max)))
    int n_stiff = element[1-1]->elprop->Getn_stiff();
    int n_tot = n_stiff+4;  // n_stiff + 4 flanges
    addouble r = 50;       // Aggregation parameter
    int id_fe;


    addouble g_max;
    addouble summ_KS=0;


    element[1-1]->StressRetrieving();
    element[1-1]->BoomsBuckling();
    g_max= element[1-1]->g_buckl_element(1-1);

    for (id_fe=1;     id_fe <= nfem ; id_fe++) {

       element[id_fe-1]->StressRetrieving();
       element[id_fe-1]->BoomsBuckling();

        //g_max


        for(int i= 1-1 ; i<= (element[id_fe-1]->g_buckl_element).size() -1
    ; i++){

            if(element[id_fe-1]->g_buckl_element(i) >= g_max){
                g_max= element[id_fe-1]->g_buckl_element(i); }


            summ_KS  += pow(M_E,r*(element[id_fe-1]->g_buckl_element(i)));

        }

    //KS
    }
      addouble KS_buckl = g_max+(1/ r)*log(summ_KS *pow(M_E,-r*g_max )); //
   aggregated stress constraint

      if (KS_buckl > 0) {
          cout<<"!!! Stiffeners Buckling "<<endl;
      }
```

LISTING A.6. "Evaluate_no_AdaptiveKSbuckling()" function

A.7 "EvaluateWeight" function

```
addouble CStructure::EvaluateWeight(){

    addouble A=element[1-1]->elprop->GetA();            //Area  (constant)


    addouble l=element[1-1]->GetInitial_Length();     // initial length

    W=nfem*ro*l*A;                                    //weight

    //cout<<"W"<<W<<endl;
}
```

LISTING A.7. "EvaluateWeight" function

Appendix B

Python functions

B.1 "Wrapping functions"

```
class pyBeamSolver:
class pyBeamSolverAD:
```

LISTING B.1. "Wrapping Class"

these classes are built in "PyBeamLib.py" and "PyBeamLibAD.py". We are going to deal with just functions we are interested in, fundamentals for the optimization.

```

from pyBeamIO import pyBeamConfig as pyConfig
from pyBeamIO import pyBeamInput as pyInput
import numpy as np
import pyBeamAD

# -------------------------------------------------------------------------
#  Beam object
# -------------------------------------------------------------------------

class pyBeamSolverAD:
  """Description"""

  def __init__(self, file_dir, config_fileName):
    """ Description. """

    self.file_dir = file_dir
    self.Config_file = self.file_dir + '/' + config_fileName
    self.Config = {}



    # Parsing config file
    self.Config = pyConfig.pyBeamConfig(self.Config_file)  # Beam
   configuration file

    self.Mesh_file = self.file_dir + '/' + self.Config['MESH_FILE']
    self.Property = self.file_dir + '/' + self.Config['PROPERTY_FILE']
```

```
28
29      # Parsing mesh file
30      self.nDim= pyInput.readDimension(self.Mesh_file)
31      self.node_py, self.nPoint = pyInput.readMesh(self.Mesh_file, self.nDim)
32      self.elem_py, self.nElem = pyInput.readConnectivity(self.Mesh_file)
33      self.Constr, self.nConstr = pyInput.readConstr(self.Mesh_file)
34      self.RBE2_py, self.nRBE2 = pyInput.readRBE2(self.Mesh_file)
35      # Parsing Property file
36      self.Prop, self.nProp = pyInput.readProp(self.Property)
37
38      # Initializing objects
39      self.beam = pyBeamAD.CBeamSolver()
40      self.inputs = pyBeamAD.CInput(self.nPoint, self.nElem, self.nRBE2, self
    .nProp)
41
42      # Start recording
43      print("--> Initialization successful!")
44
45      # Sending to CInput object
46      pyConfig.parseInput(self.Config, self.inputs, self.Constr, self.nConstr
    )
47      # Assigning input values to the input object in C++
48      self.inputs.SetParameters()
49      # Set the discrete adjoint flag to true
50      self.inputs.SetDiscreteAdjoint()
51      # Initialize the input in the beam solver
52      self.beam.InitializeInput(self.inputs)
53
54      # Assigning values to the CNode objects in C++
55      self.node = []
56      for i in range(self.nPoint):
57          self.node.append(pyBeamAD.CNode(self.node_py[i].GetID()))
58          for j in range(self.nDim):
59              self.node[i].InitCoordinate(j, float(self.node_py[i].GetCoord()
    [j][0]))
60          self.beam.InitializeNode(self.node[i], i)
61
62      # Assigning property values to the property objects in C++
63      self.beam_prop = []
64      self.nPropDVs = 0
65      for i in range(self.nProp):
66          self.beam_prop.append(pyBeamAD.CProperty(i))
67          if self.Prop[i].GetFormat() == "N":
68              self.nPropDVs = self.nPropDVs + 4
69              self.beam_prop[i].SetSectionProperties(self.Prop[i].GetA(),
    self.Prop[i].GetIyy(), self.Prop[i].GetIzz(), self.Prop[i].GetJt())
70          elif self.Prop[i].GetFormat() == "S":
71              self.nPropDVs = self.nPropDVs + 6
72              self.beam_prop[i].SetSectionProperties(self.Prop[i].GetC_wb(),
    self.Prop[i].Geth(), self.Prop[i].Gett_sk(), self.Prop[i].Gett_sp(),\
73              self.Prop[i].GetA_fl(), self.Prop[i].Getn_stiff(),self.Prop[i].
    GetA_stiff() )
74              #print(self.Prop[i].GetA())
75          else:
76              raise ValueError("Unknown paramter for Property CARD input.
```

```
    Execution aborted")
77          self.beam.InitializeProp(self.beam_prop[i], i)
78
79      # Assigning element values to the element objects in C++
80      self.element = []
81      for i in range(self.nElem):
82          self.element.append(pyBeamAD.CElement(i))
83          self.element[i].Initializer(self.node[self.elem_py[i].GetNodes()[0,
         0] - 1], self.node[self.elem_py[i].GetNodes()[1, 0] - 1],
84                                      self.beam_prop[self.elem_py[i].GetProperty()
         - 1], self.inputs, self.elem_py[i].GetAuxVector()[0, 0],
85                                      self.elem_py[i].GetAuxVector()[1, 0], self.
    elem_py[i].GetAuxVector()[2, 0])
86          self.beam.InitializeElement(self.element[i], i)
87
88      # Here we need to pass the AeroPoint matrix of the wing grid
89      # IF ANY, assigning RBE2_element values to the RBE2 objects in C++
90      if self.nRBE2 != 0:
91          self.RBE2 = []
92          for i in range(self.nRBE2):
93              self.RBE2.append(pyBeamAD.CRBE2(i))
94              self.RBE2[i].Initializer(self.node[self.RBE2_py[i].GetNodes()
    [0, 0] - 1], self.node[self.RBE2_py[i].GetNodes()[1, 0] - 1])
95              self.beam.InitializeRBE2(self.RBE2[i], i)
96
97
98      # Initialize structures to store the coordinates and displacements
99
100     self.coordinate_X = []
101     self.coordinate_Y = []
102     self.coordinate_Z = []
103
104     self.coordinate_X0 = []
105     self.coordinate_Y0 = []
106     self.coordinate_Z0 = []
107
108     self.displacement_X = []
109     self.displacement_Y = []
110     self.displacement_Z = []
111
112     # finally intializing the structure for the solver
113     self.beam.InitializeStructure()
114
115     print("--> Initialization successful")
116     print("\n
    ------------------------------------------------------------------------\
    n")
117
118
119 def RegisterLoads(self):
120   """ This function starts load registration for AD  """
121     self.beam.RegisterLoads()
122
123 def StartRecording(self):
124   """ This function stops registration for AD  """
```

```
125     self.beam.StartRecording()
126
127   def SetDependencies(self):
128     """ This function stops registration for AD  """
129     self.beam.SetDependencies()
130
131   def StopRecording(self):
132     """ This function stops registration for AD  """
133     self.beam.StopRecording()
134
135   def StopRecordingWeight(self):
136     """ This function stops registration for AD  """
137     self.beam.StopRecordingWeight()
138
139   def StopRecordingKSstresses(self):
140     """ This function stops registration for AD  """
141     self.beam.StopRecordingKSstresses()
142
143   def StopRecordingKSbuckling(self):
144         """ This function stops registration for AD  """
145         self.beam.StopRecordingKSbuckling()
146
147   def StopRecordingSigmaBoom(self):
148     self.beam.StopRecordingSigmaBoom()
149   def StopRecordingIzz_b(self):
150     self.beam.StopRecordingIzz_b()
151
152   def StopRecordingEA(self):
153     """ This function stops registration for AD  """
154     self.beam.StopRecordingEA()
155
156   def StopRecordingNint(self):
157     """ This function stops registration for AD  """
158     self.beam.StopRecordingNint()
159
160
161   def ComputeAdjoint(self):
162     """ This function computes Adjoint for AD  """
163     self.beam.ComputeAdjoint()
164
165   def ComputeAdjointNint(self):
166     """ This function computes Adjoint for AD  """
167     self.beam.ComputeAdjointNint()
168
169   def ComputeAdjointIzz_b(self):
170     """ This function computes Adjoint for AD  """
171     self.beam.ComputeAdjointIzz_b()
172
173   def ComputeAdjointWeight(self):
174     """ This function computes Adjoint for AD  """
175     self.beam.ComputeAdjointWeight()
176
177   def ComputeAdjointKSStresses(self):
178     """ This function computes Adjoint for AD  """
179     self.beam.ComputeAdjointKSstresses()
```

```
180
181   def ComputeAdjointKSBuckling(self):
182     """ This function computes Adjoint for AD  """
183     self.beam.ComputeAdjointKSbuckling()
184
185   def ComputeAdjointSigmaBoom(self):
186     """ This function computes Adjoint for AD  """
187     self.beam.ComputeAdjointSigmaBoom()
188
189   def ComputeAdjointEA(self):
190     """ This function computes Adjoint for AD  """
191     self.beam.ComputeAdjointEA()
192
193
194
195   def SetLoads(self, iVertex, loadX, loadY, loadZ):
196
197     """ This function sets the load  """
198     self.beam.SetLoads(iVertex, 0, loadX)
199     self.beam.SetLoads(iVertex, 1, loadY)
200     self.beam.SetLoads(iVertex, 2, loadZ)
201
202   def GetLoadSensitivity(self, iVertex):
203
204     """ This function returns the load sensitivity  """
205     sensX = self.beam.ExtractLoadGradient(iVertex, 0)
206     sensY = self.beam.ExtractLoadGradient(iVertex, 1)
207     sensZ = self.beam.ExtractLoadGradient(iVertex, 2)
208
209     return sensX, sensY, sensZ
210
211   def SetDisplacementAdjoint(self, iVertex, adjX, adjY, adjZ):
212     """ This function sets the load  """
213     self.beam.StoreDisplacementAdjoint(iVertex, 0, adjX)
214     self.beam.StoreDisplacementAdjoint(iVertex, 1, adjY)
215     self.beam.StoreDisplacementAdjoint(iVertex, 2, adjZ)
216
217
218   def ComputeObjectiveFunction(self, iNode):
219
220     """ This function computes the objective function (Important to be
      recorded) """
221     displacement = self.beam.OF_NodeDisplacement(iNode)
222     print("Objective Function - Displacement(", iNode, ") = ", displacement
      )
223     return displacement
224
225
226   #####  DEBUG
227   def ComputeNint(self):
228     Nint = self.beam.EvalNint()
229     return Nint
230   def ComputeIzz_b(self):
231     Nint = self.beam.EvalIzz_b()
232     return Nint
```

```
233
234  def GetDesignVariables(self):
235      self.beam_prop = []
236      for i in range(self.nProp):
237          if self.Prop[i].GetFormat() == "S":
238              C_wb = self.Prop[i].GetC_wb()
239              h = self.Prop[i].Geth()
240              A_fl = self.Prop[i].GetA_fl()
241              A_stiff = self.Prop[i].GetA_stiff()
242              n_stiff = self.Prop[i].Getn_stiff()
243              t_sk = self.Prop[i].Gett_sk()
244              t_sp = self.Prop[i].Gett_sp()
245              DVs = (C_wb, h, t_sk, t_sp, A_fl, n_stiff, A_stiff)
246
247              return DVs
248
249  def ComputeWeight(self):
250    """ This function computes the response weight of the structure (
     important to be recorded) """
251    weight = self.beam.EvalWeight()
252    return weight
253
254
255  def ComputeResponseKSStress(self):
256    """ This function computes the KS stress on the structure (important to
     be recorded) """
257    KSStress= self.beam.EvalKSStress()
258    return KSStress
259
260
261  def ComputeResponseKSBuckling(self):
262    """ This function computes the KS stress on the structure (important to
     be recorded) """
263    KSBuckl= self.beam.EvalKSBuckling()
264    return KSBuckl
265
266  def ComputeResponseSigmaBoom(self):
267    """ This function computes the KS stress on the structure (important to
     be recorded) """
268    SB= self.beam.EvalSigmaBoom()
269    return SB
270
271  def ComputeEA(self):
272    EA = self.beam.EvalEA()
273    return EA
274
275
276  def Run(self):
277    """ This function runs the solver and stores the results.
278        Needs to be run after __SetLoads """
279
280    self.beam.Solve(0)
281
282    self.coordinate_X = []
283    self.coordinate_Y = []
```

```
284     self.coordinate_Z = []
285 
286     self.coordinate_X0 = []
287     self.coordinate_Y0 = []
288     self.coordinate_Z0 = []
289 
290     self.displacement_X = []
291     self.displacement_Y = []
292     self.displacement_Z = []
293 
294     for jNode in range(0,self.nPoint):
295 
296         self.coordinate_X.append(self.beam.ExtractCoordinate(jNode, 0))
297         self.coordinate_Y.append(self.beam.ExtractCoordinate(jNode, 1))
298         self.coordinate_Z.append(self.beam.ExtractCoordinate(jNode, 2))
299 
300         self.coordinate_X0.append(self.beam.ExtractCoordinate0(jNode, 0))
301         self.coordinate_Y0.append(self.beam.ExtractCoordinate0(jNode, 1))
302         self.coordinate_Z0.append(self.beam.ExtractCoordinate0(jNode, 2))
303 
304         self.displacement_X.append(self.beam.ExtractDisplacements(jNode, 0)
    )
305         self.displacement_Y.append(self.beam.ExtractDisplacements(jNode, 1)
    )
306         self.displacement_Z.append(self.beam.ExtractDisplacements(jNode, 2)
    )
307 
308 
309   def RunLin(self):
310     """ This function runs the solver and stores the results.
311         Needs to be run after __SetLoads """
312 
313     self.beam.SolveLin(0)
314 
315     self.coordinate_X = []
316     self.coordinate_Y = []
317     self.coordinate_Z = []
318 
319     self.coordinate_X0 = []
320     self.coordinate_Y0 = []
321     self.coordinate_Z0 = []
322 
323     self.displacement_X = []
324     self.displacement_Y = []
325     self.displacement_Z = []
326 
327     for jNode in range(0,self.nPoint):
328 
329         self.coordinate_X.append(self.beam.ExtractCoordinate(jNode, 0))
330         self.coordinate_Y.append(self.beam.ExtractCoordinate(jNode, 1))
331         self.coordinate_Z.append(self.beam.ExtractCoordinate(jNode, 2))
332 
333         self.coordinate_X0.append(self.beam.ExtractCoordinate0(jNode, 0))
334         self.coordinate_Y0.append(self.beam.ExtractCoordinate0(jNode, 1))
335         self.coordinate_Z0.append(self.beam.ExtractCoordinate0(jNode, 2))
```

```
336
337         self.displacement_X.append(self.beam.ExtractDisplacements(jNode, 0)
    )
338         self.displacement_Y.append(self.beam.ExtractDisplacements(jNode, 1)
    )
339         self.displacement_Z.append(self.beam.ExtractDisplacements(jNode, 2)
    )
340
341
342     def ReadRestart(self):
343
344         self.beam.ReadRestart()
345
346     def Restart(self):
347         """ This function runs the restart and stores the results.
348             Needs to be run after __SetLoads """
349
350         self.beam.RunRestart(0)
351
352         self.coordinate_X = []
353         self.coordinate_Y = []
354         self.coordinate_Z = []
355
356         self.coordinate_X0 = []
357         self.coordinate_Y0 = []
358         self.coordinate_Z0 = []
359
360         self.displacement_X = []
361         self.displacement_Y = []
362         self.displacement_Z = []
363
364         for jNode in range(0, self.nPoint):
365             self.coordinate_X.append(self.beam.ExtractCoordinate(jNode, 0))
366             self.coordinate_Y.append(self.beam.ExtractCoordinate(jNode, 1))
367             self.coordinate_Z.append(self.beam.ExtractCoordinate(jNode, 2))
368
369             self.coordinate_X0.append(self.beam.ExtractCoordinate0(jNode, 0))
370             self.coordinate_Y0.append(self.beam.ExtractCoordinate0(jNode, 1))
371             self.coordinate_Z0.append(self.beam.ExtractCoordinate0(jNode, 2))
372
373             self.displacement_X.append(self.beam.ExtractDisplacements(jNode,
    0))
374             self.displacement_Y.append(self.beam.ExtractDisplacements(jNode,
    1))
375             self.displacement_Z.append(self.beam.ExtractDisplacements(jNode,
    2))
376
377
378     def RestartLin(self):
379         """ This function runs the restart and stores the results.
380             Needs to be run after __SetLoads """
381
382         self.beam.RunRestartLin(0)
383
384         self.coordinate_X = []
```

```
385         self.coordinate_Y = []
386         self.coordinate_Z = []
387 
388         self.coordinate_X0 = []
389         self.coordinate_Y0 = []
390         self.coordinate_Z0 = []
391 
392         self.displacement_X = []
393         self.displacement_Y = []
394         self.displacement_Z = []
395 
396         for jNode in range(0, self.nPoint):
397             self.coordinate_X.append(self.beam.ExtractCoordinate(jNode, 0))
398             self.coordinate_Y.append(self.beam.ExtractCoordinate(jNode, 1))
399             self.coordinate_Z.append(self.beam.ExtractCoordinate(jNode, 2))
400 
401             self.coordinate_X0.append(self.beam.ExtractCoordinate0(jNode, 0))
402             self.coordinate_Y0.append(self.beam.ExtractCoordinate0(jNode, 1))
403             self.coordinate_Z0.append(self.beam.ExtractCoordinate0(jNode, 2))
404 
405             self.displacement_X.append(self.beam.ExtractDisplacements(jNode,
    0))
406             self.displacement_Y.append(self.beam.ExtractDisplacements(jNode,
    1))
407             self.displacement_Z.append(self.beam.ExtractDisplacements(jNode,
    2))
408 
409 
410     def PrintDisplacements(self, iVertex):
411 
412         """ This function prints to screen the displacements on the nodes """
413         print("\n--> Coord0({}): {:16.12f} {:16.12f} {:16.12f}".format(iVertex,
414                                         self.coordinate_X0[iVertex],
415                                         self.coordinate_Y0[iVertex],
416                                         self.coordinate_Z0[iVertex]))
417 
418         print("--> Coord({}) : {:16.12f} {:16.12f} {:16.12f}".format(iVertex,
419                                         self.coordinate_X[iVertex],
420                                         self.coordinate_Y[iVertex],
421                                         self.coordinate_Z[iVertex]))
422 
423         print("--> Displ({}) : {:16.12f} {:16.12f} {:16.12f}\n".format(iVertex,
424                                         self.displacement_X[iVertex],
425                                         self.displacement_Y[iVertex],
426                                         self.displacement_Z[iVertex]))
427 
428 
429 
430     def PrintSensitivitiesAllLoads(self):
431 
432         """ This function prints the sensitivities of the objective functions
    for all the loads"""
433         print("E', Nu' = (", self.beam.ExtractGradient_E(), self.beam.
    ExtractGradient_Nu(), ")")
434         for iNode in range(0,self.nPoint):
```

```
435       print("F'(",iNode,") = (", self.beam.ExtractLoadGradient(iNode,0),
      self.beam.ExtractLoadGradient(iNode,1), self.beam.ExtractLoadGradient(
      iNode,2), ")")
436
437
438   def PrintSensitivityLoad(self, iNode):
439
440       """ This function prints the sensitivities of the objective functions
      for the single load"""
441       sensX = self.beam.ExtractLoadGradient(iNode,0)
442       sensY = self.beam.ExtractLoadGradient(iNode,1)
443       sensZ = self.beam.ExtractLoadGradient(iNode,2)
444
445       print("F'(",iNode,") = (", sensX, sensY, sensZ, ")")
446
447       return sensX, sensY, sensZ
448
449
450   def PrintSensitivityE(self):
451
452       """ This function prints the sensitivities of the objective functions
      for all the loads"""
453       print("E' = ", self.beam.ExtractGradient_E())
454
455       return self.beam.ExtractGradient_E()
456
457
458   def PrintSensitivityPropDVs(self):
459
460     """ This function prints the sensitivities of the objective functions
     wrt  Prop DVs"""
461     grad_DVs =[]
462     for iPDV in range(0,self.nPropDVs):
463         grad_DVs.append(self.beam.ExtractPropGradient(iPDV))
464         print("Prop DV '(",iPDV,") = (", grad_DVs[iPDV], ")")
465     return grad_DVs
466
467     #return self.beam.ExtractGradient_E()
468
469   def PrintSensitivityPropA(self):
470
471       """ This function prints the sensitivities of the objective functions
      wrt to first Property """
472       print("C_WB or A' = ", self.beam.ExtractGradient_A() )
473       return self.beam.ExtractGradient_A()
474
475
476
477   def TestSensitivityE(self, sensEres, sensECheck):
478
479       """ This function prints the sensitivities of the objective functions
      for all the loads"""
480
481       test_val = 100*abs((sensEres - sensECheck)/sensECheck)
482
```

```
483       # Tolerance is set to 0.0001%
484       if (test_val < 0.000001):
485           print("--> Test E sensitivity: Error to reference {:10.7f}% (<
      0.000001%) -> PASSED".format(test_val))
486           return (0)
487       else:
488           print("--> Test E sensitivity: Error to reference {:10.7f}% (>
      0.000001%) -> FAILED".format(test_val))
489           return (1)
490
491   def TestLoadSensitivity(self, iNode, sensX, sensY, sensZ, sensX_FD,
      sensY_FD, sensZ_FD):
492
493       errorX = 100 * abs((sensX - sensX_FD) / sensX_FD)
494       errorY = 100 * abs((sensY - sensY_FD) / sensY_FD)
495       errorZ = 100 * abs((sensZ - sensZ_FD) / sensZ_FD)
496
497     # Tolerance is set to 1E-8
498       if errorX < 0.5 and errorY < 0.5 and errorZ < 0.5:
499         print("--> Test Load sensitivity: Error to FD below 0.5% -> PASSED"
      )
500         return(0)
501       else:
502         print("--> Test Load sensitivity: Error to FD above 0.5% -> FAILED"
      )
503         return(1)
504
505   def GetInitialCoordinates(self,iVertex):
506
507     """ This function returns the initial coordinates of the structural
      beam model  """
508     coordX = self.beam.ExtractCoordinate0(iVertex, 0)
509     coordY = self.beam.ExtractCoordinate0(iVertex, 1)
510     coordZ = self.beam.ExtractCoordinate0(iVertex, 2)
511
512     return coordX, coordY, coordZ
513
514   def ExtractDisplacements(self,iVertex):
515
516     """ This function returns the initial coordinates of the structural
      beam model  """
517     dispX = self.beam.ExtractDisplacements(iVertex, 0)
518     dispY = self.beam.ExtractDisplacements(iVertex, 1)
519     dispZ = self.beam.ExtractDisplacements(iVertex, 2)
520
521     return dispX, dispY, dispZ
522
523   def SetLowVerbosity(self):
524
525       self.beam.SetLowVerbosity()
526
527   def SetHighVerbosity(self):
528
529       self.beam.SetHighVerbosity()
530
```

```
531
532
533
534
535
```

LISTING B.2. Wrapping Functions

B.2 OpenMDAO functions

```
1
2  from pyBeamIO import pyBeamConfig as pyConfig
3  from pyBeamIO import pyBeamInput as pyInput
4  import numpy as np
5  import math
6  import re
7  import os,os.path
8  import shutil
9  import pyBeam
10 from pyBeamLibAD import pyBeamSolverAD
11
12
13
14 class pyBeamOpt:
15     def __init__(self, file_dir, config_fileName,Loads_Values):
16         self.file_dir = file_dir
17         self.config_fileName = config_fileName
18         self.Config_file = self.file_dir + '/' + config_fileName
19         self.Config = {}
20         self.Config = pyConfig.pyBeamConfig(self.Config_file)  # Beam
       configuration file
21         self.Property = self.file_dir + '/' + self.Config['PROPERTY_FILE']
22         # Parsing Property file
23         self.Prop, self.nProp = pyInput.readProp(self.Property)
24         self.Loads = Loads_Values
25
26
27
28     def SetInitialParameters(self):
29
30         beam = pyBeamSolverAD(self.file_dir, self.config_fileName)
31         [x_tip,y_tip,z_tip] = beam.GetInitialCoordinates(self.nProp)
32         [x_root, y_root, z_root] = beam.GetInitialCoordinates(0)
33
34
35         #L = math.sqrt((x_tip-x_root)**2 + (y_tip-y_root)**2 + (z_tip-
       z_root)**2) # initial length
36         L = x_tip-x_root
37         print("Lunghezza ", L)
38
39         prop_file = open('property.prt', 'r')            #read original
       property file
40         prop_file = prop_file.read()
41         dvs = re.findall(r"[-+]?\d*\.\d+|\d+", prop_file)
```

```
42          dvs = np.delete(dvs, [0, 1, 2, 3, 4, 5])          # Delete old
    format lines from property.prt file
43          DVs= np.array(list(map(float,dvs)))
44          nDVs =len(DVs[0:7])                               # new format
45
46
47
48          return DVs,nDVs,int(self.nProp),L
49
50      def NewDesign(self,DVs):
51
52          """ This function create a folder in which are copied the input
    files,appending the new Design Variables (DVs) in property file   """
53
54          path = os.path.abspath(os.path.join(self.file_dir, ".."))
55          final_directory = os.path.join(path, r'Optimization')
56          if not os.path.exists(final_directory):
57              os.makedirs(final_directory)
58
59          file_names = os.listdir(self.file_dir)
60
61          for file_name in file_names:
62              new_path = os.path.join(self.file_dir, file_name)
63              if os.path.isdir(new_path):
64                  continue
65              shutil.copy(os.path.join(self.file_dir, file_name),
    final_directory)
66
67
68          print("lunghezza",len(DVs))
69          if len(DVs) == 7:
70
71              """ One Property"""
72              Name_prop = os.path.join(final_directory, "property.prt")
73
74              with open(Name_prop, "w") as f_prop:
75                  f_prop.write("% 1ChordofWB  HeightofWB thickskin
    thicksparweb Areasparcap numberstiff Astiff" +
76                               " \n NPROPS=" + str(self.nProp) + "\n" + "S\n"
     + str(DVs[0]) + "   " + str(
77                      DVs[1]) + "   " + str(
78                      DVs[2]) + "   " + str(DVs[3]) + "   " + str(DVs[4]) + "
    " + str(int(DVs[5])) + "   " + str(DVs[6]))
79                  f_prop.close()
80
81          else:
82
83              DVs = DVs.reshape(self.nProp, 7)
84              DVs = np.matrix(DVs)
85              Name_prop = os.path.join(final_directory, "property_opt.prt")
86              Name_prop_opt = os.path.join(final_directory, "property.prt")
87              n_stiff = int(DVs[0, 5])
88
89              with open(Name_prop, "w") as f_prop:
90                  for line in DVs:
```

```
                    np.savetxt(f_prop, line, fmt='%.2f')

            f_prop.close()


            with open(Name_prop, 'r') as f:
                with open(Name_prop_opt, 'w') as outfile:
                    outfile.write("% ChordofWB  HeightofWB thickskin thicksparweb Areasparcap numberstiff Astiff" +
                                  " \n NPROPS=" + str(self.nProp) + "\n" + "S\n")
                with open(Name_prop_opt, 'a') as outfile:
                    for line in f:
                        line = line.split()
                        new_line = '{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}'.format(line[0], line[1],line[2],line[3],line[4],n_stiff,line[6])
                        outfile.write(new_line + "\n")
            os.remove(Name_prop)
            f.close()


        return final_directory

    def ComputeResponseKSStress_opt_lin(self, DVs):

        final_directory = self.NewDesign( DVs)
        beam = pyBeamSolverAD(final_directory, self.config_fileName)
        beam.SetLoads(self.Loads[0],self.Loads[1],self.Loads[2], self.Loads[3])
        beam.RunLin()
        KS=beam.ComputeResponseKSStress()

        del beam.file_dir
        Name_KS = os.path.join(final_directory, "constraint_KS.txt")

        with open(Name_KS, "w") as f_KS:
            f_KS.write(str(KS))
            f_KS.close()

        return KS

    def ComputeResponseKSStress_opt_non_lin(self, DVs):

        final_directory = self.NewDesign( DVs)
        beam = pyBeamSolverAD(final_directory, self.config_fileName)
        beam.SetLoads(self.Loads[0],self.Loads[1],self.Loads[2], self.Loads[3])
        beam.Run()
        KS= beam.ComputeResponseKSStress()
        del beam.file_dir
        Name_KS = os.path.join(final_directory, "constraint_KS.txt")

        with open(Name_KS, "w") as f_KS:
            f_KS.write(str(KS))
            f_KS.close()

```

```
142         return KS
143 
144     def ComputeResponseKSBuckling_opt_lin(self, DVs):
145 
146         final_directory = self.NewDesign(DVs)
147         beam = pyBeamSolverAD(final_directory, self.config_fileName)
148         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
149         beam.RunLin()
150         KS_buckl = beam.ComputeResponseKSBuckling()
151 
152         del beam.file_dir
153         Name_KS_buckl = os.path.join(final_directory, "
    constraint_KS_buckling.txt")
154 
155         with open(Name_KS_buckl, "w") as f_KS:
156             f_KS.write(str(KS_buckl))
157             f_KS.close()
158         print("Buckling", KS_buckl)
159         return KS_buckl
160 
161     def ComputeResponseKSBuckling_opt_non_lin(self, DVs):
162 
163         final_directory = self.NewDesign(DVs)
164         beam = pyBeamSolverAD(final_directory, self.config_fileName)
165         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
166         beam.Run()
167         KS_buckl = beam.ComputeResponseKSBuckling()
168         del beam.file_dir
169         Name_KS_buckl = os.path.join(final_directory, "
    constraint_KS_Buckling.txt")
170 
171         with open(Name_KS_buckl, "w") as f_KS:
172             f_KS.write(str(KS_buckl))
173             f_KS.close()
174 
175         return KS_buckl
176 
177 
178 
179 
180     def ComputeWeight_opt( self,  DVs):
181 
182           final_directory = self.NewDesign(DVs)
183           beam = pyBeamSolverAD(final_directory, self.config_fileName)
184           """ This function computes the response weight of the structure (
    important to be recorded) """
185           weight = beam.ComputeWeight()
186           del beam.file_dir
187 
188           Name_weight = os.path.join(final_directory,"weight.txt")
189 
190           with open(Name_weight, "w") as f_weight:
191               f_weight.write(str(weight))
```

```
192             f_weight.close()
193 
194         return weight
195 
196     def ComputeAdjointWeight_opt(self, DVs):
197 
198         final_directory = self.NewDesign(DVs)
199         beam = pyBeamSolverAD(final_directory, self.config_fileName)
200         """ This function computes the response weight of the structure (
    important to be recorded) """
201 
202         # ---- Ad-hoc for
203         beam.StartRecording()
204         beam.SetDependencies()
205         # ----------------------
206         beam.ComputeWeight()
207 
208         beam.StopRecordingWeight()
209 
210         beam.ComputeAdjointWeight()
211 
212         Weight_sens = beam.PrintSensitivityPropDVs()
213 
214 
215         del beam.file_dir
216 
217 
218         Name_weight_sens = os.path.join(final_directory, "weight_sens.txt")
219 
220         with open(Name_weight_sens, "w") as f_weight:
221             f_weight.write(str(Weight_sens))
222             f_weight.close()
223 
224         return Weight_sens
225 
226     def ComputeAdjointKSstresses_opt_lin(self, DVs):
227 
228         final_directory = self.NewDesign(DVs)
229         beam = pyBeamSolverAD(final_directory, self.config_fileName)
230         """ This function computes the response weight of the structure (
    important to be recorded) """
231 
232         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
233         beam.RunLin()
234         # ---- Ad-hoc for
235 
236         restart_file = 'restart.pyBeam'
237         restart_dir = os.getcwd()
238         restart_file_location = os.path.join( restart_dir, restart_file)
239 
240         solution_file='solution.pyBeam'
241         solution_dir = os.getcwd()
242         solution_file_location=os.path.join(solution_dir, solution_file)
243 
```

```
244         with open(restart_file_location, 'r') as f1:
245             with open(solution_file_location, 'w') as f2:
246                 for line in f1:
247                     f2.write(line)
248 
249         beam.ReadRestart()
250 
251         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
252 
253 
254         beam.StartRecording()
255         beam.SetDependencies()
256         # ----------------------
257         beam.RestartLin()
258 
259         beam.ComputeResponseKSStress()
260         #beam.ComputeResponseSigmaBoom()
261 
262         # ---- Ad-hoc for
263         #beam.StopRecordingSigmaBoom()
264 
265         # ---------------
266         #beam.ComputeAdjointSigmaBoom()
267         # ----------------------
268         # beam.PrintSensitivitiesAllLoads()
269         beam.StopRecordingKSstresses()
270 
271         # ---------------
272         beam.ComputeAdjointKSStresses()
273 
274         KSstress_sens = beam.PrintSensitivityPropDVs()
275 
276         del beam.file_dir
277 
278         return KSstress_sens
279 
280 
281 
282 
283     def ComputeAdjointKSstresses_opt_non_lin(self, DVs):
284 
285         final_directory = self.NewDesign(DVs)
286         beam = pyBeamSolverAD(final_directory, self.config_fileName)
287         """ This function computes the response weight of the structure (
    important to be recorded) """
288 
289         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
290         beam.Run()
291         # ---- Ad-hoc for
292 
293         restart_file = 'restart.pyBeam'
294         restart_dir = os.getcwd()
295         restart_file_location = os.path.join( restart_dir, restart_file)
```

```
296
297         solution_file='solution.pyBeam'
298         solution_dir = os.getcwd()
299         solution_file_location=os.path.join(solution_dir, solution_file)
300
301         with open(restart_file_location, 'r') as f1:
302             with open(solution_file_location, 'w') as f2:
303                 for line in f1:
304                     f2.write(line)
305
306         beam.ReadRestart()
307
308         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
309
310
311         beam.StartRecording()
312         beam.SetDependencies()
313         # ----------------------
314         beam.Restart()
315         beam.ComputeResponseKSStress()
316
317
318         # ---- Ad-hoc for
319         beam.StopRecordingKSstresses()
320
321         # ---------------
322         beam.ComputeAdjointKSStresses()
323         # ----------------------
324
325         KSstress_sens=beam.PrintSensitivityPropDVs()
326
327         del beam.file_dir
328
329
330         return KSstress_sens
331
332     def ComputeAdjointKSBuckling_opt_lin(self, DVs):
333
334         final_directory = self.NewDesign(DVs)
335         beam = pyBeamSolverAD(final_directory, self.config_fileName)
336         """ This function computes the response weight of the structure (
    important to be recorded) """
337
338         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
339         beam.RunLin()
340         # ---- Ad-hoc for
341
342         restart_file = 'restart.pyBeam'
343         restart_dir = os.getcwd()
344         restart_file_location = os.path.join(restart_dir, restart_file)
345
346         solution_file = 'solution.pyBeam'
347         solution_dir = os.getcwd()
```

```
348         solution_file_location = os.path.join(solution_dir, solution_file)
349 
350         with open(restart_file_location, 'r') as f1:
351             with open(solution_file_location, 'w') as f2:
352                 for line in f1:
353                     f2.write(line)
354 
355         beam.ReadRestart()
356 
357         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
358 
359         beam.StartRecording()
360         beam.SetDependencies()
361         # ----------------------
362         beam.RestartLin()
363 
364         beam.ComputeResponseKSBuckling()
365         # beam.ComputeResponseSigmaBoom()
366 
367         # ---- Ad-hoc for
368         # beam.StopRecordingSigmaBoom()
369 
370         # ---------------
371         # beam.ComputeAdjointSigmaBoom()
372         # ----------------------
373         # beam.PrintSensitivitiesAllLoads()
374         beam.StopRecordingKSbuckling()
375 
376         # ---------------
377         beam.ComputeAdjointKSBuckling()
378 
379         KSbuckl_sens = beam.PrintSensitivityPropDVs()
380 
381         del beam.file_dir
382 
383         return KSbuckl_sens
384 
385     def ComputeAdjointKSBuckling_opt_non_lin(self, DVs):
386 
387         final_directory = self.NewDesign(DVs)
388         beam = pyBeamSolverAD(final_directory, self.config_fileName)
389         """ This function computes the response weight of the structure (
    important to be recorded) """
390 
391         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
392         beam.Run()
393         # ---- Ad-hoc for
394 
395         restart_file = 'restart.pyBeam'
396         restart_dir = os.getcwd()
397         restart_file_location = os.path.join(restart_dir, restart_file)
398 
399         solution_file = 'solution.pyBeam'
```

```
400         solution_dir = os.getcwd()
401         solution_file_location = os.path.join(solution_dir, solution_file)
402 
403         with open(restart_file_location, 'r') as f1:
404             with open(solution_file_location, 'w') as f2:
405                 for line in f1:
406                     f2.write(line)
407 
408         beam.ReadRestart()
409 
410         beam.SetLoads(self.Loads[0], self.Loads[1], self.Loads[2], self.
    Loads[3])
411 
412         beam.StartRecording()
413         beam.SetDependencies()
414         # ----------------------
415         beam.Restart()
416         beam.ComputeResponseKSBuckling()
417 
418         # ---- Ad-hoc for
419         beam.StopRecordingKSbuckling()
420 
421         # ---------------
422         beam.ComputeAdjointKSBuckling()
423         # ----------------------
424 
425         KSbuckl_sens = beam.PrintSensitivityPropDVs()
426 
427         del beam.file_dir
428 
429         return KSbuckl_sens
```

LISTING B.3. pyBeamOpt class

B.3 Regr_optimization_openMDAO_AD

```
1  import numpy as np
2  import os,os.path
3  import shutil
4  import openmdao.api as om
5  import sys
6  from PIL import Image
7  from pyBeamLibAD import pyBeamSolverAD
8  from pyBeamOpenMDAO import pyBeamOpt
9  from pprint import pprint
10 import sqlitedict
11 import matplotlib.pyplot as plt
12 
13 
14 class Weight(om.ExplicitComponent):
15 
16     def setup(self):
17 
18         self.add_input('C_wb',shape = len(C_wb),units='mm')
```

```
19          self.add_input('h', shape = len(h),units='mm')
20          self.add_input('t_sk',shape = len(t_sk),units='mm')
21          self.add_input('t_sp',shape = len(t_sp),units='mm')
22          self.add_input('A_fl',shape = len(A_fl),units='mm**2')
23          self.add_input('n_stiff',shape = len(n_stiff),units='mm')
24          self.add_input('A_stiff',shape = len(A_stiff),units='mm**2')
25
26
27
28          self.add_output('Obj_f')
29
30          # Finite difference all partials.
31          self.declare_partials('Obj_f', 'C_wb')
32          self.declare_partials('Obj_f', 'h')
33          self.declare_partials('Obj_f', 't_sk')
34          self.declare_partials('Obj_f', 't_sp')
35          self.declare_partials('Obj_f', 'A_fl')
36          self.declare_partials('Obj_f', 'A_stiff')
37
38      def compute(self, inputs, outputs):
39
40
41          C_wb = inputs['C_wb']
42          h    = inputs['h']
43          t_sk = inputs['t_sk']
44          t_sp = inputs['t_sp']
45          A_fl = inputs['A_fl']
46          n_stiff = inputs['n_stiff']
47          A_stiff = inputs['A_stiff']
48
49          DesignVArs = np.array([C_wb, h, t_sk, t_sp, A_fl, n_stiff, A_stiff
    ]).transpose()
50          DesignVArs = np.array(DesignVArs.tolist())
51          DesignVArs = np.reshape(DesignVArs, DesignVArs.size)
52
53
54
55
56          outputs['Obj_f'] = beam_opt.ComputeWeight_opt(DesignVArs)
57
58
59
60      def compute_partials(self,inputs,partials):
61
62          C_wb = inputs['C_wb']
63          h = inputs['h']
64          t_sk = inputs['t_sk']
65          t_sp = inputs['t_sp']
66          A_fl = inputs['A_fl']
67          n_stiff = inputs['n_stiff']
68          A_stiff = inputs['A_stiff']
69
70          DesignVArs = np.array([C_wb, h, t_sk, t_sp, A_fl, n_stiff, A_stiff
    ]).transpose()
71          DesignVArs = np.array(DesignVArs.tolist())
```

```
72          DesignVArs = np.reshape(DesignVArs, DesignVArs.size)
73
74          Weight_adj=beam_opt.ComputeAdjointWeight_opt(DesignVArs)
75
76          partials['Obj_f', 'C_wb']=Weight_adj[::6]
77          partials['Obj_f', 'h'] = Weight_adj[1::6]
78          partials['Obj_f', 't_sk'] = Weight_adj[2::6]
79          partials['Obj_f', 't_sp'] = Weight_adj[3::6]
80          partials['Obj_f', 'A_fl'] = Weight_adj[4::6]
81          partials['Obj_f', 'A_stiff'] = Weight_adj[5::6]
82
83
84
85
86  class KSStress_constraint(om.ExplicitComponent):
87
88      def setup(self):
89          self.add_input('C_wb',shape=len(C_wb),units='mm')
90          self.add_input('h',shape=len(h),units='mm')
91          self.add_input('t_sk',shape=len(t_sk),units='mm')
92          self.add_input('t_sp',shape=len(t_sp),units='mm')
93          self.add_input('A_fl',shape=len(A_fl),units='mm**2')
94          self.add_input('n_stiff',shape=len(n_stiff),units='mm')
95          self.add_input('A_stiff',shape=len(A_stiff),units='mm**2')
96
97
98          self.add_output('Const_KS')
99
100         # Finite difference all partials.
101
102         self.declare_partials('Const_KS', 'C_wb')
103         self.declare_partials('Const_KS', 'h')
104         self.declare_partials('Const_KS', 't_sk')
105         self.declare_partials('Const_KS', 't_sp')
106         self.declare_partials('Const_KS', 'A_fl')
107         self.declare_partials('Const_KS', 'A_stiff')
108
109     def compute(self, inputs, outputs):
110
111
112         C_wb = inputs['C_wb']
113         h    = inputs['h']
114         t_sk = inputs['t_sk']
115         t_sp = inputs['t_sp']
116         A_fl = inputs['A_fl']
117         n_stiff = inputs['n_stiff']
118         A_stiff = inputs['A_stiff']
119
120         DesignVArs = np.array([C_wb, h, t_sk, t_sp, A_fl, n_stiff, A_stiff
    ]).transpose()
121         DesignVArs = np.array(DesignVArs.tolist())
122         DesignVArs = np.reshape(DesignVArs, DesignVArs.size)
123
124         if flag_lin == 'lin':
125          outputs['Const_KS'] = beam_opt.ComputeResponseKSStress_opt_lin(
```

```
    DesignVArs)
126         elif flag_lin == 'nonlin':
127             outputs['Const_KS'] = beam_opt.
    ComputeResponseKSStress_opt_non_lin(DesignVArs)
128
129
130     def compute_partials(self,inputs,partials):
131
132         C_wb = inputs['C_wb']
133         h = inputs['h']
134         t_sk = inputs['t_sk']
135         t_sp = inputs['t_sp']
136         A_fl = inputs['A_fl']
137         n_stiff = inputs['n_stiff']
138         A_stiff = inputs['A_stiff']
139
140         DesignVArs = np.array([C_wb, h, t_sk, t_sp, A_fl, n_stiff, A_stiff
    ]).transpose()
141         DesignVArs = np.array(DesignVArs.tolist())
142         DesignVArs = np.reshape(DesignVArs, DesignVArs.size)
143
144         if flag_lin == 'lin':
145             KS_adj = beam_opt.ComputeAdjointKSstresses_opt_lin(DesignVArs)
146         elif flag_lin == 'nonlin':
147             KS_adj=beam_opt.ComputeAdjointKSstresses_opt_non_lin(DesignVArs
    )
148
149
150         partials['Const_KS', 'C_wb'] = KS_adj[0::6]
151         partials['Const_KS', 'h']    = KS_adj[1::6]
152         partials['Const_KS', 't_sk'] = KS_adj[2::6]
153         partials['Const_KS', 't_sp'] = KS_adj[3::6]
154         partials['Const_KS', 'A_fl'] = KS_adj[4::6]
155         partials['Const_KS', 'A_stiff'] = KS_adj[5::6]
156
157
158
159 class KSBuckling_constraint(om.ExplicitComponent):
160
161             def setup(self):
162                 self.add_input('C_wb', shape=len(C_wb), units='mm')
163                 self.add_input('h', shape=len(h), units='mm')
164                 self.add_input('t_sk', shape=len(t_sk), units='mm')
165                 self.add_input('t_sp', shape=len(t_sp), units='mm')
166                 self.add_input('A_fl', shape=len(A_fl), units='mm**2')
167                 self.add_input('n_stiff', shape=len(n_stiff), units='mm')
168                 self.add_input('A_stiff', shape=len(A_stiff), units='mm**2'
    )
169
170                 self.add_output('Const_KS_buckl')
171
172                 # Finite difference all partials.
173
174                 self.declare_partials('Const_KS_buckl', 'C_wb')
175                 self.declare_partials('Const_KS_buckl', 'h')
```

```
176                 self.declare_partials('Const_KS_buckl', 't_sk')
177                 self.declare_partials('Const_KS_buckl', 't_sp')
178                 self.declare_partials('Const_KS_buckl', 'A_fl')
179                 self.declare_partials('Const_KS_buckl', 'A_stiff')
180 
181             def compute(self, inputs, outputs):
182 
183                 C_wb = inputs['C_wb']
184                 h = inputs['h']
185                 t_sk = inputs['t_sk']
186                 t_sp = inputs['t_sp']
187                 A_fl = inputs['A_fl']
188                 n_stiff = inputs['n_stiff']
189                 A_stiff = inputs['A_stiff']
190 
191                 DesignVArs = np.array([C_wb, h, t_sk, t_sp, A_fl, n_stiff,
    A_stiff]).transpose()
192                 DesignVArs = np.array(DesignVArs.tolist())
193                 DesignVArs = np.reshape(DesignVArs, DesignVArs.size)
194 
195                 if flag_lin == 'lin':
196                     outputs['Const_KS_buckl'] = beam_opt.
    ComputeResponseKSBuckling_opt_lin(DesignVArs)
197                 elif flag_lin == 'nonlin':
198                     outputs['Const_KS_buckl'] = beam_opt.
    ComputeResponseKSBuckling_opt_non_lin(DesignVArs)
199 
200             def compute_partials(self, inputs, partials):
201 
202                 C_wb = inputs['C_wb']
203                 h = inputs['h']
204                 t_sk = inputs['t_sk']
205                 t_sp = inputs['t_sp']
206                 A_fl = inputs['A_fl']
207                 n_stiff = inputs['n_stiff']
208                 A_stiff = inputs['A_stiff']
209 
210                 DesignVArs = np.array([C_wb, h, t_sk, t_sp, A_fl, n_stiff,
    A_stiff]).transpose()
211                 DesignVArs = np.array(DesignVArs.tolist())
212                 DesignVArs = np.reshape(DesignVArs, DesignVArs.size)
213 
214                 if flag_lin == 'lin':
215                     KS_adj = beam_opt.ComputeAdjointKSBuckling_opt_lin(
    DesignVArs)
216                 elif flag_lin == 'nonlin':
217                     KS_adj = beam_opt.ComputeAdjointKSBuckling_opt_non_lin(
    DesignVArs)
218 
219                 partials['Const_KS_buckl', 'C_wb'] = KS_adj[0::6]
220                 partials['Const_KS_buckl', 'h'] = KS_adj[1::6]
221                 partials['Const_KS_buckl', 't_sk'] = KS_adj[2::6]
222                 partials['Const_KS_buckl', 't_sp'] = KS_adj[3::6]
223                 partials['Const_KS_buckl', 'A_fl'] = KS_adj[4::6]
224                 partials['Const_KS_buckl', 'A_stiff'] = KS_adj[5::6]
```

```
225
226
227
228
229
230
231
232 if __name__ == "__main__":
233
234
235
236
237     prob = om.Problem()
238
239     flag_lin = input('Which optimization do you want to run ? (lin or
        nonlin) \n')
240
241     Loads = (19, -1000 ,0, 5000)
242     file_dir = os.path.dirname(os.path.realpath(__file__))
243
244     path_opt = os.path.abspath(os.path.join(file_dir, ".."))
245     final_directory_opt = os.path.join(path_opt, r'Optimization')
246     if os.path.exists(final_directory_opt):
247         shutil.rmtree(final_directory_opt)
248     os.makedirs(final_directory_opt)
249
250
251
252
253     if  flag_lin == 'lin':
254
255         beam_opt = pyBeamOpt(file_dir, 'config_lin.cfg', Loads)
256     elif flag_lin =='nonlin':
257
258         beam_opt = pyBeamOpt(file_dir, 'config_non_lin.cfg', Loads)
259     else:
260         raise ValueError("the type of analysis to be performed is missing .
        Execution aborted")
261
262
263
264
265     DVs    = (beam_opt.SetInitialParameters()[0])
266     nDVs   = beam_opt.SetInitialParameters()[1]
267     nProp = beam_opt.SetInitialParameters()[2]
268     L      =beam_opt.SetInitialParameters()[3]
269     print("Length", L)
270
271     C_wb = DVs[::nDVs]
272     h = DVs[1::nDVs]
273     t_sk = DVs[2::nDVs]
274     t_sp = DVs[3::nDVs]
275     A_fl = DVs[4::nDVs]
276     n_stiff = DVs[5::nDVs]
277     A_stiff = DVs[6::nDVs]
```

```
278
279     prob.model.add_subsystem('weight_comp', Weight(),
280                              promotes_inputs=['C_wb', 'h', 't_sk', 't_sp',
    'A_fl', 'n_stiff', 'A_stiff'])
281
282     prob.model.add_subsystem('KS_comp_stress', KSStress_constraint(),
283                              promotes_inputs=['C_wb', 'h', 't_sk', 't_sp',
    'A_fl','n_stiff','A_stiff'])
284
285     prob.model.add_subsystem('KS_comp_buckl', KSBuckling_constraint(),
286                              promotes_inputs=['C_wb', 'h', 't_sk', 't_sp',
    'A_fl', 'n_stiff', 'A_stiff'])
287     if not n_stiff.all() == 0:
288         prob.model.add_subsystem('Astiff_Afl_comp', om.ExecComp('g1 =
    A_stiff-0.25*A_fl', g1=np.ones(nProp),
289                                                                 A_stiff=np.
    ones(nProp), A_fl=np.ones(nProp)),promotes=['*'])
290
291     # If properties > 1
292
293     if nProp > 1:
294         prob.model.add_subsystem('t_sk_con', om.ExecComp('g2 = t_sk[0:-1] -
     t_sk[1:]', g2=np.ones(nProp-1), t_sk=np.ones(nProp)),
295                                                             promotes=['*'])
296         prob.model.add_subsystem('t_sp_con', om.ExecComp('g3 = t_sp[0:-1] -
     t_sp[1:]', g3=np.ones(nProp-1), t_sp=np.ones(nProp)),
297                                                            promotes=['*'])
298         prob.model.add_subsystem('Afl_con',om.ExecComp('g4 = A_fl[0:-1] -
    A_fl[1:]', g4=np.ones(nProp- 1),A_fl=np.ones(nProp)),
299                                  promotes=['*'])
300         if not n_stiff.all() == 0:
301             prob.model.add_subsystem('Astiff_con',om.ExecComp('g5 = A_stiff
    [0:-1] - A_stiff[1:]', g5=np.ones((nProp)-1),
302                                                     A_stiff=np.ones(nProp)),
    promotes=['*'])
303
304
305     # setup the optimization
306
307     prob.driver = om.pyOptSparseDriver()
308     #prob.driver = om.ScipyOptimizeDriver()
309     prob.driver.options['optimizer'] = 'SLSQP'
310
311
312     #prob.driver.options['maxiter'] = 10
313     prob.driver.options['debug_print']=['desvars','objs','nl_cons','totals'
    ]
314
315     prob.model.set_input_defaults('C_wb', C_wb, units='mm')
316     prob.model.set_input_defaults('h', h, units='mm')
317     prob.model.set_input_defaults('t_sk', t_sk, units='mm')
318     prob.model.set_input_defaults('t_sp', t_sp, units='mm')
319     prob.model.set_input_defaults('A_fl', A_fl, units='mm**2')
320     prob.model.set_input_defaults('n_stiff', n_stiff, units='mm')
321     prob.model.set_input_defaults('A_stiff', A_stiff, units='mm**2')
```

```
322
323
324
325
326     filename = "opt_results.sql"
327     recorder = om.SqliteRecorder(filename)
328     prob.driver.add_recorder(recorder)
329     prob.driver.recording_options['record_desvars'] = True
330     prob.driver.recording_options['record_objectives'] = True
331     prob.driver.recording_options['record_constraints'] = True
332     prob.driver.recording_options['record_derivatives'] = True
333     prob.driver.recording_options['includes'] = []
334     prob.driver.recording_options['excludes'] = []
335
336
337     """""""""""""""""""""""""""""""" ADD constraints , Objective function
    and design variables """""""""""""""""""""""""""""""""
338
339     #prob.model.add_design_var('C_wb')
340     #prob.model.add_design_var('h')
341     prob.model.add_design_var('t_sk', lower=0.5)
342     prob.model.add_design_var('t_sp', lower=0.5)
343     prob.model.add_design_var('A_fl',lower=0)
344     if not n_stiff.all() == 0:
345         prob.model.add_design_var('A_stiff',lower= 10 )
346
347     prob.model.add_objective('weight_comp.Obj_f')
348
349     prob.model.add_constraint('KS_comp_stress.Const_KS', upper=0.0)
350
351     prob.model.add_constraint('KS_comp_buckl.Const_KS_buckl', upper=0.0)
352
353     if not n_stiff.all() == 0:
354         prob.model.add_constraint('g1', upper=0)
355     if nProp > 1:
356         prob.model.add_constraint('g2', equals=0)
357         prob.model.add_constraint('g3', equals=0)
358         prob.model.add_constraint('g4', equals=0)
359         if not n_stiff.all() == 0:
360             prob.model.add_constraint('g5', equals=0)
361
362     prob.set_solver_print(level=0)
363
364     prob.setup(check=False, mode='rev')
365
366
367
368     prob.run_driver()
369
370     print("Minimum Weight =", prob.get_val('weight_comp.Obj_f'))
371     print("Constraint_stress =", prob.get_val('KS_comp_stress.Const_KS'))
372     print("Constraint_buckling =", prob.get_val('KS_comp_buckl.
    Const_KS_buckl'))
373
374     prob.cleanup()
```

```
375
376
377     """  Record Data : """
378
379
380     cr = om.CaseReader("opt_results.sql")
381
382     case_names = cr.list_cases(out_stream=None)
383
384     path = os.path.abspath(os.path.join(file_dir, "../Optimization"))
385     final_directory = os.path.join(path, r'opt_iter')
386     final_directory_figures = os.path.join(path, r'Figures')
387     if not os.path.exists(final_directory):
388         os.makedirs(final_directory)
389     if not os.path.exists(final_directory_figures):
390         os.makedirs(final_directory_figures)
391
392     path_iter = os.path.abspath(os.path.join(file_dir, "../Optimization/
    opt_iter"))
393     path_figures = os.path.abspath(os.path.join(file_dir, "../Optimization/
    Figures"))
394
395
396
397
398
399     for it in range(0,len(case_names),1):
400
401             case = cr.get_case(it)
402             derivs=cr.get_case(it).derivatives
403
404             final_directory_iter = os.path.join(path_iter, r'000'+ str(it))
405             if not os.path.exists(final_directory_iter):
406                 os.makedirs(final_directory_iter)
407
408             if derivs is not None :
409              Weight_deriv = os.path.join(final_directory_iter, "Weight_sens
    " + ".txt")
410
411              Deriv_obj = open(Weight_deriv, "w")
412              Deriv_obj.write(
413                               'Weight wrt t_sk = ' + str(derivs['weight_comp
    .Obj_f', 't_sk']) + '\n'+
414                               'Weight wrt t_sp = ' + str(derivs['weight_comp
    .Obj_f', 't_sp']) + '\n'+
415                               'Weight wrt A_fl = ' + str(derivs['weight_comp
    .Obj_f', 'A_fl']) + '\n')
416              if not n_stiff.all() == 0:
417                 Deriv_obj.write('Weight wrt A_stiff = ' + str(derivs['
    weight_comp.Obj_f', 'A_stiff']))
418              Deriv_obj.close()
419
420
421              KS_deriv_buckl = os.path.join(final_directory_iter, "
    KSBuckling_sens" + ".txt")
```

```
422             Deriv_KS_buckl = open(KS_deriv_buckl, "w")
423             Deriv_KS_buckl.write(
424                             'KS_buckl wrt t_sk = ' + str(derivs['
    KS_comp_buckl.Const_KS_buckl', 't_sk']) + '\n'
425                             'KS_buckl wrt t_sp = ' + str(derivs['
    KS_comp_buckl.Const_KS_buckl', 't_sp'])+ '\n')
426             if not n_stiff.all() == 0:
427                 Deriv_KS_buckl.write('KS_buckl wrt A_stiff = ' + str(
    derivs['KS_comp_buckl.Const_KS_buckl', 'A_stiff']))
428             Deriv_KS_buckl.close()
429
430
431
432             KS_deriv = os.path.join(final_directory_iter, "KSStress_sens"
    + ".txt")
433             Deriv_KS = open(KS_deriv, "w")
434             Deriv_KS.write(
435                 'KS wrt t_sk = ' + str(derivs['KS_comp_stress.Const_KS', '
    t_sk']) + '\n'
436
       'KS wrt t_sp = ' + str(
437                         derivs['KS_comp_stress.Const_KS', 't_sp']) + '\n')
438             if not n_stiff.all() == 0:
439                 Deriv_KS.write('KS wrt A_stiff = ' + str(derivs['
    KS_comp_stress.Const_KS', 'A_stiff']))
440             Deriv_KS.close()
441
442
443             if derivs is None:
444                 pass
445
446             Obj_func = os.path.join(final_directory_iter,  "Weight" + ".txt
    ")
447
448             outputs_obj = open(Obj_func , "w")
449             outputs_obj.write(str(case['weight_comp.Obj_f']))
450             outputs_obj.close()
451
452             KS = os.path.join(final_directory_iter, "KSstress_constraint" +
     ".txt")
453
454             outputs_KS = open(KS, "w")
455             outputs_KS.write(str(case['KS_comp_stress.Const_KS']))
456             outputs_KS.close()
457
458             KS_buckl = os.path.join(final_directory_iter, "
    KSBuckling_constraint" + ".txt")
459
460             outputs_KS_buckl = open(KS_buckl, "w")
461             outputs_KS_buckl.write(str(case['KS_comp_buckl.Const_KS_buckl'
    ]))
462             outputs_KS_buckl.close()
463
464
465
```

```
466             t_skin = os.path.join(final_directory_iter, "t_sk" + ".txt")
467 
468             outputs_t_sk = open(t_skin, "w")
469             outputs_t_sk.write(str(case['t_sk']))
470             outputs_t_sk.close()
471 
472             t_spar= os.path.join(final_directory_iter, "t_sp" + ".txt")
473 
474             outputs_t_sp = open(t_spar, "w")
475             outputs_t_sp.write(str(case['t_sp']))
476             outputs_t_sp.close()
477 
478 
479             A_flanges = os.path.join(final_directory_iter, "A_fl" + ".txt")
480 
481             outputs_A_fl = open(A_flanges, "w")
482             outputs_A_fl.write(str(case['A_fl']))
483             outputs_A_fl.close()
484 
485             if not n_stiff.all() == 0 :
486                 A_stiffeners = os.path.join(final_directory_iter, "A_stiff"
    + ".txt")
487 
488                 outputs_A_stiff = open(A_stiffeners, "w")
489                 outputs_A_stiff.write(str(case['A_stiff']))
490                 outputs_A_stiff.close()
491 
492 
493 
494     """                             PLOT                                """
495 
496 
497     """Weight"""
498     fig1= plt.figure()
499     for i in range(0, len(case_names), 1):
500         case_plot = cr.get_case(i)
501         plt.plot(i, float(case_plot['weight_comp.Obj_f']), 'ro', label='
    Weight')
502     plt.legend(['Weight'])
503     plt.xlabel('Driver Iterations', fontsize=18)
504     plt.ylabel('Weight', fontsize=16)
505 
506     plt.savefig('../Optimization/Figures/ONERA_Weight_' + str(int(
507             n_stiff[0])) + '_stiff_' + str(flag_lin)+'.pdf')  # save the
    figure to file
508     plt.close(fig1)
509 
510     """KS constraint"""
511     fig2=plt.figure()
512     for i in range(0, len(case_names), 1):
513         case_plot = cr.get_case(i)
514         plt.plot(i, float(case_plot['KS_comp_stress.Const_KS']), 'ro')
515     plt.legend(['KS_Stresses_constraint'])
516     plt.xlabel('Driver Iterations', fontsize=18)
517     plt.ylabel('KS_stresses Constraint', fontsize=16)
```

```
518     plt.savefig('../Optimization/Figures/ONERA_KSStressconstraint_' + str(
    int(
519             n_stiff[0]))  + '_stiff_' + str(flag_lin)+'.pdf')  # save the
    figure to file
520     plt.close(fig2)
521
522
523
524     """KS_Buckling constraint"""
525     fig3 = plt.figure()
526     for i in range(0, len(case_names), 1):
527         case_plot = cr.get_case(i)
528         plt.plot(i, float(case_plot['KS_comp_buckl.Const_KS_buckl']), 'ro')
529     plt.legend(['KS_Buckling_constraint'])
530     plt.xlabel('Driver Iterations', fontsize=18)
531     plt.ylabel('KS_Buckling Constraint', fontsize=16)
532     plt.savefig('../Optimization/Figures/ONERA_KSBucklingconstraint_' + str
    (int(
533         n_stiff[0])) + '_stiff_' + str(flag_lin) + '.pdf')  # save the
    figure to file
534     plt.close(fig3)
535
536
537     """Design Variables """
538
539     if nProp > 1:
540
541         """Design Variables property > 1"""
542
543         fig_root = plt.figure()
544         """Root"""
545
546         for i in range(0, len(case_names), 1):
547             case_plot = cr.get_case(i)
548             plt.plot(i, float(case_plot['t_sk'][0]), 'g^', label='t_sk')
549             plt.plot(i, float(case_plot['t_sp'][0]), 'yo', label='t_sp')
550             plt.plot(i, float(case_plot['A_fl'][0]), 'k+', label='A_fl')
551             if not n_stiff.all() == 0:
552              plt.plot(i, float(case_plot['A_stiff'][0]), 'co', label='
    A_stiff')
553
554
555
556         if not n_stiff.all() == 0:
557             plt.legend(['t_sk', 't_sp', 'A_fl','A_stiff'], bbox_to_anchor
    =(1.05, 1), loc='upper left')
558         plt.legend([ 't_sk', 't_sp', 'A_fl'], bbox_to_anchor=(1.05, 1), loc
    ='upper left')
559         plt.xlabel('Driver Iterations', fontsize=18)
560         plt.ylabel('Design Variables', fontsize=16)
561         plt.tight_layout()
562         plt.savefig('../Optimization/Figures/ONERA_DesignVAriables_at_root_
    ' + str(int(
563             n_stiff[0])) + '_stiff_' + str(flag_lin)+'.pdf')  # save the
    figure to file
```

```
564         plt.close(fig_root)
565 
566         fig_tip = plt.figure()
567         """Tip"""
568         for i in range(0, len(case_names), 1):
569             case_plot = cr.get_case(i)
570             plt.plot(i, float(case_plot['t_sk'][-1]), 'g^', label='t_sk')
571             plt.plot(i, float(case_plot['t_sp'][-1]), 'yo', label='t_sp')
572             plt.plot(i, float(case_plot['A_fl'][-1]), 'k+', label='A_fl')
573             if not n_stiff.all() == 0:
574                 plt.plot(i, float(case_plot['A_stiff'][-1]), 'co', label='
    A_stiff')
575 
576         if not n_stiff.all() == 0:
577             plt.legend(['t_sk', 't_sp', 'A_fl', 'A_stiff'], bbox_to_anchor
    =(1.05, 1), loc='upper left')
578         else:
579             plt.legend(['t_sk', 't_sp', 'A_fl'], bbox_to_anchor=(1.05, 1),
    loc='upper left')
580 
581         plt.xlabel('Driver Iterations', fontsize=18)
582         plt.ylabel('Design Variables', fontsize=16)
583         plt.tight_layout()
584         plt.savefig('../Optimization/Figures/ONERA_DesignVAriables_at_tip_'
     + str(int(
585             n_stiff[0])) + '_stiff_' + str(flag_lin)+'.pdf')  # save the
    figure to file
586         plt.close(fig_tip)
587 
588 
589         """Design variables Vs Span otimization  """
590         fig_DVs_span_opt = plt.figure()
591 
592         span = np.linspace(0,L,nProp)
593         case_plot = cr.get_case(-1)
594 
595         plt.plot(span, (C_wb), 'ro', label='C_wb')
596         plt.plot(span, (h), 'b^', label='h')
597         plt.plot(span, (case_plot['t_sk'][:]), 'g^', label='t_sk')
598         plt.plot(span, (case_plot['t_sp'][:]), 'yo', label='t_sp')
599         plt.plot(span, (case_plot['A_fl'][:]), 'k+', label='A_fl')
600         if not n_stiff.all() == 0:
601             plt.plot(span, (case_plot['A_stiff'][:]), 'co', label='A_stiff'
    )
602             plt.legend(['C_wb', 'h','t_sk', 't_sp', 'A_fl', 'A_stiff'],
    bbox_to_anchor=(1.05, 1), loc='upper left')
603         else:
604          plt.legend([ 'C_wb', 'h','t_sk', 't_sp', 'A_fl'], bbox_to_anchor
    =(1.05, 1), loc='upper left')
605 
606         plt.xlabel('Span', fontsize=18)
607         plt.ylabel('Design Variables', fontsize=16)
608         plt.tight_layout()
609         plt.savefig('../Optimization/Figures/
    ONERA_DesignVAriables_Vs_Span_Optimization' + str(int(
```

```
610             n_stiff[0])) + '_stiff_' + str(flag_lin) + '.pdf')  # save the
    figure to file
611         plt.close(fig_DVs_span_opt)
612 
613 
614         """Design variables Vs Span Initial parameters """
615         fig_DVs_span_init_design = plt.figure()
616         plt.plot(span, (C_wb), 'ro', label='C_wb')
617         plt.plot(span, (h), 'b^', label='h')
618         plt.plot(span, (t_sk), 'g^', label='t_sk')
619         plt.plot(span, (t_sp), 'yo', label='t_sp')
620         plt.plot(span, (A_fl), 'k+', label='A_fl')
621         if not n_stiff.all() == 0:
622             plt.plot(span, (A_stiff), 'co', label='A_stiff')
623             plt.legend(['C_wb', 'h','t_sk', 't_sp', 'A_fl', 'A_stiff'],
    bbox_to_anchor=(1.05, 1), loc='upper left')
624         else:
625             plt.legend(['C_wb', 'h','t_sk', 't_sp', 'A_fl'], bbox_to_anchor
    =(1.05, 1), loc='upper left')
626 
627         plt.xlabel('Span', fontsize=18)
628         plt.ylabel('Design Variables', fontsize=16)
629         plt.tight_layout()
630         plt.savefig('../Optimization/Figures/
    ONERA_DesignVAriables_Vs_Span_initial_design' + str(int(
631             n_stiff[0])) + '_stiff_' + str(flag_lin) + '.pdf')  # save the
    figure to file
632         plt.close(fig_DVs_span_init_design)
633 
634     else:
635 
636         """ Design variables property =1 """
637         fig4 = plt.figure()
638         for i in range(0, len(case_names), 1):
639             case_plot = cr.get_case(i)
640 
641 
642             plt.plot(i, float(case_plot['t_sk']), 'g^', label='t_sk')
643             plt.plot(i, float(case_plot['t_sp']), 'yo', label='t_sp')
644             plt.plot(i, float(case_plot['A_fl']), 'k+', label='A_fl')
645             if not n_stiff.all() == 0:
646              plt.plot(i, float(case_plot['A_stiff']), 'co', label='A_stiff'
    )
647 
648 
649         if not n_stiff.all() == 0:
650             plt.legend([ 't_sk', 't_sp', 'A_fl','A_stiff'], bbox_to_anchor
    =(1.05, 1), loc='upper left')
651         else:
652             plt.legend(['t_sk', 't_sp', 'A_fl'], bbox_to_anchor=(1.05, 1),
    loc='upper left')
653 
654         plt.xlabel('Driver Iterations', fontsize=18)
655         plt.ylabel('Design Variables', fontsize=16)
656         plt.tight_layout()
```

```
657         plt.savefig('../Optimization/Figures/ONERA_DesignVAriables' + str(
    int(n_stiff)) + '_stiff_' + str(flag_lin)+'.pdf')  # save the figure to
    file
658         plt.close(fig4)
```

LISTING B.4. Script for the optimization using Adjoint Method