U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Topology Optimization of High-strain Capable Damage Tolerant Composite Structures

Pedro Nuno Martins Fernandes

Supervisor: Nuno André Curado Mateus Correia

Co-Supervisor: Ricardo Fernando Rodrigues Pinto

Co-Supervisor: Àlex Ferrer Ferré

Doctoral Program in Mechanical Engineering
July 2022

NORTE2020 PROGRAMA OPERACIONAL REGIONAL DO NORTE

inegi driving science & innovation

REPÚBLICA PORTUGUESA | CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

UNIÃO EUROPEIA Fundo Social Europeu

FCT Fundação para a Ciência e a Tecnologia

Faculdade de Engenharia da Universidade do Porto

Topology Optimization of High-strain Capable Damage Tolerant Composite Structures

Pedro Nuno Martins Fernandes

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

Doctor of Philosophy in Mechanical Engineering

President: Pedro Manuel Ponces Rodrigues de Castro Camanho

Referee: Àlex Ferrer Ferre
Referee: Francisco Manuel Andrade Pires
Referee: Paulo Rui Alves Fernandes
Referee: José Manuel Cardoso Xavier

July 2022

To my family.

Abstract

The design of lightweight structures is a priority for the transportation industry, as it represents an improvement in the structural efficiency and an associated reduction of the fuel consumed. For the space industry, a possible path to tackling this issue is the development of self-deployable structures, whose main advantage is the capability of being stored in a compact form and then transitioning to an operational configuration by releasing the elastic energy stored during their contraction. This characteristic leads to efficient use of the space available in the spacecraft, which is further enhanced by the use of composite materials, reducing their total mass.

In most cases, the development of self-deployable structures addresses two main concerns. The first concern is meeting the operational requirements necessary for a specific application, which is usually a stiffness related requirement, such as a minimum pointing accuracy or the natural frequency of vibration. The nature of these requirements contrasts with the need for the self-deployable structure to be flexible enough to be stowed into a compact configuration, which normally limits the maximum stiffness achievable in the design. The second concern is predicting accurately the behaviour of the self-deployable structure once it is in orbit. In between the contraction and the deployment of the structure, there is potentially a long period during which the structure is transported, stored in the spacecraft, launched into space, and finally reaches the intended orbit. This period may allow the occurrence of relaxation in the composite material and, consequently, change the deployment behaviour of the structure. Therefore, the successful design and application of a self-deployable structure depend on finding the optimal balance between two opposing stiffness related requirements and on the accurate prediction of the influence of relaxation phenomenon in the composite material.

Accordingly, this thesis presents a critical review of the state-of-the-art methods used to develop composite self-deployable structures, using an advanced space-engineering problem as a benchmark. The conclusions of this research indicate that the most challenging part of the design process, and also most limiting in terms of potential applications, is the balance of the contradictory stiffness requirements in the design process. To address this issue, two possible solutions are explored: the use of a damage-tolerant design, and the use of topology optimization during the design process.

The possibility of using a damage-tolerant design is benchmarked against the state-of-the-art approach, comparing the first natural frequency of two self-deployable composite elastic hinges. Here, one is allowed to become damaged during the stowing phase, and another, follows the state of the art where no damage is sustained during this phase. This work demonstrates that the damage-tolerant design has a superior performance and potentially a wider range of possible applications.

The use of topology optimization is then discussed, highlighting the similarities between stress constrained compliance minimization problems and the design of a composite self-deployable structure, identifying potential limitations during the process. Within the scope of this thesis and

to promote further research in the field of topology optimization, an open-access Python code has been developed and published. This code works in conjunction with the commercial finite element software ABAQUS® and allows the application of different topology optimization approaches to both 2 and 3-dimensional problems.

Keywords: Composite materials. Damage tolerance. Topology optimization. Deployable structures.

Resumo

O projeto de estruturas com menor peso é uma prioridade para a indústria de transportes, representando uma melhoria na eficiência estrutural e uma consequente redução do consumo de combustível. Para a indústria do espaço, uma possível tentativa de solucionar este problema é o desenvolvimento de estruturas *self-deployable*, cuja principal vantagem é a capacidade de serem armazenadas numa forma compacta e, posteriormente, transitarem para uma configuração operacional expandida através da libertação da energia elástica acumulada durante a sua contração. Esta característica leva a uma utilização otimizada do espaço disponível na aeronave, cujo benefício é melhorado com a utilização de materiais compósitos, tendo como consequência a redução da sua massa total.

Na maioria dos casos, o desenvolvimento de estruturas *self-deployable* é focado em dois pontos. O primeiro ponto é o cumprimento dos requisitos operacionais necessários para a aplicação específica, geralmente associados a requisitos de rigidez, como um valor mínimo de precisão direcional de abertura ou a frequência natural de vibração. A natureza destes requisitos contrasta com a necessidade da estrutura *self-deployable* ser suficientemente flexível para atingir uma configuração compacta, limitando a rigidez máxima atingível durante o projeto. O segundo ponto é a previsão, com exatidão, do comportamento da estrutura *self-deployable* quando em órbita. Entre o momento da compactação e de abertura da estrutura existe um período de tempo, potencialmente longo, durante o qual a estrutura é transportada, armazenada na aeronave, lançada para o espaço e colocada na órbita pretendida. Este período de tempo pode permitir a ocorrência de relaxamento do material compósito e, consequentemente, mudar o comportamento da estrutura durante a sua abertura. Assim, o projeto e aplicação com sucesso de uma estrutura *self-deployable* depende de encontrar um equilíbrio ótimo entre dois requisitos de rigidez opostos e em prever, com precisão, a influência de fenómenos de relaxamento no material compósito.

Como tal, esta tese apresenta uma revisão crítica dos métodos atualmente usados no desenvolvimento de estruturas compósitas *self-deployable*, tendo como ponto de partida um problema avançado de engenharia como referência. As conclusões desta investigação indicam que o maior desafio neste processo de projeto, e também o mais limitador em termos de potenciais aplicações, é encontrar o equilíbrio entre os requisitos de rigidez contraditórios. Na tentativa de apresentar propostas para solucionar este problema, são exploradas duas soluções possíveis: o uso de uma estrutura tolerante ao dano e o uso do método de otimização topológica durante o processo de projeto.

O uso de uma opção estrutural tolerante ao dano é comparado com o método atualmente reportado na literatura, avaliando a primeira frequência natural de vibração de duas articulações elásticas *self-deployable* compósitas. Uma que inicia dano durante a compactação e outra que só funciona em regime elástico. Este trabalho demonstra que o projeto tolerante ao dano tem uma performance superior e, potencialmente, um maior número de possíveis aplicações.

Posteriormente, o uso do método de otimização topológica é discutido. Para tal, são destacadas

as similaridades entre um problema de minimização de *compliance*, restringido por uma tensão máxima, com o problema de projeto de uma estrutura *self-deployable* compósita e identificadas potenciais limitações neste método. Dentro do âmbito desta tese, foi desenvolvido e publicado um código Python, de acesso livre. Este código opera em conjunto com o *software* comercial de método de elementos finitos ABAQUS® e permite a aplicação de vários métodos de otimização topológica a problemas de 2 ou 3 dimensões.

Keywords: Materiais compósitos. Tolerância ao dano. Otimização topológica. Estruturas *deployable*.

Acknowledgements

This thesis was developed between 2018 and 2021 at INEGI - Institute of Science and Innovation in Mechanical and Industrial Engineering, which financed the first year of this research. I also acknowledge the financial support provided by FCT - *Fundação para a Ciência e a Tecnologia*, I.P., between 2019 and 2021, under the scope of the Ph.D. Grant SFRH/BD/145425/2019, co-financed by the European Social Fund NORTE 2020.

Developing this research at INEGI allowed for my contact with several projects who have indirectly contributed to this thesis. In particular, I would like to highlight and acknowledge projects COMETH - Composite Elastic Hinge For Antenna Deployment Structures, financed by the European Space Agency (ESA) through the ESA ARTES program (ESA Contract No. 4000120195/17/NL/EM), and Project NewSat — Development of a compact integrated sensor and satellite for earth observation, nº 45918 financed by *Programa Operacional Regional do Norte*, through *Fundo Europeu de Desenvolvimento Regional* (FEDER).

I would like to express my gratitude to the supervisors of this research, for their support, guidance, and patience to answer my numerous questions. In particular, to Dr Nuno Correia and INEGI, for giving me the opportunity and encouragement to start and pursue this journey. To Dr Ricardo Pinto, for the helpful advice and belief in the quality of the research produced. To Professor Àlex Ferrer, not only for his contribution and support, but also for not hesitating to help, teach, and later on supervise me in this research, when I first reached out to him. Their knowledge and supervision are, certainly, crucial contributing factors to the work here presented.

I would like to thank my colleagues from INEGI and, particularly, to the Unit of Composite Materials and Structures (UMEC). A special thank you to Paulo Gonçalves, Rui Marques, Susana Sousa, Fermin Otero, Sónia Ribeiro, Ricardo Rocha, Bruno Sousa, Jhonny Rodrigues, and João Machado, for their support, friendship, and expertise that has greatly aided this research. This gratitude is also extended to Filipe Ferreira, who worked closely with me during his master thesis and made relevant contributions to this journey, to Professor Lucas da Silva and the Advanced Joining Processes Unit (UPAL), who contributed to the experimental components of the research developed, and to Gonçalo Rodrigues from European Space Research and Technology Centre (ESTEC), whose insight greatly contributed to and also motivated this research.

A final, and warm, word of gratitude goes to my friends and family. To Ana, Inês, Fernando, Marco, Mariana, and Carlos for their friendship and all the good moments shared. To Ana Dulce, for always being there with your bright-fullness and going above and beyond to try to help me with the many challenges faced during this period. To my family, especially to my parents, who not only have supported me for so many years but were also always curious and interested in reading my work. It has been a pleasure to share this journey with all of you!

Pedro Fernandes

Publications

Appended in this Thesis

Publications in refereed conference proceedings

P. Fernandes, R. Marques, R. Pinto, P. Mimoso, J. Rodrigues, A. Silva, João Manuel R.S. Tavares, G.Rodrigues, N.Correia. *Design and optimization of a self-deployable composite structure*. In MAT-COMP'19 - Composites for Industry 4.0, Vigo, Spain, 2020. ISSN: 2531-0739.

Publications in refereed journals

P. Fernandes, B. Sousa, R. Marques, João Manuel R.S. Tavares, A.T. Marques, R.M. Natal Jorge, R. Pinto, N. Correia. (2021) *Influence of relaxation on the deployment behaviour of a CFRP composite elastic-hinge*. Composite Structures, 259, 113217. doi.org/10.1016/j.compstruct.2020.113217.

P. Fernandes, R. Pinto, N. Correia. (2021) *Design and optimization of self-deployable damage tolerant composite structures: a review*. Composites Part B: Engineering, 221, 109029. doi.org/10.1016/j.compositesb.2021.109029.

P. Fernandes, R. Pinto, A. Ferre, N. Correia. (2022) *Performance analysis of a damage tolerant composite self-deployable elastic-hinge*. Composite Structures, 288, 115407. doi.org/10.1016/j.compstruct.2022.115407.

P. Fernandes, A. Ferre, P. Gonçalves, M. Parente, R. Pinto, N. Correia. *Stress constrained topology optimization for commercial software: a Python implementation for ABAQUS®*. (Submitted to the Journal of Advances in Engineering Software, Elsevier).

Not Appended in this Thesis

Publications in refereed journals

F. Ferreira, P. Fernandes, N. Correia, A.T. Marques. (2021) *Development of a Pultrusion Die for the Production of Thermoplastic Composite Filaments to Be Used in Additive Manufacture*. Journal of Composites Science 5(5), 120. doi.org/10.3390/jcs5050120.

Presentations in conferences

P. Fernandes. *On the design and development of composite deployable elastic-hinges*. In SpaceCarbon Workshop & Conference, Porto, Portugal, 2022. https://spacecarbon-project.eu/wp-content/uploads/2022/06/P.-Fernandes-On-the-Design-and-Development-of-Composite-Deployable-Elastic-Hinges.pdf.

Contents

List of Figures

List of Tables

List of Abbreviations

Acronyms

Acronym	Meaning
2DQ4	2D element with 4 nodes
3D	3-dimensional
ACO	Ant Colony Optimization
BB	Barzilai-Borwein method
BESO	Bi-Directional Evolutionary Structural Optimization
BPNN	Back-Propagation Neural Network
C3D8	3D cube element with 8 nodes
CA	Cellular Automata
CAI	Crash After Impact
CDM	Continuum Damage Mechanics
CFRP	Carbon Fibre Reinforced Polymer
CPE4	Plane-strain element with 4 nodes
CPS4	Plane-stress element with 4 nodes
CPU	Computer Processing Unit
CTO	Concurrent Topology Optimization
CZM	Cohesive Zone Modelling
DIC	Digital Image Correlation
DIW	Direct Ink Writing
DMO	Discrete Material Optimization
EBA	Enhanced Bat Algorithm
EGO	Efficient Global Optimization
ESA	European Space Agency
ESO	Evolutionary Structural Optimization
FEA	Finite Element Analysis
FEUP	Faculty of Engineering of the University of Porto
FVF	Fibre Volume Fraction
GA	Genetic Algorithm
GOLS	Gravity Offloading Systems
IHPA	Improved Hybrid Perturbation Analysis
LDPE	Low-density polyethylene
LSF	Level-Set Function
LSM	Level-Set Topology Optimization Method
MARSIS	Mars Advanced Radar for Surface and Ionosphere Sounding
MIGA	Multi-Island Genetic Algorithm
MMA	Method of Moving Asymptotes

MWCNT	Multi-Walled Carbon Nanotubes
NCGC	Non-Uniform Curved Grid-Stiffened Composite Structure
NDFO	Normal Distribution Fibre Optimization
NDI	Non-Destructive Inspection
NDSGA	Non-Dominated Sorting Genetic Algorithm
NGM	Normalized Gradient by Maximum
NSGA-II	Non-Dominated Sorting Genetic Algorithm II
NURBS	Non-Uniform Rational B-Spline
OC	Optimality Criteria
PCA	Principal Component Analysis
PDS	Ply Drop Sequence
PEP8	Style Guide for Python Code
PHC	Polynomial Homotopy Continuation
PPS	Permutation for Panel Sequence
PSO	Particle Swarm Optimization
RBFNN	Radial Basis Function Neural Networks
RBRDO	Reliability-based Robust Design Optimization
RCC	Representative Cell Configuration
RIA	Reliability Index Approach
RVE	Representative Volume Element
S4	Shell element with 4 nodes
SA	Simulated Annealing
SFV	Streamline Function Value
SHM	Structural Health Monitoring
SIMP	Solid Isotropic Material Penalization
SLSQP	Sequantial Least Squares Programming
SMM	Shape Memory Materials
SSPO	Streamline Stiffener Path Optimization
TTS	Time-Temperature Superposition Principle
WLF	Williams-Landel-Ferry equation

Symbols

Symbol	Meaning
β	Stress penalization factor.
ε	Strain
η	Numerical damping coefficient
γ	Generic vector
ψ	Lagrange multiplier
ρ	Filtered Design Density
σ	Stress
σ_a	Amplified stress
σ_a^{VM}	Amplified von Mises stress
A	Ply angle
B_a	Strain-displacement matrix

B_e	Optimality condition parameter
$\mathbb{C}$	Element stiffness
E	Young's modulus
E^e	Elastic strain energy
E^p	Plastic strain energy
E_{rr}	Pondered error
e_{vol}	Material constraint evolution ratio
F	Force
G	Shear modulus
g	Gradient
K	Stiffness
l	External load forces
L_f	Slot length scalling factor
$L^\infty(\omega)$	Space of bounded functions
$H_0^1(\omega)$	Space of functions with square integrable derivatives and homogeneous values on the boundary
m	Mass
M	von Mises matrix
N	Number of evaluation points
N_f	Natural frequency
P	SIMP penalty factor
P_f	Penalty factor
P_n	Number of plies of the composite
$Perf$	Performance value
Q_F	Final P-norm approximation factor
Q_i	Initial P-norm approximation factor
R_{1r} and R_{2r}	Ratio between the slot's width and R_1 or R_2 on the centre of each circle
r_{max}	Maximum filter search radius
R_1 and R_2	Radius 1 and 2
R_i	Internal radius of the tube
S	Shear strength
S_D	Slot's displacement from the longitudinal axis of the tube.
S_L	Slot's length
T	Temperature
u	Displacement
U_p	Potential energy
v	Poisson's coefficient
V	Volume
x	Design density variable
X, Y, and Z	Longitudinal and transverse strengths
$X_{1,\ldots,6}$	Design variables 1 through 6
$\bigtriangledown^s$	Symmetric gradient

Chapter 1

Introduction

1.1 Motivation and Background

The need to efficiently transport payloads, store large equipment or cargo in a limited space, and reduce the fuel necessary for the transportation are examples of requirements that can be found in a wide variety of fields and industries, ranging from the automotive industry to military applications. In space applications, particularly in spacecraft design, these dimensional and mass constraints are set by the capability of the launcher. The difficulties associated to the development of launchers with increased capacity has created a theoretical limit on the size and mass of the payloads. This constraint limits the quality, the further development, and the launch of better performing equipment, such as telescopes [1, 2]. To deal with this constraint, several researches have studied the possibility of developing structures that change their configuration, allowing their storage in a compact form and a transition into a large operating configuration when required. These structures are referred to as "deployable structures" and can take on, at least, two possible configurations: one "retracted" and another "deployed" [1–4]. One application of this concept is the elastic hinges in the monopole and dipole antennas of MARSIS (Mars Advanced Radar for Surface and Ionosphere Sounding), which allowed the folding of antennas with lengths of up to 20 m into segments with approximately 1.5 m in length [5].

Recently, the European Space Agency (ESA) identified telecommunication satellites as a possible application for this technology, replacing mechanical arms with a lighter and more compact solution. This trend is evident in projects proposed by ESA, such as the call for the development of an antenna deployment arm with integrated elastic hinges, in 2016 [6]. Typically, the development of advanced technology presents conflicting requirements. In the case of deployable systems, the structure should be flexible to sustain high strain deformations and rigid enough to support external loads and/or reach certain natural frequencies [2–5, 7–9]. Furthermore, the statement of work presented by ESA [6] requests that the deployment arm developed for telecommunication purposes should have a natural frequency higher than 1.0 Hz [6], which is 20 times larger than what was considered in the monopole and dipole antennas developed for MARSIS [6] and 100 times larger than similar solutions reported in the literature [10]. This scenario presents a challenge and the need for a significant scientific development if one is to have such an improvement over the current state-of-the-art solutions. Excluding the possibility of redefining the requirements of the structure, failing to find a balance between them can lead to one of two situations: either the structure becomes too flexible, failing to meet the rigidity requirements necessary to operate, or the structure becomes too rigid, making it impossible to retract without initiating damage.

Further research on the design and optimization of composite deployable structures is a possible path to solve the challenges that arise from the use of these structures and enable their application in telecommunication satellites. Additionally, the development of new design methodologies suitable for composite materials has a significant potential for technological transfer, especially to industries that benefit from structural weight reduction.

1.2 Problem Statement

Due to its novelty, information regarding the design and optimization of deployable structures considering multiple requirements is scarce. Similarly, limited research and study have been performed on the evaluation of possible solutions that may lead to increased compatibility between the requirements that characterize deployable structures.

It is the scope of this research to review the technical challenges associated with the design of composite deployable structures, identify limiting factors in the design approaches currently used, as well as to propose and evaluate new methodologies that may lead to relevant performance improvements of the designs obtained.

Throughout this thesis, the design requirements presented by ESA in [6] will be used as a reference and a starting point. It is important to note that, despite having been first published in 2016, these requirements are a representative example of an up-to-date engineering problem whose project and development are still on-going. Furthermore, the requirements that define this problem can be generalized to other engineering case studies involving the design of high-performing composite structures.

1.3 Scientific Objectives

With this research, the following scientific and technical aims are foreseen:

- The identification and determination of material parameters for numerical models that predict damage initiation and propagation during operation of a deployable structure.
- The analysis of the design methodologies currently used in the design of composite deployable structures.
- The development and evaluation of a design method suitable for the optimization of the topology of deployable composite structures.
- The inclusion and analysis of damage constraints in the design of deployable composite structures.

1.4 Thesis structure

This thesis is constituted by a total of eight chapters, the bulk of the work is based on peer-reviewed published articles. The attentive reader will notice that the sequence of chapters presented in this thesis does not represent the chronological order in which this research was developed.[1] This

[1] Throughout this thesis, it is possible to find footnotes with brief orientations that will guide the reader interested in following the chronological order of research, which has a more linear increase in scientific complexity. Nonetheless, summary overview of the chronological developments reported in this thesis would correspond to the following order of chapters: section 2.2 of Chapter 2, Chapters 3 and 4, sections 2.3 through 2.4 of Chapter 2, and finally Chapters 5 through 7.

change was intentional and meant for the sake of having a classic thesis structure, beginning with a single state of the art review that provides a global perspective of the research developed. Therefore, the thesis is organized in the following order.

Chapter 2 presents a state-of-the-art review of the topics addressed in this research and is divided into two parts. The first part, section 2.2, reviews the methodologies used for the experimental characterization, numerical modelling, design, and optimization of a self-deployable structure. The information reviewed in section 2.2 is applied in Chapters 3 and 4. The second part, from section 2.3 to section 2.4, presents a review of the methodologies applied to the design of composite structures and on the use of damage tolerance concepts. This second part of the literature review is applied from Chapter 5 onwards.

Chapters 3 and 4 apply the methods identified in section 2.2 to the challenges associated with the design of a composite deployable structure. More specifically, Chapter 3 focuses on the design and optimization of a self-deployable elastic-hinge, addressing the difficulties that arise from balancing the flexibility and first natural frequency requirements listed by ESA [6], while Chapter 4 addresses the influence of relaxation on the deployment behaviour of a similar structure.

Since it was possible to accurately predict the influence of relaxation on the deployment behaviour of an elastic-hinge but not to meet both design requirements defined, Chapter 5 evaluates the possibility of using a multidisciplinary approach to improve the performance of self-deployable composite structures. This chapter proposes and discusses the use of a damage-tolerant design as a means to meet the first natural frequency requirements defined by ESA [6].

An alternative route to possibly meeting the design requirements set by ESA [6] is the application of the method of topology optimization to the design of deployable composite structures. However, the use of topology optimization to this particular design problem is scarcely reported in the literature. The novelty of the method, the still on-going research on the application to composite materials and on the implementation of stress and/or damage constraints, as well as the reduced number of open-source implementations available to the public are some of the factors that contribute to the limited use of this method. Therefore, Chapter 6 presents a Python script that can be used in the commercial software ABAQUS® [11], which includes a possible implementation of the state-of-the-art methods for stress-constrained topology optimization suitable for 2 and 3-dimensional problems.

Chapter 2

State of the art review on the design, experimental characterization, numerical modelling, and optimization of self-deployable elastic hinges

The present chapter is based on the following refereed publication:

P. Fernandes, R. Pinto, N. Correia. (2021) *Design and optimization of self-deployable damage tolerant composite structures: a review*. Composites Part B: Engineering, 221, 109029. doi.org/10.1016/j.compositesb.2021.109029

2.1 Introduction

For the space industry, the design of light and compact structures capable of deploying, once the payload is in orbit, has been of significant interest for more than two decades [1, 2, 12, 13]. This interest is based on the consequent increase in efficiency of the design of spacecrafts, as they become capable of transporting more equipment while reducing costs associated with fuel consumption [1, 2]. Initial developments that used metals later used also more efficient and lighter composite systems [3, 14–16].

Stored elastic energy deployable structures have been the leading candidates for space applications. These structures are designed to be stored in a retracted form with the capability of self-deploying by releasing the elastic strain energy accumulated during the retraction/folding process, which occurs in the elastic regime [1]. The simplest implementation of this concept is usually referred to as "tape-springs" (figure 2.1), of which an example is the steel tape measure (also known as carpenter tape). The first use of tape-springs as deployable structures dates to 1968 [17], which are a useful replacement to traditional hinge mechanisms since they provide high repeatability and pointing accuracy. Well-known applications include the monopole and dipole antennas of MARSIS in 2003 (figure 2.2) [5].

The use of composite deployable structures in telecommunication satellites has been proposed by ESA in 2016 [6]. Doing so represents the possibility of replacing mechanical arms with a lighter and compact solution but also a significant engineering challenge. Designing a deployable structure for this application requires a delicate balance between structural flexibility, to fold and contract the structure into a compact configuration, and structural stiffness, to reach a required minimum first natural frequency. This challenge is aggravated by the intention of meeting requirements significantly more ambitious than the solutions currently reported in the literature [10, 5].

In other fields of application of composite materials, the design methodologies have evolved significantly. Concepts such as damage tolerance have been investigated, leading to an extensive characterization of composite materials under a considerable plethora of loading cases and operating conditions. Aeronautic and energy industries, amongst others, have adopted this concept as a means of increasing the efficiency and/or factor of safety in the designed structures. Also, the cost reduction is associated with the maintenance or disposal of damaged parts [19–22]. Similarly, topology optimization methods have been under the spotlight of many researchers [23–25], developing more refined methodologies that can be applied to more advanced materials.

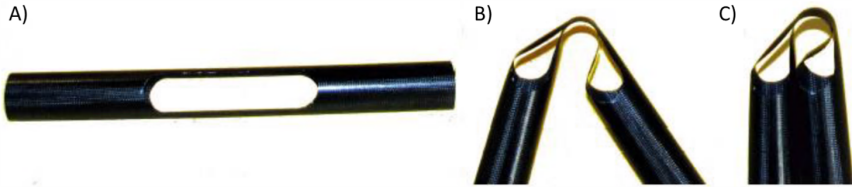

Figure 2.1: Tape-spring hinge in a deployed (A), partially contracted (B) and fully contracted (C) configurations (figure adapted from [18]).

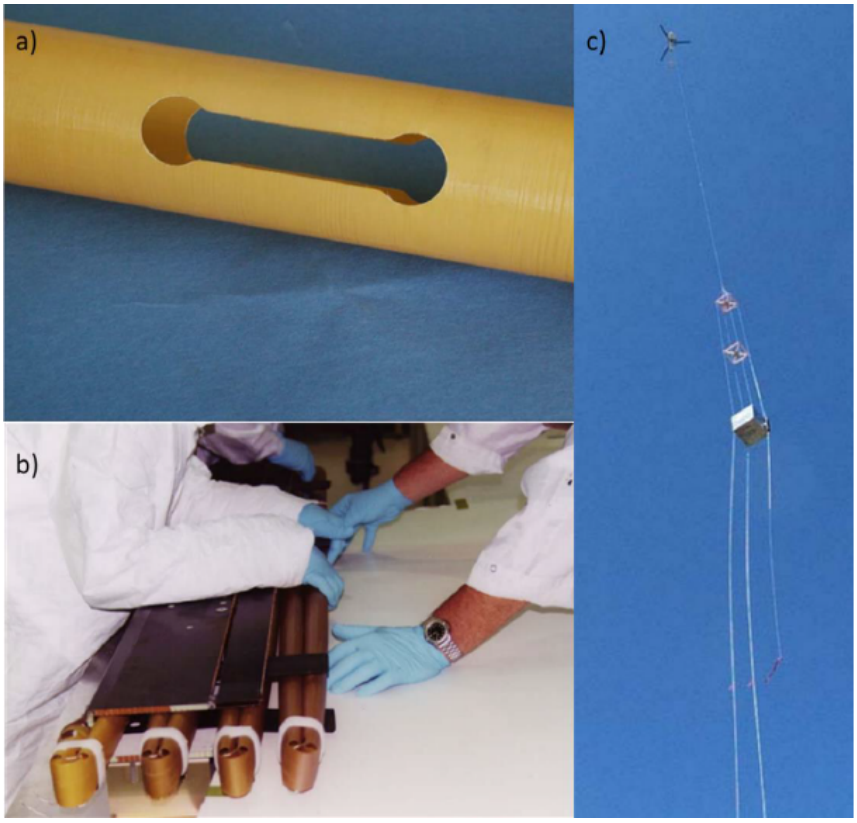

Figure 2.2: Elastic hinge used in the monopole and dipole antennas of MARSIS. a) detail view of the elastic hinge; b) stowage process; c) experimental deployment of the prototype using a helicopter (adapted from [5]).

Despite the novelty or interest of damage tolerance, topology optimization, and deployable structures, the literature available on the combination of any two of these concepts and their application to deployable structures is either scarce or non-existent. Furthermore, the review of the combination of the three domains is unexplored.

Bringing together the benefits of damage tolerance and topology optimization can significantly impact the design and application of deployable structures. This type of structural mechanism has unique requirements and operating conditions. The need to meet flexibility and stiffness requirements is a challenging and daunting task that, due to its contradictory nature, may not be accomplished and will limit the use of deployable structures. However, since during the operating life of most deployable structures only one deployment is expected, it may be acceptable to design a structure that initiates damage, in a controlled manner, during its retraction process. As a result, the need for a flexible design can be relaxed, making it easier to reach stiffness-related requirements. Therefore, the benefits of using topology optimization to search for optimal solutions could be enhanced with an approach that, not only leads to a more efficient design but also increases the design space and the number of possible solutions suitable for the respective application.

This literature review explores the possibility of using topology optimization with a damage tolerant design approach suitable for composite materials. To do so, the review is divided into

three main sections. Section 2.2 reviews the methodologies applied to the design of self-deployable structures, section 2.3 addresses the design methods of composite structures, and section 2.3.2 presents an overview of the different topology optimization methods. Section 2.3.3 reviews the use of damage tolerance. Finally, the possibility of combining these concepts into a methodology that leads to a topology optimized self-deployable damage tolerant composite structure is discussed in Section 2.4.

2.2 Composite deployable structures

The compact configuration characteristic of deployable structures is the result of the retraction process, which imposes the need for the structure to be flexible enough to sustain high-strain deformations [1, 2].

Similarly to the design process used for composite structures in the aeronautic sector [26], the following sub-sections overview the tools used to design deployable structures, including experimental characterization methods, numerical simulation approaches and design methodologies. For a more detailed review of different folding mechanism concepts and their trade-off, the interested reader is referred to [1] and to [27] for a more detailed classification.

2.2.1 Experimental characterization of deployable tape-springs

Despite the first research dating back to 1973 [28], reporting that the deployment of a tape-spring could cause a snap-through behaviour due to buckling loads, the literature available on methodologies to characterize tape-springs is quite limited. It was only in 1999 that Seffen and Pellegrino [14] proposed the first of five characterization methodologies used in the study and development of tape-springs.

Research reported in [14] has shown that the experimental characterization of this component can be performed by measuring the torque-angle relationship (figure 2.3). Due to the curvature of the tape-spring, the behaviour is highly dependent on the direction in which the tape-spring is bent. When the torque applied causes tensile stresses on the edges of the tape-spring (considered positive torque, by convention), the tape-spring shows a linear behaviour followed by a sudden bend, which flattens the tape and causes a return to the linear behaviour. When a negative torque is applied, compressing the edges of the tape-spring, the linear behaviour ends sooner. Seffen and Pellegrino [14] reported that this case of loading promotes the deployment to occur along the same path, increasing the repeatability of the process. Later research [18, 8, 10, 29–35], more focused on composite materials, further supports the results obtained on the behaviour of tape-springs and on the use of torque-angle curves to characterize the retraction of tape-springs, leading to their application in several studies.

While the methodology described above is suitable to assess the retraction of the tape-spring, it does not address the dynamics involved in deployment. To study the release of the tape-spring, several authors [14, 18, 10, 30, 37] have used high-speed cameras to measure the angle formed by the tape-spring, or by the deployable structure, as a function of time. The main advantage reported is

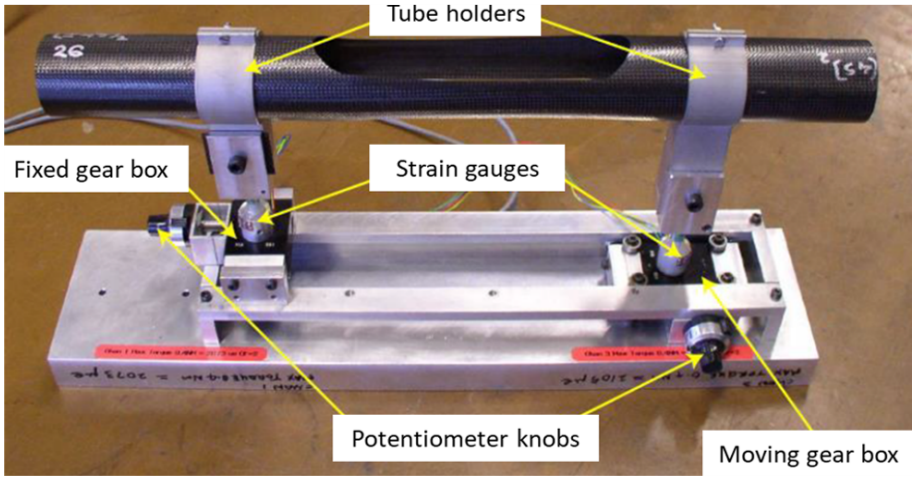

Figure 2.3: Example of an experimental setup used to measure the torque-angle curve (adapted from [36]).

the possibility of measuring any over-shooting angle that may occur. In this context, over-shooting angle refers to the angle formed by the tape-spring due to buckling caused by compressive loads installed during the deployment process.

Due to the functioning and application of deployable structures, most of the developed designs only deploy once throughout a mission. Nevertheless, research developed by Warren *et al.* (2005) [33] analysed the stowage fatigue cycling of a deployable system. The experimental process simply consisted of the automatic stowage of the structure every 20 seconds. In the case studied, no damage was observed after 10,000 folding and deployment cycles.

A relevant factor to be considered when designing a deployable structure is creep behaviour. The stowage of the deployable structure for long periods may cause anomalies in the deployment, as seen in the case of MARSIS [38]. The study of this phenomenon is usually performed on tape-springs, maintaining the retracted configuration for determined time periods [38–43], although temperature has been used to accelerate the ageing process [44, 45]. Following this methodology, Kwok and Pellegrino [43] have been able to detect energy reductions of up to 60 % when investigating the storage of composite tape-springs.

In some cases, Gravity Offloading Systems (GOLS) have been used to offset the effect of gravity. The purpose of this methodology is to replicate the operating condition of the tape-spring during deployment in space. In 2017, Mao *et al.* [46] used a GOLS that consisted of a set of braided cords that suspend the deployable structure. This allowed the structure to deploy along a plane parallel to the ground, resorting to a set of soft extension springs that compensated for the changes in distance between the hanging points.

In summary, the characterization of the mechanical properties and behaviour of tape-springs is based on the experimental measurement of the torque involved in the retraction process and on the observation of the deployment. However, for being applied in space, the main obstacle is recreating the operating conditions. Offsetting variables such as the influence of gravity or air friction is a difficult task that may hinder an extensive testing of a design, particularly for solutions with low

deployed stiffness or low deployment torque.

2.2.2 Numerical modelling of composite deployable structures

The design and development of deployable structures require the validation of the respective in-service conditions. However, replicating the operating conditions can be difficult, if not impossible [47]. This is evident in scenarios such as the one of MARSIS, where the deployable system used had a low deployment torque. This caused the deployment to be compromised by factors such as air friction and the influence of gravity [5]. Implementing numerical models that can accurately predict the behaviour and performance of the deployable system, taking into account the majority of factors that influence the process, would greatly aid the design process and allow a reduction in the experimental campaign necessary to validate each concept. Several authors [14, 18, 10, 36, 48–50, 30, 40, 43, 46, 47, 51–62] have published numerical modelling of deployable structures, where this topic is explored at different scales, from the micro to the macro-scale.

In 2011, Mallikarachchi *et al.* [10, 51, 36] explored the possibility of using a micro-to-macro scale approach to model an elastic hinge during both deployment and retraction processes. Here, a failure criterion developed specifically to analyse plain-weave carbon fibre reinforced polymer (CFRP) under in-plane, bending, and the combined effect of in-plane and bending loads was considered. At the micro-mechanical scale, the authors estimated the elastic properties of the tow using the rule of mixtures and semi-empirical relations, detailed in [63], to determine the Poisson's ratios, longitudinal and transverse extensional modulus, and the shear modulus. Then, at a meso-scale level, the authors recreated the tow's cross-section architecture, the waviness of the fabric used and the ply arrangement considering geometric data obtained from the material microscopic analysis. The tows were modelled as wavy beams, defined by a sine wave, according to [64]. For the ply arrangement, Mallikarachchi included 6-node triangular prisms in the gaps formed between tow surfaces, representing additional neat resin. This approach would later lead the author to adjust the fibre volume fraction (FVF) within the tow to achieve a value equal to the one measured experimentally. Before transitioning to the macro-scale, the representative volume element (RVE) of the fabric was tested virtually and used to homogenize the ply properties.

The macro-scale model was implemented in ABAQUS® [11] using an explicit formulation that considered the interaction of the elastic hinge with solid elements that replicate the experimental testing process used to obtain the torque-angle curve. These solid elements represent the two holders that support the elastic hinge on each end and are used to apply: 1) a torque that forces the system to fold; and 2) a pair of solid plates, responsible for pinching the tape-springs and flattening them before initiating the retraction process [36].

The sequence of steps of the retraction process recreated by the author is described in figure 2.4. However, during the experimental validation of the numerical model, the author would suppress the pinching step, reporting that this was a manual step and, therefore, its removal would increase the repeatability and better standardize the experimental testing process. In other scenarios [10, 51], the pinching step was included as a means of guaranteeing that the folding of the tape-springs would occur at their mid-longitudinal section.

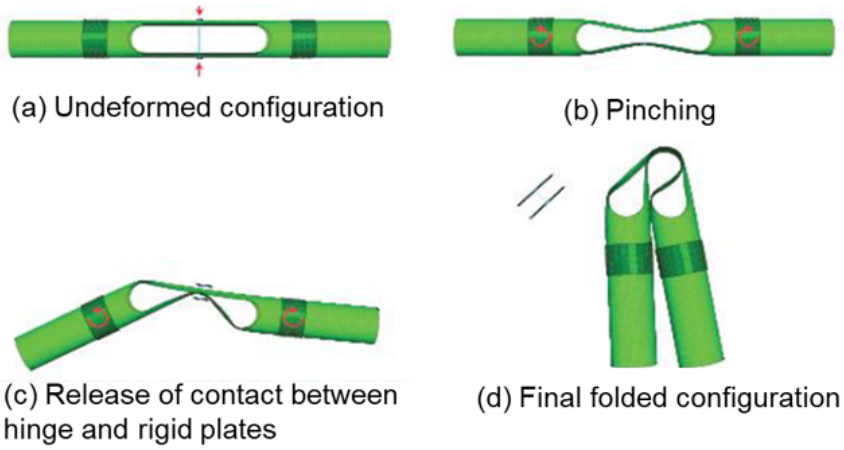

Figure 2.4: Stages of the retraction (or folding) process simulation (adapted from [51]).

Regarding the experimental validation of the model, the author reported the presence of instabilities reflected as an oscillating movement of the elastic hinge during the simulated retraction process. This divergence between numerical and experimental results was overcome using a viscous pressure parameter that functions as a damping mechanism. The value of the viscous pressure was tuned iteratively, ensuring that the energy dissipated through viscous mechanisms remained below 1 % of the energy balance of the system [10, 51].

The research of Mallikarachchi *et al.* [10, 51] includes a numerical simulation of the deployment behaviour. The numerical model is equal to the one used to simulate the retraction of the elastic hinge but includes a step that releases the boundary conditions applied on one end of the structure.

The comparison between the experimental and numerical results indicates that the finite element simulations could capture the behaviour of the elastic hinge. However, it revealed some significant differences in the estimation of the maximum torque applied during the deployment stage. Furthermore, the maximum torque observed during the retraction process was, approximately, two times larger than the maximum torque installed during deployment. The authors justify this difference as a characteristic of structures with an unstable post-buckling equilibrium path [51], as described by Brush *et al.* [65] and by Van der Heijden *et al.* [66].

Finally, Mallikarachchi *et al.* [52] defined a failure criterion suitable to analyse plain-weave CFRP in three different loading cases: failure due to in-plane, bending, and in-plane plus bending loads. According to this criterion, the absence of the pinching step would cause the initiation of damage during the retraction process due to the installed torque [10, 51].

The research developed by Mallikarachchi *et al.* [18, 10, 36, 48–53] is quite significant in the field of deployable structures for its extensive scope and approaches. However, several other researchers made relevant contributions. Research with similar approaches include: Seffen and Pellegrino (1999) [14] for the full deployment simulation and initial design of a single tape-spring, Boesch *et al.* (2007), Givois *et al.* (2001) and Jeong *et al.* (2016) [30, 54, 55] for structures with multiple tape-springs, Mao *et al.* (2017) [46] for integral slotted hinges, and Cook and Walker (2016) [56] for the use of tape-springs in the deployment of an inflatable structure.

In 2016, Dewalque *et al.* [57] performed a quasi-static analysis to assess the influence of geometric and material parameters of a single beryllium copper tape-spring to assess the relationship between the bending moment and the rotation angle. The parametric analysis and the optimization process used in this research resulted in a tape-spring geometry that minimized the installed stresses, according to the Von-Mises failure criterion, while maximizing the range of motion.

Nonlinear dynamic analysis was also used to evaluate buckling, hysteresis and self-locking phenomena that characterize the deployment of tape-springs. Walker and Aglietti (2004) [58] addressed the retraction and deployment of a tape-spring but considered a complex three-dimensional array fold. Hoffait *et al.* (2010) [59] addressed the hysteresis and buckling phenomenon observed during the deployment using a geometry similar to the one patented in [17], while Dewalque *et al.* (2015) [60] focused on using both numerical and structural damping to accurately simulate the deployment behaviour of tape-springs.

The viscoelastic effect that tape-springs may suffer due to the long-term stowage has also been addressed in the literature. Kwok and Pellegrino (2011, 2013) [40, 43] successfully captured the viscoelastic effect of the stowage implementing finite-element simulations based on linear viscoelastic material models. In 2012, the same authors [61] presented a micro-mechanical finite element model and used it to study the deployment and recovery of a bent thin-walled viscoelastic tape-spring. The outcome was a close agreement between numerical and experimental results. Nevertheless, the model was difficult to implement in the deployment simulation of a complete structure due to its expensive computational cost. In 2013, Peterson and Murphey [62] have shown through experimental test results, modified micromechanics and classical laminate theory analysis that the longitudinal and transversal bending stiffness of the tape-springs decreases with the shear modulus over time. The results obtained were 10 % lower than expected, which were justified by inaccuracies in the thickness of the material.

The literature on the numerical modelling of composite deployable structures focuses two different topics: the stowage and/or deployment sequence and the structural behaviour. The analysis of the stowage and/or deployment sequence is mainly concerned with the accurate representation and prediction of the kinematic movement, especially those that result from the release of the strain energy stored during the retraction process. On the other hand, the structural analysis addresses the integrity of structure from two different points of view. The first, considering the internal loads resulting from the deployment sequence, which can often rely on multi-scale approaches to bring together the properties of the composite material observed at a micro-scale and the loads that result from the movement of the structure at a macro-scale. The second, considers the changes in properties of the material, especially due to relaxation phenomenon that result from extended stowage periods.

2.2.3 Design of composite deployable structures

The successful design of a deployable structure relies on the capability of achieving the necessary flexibility to sustain the high-strain deformations that allow their characteristic high-compact ratio [1].

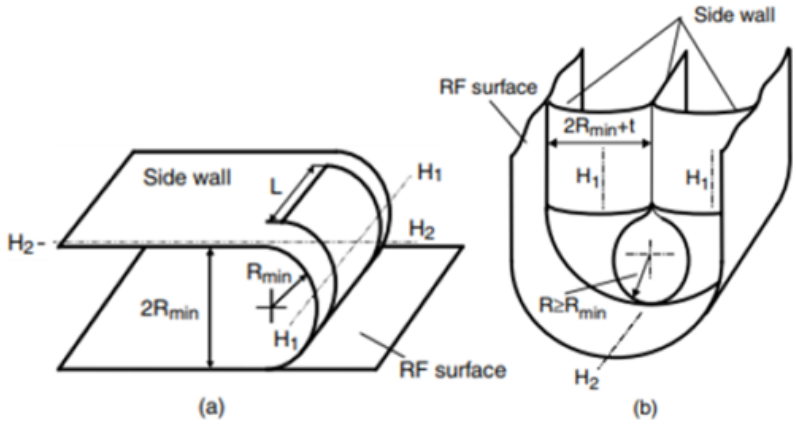

Figure 2.5: Representation of the two-step retraction sequence. (a) Folding of side wall about line H1H1; (b) folding of RF surface around the folded side wall (adapted from [67]).

Soykasap *et al.* (2004) [67] reported the design process of a deployable reflector concept. The authors used the maximum strain failure criterion to theoretically estimate the minimum bend radius that the CFRP sheets could sustain without initiating damage. The estimation was experimentally validated but the authors mentioned that the use of angle plies on the outer surfaces of the plate significantly decreased both the predicted and the minimum measured bend radius. The size of the cut-outs was defined considering that, although they reduce the maximum strain in the region close to the hinge, their dimension should be as small as possible to avoid the loss of stiffness in the structure. Furthermore, the authors observed that reinforcing the stress concentration points near the edges of the cut-out would increase the maximum stress due to the increased thickness of the composite. Parameters such as the length and width of the cut-out were defined taking into consideration the minimum bend radius that would allow a two-step retraction sequence (figure 2.5). In 2008, the authors also applied this approach to a similar structure, imposing a minimum natural frequency requirement [68].

In 2010, Mallikarachchi and Pellegrino [48] addressed the capability of an elastic hinge to bend at 180° through a parametric study using a finite element model. Each combination of parameters was evaluated through a structural finite element analysis (FEA), assessing if there was initiation of damage according to the maximum strain failure criteria. The authors then selected three possible designs that were further analysed in [49]. This time, the elastic hinges were simulated considering their stowage configuration around a satellite and were evaluated according to their failure criteria proposed in [52], suitable for the evaluation of plain-weave CFRP.

Tan and Pellegrino (2006, 2012) [7, 69] reported the development of a cap that improved the deployed stiffness of a deployable reflector. The stiffener was applied on the edges of the tape-spring that deploys the structure. The authors applied the Hooke and Jeeves direct search method [70] to identify the optimal angle and width of the stiffener, as well as the angles of slit at end of the diameter. Each solution was evaluated through a FEA in ABAQUS® [11] to determine the influence on the natural frequency of the deployed structure. This methodology led to an improvement of the stiffness and natural frequency by a factor of 31 and 4, respectively, with a

mass increase of only 16 % when compared to the same solution without the stiffener.

The use of shape memory materials (SMM) in deployable systems deserves to be mentioned. To prevent the latch-up shock from the deployment of high-stiffness tape-springs, Jeong *et al.* (2014) [55] included a shape memory alloy tape aligned with the tape-spring. The reason behind this design is the transition from a dynamic, and possibly unstable, deployment to a quasi-static one. A sequential quadratic programming algorithm was used to solve the optimization problem, iterating over the length, thickness (number of plies) and width of the tape-springs. The dimensions of the shape memory alloy used as a damping mechanism were selected considering the torque applied by the tape-spring. The use of SMM has been a point of interest for several studies [71–76], which are not within the scope of the present review. For more information and an updated review on the use of SMM in deployable systems, the interested reader is referred to [71, 72].

Chu and Lei (2014) [77] presented a design theory and dynamic analysis of a lenticular boom with a mechanism that aids the deployment and retraction process. The design process began with the definition of a relationship between the geometry of the lenticular boom and properties or factors, such as bending stresses, torsional stresses, the natural frequency of the structure and the strain energy involved. The analytical expressions defined were then used in an optimization cycle, using the sequential quadratic programming method. The objective of this optimization was to minimize the stress installed in the structure considering geometric and natural frequency constraints.

To design the tape-springs of a deployable solar panel, Dewalque *et al.* (2016) [57] used a parametric study and an optimization algorithm to explore possible combinations of thickness, radius of curvature and subtended angle of the tape-spring. In this case, and due to the nature of the selected algorithm (*fmincon* in MATLAB [78]), the parametric study served as a means of performing an educated initial guess of the optimal solution. The optimization algorithm was used in conjunction with the software SAMCE [79], which simulated the performance of the geometry selected by the algorithm when retracting 120° and posterior deployment.

Wu and Vinquerat (2017) [9] designed a braided bi-stable carbon-epoxy tube considering the optimization of its natural frequency. To do so, the braid angles and stacking sequences were optimized, using the coiled diameter of the structure as a constraint and the maximum and minimum physically achievable braid angles as bounds. The authors utilized a pattern search method, available in MATLAB global optimization toolbox [78], to define the parameters to be used by a Python [80] script to generate the models to be simulated in ABAQUS® [11]. The implemented design process allowed the authors to find a design that met the defined requirements. During this process, it was also observed that the braid angle had a greater influence on the natural frequency than the ply location within the stacking sequence for long slender designs, whereas both factors have a significant influence on the natural frequency of shorter designs.

Between 2015 and 2019, Sakovsky *et al.* [35, 81–83] have researched the use of a dual-matrix concept to design deployable antennas. According to the authors, thin shell deployable structures offer efficient packaging but reduced surface precision, while elastomer composite shells have a smaller fold radius upon packaging but are limited by the stiffness of the deployed

structure. Therefore, the concept of a dual-matrix composite, consisting of a continuous CFRP with localized elastomer matrix embedded in hinge regions, should allow for small fold radii, strain energy deployment, high deployed stiffness, and enable larger antenna apertures. The authors have compared dual-matrix structures with existing antenna designs considering both structural and electromagnetic performances. This approach led to the design of a deployable dual-matrix composite conical log spiral antenna to be used on CubeSats, which outperformed existing off-the-shelf designs regarding gain, bandwidth, and packaging efficiency.

In another publication [84], Yang *et al.* (2019) presented the design of a self-deployable composite boom with an N-shaped cross-section. The design process adopted by the authors includes four distinct steps. First, a design of experiments approach was used, where the sampling designs for the N-boom were created. Then, each design sample was evaluated through numerical analysis in ABAQUS® [11]. The third step was the use of the response surface method to create a surrogate model of the bending stiffness around the x and y axes (transverse) and torsional stiffness around the z axis (longitudinal). Finally, the authors applied a modified non-dominated sorting genetic algorithm (NDSGA) to perform a multi-objective optimization, maximizing the bending stiffness around the x and y axes and the torsional stiffness around the z axis. A mass constraint was imposed, and the two design variables were selected, defining the bonded web height and central angle of the middle tape. The authors concluded that the final design obtained was feasible and that the surrogate model predicted accurately the behaviour of the deployable system with a maximum error of 8.81 %.

More recently, Ferraro and Pellegrino (2019) [29] explored the use of a topology optimization approach to define the geometry of the cut-outs in a composite deployable corner joint. The adopted topology optimization method differs from the original concept proposed by Bendsøe and Kikuchi in 1988 [85], where the element of a material volume had their density changed depending on a specific criterion (such as maximizing the stiffness). Instead, two different approaches were used and compared in [29]: the first was the parametric optimization of the points defining a spline, and the second was a LSM approach. In the first case, the design variables were eight points that could move freely in a constrained space, defining the shape of a spline. The Basin-Hopping algorithm was then used, considering the minimization of the failure index at each consecutive folding step, while maximizing the bending stiffness of the deployed joint. The second approach allowed for a broader exploration of shapes, number, position of the cut-outs. In this case, the definition of the shape of the cut-out is achieved through the intersection of a cutting-plane with a 3-D basis function, $z = f(x, y)$. Two basis functions were investigated, a series of cosines squared, and a series of cosines and sines squared. The optimization process then iterates over the z-coordinate at which the plane will intersect the basis function. To increase the diversity of the solutions, the authors allowed the cutting plane to have an inclination angle to the x-y plane. Both approaches were used to generate the geometry to be simulated in a finite element model, in ABAQUS® [11], that was previously developed and validated in [3]. Despite the innovative approach, the solutions obtained by both methods did not meet the failure criteria. However, the authors mentioned that the designs proposed by the optimization method led to a reduction of the area where damage was

initiated [29].

In previous research, Fernandes *et al.* (2020) [86] attempted the optimization of a self-deployable elastic hinge considering the requirements defined by ESA in the statement of work published in 2016 [6]. The authors considered a total of 10 design variables, two defined the number of plies and orientation of the composite laminate, and a set of eight variables described the geometrical shape of a slot cut-out (inspired in the work of Mallikarachchi *et al.* [49]). The optimization approach consisted of using a genetic algorithm (GA) to perform a global search, iterating over a discrete version of the design variables, followed by a continuous and local search using a particle swarm optimization (PSO) algorithm. The objective was the minimization of the maximum index of failure resulting from the following set of criteria: Hashin's, Azzi-Tsai-Hill, Tsai-Hill, Tsai-Wu, and Maximum stress failure criteria. Additionally, the authors penalized solutions with a natural frequency below the minimum requirement imposed in [6] equivalent to 1 Hz. Although the final design obtained met the frequency requirement, the algorithm did not minimize the maximum index of failure below the maximum acceptable value of 1.0, which indicates damage initiation. Using a parametric analysis, the authors concluded that the design variables suggested in the literature are scarce and limit the design space and possibly exclude better design possibilities. The definition of the slot cut-out through a spline or the use of a topology optimization has been identified as a possible subject of improvement.

In summary, the present literature review shows a clear evolution in the adopted methodologies in the design of deployable structures. The first approaches were based on the analytical evaluation of the stress-strain state of the composite material, establishing a relationship between the maximum strain failure criteria and the folding radius [67, 68]. This methodology was followed by parametric evaluations having finite element models as a valid resource, using either strain [48] or stress-based [49] failure criterion. Finally, the latest observable trend is the inclusion of optimization algorithms in the design process, as well as the inclusion of additional design constraints, such as the natural frequency of vibration, deployed stiffness or the deployment torque [9, 29, 86, 35, 55, 57, 77, 81–84]. However, despite its novelty and popularity in several fields, it is notable the absence of topology optimization-based methodologies, with only one research exploring this possibility [29]. Additionally, it is relevant to note that all the design methodologies reviewed share a conservative approach, disregarding the fact that most deployable structures are single use systems. The use of damage tolerant designs, or allowing the initiation of damage, would allow more demanding requirements, such as higher natural frequencies or a larger deployment stiffness, which are yet to be addressed. [1]

[1] The information reviewed and reported in this section was used in the development of the research described in chapter 3 and chapter4. The reader interested in following the chronological order of the research developed during this project should, therefore, read chapter 3 and chapter4 before moving on to section 2.3. In chapter 3, the limitations of the design methodologies described so far are made evident through their application to the design problem proposed by ESA [6]. On the other hand, chapter 4 demonstrates that the research currently available on relaxation phenomenon can be used to accurately predict the deployment sequence of a self-deployable composite elastic hinge after long periods of stowage. Since it is demonstrated that the influence of relaxation is a predictable problem, the following sections of the literature review focus on design methodologies that may present a solution to meeting both design requirements associated with the development of a composite deployable structure.

2.3 Design of composite structures

Section 2.2 addressed the different methodologies used to design a composite deployable structure. It became evident that the methodologies adopted usually involved the use of FEA and, more recently, the use of optimization methods. The purpose of applying these methodologies is, frequently, to achieve a high-strain capable design that complies with specific requirements, such as: minimum natural frequency, deployed stiffness, or deployment torque.

In other fields of application, damage tolerant design [19, 20] and topology optimization [85] are concepts and methods that have been extensively studied and applied to composite materials and their applications. However, when it comes to deployable structures, the information available on the use of these concepts is either scarce or non-existent.

The following three sub-sections review the literature available on the design of composite structures (2.3.1), topology optimization (2.3.2) and damage tolerance (2.3.3). However, the purpose is not to perform an extensive review on all the design and optimization methods but to focus on assessing the possible application of damage tolerance, topology optimization or other design methods to deployable structures and identify possible advantages of their use.

2.3.1 Design methods

In 2018, Xu *et al.* [87] and Nikbakht *et al.* [88, 89] have presented extensive state-of-the-art reviews on the different design methodologies applied to composite materials and structures. Both conclusions are similar, expressing the general interest of optimizing variables such as fibre orientation in each ply, ply thicknesses, ply number, and stacking sequence. The authors categorized the optimization methods into the following three groups: gradient-based methods, heuristic methods, and hybrid methods. Both research groups concluded that gradient-based methods are usually faster but may require information that is either unavailable or have a high computational cost. In contrast, heuristic methods that do not require gradient information have been widely used, with the most popular algorithms being GA, simulated annealing (SA), PSO, and ant colony optimization (ACO). Finally, the authors have not observed a large application of hybrid methods but consider them a promising tool in the future, when including artificial intelligence. Xu *et al.* [87] mention the relevance of topology optimization applied to laminated composite structures for allowing the simultaneous design of the structural layout and of the fibre orientations.

This section will report and develop on the information published since the release of the reviews made by Xu *et al.* and Nikbakht *et al.* [87–89], sorting the work according to the use of heuristic (2.3.1.1), gradient-based (2.3.1.2), or hybrid and other methods (2.3.1.3). A brief discussion of these three sub-sections is presented at the end of section 2.3.1.3. For its potential, interest and popularity, topology optimization methods will be addressed separately in section 2.3.2.

2.3.1.1 Heuristic methods

In Jin *et al.* [90], the blending of a composite laminate (continuity of the composite stacking sequences as defined in [91]), is addressed proposing the Permutation for Panel Sequence (PPS) blending model, which is an improved version of the Ply Drop Sequence (PDS) concept. Both approaches use a GA whose population has three chromosomes. The first two define the fibre angle and guide distance. The third chromosome differs between PPS and PDS, representing a permutation variable or the existence/absence of a given ply, respectively. The authors state that this change allows the algorithm to avoid the problem of repeated search of discrete points in the design space for the previous PDS blending model, leading to a faster convergence. An *et al.* (2019) [92] also addressed the blending of the composite material but included design constraints. The approach used is an extension of the two-level multi-point approximation method described in [93–95]. The process is divided into two phases. The first begins with an initial stacking sequence design that is used to generate a first-level approximation problem. A GA is used to decide the presence and absence of each ply in the initial design. To ensure the continuity and blending of the different laminate regions, the authors proposed a shared-layer and local mutation method, which forces the individuals in the GA to satisfy the imposed blending rule. The second phase uses a second-level approximation problem to optimize the thickness of the retained layers as continuous variables. The authors efficiently obtained solutions that satisfy both design and manufacturing requirements and successfully applied them to the design of a satellite cylinder.

Regarding performance optimization of the composite, Esmaeeli *et al.* (2019) [96] used a multi-objective implementation of the ACO algorithm to optimize the characteristics of a multifunctional laminated composite. By changing the angles of the plies and simulating an RVE of the material, the authors were able to maximize the effective in-plane elastic constants. Conducting different benchmark problems, the authors observed a quick and successful convergence towards optimal ply angles and the determination of the Pareto optimal frontier regardless of the dimensions of the problem. Similarly, although on the design of active composites, Hamel *et al.* (2019) [97] studied and applied the use of an evolutionary-based design method for 4D printed composites, where 4D refers to the time-evolving shape of 3D printed parts. To do so, the authors combined an evolutionary algorithm with the finite element method to determine the distribution of active and passive material elements. The authors demonstrated the application of this method to single-objective and multi-objective optimization problems.

Neves Carneiro and Conceição António [98], applied a novel methodology, combining the Reliability Index Approach (RIA) [99] with the Reliability-based Robust Design Optimization (RBRDO) [100, 101], to the design of a composite laminate structure. This bi-objective optimization problem aims at minimizing both the weight and the determinant of the variance-covariance matrix of the response functionals of the system (robustness), while considering displacement and stress constraints. The reliability assessment is performed in an inner cycle of design optimization. The authors highlight the exclusive use of a GA with elitist strategy as the key concept of this methodology, avoiding the need of sensitivity analysis, convexity and continuity of the search space,

and guaranteeing global convergence. With this approach, the authors were able to optimize the ply orientation and thickness of the laminate while considering uncertainty in both random design variables and random parameters (robustness assessment), and uncertainty on the individual mechanical properties of each laminate (reliability assessment). More recently, Yoo *et al.* (2021) [102] proposed a novel multi-fidelity modelling-based optimisation framework, aimed at the robust design of composite structures. Through the use of high-fidelity model for exploitation and low-fidelity model for the exploration of the design space, the authors were capable of achieving computational time savings of at least 50% compared to conventional multi-fidelity and high-fidelity modelling methods. This method was successfully applied to the robust design optimization of a stiffened composite panel, considering design uncertainty under non-linear post-buckling regime.

Seeking the simultaneous optimization of both composite geometries and laminate stacking sequence, San *et al.* (2019) [103] applied a multi-island genetic algorithm (MIGA) to maximize the natural frequency of a composite structure. The design variables that characterized the geometry were the points of a non-uniform rational B-spline (NURBS), defined as continuous variables. On the other hand, the ply orientation was defined as a discrete variable with four possible values (0 °, $\pm$45 °, and 90 °). San *et al.* reached two main conclusions. First, that a two-phase optimization was seizing local suboptimal results, contrasting with a simultaneous optimization which led to the global optimum. Second, that increasing the number of control points of the NURBS led to improved optimal results due to the increase in degrees of freedom. These results are in line with the conclusions obtained by Fernandes *et al.* (2020) [86] when using the same design variables to optimize the design of a deployable elastic hinge. However, instead of a MIGA, Fernandes *et al.* used a GA for a global discrete search, followed by a PSO approach for a continuous local optimization.

2.3.1.2 Gradient-based methods

In 2018, Duan *et al.* [104] optimized the topology of a composite truss structure. The objective was to maximize the first natural frequency of the system, made of filament-wound profiles with a circular cross-section and a continuous winding angle. The design variables were the radius and composite orientation angle of each profile. The authors determined the sensitivity of the objective function to each variable and used the Method of Moving Asymptotes (MMA) [105] to update the design variables. The approach was tested and validated in both two and three-dimensional case-studies.

Nasab *et al.* (2018) [106] used a gradient-based approach to the thickness optimization of stiffened composite skins, guaranteeing the blending of plies over individual panels. The approach considered design guidelines such as symmetry, covering ply, disorientation, percentage rule, balance, and contiguity of the layup. To do so, a stacking sequence table is generated and a level-set gradient-based method is used to optimize the location of ply drops. The objective of this procedure is to convert discrete design variables associated with the number of plies into a continuous problem. The output is the optimum thickness distribution over the structure in relation to a specific stacking sequence table. This method was applied to both the 18-panel Horseshoe

Problem and the optimization of a composite stiffened skin of a wing torsion box. In comparison with a GA, the authors concluded that the approach proposed was, in general, faster, and less expensive. Avoiding the direct optimization of the ply-thickness and fibre orientation, Demir *et al.* (2019) [107] applied a least-square optimization approach, with continuity constraints, to optimize the lamination parameters of a composite [108]. The authors argue in favour of this parametrization as it leads to the stiffness becoming a linear function of the lamination parameters, instead of a nonlinear trigonometric function of the fibre angle, making it more suitable for gradient-based methods. Additionally, this parametrization accounts for any number of layers and possible fibre angles [108]. After obtaining the optimized lamination parameters, the authors used a material library consisting of desired fibre angle and stacking sequences. The material whose lamination parameters was most like the optimized result, was the material selected. In other research [109], Shafighfard *et al.* (2019) followed the same approach to determine the lamination properties of different open-hole composite plates, observing a 13 % improvement in compliance of the final design when compared to other state-of-the-art optimization methods.

In 2019, Wang *et al.* [110] proposed the streamline stiffener path optimization (SSPO), a multi-scale-based method for curved stiffener layout design of non-uniform curved grid-stiffened composite structures with embedded stiffeners (NCGCs). The SSPO is based on 4 steps, starting from a homogenization-based analysis to calculate the unstiffened global model. Then, a discrete distribution of two-dimensional curved stiffener paths is converted into a continuous distribution of a streamline function values (SFVs) on a 3D level-set surface. Projected points with the same SFVs are then used to define one stiffener path. The contribution of stiffeners is considered in the global model through the calculation of equivalent material properties of a parallelogram representative cell configurations (RCCs), via homogenization. The third step is the optimization of the curved stiffener layout, achieved using a sensitivity-based shape design of the local parallelogram RCCs with analytical sensitivities. The final step is the maximization of the buckling load within a given weight, considering manufacturing, stiffener spacing and angle constraints. Wang *et al.* validated the effectiveness of steering the path of the stiffeners through the numerical evaluations of a laminate panel under compressive loads.

Oriented towards the design of additively manufactured components, Fernandez *et al.* (2019) [111] included manufacturing constraints of this process in the design of Direct Ink Writing (DIW) short carbon fibre and thermoset resin composites. The authors defined the extrudate trajectory as the contours of level-set functions, which define the orientation of the fibres, their FVF and hence the structural response of the composite. Using nonlinear programming and the finite element method to obtain information on the mass, compliance, and design sensitivities of the structure, it was possible to find a local optimal. In this process, the authors imposed constraints such as: no-overlap, no-sag, minimum allowable radius of curvature and continuity of the toolpaths that must begin and end at a boundary. Finally, the authors formulated a traveling salesman problem to determine the continuous shortest path for each layer, minimizing the manufacturing time. Shen and Branscomb (2020) [112] studied how to find the optimal material orientation of an anisotropic material in an additively manufactured structure. The authors proposed the use of a rationalized formula,

referred to as normalized gradient by maximum (NGM), to calculate the step length of a gradient descent method and find an optimal material orientation that minimized/maximized the compliance of a structure under plane stress conditions. This method was compared with a stress-based and a strain-based method. The strain-based method had the worst performance. The authors also observed that although the stress based method had a faster convergence, the combination of the NGM with the Barzilai-Borwein method (BB) provided a gradual change in orientation in the process and searches the maximum as well, making it a more general method and better suited to handle arbitrary constraints and loads in the finite element method.

In 2020, Nasab *et al.* [113] adopted a decomposition strategy for the structural optimization of a fibre-reinforced aircraft wing box, dividing the problem into a system-level and subsystem-level optimizations. The subsystem problem is the optimization of the ribs which are subjected to the crushing loads resulting from the bending of the wing. The system-level problem is the optimization of the wing-box skins, accounting for the effect of the skin design on the loads applied to the ribs. A principal component analysis (PCA) was used to assess the influence of the changes in loads on the ribs, increasing the numerical efficiency of the decomposition strategy. A level-set strategy that allows the use of both coarse and fine finite element models was adopted to solve the optimization problems in both system and subsystem levels. The results obtained by Nasab *et al.* show that the decomposition strategy allowed the solving of a complex problem at a reasonable computation cost.

2.3.1.3 Hybrid and other methods

Rongrong *et al.* (2018) [114] applied a hybrid approach to the design and optimization of a composite forward-swept wing. The authors generated a surrogate model of the aeroelastic torsion divergence problem using radial basis function neural networks (RBFNNs), which was then solved with a GA. The objective was to minimize the deformation of the wing by optimizing the number of composite layers and their orientation, which were defined as a set of discrete possible values. The authors report that this process allowed a 32.5% reduction of the displacement of the wing. Still related to the use of artificial intelligence, a multi-scale optimization and design approach is presented by Hai *et al.* (2020) in [115]. The authors aimed at maximizing the buckling load of several composite shells, with different cut-out geometries in their centre, by modifying a set of five parameters that define the geometry of the meso-scale RVE. The approach used begins at the micro-scale, estimating the properties of a fibre bundle through a FEA in ABAQUS® [11]. This information is introduced into the open-source software TexGen [116], which can generate meso-scale models of the RVE of the composite tow. This model is used to estimate the mechanical properties of the material as a function of the configuration of the RVE. Finally, the mechanical properties of the material are introduced into a macro-scale numerical model, estimating of the buckling load. The data obtained from the buckling analysis was used to train a back-propagation neural network (BPNN). The BPNN was then integrated, with an Efficient Global Optimization (EGO) algorithm [117], allowing the iteration of the design variables and optimization of the design problem with a reduced computational cost.

Johnston *et al.* (2019) [118] proposed a methodology for the design of a composite hub-web structure that integrates a hybrid composite-steel gear. The geometry consists of a planar structure whose thickness decreases from the hub to the rim. The methodology proposed consists of several steps. The first is the evaluation of an initial simple design. Then, the information on the structural performance of the initial design is considered in a free-shape optimization, which applies orthotropic composite properties to a 3D element mesh of the hybrid gear structure, represented by a bulk volume model. This optimization step aims at minimizing stresses or strains in the part while maintaining key design requirements, such as the connection to other components. The following step is the stress-constrained topology optimization of the result obtained so far, minimizing the weight or volume. In addition to the optimized topology, the expected output includes a loading map of the cross-section of the gear, indicating where continuous-fibre layers are required for greater strength and stiffness. Likewise, lower density filler layers can be added to more voluminous regions where load-bearing capabilities are less required. In case the filler layer is made of a different material from the continuous-fibre composite layer, a second free-shape optimization step is performed to minimize stress or strain in the filler region. Finally, the laminate sequence is optimized, leading to the final design.

Adopting the use of lamination parameters, Liu *et al.* (2019) [119] optimized the lay-up of a composite laminate with the Wittrick-Williams algorithm [120]. Then, instead of selecting a material based on an available library (as in [107, 109]), Liu *et al.* applied a logic-based procedure combining the branch and bound method with a global layer-wise technique to find the optimal stacking sequence that best matches the optimized lamination parameters. An alternative to this second step is presented and validated by Viquerat (2020) in [121], where a set of up to 12 lamination parameters are used to define a laminate configuration through a polynomial homotopy continuation (PHC) technique [122, 123]. The PHC treats the ply angles as continuous variables, taking any value between -90° and +90°.

Khodaygan *et al.* (2020) [124] applied the multi-objective algorithm NSGA-II (Non-dominated Sorting Genetic Algorithm II) to the design of a stiffened laminated composite cylindrical shell with piezoelectric actuators. The objective was to maximize the buckling load and minimize the total weight. The design variables considered were the thicknesses of the three layers (composite, piezoelectric, and stiffener) constrained to a constant total shell thickness. To support the selection of solution from the Pareto fronts, the authors used Shannon's entropy-based TOPSIS algorithm [125, 126], avoiding the use of weighting factors of the objective functions when selecting the final optimal design from the Pareto front. Considering similar approaches and conditions, Albanesi *et al.* (2020) [127] describe the design of a composite wind turbine blade. First, the authors utilized a GA to determine the optimal laminate layout in the outer shell skin of the turbine blade. Then, a topology optimization approach was applied to remove material from the shear webs, considering constraints on the tip displacement, stresses, natural vibration frequencies, and buckling phenomena. The sequential use of a GA and a topology optimization approach led to mass savings of up to 23 % according to the authors.

Although the definition of ply-orientation angles, considering a manufacturing or operational

constraint, is a recurring purpose for the use of heuristic, gradient-based, or hybrid methods, the approach selected to do so differs significantly. The publications reviewed that used heuristic methods were more likely to optimize the ply-orientation considering a discrete set of values [90, 92, 96–98, 103]. On the other hand, research using gradient-based or hybrid methods preferred continuous variables [104, 106, 112, 113], or the optimization of the lamination parameters followed by a second optimization or selection criteria to choose the stacking sequence with the most similar characteristics [107, 109]. This review does not have the extension of the work presented by Xu *et al.* [87] and Nikbakht *et al.* [88, 89], limiting the validity of the statistical analysis on the popularity of the different methods used in the literature. Nevertheless, in this sample, the GA and its variants (NSGA, or MIGA) were the most referred optimization method, which is in line with the reviews of Xu *et al.* [87] and Nikbakht *et al.* [88, 89].

2.3.2 Topology optimization

Topology optimization is one of the three sub-fields of structural optimization, amongst size and shape optimization, and is usually applied in the early stage of structural design. Its purpose is to find an optimal distribution of material [19, 128]. The method defines a design region, divided into several finite elements, to be occupied by the structural component. According to an objective function, the optimization method adjusts the density of each element, defining which elements should have material and which should not [129]. This method also branches into the particular classification of material optimization, when considering micro-structures.

The topology design approach has been applied to composite structures. However, the purpose of this combination is to allow the simultaneous definition of material distribution and fibre orientation. A simple procedure is to align the fibres in the direction of the first principal stress, as adopted by Fuchs *et al.* (1999) [130] and Ma *et al.* (2006) [131]. However, this approach consists of transforming a design variable into a constraint, rather than the optimization of both variables.

In 1999, Hansel and Becker [132] presented a layer-wise topology optimization that considered both material density and fibre orientation. In each layer, the material was removed in areas that either had low stress installed, or that had a fibre orientation that differed significantly from the principal stress direction. The removal was done by changing the density of the element to zero. According to the author, each "laminate element" is constituted by four layers with the common lay-up [0°/±45°/90°], referred to as "single-layer elements". Single-layer elements that are not necessary for the load transfer are removed by reducing the layer thickness to zero. In later research [133], Hansel *et al.* (2002) applied this method to design a laminated composite cantilever plate and an L-shaped cantilever, using a GA to remove the unnecessary material.

To address the topology optimization of laminated composite plates, Stegmann and Lund proposed the gradient-based method called Discrete Material Optimization (DMO), in 2005 [134]. The method is based on the idea of multiphase topology optimization, which does not select exclusively between the inclusion of material or voids, but between the inclusion of any distinct number of materials. The purpose is to find, for each element, one distinct material of the possible candidates such that the objective function is minimized. Stegmann and Lund applied this method

to design a structure subject to a four-point bending test [134] and Niu *et al.* (2010) [135] to design a vibrating laminated composite plate for minimum sound radiation.

With the same purpose, Setoodeh *et al.* (2005) [136] developed a single criterion that simultaneously optimized the composite orientation and thickness of cantilever plates, minimizing the strain energy expressed in terms of fibre orientations and pseudo-densities. To do so, the authors extended the Solid Isotropic Material Penalization (SIMP) technique of topology design with a cellular automata (CA) framework. SIMP is a material interpolation scheme used in topology optimization that considers a continuous variable ρ ($0 < \rho < 1$) that resembles the density of the material [137, 138]. The CA is a methodology used to simulate physical phenomenon based on iterative local updates of both field and design variables. During its functioning, the CA divides the domain of interest into several cells, which only interact with other adjacent cells, performing local computations [139]. Combining both SIMP and CA, the displacements of the structure were calculated and updated to satisfy the local equilibrium of CA cells. Similarly, the fibre angles and density measures were updated based on the optimality criteria.

A different approach was proposed by Zhou and Li [140], in 2008, where the fibre orientations were determined by solving the minimum compliance problem and the densities by a resizing approach based on stress and strain energy. To avoid numerical instabilities, the approach did not remove elements with low density. Instead, after the algorithm had defined the orientation and density of the elements, the authors used a program that defined a truss-like structure based on the orientation of the fibres and the densest areas of the structure. This second phase of the optimization process is usually referred to as a discrete material selection problem. For not removing material from the design volume, the authors deem it unsuitable for the topology optimization of structures with empty spaces, such as a plate with holes.

Gao and Duysinx (2012) [141] proposed a bi-value coding parametrization method, reducing the number of design variables. Instead of considering four different materials, with four design variables representing the density of each possible candidate material, the author considers two design variables with the possible values of -1 and 1. In this case, the material to be attributed to each element is a result of the combination of the two design variables used, leading to a total of 4 possible combinations using only two design variables. These represent the number of candidate materials in a logarithmic form, which makes this approach efficient to deal with large-scale problems, as shown in [142].

In 2013, Huang *et al.* [143] introduced a topology optimization algorithm capable of designing cellular materials and composites with periodic microstructures. The algorithm searches for the microstructure that maximizes the stiffness of the resulting macrostructure. The method is based on the bi-directional evolutionary structural optimization (BESO), which, unlike the evolutionary structural optimization (ESO), considers both the addition and removal of material to the structure [144, 145]. In its essence, the method proposed by Huang *et al.* considers a macro to micro-scale analysis and vice-versa, using two macro-scale and micro-scale finite elements models. The loads applied in the macro-scale are applied to an RVE whose phases are distributed using BESO (figure 2.6). The properties of the new RVE are then used to estimate the behaviour of the

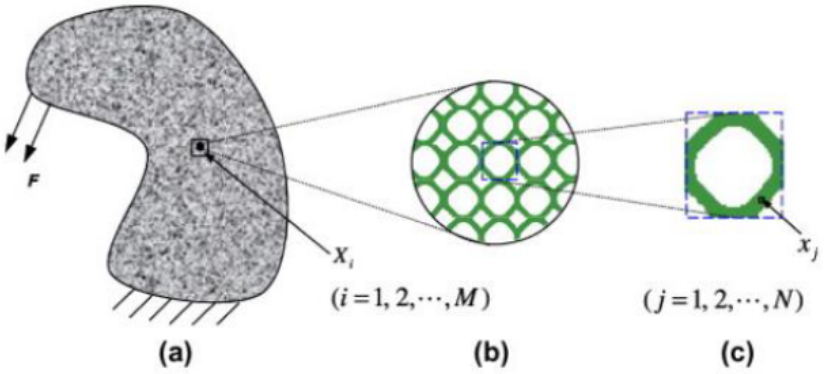

Figure 2.6: Macro-structure composed of cellular or composite materials: (a) macrostructure; (b) microstructure; (c) unit cell or RVE (image adapted from [143]).

structure at a macro-scale, aiming to maximize its stiffness [143]. Several case studies and a more detailed analysis of the application of this method was described by the same authors in [146], published one year later. The cases studied allow the authors to conclude that there was a strong interaction between the two designs at both scales and that the two-scale optimization allowed a better design to be obtained due to the significant increase in the degrees of freedom.

Ren *et al.* (2016) [147] presented a topology optimization method for composite beams. The motivation to this work was the need to address various load cases characteristic of beams used in aircrafts. The authors applied the finite element method to simulate the behaviour of the beam, considering warping and shear deformations. A multi-material optimization model was employed, considering the density of each candidate material at each element as the design variables. The updates of each design variable were done according to a sequential linear programming method. The authors state that the used approach led to a "checkerboard problem". This issue, detailed in [148], is overcome by applying a sensitivity filtering approach, which uses information regarding the neighbourhood of the element under evaluation.

A year later, Wang *et al.* (2017) [149] proposed a level-set topology optimization method (LSM) suitable for the optimization and design of metamaterials. The authors used a numerical homogenization method to evaluate the effective properties of the microstructure and a multiphase level-set model to evolve the boundaries of the multimaterial microstructure. The level-set model was used to implicitly define the interfaces between the material phases by iso-contours of a level-set function (LSF). Depending on the representation of the interface, the use of LSM may improve the accuracy with which the numerical model captures the mechanical response in the vicinity of the boundaries, thus avoiding ambiguities of intermediate material phases associated with the use of density-based approaches [150]. Using the multi-phase level-set model, Wang *et al.* [149] obtained a material geometry with distinct interfaces and smoothed boundaries that facilitate the fabrication of the topologically optimized design. Its application allowed the authors to define a microstructure with a negative Poisson's ratio and a negative coefficient of thermal expansion (figure 2.7). Similar research was reported by Nishi *et al.* (2018) [25], following the

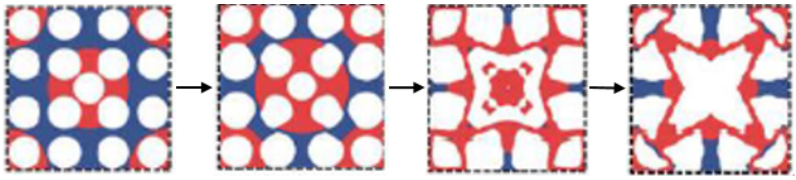

Figure 2.7: Convergence of the method towards a metamaterial with -0.5 Poisson's ratio (image adapted from [149]).

same two-scale topology optimization approach. Likewise, Anaya *et al.* (2019) used a similar procedure but oriented towards the minimization of the effective thermal expansion coefficient of a composite material. The potential improvements obtained through the use of this approach is also supported by the results reported by Panesar *et al.* [151], who measured an improvement between 40% and 50% in terms of stiffness when using an approach that derives graded lattice structures from topology optimized solutions. Furthermore, the strategies identified by Panesar *et al.* also allow easier production of the structure through additive manufacturing.

Dai *et al.* (2017) [152] performed the topology optimization of a composite structure with design dependent loads. To do so, the authors proposed a methodology based on the isoline method, the sensitivity filter with density gradient weighting, and the solid isotropic material with penalization (SIMP) method. The work published reports successful results but is limited to a two-dimensional design.

Sousa *et al.* (2018) [153] analysed the design of multi-layered composite laminates and the topological sensitivity in anisotropic elastistatics, considering the shape/topology of each ply and the stacking sequence as the design variables. To do so, the authors combined a topological sensitivity analysis and an ACO algorithm. The topological sensitivity analysis, based on the total potential energy of the system, provided information on the region where new material should be added. This information can be interpreted as a mapping of the different regions of the structure based on their contribution to the resistance of the component. Then, the ACO method was used to determine the orientation of the added material.

More recently, Tong *et al.* (2019) [24] applied a topology optimization method to design a composite compliant mechanism. This type of mechanism is characterized by its monolithic structure and for transmitting force, energy, and motion through self-deformation [154] (the reader is referred to [155] for a detailed review on the topology optimization of compliant mechanisms). Tong *et al.* designed compliant inverters and grippers, which are commonly used as benchmark problems. The authors predefined the lay-up thickness and the orientation of the composite laminate. The topology optimization was done considering the maximization of the deformation ability of the structure based on the stiffness penalization model. Furthermore, the authors reported that each update of the design variables was done considering a sensitivity analysis, evaluating its influence on the structural behaviour of the component by FEA. Zhu *et al.* (2019) [23] reported a similar research, differing on the use of the LSM and on the use of multi-objective optimization considering one symmetry constraint. In this case, the design volume was limited to the flexible region of the compliant mechanism and the multiple objectives are related to the combination of displacement

and rigidity requirements.

Zhao *et al.* (2019) [156] performed the topology optimization of a compliant mechanism of a composite wing leading edge. The authors used the discrete material optimization method to distribute the multiphase composite material through the design volume. The objective of the optimization was to minimize the least square difference between the deformed curve of the wing and the desired aerodynamic shape. Despite the successful implementation and positive result, it was observed that the number of design variables and consequent computational cost is a major drawback in the discrete material optimization method.

Almeida *et al.* (2019) [157] proposed a methodology to optimize the cross-section of topologically-optimized variable-axial anisotropic composite structures, maximizing specific stiffness. The process relies on sequential optimizations of the topology using SIMP and then optimizations of the cross-section using a GA. The objective was to optimize the number of carbon fibre rovings to be placed at each truss section of a CFRP brake booster. During the topology optimization, the material is considered to be isotropic. The fibre orientation is defined according to the loading direction in the final topology. With this approach, the authors observed a 330 % increase in stiffness of a brake booster system when compared to a commercially available solution.

Safonov (2019) [158] explored the use of topology optimization to design a structure reinforced with continuous fibre via a natural evolution method. The proposed method combined two techniques, simultaneously searching for density distribution and local reinforcement layup in 3D transversely isotropic composite structures. The direction of the reinforcement was aligned in each iteration, considering the direction of the principal stresses and the local minimum compliance, while the density distribution was optimized through a dynamical system method, which replaces the optimization problem with an ordinary differential equation whose equilibrium points coincide with the local optima [159]. When applied to the optimization of a simply supported 2D beam under central point load and a 3D cantilever beam, the authors obtained a mass reduction of 66 % and 90 %, respectively.

In the work published by Jong W. Lee *et al.* (2019) [160], the authors improved a stress-based topology optimization method for laminated composites through the application of the layer-wise theory. In this research, the stress constraint is set so that the composite does not exceed the Tsai-Hill failure criteria. To obtain information on the sensitivity of the stress to the element density and orientation, the authors used one of two types of p-norm stress approximations: one considering all composite layers and another considering a specific layer. The authors observed that the implementation of the proposed approach led to designs that avoided stress concentration points, such as the corner of an L-bracket beam case study, and that the optimization of the laminate angles allowed a further reduction of the used material.

Jaewook Lee *et al.* (2018) [161] proposed a sequential three-step optimization procedure. The first step aims at finding the optimal topology design of matrix material when the FVF is set as zero. Then, the second step determines the optimal density distribution and continuous orientation of fibre material. The third step penalizes intermediate fibre orientations between target orientations, leading to a more discrete design. Lee *et al.* confirmed the advantage of the procedure by comparing

the objective functional in the design results with evenly fixed and optimized target orientations. Furthermore, the authors confirmed that the approach could find the optimal layout of matrix and fibre rich regions in functionally graded composite structures through numerical examples.

Zhou *et al.* (2018) [162] proposed a topology optimization method that simultaneously modifies the topology and material orientation of multi-component composite structures. This approach considers the existence of K components and three layers of design fields. The first layer contains the density field for all components, representing the density of each element. The second layer describes the inclusion of each design point into component k (where k=1,2,...,K). Finally, the third layer determines the material orientation of each component k, considering a continuous variation of the orientation angle as a function of the tension and material stiffness tensor field, for anisotropic materials. The reader is referred to Nomura *et al.* [163] for a detailed description of the material orientation process. Using a single load cantilever beam and a multi-load tandem bicycle frame as benchmark problems, Zhou *et al.* observed that the proposed method generated designs with better structural performance than conventional single-piece isotropic topology optimization methods and a continuous orientation method.

In the work of Jiang *et al.* (2019) [164], the authors report the use of a continuous fibre angle topology optimization method for polymer composite deposition additive manufacturing applications. The authors performed a compliance minimization optimization considering two design variables for each element: the element density and orientation. This parametrization is suitable for the problem as the structure is manufactured layer by layer, avoiding a three-dimensional orientation of the material and simplifying the parametrization of the problem. In [165], Nomura *et al.* (2019) present a different perspective on the material orientation initially proposed in 2015 [163]. This reformulated approach increased its versatility by introducing a different orientation parametrization and by removing underlying assumptions, while still aiming for a simultaneous optimization of the topology and material orientation. In this case, the orientation design variable is formulated as a tensor field, equivalent to a reduced version of the orientation tensor to represent a single direction at a point. This method avoids singular point issue that occur in the rotation-based approach (equivalence of angle 0 and 2π radians) and ensures the unique correspondence between each tensor value and material property. Nomura *et al.* observed clear performance advantages in both two and three-dimensional problems, as well as single and multi-load cases when compared to methods that define the topology first and then the material orientation.

Tong *et al.* (2019) [166, 167] performed the topology optimization of a composite laminate, as well as the optimization of the ply-angles considering the lamination parameters. This sensitivity-based approach is divided into two steps. In the first step, the lamination parameters and the density are set as the design variables, while the structural stiffness is parametrized by the lamination parameters. Minimizing the compliance with a volume constraint led to a final topology and to a set of lamination parameters that maximize the stiffness of the structure. Then, the orientation of the plies is retrieved by solving a set of nonlinear equations. To solve this second problem, the authors transformed it into a least-square optimization problem, finding the stacking sequence that

best matches the initial lamination parameters.

Fu *et al.* (2019) [168] proposed a substructuring approach that allows the topology optimization on both macro and micro-scales. The microstructure of the material is optimized considering its element volume constraint while minimizing the mean compliance of the macrostructure. The transition between the two scales is done through the homogenization of the material properties of the RVE, which are then introduced in the macro-scale FEA. A similar approach was proposed by Wu *et al.* (2019) [169] and Jansen and Pierard (2020) [170]. However, the authors used the multi-scale design approach to hierarchical lattice structures and functionally graded lattice structures, respectively. Regarding composite materials, Wu *et al.* (2020) [171] studied the robust concurrent topology optimization (CTO [172–175]) of two-phase composite materials, using an improved hybrid perturbation analysis (IHPA) method to assess the worst performance of the structure under a random model based imprecise probability. More focused on fibre reinforced materials, Yan *et al.* (2019) [176] presented a CTO design method that optimizes the topology of the macrostructure, the material microstructure, and the material orientation. This two-scale approach is based on the BESO method and utilizes the information of the local principal stress direction to determine the orientation of the material in each element. Yan *et al.* observed that optimizing the topology and microstructure improved the structural performance of the final design and that, usually, the resulting microstructure was anisotropic. In later research [177] Yan *et al.* extended the application of this method to three-dimensional case-studies, observing similar results. Gao *et al.* (2019) [178] and Li *et al.* (2019) [179] followed a similar procedure but applied the SIMP and LSM methods instead of BESO, respectively. Likewise, Zhang *et al.* (2020) [180–182] applied a multiscale method to find the optimal topology that minimizes the frequency response of a cellular composite within a given frequency range.

Focused on the definition of the material orientation, Silva *et al.* (2020) [183] revisited and modified the normal distribution fibre optimization (NDFO) method. Therefore, the values of angles are inserted directly into a normal distribution function and the output is modified with a Helmholtz filter, ensuring the continuity of the fibres. This modification allowed the lifting of some restrictions, such as imposing small strains, displacements, and rotations, and consequently expanding the range of application of this approach.

In the current state of the art, it is possible to observe an evolution of the topology optimization methods applied to composite materials. The focal point of research in this area has been the inclusion of design variables and their efficient management. From a physical perspective, this means the addition of material properties, such as the strength of the material [160], the orientation of the fibre reinforcements [24, 132, 133, 136, 140, 141, 153, 158, 160, 162, 163, 165, 183], lamination parameters [166, 167] or information on the microstructure of the composite and its influence on a macro-scale level [25, 143, 146, 149, 168–170, 176–182, 184], the inclusion of multiple materials [134, 135, 156], and manufacturing or process constraints [164]. A topic of research scarcely explored is the inclusion of fracture or damage predictive models in the topology optimization of composites, only referred to in [160] where the authors used the Tsai-Hill failure criteria, which was imposed as a maximum stress constraint. In research not restricted

to composite materials, the inclusion of damage tolerance or damage predictive models, in the topology optimization process, has already been under investigation since 1998 [185, 186]. The combination of these concepts has been used as a conservative approach, leading to designs with multiple load paths that enable a structure to function in the presence of damage [187].

2.3.3 Damage tolerance

The existence of defects in composite materials, either resulting from the manufacturing process or due to loads applied during operation, can cause changes in the performance of the structure. The study of damage tolerance of materials has been a subject of research for many years, whether motivated by increased performance and/or safety factors, or simply to avoid the costs of maintenance and discard of components [19, 20]. This concept defines the ability of a structure to sustain a determined level of fatigue, corrosion, or impact damage able to be detected and repaired. Traditionally, the damage tolerance of metallic structures is governed by damage resulting from fatigue and crack propagation [21].

Whether the distinction between the relevant sources of damage, that is more harmful to each type of material is correct or not, it is expected that the exposure to operating conditions and external factors promotes the initiation, accumulation and propagation of damage [19]. This has led researchers to perform extensive research on the influence of different load cases and external conditions on the damage tolerance of composite materials, such as: the influence of fabric architecture and resin toughness on the impact resistance ([188–196] and [194–204], respectively) and damage tolerance ([194, 195, 205–208] and [203, 209–213], respectively) of CFRP, the effect of fracture toughness [192, 193, 198, 202, 214–218], repeated impact [219–224], impact geometry [225–234], stacking sequence [201, 235–239], environmental conditions [240–247] including radiation [248–251], and fabric [199, 200, 235, 252] or matrix hybridization [247, 252–256] on the damage tolerance of the material.

The extensive research performed throughout the years has supported several industries, promoting the adoption of damage tolerance concepts in the design of multiple structures. Braga *et al.* (2014) [257] reviewed and discussed possible prospects of the major design philosophies that have been employed in aircraft structures, including damage tolerance. In this sector, the concept of damage tolerance introduced the assumption that initial structural damage exists in the structure, making it a requirement that needs to be considered. The objective of this assumption was to determine inspection thresholds and intervals. To do so, fracture mechanics evaluations of crack growth and residual strength characteristics were coupled with damage detection assessments. The data obtained from service-based crack detection procedures, combined with the residual strength and fatigue crack growth data, is used to define detection reliability ratings, considering multiple types of inspections. The definition of the inspection thresholds and intervals promoted the focus of research on non-destructive inspection (NDI) techniques, such as: dye penetrant inspection, magnetic particle inspection, radiography, ultrasonic inspection, Eddy currents, thermal imaging, and digital image correlation (DIC).

In 2015, McGugan *et al.* [20] proposed a design and maintenance methodology to be applied in wind turbine rotor blades, assuming that either it is not possible to manufacture a perfect structure, or that discarding or repairing defects in a structure is too costly. Besides the use of a structural health monitoring system (SHM) to assess the state of the structure, one of the key aspects of the design method proposed was the inclusion of a damage tolerance index, coupling the materials and the structure. This methodology requires the use of materials whose strength is significantly higher than the linear-elastic limit, to allow an easy detection of the damage and enable the possibility of repairing or replacing the damaged part. Similarly, Yue *et al.* [258] proposed a design philosophy that is based on the structural health monitoring and associated testing at different levels of complexity, starting from a coupon level. According to the authors, using the information obtained through the analysis of previous levels, it is possible to detect multiple barely visible impact damage in large composite panels by outlier analysis using a reference pristine database gathered from simple coupons. From a more general point of view, this bottom up approach of increasing complexity provides valuable knowledge that will more accurately predict the health status of the design component.

The interest of the industry in this concept, combined with the limitations of the analytical solutions available, promoted the research and development of numerical tools capable of predicting the damage initiation and propagation in a structure after a certain loading condition. Damage modelling can be divided into four categories: failure criteria based, continuum damage-mechanics based, fracture-mechanics based, and plasticity or yield-surface based models. Failure criteria-based models use polynomial expressions to define a failure envelope that indicates the initiation of damage as a function of the stress or strain installed. However, it does not include information regarding the position, size, and progression of the crack. Fracture mechanics addressed this issue through the inclusion of information regarding the energy required to create and propagate a crack. In turn, fracture mechanics require information regarding the initial flaw. In composite materials, the prediction of progressive damage can be achieved through the combination of these two methods. For ductile composites, this approach can be complemented with plasticity-based damage models [22].

Likewise, the prediction of damage tolerance is a topic of significant interest. However, simulating residual strength tests is quite challenging. Gonzalez *et al.* (2012) [208] simulated the low-velocity impact and the compression after impact (CAI) tests, using both interlaminar and intralaminar (LaRC04 failure criteria [259, 260]) damage models. This extremely expensive computational model predicted the residual strength with a 20 % error, resulting from the comparison between numerical and experimental results.

Rivallant *et al.* (2013) [261] and Hongkarnjanakul *et al.* (2013) [262] successfully captured the permanent indentation resulting from a CAI test, as well as the crack propagation and buckling of sub laminates due to the impact. The simulations were an improved version of the model proposed by Bouvet, et al (2012) [263], considering the failure of fibres under compressive loads.

Camanho *et al.* (2015) [264], proposed a three-dimensional failure criterion for CFRP based on structural tensors. The criteria for the transverse failure was formulated from the invariant theory,

while the tensile fracture in the fibre direction is predicted using the maximum strain criterion. A three-dimensional kinking model was used to predict the longitudinal compressive failure, capable of accounting for the nonlinear shear response. The criteria was proved both accurate and useful in the validation of failure under complex three-dimensional stress states.

Tan *et al.* (2015) [265] proposed a nonlinear shear-based damage model. In their research, the authors aimed to predict the compressive residual strength of the composite through the coupling of the matrix tensile failure criteria proposed by Puck and Schurmann (2004) [266] and the compressive failure criteria proposed by Catalanotti *et al.* (2013) [267]. A three-step process was used to simulate the low-velocity impact, stabilization of the specimen and update of the boundary conditions, and the CAI analysis. The results report a high accuracy in the residual strength predicted and successfully captured the permanent indentations.

Caputo *et al.* (2015) [268] simulated a low-velocity impact and CAI test in ABAQUS® [11]. Using a single step analysis allowed the authors to consider the effect of impact damage distribution as the starting configuration of the CAI test. Through this methodology, the authors were able to predict the interlaminar and intralaminar damage with a 9 % accuracy. Elias et. Al (2017) [269] divided this method into a two-step process. The purpose of this division was to execute the low-velocity impact first, to obtain the damage pattern, including indentation depth and damage area, as well as the damage indices on each element. The second step focuses on estimating the residual strength. The results further support an accurate prediction of the damage mechanisms.

A simplified approach to predict CAI strength in laminated composites was proposed by Rozylo *et al.* (2017) [270]. In this research, a relationship between the thickness of the plies and the impact energies was established. A progressive damage criterion describing the initiation of damage according to the Hashin model was used, while the propagation was determined using the established energy model. The numerical results were in good agreement with the experimental data reported by Tan *et al.* (2015) [265].

Abir *et al.* (2017) [217] adopted a single-step approach using Tsai-Wu failure criteria for damage initiation. The authors observed that local buckling and delamination growth caused the failure of the composite under CAI. A high influence of the Mode II interlaminar and fibre compressive fracture toughness was also observed, whose increase reduced the delamination size and improved the damage tolerance.

In 2018, Cugnoni *et al.* [271] performed an extensive experimental testing campaign, addressing the influence of ply thickness, fibre, matrix, and interlayer toughening on strength and damage tolerance. The detailed characterization allowed the authors to obtain a master curve diagram, modelling the reduced in-situ strength of the composite as a function of the ply thickness. This master curve was further expanded for larger values of thickness using the models proposed by Dvorak and Laws (1987) [272], and was later extended by Camanho *et al.* (2006) [273]. The master curve obtained is overall conservative and suitable for first order estimates.

Liu *et al.* (2018) [197] presented a numerical model to simulate the CAI of hybrid unidirectional or woven CFRP laminates. Through a user-defined material subroutine in ABAQUS® [11], the authors implemented a three-dimensional damage model based on continuum damage mechanics

and linear elastic fracture mechanics. The model considered interlaminar and intralaminar damage, load reversal, and nonlinear shear profiles to account for matrix plasticity. The experimental results agreed with the numerical predictions, with the numerical model capturing the behaviour of the composite under compressive loading.

Recent research has studied the possibility of improving the damage tolerance through novel materials. To improve the resistance of composite laminates to delamination when subjected to impact loadings, Daelemans *et al.* (2019) [274] studied the influence of electrospun nanofibers on the occurrence of delamination failure in composites. An increase of 60 % of the delamination resistance under both Modes I and II was estimated. Dimoka *et al.* (2019) [275] have also investigated the influence of nanomaterials in the damage tolerance of a CFRP laminate reinforced with multi-walled carbon nanotubes (MWCNTs) subject to low-velocity impact and then tested under compression loads. The results revealed an improved performance in the residual compressive strength after impact when compared to the non-modified CFRP. For impact tests with high energy levels (30 J), an increase in the delamination area was observed. However, the opposite phenomenon was observed for impact energy levels of 8 J and 15 J.

Tie *et al.* (2020) [276] maximized the impact-resistance of a patch repaired CFRP laminate using a surrogate-based model based on the Diffuse Approximation and Design of Experiments, which obtained information from FEM of patch-repaired CFRP laminates that consider continuum damage mechanics (CDM) and cohesive zone modelling (CZM). The outcome observed by Tie *et al.* was a significant increase in the impact-resistance, decreasing the impact energy absorption and the delamination surface area.

On the use of these numerical models towards the design of composite structures, Sellitto *et al.* (2020) [277] applied a GA to optimize the stacking sequence of an aeronautical stiffened panel. The optimization had two objectives: maximizing the buckling load and minimizing the weight among the configurations capable of withstanding a low-velocity impact. The authors considered four possible ply orientations [0°,+45°,-45°,90°] and applied a linear damage criterion to assess the structural integrity of each configuration. This approach led to two possible configurations that minimize the number of plies of the panel and maximize the buckling load. Similarly, Reddy *et al.* (2020) [278] developed an enhanced bat algorithm (EBA) and used it to optimize the number of plies and their orientation. The objective was the weight minimization of the laminate, constrained by the initiation of damage according to the Tsai-Wu failure criterion.

Particularly concerned with the buckling and post-buckling behaviour of thin-walled composite laminated beams and columns, Mittelstedt (2020) [279] reported and extensive literature research on the topic. Mittelstedt has observed that almost all theoretical approaches toward the constitutive modelling and global-local buckling analysis rely on either classical laminated plate theory or first-order shear deformation theory, contrasting with the often need of using higher-order shear deformation theory for composite laminated materials. Furthermore, Mittelstedt identifies the need for further investigation on the influence of delamination on the static performance, buckling resistance, buckling and post-buckling behaviour of thin-walled composite beams. The author also suggests further investigation on progressive failure analysis, studying composite beams in the

post-buckling considering damage accumulation models. For a more detailed analysis on each of these topics, the interested reader is referred to [279].

A first special mention is given to Fedulov and Fedorenko (2020) [280], for combining topology optimization concepts with the analysis of damage propagation in a composite material. The authors proposed a method based on the inverse approach of a standard compliance minimization topology optimization problem to estimate the residual strength of a laminated composite with barely visible impact damage. In other words, the energy transmitted by a given impact is distributed in the composite material through a sensitivity analysis, maximizing the compliance of the structure. This energy is then used to degrade the material properties of the composite according to a progressive degradation material model. A good correlation between predicted and experimental results was observed.

A second, and final, special mention is given to the study of imperfections in the design of deployable structures. As stated by Chen *et al.*, kinematic singularity can frequently exist in deployable structures, especially when adjacent links become coplanar, affecting the accuracy, deployment performance and structural stiffness [281]. Addressing mechanisms on a more general level, of which deployable structures are part of, Lengyel and You [282] have demonstrated that the bifurcations of several mechanisms correspond to various catastrophe germs, highlighting the influence of imperfections on the behaviour of the mechanism. An interesting approach is taken by Steinboeck *et al.* [283], who investigated the necessary and/or sufficient conditions to achieve an imperfection insensitive system based on Koiter's initial post-buckling analysis [284]. Since kinematic bifurcation is generally unavoidable for deployable structures, Chen *et al.* [285] follow a different route, proposing a methodology that decomposes the compatibility matrix of a deployable structure and extracts new mechanism modes with lower-order symmetries associated with independent bifurcation paths. Then, a prediction-correction algorithm is used, leading the structure into the expected bifurcation paths.

Through the review of the state of the art, it is possible to infer that the concept of damage tolerance is relevant for several industries. This interest has led to extensive research on the influence of external factors and loads on the damage tolerance of materials, the simulation of damage initiation and propagation, and the prediction of residual properties. More recently, the development of new materials has promoted the study and research of the influence of nanomaterials on the damage tolerance of the modified structure. However, the vast majority of these investigations are focused on the development or improvement of existing tools to allow an accurate prediction of the damage incurred in a given operating condition, with only a few recent articles actively using this information in the design of a composite structure [29, 49, 86, 67, 276–278].

2.4 Final remarks on the design and optimization of self-deployable damage tolerant composite structure

The present review explores the possibility of increasing the performance of a composite self-deployable structure by means of the allowance of damage initiation during operation. The use of

Table 2.1: Number of publications on the subject of composite structures design sorted by the methodology and use, or not, of damage tolerance.

Composite structure type	**Damage tolerance**	**Design approach**				
		Analytical	**Numerical**			**Topology Optimization**
			Heuristic	**Gradient-based**	**Hybrid**	
Elastic hinge	Without	2	11	4	1	1
	With	0	0	0	0	0
Others	Without	1	8	9	7	48
	With	0	3	1	2	1

a damage tolerant structure is identified as a possible solution to meet highly demanding design requirements identified by the ESA for self-deployable telecommunication satellites [6]: achieving a design that is, simultaneously, flexible to fold in the elastic regime, but also rigid enough to reach high natural frequencies. Opting for a damage tolerant design, and relaxing the need for the structure to operate in the elastic regime, is seen as a potential solution to reaching the required natural frequency of vibration through the increase of the stiffness of the structure previously limited by the requirement of an elastic design.

The proposal of a damage tolerant design is justified by two particularities of this application. The first, is the life cycle of the structure, as most deployable systems are expected to perform a single deployment operation once in orbit. While imposing the functioning of the deployable system in the elastic regime does lead to a higher safety factor, it also implies a significant over-design of the structure and, in this case, limits the maximum stiffness and its resulting natural frequency. The second, is the expected life-time of a satellite. Apart from their size and cost, the development of nanosatellites and CubeSats is also motivated by their reduced development time. Average or large-sized satellites require between 5 and 15 years to be placed in orbit under normal parameters, incurring the risk of no longer being market-relevant due to the pace of technological progress. In contrast, CubeSats and nanosatellites require less than 8 months to reach orbit. This trend towards a shorter development time allows a frequent renewal, guarantees the robustness of nano-satellite constellations, and removes the need for a conservative long-term design [286–289]. However, it is

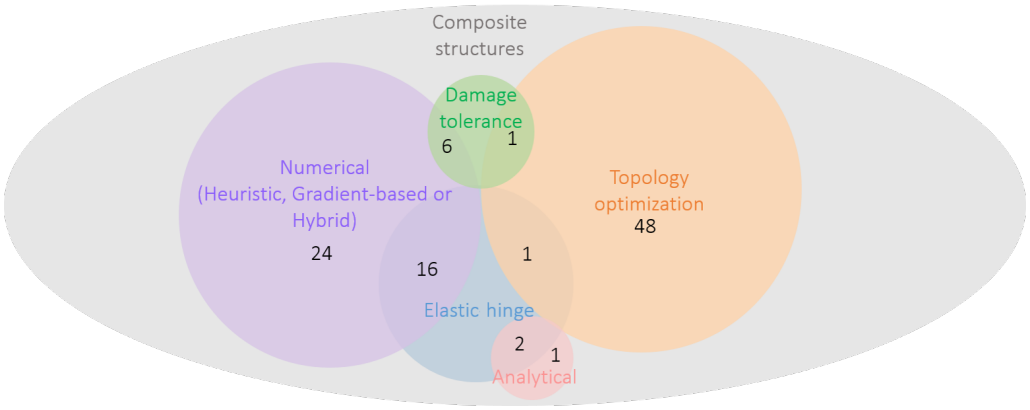

Figure 2.8: Venn diagram showing the number of publications on the subject of composite structures design sorted by the methodology and use, or not, of damage tolerance.

important to acknowledge a potential downside of using a damage tolerant design in this application, which is the possible release of debris when the initiation of damage occurs. The inclusion of containing systems, similar to a membrane that covers the elastic hinge, can be used to overcome this phenomenon. Therefore, small debris will no longer be considered as a limiting factor to the exploration of this concept.

Three research topics were surveyed to look for a suitable design methodology that could lead to a composite self-deployable damage tolerant design. These research topics included: the design of deployable structures (section 2.2), the design of composite structures (section 2.3.1), and the use of damage tolerance in composite structures (section 2.3.3).

The survey of the design approaches applied to deployable structures allowed the observation of a clear evolution from the use of analytical methods [67, 68] towards the use of numerical approaches combined with optimization algorithms [9, 29, 86, 35, 55, 57, 77, 81–84]. The design tools changed and stress [49] and strain-based [48] criteria were developed specifically for materials used in deployable systems. However, the same conservative design philosophy was maintained across all research: ensuring the functioning of the deployable system in the elastic regime.

Comparing the design methodologies used for deployable structures with methodologies used for other composite structures, it is possible to identify similarities in the heuristic, gradient-based, and hybrid methods. In all cases, it is common for the design process to involve the optimization of the stacking sequence of the laminate, considering either a discrete or continuous ply-angle variation, respectively, and the optimization of several geometrical variables. The only difference noticed was the optimization of the lamination parameters, which was not observed when designing deployable structures, probably due to the fewer publications on composite self-deployable structures.

On the other hand, the popularity of using a topology optimization is significantly different. Only one research [29] explored the use of this approach to design an elastic hinge. In other fields of application, the topology optimization of composite materials has evolved significantly, in particular regarding the efficient inclusion of additional material properties such as the strength of the material [160], the orientation of the fibre reinforcements [24, 132, 133, 136, 140, 141, 153, 158, 160, 162, 163, 165, 183], lamination parameters [166, 167] or information of the microstructure of the composite and its influence on a macro-scale level [25, 143, 146, 149, 184, 168–170, 176–182], the inclusion of multiple materials [134, 135, 156], and manufacturing or process constraints [164].

Once again, the inclusion of fracture or damage predictive models in the topology optimization of composite materials is scarcely addressed [160]. This observation was unexpected. In research not related to composite materials, this has been a topic of research since 1998 [185, 186] and the outputs have been relevant in several industries [19, 20]. Regarding composite materials, the capability to predict the initiation and the propagation of damage in composite materials, as well as to estimate the residual properties has been extensively explored [188–202, 204–208, 203, 209–247, 252–256] and is particularly relevant in industries such as aeronautics [290, 291], where this information can be useful to ensure the functioning of the structure in anomalous conditions. However, only few recent articles actively use this information in the design of a composite structure [29, 49, 86, 67, 276–278].

In summary, reviewing the published literature on these subjects allowed the identification of several research topics that have not been addressed in the literature (highlighted in Table 2.1 and figure 2.8). The use of topology optimization coupled with damage has either been rarely or never used in the design of self-deployable composite structures. The extensive research of these concepts in other areas indicates that these approaches have reached a relevant maturity level that could justify exploring their application to this particular scenario. Furthermore, no limiting factors that could exclude the use of these concepts in the design of self-deployable composite structures were identified in this review. Considering this information, modifying a stress-constraint topology optimization approach to consider the maximum index of failure resulting from composite failure criterion, the energy dissipated through material degradation, or a maximum area of damaged material, could be a possible approach to consider damage tolerance in the topology optimization of a self-deployable composite structure. [2]

[2]The reader interested in following the chronological order of the research reported in this thesis should move to chapter 5, where the use of a damage tolerant elastic hinge design is proposed and evaluated. Then, chapter 6 takes a closer look at the method of topology optimization, analysing its possible implementation according to the state of the art, and ultimately identifying some limitations that must be overcome before adapting it to consider a damage constraint. From chapter 5 onwards, the chapter sequence matches the chronological order.

Chapter 3

Design and optimization of a self-deployable composite elastic hinge

The present chapter is based on the following refereed publication:

P. Fernandes, R. Marques, R. Pinto, P. Mimoso, J. Rodrigues, A. Silva, João Manuel R.S. Tavares, G.Rodrigues, N.Correia. *Design and optimization of a self-deployable composite structure*. In MAT-COMP'19 - Composites for Industry 4.0, Vigo, Spain, 2020. ISSN: 2531-0739

3.1 Introduction

Given the difficulties involved in the design of deployable structures, the increasing requirements and their potential use, this paper aims at applying optimization algorithms to find an optimal solution of an elastic hinge. The damage and frequency requirements set by ESA in [6] are used as an example of a state-of-the-art engineering problem. During this research, numerical models capable of estimating the natural frequency and the structural integrity of the structure were implemented. The structural model was experimentally validated using a representative specimen of the elastic hinge, comparing the numerical and experimental folding behaviour and the forces involved in the process. The experimental validation of the natural frequency model was performed and reported in [82], therefore, it was not repeated in this investigation. The numerical models were integrated into an optimization process that used them to estimate the frequency and structural behaviour of possible elastic hinge designs. The optimization process was divided into two sections: first, a global and discrete search that utilized a genetic algorithm, then, a local and continuous search through a particle swarm optimization method. However, the output obtained did not comply with the design requirements. Therefore, a second iteration of the slot hinge geometry was investigated numerically, and the results were discussed.

3.2 Design requirements

In this research, the deployable structure considered is a composite arm with two integrated elastic hinges. Each end of the arm should be connected to the satellite and to an antenna, respectively. Figure 3.1 best describes the application.

The design requirements include geometric and functional requisites. From a functional point of view, the deployable structure should have a natural frequency higher than 1.0 Hz and should not initiate damage during operation. In this research, the damage requirement is evaluated using different failure criteria: Hashin's failure criteria, Azzi-Tsai-Hill, Tsai-Hill, Tsai-Wu and Maximum Stress. It is assumed that no damage has been initiated during operation if the maximum index of failure (Max. IF) of these criteria is lower than 1.0. From a geometric perspective, it is imposed that the arm should have a circular cross-section, whose diameter cannot exceed 200.0 mm. These correspond to the minimum requirements for this application, as reported in [6]. To obtain a more robust solution, a safety factor of 10 % was applied to the frequency and damage requirements. Therefore, the natural frequency should be higher than 1.1 Hz and the index of failure lower than

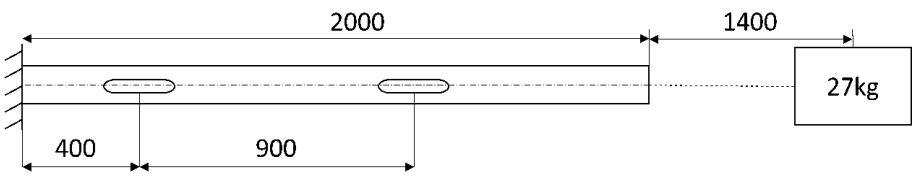

Figure 3.1: Description of the deployable system considered in this research.

0.9 for all the criteria under assessment. Additionally, it is assumed that the location of the elastic hinges in the deployable arm is fixed and that the antenna to be attached on its free-end has a mass of 27 kg. The centre of mass of the antenna is located 1.4 m away from the tip of the arm. As a result of these restrictions, the modifiable parameters are: the radius of the arm, the material, number of plies, ply-orientation and the geometry of the slot that characterizes the elastic hinge.

3.3 Numerical details

Finite element analysis was carried out to estimate the natural frequency and structural integrity of the system, using ABAQUS® [11]. In this section, the implemented numerical models are described. Both models consider a parametrization of the modelled geometry, which is described in detail in section 3.5.1, allowing an easy modification of the numerical models.

3.3.1 Natural frequency model

The natural frequency finite element model considers the integrated system composed of the composite arm with two elastic hinges and the antenna attached to its free-end. The antenna is represented as a point mass of 27 kg, located 1.4 m away from the tip of the composite arm and connected to it through a beam multi-point constraint. The other end of the composite arm is fixed, representing its attachment to the satellite. Figure 3.2 represents these boundary conditions applied to a generic antenna arm.

The deployable arm was modelled with three-dimensional deformable shell elements with 4 nodes and reduced integration (S4R in ABAQUS® [11]). Each element had an average size of 2.5 mm. The material properties introduced in the numerical model are summarized in Table 3.1. The natural frequency is determined using a linear perturbation analysis, which applies the Lanczos algorithm. This approach has been used several times in the literature to estimate the natural frequency of similar deployable structures [10, 53, 49] and even some different concepts [292]. It has been experimentally validated in [82], reporting an error of 6 % between numerical and experimental results for the first natural frequency and 20 % for the higher-order frequencies. Since the objective of the natural frequency model is to estimate the first natural frequency of the system and that an error of only 6 % was observed through experimental validation, the numerical model is considered suitable for this investigation. This numerical model considers a parametrization of the geometry of the structure, allowing an easy modification of the geometry of the modelled structure. This FEA is then solved through an implicit analysis.

Figure 3.2: Highlight of the boundary conditions considered in the natural frequency model.

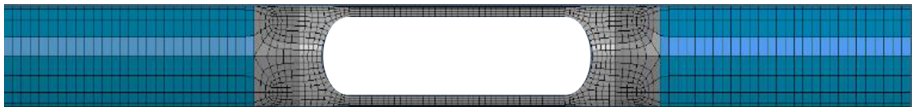

Figure 3.3: Highlight the different regions considered in the structural model. Each region is represented with a different colour (blue, dark grey and light grey) and has a different mesh size.

3.3.2 Structural model

The structural model considers a segment of the composite arm, corresponding to only one of the elastic hinge present in the design. Similarly to the natural frequency model, three-dimensional deformable shell elements with 4 nodes and reduced integration (S4R in ABAQUS® [11]) were used to model the elastic hinge. For optimization, convergence reasons and due to the non-linear nature of the model, an explicit analysis was preferred and the elastic hinge was divided into three regions, as shown in figure 3.3.

The central region, highlighted with a light-grey colour, corresponds to the area where the slot of the elastic hinge is located and where the highest stress concentration occurs. Therefore, the numerical model considers the existence of a composite layup feature with multiple plies in this region, allowing a more detailed analysis of the stress and strain state of each element. The elements in this region had an approximate size of 2.5 mm. The other two regions do not sustain significant stresses or strains. This fact makes them only relevant for the analysis of the folding and deployment behaviours of the elastic hinge to capture the full component rigidity behaviour when solicited. Since the stress and strain states are not significant from a design point of view, these were modelled as a single layer of elements with the homogenized properties of the composite. This procedure reduces the number of calculations in the model and, therefore, the computational time. The elements in the blue region had an approximate average size of 4.0 mm, while the elements in the dark-grey region create a transition between neighbouring meshes. In order to replicate the experimental testing equipment, the applied boundary conditions consider the existence of two holders, responsible for supporting the ends of the composite arm. An angular rotation is applied, forcing the arm to fold, and two plates, that pinch the tapes of the elastic hinge, force them to bend before the folding of the component starts. These boundary conditions are per [10] and represented in figure 2.4 (image cited from [10]).

3.4 Validation of the structural model

The structural model was validated experimentally by comparing the experimental and numerical torque-angle curves. These curves indicate the total torque that is applied at the ends of the elastic hinge to fold it, as a function of the folding angle, and were first used as a validation method in [49] and [30]. The following sections describe in detail the materials used, geometry of the specimens, experimental setup, calibration procedure and output comparison.

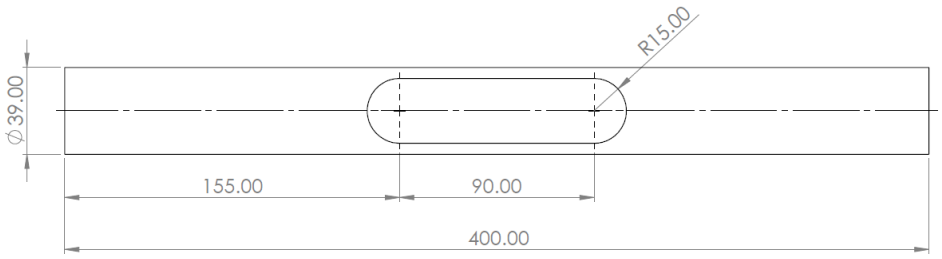

Figure 3.4: Geometry of the specimen used for the experimental validation of the structural model.

3.4.1 Materials and specimens

The material considered in this research was AS4/8552, a unidirectional carbon/epoxy system provided by Hexcel®. The mechanical properties implemented in the models are summarized in Table 3.1, where E is the Young's Modulus, v the Poisson's ratio, G the shear modulus, X the longitudinal strength, Y the transverse strength, S the shear strength, the subscripts 1, 2 and 3 indicate the longitudinal, transverse, and normal directions, and the subscripts t and c denote the tensile or compressive strength.

The geometry of the specimens used in the experimental validation is described by figure 3.4. This geometry consists of a tubular structure with two 0.18 mm thick plies (rounded to two decimal places) oriented at $\pm 45°$, an internal diameter of 39 mm and a cut-out, defining the slot of the elastic hinge. Each specimen was manufactured by hand lay-up and cured in an autoclave according to the manufacturer specifications. Then, the slot was cut using a CNC machine.

3.4.2 Experimental setup and procedure

The torque necessary to fold the elastic hinge as a function of the folding angle was used to validate the numerical model. A custom test rig (figure 3.5) was developed based on the work described in [49, 30]. The rig consists of two sets of holders that support the composite tube, and can apply torsion at both ends of the tube. The torsion is controlled by the movement of two stepper motors. One of the stepper motors is fixed to the base of the machine, allowing the holder to rotate only

Table 3.1: Elastic and strength properties of AS4/8552, according to [208] and [293], respectively.

	Property	Value	Unit
Elastic properties	E_{11}	128	GPa
	E_{22}	7625	MPa
	v_{12}	0.35	-
	v_{23}	0.45	-
	G_{12}	4358	MPa
Strength properties	X_t	2300	MPa
	X_c	1200	MPa
	Y_c	220	MPa
	S_{12}	100	MPa

around a fixed axis. The other motor is fixed to a moving base, allowing the holder to rotate while moving along a linear track. Figure 3.5 shows the mechanism.

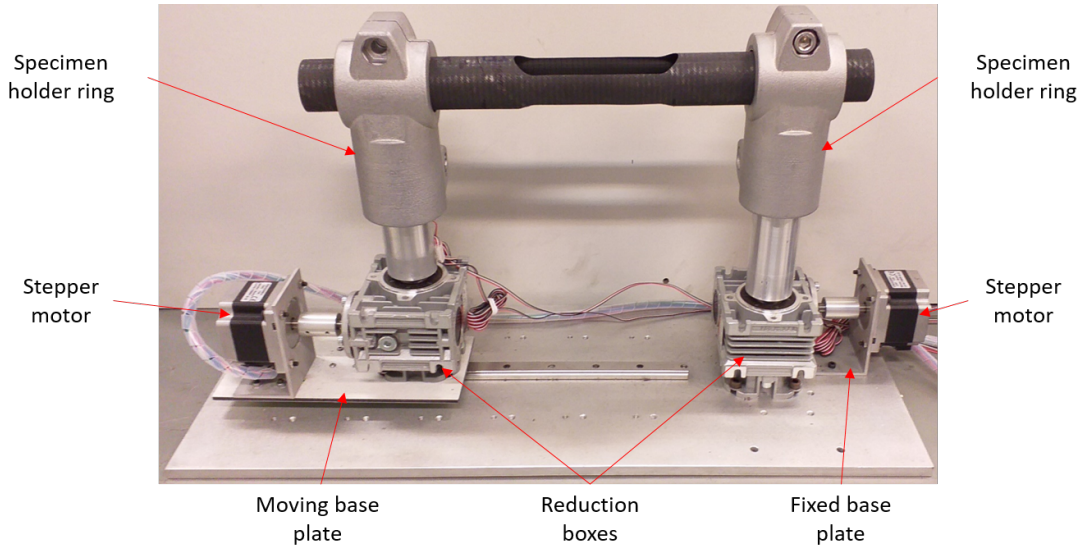

Figure 3.5: Test rig used to measure the torque as a function of the folding angle.

The torque measurement is made using a $\pm 45°$strain gauge attached to the shaft of the specimen holder ring. The strain gauge is connected to a signal amplifier HX711, that delivers the information to a User Interface using an Arduino based microcontroller. Before testing, each holder is calibrated. The calibration procedure requires fixing a rod, with a known length, to the holder and placing a calibrated weight on its end (figure 3.6). The torque is determined mathematically and the output from the electronic system is recorded. The next step is to place, in the same location, another calibrated weight, so the system output can be sensed and recorded. This step has to be repeated until the estimated torsion value is reached corresponds to that of the calibrated weights. Once all the data is collected, a simple linear, quadratic or higher-order regression has to be applied to fit the data and obtain the torque equation for this particular system.

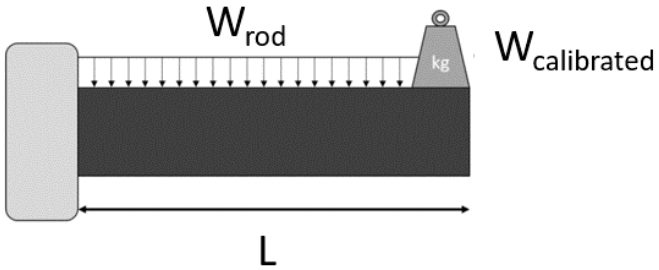

Figure 3.6: Calibration method using a rod and a calibrated weight.

Since the pinching process has to be applied manually, it was excluded from the numerical model and experimental procedure during the validation phase, to improve repeatability and remove the human stochastic component of the process.

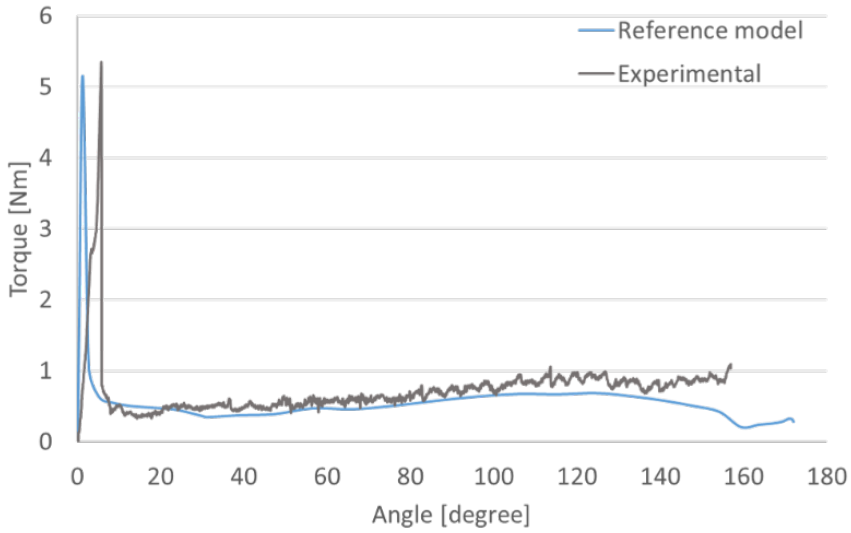

Figure 3.7: Representative curve of the torque applied to an elastic hinge as a function of the folding angle.

3.4.3 Output comparison

The experimental procedure was repeated for three valid experiments, with different test specimens. A representative curve of the torque-angle measurement is shown in figure 3.7 and compared to the numerical prediction, revealing a good agreement between both results. It is possible to observe some irregularities in the experimental curve, which do not exist in the numerical prediction. This difference is caused by the use of the two stepper motors, which do not allow a continuous movement, but rather a movement caused by a series of small increments. This factor, in conjunction with the existence of friction, causes some irregularities in the experimental torque-angle curve. The maximum torque values observed during the different experimental tests are reported in Table 3.2, indicating consistent results with an average deviation of 0.06 Nm.

Table 3.2: Maximum torque according to experimental and numerical results.

		Maximum torque (N.m)
Experimental results	Sample 1	5.25
	Sample 2	5.33
	Sample 3	5.16
	Average	5.25
	Average deviation	0.06
Numerical results		5.14
Average difference (numerical and experimental)		0.11

3.5 Design and optimization

This section describes the design and optimization process of the composite deployable arm, including: the design variables, objective function, selected optimization algorithms, internal parameters selected for each algorithm and an analysis of the obtained outputs. Section 3.5.1 describes the design variables used. This includes a description of the possible geometries that can be obtained, using the selected parametrization, and a definition of the design space. The description of the objective function (section 3.5.2) explains how the information obtained from the numerical models is utilized to explore and improve possible solutions. The section dedicated to the optimization process (section 3.5.3) explains how a genetic algorithm and a particle swam optimization method were combined. In this integration, the respective discrete and continuous natures of these methods were used to develop an optimization process divided into a global and local search. This enables a quick overview of the design space, followed by a detailed search in the neighbourhood of the most promising solution.

3.5.1 Design variables

The considered design variables define the geometry of the slot and the composite arm as a combination of geometrical figures. This parametrization, described in figure 3.8, is inspired in the work reported in [49] but allows a larger number of possible designs and the inclusion of asymmetries, as shown in figure 3.9.

In total, nine design variables were considered. These include the number of plies of the composite (P_n), ply angle (A) and the following geometric variables:

- R_i – Internal radius of the tube;
- R_1 and R_2 – Radius 1 and 2, defined as a percentage of R_i;
- R_{1r} and R_{2r} – Ratio between the slot's width and R_1 or R_2 on the centre of each circle (visually described by W_1 and W_2, where $W_n = R_n R_{nr}$, with $n = 1, 2$);
- S_L – Slot's length, defined as the product of a scalar and R_i;

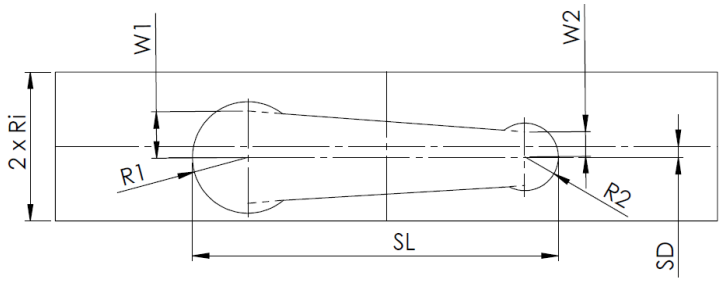

Figure 3.8: Visual representation of the design variables that define the geometry of the slot of the elastic hinge.

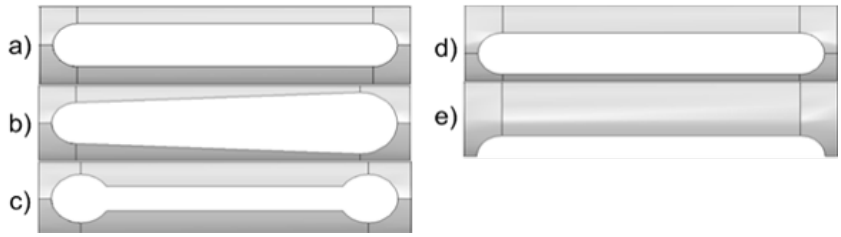

Figure 3.9: Examples of possible geometries considered by the parametrization used.

- S_D – Slot's displacement from the longitudinal axis of the tube. Defined as a percentage of the largest value between R_1 and R_2;

Each variable has the minimum and maximum values shown in Table 3.3. These variables were mainly defined as percentages and ratios to ensure that the algorithm cannot generate impossible solutions by combining incompatible design values. An example of an impossible solution would be a slot with R_1 larger than R_i. Avoiding these unrealistic cases leads to a smaller design space and a higher probability of success in finding the global optimum.

3.5.2 Objective function

The design process of the composite deployable arm considers geometrical, damage and frequency requirements. The geometrical requirements, such as the diameter, which cannot exceed 200.0 mm, can be addressed by restraining the domain of the design variable(s), as described in section 3.5.1. The frequency requirement can be seen as another constraint, where solutions whose natural frequency is lower than 1.0 Hz are rejected. These two considerations leave the minimization of the installed damage as the only objective of the optimization process and avoid the use of multi-objective optimization processes where the complexity is far greater. With this strategy in mind, the objective function was defined as the minimization of the damage, where the damage installed in each possible design was estimated using the structural numerical model described in section 3.3.2. However, executing a structural analysis on every possible solution would lead to an exceedingly large computational cost. To avoid this situation, each candidate solution had its natural frequency estimated through the numerical model described in section 3.3.1. If the natural frequency of a possible solution was larger than 1.1 Hz and lower than 1.25 Hz, it would be

Table 3.3: Maximum and minimum values of each design variable

Variable	Minimum	Maximum	Unit
R_i	20	100	mm
R_1,R_2	15	80	%
R_{1r},R_{2r}	0.1	1.0	-
S_L	3	10	-
S_D	0	100	%
P_n	1	4	pair(s)
A	0	90	°

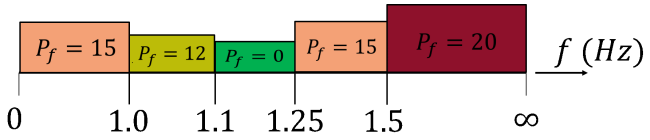

Figure 3.10: Graphic representation of the penalty factor as a function of the natural frequency of the design considered.

evaluated through structural analysis and its performance value ($Perf$) set equal to the maximum index resulting from the application of the failure criteria included in the numerical model (listed in section 3.2). If the natural frequency was not within this range, the performance value was set equal to a penalty factor (P_f) and the structural analysis skipped. Equation (3.1) shows the formula used to determine the performance value of a solution, while figure 3.10 shows a graphic representation of the penalty factor as a function of the natural frequency.

$$\begin{cases} Perf = 15, & \text{if } 0.0 \leq f < 1.0 \quad \vee \quad 1.25 < f < 1.5 \\ Perf = 12, & \text{if } 1.0 \leq f \leq 1.1 \\ Perf = Max.\ IF, & \text{if } 1.1 \leq f \leq 1.25 \\ Perf = 20, & \text{if } 1.5 < f < +\infty. \end{cases} \tag{3.1}$$

Where $Max.\ IF$ is the largest index of failure of the failure criteria included in the structural analysis (listed in Section 3.2).

Solutions whose natural frequency was lower than 1.1 Hz received a penalty factor for not meeting the frequency requirement. This penalty was more severe for solutions whose natural frequency was lower than 1.0 Hz, since they were further away from the minimum requirement. The penalty applied to solutions with a natural frequency above 1.25 Hz is justified by the proportionality between the stiffness of a structure and its natural frequency. The increase in natural frequency implicates an increase in stiffness, making the structure less flexible and less likely to fold without initiating damage. Even more severe penalties were applied to solutions with a frequency larger than 1.5 Hz, for the same reason.

3.5.3 Optimization process

The optimization process considered two different optimization algorithms: a genetic algorithm, inspired by the theory of natural selection [294, 295] and a particle swarm optimization method, a heuristic search method inspired by the collaborative behaviour of biological populations [296]. Considering the classic implementation of these algorithms, it is possible to classify the GA as naturally discrete and the PSO as naturally continuous. This naturally implies that a classic GA is more suitable to optimize discrete variables and that a classic PSO is more suitable to optimize continuous variables. Given these characteristics, the optimization process was divided into a global and a local search.

The GA was applied during the global search, where only discrete values of the 9 design variables were considered. This decision led to a reduction in the number of possible solutions that the optimization algorithm had to explore. Once a solution was found, the local search was initiated. The local search consisted of applying the PSO method to search for an optimal solution in the neighbourhood of the solution found during the global search. The local search domain was defined by a ±10% variation of each design variable, except for the number of plies, which was not included in the local search. This sequence of operations allows for a quick survey of the design space, followed by a more refined search near the optimum found. Convergence was assumed after 30 iterations with no improvement of the elite group, for the GA, or the best solution found, for the PSO.

A detailed description of the GA and PSO methods implemented, as well as the internal parameters chosen, can be found in Appendix A and Appendix B, respectively.

3.6 Results and discussion

Through multiple optimization attempts, it was observed that the best solutions found in each attempt usually has a large slot radiae (R_1, R_2) and smaller widths ($R_1 r, R_2 r$). Cases a), b) and c), shown in figure 3.11, are examples of this scenario. However, these solutions did not meet the design requirements due to their Max. IF being 2.01, 2.04 and 2.05, respectively. Further evaluation of the geometries generated by the optimization algorithm led to the discovery of case d), which is also representative of this situation and corresponds to the solution with a lower index of failure (1.86), but did not meet the natural frequency requirement (0.57 Hz). The best result obtained through the optimization process is shown in figure 3.11 e) and has a Max. IF of 1.63, according to Hashin's failure criteria for matrix under compression, and a natural frequency of 1.22 Hz. However, despite having a natural frequency higher than 1.1 Hz, this solution fails to meet the damage requirements. The indexes of failure of this solution are shown in Table 3.4, while the design variables that define the geometry are shown in Table 3.5.

Furthermore, by analyzing the outputs of the structural models, it was possible to observe two main stress concentration regions, highlighted in red and orange in figure 3.12. It was also observed

Table 3.4: Indexes of failure observed according to each failure criterion.

Criteria		Value
Azzi-Tsai-Hill		1.24
Hashin	Fibre compression	1.04
	Fibre tension	0.31
	Matrix compression	1.63
	Matrix tension	1.48
Maximum stress		1.22
Tsai Hill		1.25
Tsai Wu		1.40
Max. IF		**1.63**

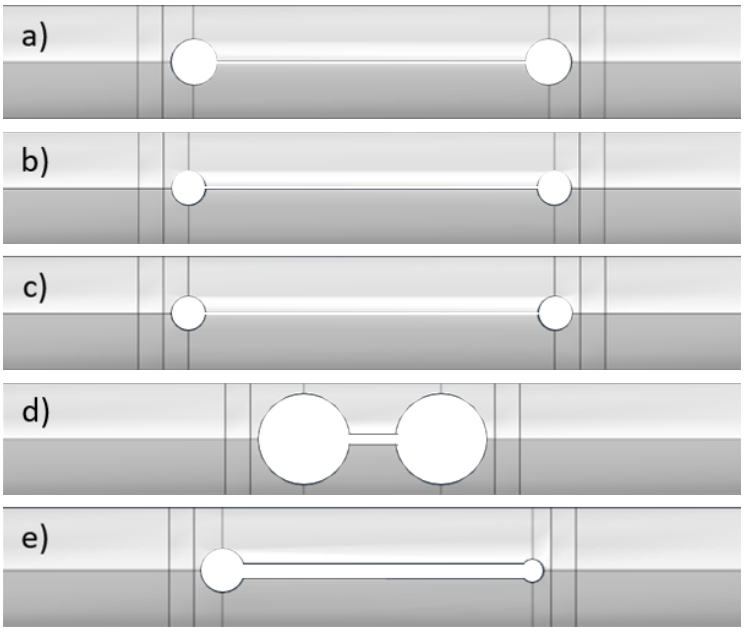

Figure 3.11: Best solutions found through the optimization processes. Cases b) and c) differ slightly in the slot's width. Case d) has a lower index of failure (1.86) but does not meet the frequency requirement (0.57 Hz). Case e) is the solution with the lowest index of failure (1.63) that meets the frequency requirements (1.22 Hz).

Table 3.5: Variables defining the slot geometry used to compare the influence of the boundary conditions and the removal of material. (*) Rounded value in mm, after obtaining the product between the percentage and R_i.

Variable	**Value**	**Unit**
R_i	85.0	mm
R_1	35	%
R_2	15	%
$R_1 r$	0.35	-
$R_2 r$	0.65	-
S_L	480*	mm
S_D	4.0*	mm
P_n	2	pair(s)
A	40	°

that the amount of damaged material (red region) would decrease as the radius of the slot increased. The location of the stress concentration points, as well as the type of geometry found during the optimization process, is in agreement with the analysis reported in [49, 48, 51], which found similar designs through a parametric analysis.

Additionally, it is noticeable that the solution with the lowest index of failure (case e) has an asymmetric design, which results in a stiffer region located on one end of the slot. This solution implies a direct relationship between the boundary conditions, that define the folding process, and the installed damage initiation in the component. Therefore, the boundary conditions implemented in the folding process need to be revised for their effect in the damage initiation to be minimized.

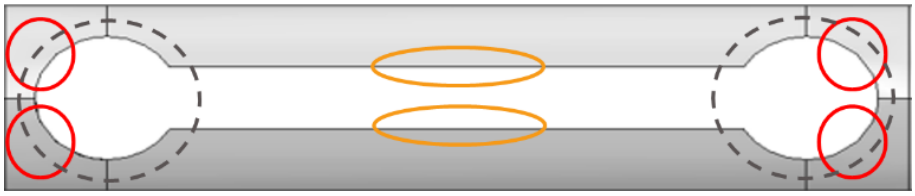

Figure 3.12: Highlight of the stress concentration zones. The regions marked in red have the largest failure indexes, followed by the orange region.

Based on these observations, it can be theorized that further improvements on the design, at the damage and frequency levels, of the elastic hinge should be achieved by implementing two modifications. The first, concerning the boundary conditions. The loads installed on the edges of the tapes of the elastic hinge are the result of the rotations applied to each holder. Therefore, as an alternative, the folding process should consider the vertical displacement of the lowest tape, resulting in a reduction of compressive loads. The second modification involves the design variables for the optimization process of the slot geometry to consider the possible removal of material from the stress concentration points highlighted in figure 3.12.

3.7 Conclusion

This research consisted of the integration of finite element numerical models, used to estimate the structural performance and natural frequency of a deployable structure, in an optimization process that involved a genetic algorithm, for a global search followed by a particle swarm optimization method, for a local, more refined, search. Through this research, the following conclusions are summarized:

- The numerical results obtained from the structural analysis of the elastic hinge were in good agreement with the experimental results.

- Analysis of the results obtained led to the identification of limitations of the design variables commonly used in the literature to design this type of structure. These variables do not include geometries that may minimize the stress installed in the deployable structure.

- A geometry capable of reaching a natural frequency of 1.22 Hz and a maximum index of failure of 1.63 was found. This solutions meets the natural frequency requirement but initiates damage during operation, and therefore is rejected.
- It is observed that the folding process directly affects the damage installed. This is the result of a non-optimized folding process, which translates into large compressive loads observed in the elastic hinge. An alternative folding process should allow the reduction of damage installed and further enable the use of this type of methodologies.
- Future work should look towards including geometrical modifications to the stress hotspot areas in the algorithm. This can be achieved through the redefinition of the design variables or, potentially, the application of a different optimization method, such as topology optimization. Doing so will allow the computational process to have new alternatives to test and provide a better assessment of the slot geometry as well as further optimize the damage onset and frequency, fulfilling the restrictions imposed by the design requirements.

Chapter 4

Influence of relaxation on the deployment behaviour of a CFRP composite elastic hinge

The present chapter is based on the following refereed publication:

P. Fernandes, B. Sousa, R. Marques, João Manuel R.S. Tavares, A.T. Marques, R.M. Natal Jorge, R. Pinto, N. Correia. (2021) *Influence of relaxation on the deployment behaviour of a CFRP composite elastic-hinge*. Composite Structures, 259, 113217. doi.org/10.1016/j.compstruct.2020.113217.

4.1 Introduction

The tape-spring concept is most notable for its application as a support of the monopole and dipole antennas of MARSIS, made of thin-walled S-Glass and Kevlar composite [5]. Regardless of the success of the mission, which led to the discovery of liquid water on Mars in 2018 [297], the hinges had an extremely low deployment moment (0.2 Nm), which translated to added challenges in the experimental testing on earth. The structure was qualified for launching based solely on simulations, component testing, and the experimental deployment of the structure using a helicopter (figure 2.2), reporting "no significant damage to the full flight-representative hardware" [5]. Once in orbit, these difficulties related to the experimental validation of the design contributed to a half-deployed configuration that only completed its deployment after re-orienting the antenna towards the sun, causing the thermal expansion of the tape-springs and the complete opening of the elastic hinge [298, 38]. Due to the difficulties found during its development and operation, MARSIS marked the use and application of deployable structures, leading to more focused research on the characterization and numerical modelling of deployment mechanisms.

Several authors have used high-speed cameras to study the release of the tape spring, by measuring the angle formed by the tape-spring, or by the deployable structure, as a function of time (figure 4.1) [18, 14, 10, 30, 37]. The main advantage reported was the possibility of measuring any over-shooting angle that may occur. In this context, over-shooting angle refers to the angle formed by the tape-spring due to buckling caused by compressive loads that arise during the deployment process.

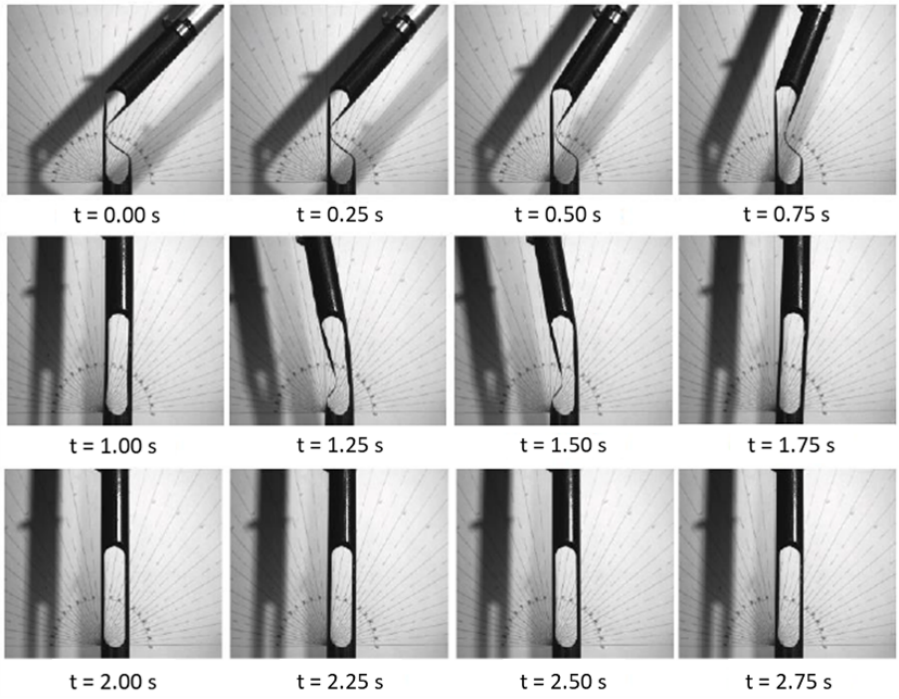

Figure 4.1: Deployment of an elastic hinge. Overshooting occurs between t=1.0 s and t=1.75 s (adapted from [18]).

In 2009, Mobrem and Adams [38] studied several factors that affected the deployment of the MARSIS antenna. The research included the evaluation of the hinge buckling strength in a four-point bending fixture, measurement of the torque-angle curve at room temperature and at -70 °C, the evaluation of the stored energy through the deployment of the elastic hinge in a vacuum chamber at -70 °C, natural frequency analysis, and a vertical pendulum test to assess the dynamic buckling. The experimental data obtained was used to validate numerical models implemented in both ABAQUS® [11] and ADAMS [299]. The authors observed that the thermal environment and aging severely affected the properties of the elastic hinge, decreasing strength properties, reducing the deployment torque, and enabling buckling phenomenon in locations adjacent to the hinge section. Kwok and Pellegrino (2010) [39] investigated the shape recovery behaviour of a beam and a tape-spring made of low-density polyethylene (LDPE), at room temperature. The authors obtained the linear viscoelastic material properties of LDPE through creep tests and analysed the LDPE beam under four-point bending, imposing a history of vertical deflection and reaction forces. The numerical model proposed was capable of capturing the behaviour of the material, which recovered its deployed shape regardless of the bending direction of the tape-spring. However, in 2011, further research performed by the same authors [40] reported that the stowage of the tape-spring for extended periods could result in a reduction of 60 % in the load resultant from the folding of the tape-spring due to the relaxation phenomenon. In 2013, Kwok and Pellegrino [43] further detailed this research with the inclusion of a relaxation modulus master curve in the numerical model, capturing the effect of the compaction rate and temperature in the non-linear load-displacement response during retraction, load relaxation over long stowage durations, short-term deployment, and long-term shape recovery. In 2013, Brinkmeyer *et al.* [41] developed similar research, studying the effect of viscoelasticity on the deployment of a "coilable" bistable tape-spring made of ultra-thin CFRP. A viscoelastic analytical model was used to predict the relaxation of the structure when contracted. Similar to the previous results, the authors observed that the stowage caused an increase in the deployment time. This phenomenon was further aggravated with the influence of high temperatures, reporting that the structure would not deploy after being stowed at 100 °C. The analytical model used was capable of predicting the deployment behaviour of the structure for short storage periods. More recently, Brinkmeyer *et al.* (2016) [44] enhanced their research, including the master curve approach proposed by Kwok and Pellegrino [43] in 2013. Through this work, Brinkmeyer *et al.* validated the use of this approach for ultra-thin CFRP materials and were able to capture the long-term stowage time effect that was not accounted for in the initially proposed analytical model. In 2017, Khan *et al.* [300] studied the energy dissipation of a composite deployable tape-spring due to the viscoelasticity of the matrix. The authors used a finite element model to simulate the stress-relaxation response of a CFRP laminate. After 6 months of relaxation, the author observed a 22 % reduction of the original modulus.

In this research, the mechanical properties of a space certified carbon/epoxy composite system were determined, and its relaxation behaviour was assessed experimentally. The experimental data obtained was used to determine the relaxation master curve of the material that was introduced in the finite element model code to simulate the deployment behaviour of a deployable elastic

hinge before and after material relaxation due to the stowage effect. The numerical model was experimentally validated using the deployment behaviour of elastic hinges before and after the material relaxation analysis.

4.2 Experimental work

The following sub-sections describe the experimental component of this investigation, including: the material characterization experimental campaign, manufacturing details of both specimens and representative prototypes, processing of the experimental data, and description of the obtained experimental results. The accomplished test campaign served two main purposes. The first was to characterize the material studied in this research, regarding both mechanical and relaxation properties. The second was to obtain experimental data that could be correlated with the numerical results, allowing the validation of the numerical models. Unless stated otherwise, all experiments included a minimum of 3 valid test results. The data necessary to reproduce the work mentioned in Section 4.2.2 is available in [301].

4.2.1 Material characterization

A space certified composite prepreg system made of carbon fibre and a high-performance tough epoxy matrix was used (AS4/8552 provided by HEXCEL Composites®, Madrid) [1]. Tensile, compressive, and shear tests were conducted according to the applicable ASTM standards [302, 303] to determine the elastic and strength properties of the composite system. The specimens were cut from CFRP plates manufactured by prepreg hand lay-up and cured in an autoclave according to the supplier recommendations: 1 hour at 110 °C followed by 2 hours at 180 °C, applying a 7 bar pressure (0.7 MPa) through the whole process. These conditions resulted in an average ply thickness of 0.18 mm. The characterization was performed in an Instron 5900R universal testing machine, under the conditions summarized in Table 4.1.

Table 4.1: Summary of the load cell, displacement control and data acquisition rate used for each test. In the present document the "traction" term is adopted to define a positive stress, in the same way that "compression" is used in reference to a compressive stress.

Test	**Ply orientation (º)**	**Load cell (kN)**	**Displacement control (mm/min)**	**Data acquisition rate (Hz)**
Traction	0	200	1.0	2.0
	90	200	1.0	5.0
Compression	0	100	1.3	5.0
	90	100	1.3	5.0
Shear	45	30	1.0	1.0

[1] The material batch used in Chapter 3 is different than the one used in Chapter 4 onwards, justifying the difference in the material properties reported.

Table 4.2: Elastic and strength properties of AS4/8552.

Elastic properties	**Average value**	**Standard deviation**	**Unit**
E_{11}	122.84	±4.33	GPa
E_{22},E_{33}	8.04	±0.21	GPa
G_{12},G_{13}	4.90	±0.08	GPa
v_{12}	0.29	±0.03	-
Strength properties			
X_t	1987.15	±53.26	MPa
Y_t,Z_t	51.83	±1.25	MPa
S_{12},S_{13}	139.05	±0.72	MPa
X_c	963.03	±43.77	MPa
Y_c,Z_c	258.13	±18.24	MPa

A digital image correlation (DIC) system was used to measure the strain during testing. The measurement was performed considering a 2D deformation (2D-DIC), tracking the displacement of scattered dots painted in the surface of the specimens. The image processing was done with the commercial software DIC Replay, provided by INSTRON, using default settings. The mechanical properties determined are indicated in Table 4.2, where E is the Young's Modulus, η the Poisson's ratio, G the shear modulus, X, Y and Z are the longitudinal and transverse strengths, S_L the shear strength, the subscripts 1, 2 and 3 indicate the longitudinal, transverse, and normal directions, and the subscripts t and c denote the tensile or compressive strength.

The remaining material resulting from the cut of the CFRP plates was used to determine the density [304] and fibre volume fraction (FVF) [304, 305] of the composite according to ASTM standards (following procedure G – matrix burn off of ASTM D3171 [305] to determine the FVF). The average values and standard deviations for the density and FVF determined are, respectively: 1580.76 ± 2.38 kg/m^3 and 58.37 ± 0.47 %. To assess the relaxation of the material, specimens were manufactured according to the geometric specifications detailed in [306] (25 mm wide, 150 mm long, thickness within 1 mm and 1.5 mm, and plies oriented at $\pm$ 45°). The relaxation tests were performed using a load cell of 30 kN and at three different temperatures: room temperature (RT), 50 °C and 80 °C. These temperatures were selected based on the maximum temperature achievable by the testing chamber, and defining a constant increment of 30 °C between each test. Each specimen was loaded at a rate of 1 mm/min until a load of 2.6 kN, this represented a stress value of 35 MPa. This load was selected for being within the elastic regime of the material, whose yield stress is approximately 40 MPa measured in a preliminary tensile test. The displacement necessary to initiate the initial load of 2.6 kN was maintained for 3 hours, during which the load decay due to the relaxation of the material was recorded. Figure 4.2 shows the relaxation of the material at the different tested temperatures, reporting a load reduction of 13.27 %, 14.79 %, and 20.31 % at room temperature, 50 °C, and 80 °C, respectively.

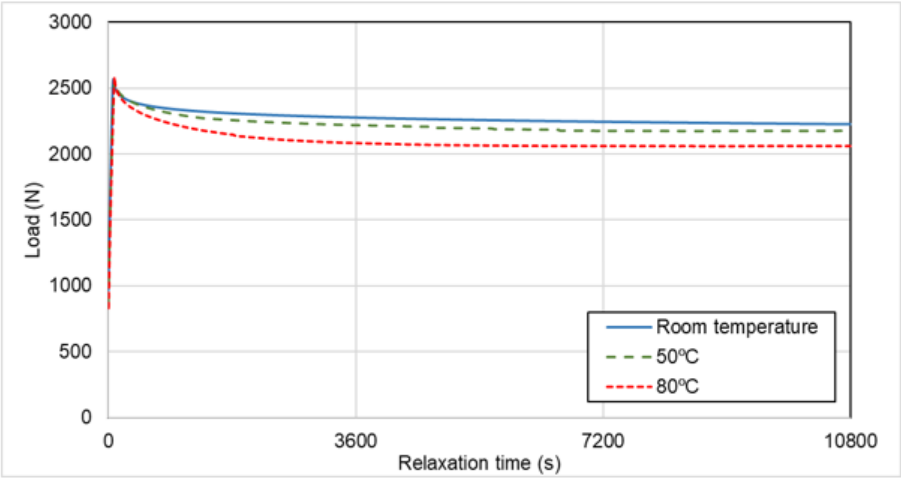

Figure 4.2: Relaxation curves of AS4/8552 at room temperature, 50 °C and 80 °C.

4.2.2 Relaxation master curve

The adopted test procedure is based on the shear loading, due to the ply orientation of the laminate, of a composite sample. Therefore, the load decay observed is the result of the relaxation of the shear modulus G_{12} of the composite material, which includes the contribution of both fibre and matrix. According to the rule of mixtures [307], the shear modulus of the composite material can be determined according to:

$$G_{12} = G_m \left[\frac{(1+V_f) + \frac{G_m(1-V_m)}{G_f}}{(1-V_f) + \frac{G_m(1+V_m)}{G_f}} \right]. \tag{4.1}$$

where G and V represent the shear modulus and volume fraction of either fibre or matrix, discriminated by the subscript f and m, respectively. In cases where G_f is much larger than G_m ($G_f \gg G_m$), equation (4.1):

$$G_{12} = G_m \left(\frac{1+V_f}{1-V_f} \right). \tag{4.2}$$

In a uniaxial stress-relaxation test, the relaxation modulus is defined as:

$$E(t) = \frac{\sigma(t)}{\varepsilon_0} \tag{4.3}$$

where $\sigma(t)$ is the stress installed as a function of time and ε_0 is the constant strain applied during the relaxation test. Additionally, considering an isotropic material and assuming that the Poisson's coefficient of a viscoelastic material is constant in time, it is possible to correlate the shear modulus and Young's modulus using:

$$E(t) = 2G(t)(1+\nu). \tag{4.4}$$

Taking into account equations (4.2) and (4.4), it is possible to establish a direct relationship between the load decay observed in the relaxation test and the decay in both G_m and E_m, allowing the determination of the curves shown in figure 4.3. Furthermore, the material that corresponds to the fibre reinforcements in a composite material is less prone to suffer relaxation than the matrix [307, 308]. Considering the relaxation time of 3 hours used in the experimental tests, it is safe to assume that the relaxation observed is caused by the matrix and that the relaxation of the fibre is negligible [43, 308, 309].

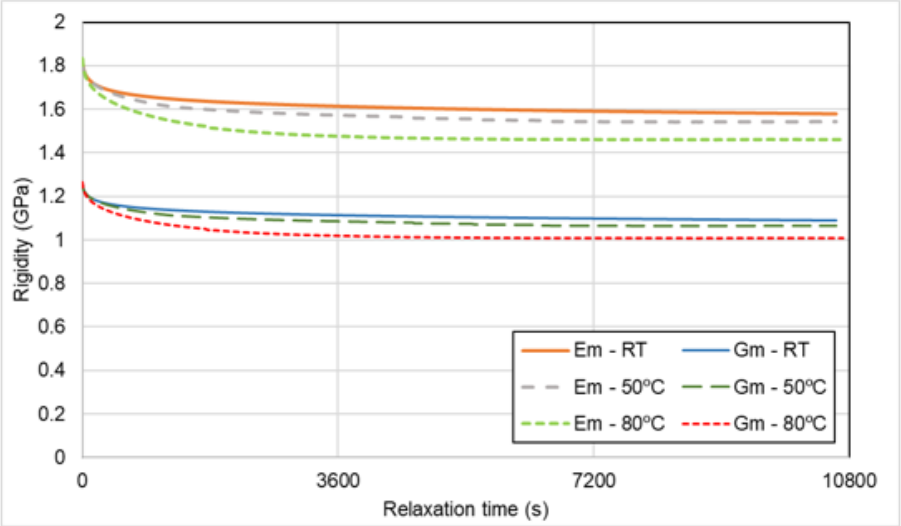

Figure 4.3: Relaxation of the matrix shear and Young's modulus at each temperature.

According to the time-temperature superposition principle (TTS) [308, 310–313], the viscoelastic behaviour of a polymeric material at two different temperatures, reference temperature T_0 and another temperature T, can be related using the change of the experimental time scale:

$$E(t, T_0) = E(\alpha_T t, T) \tag{4.5}$$

where E and t are the relaxation modulus and relaxation time, respectively. This equation means that the relaxation modulus determined at time t and temperature T_0 is equivalent to the relaxation modulus determined at time $\alpha_T t$ and temperature T. The explanation behind this equation is that, at low temperatures, the relaxation process requires a longer time for experimental observation, while the opposite occurs at high temperatures. Therefore, the TTS assumes an equivalent exchange between the testing temperature and the relaxation time, allowing the building of a master curve that extends the relaxation modulus beyond the range of the testing time scale, at a given temperature. Considering the TTS and the room temperature as the reference temperature, a master curve was built by shifting the relaxation curves determined at 50 °C and 80 °C to the right, along the time axis (increased time). The applied horizontal shift factors are a function of the temperature and can be fitted using the Williams-Landel-Ferry equation (WLF) [310, 313]. The resulting master curves are shown in figure 4.4. It is observable that the TTS allowed the prediction of the relaxation

observed at room temperature after, approximately, 24 hours (105 s) based on the data gathered during 3 hours of experimental testing at three distinct temperatures (RT, 50 and 80 °C).

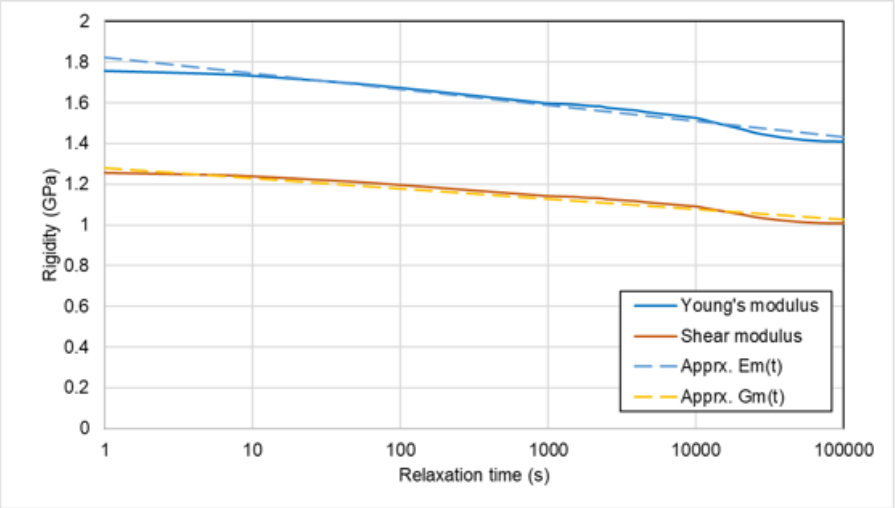

Figure 4.4: Relaxation of the matrix shear and Young's modulus at each temperature.

The obtained master curves show an approximately linear relaxation in a logarithmic scale, commonly observed in the relaxation of several materials [314–317]. Both Young's modulus and shear modulus relaxation master curves can be approximated applying the following logarithmic equations (also represented in figure 4.4), respectively:

$$E_m(t) \approx -0.034 \ln(t) + 1.8233 \tag{4.6}$$

$$G_m(t) \approx -0.022 \ln(t) + 1.2797. \tag{4.7}$$

These approximations capture the linear relaxation observed between relaxation times between 10 and 10^4 s. The increase in slope of each experimental curve, observed at 105 s, was not included in this approximation, allowing for a more conservative estimate of both shear and Young's modulus when extrapolating the experimental results to larger relaxation times (further detailed in sections 4.3.1 and 4.3.2). The data necessary to reproduce this section is available in [301].

4.2.3 Elastic hinge manufacturing and deployment

Elastic hinge specimens were manufactured through the hand lay-up of two plies of AS4/8552 prepreg, oriented at ±45°, placed on an aluminium mandrel with an external diameter of 100 mm. The elastic hinge specimens were cured in the same conditions as the samples used to determine the mechanical properties of the material. After the curing process was completed, each specimen was machined in order to obtain the final geometry of the elastic hinge, depicted in figure 4.5. This geometry was selected based on the result of a finite element model (detailed in section 4.3.1), developed and validated in previous research [51, 86], ensuring that no damage would initiate

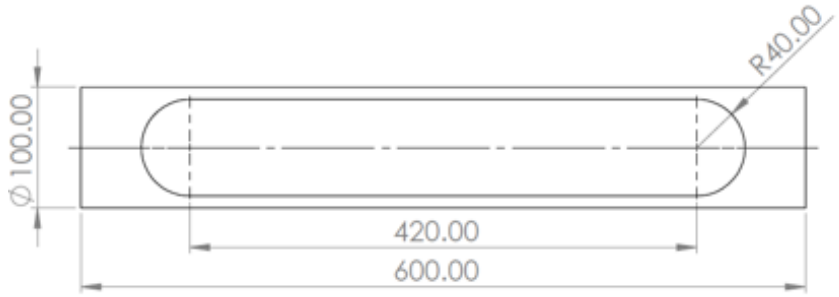

Figure 4.5: Geometry of the elastic hinge specimen.

Table 4.3: Maximum indexes of failure observed during the retraction of the elastic hinge specimen.

Failure criterion	**Maximum index of failure**
Maximum stress	0.80
Tsai-Wu	0.77
Tsai-Hill	0.80
Azzi-Tsai	0.80
Hashin (matrix compression)	0.20
Hashin (matrix tension)	0.65
Hashin (fibre compression)	0.01
Hashin (fibre tension	0.01

during the retraction of the elastic hinge. Therefore, the results obtained from the experimental testing were not influenced by the initiation of damage. Table 4.3 reports the maximum index of failure, for several damage failure criteria, observed during the retraction of the elastic hinge specimen according to the numerical model.

The deployment of each elastic hinge specimen was recorded with a camera at a rate of 120 frames per second. One end of each specimen was attached to a mandrel, fixed with plastic clamps and placed in the vertical position in front of a scale, with incremental angle measurements every 5°, as shown in figure 4.6. To allow a clearer measurement of the angle, the tape-spring seen on the left side of the elastic hinge (shown in figure 4.6) was aligned with the vertical axis of the scale, representing the 90° angle, and the folding of the elastic hinge was done to the right side in a clock-wise motion. As a result, the deployment angle is equal to the angle formed by the left tape in relation to the vertical axis.

The specimens were tested immediately after the manufacturing process. A typical set of photographs from the deployment tests is shown in figure 4.7. As can be observed, the specimen reaches a deployed configuration after, approximately, 0.4 s and no over-shooting occurs. From this point onwards, only a small oscillation is observed.

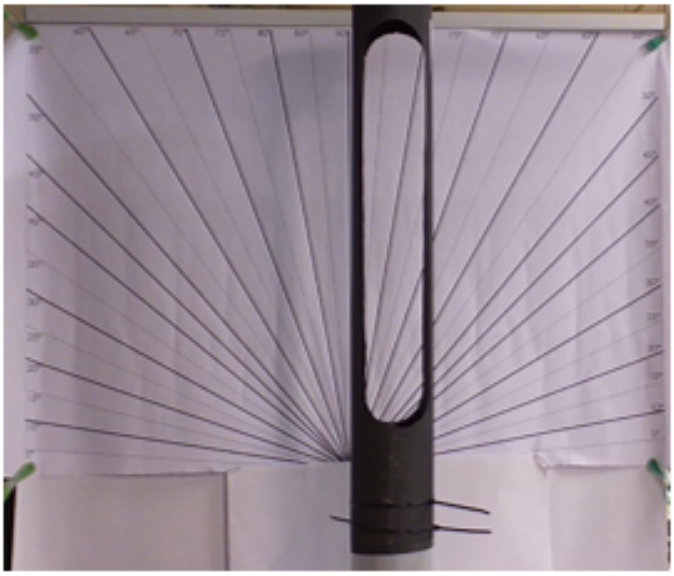

Figure 4.6: Representation of the setup used to evaluate the deployment of the elastic hinge.

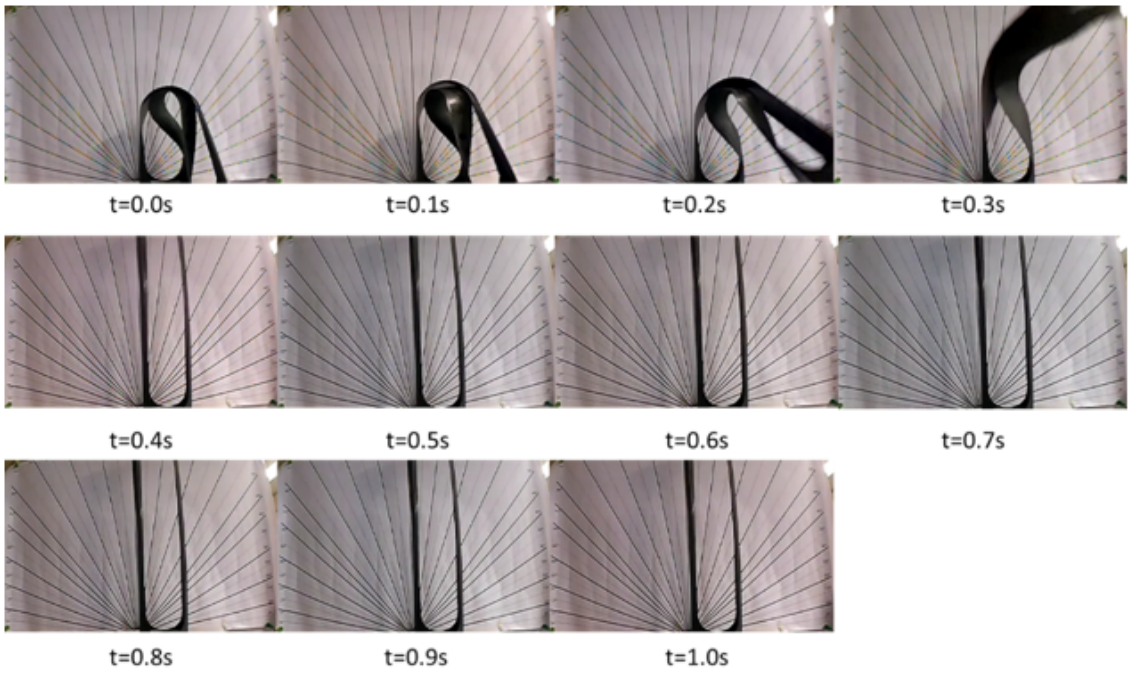

Figure 4.7: Photographs of a typical deployment of an unaged elastic hinge specimen.

4.3 Numerical analysis

4.3.1 Numerical modelling of deployable structures

The numerical modelling procedure described in this section is similar to the one described in section 3.3. The main differences focus on the inclusion of the deployment step, and an adaptation of the folding sequence, changing it from rotation to a displacement-driven movement. These details and differences are listed as follows.

The finite element models used in this research were implemented in ABAQUS® [11], using an explicit approach and a non-linear geometry. These consider the retraction, relaxation and deployment of the elastic hinge. The composite material was modelled with two-dimensional deformable shell elements with 4 nodes and reduced integration (S4R in ABAQUS® [11]). The geometry of the elastic hinge includes two different sections, highlighted in blue and grey in figure 4.8. In the central region, where the tape-springs of the elastic hinge are located, the model considers the existence of a composite layup feature with multiple plies, allowing a detailed analysis of the stress and strain state of each element. In this region, the integrity of the material is addressed using several failure criteria, including: Hashin's, Azzi-Tsai-Hill, Tsai-Hill, Tsai-Wu, and Maximum stress. The region highlighted with a grey colour does not sustain significant stresses or strains and is only used to apply representative boundary conditions and capture the rigidity of the component. Therefore, the model assumes a single layer of elements with homogenized properties in the material elastic regime. The mesh applied to both regions has an approximate size of 2.5 mm.

Figure 4.8: Representation of the mesh and of the different regions included in the numerical model.

The sequence of retraction, relaxation and deployment of the elastic hinge was modelled considering three different steps. The elastic hinge is folded by applying a 90° rotation on each end of the tubular section along with the vertical displacement of a rigid plate, located below the lower tape of the specimen, at its mid-length (figure 4.9). The vertical displacement of the rigid plate was set equal to 280.0 mm, 20.0 mm less than half the length of the specimen to avoid tensioning of the tape-springs. When compared to an equivalent process, without the rigid plate, the inclusion of this component prevents the lower tape-spring from buckling due to the rotation applied on each end of the part, minimizes the stresses installed during the folding process and promotes a more repeatable folding sequence.

Applying this sequence of operations forces the elastic hinge specimen to transition towards the folded configuration represented in figure 4.10.

The material properties were defined as a function of a fictitious temperature to simulate the relaxation of the material; this is a procedure that is built into the ABAQUS® code [11]. This

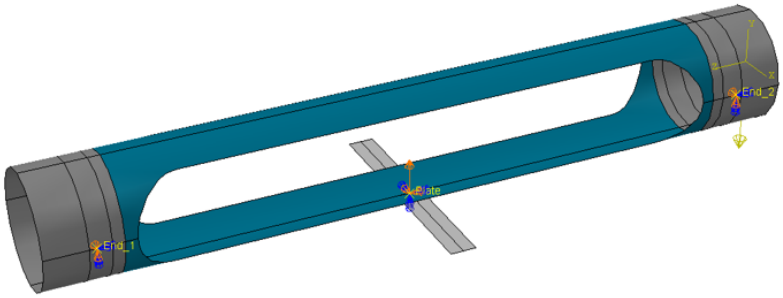

Figure 4.9: Representation of the boundary conditions applied to the elastic hinge tube specimen and to the rigid plate component.

fictitious temperature does not represent the physical temperature of the specimen and its change does not result in geometrical variations resulting from the thermal-expansion phenomenon or any other change observed in nature. Instead, the change of this fictitious temperature causes the numerical model to update the material properties of the composite from an unrelaxed to a relaxed state, equivalent to the relaxation observed due to an extended period of stowage. The properties of the relaxed material were estimated considering the matrix relaxation described by the shear and Young's modulus master curves determined in section 4.2.2 (equations (4.6) and (4.7)), and assuming that the fibre properties are not affected by relaxation and the fibre volume fraction is constant. Considering the rule of mixtures and the relation between shear modulus, Young's modulus and the Poisson's coefficient in the different laminate directions [307], it was possible to reach the values indicated in Table 4.4.

It is important to note that introducing the material properties listed in Table 4.4 assumes an equal stress-strain state between the elastic hinge specimen, when folded, and the composite specimen used in the relaxation experiments, when loaded. In reality, these two stress-strain states are different. According to the maximum stress failure criterion, shown in Table 4.3, the elastic hinge specimen is loaded at 80 % of its maximum strength, while the composite specimen used in the relaxation experiments was loaded at 87.5 % of its strength (35 MPa out of, approximately, 40 MPa). Therefore, the properties listed in Table 4.4 assume a more critical stress-strain state, which will lead to a more conservative prediction of the deployment behaviour of the elastic hinge specimen after relaxation. Finally, after updating the material properties, the boundary condition

Table 4.4: Estimated material properties after 1, 6, 12, 18 and 24 months of relaxation.

Fictitious temperature	**Relaxation time (months)**	E_1 (GPa)	E_2,E_3 (MPa)	ν_{12},ν_{13} (-)	ν_{23} (-)	G_{12},G_{13} (MPa)	G_{23} (MPa)
0	0	122.84	8040.00	0.29	0.43	4900.00	2815.21
1	1	122.46	4133.17	0.28	0.68	1925.63	1960.50
2	6	122.44	3963.66	0.28	0.68	1839.36	1912.66
3	12	122.43	3896.47	0.28	0.68	1805.35	1893.30
4	18	122.42	3859.96	0.28	0.68	1786.91	1882.68
5	24	122.42	3831.82	0.28	0.68	1772.72	1874.45

Figure 4.10: Representation of the elastic hinge specimen in a folded configuration.

applied to one of the ends of the elastic hinge specimen was removed. As a result, the elastic hinge is allowed to deploy by releasing the stored elastic strain energy. During the analysis process, the simulation takes into account the influence of the force of gravity (represented in figure 4.9 by the yellow vector below the coordinate system). While this factor does not exist in operation, it was included in the numerical model to allow a better correlation with the experimental tests.

The finite element simulation described in this section required an average CPU time of three hours, considering the use of double-precision, four CPUs (Intel® Core(TM) i7-3820 @ 3.60 GHz) at full capacity, and a mass scaling factor of 4.0×10^{-7}. The mass scaling factor was chosen through an iterative process, ensuring that the resulting percentage mass increase per element was lower than 0.5%.

4.3.2 Correlation with experimental data

To validate the numerical model, a finite element analysis was conducted with the relaxation step suppressed. The objective was to allow a direct comparison between the gathered experimental test data (detailed in section 4.2.3) and the numerical model prediction for the deployment behaviour of the elastic hinge specimen. The deployment sequence predicted by the numerical model is shown in figure 4.11, overlapping a set of photographs of a typical experimental deployment test. To better compare the numerical and experimental data, each snapshot image was processed to accurately measure the angle formed by the elastic hinge specimen as a function of time. Figure 4.12 compares the numerically predicted deployment angle vs the average time of the experimentally collected data.

Comparing the results, it is possible to observe that the experimental deployment is slightly lagging behind the numerical prediction at the early stages of the deployment process. One possible factor contributing to this difference is the existence of air friction during the experimental tests, which is not included in the numerical model. The existence of air friction also explains the quicker damping observed in the experimental deployment tests when compared to the numerical result. Another difference between numerical and experimental results can be seen at the time instant of 0.3 s (figure 4.12), where the deployment angle of the experimental test is higher than the numerical

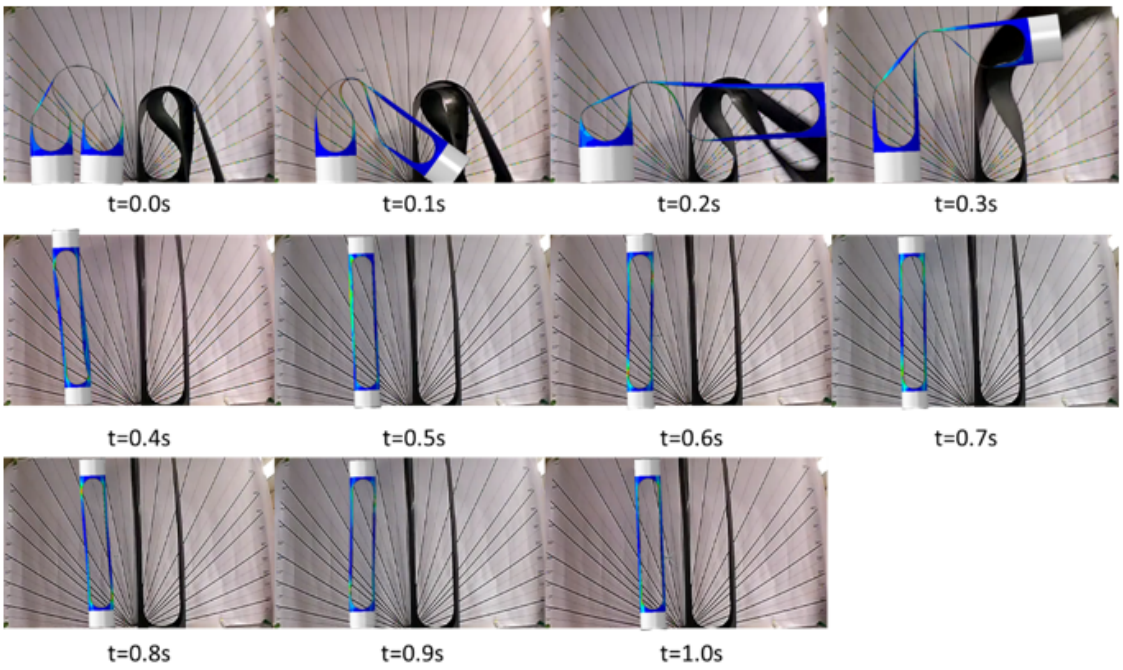

Figure 4.11: Comparison between the numerical model and a set of snapshots of a typical deployment test.

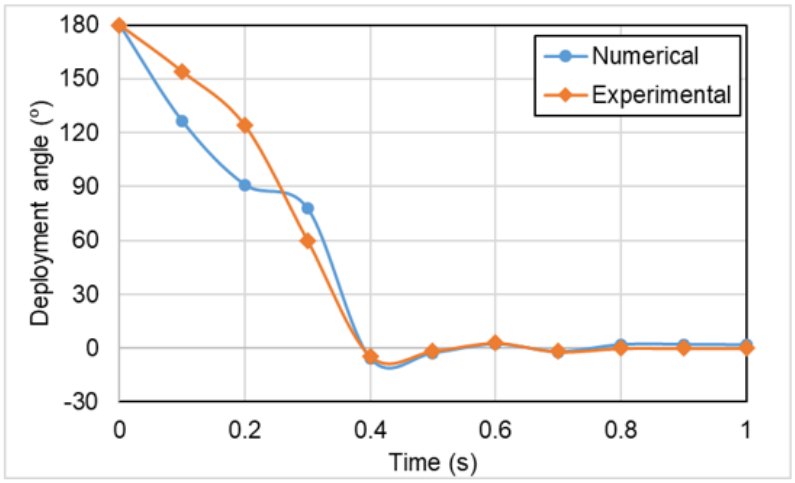

Figure 4.12: Experimental and numerical angle measurements as a function of time.

prediction. The reason behind this difference is that the numerical model predicted the extension to the right-most tape-spring, pushing the left tape-spring while the latter maintained an angle close to 90°. According to the experimental tests, this phenomenon does occur but to a lesser extent: the right tape-spring does push the left tape-spring but the angle formed by the left tape-spring is also increasing during this process. Finally, the deployment time is properly captured by the model. As can be observed, the full deployment occurs at 0.4 s, after this time mark, the elastic hinge slightly overshoots the 0° angle and then stabilizes. All this behaviour is properly captured by the model. Taking into account the observed behaviour and the obtained results, it can be concluded that the finite element model allows an appropriate estimation of the deployment behaviour of the elastic hinge specimen.

4.3.3 Prediction of the elastic hinge deployment behaviour after relaxation

It was shown that the model aforementioned described correlates well with the experimental data, therefore it was used to estimate the influence of the material relaxation on the deployment of the elastic hinge specimen. Due to the relaxation of the material, it is expected that the internal energy of the elastic hinge decreases as a consequence of the rearrangement of the material towards a lower energy state. This process should reduce the strain energy stored and lead to a slower deployment behaviour. To estimate the influence of this phenomenon on the deployment of the elastic hinge, the finite element model was used to estimate the effect of the 6 relaxation times described in Table 4.4. These results allow a comparison between the different internal energies and between the different angles formed by the elastic hinge specimen as a function of time. Figure 4.13 shows the different internal energies obtained after relaxation predicting a reduction of 32.3 % of the internal energy stored after the first month of stowage [2]. The remaining numerically estimated cases do not indicate a significant reduction of the internal energy from the first month onward, reaching a 34.65 % reduction after 2 years of relaxation. This deceleration is to be expected as the relaxation measured during the experimental characterization (detailed in sections 4.2.1 and 4.2.2) presented a linear logarithmic behaviour.

The angle formed by the elastic hinge specimen as a function of time was also measured, for the different conditions, and depicted in figure 4.14. These numerical results predict a reduction in the deployment speed of the elastic hinge when compared to the unrelaxed case. As a result, for relaxation times larger than 6 months, the numerical model predicts that the elastic hinge does not reach a deployed configuration (0° deployment angle) within the 1.0 s analysed time interval. Furthermore, the results indicate that the relaxation of the material causes the elastic hinge specimen to deploy in an oscillating movement.

[2] The data reported in this graphic is not obtained directly from ABAQUS® [11] in a single numerical simulation. The data reported is the internal energy observed if the elastic hinge was folded considering the material properties that the composite would have after each stowage period.

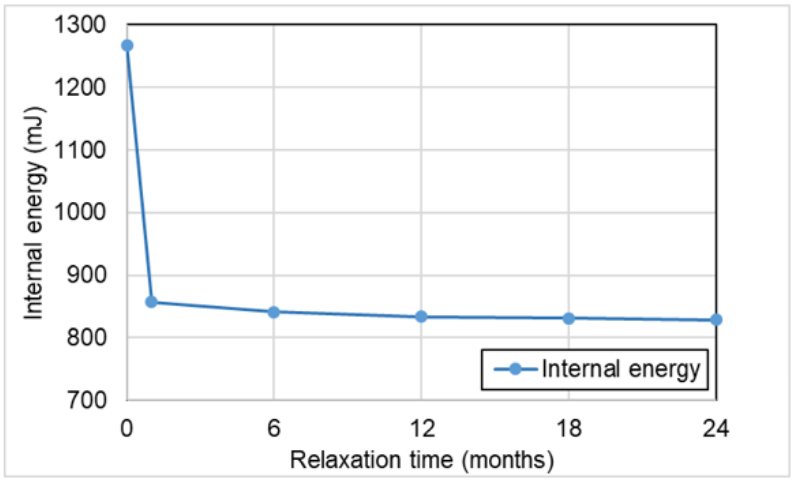

Figure 4.13: Estimated internal energy of the elastic hinge specimen, according to the finite element model.

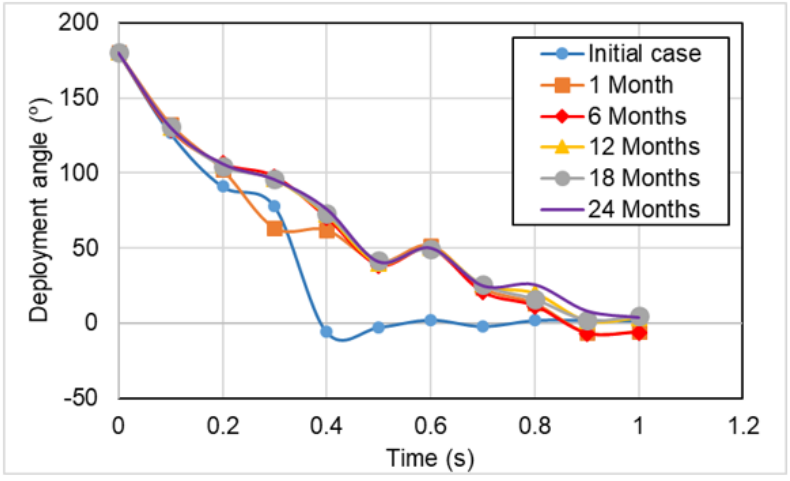

Figure 4.14: Estimated deployment behaviour as a function of time for different relaxation times, according to the finite element model results.

4.4 Conclusions

The present investigation studied the influence of material relaxation on the deployment behaviour of an elastic hinge specimen, using experimental tests to characterize the material and validate a finite element model. This model was later used to extrapolate the results obtained to larger relaxation periods. The main conclusions that can be drawn from this work are the following:

- Exposing the test specimens to a constant deformation for 3 hours led to an average material relaxation of 13.27 %, 14.79 %, and 20.31 % in the load applied at room temperature, 50, and 80 °C, respectively.
- The time-temperature superposition principle (TTS) allowed the prediction of the relaxation observed at room temperature after, approximately, 24 hours based on the data gathered during the 3 hours of experimental testing at temperatures of up to 80 °C.
- The material relaxation observed for the AS4/8552 prepreg UD carbon epoxy composite system followed a logarithmic linear behaviour.
- A finite element model was implemented to estimate the deployment behaviour of an elastic hinge specimen. The predictions of this model correlated well with the experimental results.
- It was predicted that a 32.3 % reduction of the internal energy of the elastic hinge specimen is observed as a result of the material relaxation within 1 month of stowage.
- For relaxation times larger than 6 months, the numerical model predicts that the elastic hinge does not reach a deployed configuration within the 1.0 s time interval analysed.
- A topic not addressed in this work, that should be explored in future research, is the correlation of the numerically predicted and experimental deployment behaviour of the elastic hinge after relaxation. To do so, and to ensure an accurate correlation, it would be necessary to guarantee constant temperature and humidity conditions during the ageing process, as well as using a rig to hold and maintain the configuration of the elastic hinge specimens once stowed. Furthermore, the influence of the air friction should be addressed, which can lead to a more accurate correlation between the numerical and the experimental data.
- The bending stress-strain state installed in the elastic hinge may cause compressive and tensile effects that contribute to a viscoelastic effect. Future research should also attempt to quantify the influence of this factor on the deployment behaviour of the elastic hinge, leading to a better understanding of which factor has a larger contribution to the increase of the deployment time: the stowage period or the stress-strain state determined by the compaction of the foldable elements.

The results reported in chapter 3 and chapter 4 indicate that, while it is possible to predict the deployment behaviour of a composite deployable structure even with the influence of relaxation phenomenon, the methodologies reported in the state of the art lead to some limitations in the design of these structures. Therefore, the following chapters will focus on possible design methods that may improve the performance of composite deployable structures [3].

[3] The reader interested in following the chronological order of research, should read sections 2.3 through 2.4 before moving on to chapter 5

Chapter 5

Performance analysis of a damage tolerant composite self-deployable elastic hinge

The present chapter is based on the following article submitted for refereed publication:

P. Fernandes, R. Pinto, A. Ferrer, N. Correia. *Performance analysis of a damage tolerant composite self-deployable elastic-hinge*. (Submitted to the Journal of Composite Structures, Elsevier)

5.1 Introduction

The design of self-deployable composite structures is characterized by two main opposing requirements [2, 3, 5, 4, 7–9]: flexibility of sustain high strain deformations; and rigidity to meet a stiffness-related requirement (such as the pointing accuracy, natural frequency, or deployment torque). The contradictory nature of these requirements leads to a challenging design process that may not always be successful in meeting both.

In chapter 3, a parametric optimization approach explored different elastic hinge designs, attempting to maximize the first natural frequency within a damage constraint [86]. Although the methodology was in-line with the state-of-the-art methods used to design these structures, it was not possible, for this specific case and geometry, to meet both design requirements. Failing to find a balance between these requirements leads to one of two situations: either the structure becomes too flexible, failing to meet the rigidity requirements necessary to operate, or the structure becomes too rigid, making it impossible to retract without initiating damage. Between these two, failing to meet the frequency requirement is the most detrimental as it completely invalidates the use of the elastic hinge in the desired range of natural frequencies / applications.

As shown in chapter 2, creating a damage tolerant elastic hinge design, capable of functioning even after initiating damage, has not been explored in the literature [318] and can be justified by two particularities of this application. The first is the life-cycle of the structure, as most deployable systems are expected to perform a single deployment operation once the spacecraft is in orbit. The second is the expected lifetime of a satellite. Apart from their size and cost, the development of nanosatellites and CubeSats is also motivated by their reduced development time [286–289]. Average or large-sized satellites require between 5 and 15 years to place in orbit under normal parameters, at the risk of market relevance, due to the pace of technological developments. In contrast, CubeSats and nanosatellites require less than eight months to place in orbit. This trend towards a shorter development time allows a frequent renewal, guarantees the robustness of nanosatellite constellations, and removes the need for a conservative long-term design [286–289]. As a result, allowing a controlled and limited damage initiation can also be a potential solution for developing a design capable of meeting the natural frequency requirements presented by ESA in [6].

The purpose of this research is to explore this possibility, evaluating if a damage tolerant design may be a valid design concept. The framework proposed herein includes the design of two elastic hinges, both obtained through an optimization algorithm whose objective function is the maximization of the first natural frequency. One of them is limited by a Max. IF < 1.0, imposing the absence of damage, while the other is limited by a Max. IF ≤ 1.10, allowing a limited initiation and propagation of damage. The damage tolerant design is numerically re-evaluated, leading to an estimate of its natural frequency considering a stiffness reduction resulting from damage initiation. The performance of both designs is then compared in terms of the first natural frequency, and the applicability of a damage tolerant design is evaluated.

A subject not included within the scope of this research is the influence of material relaxation

on the deployment of the structure. Although it can be argued that the literature addressing this issue does not account for the combined effect of damage and material relaxation [38–43, 319], the applications discussed in this article should reach orbit within a short time-frame that does not allow a significant development of this phenomenon.

Furthermore, it is not within the scope of this research the proposal of an optimization approach. In this work, the optimization algorithms used serve as a means to avoid human bias and the preference of one particular design over the other. For this reason, the hyper-parameters of the optimization algorithms used may not be optimal, as the only requirement for this research is that both possible designs may have equal opportunities of computational formulation and convergence.

5.2 Design requirements

The design requirements stated by ESA in [6] are used as a reference for the design problem, which is detailed in section 3.2. Similarly, the following failure criteria are considered: Hashin's failure criterion, Azzi-Tsai-Hill, Tsai-Hill, Tsai-Wu, and Maximum Stress.

It is assumed that the structure has not initiated damage if the Max. IF of these criteria is lower than 1.0. Likewise, it is assumed that the material damaged has a Max. IF larger or equal to 1.0. Therefore, one of the elastic hinges will have its Max. IF constrained to be lower than 1.0, imposing its functioning in the elastic regime, while the damage tolerant design will have its Max. IF constrained to be less than or equal to 1.10, allowing a limited initiation and propagation of damage. Note that the use of multiple failure criteria leads to a more conservative approach and mitigates the potential flaws that may be associated to each individual failure criteria.

5.3 Numerical analysis

Two finite element models, implemented in the commercial software ABAQUS® [11], were used to estimate and evaluate the natural frequency and structural integrity of the system. The natural frequency model is identical to the one described in section 3.3.1. The structural model is identical to one described in section 4.3.1 but with constant material properties (does not account for the influence of relaxation) and without the deployment step.

The interested reader is referred to the dataset [320], which contains the ABAQUS® input files for the structural and frequency models used.

5.4 Experimental work

The following sub-sections describe the experimental component of this investigation, including manufacturing details of both specimens and representative prototypes, processing of the experimental data, and description of the obtained experimental results. The composite system used in this research was AS4/8552, a space-certified prepreg made of carbon fibre and a high-performance

tough epoxy matrix, provided by HEXCEL Composites®, Madrid, whose material characterization campaign has been previously described in section 4.2.1

The accomplished test campaign served two primary purposes. The first was to characterize the material studied in this research regarding its mechanical properties. The second was to obtain experimental data that could be correlated with the numerical results, allowing the validation of the numerical models regarding the prediction of the stress and strain states of the composite material. Unless stated otherwise, all experiments included a minimum of 3 valid test results.

5.4.1 Preliminary model validation

Preliminary numerical model validation was performed before proceeding with the design of the two elastic hinges. This validation focused on correlating the strain installed in the composite material, in the elastic regime, with the corresponding prediction of the structural model. The reader is reminded that the natural frequency model has been experimentally validated by Sakovsky *et al.* [82], as detailed in section 3.3.1.

A dedicated rig was designed and coupled to an Instron 5900R universal testing machine (CAD representation shown in figure 5.1 a) to recreate the folding of the elastic hinge with the necessary control repeatability. This rig has three main components: a pair of holder rings that are attached to the ends of the elastic hinge, a guiding system that ensures a linear movement of the holder rings, and a folding tool that pulls the elastic hinge, causing it to fold in half (shown in figure 5.1 b).

For this experiment, three elastic hinge specimens were manufactured according to the recommendations and requirements of the material supplier (section 3.4.1) and each one was equipped with seven tri-axial extensometer rosettes (Vishay model L2A-06-031WW-120), suitable to measure large deformations up to 7000 micro strains. The extensometers measured the $\pm 45°$direction and the direction transverse to the fibre. Deformation in the longitudinal fibre direction is then computed using mechanics of materials basic concepts and reported accordingly. The output data of the numerical models allow for the direct comparison of the strain in the direction of the fibre (direction 11, aligned with the longitudinal axis of the elastic hinge) and in the direction transverse to the fibre (direction 22). The composite ply stacking sequence was [0°,90°,90°,0°] and, therefore, the outer and inner layer of the composite material in the elastic hinge are aligned with the longitudinal axis of the tube.

All specimens were loaded three times at a 50 mm/min displacement rate, leading to a total of nine records of the local deformation for each one of the seven extensometers, acquired at a frequency of 100Hz. The geometry of the specimen is described in figure 4.5 and the location of the extensometer rosettes are illustrated in figure 5.2 and figure 5.1 c).

The strain in the elements within an approximate radius of 25 mm of the extensometer's physical location was extracted from the numerical simulations to correlate with the experimental results. Both experimental and numerical results were then compared to evaluate the accuracy of the numerical prediction versus the range of strains measured experimentally. For a more precise analysis, the range of strains predicted by the numerical model was represented by two lines, "N.

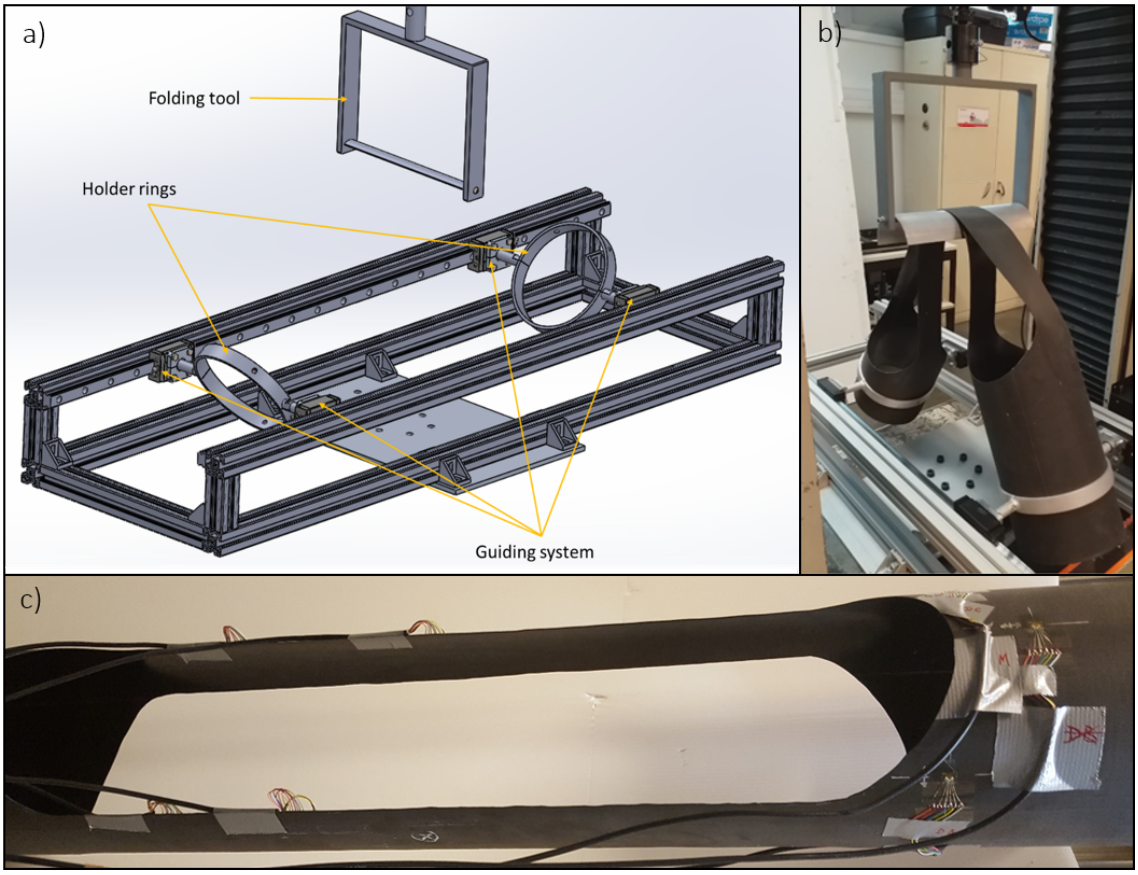

Figure 5.1: a) CAD representation of the rig used to fold the elastic hinge; b) setup of the rig with an elastic hinge, shown without extensometers for a clearer interpretation; c) specimen equipped with the extensometers.

min" and "N. max", that respectively indicate the minimum and maximum strains expected in that finite area (figure 5.3).

Due to the amount of data extracted during the experimental tests, the information regarding the remaining extensometers and their respective comparison with the numerical results are summarized in Table 5.1. For both numerical and valid experimental results (excluding sensors with abnormal behaviour), the summary includes the minimum and maximum strains (in percentage) observed during the folding of the elastic hinge, at the location of the different extensometers (direction 11 and 22). Each minimum and maximum was used to determine a range of deformation measured and numerically predicted to exist in each location. The third column (Difference to Numerical) presents the difference of the minimum, maximum, and range values between numerical and experimental results. The last column ("Within numerical range?") indicates if the experimental range of strains measured is within the numerical prediction. There are three possible results for this analysis: either the experimental result is completely within the numerical prediction ("Yes"), outside of both ranges of the numerical prediction ("No"), or outside of only one of either upper or lower range of the numerical prediction ("Partially"), as visually described in figure 5.4. It is important to note that being completely within the range of the numerical prediction indicates that the numerical result is

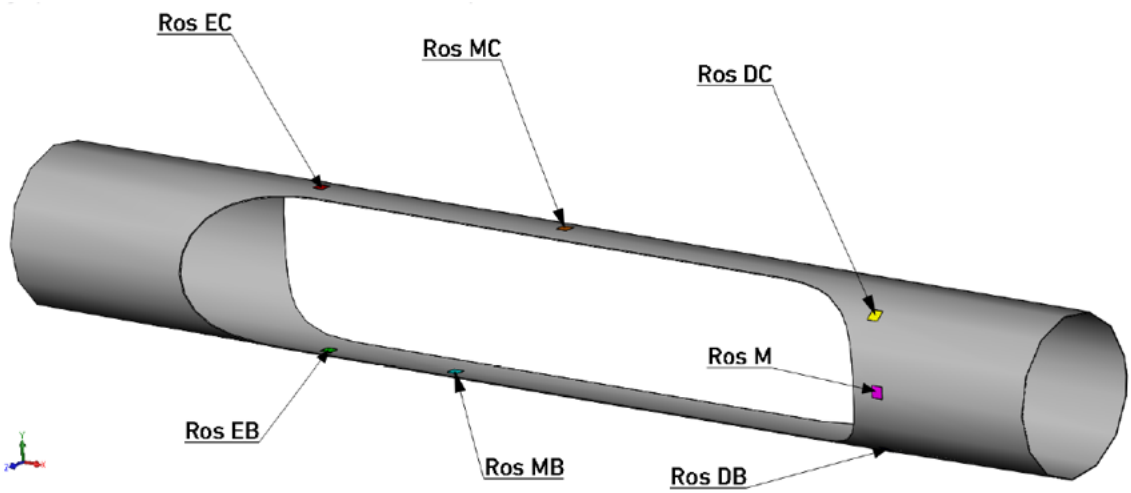

Figure 5.2: Location of the seven extensometers installed in each elastic hinge.

either accurate or conservative, therefore validating its correct functioning.

The results summarized in Table 5.1 indicate that the majority of the sensors (9 out of 14 results) validate the numerical predictions, with only five results partially complying with the numerical predictions. The observed result discrepancies can be attributed to:

- The existence of out-of-plane torsion, which, although almost unnoticeable to the naked eye, affects the recorded data in these locations due to the instabilities of the tape.
- The effect of the boundary conditions on the measurement, such as the fixation at both ends (case M and DB).
- The numerical model considers that the elastic hinge is folded in a perfect scenario, while in the experimental test, the tube is subjected to vibrations and instabilities due to the flexibility of the tape. Furthermore, the model considers a linear movement of the holders, while experimentally, a certain degree of friction is observed in the tool (affecting case MC). This creates noise that is captured by the extensometer.

Therefore, it is assumed that the outputs obtained from the numerical model are either accurate or conservative, validating the use of the structural numerical model for the design of the elastic hinges proposed in this article.

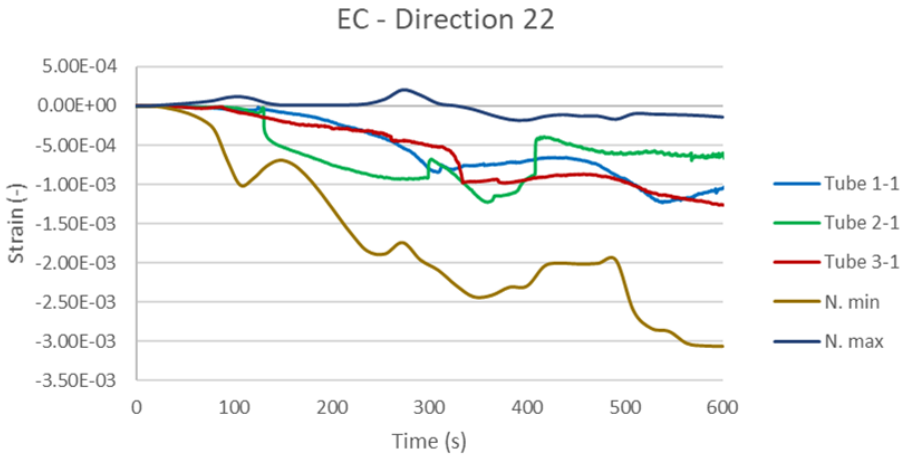

Figure 5.3: Example of the correlation between numerical and experimental strains observed at location EC, in direction 22.

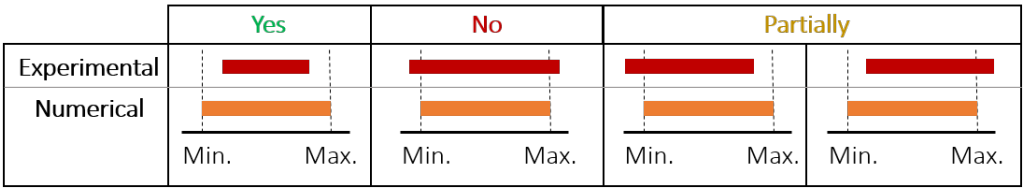

Figure 5.4: Possible correlations between experimental and numerical strain measurements.

Table 5.1: Correlation between valid experimental and numerical results. All results in this table are in percentage to improve readability.

Extensometer		Experimental			Numerical			Difference to Numerical			Within numerical range?
		Min.	Max.	Range	Min.	Max.	Range	Min.	Max.	Range	
EC	11	-0.15	0.025	0.175	-0.125	0.025	0.15	0.025	0.0	-0.025	Partially
	22	-0.15	0.01	0.16	-0.3	0.02	0.32	-0.15	0.01	0.16	Yes
MC	11	-0.4	1.0	1.4	0.0	6.0	6.0	0.4	5.0	4.6	Partially
	22	-0.45	-0.4	0.05	-0.5	-0.25	0.25	-0.05	0.15	0.2	Yes
MB	11	0.3	0.45	0.15	0.15	0.5	0.35	-0.15	0.05	0.2	Yes
	22	0.1	0.35	0.25	0.2	0.5	0.3	0.1	0.15	0.05	Yes
EB	11	-0.0025	0.01	0.0125	-0.01	0.02	0.03	-0.0075	0.01	0.0175	Yes
	22	0.02	0.12	0.1	0.01	0.175	0.165	-0.01	0.055	0.065	Yes
M	11	-0.004	0.0325	0.0365	-0.005	0.02	0.025	-0.001	-0.0125	-0.0115	Partially
	22	-0.0025	0.025	0.0275	-0.0025	0.009	0.0115	0.0	-0.016	-0.016	Partially
DC	11	-0.01	0.03	0.04	-0.015	0.04	0.055	-0.005	0.01	0.015	Yes
	22	-0.01	0.04	0.05	-0.01	0.12	0.13	0.0	0.08	0.08	Yes
DB	11	-0.02	0.08	0.1	-0.08	0.06	0.14	-0.06	-0.02	0.04	Partially
	22	0.02	0.085	0.065	-0.1	0.3	0.4	-0.12	0.215	0.335	Yes

5.5 Elastic hinge design and optimization

This section describes the design and optimization process of the composite deployable arm, including the design variables, objective function, and an analysis of the obtained outputs. The optimization process was performed through a GA. A description of its functioning, differences from the classic GA implementation and internal parameters chosen can be found in Appendix A. Section 5.5.1 describes the design variables used. The description of the objective function (section 5.5.2) explains how the information obtained from the numerical models was utilized to explore possible solutions and improve them.

5.5.1 Design variables

The layup considered is always symmetric. The geometry of the elastic hinge is defined by its internal radius (R_i) and by a double-symmetric spline (as shown in figure 5.5).

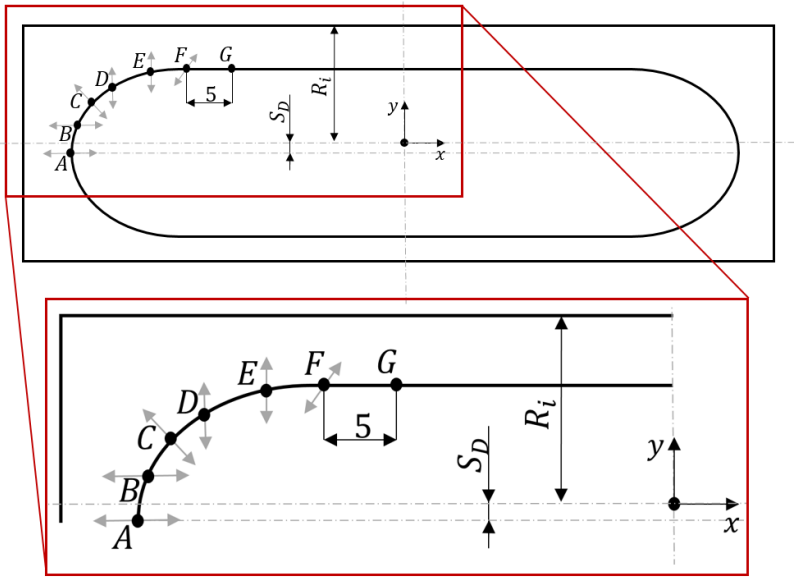

Figure 5.5: Parametrization of the design variables defining the slot cut-out of the elastic hinge. The grey arrows indicate the direction in which the control points are allowed to move.

The coordinates of the control points (points A through F) are defined as a function of six design variables (variable X_1 through X_6) and are proportional to R_i, promoting the scalability of the parametrization used and allowing a wide range of possible geometries. The position of the slot cut-out in relation to the longitudinal axis of the elastic hinge is defined by the design variable S_D (figure 5.5). Furthermore, a scaling factor L_f was included to consider different lengths of the slot cut-out.

The design variables are defined within the range shown in Table 5.2. Equations (5.1) through (5.6) establish the relationship between the design variables and the coordinates of the control points, leading to the minimum and maximum coordinates summarized in Table 5.3.

In previous research addressing this specific design problem [86], it was concluded that the composite layup should have four plies oriented at a ±40°angle, maximizing the first natural frequency of vibration at a minimal increase of the Max. IF. For this reason and to reduce the computational cost of the optimization algorithm, it was decided to exclude the number of plies and their orientation angle from the optimization problem.

$$A = \left(-R_i \times L_f(2.05 + 0.1X_1),\ -S_D\right) \tag{5.1}$$

$$B = \left(-R_i \times L_f(2.15 + 0.1X_2),\ 0.15R_i - S_D\right) \tag{5.2}$$

$$C = \left(-R_i \times L_f(2.15 + 0.1X_3),\ R_i(0.15X_3 + 0.15) - S_D\right) \tag{5.3}$$

$$D = \left(-2.15R_i \times L_f,\ R_i(0.1X_4 + 0.15) - S_D\right) \tag{5.4}$$

$$E = \left(-2R_i \times L_f,\ R_i(0.1X_5 + 0.05) - S_D\right) \tag{5.5}$$

$$F = \left(R_i \times L_f(0.9(0.1X_5 + 0.05) - 2),\ R_i(0.1X_5 + 0.05)(0.2X_6 + 0.2) - S_D\right). \tag{5.6}$$

Table 5.2: Minimum and maximum values of the design variables.

Design Variable	**Min.**	**Max.**	**Increment**	**Unit**
R_i	50	100	10	mm
S_D	0	100	20	% of R_i
$X_1, ..., X_6$	0	5	1	-
L_f	1	1.5	0.1	-

Table 5.3: Minimum and maximum coordinates of the control points, normalized as a function of R_i, L_f and S_D.

Control Point	**X** × $(1/R_i\ L_f)$		**Y**/Ri +S_D	
	Min.	Max.	Min.	Max.
A	-2.05	-2.55	0	0
B	-2.15	-2.65	0.15	0.15
C	-2.15	-2.65	0.15	0.65
D	-2.15	-2.15	0.15	0.65
E	-2	-2	0.05	0.45
F	-1.995	-1.595	0.01	0.54

5.5.2 Objective function

The objective of the design problem is to achieve the highest natural frequency of vibration within the damage constraints imposed. The design problem was defined as the minimization of equation (5.7) for the elastic design and the minimization of equation (5.8) for the damage tolerant design. As a result, the GA will first try to find solutions that meet the damage constraint, and among them, the designs that have the highest first natural frequency.

Both objective functions are divided into two branches. The first branch evaluates the performance of designs that do not meet the damage constraint (Max. IF$<$1.0 for the elastic design and Max. IF$\leq$1.1 for the damage tolerant design). In this situation, the performance (P) of the elastic hinge design is set equal to its Max. IF. Therefore, the optimization algorithm will prefer solutions that minimize the Max. IF and are considered suitable for this application. The second branch evaluates the performance of designs that meet the damage constraint. In this case, the performance of the solution is set equal to the symmetric value of the first natural frequency of vibration ($-N_F$) of the elastic hinge.

$$P = \begin{cases} Max.\,IF, & if\, Max.\,IF \geq 1.0 \\ -N_F, & if\, Max.\,IF < 1.0 \end{cases} \tag{5.7}$$

$$P = \begin{cases} Max.\,IF, & if\, Max.\,IF > 1.1 \\ -N_F, & if\, Max.\,IF \leq 1.1. \end{cases} \tag{5.8}$$

Overall, the objective functions selected allow the optimization algorithm to search for solutions within the damage initiation constraint and then promote the selection of characteristics that maximize the first natural frequency of vibration (minimize the symmetric value of the natural frequency of vibration).

5.5.3 Design evaluation

The optimization process described in section 5.5 was used to obtain two elastic hinge designs: an elastic hinge that does not initiate damage during operation, resulting from the objective function described in equation (5.7), and a damage tolerant elastic hinge, obtained from the objective function detailed in equation (5.8). Table 5.4 summarizes the design variables that describe the obtained solutions.

Despite converging, the optimization process that used equation (5.7) did not find a solution with a Max. IF below 1.0. Therefore, the following comparison will assume the solution closest to meeting this requirement. The best solution found has a Max. IF of 1.04 and a first natural frequency of vibration of 1.16 Hz (shown in figure 5.6).

It is important to note several observations regarding this choice. First, no solution was found with a Max. IF $<$1.0 highlights how severely this restriction affects the design space and the difficulty of designing a structure in this condition. Second, the attentive reader will notice that not having a solution with a Max. IF $<$1.0 implies that the algorithm only attempted to minimize the Max. IF

Table 5.4: Design variables defining the elastic and damage tolerant designs.

	Elastic design	**Damage tolerant**
R_i	10	10
X_1	4	1
X_2	5	0
X_3	3	3
X_4	1	4
X_5	5	5
X_6	4	3
S_D	1	1
L_f	5	0

until an acceptable value was met, completely disregarding the second phase the optimization process in which it to maximize the first natural frequency. Such reasoning would lead to an argument stating that the first natural frequency of the elastic design could be higher than 1.16 Hz and make this solution an unsuitable representative of the elastic design as a concept. However, this possibility is improbable for two reasons. First, the Max. IF decreases with the stiffness of the elastic design, which would cause a further decrease in the first natural frequency, as shown by equation (5.9), where K and m are the global stiffness and mass of the elastic hinge, respectively. The only exception to this argument is the possibility of existing a topologically optimized solution such that the global stiffness of the elastic hinge is maintained but locally redistributed to avoid the initiation of damage. However, the same argument could be made for the damage tolerant design and potentially increase its natural frequency as well, this possibility has already been explored by the GA when attempting to minimize the Max. IF. As a result, although it is not possible to prove that a global optimum has been found, based on the number of solutions evaluated during this design process and previous research [86], it is safe to assume that it is implausible that a better elastic design solution exists. Therefore, it is reasonably safe to assume that, by choosing a design that has a Max. IF = 1.04, we are overestimating the natural frequency that the elastic design approach can reach, making this solution a valid representative of the elastic design as a concept.

$$N_F = \sqrt{\frac{K}{m}}. \tag{5.9}$$

The damage tolerant elastic hinge design has a Max. IF of 1.095 and a first natural frequency of vibration of 1.338 Hz. Figure 5.6 shows both elastic and damage tolerant designs obtained through their respective optimization processes. Both designs have a diameter of 200.0 mm. The length of the cut-out slot is approximately 795.0 mm for the elastic design and 490.0 mm for the damage tolerant design. To better understand and explore the particularities of both solutions, the interested reader is referred to the dataset [320], which contains the ABAQUS® input files for the structural and frequency models used to simulate both designs.

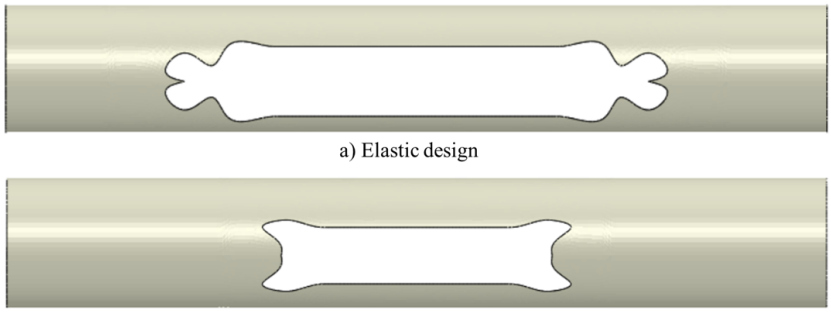

Figure 5.6: Representation of the elastic (a) and damage tolerant (b) designs.

5.6 Performance comparison and discussion

Figure 5.7 highlights, in grey, the location and extension of the regions that have initiated damage during the folding process of the damage tolerant design. The natural frequency model was used to estimate the influence of the damage initiation on the first natural frequency of the damage tolerant design. The stiffness material properties of the elements highlighted in grey were multiplied by a factor of 10^{-6}, leading to a stiffness reduction of the structure. In these conditions, the natural frequency of the damage tolerant design is reduced to 1.3374 Hz, equivalent to a reduction of 0.05 %. It is important to note that the stiffness reduction was applied to all layers of the composite lay-up, which is a conservative approach. If damage had initiated in all layers of the composite lay-up, the plot of Max. IF would have been symmetric, as the layup and the geometry are also symmetric. Nonetheless, despite the plot of Max. IF being asymmetric, the numerical results presented in this section still assume a pessimistic approach and account for the initiation and propagation of damage across all plies.

Comparing the final natural frequencies of the elastic and damage tolerant designs, it is observable that allowing the initiation of damage initiation is beneficial. It led to an increase of 15.3 % of the first natural frequency of vibration compared to the analogous parameter of the elastic design. However, it is also noticeable that this improvement was only possible due to the very localized initiation of damage in the damage tolerant design. A series of parametric analyses were performed to better understand how the potential damage propagation affects the natural frequency, and the performance comparison was discussed. As shown in figure 5.8, each case simulates the propagation of damage to regions where the Max. IF is lower than 1.0. Then, for each of these conditions, the value of the first natural frequency of vibration was re-determined, considering that the elements with a Max. IF above the indicated threshold are damaged. Figure 5.9 illustrates the reduction in the natural frequency of vibration as the damage propagation increases, in other words, as the Max. IF threshold decreases.

Observing the graphic shown in figure 5.9, it is possible to conclude that the damage tolerant design has a better performance than the elastic design as long as the damage does not propagate beyond the material with a Max. IF = 0.92. As shown in figure 5.8, this scenario corresponds to the propagation of damage through a length larger than half of the width of the lower tape-spring

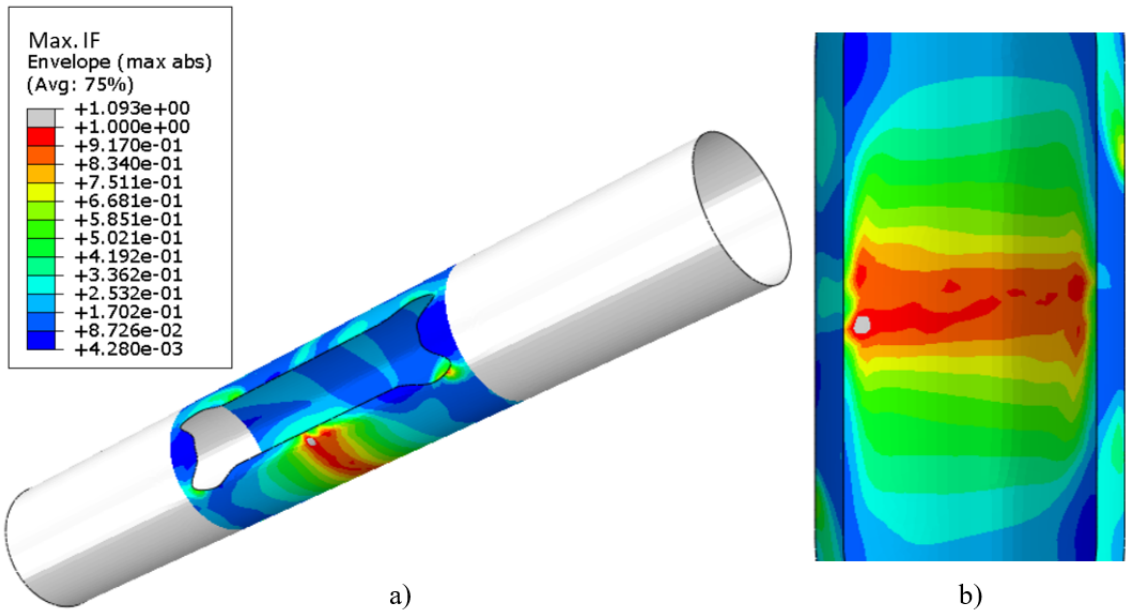

Figure 5.7: Max. IF observed in the damage tolerant design: a) bottom-up perspective view of the damage tolerant elastic hinge; b) close-up view of the lower tape-spring. The grey colour in the central section of the hinge highlights the regions where the Max. IF is higher than 1.0.

(Max. IF threshold between 0.91 and 0.92, in figure 5.8). To better understand the extension of the damage propagation represented in this case, it is essential to note that the length of the damage propagated corresponds to almost 20 % of the perimeter formed by the cross-section of both upper and lower tape-springs.

Based on these observations, it is possible to conclude that at least one case has very well-defined conditions in which a damage tolerant design has a better performance than the conventional elastic design. These results are expectable and can be explained from two different perspectives. From an optimization perspective, increasing the allowable Max. IF by a factor of 10 % represents the relaxation of this constraint, allowing the optimization algorithm to search a region of the design space that was previously unavailable and potentially find more suitable solutions. From a structural perspective, it can be interpreted that the global stiffness increase has a more considerable benefit in the first natural frequency of vibration than the prejudice caused by the local failure caused by the damage initiation and its eventual propagation.

Finally, it is essential to note that this output is the result of allowing the Max. IF to reach a maximum value of 1.1, which was selected as an initial guess. Allowing a higher or lower threshold may further enable the damage tolerant design. Therefore, it is relevant to evaluate the influence of this threshold on the structural performance of the damage tolerant design, as it will provide significant insight on the possible influences of increasing or decreasing the allowable Max. IF.

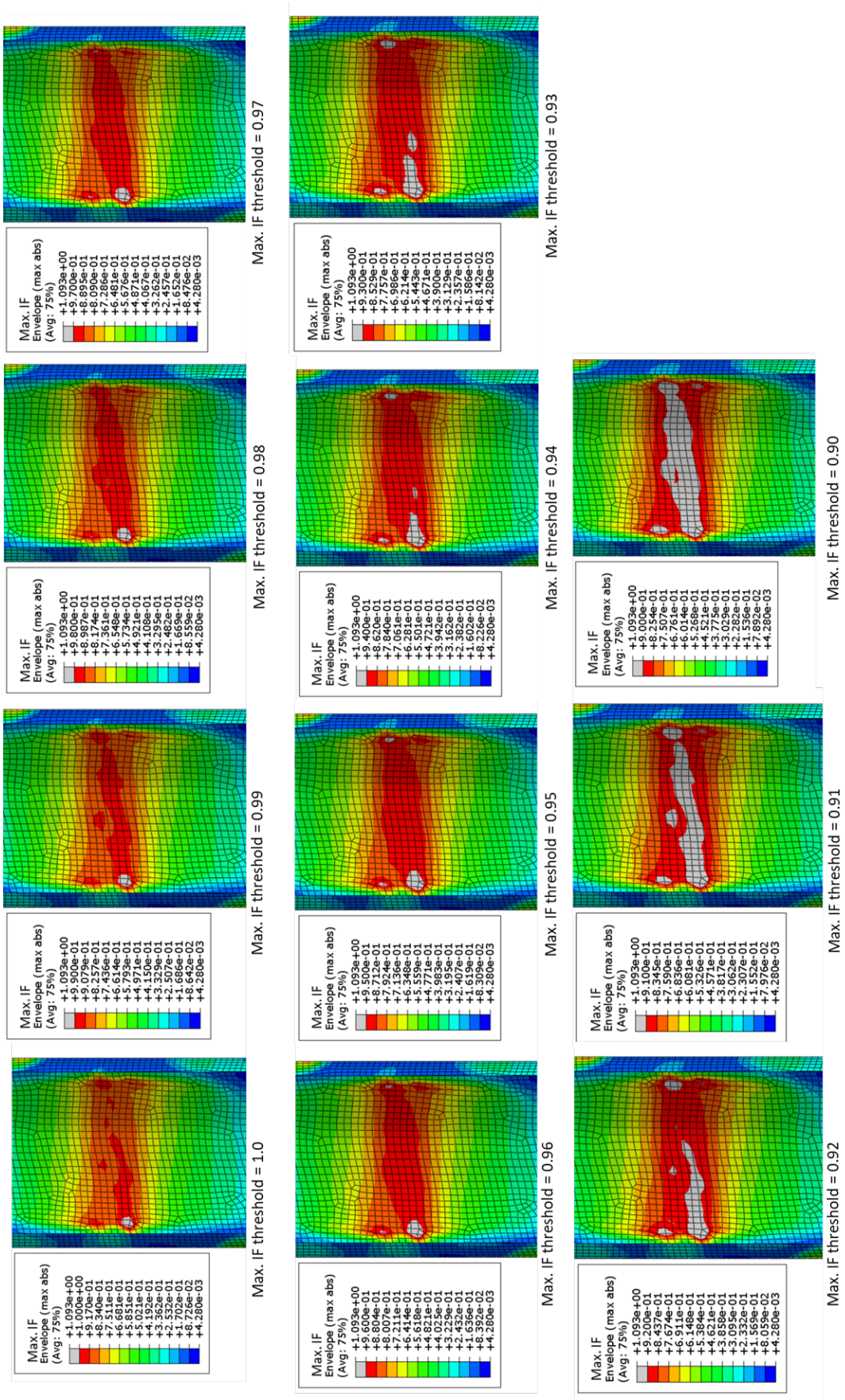

Figure 5.8: Representation of the expected damage propagation as a function of the Max. IF. The elements with a Max. IF above the indicated threshold are assumed to be damaged.

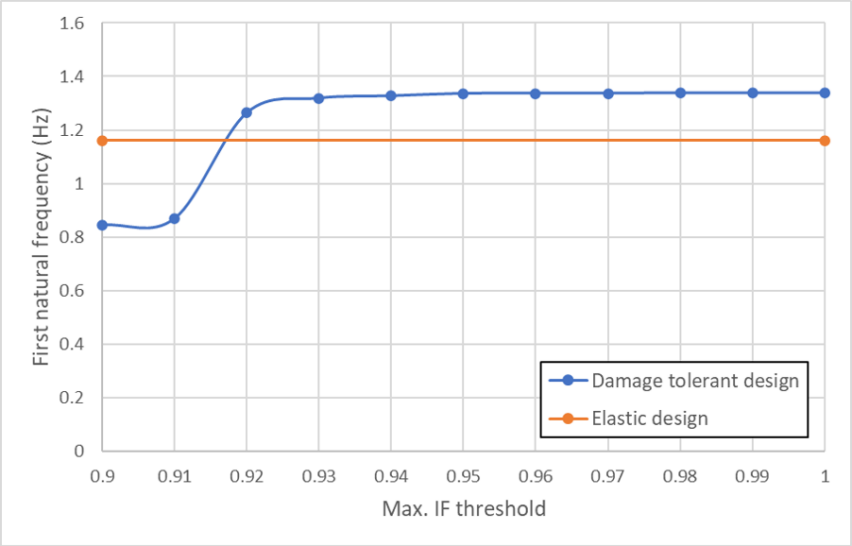

Figure 5.9: First natural frequency of both elastic and damage tolerant designs as a function of the damage propagation. For each Max. IF, the material removed in the damage tolerant design is equal to the elements highlighted in grey in Figure 5.8.

5.7 Conclusions

The present research studied the possibility of developing a damage tolerant elastic hinge design, comparing the first natural frequency of vibration of a damage tolerant design with a design that does not initiate damage during operation. Both elastic hinge designs used in this comparison were obtained through an optimization process that utilized data estimated by two finite element models that have been experimentally validated. Finally, the performance of the damage tolerant design was evaluated considering the structural stiffness reduction resulting from the initiation of damage during operation.

The main conclusions that can be drawn from this work are the following:

- The design of an elastic hinge that does not initiate damage during operation (Max. IF$<$1.0) led to a maximum natural frequency of 1.16 HZ. On the other hand, the damage tolerant design (Max. IF$\leq$1.1) had a maximum natural frequency of vibration of 1.338 Hz.

- The very localized initiation of damage observed in the damage tolerant elastic hinge caused a reduction of the first natural frequency of the vibration from 1.338 Hz to 1.3374 Hz, resulting in a decrease of 0.05 %.

- A parametric analysis was performed to evaluate the influence of the potential further propagation of damage. The results indicate that the damage tolerant design will maintain a larger first natural frequency of vibration than the elastic design unless the length of the damaged material is approximately larger than half the width of the lower tape-spring of the damage tolerant design.

Overall, the results obtained in this research indicate that the use of a damage tolerant elastic hinge design may be a good approach to increasing the first natural frequency of vibration achieved. Furthermore, it was observed that the damage tolerant design could meet the first natural frequency requirements defined by ESA in [6].

The improvement in performance suggested by these results can be explained from two different perspectives. From an optimization perspective, allowing the initiation of damage enables the optimization algorithm to search a previously restricted region of the design space and potentially find more suitable solutions. From a structural perspective, it can be interpreted that the global increase in stiffness has a more beneficial influence than the prejudice caused by the initiation, and propagation, of damage.

Nonetheless, further research is required before implementing a damage tolerant concept. From a performance point of view, the definition of the Max. IF allowed during the optimization process needs to be further studied, as it may lead to drastic changes in the design and performance of the solution obtained. From an implementation point of view, the use of a damage tolerant solution requires additional caution, as it is prone to the release of debris caused by the initiation of damage in the composite material.

Chapter 6

Stress constrained topology optimization

The present chapter is based on the following submitted article:

P. Fernandes, A. Ferrer, P. Gonçalves, R. Pinto, N. Correia. *Python code for stress constrained topology optimization in ABAQUS® - Theory, implementation, and case studies*. (Submitted to the Journal of Structural and Multidisciplinary Optimization, Springer)

6.1 Introduction

The popularity of topology optimization has led to the publishing of several educational articles. These include the MATLAB codes written by Ole Sigmund [148], later revisited by Andreassen el al. [321], and an equivalent implementation for 3D problems by Liu [322], all of them using the Solid Isotropic Material with Penalization (SIMP) method [323, 324] to define the material properties and the Optimality Criteria (OC) [325] to determine the optimal design variables. Other publications include the MATLAB implementation of the level-set method [326, 327] by Challis [328], and the implementations of the Bi-directional Evolutionary Structural Optimization (BESO) method [329] in MATLAB and Python by Huang and Xie [330] and Zuo and Xie [331], respectively. The problems solved by these codes focus on structural or thermal optimizations considering a single material constraint, applied to the mass or volume of the structure.

The present chapter builds upon these publications, adding the implementation of stress constrained compliance minimization and stress minimization problem statements. Furthermore, in addition to the possibility of using the OC (for discrete and continuous variables) and the MMA algorithms, the code allows the use of Sequential Least Squares Programming (SLSQP) and the trust-region algorithm 'Trust-constr' available in the Python module SciPy [346].

The code provided is written in Python and uses ABAQUS® as an interface and finite element method (FEM) module. By doing so, the user is given a means to easily apply the code to new problems while taking advantage of ABAQUS® finite element analysis (FEA) capabilities. The complete source code is available in appendix C and can be downloaded from the following Dataset [332] or repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS.

The following sections of the article are organized as follows. Sections 6.2 and 6.3 formulate the topology optimization problem statements and deduce the necessary function derivatives in both continuous and discrete formulations, respectively. Section 6.4 describes the filter technique to avoid numerical instabilities. Section 6.5 overviews the optimization algorithms available in the provided code. Section 6.6 summarizes the most relevant aspects of the Python code implementation, while in section 6.7 the case studies considered are presented. Section 6.8 demonstrates the suitability of the code implementation to solve the case study proposed. The results of the topology optimization processes are presented in section 6.9. Finally, section 6.10 summarizes the main conclusions of the present work.

6.2 Continuous formulation of topology optimization problem statements and sensitivities

6.2.1 Topology optimization problem

Consider the formulation of a generic topology optimization problem, defined as find $\rho \in L^\infty(\Omega)$ such that:

$$\min_{\rho} : J(\rho, u(\rho)) \tag{6.1}$$

subject to:

$$\rho_{min} \leq \rho \leq 1 \tag{6.2}$$

$$\int_\Omega \rho \; dx \leq V^* \tag{6.3}$$

and when considering a maximum stress constraint, also subject to:

$$\int_\Omega \sigma_a^{VM}(\rho, u(\rho)) \; dx \leq \sigma^* \tag{6.4}$$

where $L^\infty(\Omega)$ is the space of bounded functions, and $u(\rho)$ are the displacement solutions of the standard equilibrium equation presented as find $u \in H_0^1(\Omega)$, such that:

$$a(\rho, u, v) = l(\rho, v) \; \forall \; v \in H_0^1(\Omega) \tag{6.5}$$

where $H_0^1(\Omega)$ are the space of functions with square integrable derivatives and homogeneous values on the boundary. The bilinear form is:

$$a(\rho, u, v) = \int_\Omega \bigtriangledown^s u \mathbb{C}(\rho) \bigtriangledown^s v \; dx \tag{6.6}$$

with the external forces $l(\rho, v)$ written as:

$$l(\rho, v) = \int_\Omega f(\rho) \; v \; dx + \int_{\partial\Omega} t \cdot nv \; dx \tag{6.7}$$

Here, the density ρ represents the design variables, which can vary within the interval $[\rho_{min}, 1]$, v the corresponding test function, n the normal direction pointing outwards of the boundary $\partial\Omega$ of v, $f(\rho)$ a generic volumetric load function, and Ω the domain of the topology optimization problem. t indicates the applied boundary forces, if they exist. $\mathbb{C}(\rho)$ is the constitutive tensor. $\bigtriangledown^s$ represents symmetric gradient, and thus the strains $\xi(u) = \bigtriangledown^s u$ are in accordance to linear elasticity.

$\int \rho \, dx$ represents a generic volume constraint function, and V^* its allowable maximum value. It is important to note that this material constraint can be applied to either the volume of mass of the structure, following the same expressions. The volume constraint is preferred and used in this research to conform with the most used term in the literature. $\int_\Omega \sigma_a^{VM}(\rho, u(\rho)) \, dx$ represents a maximum stress constraint with a maximum allowable value σ^*.

6.2.2 Regularization and penalization

Regarding this particular work and implementation, it is important to note two adopted considerations. The first one is the use of two penalization factors applied to the material stiffness, and stress. These factors aim at making intermediate design solutions uncompetitive, and in turn promoting black-and-white solutions.

The material stiffness is penalized by the SIMP penalization parameter $P = 3.0$, in accordance to [137], when using continuous design variables (equation (6.8)). In this implementation, this factor is also applied to the other material properties, as:

$$\mathbb{C}(\rho) = \rho^P \mathbb{C}_0 \tag{6.8}$$

where $\mathbb{C}_0$ is the material stiffness of a fully solid element. Note that $\mathbb{C}'(\rho) = P\rho^{P-1}\mathbb{C}_0$ for this type of stiffness penalization.

The stress is also penalized according to a factor equal to ρ^β, with $\beta = \frac{1}{2}$ as adopted in [333], following the initial proposal by Bruggi et al. in [334] with the exponent suggested in [335]. This penalization leads to a non-physical stress for intermediate design densities, but not for black-and-white solutions, and tends towards 0 when the design density decreases, avoiding singularity problems [333]. Thus we define σ_a as the amplified stress, described by the following expression:

$$\sigma_a(\rho) = \rho^\beta \hat{\sigma}(\rho) \tag{6.9}$$

with:

$$\hat{\sigma}(\rho) = \mathbb{C}_0 \bigtriangledown^s u(\rho) \tag{6.10}$$

where $\hat{\sigma}(\rho)$ is the stress vector, written in Voigt notation.

The second consideration is the use of a regularization approach, where removed or void elements will maintain a minimum design density, $\rho_{min} = 0.01$ by default, which contrasts with the complete element removal. In the literature, these regularization techniques may be referred to as "soft-kill" and "hard-kill" approaches [331]. The regularization approach was introduced in the code implementation to avoid the stiffness matrix becoming singular, as well as allowing a constant mesh in the finite element model during the topology optimization loop [331, 333].

6.2.3 Sensitivity analysis

The derivative of the cost function in the direction $\tilde{\rho} \in L^\infty(\Omega)$ is then defined as:

$$DJ(\rho,u(\rho))\tilde{\rho} = D_\rho J(\rho,u(\rho))\tilde{\rho} + D_u J(\rho,u(\rho))\; D_\rho u(\rho)\tilde{\rho} \quad \forall \quad \tilde{\rho} \in L^\infty(\Omega) \tag{6.11}$$

Taking the derivative in the equilibrium equation, in order to find $D_u J(\rho,u(\rho))\; D_\rho u(\rho)\tilde{\rho}$, we have:

$$\begin{aligned} &[D_\rho a(\rho,u(\rho),v) - D_\rho l(\rho,v)]\tilde{\rho} + D_u a(\rho,u(\rho),v)\; D_\rho u(\rho)\tilde{\rho} = 0 \\ &\quad \forall\, v \in H_0^1(\Omega),\; \forall\, \tilde{\rho} \in L^\infty(\Omega). \end{aligned} \tag{6.12}$$

Since $a(\rho,u(\rho),v)$ is linear, we have:

$$D_u a(\rho,u(\rho),v)\; D_\rho u(\rho)\tilde{\rho} = a(\rho,D_\rho u(\rho)\tilde{\rho},v) \tag{6.13}$$

thus equation (6.12) can be rewritten as:

$$-a(\rho,D_\rho u(\rho)\tilde{\rho},v) = [D_\rho a(\rho,u(\rho),v) - D_\rho l(\rho,v)]\tilde{\rho} \;\; \forall\, v,\tilde{\rho}. \tag{6.14}$$

Solving equation (6.14) for all values of $\tilde{\rho}$ would be too expensive. For this reason, the use of the adjoint method is preferred. To do so, we define an adjoint variable λ, solution of:

$$\begin{aligned} &a(\rho,\lambda,w) = -D_u J(\rho,u(\rho))w \;\; \forall\, w \in H_0^1(\Omega) \Leftrightarrow \\ &\Leftrightarrow -a(\rho,D_\rho u(\rho)\tilde{\rho},\lambda) = D_u J(\rho,u(\rho))\; D_\rho u(\rho)\tilde{\rho} \;\; \forall\, \tilde{\rho}. \end{aligned} \tag{6.15}$$

Then, taking $w = D_\rho u(\rho)\tilde{\rho}$ in equation (6.15):

$$\begin{aligned} D_u J(\rho,u(\rho))\; D_\rho u(\rho)\tilde{\rho} &= -a(\rho,\lambda,D_\rho u(\rho)\tilde{\rho}) = \\ &= -a(\rho,D_\rho u(\rho)\tilde{\rho},\lambda) = [D_\rho a(\rho,u(\rho),\lambda) - D_\rho l(\rho,v)]\tilde{\rho} \end{aligned} \tag{6.16}$$

where the self-adjoint property of $a(\rho,\lambda,w)$ and equation (6.12) are applied.

The generic optimization process, which can be defined in four main steps, is proposed:

1. find $u(\rho)$ solution of: $a(\rho,u,v) = l(\rho,v)\; \forall\, v$
2. find λ solution of: $a(\rho,\lambda,w) = -D_u J(\rho,u(\rho))w\; \forall\, w$
3. compute the derivative as $DJ(\rho,u(\rho))\tilde{\rho} = [D_\rho J(\rho,u(\rho)) + a_\rho(\rho,u(\rho),\lambda) - l_\rho(\rho,\lambda)]\tilde{\rho}$
4. update the design variables using the gradient information.

The first two steps correspond to solving the state and adjoint problems, respectively. The third step is the gradient calculation, which is defined in terms of the state and adjoint variables. The fourth and final step consists on using the gradient (here represented by the letter g) to determine the next value of the design variables. This step, which can be performed by any suitable optimization

algorithms (such as the algorithms described in section 6.5), is here represented in a generic form in the fourth step.

6.2.4 Compliance functional

In the particular problem statement of compliance minimization, the objective function can be rewritten as:

$$J(\rho, u(\rho)) = \int_\Omega f u \, dx \tag{6.17}$$

and its derivative in the direction $w \in H_0^1(\Omega)$ is then:

$$D_u J(\rho, u(\rho))w = \int_\Omega f w \, dx = l(w). \tag{6.18}$$

Notice that in this case, it is assumed that the external loads do not depend on ρ, therefore:

$$a(\rho, u, v) = l(v). \tag{6.19}$$

In this particular case, the adjoint variable leads to the same expression shown in equation (6.15), and by the definition of equation (6.5), we can also imply that:

$$\begin{aligned} &a(\rho, \lambda, w) = -D_u J(\rho, u(\rho))w \quad \forall\, w \\ &\Leftrightarrow -a(\rho, u, w) = -D_u J(\rho, u(\rho)) = -l(w). \end{aligned} \tag{6.20}$$

Note that equation (6.20), which describes the adjoint problem of the compliance minimization problem statement, returns the definition of the state problem, setting them equal to each other. Functionals that are "self-adjoint" lead to clear computational benefits since equations (6.5) and (6.15) can be solved with the same computation, a single FEA in the case of the code provided.

Since $l(v)$ and $l(w)$ do not depend on ρ, the terms $D_\rho J(\rho, u(\rho))\tilde{\rho} = D_\rho l(w)\tilde{\rho} = D_\rho l(v)\tilde{\rho} = 0$ and thus the derivative is just:

$$DJ(\rho, u(\rho))\tilde{\rho} = -D_\rho a(\rho, u(\rho), u(\rho))\tilde{\rho}. \tag{6.21}$$

Following the definition of the compliance, the derivative of the bilinear form is:

$$D_\rho a(\rho, u(\rho), v) = \int_\Omega \bigtriangledown^s v \mathbb{C}'(\rho) \bigtriangledown^s u \tilde{\rho} \, dx \tag{6.22}$$

Finally, considering the stiffness penalization used, equation (6.22) can be introduced in (6.21)

obtaining:

$$DJ(\rho,u(\rho))\tilde{\rho} = \int_\Omega \bigtriangledown^s u(P\rho^{P-1}\mathbb{C}_0)\bigtriangledown^s u\tilde{\rho}\ dx = \int_\Omega g\tilde{\rho}\ dx \tag{6.23}$$

where g is the gradient of the compliance objective function, thus obtained as $g = \bigtriangledown^s u(P\rho^{P-1}\mathbb{C}_0)\bigtriangledown^s u$. For this particular case, the first three optimization steps can be rewritten as:

1. find u solution of: $a(\rho,u(\rho),v) = l(v)\ \forall\ v$
2. take $\lambda = -u$, since $a(\rho,\lambda,w) = -D_u J(\rho,u(\rho))w = -l(v)\ \forall\ w$ is exactly the same problem as the first step.
3. compute: $DJ(\rho,u(\rho))\tilde{\rho} = -a_\rho(\rho,u(\rho),u(\rho))\tilde{\rho} = \int_\Omega \bigtriangledown^s u(P\rho^{P-1}\mathbb{C}_0)\bigtriangledown^s u\tilde{\rho}\ dx$

6.2.5 Stress functional

In the stress minimization or stress-constrained compliance minimization problem statements, the derivative of the stress norm functional should also be taken. Here, the maximum function is approximated by means of a modified p-norm function. This approximation is necessary to provide a derivable function that approximates the maximum function, which is non-differentiable. Adopting the modified p-norm approximation proposed in [333], we can redefine the maximum function generically represented in equation (6.4) as:

$$J(\rho,u(\rho)) = \left(\int_\Omega \left(\sigma_a^{VM}(\rho,u(\rho))\right)^q\ dx\right)^{\frac{1}{q}} \tag{6.24}$$

where q is the exponential factor, and the von Mises amplified stress norm is $\sigma_a^{VM} = (\sigma_a \mathbf{M}\sigma_a)^{\frac{1}{2}}$, with $\mathbf{M}$ being the von Mises matrix.

The term $D_\rho J(\rho,u(\rho))(\tilde{\rho})$ is defined as:

$$\begin{aligned} &D_\rho J(\rho,u(\rho))(\tilde{\rho}) = \\ &\quad = \left(\int_\Omega \left(\sigma_a^{VM}(\rho,u(\rho))\right)^q\ dx\right)^{\frac{1}{q}-1} q\int_\Omega \sigma_a^{VM}(\rho,u(\rho))^{q-1} D_{\sigma_a}\sigma_a^{VM}(D_\rho\sigma_a(\tilde{\rho}))\ dx \end{aligned} \tag{6.25}$$

where:

$$D_\rho\sigma_a(\tilde{\rho}) = \beta\rho^{\beta-1}\mathbb{C}_0\bigtriangledown^s u\tilde{\rho}. \tag{6.26}$$

The derivative in the direction w, which allows us to define the adjoint problem, is:

$$\begin{aligned} &D_u J(\rho,u(\rho))(w) = \\ &\quad = \left(\int_\Omega \left(\sigma_a^{VM}(\rho,u(\rho))\right)^q\ dx\right)^{\frac{1}{q}-1} \int_\Omega \sigma_a^{VM}(\rho,u(\rho))^{q-1} D_{\sigma_a}\sigma_a^{VM}(D_u\sigma_a(w))\ dx \end{aligned} \tag{6.27}$$

where:

$$D_{\sigma_a}\sigma_a^{VM}(D_u\sigma_a(w)) = (\sigma_a \mathbf{M}\sigma_a)^{-\frac{1}{2}}\sigma_a \mathbf{M} D_u\sigma_a(w) \tag{6.28}$$

with:

$$D_u\sigma_a(w) = \rho^{\beta}\mathbb{C}_0 \bigtriangledown^s w. \tag{6.29}$$

Finally, before determining the value of $DJ(\rho,u(\rho))\tilde{\rho}$, it is required to find $D_\rho a(\rho,u(\rho),\lambda)$ as follows:

$$D_\rho a(\rho,u(\rho),\lambda)\tilde{\rho} = \int_\Omega \bigtriangledown^s \lambda \mathbb{C}'(\rho) \bigtriangledown^s u\tilde{\rho}. \tag{6.30}$$

As stated previously, note that $\bigtriangledown^s\lambda$ and $\bigtriangledown^s u$ represent the strains of the adjoint and state problems, respectively. Therefore, in the provided code implementation, these terms are obtained directly from the ABAQUS® FEA. Also, note that for the stress functional $D_\rho l(\lambda) = 0$. Finally, the first three optimization steps can then be rewritten as:

1. find u solution of $a(\rho,u(\rho)v) = l(\rho,v)$
2. using equation (6.27), find λ such that $a(\rho,\lambda,w) = -D_u J(\rho,u(\rho))w \;\forall\; w$. Note that this functional is not self-adjoint, leading to a different procedure than the one shown in the previous section.
3. compute $DJ(\rho,u(\rho))\tilde{\rho} = [D_\rho J(\rho,u(\rho)) \;+\; D_\rho a(\rho,u(\rho),\lambda)]\tilde{\rho}$

6.3 Discrete formulation of topology optimization problem statements and sensitivities

In this section, the topology optimization problem statement and sensitivity analysis defined in section 6.2 are now presented in their discretized versions. This information is included here to bridge the gap between the continuous framework shown in section 6.2 and the implementation used in the code provided with this work, allowing an easier understanding of both theoretical and implementation fundamentals of a topology optimization method.

6.3.1 Topology optimization problem

The discrete version of problem (6.1) results in:

$$\min_\rho : J(\rho,u(\rho)) \tag{6.31}$$

subject to:

$$\rho_{min} \leq \rho_e \leq 1 \;\forall\; e = 1,...,N \tag{6.32}$$

$$V(\rho) = \sum_{\rho} \rho_e v_e \leq V^* \tag{6.33}$$

and when considering a maximum stress constraint, also subject to:

$$\sigma^{PN}(\rho) \leq \sigma^* \tag{6.34}$$

where $u(\rho)$ is the solution of the state problem:

$$F = K(\rho)u. \tag{6.35}$$

$J(\rho)$ is the cost function; F and u are the force and displacement vectors, and $K(\rho)$ is the stiffness matrix. $K(\rho)$ and F can be defined as:

$$K(\rho) = \int_{\Omega} B_a^T \mathbb{C} B_a dx \tag{6.36}$$

$$F = \int_{\Omega} N\ f\ dx + \int_{\partial\Omega} N\ t \cdot n\ dx \tag{6.37}$$

with N representing a linear shape function, f a generic load function, and B_a the strain-displacement matrix in the evaluation point a. $V(\rho)$ is the total volume, with v_e representing the volume of element e when its design density is 1, and V^* is the maximum value of the volume constraint. $\sigma^{PN}(\rho)$ represents the maximum stress and σ^* its maximum allowable value.

Note that the penalization factors and regularization process described in section 6.2.2 are not changed.

6.3.2 Sensitivity analysis

The gradient of the cost function can be defined as:

$$\bigtriangledown_\rho J = \frac{\partial J}{\partial \rho} + \frac{\partial J}{\partial u}\frac{\partial u}{\partial \rho}. \tag{6.38}$$

In order to find $\frac{\partial J}{\partial u}$, we can reorganize the state equation, multiply it for a generic vector v and derive the expression, leading to:

$$v^T \left[\frac{\partial K(\rho)}{\partial \rho} u(\rho) - \frac{\partial F(\rho)}{\partial \rho} + K(\rho)\frac{\partial u(\rho)}{\partial \rho} \right] = 0 \tag{6.39}$$

therefore:

$$-\left(\frac{\partial u(\rho)}{\partial \rho}\right)^T K^T(\rho)\ v = v^t \left[\frac{\partial K(\rho)}{\partial \rho} u - \frac{\partial F(\rho)}{\partial \rho} \right]. \tag{6.40}$$

As stated in section 6.2, solving equation (6.40) for all values of ρ would be too expensive, motivating the use of the adjoint method. Defining the adjoint variable as $\lambda = v$ such that $K^T(\rho)\lambda = -\frac{\partial J}{\partial u}$, we get:

$$-\left(\frac{\partial u(\rho)}{\partial \rho}\right)^T K^T(\rho)\lambda = \left(\frac{\partial u}{\partial \rho}\right)^T \frac{\partial J}{\partial u} = \lambda^T \left[\frac{\partial K(\rho)}{\partial \rho}u - \frac{\partial F(\rho)}{\partial \rho}\right]. \tag{6.41}$$

Thus, we can finally rewrite the gradient as:

$$\bigtriangledown_\rho J = \frac{\partial J}{\partial \rho} + \frac{\partial J}{\partial u}\frac{\partial u}{\partial \rho} = \frac{\partial J}{\partial \rho} + \lambda^T \left[\frac{\partial F(\rho)}{\partial \rho} - \frac{\partial K(\rho)}{\partial \rho}u\right]. \tag{6.42}$$

With this information, we can rewrite the equivalent three step process described in section 6.2.3:

- find u such that: $K(\rho)u = F(\rho)$
- find λ such that: $K(\rho)\lambda = -\frac{\partial J}{\partial u}$
- compute: $\bigtriangledown_\rho J = \frac{\partial J}{\partial \rho} + \lambda^T \left[\frac{\partial F(\rho)}{\partial \rho} - \frac{\partial K(\rho)}{\partial \rho}u\right]$

6.3.3 Compliance functional

The compliance can be defined as $C(\rho) = F\ u(\rho)$ and its sensitivity can be determined as follows [336–338]:

$$\frac{\partial C}{\partial \rho_e} = F\frac{\partial u}{\partial \rho_e} \tag{6.43}$$

since $u(\rho) = K^{-1}(\rho)F$ and $\frac{\partial u}{\partial \rho_e} = -K^{-1}\frac{\partial K}{\partial \rho_e}K^{-1}F$, equation (6.43) becomes:

$$\frac{\partial C}{\partial \rho_e} = -FK^{-1}\frac{\partial K}{\partial \rho_e}K^{-1}F = -u\frac{\partial K}{\partial \rho}u. \tag{6.44}$$

Considering that $K = A\int_{\Omega_e} B_a^T \mathbb{C} B_a\ dx$, with A being the assembly operator, and that ρ is constant in Ω_e, we can write $\frac{\partial K}{\partial \rho}$ as follows:

$$\frac{\partial K}{\partial \rho} = \int_{\Omega_e} B_a^T \mathbb{C}' B_a\ dx = \int_{\Omega_e} B_a^T P\rho^{P-1}\mathbb{C}_0 B_a\ dx = \int_{\Omega_e} \frac{P}{\rho} B_a^T \mathbb{C} B_a\ dx = \frac{P}{\rho}K. \tag{6.45}$$

Introducing equation (6.45) in (6.44) leads to:

$$\frac{\partial C}{\partial \rho_e} = -\frac{P}{\rho}\rho^P u_e^T K_0 u_e = -P\frac{E_e}{\rho} \tag{6.46}$$

where u_e and K_0 are the elemental displacement vector and stiffness matrix of a fully solid element (i.e. $\rho = 1.0$). The term $\rho^P u_e^T K_0 u_e$ is the strain energy (E_e), missing only the $\frac{1}{2}$ constant. However, because this constant is applied to all elements, it can be neglected and set the term $\rho^P u_e^T K_0 u_e$ equal to the strain energy (E_e) automatically calculated in ABAQUS®.

In this particular case, the first three steps of the generic optimization process can be simplified and rewritten as follows:

- find u such that: $K(\rho)u = F(\rho)$. This procedure can be done using an ABAQUS® FEA.
- take $\lambda = -u$, since the problem is self-adjoint.
- compute: $\frac{\partial C}{\partial \rho_e} = -P\frac{E_e}{\rho}$. Note that E_e can be obtained from the ABAQUS® FEA executed in the first step.

This information is included to allow an easier understanding of the code implementation and its correlation with the formal mathematical formulation.

6.3.4 Stress functional

The maximum function is approximated by means of a modified p-norm function. This approximation is necessary to provide a derivable function that approximates the maximum function, which is non-differentiable. This implementation adopts the modified p-norm approximation proposed in [333]:

$$\sigma^{PN}(\rho) = \left(\frac{1}{N_i} \sum_{\Omega} \left(\sigma_a^{vM}(\rho)\right)^q \right)^{\frac{1}{q}} \tag{6.47}$$

where q is the exponential factor, Ω is the set of stress evaluation points in the topology optimization problem, and σ_a^{vM} is the value of the amplified von Mises stress in point a.

The derivative of the p-norm approximation with respect to the design density of an element can be obtained by the chain-rule, multiplying three intermediate terms. The first term is the derivative of the p-norm approximation w.r.t the amplified von Mises stress:

$$\frac{\partial \sigma^{PN}(\rho)}{\partial \sigma_a^{vM}} = \left(\frac{1}{N_i} \sum_{\alpha \in \Omega} \left(\sigma_a^{vM}(\rho)\right)^q \right)^{\frac{1}{q}-1} \frac{1}{N_i} \left(\sigma_a^{vM}(\rho)\right)^{q-1}. \tag{6.48}$$

The second term is the derivative of σ_a^{vM} w.r.t. the stress vector in point a. Since σ_a^{vM} can be written in matrix form as $\sigma_a^{vM} = (\sigma_a M \sigma_a)^{\frac{1}{2}}$, its derivative becomes:

$$\frac{\partial \sigma_a^{vM}(\rho)}{\partial \sigma_a(\rho)} = \left(\sigma_a(\rho) M \sigma_a(\rho)\right)^{-\frac{1}{2}} \times \sigma_a(\rho) M \frac{\partial \sigma_a(\rho)}{\partial \rho}. \tag{6.49}$$

The third and last term, is the derivative of the stress vector σ_a w.r.t. the design density. Considering equations (6.9) and (6.10):

$$\frac{\partial \sigma_a(\rho)}{\partial \rho} = \beta \rho^{\beta-1} \mathbb{C}_0 B_a u(\rho) + \rho^{\beta} \mathbb{C}_0 B_a \frac{\partial u(\rho)}{\partial \rho}. \tag{6.50}$$

The term $\frac{\partial u(\rho)}{\partial \rho}$ is obtained from the state equation (6.35):

$$\begin{aligned} \frac{\partial K(\rho)}{\partial \rho} u(\rho) + K(\rho)\frac{\partial u(\rho)}{\partial \rho} = \frac{\partial F(\rho)}{\partial \rho} = 0 \Leftrightarrow \\ \Leftrightarrow \frac{\partial u(\rho)}{\partial \rho} = -K^{-1}(\rho)\frac{\partial K(\rho)}{\partial \rho} u(\rho) \end{aligned} \tag{6.51}$$

and, therefore, with $\beta = \frac{1}{2}$:

$$\frac{\partial \sigma_a(\rho)}{\partial \rho} = \frac{1}{2}\rho^{-\frac{1}{2}}\mathbb{C}_0 B_a u(\rho) - \rho^{\frac{1}{2}}\mathbb{C}_0 B_a K^{-1}(\rho)\frac{\partial K(\rho)}{\partial\, \rho} u(\rho). \tag{6.52}$$

Note that it is assumed that $\frac{\partial F(\rho)}{\partial \rho} = 0$. This assumption is valid for static load-driven problems, where the load applied is constant and independent of the material distribution. For displacement-driven problems, this assumption is not valid, as the forces resulting from displacement applied will change depending on the material distribution.

With equations (6.48) through (6.51) and applying the chain rule, it is possible to define $\frac{\partial \sigma^{PN}(\rho)}{\partial \rho}$ as:

$$\begin{aligned} \frac{\partial \sigma^{PN}(\rho)}{\partial \rho} = \frac{\partial \sigma^{PN}(\rho)}{\partial \sigma_a^{vM}} \frac{\partial \sigma_a^{vM}(\rho)}{\partial \sigma_a} \frac{\partial \sigma_a(\rho)}{\partial \rho} = \\ = \frac{\partial \sigma^{PN}(\rho)}{\partial \sigma_a^{vM}} \frac{\partial \sigma_a^{vM}(\rho)}{\partial \sigma_a} \left(\frac{1}{2}\rho^{-\frac{1}{2}}\mathbb{C}_0 B_a u(\rho) - \rho^{\frac{1}{2}}\mathbb{C}_0 B_a K^{-1}(\rho)\frac{\partial K(\rho)}{\partial \rho} u(\rho) \right). \end{aligned} \tag{6.53}$$

Renaming these two terms as $\partial \sigma_{spf}^{PN}(\rho)$ (equation (6.54)), referring to the component of the derivative that is dependent on the stress penalization factor, and $\partial \sigma_u^{PN}(\rho)$ (equation (6.55)), referring to the component of the derivative that is dependent on the nodal displacement:

$$\partial \sigma_{spf}^{PN}(\rho) = \frac{\partial \sigma^{PN}(\rho)}{\partial \sigma_a^{vM}} \frac{\partial \sigma_a^{vM}(\rho)}{\partial \sigma_a} \left(\frac{1}{2}\rho^{-\frac{1}{2}}\mathbb{C}_0 B_a u(\rho) \right) \tag{6.54}$$

$$\partial \sigma_u^{PN}(\rho) = \frac{\partial \sigma^{PN}(\rho)}{\partial \sigma_a^{vM}} \frac{\partial \sigma_a^{vM}(\rho)}{\partial \sigma_a} \left(-\rho^{\frac{1}{2}}\mathbb{C}_0 B_a K^{-1}(\rho)\frac{\partial K(\rho)}{\partial\, \rho} u(\rho) \right). \tag{6.55}$$

Notice that $\partial \sigma_{spf}^{PN}(\rho)$ can be easily determined, since the only information required is the stress vector ($\hat{\sigma}_a(\rho) = \mathbb{C}_0 B_a u(\rho)$) and the design densities. $\frac{\partial \sigma^{PN}(\rho)}{\partial \sigma_a^{vM}}$ can be determined with a simple summation, and $\frac{\partial \sigma_a^{vM}(\rho)}{\partial \sigma_a}$ through the product of two vectors and a matrix.

The term $\partial \sigma_u^{PN}(\rho)$, on the other hand, requires the explicit definition of the matrices B_a, $K^{-1}(\rho)$ and $\frac{\partial K(\rho)}{\partial \rho}$, which are dependent on the element formulation used in the numerical model. Furthermore, for topology optimization problems where the number of design variables is larger

than the number of constraints, the most efficient way to determine $\partial\sigma_u^{PN}(\rho)$ is using an adjoint model, defining the adjoint variable as:

$$\begin{aligned} K(\rho)\lambda &= \frac{\partial\sigma^{PN}(\rho)}{\partial\sigma_a^{vM}}\frac{\partial\sigma_a^{vM}(\rho)}{\partial\sigma_a}\mathbb{C}_0 B_a \Leftrightarrow \\ \Leftrightarrow \lambda &= \frac{\partial\sigma^{PN}(\rho)}{\partial\sigma_a^{vM}}\frac{\partial\sigma_a^{vM}(\rho)}{\partial\sigma_a}\mathbb{C}_0 B_a K^{-1}(\rho). \end{aligned} \tag{6.56}$$

Therefore, the adjoint variable can be extracted from a finite element analysis, where the load applied on each node is equal to $\frac{\partial\sigma^{PN}(\rho)}{\partial\sigma_a^{vM}}\frac{\partial\sigma_a^{vM}(\rho)}{\partial\sigma_a}\mathbb{C}_0 B_a$, and λ is equal to the node displacement. Introducing the adjoint variable in equation (6.54) then leads to:

$$\partial\sigma_u^{PN}(\rho) = \frac{\partial\sigma^{PN}(\rho)}{\partial\sigma_a^{vM}}\frac{\partial\sigma_a^{vM}(\rho)}{\partial\sigma_a}\left(-\rho^{\frac{1}{2}}\lambda\frac{\partial K(\rho)}{\partial\,\rho}u(\rho)\right). \tag{6.57}$$

Finally, since K can be defined as $K = AB_a^T\mathbb{C}(\rho)B_a$, since $B_a\lambda$ and $B_a u$ are equal to the deformation vectors of the adjoint and state models (ξ_{adj} and ξ_s, respectively), equation (6.57) can be simplified to:

$$\partial\sigma_u^{PN}(\rho) = \frac{\partial\sigma^{PN}}{\partial\sigma_a^{vM}}\frac{\partial\sigma_a^{vM}}{\partial\sigma_a}\left(-\rho^{\frac{1}{2}}\xi_{adj}^T(\rho)\mathbb{C}'(\rho)\xi_s(\rho)\right). \tag{6.58}$$

With this information, the first three steps of the generic optimization process can be rewritten as follows:

- find u such that: $K(\rho)u = F(\rho)$. This procedure can be done through an ABAQUS® FEA.
- find λ such that: $K(\rho)\lambda = \frac{\partial\sigma^{PN}(\rho)}{\partial\sigma_a^{vM}}\frac{\partial\sigma_a^{vM}(\rho)}{\partial\sigma_a}\mathbb{C}_0 B_a$
- compute: $\frac{\partial\sigma^{PN}(\rho)}{\partial\rho} = \partial\sigma_{spf}^{PN}(\rho) + \partial\sigma_u^{PN}(\rho)$, using $\xi_{adj} = B_a\lambda$ and equation (6.58) to find $\partial\sigma_u^{PN}(\rho)$.

As demonstrated in equations (6.55) through (6.58), determining $\partial\sigma_u^{PN}(\rho)$ is significantly more complex and computationally expensive, due to the necessity of using the adjoint model. Some researchers [339] have proposed approximating this term as $\partial\sigma_u^{PN}(\rho) = 0$, in an attempt to reduce the complexity and computational cost of stress dependent topology optimization problems. While doing so does have evident advantages, the results obtained for each element of the case study described in section 6.7.2 show that, on average, $|\partial\sigma_u^{PN}(\rho)| > |\partial\sigma_{spf}^{PN}(\rho)|$. As a result, assuming that $\partial\sigma_u^{PN}(\rho) = 0$ would lead to an average error larger than 50% in the calculation of the maximum stress sensitivity, at least for the particular case of the benchmark problem described in section 6.7.2. The error in this approximation is aggravated by the fact that $\partial\sigma_u^{PN}(\rho)$ has a negative sign from equation (6.53), and, as a consequence, ignoring this term could potentially point the optimization

algorithm in the opposite direction of the gradient. Therefore, the simplification proposed in [339] is not adopted in this work.

6.3.5 Volume constraint

The value of the sensitivity of the volume constraint to changes in the design density of each element is equal to the value of its volume, as defined in equation (6.59). Note that the equivalent sensitivity can be obtained for a mass constraint, replacing the volume of the element with its mass.

$$\frac{\partial V(\rho)}{\partial \rho_e} = V_e. \tag{6.59}$$

6.4 Mesh-dependency and data filtering

In order to obtain a mesh-independent solution and avoid the "checkerboard" instability, the raw sensitivities and/or design densities are processed with a blurring filter [340, 148]. This work adopts the filter scheme used in [331], which is a simplification of the scheme proposed by Huang and Xie [341], described as follows:

$$\rho_e = \sum_j \left(\frac{w(r_{ej})}{\sum_j w(r_{ej})} \rho_j \right) \tag{6.60}$$

where the value of $w(r_{ej})$ is equal to the difference between the maximum range of the filter (r_{max}) and the actual distance between the central element e and the j elements in its neighbourhood (r_{ej}), defined as follows:

$$w(r_{ej}) = max(0, r_{max} - r_{ej}). \tag{6.61}$$

This parameter can be interpreted as a pondered measurement of how close the two elements are. Note that the code implementation provided allows the user to select if the blurring filter should be applied to the sensitivities (excluding volume constraint sensitivities, or equivalent), design densities, or both.

6.5 Optimization algorithms

6.5.1 Optimality criteria - OC

The OC method implemented in this research follows the approach proposed by Bendsøe [338] and implemented by Sigmund in [148]. It is usually applied to problems of compliance minimization

with a volume constraint. According to this procedure, the design variables can be updated according to the following expression:

$$\rho = \begin{cases} \rho_{lower}, & \text{if } \rho B_e^{\eta} \leq \rho_{lower} \\ \rho B_e^{\eta}, & \text{if } \rho_{lower} \leq \rho B_e^{\eta} \leq \rho_{upper} \\ \rho_{upper}, & \text{if } \rho_{upper} \leq \rho B_e^{\eta} \end{cases} \tag{6.62}$$

$$\rho_{lower} = max(\rho_{min}, \rho - \delta\rho) \tag{6.63}$$

$$\rho_{upper} = min(1.0, \rho + \delta\rho) \tag{6.64}$$

where ρ_{lower} and ρ_{upper} represent the design densities move-limits imposed by the parameter $\delta\rho$. η is a numerical damping coefficient set equal to 0.5 in accordance with [148]. B_e is a parameter obtained from the optimality condition, described as the ratio between the gradient of the objective and constraint functions divided by the Lagrangian multiplier (ψ), as follows:

$$B_e = \frac{-\frac{\partial C}{\partial \rho}}{\psi \frac{\partial V}{\partial \rho}} = \frac{-\frac{\partial C}{\partial \rho}}{\psi V_e}. \tag{6.65}$$

The Lagrangian multiplier ψ can be found using a bisection method, updating its value until the volume of the solution meets the imposed constraint.

This method can also be simplified in order to obtain an equivalent version suitable for discrete design variables. This simplification corresponds to the approach used by Zuo and Xie in [331]. For the current iteration, the code determines the sensitivities of all elements. Then, a bisection method is used to determine a threshold sensitivity. All elements with a sensitivity larger than the threshold become solid (i.e. $\rho = 1.0$), while the other elements are removed according to the "soft-kill" approach (i.e. $\rho = \rho_{min}$). Similarly to the continuous version, the value of the sensitivity threshold is updated until the mass or volume of the structure meets the volume constraint imposed.

When using the OC method, especially the discrete version, it is usual to consider a fully solid design for the initial iteration, and gradually reducing the volume constraint in each iteration until the target value is reached. This procedure is adopted to obtain a convergent solution, and can be defined through the following equation:

$$V^{k+1} = max(V^*, V^k(1.0 - e_{vol})) \tag{6.66}$$

where k is the number of current iteration, and e_{vol} is the ratio at which the volume constraint decreases in each iteration, until the target volume constraint (V^*) is reached.

Nonetheless, it is important to understand that this discrete version should be regarded as an heuristic, that was observed to work well when solving compliance minimization problems. However, this procedure can be criticized from an optimization perspective, as there is no mathematical guarantee that the objective function will decrease during each iteration.

6.5.2 Method of moving asymptotes - MMA

The MMA is a non-linear programming method that can be applied to structural optimization problems. Its functioning is based on an iterative process, generating a series of strictly convex approximating sub-problems. As an input to generate the sub-problem approximation, the MMA takes the value of the objective function, value of the constraints, and respective gradients as inputs. Then, using dual methods such as [342, 343], the sub-problem is solved and the solution is taken as the value of the design variables in the next iteration of the main optimization problem [105].

The popularity of the application of the MMA to topology optimization problems is justified by several factors. From a computational point of view, the use of an approximation sub-problem and gradient information provides a means to reduce the number of structural problems solved, which are usually expensive. Furthermore, the versatility of the method and capability of handling multiple types of constraints make it suitable and applicable to a wide range of cases, from simple compliance minimization problems with a single volume constraint, to cases where multiple constraints with different natures are applied. Its stability is also a positive characteristic, even for initial non-fully solid designs. Due to its extension and complexity, the interested reader is referred to [105] for a detailed explanation of this method.

The present code establishes a connection between ABAQUS® and the implementation of the MMA created by Deetman [344] ('mmasub' and 'subsolve' functions of the code provided), which is a Python version of the MATLAB code created by Svanberg [345].

6.5.3 Sequential Least-Squares Programming - SLSQP

SLSQP is a sequential least-squares programming algorithm that utilizes the Quasi-Newton method to obtain an optimal solution. The strategy behind this algorithm consists on approximating the objective function with a quadratic equation, and using the minimum of the approximation function as a possible solution in the next iteration. To improve the efficiency, the SLSQP can use Quasi-Newton methods to approximate the objective function Hessian.

The code provided uses the SLSQP implementation available in the SciPy module [346], which is a library of numerical routines for the Python programming language that provides fundamental building blocks for modeling and solving scientific problems.

6.5.4 Trust-constr

Trust-constr is a trust-region algorithm for constrained optimization, available in the SciPy module [346]. Both line search methods and trust-region methods generate steps with the help of a quadratic model of the objective function. However, while line-search methods use a step length along a given search direction, trust-region methods define a region around the current iteration, within which they trust that the quadratic model is an accurate approximation [347].

This algorithm utilizes the trust-region interior point method described in [348] to solve the inequality constraints imposed in the topology optimization problem statement. The strategy behind this interior point method [348] consists of solving inequality constraints by introducing slack

variables. Then, the process is repeated for a sequence of equality-constrained barrier problems with progressively smaller values of the barrier parameter.

6.6 Python implementation and usage

The following sub-sections present a brief guide on how to use the Python code provided in appendix C, and available in digital format in Dataset [332] or repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS. A brief overview of the most important details of the main code classes and functions is also included in these sub-sections. The detailed information of each class and function can be found in their respective doc-string.

This code can be called in ABAQUS® using the *Run Script* command in the *File* tab or by copying it into the command line. Note that the Run Script command loads the code faster but automatically assumes that the user intends to solve the topology optimization in its totality. Copying it into the command line leads to a slower input but allows a line-by-line execution, which may be useful to understand the functioning of the code implementation.

The PEP8 [80] Python style guide was followed as closely as possible. However, the interested reader is warned that some exceptions exist, caused by compatibility constraints with the data structures and pre-defined variables existing in ABAQUS®. An example of these exceptions includes the naming of constant material properties without full-capital letters, since these names are already used by ABAQUS® in a different data structure.

6.6.1 Code usage

Upon executing the code, ABAQUS® will create a series of prompt dialogue boxes. This allows the user to select the model database file (.cae), structural part, material, problem statement, optimization algorithm, and the internal parameters to be considered during the topology optimization.

Besides being fully functional, the ABAQUS® model does not require any specific formatting, except for the following considerations:

- Within the same part, the user can define specific elements to be optimized, dividing the part in two regions: one with editable elements, and another with elements that should not be included in the optimization process (i.e. passive elements, from this point onward referred to as 'frozen region' or 'frozen elements'). To do so, the user should create a set named 'editable_elements', containing the elements to be edited. If this set does not exist, the whole part will be optimized. Note that the name 'editable_elements' is case sensitive. This is done in order to promote the optimization only of the targeted area and also to reduce computational cost.

- The region to be optimized should not have a material section assigned to it, as the code will automatically assign the material properties and sections directly to the elements. Therefore,

if the user defined a set with the editable elements, only the elements not included (frozen elements) should have a material section assigned to them.

- It is possible to allow the frozen elements to be considered during the filtering operations. To do so, the user should create a second element set with the case sensitive name 'neighbouring_region'.

- If multiple copies of the part to be optimized are included in the model assembly, only the first copy will be optimized.

6.6.2 Model formatting, job submission, and sensitivities (lines 28-3452)

The *ModelPreparation* class modifies the ABAQUS® model file (.cae) in order to allow the assignment of material properties and material sections to each individual element, as well as the property update during each iteration. Furthermore, this class requests the output of the strain energies, external work, and the node strains if solving a stress dependent problem. In ABAQUS®, the strain energies of geometrically non-linear problems are stored in the variables "SENER" and "PENER", respectively for the elastic and plastic strain components. Note that, according to the literature [349, 328, 341, 337, 331], the sensitivity of each element in a geometrically nonlinear compliance minimization problem is given by the sum of the elastic strain (E_e^e) and plastic strain components (E_e^p), as described in equation (6.67). However, for geometrically linear problems, the plastic strain component is zero, and ABAQUS® stores the strain energies in the variable "ESEDEN". The external work is only requested for a critical analysis of the result, as it should be equal to the objective function considering no energy dissipation.

$$\frac{\partial C}{\partial \rho} = E_e^e + E_e^p. \tag{6.67}$$

The submission of the FEA simulations for the state and adjoint models is managed by the classes *AbaqusFEA* and *AdjointModel*, respectively. *AbaqusFEA* is also responsible for determining the sensitivity of the compliance objective function $\left(\frac{\partial C}{\partial \rho}\right)$, which is stored as the class attribute 'ae'. On the other hand, the *AdjointModel* class also applies the adjoint loads and determines the stress sensitivity. The stress sensitivity $\left(\frac{\partial \sigma^{PN}(\rho)}{\partial \rho}\right)$, and its most relevant intermediate terms ($\partial \sigma_{spf}^{PN}(\rho)$ and $\partial \sigma_u^{PN}(\rho)$) are stored as attributes of this class, along with the stress and strain vectors. In particular:

- 'elmt_stress_sensitivity_continuous' stores the values of $\frac{\partial \sigma^{PN}(\rho)}{\partial \rho}$ in a mesh-independent and mesh-independent form, making more suitable the comparison of the value of this derivative with the results obtained through finite differences.

- 'elmt_stress_sensitivity_discrete' stores the values of $\frac{\partial \sigma^{PN}(\rho)}{\partial \rho}$ in a mesh-dependent form, which is more suitable to be used as an input to optimization algorithms, since they generally do not have access to information of the mesh used in the FEA model. The difference

between 'elmt_stress_sensitivity_discrete' and 'elmt_stress_sensitivity_continuous' is the multiplication by the determinant of the Jacobian matrix.

- 'd_pnorm_spf' stores the values of $\partial \sigma_{spf}^{PN}(\rho)$.
- 'd_pnorm_displacement' the values of $\partial \sigma_{u}^{PN}(\rho)$.
- 'stress_vector' and 'stress_vector_int' store the stress vectors determined at the element nodes and integration points.
- 'deformation_vector' and 'deformation_vector_int' store the strain vectors determined at the element nodes and integration points.

The sensitivity of the material constraint is determined by the *material_constraint_sensitivity* function, accounting for the possibility of having a non-uniform mesh with elements of different sizes, and the use of either a volume or mass constraint.

6.6.3 Material and stress constraints (lines 3453-3607)

The values of the material and maximum stress constraints are determined by the *MaterialConstraint* class and the *stress_constraint_evaluation* function, respectively. The maximum stress value is approximated with the modified p-norm function described in equation (6.47), which is determined by the *p_norm_approximation* function.

6.6.4 Data filtering (lines 3608-3846)

The blurring filter is defined by the *DataFilter* class, which determines how each element is influenced by its neighbourhood. The neighbourhood of each element is defined by the elements that are fully within a maximum search radius defined by the user. The code lines 3760-3762, intentionally left commented to reduce computational cost, generate element sets that allow an easy visual interpretation of the neighbourhood of each element.

6.6.5 Optimization algorithms: OC, MMA, SLSQP, and Trust-constr (lines 3847-5622)

The present code provides five optimization algorithms, described in section 6.5, that can be used to solve the topology optimization problem.

The discrete and continuous version of the OC algorithm, implemented in the functions *oc_discrete* and *oc_continuous*, are only suitable for compliance minimization problems. The remaining optimization algorithms are suitable for all problem statements included.

The *mma* function is a decorator that links ABAQUS® to the *mmasub* and *subsolv* functions implementated by Arjen Deetman [344]. Therefore, the main purpose of the *mma* function is the processing of the inputs necessary to apply the MMA [345] as a function of the problem statement selected by the user.

The functioning of the *oc_discrete*, *oc_continuous* and *mma* functions is managed by the main program loop, described in section 6.6.11. However, this is not the case for the SLSQP and Trust-constr algorithms available in this code, as they belong to the SciPy module [346]. To be fully functional, the SLSQP and Trust-constr algorithms require the additional freedom to decide when and how many times to call the objective, sensitivity, and constraint functions. Furthermore, the problem statement must be defined using unique data structures. For these reasons, class *SciPyOptimizer* is responsible for:

- Calling the SLSQP and Trust-constr methods from the SciPy module [346].
- Re-defining the problem statement with the correct data structure required by the algorithm selected.
- Providing the freedom for these algorithms to call the objective, sensitivity, and constraint functions as many times and in any given order necessary.
- Trying to record the data of each iteration as accurately as possible. However, it is expectable that the data recorded may include information from intermediate evaluation points.

Finally, there are two details that should be considered before using the SLSQP and Trust-constr algorithms. One is that both have internal convergence criteria. To change them, the user may be required to edit the code provided with this manuscript. The second detail is that the initial solution must be feasible, otherwise the optimization process will stop. This is particularly relevant when defining the initial material distribution and/or when using a P-norm continuation approach (described in section 6.6.11). The OC and MMA methods implemented do not have these restrictions for the initial guess.

6.6.6 Display definition (lines 5623-6207)

The *SetDisplay* class is capable of changing the color codes of the element sets. In compliance minimization problems, this allows the user to plot a gray-scale representation of the material distribution. In stress dependent problems, the user has the additional options of plotting the distribution of the element stress or element amplified stress, both of which can be raised to the P-norm exponential factor. The element stress plotted is equal to the average of the stresses determined at the integration points of the element.

The graphic representation(s) plotted at each iteration are automatically saved to a .png file.

To allow an easier interpretation of 3D material distributions, the *SetDisplay* class has a built-in method (function of a class) named *hide_elements*, whose input is a design variable threshold value. This method will hide all elements whose design variable is below the threshold input.

6.6.7 Data recording (lines 6208-6349)

At the end of each iteration, the *save_data* function will create a text file that records the values of all variables necessary to describe the current solution, as well as the internal parameters selected by the user.

This text file can also be used to restart the topology optimization process, starting from any iteration. However, as mentioned in section 6.6.5, note that the algorithms SLSQP and Trust-constr may require the current iteration to be feasible with respect to the constraints imposed.

At the end of the optimization process, the *save_mdb* will create a separate ABAQUS® .cae file with the final solution obtained and a record of the objective function and material constraint values.

6.6.8 Element formulation and stiffness matrix (lines 6350-7283)

To solve stress dependent problems, it is necessary to have access to information that depends on the element formulation (as shown in section 6.3.4), such as the B_a, K, $\mathbb{C}$ and Jacobian matrix, and the definition of the element shape functions and their derivatives. The code provided includes the information required to use one 2D element with 4 nodes and one 3D element with 8 nodes. The 2D element is referred in ABAQUS® by the code 2DQ4, and more specifically as CPS4 or CPE4 for plane-stress and plane-strain cases, respectively. The 3D element is referred in ABAQUS® by the code C3D8. The information of these elements is included in the *ElementFormulation* class, while the stiffness matrix $\mathbb{C}$ is created by the *c_matrix_function* function.

Although not used in the present research, the *ElementFormulation* class also includes the formulation of a shell element (referenced by the code S4 in ABAQUS®). This information was included in order to promote the development of topology optimization methodologies suitable to this particular type of element.

6.6.9 Parameter input request, domain definition, and variable generation (lines 7284-8854)

The *ParameterInput* class generates the prompt boxes that appear when running the code. The information introduced by the user is then used to create the global variables required for the code. A detailed list of these variables and their purpose can be found in lines 8366-8461.

This information is then used by the *EditableDomain* class, to define the design space of the topology optimization problem, and by the *VariableGenerator* class, to create the lists and dictionaries that record the relevant data gathered during the topology optimization process. For a detailed description of these variables, please refer to the code lines 8721-8827.

6.6.10 Auxiliary functions (lines 8855-9013)

Finally, there are 4 auxiliary functions to be briefly mentioned.

- *average_ae* outputs the average of the compliance sensitivity of each element over the last three iterations. This leads to an easier convergence of the OC algorithms, especially when considering discrete variables.
- The *update_past_info* function updates the variables that record the design variables and compliance sensitivity used in the previous two iterations.
- The *evaluate_change* function creates a ratio that describes how the objective function has changed over the last 10 iterations, which is used as a convergence criterion in the implementation provided.
- The *remove_files* function will allow an automatic removal of the temporary files created by ABAQUS® after each FEA.

6.6.11 Main program (lines 9014-9307)

The main program is divided in two phases. Between lines 9014-9110, the code creates the classes required for the problem statement selected and prepares the ABAQUS® model accordingly. The optimization process and the convergence criteria are defined in lines 9111-9307.

The optimization process considers two loops. The first is applicable for stress dependent problems, allowing a continuous increase of the P-norm exponential factor between Qi and QF. This functionality, here referred to as "P-norm continuation approach" allows the stress constraint to be applied gradually and may be useful to improve the convergence of the algorithm. The continuation approach contrasts with the constant approach, which only considers one constant value for the P-norm factor. Note that for stress independent problems (where Qi is not used but set to 1.0) and when using the constant approach, $Qi = QF$, leading to a single loop.

The second loop represents the convergence criterion. The code assumes that the algorithm has converged if the objective function has not changed more than 0.1 % over the last 10 iterations. It was selected as the default convergence criterion to allow an easy and relatively fast recreation of the results detailed in this chapter, as well as for being similar to the criterion adopted in [331]. The interested user is reminded that this criterion does not apply to the SLSQP and Trust-constr algorithms (as mentioned in section 6.6.5) and is encouraged to create a convergence criterion adapted to the topology optimization problem studied.

6.7 Topology optimization case studies

6.7.1 Cantilever beam case study

The purpose of including this case study is to validate functioning of the compliance objective function and volume constraint implemented in the code provided, before addressing more complex stress dependent problems, such as the L-bracket detailed in section 6.7.2.

This problem was modelled using a regular mesh with an element size of 5.0 mm and the element type CPS4. The load F is equal to 100.0 N applied in a single node, in the corner represented

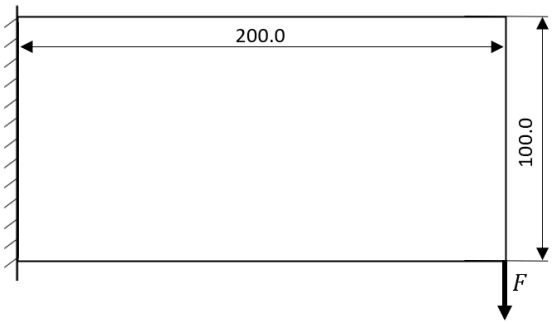

Figure 6.1: Dimensions and boundary conditions of the Cantilever beam numerical model.

in figure 6.1. The problem was solved using an implicit analysis and considering a material with 70 GPa Young's modulus, 0.33 Poisson's ratio, and a material density of 2.7×10^{-9} ton/mm^3.

The ABAQUS® models necessary to reproduce all the results described in this work are available in the Dataset [332] or repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS.

6.7.2 L-bracket case study

The L-bracket case study was chosen as a more challenging benchmark problem [333, 350, 185], characterized by its initial geometry containing a stress-concentration point in the internal corner. Unconstrained compliance minimization problems tend to ignore the stress concentration point, leading to an angular shape. On the other hand, stress minimization or stress constrained compliance minimization problems tend to create rounded shapes that avoid the stress concentration. For these reasons, the L-bracket case study will be used in this research to validate and demonstrate the functioning of the code implemented when solving stress-dependent topology optimization problems. Furthermore, the strain and stress state generated by this L-bracket geometry is used in section 6.8 to validate the correct implementation of the element formulations and maximum stress differentiation process.

The dimensions and boundary conditions considered are represented in figure 6.2. Unless stated otherwise, this problem was modelled using a regular mesh with an element size of 3.0 mm and the element type CPS4. The load F is equal to 1500.0 N, with this value being distributed over the nodes included in the 12.0 x 12.0 corner represented in figure 6.2. The elements within the 12.0 x 12.0 corner are not editable during the optimization process. The problem was solved using an implicit analysis and considering a material with 70.000 MPa Young's modulus, 0.33 Poisson's ratio, a material density of 2.7×10^{-9} ton/mm^3.

The ABAQUS® models necessary to reproduce all the results described in this work are available in the Dataset [332] or repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS.

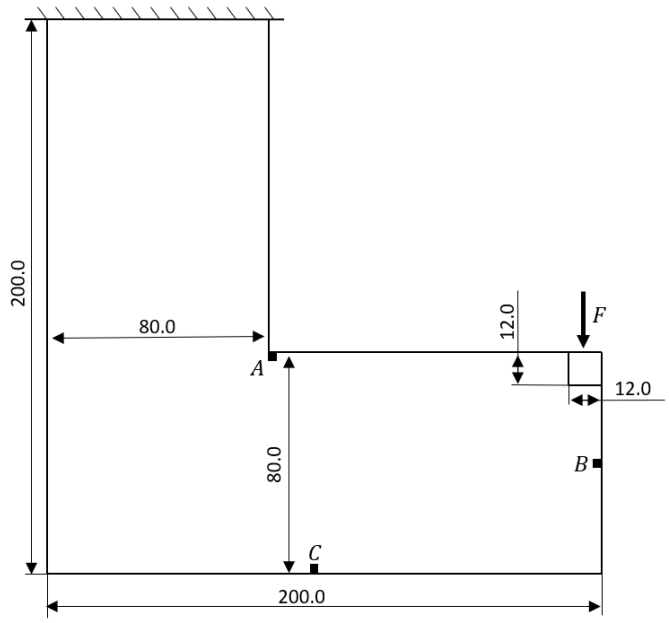

Figure 6.2: Dimensions and boundary conditions of the L-bracket numerical model.

6.8 Code validation

The data necessary to solve compliance minimization problems can be directly obtained from ABAQUS®, which is a commercially certified software. However, stress dependent problems required the explicit programming of the element formulation and of the derivation process of the maximum stress function. The purpose of the following subsections is to present a brief validation of the code implementation. Section 6.8.1 validates element formulations implemented, comparing the strains and stresses determined by the code and by ABAQUS® at each integration point. Section 6.8.2 compares the derivative of the maximum stress determined by the code with the derivative obtained through finite differences.

In this section, the L-bracket case study described in section 6.7.2 is used. The reason behind this choice is that the L-bracket problem leads to a complex strain and stress state, allowing a more challenging and generic testing of elements over a wider variety of loading conditions than the Cantilever beam problem.

6.8.1 Validation of the element formulation

The analysis of the element formulation is based on the measurement of a pondered error, described as follows:

$$E_{rr}^{i} = \delta\gamma_i \frac{\gamma i}{|\gamma|}. \tag{6.68}$$

Where E_{rr}^{i} is the pondered error determined for the component i of a generic vector γ, and $\delta\gamma_i$ is the difference between the component determined by ABAQUS® and its code counterpart. E_{rr}^{i} is

Table 6.1: Average pondered error between the elements implemented in the code and the ABAQUS® output. The dimensions are represented in millimeters.

	CPS4		CPE4		S4		C3D8	
	E^i_{rr} (%)	Int. Points E^i_{rr}<1% (%)	E^i_{rr} (%)	Int. Points E^i_{rr}<1% (%)	E^i_{rr} (%)	Int. Points E^i_{rr}<1% (%)	E^i_{rr} (%)	Int. Points E^i_{rr}<1% (%)
ξ_{11}	0,05	99,65	0,90	63,39	0,49	97,29	0,44	96,37
ξ_{22}	0,15	99,55	0,97	63,31	0,44	97,60	0,44	96,37
ξ_{33}	-	-	-	-	-	-	0,44	96,37
ξ_{12}	0,04	99,51	0,04	99,51	1,95	32,45	<0,01	100,0
ξ_{13}	-	-	-	-	-	-	<0,01	100,0
ξ_{23}	-	-	-	-	-	-	<0,01	100,0
σ_{11}	0,00	99,88	1,85	31,74	0,61	95,89	1,45	44,81
σ_{22}	0,15	99,65	2,00	63,41	0,61	97,29	1,45	96,36
σ_{33}	-	-	-	-	-	-	1,45	44,82
σ_{12}	0,01	99,55	0,01	99,55	0,77	77,44	<0,01	100,0
σ_{13}	-	-	-	-	-	-	<0,01	100,0
σ_{23}	-	-	-	-	-	-	<0,01	100,0

determined as the product of $\delta\gamma_i$ by the ratio between the component γ_i and the magnitude of the vector γ.

This metric is preferred over the direct comparison of $\delta\gamma_i$, since the latter is highly influenced by floating point errors. One of the reasons for this influence is that, while ABAQUS® operates internally with double precision, the output received by the code is in a single precision format. This difference reduces the number of decimal places considered in each value and could lead to disproportional artificial errors in the vector components that contribute the least to the magnitude of the vector or in vectors whose magnitude tends towards zero. The second reason is that $\delta\gamma_i$ tends to estimate large error values for vector components that have a significantly smaller magnitude when compared to the magnitude of the vector.

Table 6.1 summarizes the average pondered error observed for each integration point of four variants of the L-bracket numerical model described in section 6.7.2, each with a different element type. For the particular case of using a 3D element (element type C3D8 in ABAQUS®), a thickness of 1.0 mm was considered for the L-bracket geometry. Since the largest average pondered error observed is equal to 2.0 %, it can be concluded that the element formulations included in the code provided have been implemented successfully for the case study selected. Furthermore, for each element type, the table indicates the percentage of integration points with a pondered error smaller than 1.0 %.

Although not included in the present work due to its extension, the interested reader is informed that the Dataset [332] and repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS associated with this research include a detailed comparison of the every integration point, considering both the pondered and regular error measurements.

Table 6.2: Analysis of the continuous maximum stress derivative determined by the Python code and the derivative obtained through finite differences.

Mesh size	Number of elements	P-norm (MPa) @ ρ = 1.0	Element location	Element Label	$\Delta\rho$	P-norm (MPa) @ ρ = 1-$\Delta\rho$	Finite differences	Continuous derivative
0,5	102400	314,27741	A	69184	0,1 0,01	335,12679 315,23590	-208,4939 -95,8493	-129,6819
			B	102332	0,1 0,01	314,27740 314,27741	0,0000 0,0000	0,0001
			C	63841	0,1 0,01	314,28104 314,27784	-0,0363 -0,0438	-0,0301
1	25600	272,96255	A	17269	0,1 0,01	272,87736 272,95502	0,8519 0,7534	0,4426
			B	25560	0,1 0,01	272,96249 272,96255	0,0006 0,0000	0,0004
			C	23901	0,1 0,01	272,97007 272,96326	-0,0752 -0,0713	-0,0685
3	2889	221,08948	A	1953	0,1 0,01	232,36352 221,48312	-112,7404 -39,3637	-65,2413
			B	2875	0,1 0,01	221,08923 221,08941	0,0025 0,0066	0,0029
			C	2683	0,1 0,01	221,13180 221,09332	-0,4232 -0,3842	-0,4054

6.8.2 Validation of the maximum stress derivative

To validate the stress norm derivative of the code provided, its result was compared with the derivative obtained through finite differences. To do so, the design density of the elements located in points A, B, and C (shown in figure 6.2) were changed individually, and the value of the maximum stress approximation was used to apply the finite differences. This procedure was repeated for the three elements, considering three different mesh sizes of 3.0 mm, 1.0 mm, and 0.5 mm, each increasing the order of magnitude of the total number of elements by 1, and different design density changes of 0.1, and 0.01. The results obtained are summarized in Table 6.2.

It can be observed that the results obtained through finite differences are in good agreement with the derivative obtained by the Python code implemented, especially when the mesh size and the design density decrease.

Table 6.3 indicates the average values of $\frac{\partial \sigma^{PN}(\rho)}{\partial \rho}$ and its components $\partial \sigma^{PN}_{spf}(\rho)$ and $\partial \sigma^{PN}_{u}(\rho)$ observed in the L-bracket model as a function of the element mesh size and design density. This data indicates that the approximation proposed in [339] (described at the end of section 6.3.4) should not be adopted for the present case study.

Table 6.3: Average value of the stress derivative, and its components, observed in the L-bracket model at different design density values, considering three different mesh sizes.

Mesh size	ρ	$\frac{\partial \sigma^{PN}(\rho)}{\partial \rho}$	$\partial \sigma^{PN}_{spf}(\rho)$	$\partial \sigma^{PN}_{u}(\rho)$	$\frac{\lvert\partial \sigma^{PN}_{u}(\rho)\rvert}{\lvert\partial \sigma^{PN}_{spf}(\rho)\rvert}$
0,5	1	-0,0088	0,0004	-0,0092	24,037
	0,5	-2304,3703	0,5426	-2304,9129	4248,295
	0,1	-90,9833	1,2132	-92,1965	75,996
1	1	-0,0305	0,0013	-0,0318	23,864
	0,5	-7949,4335	1,8849	-7951,3184	4218,445
	0,1	-313,8380	4,2147	-318,0527	75,462
3	1	-0,2140	0,0096	-0,2236	23,289
	0,5	-3,4697	0,1086	-3,5784	32,936
	0,1	-2206,1706	30,3676	-2236,5382	73,649

6.9 Topology optimization results

Figure 6.3 summarizes the results obtained for the compliance minimization of the Cantilever beam. Figure 6.4 summarizes the results obtained for the topology optimization of the L-bracket considering the different problem statements, plotting the objective function at each iteration and illustrating one of the final geometries obtained.

The results obtained for each problem statement and any particular choice of parameters is discussed in the following subsections. In common, these examples share the volume constraint $V^* = 0.5$. In the particular case of the solution "OC - Discrete and decreasing", the initial density is set to 1.0 and gradually reduced by a factor $e_{vol} = 0.05$, while the remaining cases consider an initial density equal to 0.5. The maximum move-limit is equal to 0.2 for the OC or MMA and 1.0 for the SciPy algorihtms SLSQP or Trust-constr, corresponding to their pre-default values. The blurring filter was applied to both sensitivity and design densities with a maximum search radius of 8.0 mm for the L-bracket and 12.0 mm for the Cantilever beam, except for the discrete version of the OC, which only applied the blurring filter to the sensitivity. The influence of the frozen elements was also considered during the application of the blurring filter. For stress-dependent problem statements, a P-norm exponential factor $Qi = 8.0$ was used.

Overall, the Trust-constr and SLSQP algorithms tend to require a larger number of iterations to reach convergence. The reason for this difference is justified by their internal convergence criteria, as described in section 6.6.11.

In the results shown, the existence of few elements with intermediate design densities is, generally, to be expected for two reasons. First, applying a blurring filter will cause a gradual transition between the solid and void regions, avoiding a full "black-and-white" solution. Second, as described in section 6.6.11, the convergence criterion depends only on the values of the objective function and does not impose a strict restriction on the final solution being constituted only by solid elements. These two factors justify the apparent better performance of the discrete version of the OC, since it only uses solid elements and did not have a blurring filter applied to the element densities. However, it is most likely that this optimization method converged to a better local

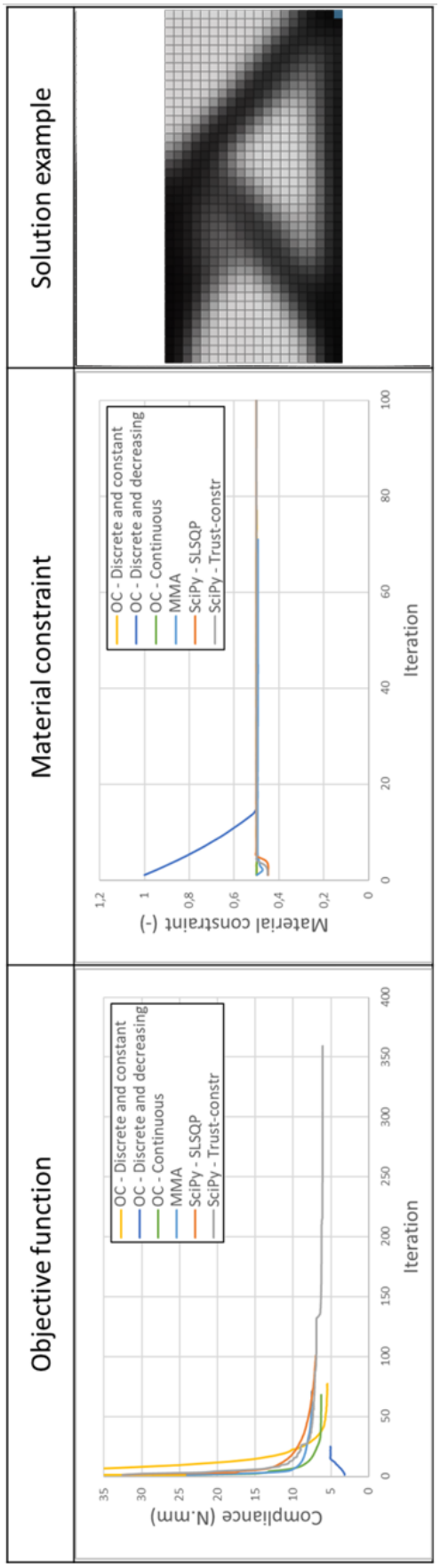

Figure 6.3: Graphic representation of the objective function and volume constraint obtained for the compliance minimization of the cantilever beam. The geometry displayed represents the final solution obtained by the MMA algorithm.

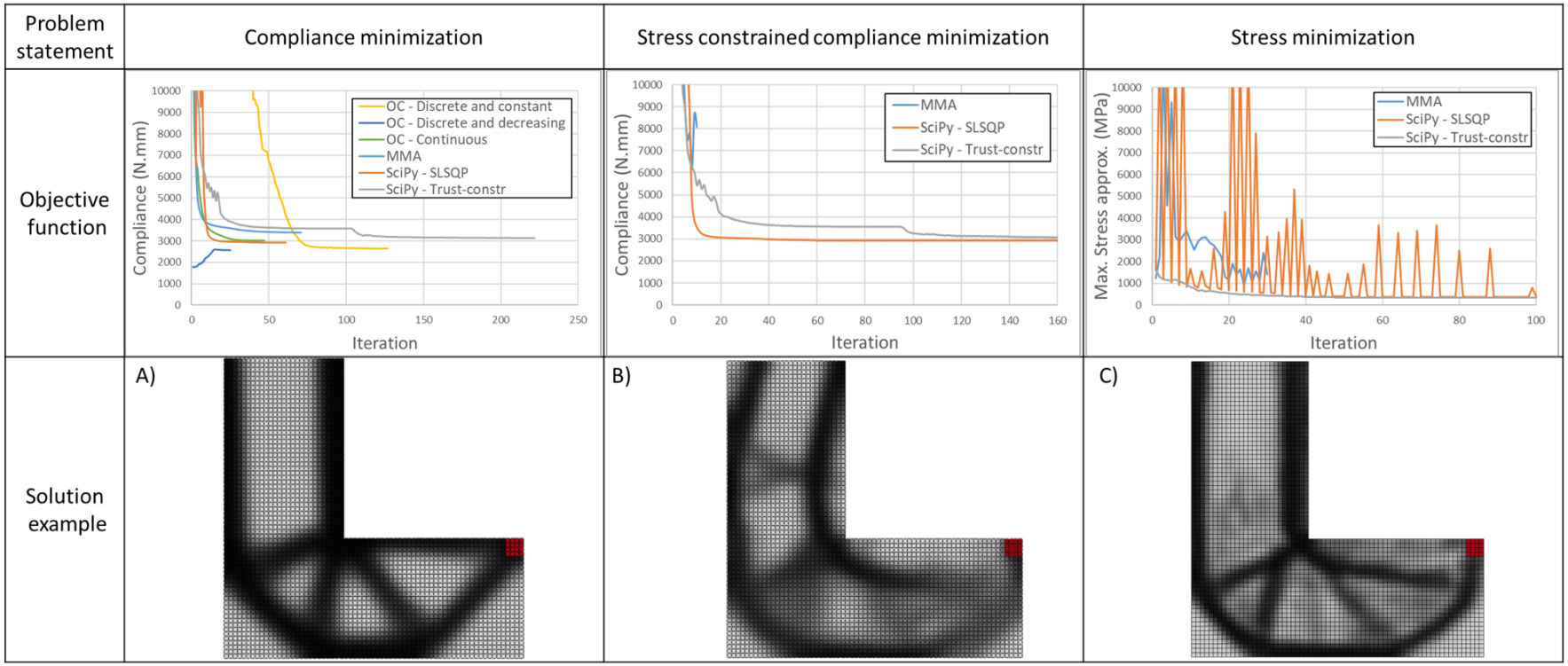

Figure 6.4: Graphic representation of the objective function obtained for each optimization algorithm and problem statement. The geometries displayed represent the final solution obtained by the algorithms: A) OC - Continuous, B) MMA, C) SciPy - SLSQP.

minimum.

Nevertheless, it is also relevant to note that the mesh size has an influence regarding the use of elements with intermediate design densities.

Finally, the problems solved in these sections could also be replicated in the 3D space. This is possible as the code provided includes the implementation of a 3D element (C3D8 in ABAQUS®) and as the problem solving process does not require any particular adaptation when being converted to the 3D space.

The interested reader is informed that the Dataset [332] and repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS contain all the solutions obtained by each optimization algorithm. Due to its large extension, the information presented in this section is limited to the most relevant solutions.

6.9.1 Cantilever beam: compliance minimization results

The objective function obtained by each of the different optimization algorithms are in good agreement, converging towards an approximately equal value of compliance. However, a different trend can be observed for the "OC - discrete and decreasing". In this particular case, the initial solution considers a fully solid design that must gradually reduce its mass to meet the volume constraint, causing a consequent reduction of the structural stiffness and an increase of the compliance value.

This observation is also in line with the volume constraint graphic, which shows a gradual decrease until the value 0.5 is reached. On the other hand, the remaining algorithms maintained the volume constraint within a feasible domain.

The final cantilever geometry obtained is similar across all cases considered and illustrated, in figure 6.3, with the solution found by the MMA. In this particular case, the elements with intermediate densities may also be the result of using a coarser mesh.

6.9.2 L-bracket: compliance minimization results

The solutions obtained for the compliance minimization problem are similar across all the algorithms used, differing only on the use of intermediate design densities in some elements. The solution obtained by the continuous version of the OC is shown in figure 6.4 A) as an example of this geometry.

The volume constraint was respected across all iterations. This behaviour was also observed for the problem statements presented in section 6.9.3 and section 6.9.4. For this reason, the graphic representation of the constraints was not included in these sections.

6.9.3 L-bracket: stress constrained compliance minimization results

The stress constrained compliance minimization problem statement considered a maximum allowable stress $\sigma^* = 200.0$ MPa. Imposing this constraint causes the MMA to converge towards a curved geometry, shown in figure 6.4 B), avoiding the creation of a stress concentration point in the inner corner of the L-bracket. This transition is in agreement with the results presented in [333].

It is important to note the transition from an angular geometry to a curved geometry (from case A to B of figure 6.4) is dependent on several factors and not guaranteed to occur. First, the use of the modified P-norm approximation (equation (6.47) proposed in [333]) may lead to an underestimate of the maximum stress installed. This fact may cause the optimization algorithm to overlook the influence of the stress concentration point, and justifies the selection of $\sigma^* = 200.0$ MPa, a relatively low maximum value for the stress constraint. Second, the algorithm selected may adopt a strategy that is more or less prone to explore and accept solutions that do not respect the imposed constraints. The solution obtained by the SLSQP algorithm has a slightly curved shape closer to the region where the load is applied but does not avoid the internal angular geometry. The solution obtained by the Trust-constr region is equal to the one obtained in the compliance minimization problem statement, as the implementation available in SciPy only determines the maximum stress sensitivity for the first iteration. Although this strategy reduces the computational cost, it does not allow a rigorous evaluation of the influence of each element in the maximum stress.

Finally, it is also important to note that the P-norm exponential factor has a significant influence in the result obtained, especially in the use of elements with intermediate densities.

6.9.4 L-bracket: stress minimization results

Defining a stress minimization problem statement may also lead to a transition from an angular to a curved geometry. This is particularly evident in the solution obtained by the SLSQP algorithm, shown in figure 6.4 C).

Observing the graphic in figure 6.4, it is noticeable that the SLSQP has a more unstable evolution of the objective function than the Trust-constr algorithm. Evaluating the solutions created by these algorithms at each iteration, it becomes evident that the SLSQP was more likely to perform larger density re-distributions than the Trust-constr algorithm, which may be a result of the strategy adopted by the SciPy implementation of these algorithms, and a result of the influence of their internal parameters, which largely affect the solutions obtained.

Similarly to the previous problem statement, the P-norm exponential factor has a significant influence in the result obtained, especially in the use of elements with intermediate densities. Also, although the MMA is removing material from the stress concentration area, further research should be done to find appropriate tolerance parameters.

6.10 Conclusions

This paper presents the implementation of a Python code capable of solving topology optimization problems, whose finite element analysis are executed in ABAQUS®. This implementation allows the user to apply the Optimility Criteria, Method of Moving Asymptotes, Sequential Least Square Quadratic Programing, and Trust-constr algorithms to solve compliance minimization, stress constrained compliance minimization, and stress minimization problems.

This represents a relevant contribution for the field of topology optimization. On one hand, it is the first code to include the validated tools necessary to solve stress-dependent problems, which is

a significant obstacle that requires an extensive work to be overcome and could potentially hinder further research. This is particularly true due to the complexity involved in solving the adjoint problem in ABAQUS®. On the other hand, it is a contribution particularly relevant to the large community of ABAQUS® users, allowing a better understanding, interpretation, and control over an optimization process that is often hidden behind a black-box, in commercially available FEA codes.

The code (appendix C), as well as the ABAQUS® models necessary to recreate the work reported in this paper, can be downloaded from following Dataset [332] or repository https://github.com/pnfernandes/Python-Code-for-Stress-Constrained-Topology-Optimization-in-ABAQUS. Although this content is intended mainly for educational purposes, the modularity of the code allows for an easy extension of its functionalities and applicability to different optimization cases and engineering design problems.

Chapter 7

Discussion

7.1 On the design of composite elastic hinges

The research reported in this thesis studied the design and optimization of composite deployable structures. The engineering problem presented by ESA in [6] was considered as a benchmark case study, as it is a representative example of an up-to-date design challenge.

Through the state of the art review, detailed in chapter 2, it was possible to identify two main challenges in the development of deployable structures. The first is the balancing of flexibility and stiffness requirements, to allow the structure to contract without compromising its functionality during operation. The second is accounting for the influence of relaxation phenomenon that may occur in the period between the storage and the deployment of the structure once it is in orbit.

Both of these challenges were addressed in chapters 3 and 4 using the information already available in the literature, leading to two observations. On one hand, this information was sufficient to enable an accurate prediction of the deployment behavior of a composite elastic hinge considering the influence of the relaxation phenomenon. On the other hand, the attempt made at designing and optimizing a composite elastic hinge with the requirements defined in [6] was not successful. Although it can be argued that the requirements were ambitious compared to other state-of-the-art solutions [10, 6], it is important to identify the limitations of the method adopted and propose possible solutions.

The design variables of the parametric optimization used in chapter 3 allowed the arrangement and combination of different geometrical shapes, which would define the geometry of the cut-out of the elastic hinge. Despite the agreement with the literature, it was observed that the achievable shapes did not allow the removal of material in the stress concentration points. A parametric analysis then made evident that this limitation was excluding a relevant part of the design space, hindering the optimization algorithm from generating geometries that could further reduce the Max. IF observed without significant reductions of the first natural frequency. Therefore, a possible point of improvement is the redefinition of the design variables. Within the scope of parametric optimization, the redefinition of the design variables is limited to including additional geometric shapes or to using a set of coordinate points to the create a contour line (as implemented in chapter 5). Nonetheless, the topology optimization method has a much larger potential of overcoming this limitation, as each element of the FEM could be removed. Therefore, with a sufficiently refined mesh, the topology optimization method could virtually explore all possible design geometries.

Having identified the limitations of the design approach adopted, it is time to discuss the design requirements set [6]. The first natural frequency requirement is directly related to the operating conditions of the antenna of the telecommunication satellite. Therefore, it is not reasonable to propose changes to its value, as doing so could limit the telecommunication technology being used in the satellite or reduce its functionality. On the other hand, the same generalization cannot be applied to the Max. IF requirement, which imposes the elastic hinge to function in the elastic regime of the composite material. The elastic hinge should be capable of transitioning between two configurations: one contracted configuration that is used to store inside the satellite, and a deployed configuration that meets the operation requirements. The deployment, during which the

transition between both configurations occurs, should be repeatable and predictable. While having the contraction, deployment, and operation of the elastic hinge occurring in the elastic regime of the composite material does promote the repeatability and predictability of the process, it is not a necessary condition. Damage initiating during the contraction process does not necessarily imply an unpredictable and unreliable deployment sequence, nor a catastrophic failure of the component. Furthermore, the literature review reported in chapter 2 indicates the prediction of residual properties as a relevant topic of research. Based on this information, it is possible that this requirement may be relaxed, allowing the design of a damage-tolerant elastic hinge. Doing so has the potential of leading to a design with increased stiffness, and consequently higher first natural frequency, that will initiate damage during the contraction process. If the damage initiation is controlled and has a limited propagation, the global stiffness increase may compensate for the local stiffness decreased caused by the initiation of damage. In other words, a damage-tolerant elastic hinge design may allow a better balancing of both requirements. Furthermore, the increased stiffness may even reach more demanding stiffness requirements, such as a higher first natural frequency or an improved pointing accuracy of the antenna.

In summary, it was possible to identify two possible solutions that may improve the quality of the design process of composite elastic hinges: the use of the topology optimization method, and the use of a damage-tolerant design. Each of these topics is further discussed in the following sections. Nonetheless, it is relevant to mention the development and potential use of data-driven methods to design this component. Although not explored in this research, the information gathered during the optimization processes may be useful to find correlations between the design variables and identify unexplored designs.

7.2 On the use of a damage-tolerant elastic hinge design

The possibility of adopting a damage-tolerant elastic hinge design was studied in chapter 5. Two elastic hinges were designed through the same parametric optimization method, one constrained to function in the elastic regime of the composite material (Max. IF$\leq$ 1.0), while the other allowed the initiation of damage (Max. IF$\leq$ 1.1). Through an FEA that estimated the first natural frequency of vibration of both designs considering the stiffness reduction caused by the damage initiation, it was concluded that the damage-tolerant design had a better performance, improving the first natural frequency by 15 %. The result confirmed, at least for one design, the hypothesis proposed at the beginning of the research: the global stiffness increase compensated the local stiffness decrease caused by the initiation of damage.

Given that the damage-tolerant design is less conservative than the elastic design, the possibility of damage propagation was also considered. It was assumed a gradual damaging of material, affecting all elements that had a Max. IF$\geq$ 0.90. This analysis led to the conclusion that the performance of both damage-tolerant and elastic design would only be equal if the damage propagated to all material with a Max. IF$\geq$ 0.91.

Nonetheless, some questions need to be addressed before the actual implementation of a damage-tolerant design. The first concern for the space industry is the release of debris during deployment. A simple solution to this issue is to use a containment system, such as a sleeve that covers the elastic hinge, thereby preventing debris from being released. The second concern is the deployment behaviour, which was not addressed in this research. However, it is likely such a damage-tolerant elastic hinge would store sufficient internal energy to deploy. This conclusion can be drawn from the definition of the natural frequency (equation (7.1)):

$$N_f = \sqrt{\frac{K}{m}} \tag{7.1}$$

The mass of the system (represented by the mass matrix m) is mostly defined by the mass of the antenna, which is much larger than the mass of any composite possible elastic hinge design. Therefore, it can be approximated that, in this particular application, the first natural frequency depends only on the stiffness of the elastic hinge (represented by the stiffness matrix K). Let us also assume that an elastic hinge behaves similarly to a spring system (equation (7.2)), whose potential energy (U_p) is equal to:

$$U_p = \frac{1}{2} K u^2 \tag{7.2}$$

And that the equivalent displacement (u) is approximately equal in both designs. Given that the stiffness of the damage-tolerant design is larger than the elastic design, it can be expected that the same will occur for the internal energy stored during the contraction of both designs. As a consequence, if any deployment repeatability or reliability issue is to arise from the use of a damage-tolerant design, it will most likely be from the excess of energy, which could potentially cause overshooting. In turn, the overshooting can be mitigated by a controlled release system or even by the influence of the relaxation phenomenon.

Finally, it is important to acknowledge that the damage-tolerant design will always be less conservative than the elastic design. However, the new philosophy for cubesat constellations [286–289] aims at a more frequent renewal of the satellites in orbit, serving as a means of updating the technology being used. This leads to a relevant synergy with the damage-tolerant design, as it naturally addresses the need for a long-term design and compensates for the use of a less conservative approach.

In summary, despite still needing further research, these observations suggest that the use of a damage-tolerant design may be a promising approach to design composite elastic hinges with very demanding and ambitious requirements.

7.3 On the use of topology optimization to design composite elastic hinges

As described by Bendsøe and Kikuchi in 1988 [85], topology optimization is one of the three sub-fields of structural optimization, amongst size and shape optimization, and aims at determining the optimal distribution of material [19, 128]. The method defines a design region, divided into several finite elements, to be occupied by the structural component. According to an objective function, the optimization method adjusts the density of each element, defining which elements should have material and which should not [129].

Recalling the problem statement described in chapter 6, a compliance minimization problems subject to a material constraint can be mathematically described as follows:

$$\min_{\rho} : C(\rho) = F^T \, u = u^T \, K \, u \tag{7.3}$$

subject to:

$$\rho_{min} \leq \rho \leq 1.0 \tag{7.4}$$

$$F = K \, u \tag{7.5}$$

$$\int_{\Omega} \rho \, dx \leq V^* \tag{7.6}$$

and when considering stress constraints, also subject to:

$$\sigma^{PN}(\rho) \leq \sigma^* \tag{7.7}$$

Since compliance is the inverse of the stiffness, the problem statement shown can be interpreted as the maximization of stiffness. As discussed in section 7.2, the mass of the elastic hinge is negligible when compared to the mass of the antenna, therefore, maximizing its first natural frequency can be approximated as maximizing its structural stiffness. As a consequence, it becomes evident that one of the most common problem statements used in topology optimization is suitable to the engineering problem of designing an elastic hinge, in terms of the objective function. Furthermore, choosing between an elastic or damage-design could theoretically be achieved by defining a more, or less, conservative value for the stress constraint considered. If the stress constraint is lower than the stress limit of the material used, the result of the topology optimization process would be an elastic design, and vice-versa.

Nonetheless, there is a subtle limitation that arises during the derivation of the stress constraint, which potentially hinders the applicability of the topology optimization method to the design of elastic hinges. The issue arises when deriving the force F with respect to the design variables, in the state equation (7.5). In the literature [333], it is possible to find the assumption that this derivative is equal to 0 (shown in equation (6.51)). Doing so implies that the external forces applied are constant

and independent of the material density. However, this assumption is not valid for displacement-driven numerical models, since the force that results from the displacement imposed differs as a function of the material density, which is the case of the numerical models used to simulate the elastic hinges in chapters 3 through 5, where the displacement-driven boundary condition allowed a better control of the movement simulated. Theoretically, it is possible to replace the imposed displacement with a variable load, decreasing its magnitude as the elastic hinge contracts, but it leads to an additional challenge. The load applied will result in a different displacement depending on the stiffness of the elastic hinge, which will change during the topology optimization process. Therefore, it is probably wiser to break down this case into several topology optimization problems, each starting with the elastic hinge topology obtained previously and with a load that is constant but lower than the one applied in the last problem. Doing so allows a finer control of the resulting displacement at the cost of an increased computational cost. From an engineering point of view, the necessity of fine-tuning the load after each intermediate optimization problem is a significant disadvantage for being time-consuming, case-specific, and dependent on the criteria of the user. For this reason, it is considered that the approximation made during the derivation of the stress constraint hinders the applicability of the topology optimization method to the design of elastic hinges.

From a mathematical point of view, using the topology optimization method to the design of an elastic hinge poses a decision: determining the derivative of the forces applied with respect to the design variables, or accepting an approximation. The analytical derivative of the forces applied with respect to the design variables can be determined if there is a clear definition of the influence of the design density on the load applied. A simple example of this case would be the consideration of the gravitational force, which would be proportional to the mass of the element, and consequently to the design density. If this relation is unknown, reformulating the state equation as a dynamic problem allows the redefinition of the force applied on each element as the product between the element mass and its acceleration. Doing so evidences the influence of the design density on the resulting load applied through the mass term. The possibility of accepting an approximation, in specific cases, may allow the applicability of the implementation proposed in this research as it is. However, it is important to note that no evaluation has been made on the error of this approximation. A brief comment is made regarding the approach proposed by S. Ferraro in [29], in which it is assumed that the displacement observed in the numerical model does not depend on the design variables. This approximation avoids the need for the use of an adjoint model, removes any terms that depend on the element formulation from the derivative, and is not dependent on the type of boundary condition applied, whether a load or a displacement. On the other hand, the results observed in section 6.8.2, in Table 6.3, indicate that this approximation would lead to an error of at least 50 % in the value of the stress derivative when applied to the L-bracket case study, considering a load-driven model. Therefore, when applied to a displacement-driven model, it is expectable to see an increase in the error and consequent reduction in its suitability, despite the benefits in the computational cost.

Chapter 8

Conclusions and Future Work

The research developed with this thesis focused on the design of composite deployable structures, reaching the following main conclusions and objectives:

- The material characterization of the composite system AS4/8552.
- The validation of numerical models, capable of simulating the functioning of a composite deployable structure, through the correlation of the strain data observed in the FEA and experimental tests.
- The design methodologies adopted to the development of deployable structures were reviewed, leading to the proposal of a damage-tolerant design approach.
- The evaluation of the damage-tolerant design approach, benchmarking it against the state-of-the-art methodologies.
- The topology optimization of a deployable structure through the use of a genetic algorithm.
- The implementation of a topology optimization code, written in Python, suitable for 2 and 3-dimensional problems.

Nonetheless, the topology optimization of deployable structures is a complex process that requires further development and maturing. To do so, there two main research paths that should be explored in future research:

- The validation of the damage-tolerant design approach proposed through extensive experimental testing, analysing the influence of the damage initiation and propagation on the performance of the deployable structure.
- The implementation of a topology optimization process compatible with the use of shell elements and with the use of displacement-driven models.

References

[1] L. Puig, A. Barton, and N. Rando. A review on large deployable structures for astrophysics missions. *Acta Astronautica*, 67(1-2):12–26, 2010.

[2] Gökhan Kiper and Eres Söylemez. Deployable space structures. In *RAST 2009 - Proceedings of 4th International Conference on Recent Advances Space Technologies*, pages 131–138, Istanbul, turkey, 2009.

[3] Serena Ferraro and Sergio Pellegrino. Self-deployable joints for ultra-light space structures. In *AIAA Spacecraft Structures Conference, 2018*, number 210019, pages 0694–0707, Kissimmee, Florida, 2018. AIAA Spacecraft Structures Conference.

[4] M Kroon, G Borst, M Grimminck, M Robroek, F Geuskens, and Airbus Defence. Articulated Deployment System for Antenna Reflectors. In *16th European Space Mechanisms and Tribology Symposium*, volume 2015, page 8, Bilbao, 2015.

[5] Geoffrey W. Marks, Michael T. Reilly, and Richard L. Huff. The Lightweight Deployable Antenna for the MARSIS Experiment on the Mars Express Spacecraft. In *36th Aerospace Mechanisms Symposium, Glenn Research Center*, pages 183–196, 2002.

[6] ESA. Statement of Work AO8702 - Antenna Deployment Arm with Integrated Elastic Hinges. Technical report, European Space Agency, 2016.

[7] Lin Tze Tan and Sergio Pellegrino. Thin-shell deployable reflectors with collapsible stiffeners: Experiments and simulations. *AIAA Journal*, 50(3):659–667, 2012.

[8] Davide Piovesan, Mirco Zaccariotto, Carlo Bettanini, Marco Pertile, and Stefano Debei. Design and validation of a carbon-fiber collapsible hinge for space applications: A deployable boom. *Journal of Mechanisms and Robotics*, 8(3):031007–031018, 2016.

[9] C. Wu and A. Viquerat. Natural frequency optimization of braided bistable carbon/epoxy tubes: Analysis of braid angles and stacking sequences. *Composite Structures*, 159:528–537, 2017.

[10] H. M.Y.C. Mallikarachchi and S. Pellegrino. Quasi-static folding and deployment of ultrathin composite tape-spring hinges. *Journal of Spacecraft and Rockets*, 48(1):187–198, 2011.

[11] ABAQUS User's Manual and I I Volume. Version 6.4. *Abaqus Inc*, 1080, 2003.

[12] S. D. Guest and S. Pellegrino. A new concept for solid surface deployable antennas. *Acta Astronautica*, 38(2):103–113, 1996.

[13] Z. You and S. Pellegrino. Cable-stiffened pantographic deployable structures part 1: Triangular mast. *AIAA Journal*, 34(4):813–820, 1996.

[14] K. A. Seffen and S. Pellegrino. Deployment dynamics of tape springs. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 455(1983):1003–1048, 1999.

[15] W. Szyszkowski, K. Fielden, and D. W. Johnson. Self-locking satellite boom with flexure-mode joints. *Applied Mechanics Reviews*, 50(11):S225–S231, 1997.

[16] Alan M. Watt. Deployable Structures with Self-locking Hinges, 2003.

[17] Wesley W Vyvyan. Self-actuating, self-locking hinge, jun 1968.

[18] H. M.Y.C. Mallikarachchi and S. Pellegrino. Deployment dynamics of composite booms with integral slotted hinges. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, page 2631, 2009.

[19] R. C. Alderliesten. Designing for damage tolerance in aerospace: A hybrid material technology. *Materials and Design*, 66(PB):421–428, 2015.

[20] M. McGugan, G. Pereira, B. F. Sorensen, H. Toftegaard, and K. Branner. Damage tolerance and structural monitoring for wind turbine blades. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2035):20140077, 2015.

[21] U. Zerbst, S. Beretta, G. Köhler, A. Lawton, M. Vormwald, H. Th Beier, C. Klinger, I. Černý, J. Rudlin, T. Heckel, and D. Klingbeil. Safe life and damage tolerance aspects of railway axles - A review. *Engineering Fracture Mechanics*, 98(1):214–271, 2013.

[22] S. Z.H. Shah, S. Karuppanan, P. S.M. Megat-Yusoff, and Z. Sajid. Impact resistance and damage tolerance of fiber reinforced composites: A review, 2019.

[23] Benliang Zhu, Xianmin Zhang, Min Liu, Qi Chen, and Hai Li. Topological and Shape Optimization of Flexure Hinges for Designing Compliant Mechanisms Using the Level Set Method. *Chinese Journal of Mechanical Engineering (English Edition)*, 32(1):1–12, 2019.

[24] Xinxing Tong, Wenjie Ge, Yonghong Zhang, and Zhenfei Zhao. Topology design and analysis of compliant mechanisms with composite laminated plates. *Journal of Mechanical Science and Technology*, 33(2):613–620, 2019.

[25] Shinnosuke Nishi, Kenjiro Terada, Junji Kato, Shinji Nishiwaki, and Kazuhiro Izui. Two-scale topology optimization for composite plates with in-plane periodicity. *International Journal for Numerical Methods in Engineering*, 113(8):1164–1188, 2018.

[26] F. Romano, A. Sorrentino, L. Pellone, U. Mercurio, and L. Notarnicola. New design paradigms and approaches for aircraft composite structures. *Multiscale and Multidisciplinary Modeling, Experiments and Design*, 2(2):75–87, 2019.

[27] Giulia E. Fenci and Neil G.R. Currie. Deployable structures classification: A review. *International Journal of Space Structures*, 32(2):112–130, 2017.

[28] Eric Harold Mansfield. Large-deflexion torsion and flexure of initially curved strips. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 334(1598):279–298, 1973.

[29] S. Ferraro and S. Pellegrino. Topology optimization of composite self-deployable thin shells with cutouts. In *AIAA Scitech 2019 Forum*, page 1524, 2019.

[30] C. Boesch, C. Pereira, R. John, T. Schmidt, K. Seifart, and J. M. Lautier. Ultra light self-motorized mechanism for deployment of light weight reflector antennas and appendages. In *European Space Agency, (Special Publication) ESA SP*, number SP-653, pages 7–9. Citeseer, 2007.

[31] Mark J. Silver, Jason D. Hinkle, and Lee D. Peterson. Modeling of snap-back bending response of doubly slit cylindrical shells. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 5, pages 3314–3327, 2004.

[32] J. C. Yee and S. Pellegrino. Composite Tube Hinges. *Journal of Aerospace Engineering*, 18(4):224–231, 2005.

[33] Peter A. Warren, Benjamin J. Dobson, Jason D. Hinkle, and Mark Silver. Experimental characterization of lightweight strain energy deployment hinges. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 1, pages 103–110, 2005.

[34] Ömer Soykasap. Deployment analysis of a self-deployable composite boom. *Composite Structures*, 89(3):374–381, 2009.

[35] M. Sakovsky, S. Pellegrino, and H. M.Y.C. Mallikarachchi. Folding and deployment of closed cross-section dual-matrix composite booms. In *3rd AIAA Spacecraft Structures Conference*, 2016.

[36] H.M.Y.C. Mallikarachchi and Sergio Pellegrino. Simulation of Quasi-Static Folding and Deployment of Ultra-Thin Composite Structures. In *49th AIAA / ASME / ASCE / AHS / ASC Structures, Structural Dynamics, and Materials Conference, 16th AIAA / ASME / AHS Adaptive Structures Conference, 10th AIAA Non-Deterministic Approaches Conference, 9th AIAA Gossamer Spacecraft Forum, 4th AIAA Multidisc*, page 2053, 2008.

[37] Florence Dewalque, Cédric Schwartz, Vincent Denoël, Jean Louis Croisier, Bénédicte Forthomme, and Olivier Brüls. Experimental and numerical investigation of the nonlinear dynamics of compliant mechanisms for deployable structures. *Mechanical Systems and Signal Processing*, 101:1–25, 2018.

[38] Mehran Mobrem and Douglas S. Adams. Deployment analysis of lenticular jointed antennas onboard the mars express spacecraft. *Journal of Spacecraft and Rockets*, 46(2):394–402, 2009.

[39] Kawai Kwok and Sergio Pellegrino. Shape recovery of viscoelastic deployable structures. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, page 2606, 2010.

[40] Kawai Kwok and Sergio Pellegrino. Viscoelastic effects in tape-springs. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, page 2022, 2011.

[41] A. Brinkmeyer, S. Pellegrino, P. M. Weaver, and M. Santer. Effects of viscoelasticity on the deployment of bistable tape springs. In *ICCM International Conferences on Composite Materials*, volume 2013-July, pages 370–380, 2013.

[42] Thomas W. Murphey, William H. Francis, Bruce L. Davis, Juan Mejia-Ariza, Matthew Santer, Joseph N. Footdale, Kevin Schmid, Omer Soykasap, Koorosh Guidanean, and Peter A. Warren. High strain composites. In *2nd AIAA Spacecraft Structures Conference*, 2015.

[43] Kawai Kwok and Sergio Pellegrino. Folding, stowage, and deployment of viscoelastic tape springs. *AIAA Journal*, 51(8):1908–1918, 2013.

[44] A. Brinkmeyer, S. Pellegrino, P. M. Weaver, and M. Santer. Effects of viscoelasticity on the deployment of bistable tape springs. *ICCM International Conferences on Composite Materials*, 2013-July(1):370–380, 2013.

[45] Thomas W. Murphey and Sergio Pellegrino. A novel actuated composite tape-spring for deployable structures. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 1, pages 260–270, 2004.

[46] Huina Mao, Pier Luigi Ganga, Michele Ghiozzi, Nickolay Ivchenko, and Gunnar Tibert. Deployment of Bistable Self-Deployable Tape Spring Booms Using a Gravity Offloading System. *Journal of Aerospace Engineering*, 30(4):04017007, 2017.

[47] Mehran Mobrem, Lee D. Peterson, Velibor Cormarkovic, and Farzin Montazersadgh. An evaluation of structural analysis methodologies for space deployable structures. In *4th AIAA Spacecraft Structures Conference, 2017*, page 851, 2017.

[48] H. M.Y.C. Mallikarachchi and S. Pellegrino. Optimized designs of composite booms with integral tape-spring hinges. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, 2010.

[49] H. M.Y.C. Mallikarachchi and S. Pellegrino. Design and validation of thin-walled composite deployable booms with tape-spring hinges. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, 2011.

[50] H. M.Y.C. Mallikarachchi and S. Pellegrino. Design of Ultrathin composite self-deployable booms. *Journal of Spacecraft and Rockets*, 51(6):1811–1821, 2014.

[51] H.M. Yasitha Chinthaka Mallikarachchi. Thin-walled composite deployable booms with tape-spring hinges, 2011.

[52] H. M.Y.C. Mallikarachchi and Sergio Pellegrino. Failure criterion for two-ply plain-weave CFRP laminates. *Journal of Composite Materials*, 47(11):1357–1375, 2013.

[53] H. M.Y.C. Mallikarachchi and S. Pellegrino. Deployment dynamics of ultrathin composite booms with tape-spring hinges. *Journal of Spacecraft and Rockets*, 51(2):604–613, 2014.

[54] D. Givois, J. Sicre, and T. Mazoyer. A low cost hinge for appendices deployment: Design, test and applications. In *European Space Agency, (Special Publication) ESA SP*, volume 480, pages 145–151, 2001.

[55] Ju Won Jeong, Young Ik Yoo, Dong Kil Shin, Jae Hyuk Lim, Kyung Won Kim, and Jung Ju Lee. A novel tape spring hinge mechanism for quasi-static deployment of a satellite deployable using shape memory alloy. *Review of Scientific Instruments*, 85(2):25001, 2014.

[56] Andrew J. Cook and Scott J.I. Walker. Experimental research on tape spring supported space inflatable structures. *Acta Astronautica*, 118:316–328, 2016.

[57] Florence Dewalque, Jean Paul Collette, and Olivier Brüls. Mechanical behaviour of tape springs used in the deployment of reflectors around a solar panel. *Acta Astronautica*, 123:271–282, 2016.

[58] Scott J.I. Walker and Guglielmo Aglietti. A study into the dynamics of three dimensional tape spring folds. *44th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 42(4):850–856, 2003.

[59] S. Hoffait, O. Brüls, D. Granville, F. Cugnon, and G. Kerschen. Dynamic analysis of the self-locking phenomenon in tape-spring hinges. *Acta Astronautica*, 66(7-8):1125–1132, 2010.

[60] Florence Dewalque, Pierre Rochus, and Olivier Brills. Importance of structural damping in the dynamic analysis of compliant deployable structures. *Proceedings of the International Astronautical Congress, IAC*, 8:5535–5547, 2014.

[61] Kawai Kwok and Sergio Pellegrino. Micromechanical modeling of deployment and shape recovery of thin-walled viscoelastic composite space structures. In *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference 2012*, page 1910, 2012.

[62] Michael E. Peterson and Thomas W. Murphey. Large deformation bending of thin composite tape spring laminates. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, page 1667, 2013.

[63] N.L. Hancox. *Engineering mechanics of composite materials*, volume 17. Oxford university press New York, 1996.

[64] John Whitcomb and Kanthikannan Srirengan. Effect of various approximations on predicted progressive failure in plain weave composites. *Composite Structures*, 34(1):13–20, 1996.

[65] Don O. Brush, Bo O. Almroth, and J. W. Hutchinson. Buckling of Bars, Plates, and Shells. *Journal of Applied Mechanics*, 42(4):911–911, 1975.

[66] Arnold M.A. Van Der Heijden. *W. T. Koiter's elastic stability of solids and structures*, volume 9780521515. Cambridge University Press Cambridge, UK:, 2008.

[67] Ö Soykasap, A. M. Watt, and S. Pellegrino. New deployable reflector concept. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 1, pages 683–693, 2004.

[68] Ö Soykasap, S. Pellegrino, P. Howard, and M. Notter. Folding large antenna tape spring. *Journal of Spacecraft and Rockets*, 45(3):560–567, 2008.

[69] Lin Tze Tan and Sergio Pellegrino. Thin-shell deployable reflectors with collapsible stiffeners part 1: Approach. *AIAA Journal*, 44(11):2515–2523, 2006.

[70] Robert Hooke and T. A. Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. *Journal of the ACM (JACM)*, 8(2):212–229, 1961.

[71] Fengfeng Li, Yanju Liu, and Jinsong Leng. Progress of shape memory polymers and their composites in aerospace applications. *Smart Materials and Structures*, 28(10):103003, 2019.

[72] Zhengxian Liu, Xin Lan, Wenfeng Bian, Liwu Liu, Qifeng Li, Yanju Liu, and Jinsong Leng. Design, material properties and performances of a smart hinge based on shape memory polymer composites. *Composites Part B: Engineering*, 193:108056, 2020.

[73] Thanh Duc Dao, Ngoc San Ha, Nam Seo Goo, and Woong Ryeol Yu. Design, fabrication, and bending test of shape memory polymer composite hinges for space deployable structures. *Journal of Intelligent Material Systems and Structures*, 29(8):1560–1574, nov 2018.

[74] Tianzhen Liu, Liwu Liu, Miao Yu, Qifeng Li, Chengjun Zeng, Xin Lan, Yanju Liu, and Jinsong Leng. Integrative hinge based on shape memory polymer composites: Material, design, properties and application. *Composite Structures*, 206:164–176, 2018.

[75] Fengfeng Li, Fabrizio Scarpa, Xin Lan, Liwu Liu, Yanju Liu, and Jinsong Leng. Bending shape recovery of unidirectional carbon fiber reinforced epoxy-based shape memory polymer composites. *Composites Part A: Applied Science and Manufacturing*, 116:169–179, 2019.

[76] Van Luong Le, Vinh Tung Le, and Nam Seo Goo. Deployment performance of shape memory polymer composite hinges at low temperature. *Journal of Intelligent Material Systems and Structures*, 30(17):2625–2638, sep 2019.

[77] Zhong Yi Chu and Yian Lei. Design theory and dynamic analysis of a deployable boom. *Mechanism and Machine Theory*, 71:126–141, 2014.

[78] Version Matlab. 7.10. 0 (R2010a). *The MathWorks Inc., Natick, Massachusetts*, 2010.

[79] The SAMTECH Team. SAMCEF User Manual. *LMS Samtech*, 2012.

[80] V. Rossum. *The Python Language Reference Release 3.6.0.* Network Theory Ltd., 2017.

[81] M. Sakovsky, I. Maqueda, C. Karl, S. Pellegrino, and J. Costantine. Dual-matrix composite wideband antenna structures for CubeSats. In *2nd AIAA Spacecraft Structures Conference*, AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, jan 2015.

[82] Maria Sakovsky. Design and Characterization of Dual-Matrix Composite Deployable Space Structures, 2018.

[83] Maria Sakovsky and Sergio Pellegrino. Closed cross-section dual-matrix composite hinge for deployable structures. *Composite Structures*, 208:784–795, 2019.

[84] Hui Yang, Fengshuai Lu, Hongwei Guo, and Rongqiang Liu. Design of a new N-Shape composite ultra-thin deployable boom in the post-buckling range using response surface method and optimization. *IEEE Access*, 7:129659–129665, 2019.

[85] Martin Philip Bendsøe and Noboru Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, 71(2):197–224, 1988.

[86] P Fernandes, R Marques, R Pinto, P Mimoso, J Rodrigues, A Silva, João Manuel, R S Tavares, G Rodrigues, and N Correia. Design and optimization of a self-deployable composite structure. *Revista da la Asociacion Espanola de Materiales Compuestos*, 4(1):80–89, 2020.

[87] Yingjie Xu, Jihong Zhu, Zhen Wu, Yinfeng Cao, Yubo Zhao, and Weihong Zhang. A review on the design of laminated composite structures: constant and variable stiffness design and topology optimization. *Advanced Composites and Hybrid Materials*, 1(3):460–477, 2018.

[88] S. Nikbakt, S. Kamarian, and M. Shakeri. A review on optimization of composite structures Part I: Laminated composites. *Composite Structures*, 195:158–185, 2018.

[89] S. Nikbakht, S. Kamarian, and M. Shakeri. A review on optimization of composite structures Part II: Functionally graded materials. *Composite Structures*, 214:83–102, 2019.

[90] P. Jin, Y. Wang, X. Zhong, J. Yang, and Z. Sun. Blending design of composite laminated structure with panel permutation sequence. *Aeronautical Journal*, 122(1248):333–347, 2018.

[91] B. P. Kristinsdottir and Z. B. Zabinsky. Including manufacturing tolerances in composite design. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 3, pages 1413–1422, 1994.

[92] Haichao An, Shenyan Chen, and Hai Huang. Stacking sequence optimization and blending design of laminated composite structures. *Structural and Multidisciplinary Optimization*, 59(1):1–19, 2019.

[93] Haichao An, Shenyan Chen, and Hai Huang. Laminate stacking sequence optimization with strength constraints using two-level approximations and adaptive genetic algorithm. *Structural and Multidisciplinary Optimization*, 51(4):903–918, 2015.

[94] Haichao An, Shenyan Chen, and Hai Huang. Improved Genetic Algorithm with Two-Level Approximation Method for Laminate Stacking Sequence Optimization by Considering Engineering Requirements. *Mathematical Problems in Engineering*, 2015, 2015.

[95] Shenyan Chen, Zhiwei Lin, Haichao An, Hai Huang, and Changduk Kong. Stacking sequence optimization with genetic algorithm using a two-level approximation. *Structural and Multidisciplinary Optimization*, 48(4):795–805, 2013.

[96] Meysam Esmaeeli, Behzad Kazemianfar, and Mohammad Rahim Nami. Simultaneous optimization of elastic constants of laminated composites using artificial bee colony algorithm. *Advanced Composites and Hybrid Materials*, 2(3):431–443, 2019.

[97] Craig M. Hamel, Devin J. Roach, Kevin N. Long, Frédéric Demoly, Martin L. Dunn, and H. Jerry Qi. Machine-learning based design of active composite structures for 4D printing. *Smart Materials and Structures*, 28(6):65005, 2019.

[98] Gonçalo das Neves Carneiro and Carlos Conceição António. Reliability-based Robust Design Optimization with the Reliability Index Approach applied to composite laminate structures. *Composite Structures*, 209:844–855, 2019.

[99] C A Conceição António. Optimization of structures using composite materials made of polymeric matrix, 1995.

[100] Gonçalo das Neves Carneiro and Carlos Conceiçao Antonio. A RBRDO approach based on structural robustness and imposed reliability level. *Structural and Multidisciplinary Optimization*, 57(6):2411–2429, 2018.

[101] Gonçalo das Neves Carneiro and Carlos Conceição António. Robustness and reliability of composite structures: effects of different sources of uncertainty. *International Journal of Mechanics and Materials in Design*, 15(1):93–107, 2019.

[102] Kwangkyu Yoo, Omar Bacarreza, and M. H.Ferri Aliabadi. Multi-fidelity robust design optimisation for composite structures based on low-fidelity models using successive high-fidelity corrections. *Composite Structures*, 259:113477, 2021.

[103] Bingbing San, Zhi Xiao, and Ye Qiu. Simultaneous Shape and Stacking Sequence Optimization of Laminated Composite Free-Form Shells Using Multi-Island Genetic Algorithm. *Advances in Civil Engineering*, 2019:2056460, 2019.

[104] Zunyi Duan, Jun Yan, Ikjin Lee, Jingyuan Wang, and Tao Yu. Integrated design optimization of composite frames and materials for maximum fundamental frequency with continuous fiber winding angles. *Acta Mechanica Sinica/Lixue Xuebao*, 34(6):1084–1094, 2018.

[105] Krister Svanberg. The method of moving asymptotes—a new method for structural optimization. *International Journal for Numerical Methods in Engineering*, 24(2):359–373, 1987.

[106] F. Farzan Nasab, H. J.M. Geijselaers, I. Baran, R. Akkerman, and A. de Boer. A level-set-based strategy for thickness optimization of blended composite structures. *Composite Structures*, 206:903–920, 2018.

[107] Eralp Demir, Pouya Yousefi-Louyeh, and Mehmet Yildiz. Design of variable stiffness composite structures using lamination parameters with fiber steering constraint. *Composites Part B: Engineering*, 165:733–746, 2019.

[108] Haim Abramovich. *Introduction to composite materials*. CRC Press, 2017.

[109] Torkan Shafighfard, E. Demir, and Mehmet Yildiz. Design of fiber-reinforced variable-stiffness composites for different open-hole geometries with fiber continuity and curvature constraints. *Composite Structures*, 226:111280, 2019.

[110] Dan Wang, Mostafa M. Abdalla, Zhen Pei Wang, and Zhoucheng Su. Streamline stiffener path optimization (SSPO) for embedded stiffener layout design of non-uniform curved grid-stiffened composite (NCGC) structures. *Computer Methods in Applied Mechanics and Engineering*, 344:1021–1050, 2019.

[111] Felipe Fernandez, W. Scott Compel, James P. Lewicki, and Daniel A. Tortorelli. Optimal design of fiber reinforced composite structures and their direct ink write fabrication. *Computer Methods in Applied Mechanics and Engineering*, 353:277–307, 2019.

[112] Yang Shen and David Branscomb. Orientation optimization in anisotropic materials using gradient descent method. *Composite Structures*, 234:111680, 2020.

[113] F. Farzan Nasab, H. J.M. Geijselaers, I. Baran, R. Akkerman, and A. de Boer. Optimization of the interacting stiffened skins and ribs made of composite materials. *AIAA Journal*, 58(4):1836–1850, feb 2020.

[114] Xue Rongrong, Ye Zhengyin, Ye Kun, and Wang Gang. Composite material structure optimization design and aeroelastic analysis on forward swept wing. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 233(13):4679–4695, nov 2019.

[115] Duc Hai Nguyen, Hu Wang, Fan Ye, and Wei Hu. Investigation and multi-scale optimization design of woven composite cut-out structures, jan 2020.

[116] M Sherburn and A C Long. TexGen open source project. *Online at http://texgen. sourceforge. net*, 2010.

[117] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization*, 13(4):455–492, 1998.

[118] Joel P Johnston, Gary D Roberts, and Sandi G Miller. A Design Methodology for Optimizing and Integrating Composite Materials in Gear Structures. *NASA Technical Reports Server*, (September):20, 2019.

[119] Xiaoyang Liu, Carol A. Featherston, and David Kennedy. Two-level layup optimization of composite laminate using lamination parameters. *Composite Structures*, 211:337–350, 2019.

[120] W. H. Wittrick and F. W. Williams. Buckling and vibration of anisotropic or isotropic plate assemblies under combined loadings. *International Journal of Mechanical Sciences*, 16(4):209–239, 1974.

[121] A. D. Viquerat. A continuation-based method for finding laminated composite stacking sequences. *Composite Structures*, 238:111872, 2020.

[122] Werner C. Rheinboldt and Alexander Morgan. *Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems.*, volume 51. Siam, 1988.

[123] Jan Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Transactions on Mathematical Software*, 25(2):251–276, 1999.

[124] Saeed Khodaygan and Mehdi Bohlooly. Multi-objective optimal design of stiffened laminated composite cylindrical shell with piezoelectric actuators. *International Journal on Interactive Design and Manufacturing*, 14(2):595–611, 2020.

[125] Gwo Hshiung Tzeng and Jih Jeng Huang. *Multiple attribute decision making: Methods and applications*. CRC press, 2011.

[126] K. Yoon and Ching-Lai Hwang. *Multiple Attribute Decision Making*, volume 104. Sage publications, 2011.

[127] A. E. Albanesi, I. Peralta, F. Bre, B. A. Storti, and V. D. Fachinotti. An optimization method based on the evolutionary and topology approaches to reduce the mass of composite wind turbine blades. *Structural and Multidisciplinary Optimization*, 62(2):619–643, 2020.

[128] Liang Meng, Weihong Zhang, Dongliang Quan, Guanghui Shi, Lei Tang, Yuliang Hou, Piotr Breitkopf, Jihong Zhu, and Tong Gao. From Topology Optimization Design to Additive Manufacturing: Today's Success and Tomorrow's Roadmap. *Archives of Computational Methods in Engineering*, 27(3):805–830, 2020.

[129] Weihong Zhang, Jungang Yang, Yingjie Xu, and Tong Gao. Topology optimization of thermoelastic structures: Mean compliance minimization or elastic strain energy minimization. *Structural and Multidisciplinary Optimization*, 49(3):417–429, 2014.

[130] M. B. Fuchs, M. Paley, and E. Miroshny. Aboudi micromechanical model for topology design of structures. *Computers and Structures*, 73(1-5):355–362, 1999.

[131] Zheng Dong Ma, Noboru Kikuchi, Christophe Pierre, and Basavaraju Raju. Multidomain topology optimization for structural and material designs. *Journal of Applied Mechanics, Transactions ASME*, 73(4):565–573, 2006.

[132] W. Hansel and W. Becker. Layerwise adaptive topology optimization of laminate structures. *Engineering Computations (Swansea, Wales)*, 16(7):841–851, 1999.

[133] Wilfried Hansel, André Treptow, Wilfried Becker, and Bernd Freisleben. A heuristic and a genetic topology optimization algorithm for weight-minimal laminate structures. *Composite Structures*, 58(2):287–294, 2002.

[134] J. Stegmann and E. Lund. Discrete material optimization of general composite shell structures. *International Journal for Numerical Methods in Engineering*, 62(14):2009–2027, 2005.

[135] Bin Niu, Niels Olhoff, Erik Lund, and Gengdong Cheng. Discrete material optimization of vibrating laminated composite plates for minimum sound radiation. *International Journal of Solids and Structures*, 47(16):2097–2114, 2010.

[136] S. Setoodeh, M. M. Abdalla, and Z. Gürdal. Combined topology and fiber path design of composite layers using cellular automata. *Structural and Multidisciplinary Optimization*, 30(6):413–421, 2005.

[137] M. P. Bendsøe and O. Sigmund. Material interpolation schemes in topology optimization. *Archive of Applied Mechanics*, 69(9-10):635–654, 1999.

[138] G. I.N. Rozvany. A critical review of established methods of structural topology optimization. *Structural and Multidisciplinary Optimization*, 37(3):217–237, 2009.

[139] Robert A. Meyers. *Computational complexity: Theory, techniques, and applications*, volume 9781461418. Springer, 2013.

[140] Kemin Zhou and Xia Li. Topology optimization for minimum compliance under multiple loads based on continuous distribution of members. *Structural and Multidisciplinary Optimization*, 37(1):49–56, 2008.

[141] Tong Gao, Weihong Zhang, and Pierre Duysinx. A bi-value coding parameterization scheme for the discrete optimal orientation design of the composite laminate. *International Journal for Numerical Methods in Engineering*, 91(1):98–114, 2012.

[142] Tong Gao, Weihong H. Zhang, and Pierre Duysinx. Simultaneous design of structural layout and discrete fiber orientation using bi-value coding parameterization and volume constraint. *Structural and Multidisciplinary Optimization*, 48(6):1075–1088, 2013.

[143] Xian Jie Wang and Xun An Zhang. Topology optimization of microstructures of cellular material based on the properties of macrostructures. *Gongneng Cailiao/Journal of Functional Materials*, 45(18):18078–18082, 2014.

[144] Y. M. Xie and G. P. Steven. Basic Evolutionary Structural Optimization. In *Evolutionary Structural Optimization*, pages 12–29. Springer, 1997.

[145] X. Huang and Y. M. Xie. *Evolutionary Topology Optimization of Continuum Structures: Methods and Applications*. John Wiley & Sons, 2010.

[146] X. Yan, X. Huang, Y. Zha, and Y. M. Xie. Concurrent topology optimization of structures and their composite microstructures. *Computers and Structures*, 133:103–110, 2014.

[147] Yiru Ren, Xiang Jinwu, Lin Zheqi, and Zhang Tiantian. A novel topology optimization method for composite beams. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 230(7):1153–1163, 2016.

[148] O. Sigmund. A 99 line topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 21(2):120–127, 2001.

[149] Jinglai Wu, Zhen Luo, Hao Li, and Nong Zhang. Level-set topology optimization for mechanical metamaterials under hybrid uncertainties. *Computer Methods in Applied Mechanics and Engineering*, 319(1):414–441, 2017.

[150] N. P. Van Dijk, K. Maute, M. Langelaar, and F. Van Keulen. Level-set methods for structural topology optimization: A review. *Structural and Multidisciplinary Optimization*, 48(3):437–472, 2013.

[151] Ajit Panesar, Meisam Abdi, Duncan Hickman, and Ian Ashcroft. Strategies for functionally graded lattice structures derived using topology optimisation for Additive Manufacturing. *Additive Manufacturing*, 19:81–94, 2018.

[152] Yang Dai, Miaolin Feng, and Min Zhao. Topology optimization of laminated composite structures with design-dependent loads. *Composite Structures*, 167:251–261, 2017.

[153] Bruno Silva de Sousa, Guilherme Ferreira Gomes, Ariosto Bretanha Jorge, Sebastião Simões da Cunha, and Antonio Carlos Ancelotti. A modified topological sensitivity analysis extended to the design of composite multidirectional laminates structures. *Composite Structures*, 200:729–746, 2018.

[154] Larry L. Howell, Spencer P. Magleby, and Brian M. Olsen. *Handbook of Compliant Mechanisms*. John Wiley & Sons Ltd., 2013.

[155] Benliang Zhu, Xianmin Zhang, Hongchuan Zhang, Junwen Liang, Haoyan Zang, Hai Li, and Rixin Wang. Design of compliant mechanisms using continuum topology optimization: A review, 2020.

[156] Lijie Zhao, Kai Li, Yingying Chang, and Jingkui Li. Topology Optimization Design of Compliant Mechanism of Composite Wing Leading Edge. In *Journal of Physics: Conference Series*, volume 1215, page 12002. IOP Publishing, 2019.

[157] José Humberto S. Almeida, Lars Bittrich, Tsuyoshi Nomura, and Axel Spickenheuer. Cross-section optimization of topologically-optimized variable-axial anisotropic composite structures. *Composite Structures*, 225:111150, 2019.

[158] Alexander A. Safonov. 3D topology optimization of continuous fiber-reinforced structures via natural evolution method. *Composite Structures*, 215:289–297, 2019.

[159] Anders Klarbring and Bo Torstenfelt. Dynamical systems and topology optimization. *Structural and Multidisciplinary Optimization*, 42(2):179–192, 2010.

[160] Jong Wook Lee, Jong Jin Kim, and Gil Ho Yoon. Stress constraint topology optimization using layerwise theory for composite laminates. *Composite Structures*, 226:111184, 2019.

[161] Jaewook Lee, Dongjin Kim, Tsuyoshi Nomura, Ercan M. Dede, and Jeonghoon Yoo. Topology optimization for continuous and discrete orientation design of functionally graded fiber-reinforced composite structures. *Composite Structures*, 201:217–233, 2018.

[162] Yuqing Zhou, Tsuyoshi Nomura, and Kazuhiro Saitou. Multi-component topology and material orientation design of composite structures (MTO-C). *Computer Methods in Applied Mechanics and Engineering*, 342:438–457, 2018.

[163] Tsuyoshi Nomura, Ercan M. Dede, Jaewook Lee, Shintaro Yamasaki, Tadayoshi Matsumori, Atsushi Kawamoto, and Noboru Kikuchi. General topology optimization method with continuous and discrete orientation design using isoparametric projection. *International Journal for Numerical Methods in Engineering*, 101(8):571–605, feb 2015.

[164] Delin Jiang, Robert Hoglund, and Douglas E. Smith. Continuous fiber angle topology optimization for polymer composite deposition additive manufacturing applications. *Fibers*, 7(2):14, 2019.

[165] Tsuyoshi Nomura, Atsushi Kawamoto, Tsuguo Kondoh, Ercan M. Dede, Jaewook Lee, Yuyang Song, and Noboru Kikuchi. Inverse design of structure and fiber orientation by means of topology optimization with tensor field variables. *Composites Part B: Engineering*, 176:107187, 2019.

[166] Xinxing Tong, Wenjie Ge, Xinqin Gao, and Yan Li. Optimization of Combining Fiber Orientation and Topology for Constant-Stiffness Composite Laminated Plates. *Journal of Optimization Theory and Applications*, 181(2):653–670, 2019.

[167] Xinxing Tong, Wenjie Ge, Xinqin Gao, and Yan Li. Simultaneous optimization of fiber orientations and topology shape for composites compliant leading edge. *Journal of Reinforced Plastics and Composites*, 38(15):706–716, apr 2019.

[168] Junjian Fu, Liang Xia, Liang Gao, Mi Xiao, and Hao Li. Topology Optimization of Periodic Structures with Substructuring. *Journal of Mechanical Design, Transactions of the ASME*, 141(7), mar 2019.

[169] Zijun Wu, Liang Xia, Shuting Wang, and Tielin Shi. Topology optimization of hierarchical lattice structures with substructuring. *Computer Methods in Applied Mechanics and Engineering*, 345:602–617, 2019.

[170] M. Jansen and O. Pierard. A hybrid density/level set formulation for topology optimization of functionally graded lattice structures. *Computers and Structures*, 231:106205, 2020.

[171] Y. Wu, Eric Li, Z. C. He, X. Y. Lin, and H. X. Jiang. Robust concurrent topology optimization of structure and its composite material considering uncertainty with imprecise probability. *Computer Methods in Applied Mechanics and Engineering*, 364:112927, 2020.

[172] Ling Liu, Jun Yan, and Gengdong Cheng. Optimum structure with homogeneous optimum truss-like material. *Computers and Structures*, 86(13-14):1417–1425, 2008.

[173] Jiadong Deng, Jun Yan, and Gengdong Cheng. Multi-objective concurrent topology optimization of thermoelastic structures composed of homogeneous porous material. *Structural and Multidisciplinary Optimization*, 47(4):583–597, 2013.

[174] Liang Xia and Piotr Breitkopf. Multiscale structural topology optimization with an approximate constitutive model for local material microstructure. *Computer Methods in Applied Mechanics and Engineering*, 286:147–167, 2015.

[175] Wenjiong Chen, Liyong Tong, and Shutian Liu. Concurrent topology design of structure and material using a two-scale topology optimization. *Computers and Structures*, 178:119–128, 2017.

[176] Xiaolei Yan, Qiwang Xu, Dengfeng Huang, Yong Zhong, and Xiaodong Huang. Concurrent topology design of structures and materials with optimal material orientation. *Composite Structures*, 220:473–480, 2019.

[177] Xiaolei Yan, Qiwang Xu, Haiyan Hua, Dengfeng Huang, and Xiaodong Huang. Concurrent topology optimization of structures and orientation of anisotropic materials. *Engineering Optimization*, 52(9):1598–1611, sep 2020.

[178] Jie Gao, Zhen Luo, Liang Xia, and Liang Gao. Concurrent topology optimization of multiscale composite structures in Matlab. *Structural and Multidisciplinary Optimization*, 60(6):2621–2651, 2019.

[179] Hao Li, Zhen Luo, Mi Xiao, Liang Gao, and Jie Gao. A new multiscale topology optimization method for multiphase composite structures of frequency response with level sets. *Computer Methods in Applied Mechanics and Engineering*, 356:116–144, 2019.

[180] Yan Zhang, Mi Xiao, Liang Gao, Jie Gao, and Hao Li. Multiscale topology optimization for minimizing frequency responses of cellular composites with connectable graded microstructures. *Mechanical Systems and Signal Processing*, 135:106369, 2020.

[181] Yan Zhang, Mi Xiao, Xiaoyu Zhang, and Liang Gao. Topological design of sandwich structures with graded cellular cores by multiscale optimization. *Computer Methods in Applied Mechanics and Engineering*, 361:112749, 2020.

[182] Yan Zhang, Liang Gao, and Mi Xiao. Maximizing natural frequencies of inhomogeneous cellular structures by Kriging-assisted multiscale topology optimization. *Computers and Structures*, 230:106197, 2020.

[183] Andre Luis Ferreira da Silva, Ruben Andres Salas, Emilio Carlos Nelli Silva, and J. N. Reddy. Topology optimization of fibers orientation in hyperelastic composite material. *Composite Structures*, 231:111488, 2020.

[184] Lidy Anaya, William Vicente, and Renato Pavanello. EngOpt 2018 Proceedings of the 6th International Conference on Engineering Optimization. In H C Rodrigues, J Herskovits, C M Mota Soares, A L Araújo, J M Guedes, J O Folgado, F Moleiro, and J F A Madeira, editors, *EngOpt 2018 Proceedings of the 6th International Conference on Engineering Optimization*, pages 1055–1060, Cham, 2019. Springer International Publishing.

[185] M. P. Bendsøe and A. R. Díaz. A method for treating damage related criteria in optimal topology design of continuum structures. *Structural Optimization*, 16(2-3):108–115, 1998.

[186] Mehmet A. Akgün and Raphael T. Haftka. Damage tolerant topology optimization under multiple load cases. *41st Structures, Structural Dynamics, and Materials Conference and Exhibit*, page 317, 2000.

[187] Jiazheng Du, Yunhang Guo, Zuming Chen, and Yunkang Sui. Topology optimization of continuum structures considering damage based on independent continuous mapping method. *Acta Mechanica Sinica/Lixue Xuebao*, 35(2):433–444, 2019.

[188] M. O.W. Richardson and M. J. Wisheart. Review of low-velocity impact properties of composite materials. *Composites Part A: Applied Science and Manufacturing*, 27(12 PART A):1123–1131, 1996.

[189] Stuart Dutton, Donald Kelly, and Alan Baker. *Composite Materials for Aircraft Structures, Second Edition*. AIAA, 2004.

[190] L. Greve and A. K. Pickett. Delamination testing and modelling for composite crash simulation. *Composites Science and Technology*, 66(6):816–826, 2006.

[191] R. Seltzer, C. González, R. Muñoz, J. Llorca, and T. Blanco-Varela. X-ray microtomography analysis of the damage micromechanisms in 3D woven composites under low-velocity impact. *Composites Part A: Applied Science and Manufacturing*, 45:49–60, 2013.

[192] L. Francesconi and F. Aymerich. Numerical simulation of the effect of stitching on the delamination resistance of laminated composites subjected to low-velocity impact. *Composite Structures*, 159:110–120, 2017.

[193] Mehdi Yasaee, Lawrence Bigg, Galal Mohamed, and Stephen R. Hallett. Influence of Z-pin embedded length on the interlaminar traction response of multi-directional composite laminates. *Materials and Design*, 115:26–36, 2017.

[194] Kevin R. Hart, Patrick X.L. Chia, Lawrence E. Sheridan, Eric D. Wetzel, Nancy R. Sottos, and Scott R. White. Mechanisms and characterization of impact damage in 2D and 3D woven fiber-reinforced composites. *Composites Part A: Applied Science and Manufacturing*, 101:432–443, 2017.

[195] Kevin R. Hart, Patrick X.L. Chia, Lawrence E. Sheridan, Eric D. Wetzel, Nancy R. Sottos, and Scott R. White. Comparison of Compression-After-Impact and Flexure-After-Impact protocols for 2D and 3D woven fiber-reinforced composites. *Composites Part A: Applied Science and Manufacturing*, 101:471–479, 2017.

[196] R. Umer, H. Alhussein, J. Zhou, and W. J. Cantwell. The mechanical properties of 3D woven composites. *Journal of Composite Materials*, 51(12):1703–1716, 2017.

[197] Haibao Liu, Brian G. Falzon, and Wei Tan. Predicting the Compression-After-Impact (CAI) strength of damage-tolerant hybrid unidirectional/woven carbon-fibre reinforced composite laminates. *Composites Part A: Applied Science and Manufacturing*, 105:189–202, 2018.

[198] Wei Tan, Brian G. Falzon, Mark Price, and Haibao Liu. The role of material characterisation in the crush modelling of thermoplastic composite structures. *Composite Structures*, 153:914–927, 2016.

[199] Aswani Kumar Bandaru, Shivdayal Patel, Yogesh Sachan, R. Alagirusamy, Naresh Bhatnagar, and Suhail Ahmad. Low velocity impact response of 3D angle-interlock Kevlar/basalt reinforced polypropylene composites. *Materials and Design*, 105:323–332, 2016.

[200] Aswani Kumar Bandaru, Vikrant V. Chavan, Suhail Ahmad, R. Alagirusamy, and Naresh Bhatnagar. Low velocity impact response of 2D and 3D Kevlar/polypropylene composites. *International Journal of Impact Engineering*, 93:136–143, 2016.

[201] Aswani Kumar Bandaru, Suhail Ahmad, and Naresh Bhatnagar. Ballistic performance of hybrid thermoplastic composite armors reinforced with Kevlar and basalt fabrics. *Composites Part A: Applied Science and Manufacturing*, 97:151–165, 2017.

[202] Ku Hyun Jung, Do Hyoung Kim, Hee June Kim, Seong Hyun Park, Kyung Young Jhang, and Hak Sung Kim. Finite element analysis of a low-velocity impact test for glass fiber-reinforced polypropylene composites considering mixed-mode interlaminar fracture toughness. *Composite Structures*, 160:446–456, 2017.

[203] Luigi Sorrentino, Fabrizio Sarasini, Jacopo Tirillò, Fabienne Touchard, Laurence Chocinski-Arnault, David Mellier, and Pietro Russo. Damage tolerance assessment of the interface strength gradation in thermoplastic composites. *Composites Part B: Engineering*, 113:111–122, 2017.

[204] Rafael Santiago, Wesley Cantwell, and Marcílio Alves. Impact on thermoplastic fibre-metal laminates: Experimental observations. *Composite Structures*, 159:800–817, 2017.

[205] C. H. Chiu, M. H. Lai, and G. M. Wu. Compression failure mechanisms of 3-D angle interlock woven composites subjected to low-energy impact. *Polymers and Polymer Composites*, 12(4):309–320, 2004.

[206] F. Chen and J. M. Hodgkinson. Impact behaviour of composites with different fibre architecture. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 223(7):1009–1017, 2009.

[207] P. Potluri, P. Hogg, M. Arshad, D. Jetavat, and P. Jamshidi. Influence of fibre architecture on impact damage tolerance in 3D woven composites. *Applied Composite Materials*, 19(5):799–812, 2012.

[208] E. V. González, P. Maimí, P. P. Camanho, A. Turon, and J. A. Mayugo. Simulation of drop-weight impact and compression after impact tests on composite laminates. *Composite Structures*, 94(11):3364–3378, 2012.

[209] Shang Lin Gao and Jang Kyo Kim. Cooling rate influences in carbon fibre/PEEK composites. Part III: Impact damage performance. *Composites - Part A: Applied Science and Manufacturing*, 32(6):775–785, 2001.

[210] German Reyes and Uday Sharma. Modeling and damage repair of woven thermoplastic composites subjected to low velocity impact. *Composite Structures*, 92(2):523–531, 2010.

[211] B. Vieille, V. M. Casado, and C. Bouvet. Influence of matrix toughness and ductility on the compression-after-impact behavior of woven-ply thermoplastic- and thermosetting-composites: A comparative study. *Composite Structures*, 110(1):207–218, 2014.

[212] D. J. Bull, S. M. Spearing, and I. Sinclair. Observations of damage development from compression-after-impact experiments using ex situ micro-focus computed tomography. *Composites Science and Technology*, 97:106–114, 2014.

[213] Camille Sonnenfeld, Hakima Mendil-Jakani, Romain Agogué, Philippe Nunez, and Pierre Beauchêne. Thermoplastic/thermoset multilayer composites: A way to improve the impact damage tolerance of thermosetting resin matrix composites. *Composite Structures*, 171:298–305, 2017.

[214] Wei Tan and Brian G. Falzon. Modelling the nonlinear behaviour and fracture process of AS4/PEKK thermoplastic composite under shear loading. *Composites Science and Technology*, 126:60–77, 2016.

[215] Wei Tan and Brian G. Falzon. Modelling the crush behaviour of thermoplastic composites. *Composites Science and Technology*, 134:57–71, 2016.

[216] Marcos Yutaka Shiino, Tatiane Scarabel Pelosi, Maria Odila Hilário Cioffi, and Mauricio Vicente Donadon. The Role of Stitch Yarn on the Delamination Resistance in Non-crimp Fabric: Chemical and Physical Interpretation. *Journal of Materials Engineering and Performance*, 26(3):978–986, 2017.

[217] M. R. Abir, T. E. Tay, M. Ridha, and H. P. Lee. Modelling damage growth in composites subjected to impact and compression after impact. *Composite Structures*, 168:13–25, 2017.

[218] Haibao Liu, Brian G. Falzon, and Wei Tan. Experimental and numerical studies on the impact response of damage-tolerant hybrid unidirectional/woven carbon-fibre reinforced composite laminates. *Composites Part B: Engineering*, 136:101–118, 2018.

[219] R. A.M. Santos, P. N.B. Reis, M. J. Santos, and C. A.C.P. Coelho. Effect of distance between impact point and hole position on the impact fatigue strength of composite laminates. *Composite Structures*, 168:33–39, 2017.

[220] R. A.M. Santos, P. N.B. Reis, F. G.A. Silva, and M. F.S.F. de Moura. Influence of inclined holes on the impact strength of CFRP composites. *Composite Structures*, 172:130–136, 2017.

[221] Cesim Atas and Akar Dogan. An experimental investigation on the repeated impact response of glass/epoxy composites subjected to thermal ageing. *Composites Part B: Engineering*, 75:127–134, 2015.

[222] Bulent Murat Icten. Low temperature effect on single and repeated impact behavior of woven glass-epoxy composite plates. *Journal of Composite Materials*, 49(10):1171–1178, 2015.

[223] Kandan Karthikeyan, Benjamin P. Russell, Vikram S. Deshpande, and Norman A. Fleck. Multi-hit armour characterisation of metal-composite bi-layers. *Journal of Mechanics of Materials and Structures*, 7(7):721–734, 2012.

[224] Volkan Arikan and Onur Sayman. Comparative study on repeated impact response of E-glass fiber reinforced polypropylene & epoxy matrix composites. *Composites Part B: Engineering*, 83:1–6, 2015.

[225] Bulent Murat Icten, Binnur Goren Kiral, and Mehmet Emin Deniz. Impactor diameter effect on low velocity impact response of woven glass epoxy composite plates. *Composites Part B: Engineering*, 50:325–332, 2013.

[226] Ercan Sevkat, Benjamin Liaw, and Feridun Delale. Drop-weight impact response of hybrid composites impacted by impactor of various geometries. *Materials and Design*, 52:67–77, 2013.

[227] T. Mitrevski, I. H. Marshall, and R. Thomson. The influence of impactor shape on the damage to composite laminates. *Composite Structures*, 76(1-2):116–122, 2006.

[228] J. A. Artero-Guerrero, J. Pernas-Sánchez, J. López-Puente, and D. Varas. Experimental study of the impactor mass effect on the low velocity impact of carbon/epoxy woven laminates. *Composite Structures*, 133:774–781, 2015.

[229] Damodar R. Ambur and Heather L. Kemmerly. Influence of impactor mass on the damage characteristics and failure strength of laminated composite plates. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 1, pages 724–732, 1998.

[230] Paolo Feraboli and Keith T. Kedward. A new composite structure impact performance assessment program. *Composites Science and Technology*, 66(10):1336–1347, 2006.

[231] Ronald B. Bucinell, Ralph J. Nuismer, and Jim L. Koury. Response of composite plates to quasi-static impact events. In *ASTM Special Technical Publication*, number 1110, pages 528–549. ASTM International, 1991.

[232] Ramazan Karakuzu, Emre Erbil, and Mehmet Aktas. Impact characterization of glass/epoxy composite plates: An experimental and numerical study. *Composites Part B: Engineering*, 41(5):388–395, 2010.

[233] S. Boria, A. Scattina, and G. Belingardi. Impact behavior of a fully thermoplastic composite. *Composite Structures*, 167:63–75, 2017.

[234] A. Gliszczynski, T. Kubiak, P. Rozylo, P. Jakubczak, and J. Bieniaś. The response of laminated composite plates and profiles under low-velocity impact load. *Composite Structures*, 207:1–12, 2019.

[235] N. Dubary, G. Taconet, C. Bouvet, and B. Vieille. Influence of temperature on the impact behavior and damage tolerance of hybrid woven-ply thermoplastic laminates for aeronautical applications. *Composite Structures*, 168:663–674, 2017.

[236] Kun Luan and Bohong Gu. Energy absorption of three-dimensional angle-interlock woven composite under ballistic penetration based on a multi-scale finite element model. *International Journal of Damage Mechanics*, 24(1):3–20, 2015.

[237] Sandeep Agrawal, Kalyan Kumar Singh, and P. K. Sarkar. Impact damage on fibre-reinforced polymer matrix composite - A review. *Journal of Composite Materials*, 48(3):317–332, 2014.

[238] Yue Wang, Jipeng Zhang, Guodong Fang, Jiazhen Zhang, Zhengong Zhou, and Shiyu Wang. Influence of temperature on the impact behavior of woven-ply carbon fiber reinforced thermoplastic composites. *Composite Structures*, 185:435–445, 2018.

[239] Sylwester Samborski. Analysis of the end-notched flexure test configuration applicability for mechanically coupled fiber reinforced composite laminates. *Composite Structures*, 163:342–349, 2017.

[240] Pietro Russo, Antonio Langella, Ilaria Papa, Giorgio Simeoli, and Valentina Lopresto. Low-velocity Impact and Flexural Properties of Thermoplastic Polyurethane/Woven Glass Fabric Composite Laminates. *Procedia Engineering*, 167:190–196, 2016.

[241] Hei Lam Ma, Zhemin Jia, Kin Tak Lau, Jinsong Leng, and David Hui. Impact properties of glass fiber/epoxy composites at cryogenic environment. *Composites Part B: Engineering*, 92:210–217, 2016.

[242] Pietro Russo, Antonio Langella, Ilaria Papa, Giorgio Simeoli, and Valentina Lopresto. Thermoplastic polyurethane/glass fabric composite laminates: Low velocity impact behavior under extreme temperature conditions. *Composite Structures*, 166:146–152, 2017.

[243] R. Matadi Boumbimba, M. Coulibaly, A. Khabouchi, G. Kinvi-Dossou, N. Bonfoh, and P. Gerard. Glass fibres reinforced acrylic thermoplastic resin-based tri-block copolymers composites: Low velocity impact response at various temperatures. *Composite Structures*, 160:939–951, 2017.

[244] Luigi Sorrentino, Davi Silva de Vasconcellos, Marco D'Auria, Fabrizio Sarasini, and Jacopo Tirillò. Effect of temperature on static and low velocity impact properties of thermoplastic composites. *Composites Part B: Engineering*, 113:100–110, 2017.

[245] Mingling Wang, Miao Cao, Hailou Wang, Amna Siddique, Bohong Gu, and Baozhong Sun. Drop-weight impact behaviors of 3-D angle interlock woven composites after thermal oxidative aging. *Composite Structures*, 166:239–255, 2017.

[246] He Ran Wang, Shu Chang Long, Xiao Qing Zhang, and Xiao Hu Yao. Study on the delamination behavior of thick composite laminates under low-energy impact. *Composite Structures*, 184:461–473, 2018.

[247] Yue Wang, Jipeng Zhang, Jiazhen Zhang, Zhengong Zhou, Guodong Fang, and Shiyu Wang. Compressive behavior of notched and unnotched carbon woven-ply PPS thermoplastic laminates at different temperatures. *Composites Part B: Engineering*, 133:68–77, 2018.

[248] Irina Gouzman, Eitan Grossman, Ronen Verker, Nurit Atar, Asaf Bolker, and Noam Eliaz. Advances in Polyimide-Based Materials for Space Applications. *Advanced Materials*, 31(18):1807738, 2019.

[249] A. Misra, M. J. Demkowicz, X. Zhang, and R. G. Hoagland. The radiation damage tolerance of ultra-high strength nanolayered composites. *Jom*, 59(9):62–65, 2007.

[250] A. K. Maji and S. A. Mahnke. A Review of the Degradation of Composites in Mid Earth Orbit. *Engineering Construction and Operations in Challenging Environments Earth and Space 2004: Proceedings of the Ninth Biennial ASCE Aerospace Division International Conference*, pages 567–572, 2004.

[251] Shahruddin Mahzan, Muhamad Fitri, and M. Zaleha. UV radiation effect towards mechanical properties of Natural Fibre Reinforced Composite material: A Review. In *IOP Conference Series: Materials Science and Engineering*, volume 166, page 12021. IOP Publishing, 2017.

[252] Fabrizio Sarasini, Jacopo Tirillò, Simone D'Altilia, Teodoro Valente, Carlo Santulli, Fabienne Touchard, Laurence Chocinski-Arnault, David Mellier, Luca Lampani, and Paolo Gaudenzi. Damage tolerance of carbon/flax hybrid composites subjected to low velocity impact. *Composites Part B: Engineering*, 91:144–153, 2016.

[253] G. Simeoli, D. Acierno, C. Meola, L. Sorrentino, S. Iannace, and P. Russo. The role of interface strength on the low velocity impact behaviour of PP/glass fibre laminates. *Composites Part B: Engineering*, 62:88–96, 2014.

[254] P. Russo, G. Simeoli, L. Sorrentino, and S. Iannace. Effect of the compatibilizer content on the quasi-static and low velocity impact responses of glass woven fabric/polypropylene composites. *Polymer Composites*, 37(8):2452–2459, 2016.

[255] S. Boccardi, C. Meola, G. M. Carlomagno, L. Sorrentino, G. Simeoli, and P. Russo. Effects of interface strength gradation on impact damage mechanisms in polypropylene/woven glass fabric composites. *Composites Part B: Engineering*, 90:179–187, 2016.

[256] N. H. Nash, T. M. Young, P. T. McGrail, and W. F. Stanley. Inclusion of a thermoplastic phase to improve impact and post-impact performances of carbon fibre reinforced thermosetting composites - A review. *Materials and Design*, 85:582–597, 2015.

[257] Daniel F.O. Braga, S. M.O. Tavares, Lucas F.M. Da Silva, P. M.G.P. Moreira, and Paulo M.S.T. De Castro. Advanced design for lightweight structures: Review and prospects. *Progress in Aerospace Sciences*, 69:29–39, 2014.

[258] Nan Yue, Zahra Sharif Khodaei, and M. H. Aliabadi. Damage detection in large composite stiffened panels based on a novel SHM building block philosophy. *Smart Materials and Structures*, 30(4):45004, 2021.

[259] Carlos G. Dávila, Pedro P. Camanho, and Cheryl A. Rose. Failure criteria for FRP laminates. *Journal of Composite Materials*, 39(4):323–345, 2005.

[260] S. T. Pinho, C. G. Dávila, P. P. Camanho, L. Iannucci, and P. Robinson. Failure Models and Criteria for FRP Under In-Plane or Three-Dimensional Stress States Including Shear Non-linearity, 2005.

[261] Samuel Rivallant, Christophe Bouvet, and Natthawat Hongkarnjanakul. Failure analysis of CFRP laminates subjected to compression after impact: FE simulation using discrete interface elements. *Composites Part A: Applied Science and Manufacturing*, 55:83–93, 2013.

[262] N. Hongkarnjanakul, C. Bouvet, and S. Rivallant. Validation of low velocity impact modelling on different stacking sequences of CFRP laminates and influence of fibre failure. *Composite Structures*, 106:549–559, 2013.

[263] C. Bouvet, S. Rivallant, and J. J. Barrau. Low velocity impact modeling in composite laminates capturing permanent indentation. *Composites Science and Technology*, 72(16):1977–1988, 2012.

[264] P P Camanho, A Arteiro, A R Melro, G Catalanotti, and M Vogler. Three-dimensional invariant-based failure criteria for fibre-reinforced composites. *International Journal of Solids and Structures*, 55:92–107, 2015.

[265] Wei Tan, Brian G. Falzon, Louis N.S. Chiu, and Mark Price. Predicting low velocity impact damage and Compression-After-Impact (CAI) behaviour of composite laminates. *Composites Part A: Applied Science and Manufacturing*, 71:212–226, 2015.

[266] A. Puck and H. Schürmann. Failure analysis of FRP laminates by means of physically based phenomenological models. In *Failure Criteria in Fibre-Reinforced-Polymer Composites*, pages 832–876. Elsevier, 2004.

[267] G Catalanotti, P P Camanho, and A T Marques. Three-dimensional failure criteria for fiber-reinforced laminates. *Composite Structures*, 95:63–79, 2013.

[268] F. Caputo, A. De Luca, and R. Sepe. Numerical study of the structural behaviour of impacted composite laminates subjected to compression load. *Composites Part B: Engineering*, 79:456–465, 2015.

[269] A. Elias, F. Laurin, M. Kaminski, and L. Gornet. Experimental and numerical investigations of low energy/velocity impact damage generated in 3D woven composite with polymer matrix. *Composite Structures*, 159:228–239, 2017.

[270] P. Rozylo, H. Debski, and T. Kubiak. A model of low-velocity impact damage of composite plates subjected to Compression-After-Impact (CAI) testing. *Composite Structures*, 181:158–170, 2017.

[271] J. Cugnoni, R. Amacher, S. Kohler, J. Brunner, E. Kramer, C. Dransfeld, W. Smith, K. Scobbie, L. Sorensen, and J. Botsis. Towards aerospace grade thin-ply composites: Effect of ply thickness, fibre, matrix and interlayer toughening on strength and damage tolerance. *Composites Science and Technology*, 168:467–477, 2018.

[272] George J. Dvorak and Norman Laws. Analysis of Progressive Matrix Cracking In Composite Laminates II. First Ply Failure. *Journal of Composite Materials*, 21(4):309–329, 1987.

[273] Pedro P. Camanho, Carlos G. Dávila, Silvestre T. Pinho, Lorenzo Iannucci, and Paul Robinson. Prediction of in situ strengths and matrix cracking in composites under transverse tension and in-plane shear. *Composites Part A: Applied Science and Manufacturing*, 37(2):165–176, 2006.

[274] Lode Daelemans, Nuray Kizildag, Wim Van Paepegem, Dagmar R. D'hooge, and Karen De Clerck. Interdiffusing core-shell nanofiber interleaved composites for excellent Mode I and Mode II delamination resistance. *Composites Science and Technology*, 175:143–150, 2019.

[275] Polyxeni Dimoka, Spyridon Psarras, Christine Kostagiannakopoulou, and Vassilis Kostopoulos. Assessing the damage tolerance of Out of Autoclave manufactured carbon fibre reinforced polymers modified with multi-walled carbon nanotubes. *Materials*, 12(7):1080, 2019.

[276] Ying Tie, Yuliang Hou, Cheng Li, Liang Meng, Thaneshan Sapanathan, and Mohamed Rachik. Optimization for maximizing the impact-resistance of patch repaired CFRP laminates using a surrogate-based model. *International Journal of Mechanical Sciences*, 172:105407, 2020.

[277] Andrea Sellitto. Optimum design of damage resistant reinforced composite panels. In *Key Engineering Materials*, volume 827 KEM, pages 37–42. Trans Tech Publ, 2020.

[278] M. Prashanth Reddy, Shuvajit Mukherjee, and Ranjan Ganguli. Optimal design of damage tolerant composite using ply angle dispersion and enhanced bat algorithm. *Neural Computing and Applications*, 32(8):3387–3406, 2020.

[279] Christian Mittelstedt. Buckling and post-buckling of thin-walled composite laminated beams-a review of engineering analysis methods. *Applied Mechanics Reviews*, 72(2), jan 2020.

[280] Leonid L. Firsov, Alexey N. Fedorenko, and Boris N. Fedulov. Residual strength estimation based on topology optimization algorythm. *ECCM 2018 - 18th European Conference on Composite Materials*, pages 1–19, 2020.

[281] Yao Chen, Jiayi Yan, and Jian Feng. Mechanism design with singularity avoidance of crystal-inspired deployable structures. *Crystals*, 9(8):421, 2019.

[282] András Lengyel and Zhong You. Bifurcations of SDOF mechanisms using catastrophe theory. *International Journal of Solids and Structures*, 41(2):559–568, 2004.

[283] Andreas Steinboeck, Xin Jia, Gerhard Hoefinger, and Herbert A. Mang. Conditions for symmetric, antisymmetric, and zero-stiffness bifurcation in view of imperfection sensitivity and insensitivity. *Computer Methods in Applied Mechanics and Engineering*, 197(45-48):3623–3636, 2008.

[284] W T Koiter. On the stability of elastic equilibrium; Translation of'Over de Stabiliteit von het elastisch evenwich'. In *Nasa Tt F-10,833*. Polytechnic Institute Delft, HJ Paris Publisher Amsterdam, 1945.

[285] Yao Chen, Jian Feng, and Qiuzhi Sun. Lower-order symmetric mechanism modes and bifurcation behavior of deployable bar structures with cyclic symmetry. *International Journal of Solids and Structures*, 139-140:1–14, 2018.

[286] Owen C. Brown, Paul Eremenko, and Paul D. Collopy. Value-centric design methodologies for fractionated spacecraft: Progress summary from phase 1 of the DARPA System F6 program. In *AIAA Space 2009 Conference and Exposition*, page 6540, 2009.

[287] Fred Y. Hadaegh, Soon Jo Chung, and Harish M. Manohara. On development of 100-gram-class spacecraft for swarm applications. *IEEE Systems Journal*, 10(2):673–684, 2016.

[288] USAF Chief Scientist. Technology Horizons: A Vision for Air Force Science & Technology During 2010-2030. *Technology Horizons*, 1(May):171, 2010.

[289] Saptarshi Bandyopadhyay, Rebecca Foust, Giri P. Subramanian, Soon Jo Chung, and Fred Y. Hadaegh. Correction: Review of Formation Flying and Constellation Missions Using Nanosatellites. *Journal of Spacecraft and Rockets*, 57(6):AU1, 2020.

[290] Carlos G. Dávila, Pedro P. Camanho, and Albert Turon. Effective simulation of delamination in aeronautical structures using shells and cohesive elements. *Journal of Aircraft*, 45(2):663–672, 2008.

[291] Mike R. Woodward and Rich Stover. Damage Tolerance. *Composites*, 21:295–301, 2018.

[292] Elisa C Borowski. *Viscoelastic Effects in Carbon Fiber Reinforced Polymer Strain Energy Deployable Composite Tape Springs*. PhD thesis, University of New Mexico, 2017.

[293] C. S. Lopes, O. Seresta, Y. Coquet, Z. Gürdal, P. P. Camanho, and B. Thuis. Low-velocity impact damage on dispersed stacking sequence laminates. Part I: Experiments. *Composites Science and Technology*, 69(7-8):926–936, 2009.

[294] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

[295] Eric Ziegel. *Genetic Algorithms and Engineering Optimization*, volume 44. John Wiley & Sons, 2002.

[296] Rania Hassan, Babak Cohanim, Olivier De Weck, and Gerhaid Venter. A comparison of particle swarm optimization and the genetic algorithm. In *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, volume 2, pages 1138–1150, 2005.

[297] R. Orosei, S. E. Lauro, E. Pettinelli, A. Cicchetti, M. Coradini, B. Cosciotti, F. Di Paolo, E. Flamini, E. Mattei, M. Pajola, F. Soldovieri, M. Cartacci, F. Cassenti, A. Frigeri, S. Giuppi, R. Martufi, A. Masdea, G. Mitri, C. Nenna, R. Noschese, M. Restano, and R. Seu. Radar evidence of subglacial liquid water on Mars. *Science*, 361(6401):490–493, 2018.

[298] M. Denis, A. Moorhouse, A. Smith, M. McKay, J. Fischer, P. Jayaraman, Z. Mounzer, R. Schmidt, J. Reddy, E. Ecale, R. Horttor, D. Adams, and E. Flamini. Deployment of the MARSIS radar antennas on-board Mars Express. In *SpaceOps 2006 Conference*, page 5959, 2006.

[299] W. Schiehlen. *Multibody Systems Handbook*. Springer, Berlin, 1990.

[300] Arafat I. Khan, Elisa C. Borowski, Eslam M. Soliman, and Mahmoud M. Reda Taha. Examining Energy Dissipation of Deployable Aerospace Composites Using Matrix Viscoelasticity. *Journal of Aerospace Engineering*, 30(5):04017040, 2017.

[301] Fernandes, Pedro (2020), "Master-curve data processing", Mendeley Data, v1 http://dx.doi.org/10.17632/wsfncdwt4t.1.

[302] ASTM. *D3039/D3039M Standard Test Method for Tensile Properties of Polymer Matrix Composite Materials*. ASTM International, 2014.

[303] D 3518. *Standard Test Method for In-Plane Shear Response of Polymer Matrix Composite Materials by Tensile Test of a 645 ° Laminate 1*, volume 94. ASTM International, 2007.

[304] ASTM International. Standard Test Methods for Void Content of Reinforced Plastics. *Astm D 2734-94*, 08(Reapproved):3–5, 2003.

[305] ASTM. *ASTM D3171 Standard test methods for constituent content of composite materials*. ASTM International, 2011.

[306] US Department of Defense. Composite Materials Handbook Volume 1: Polymer Matrix Composites - Guidelines for Characterisation of Structural Materials. *Composite Materials Handbook Series*, 1(June):1–586, 2002.

[307] Ever J. Barbero. *Introduction to Composite Materials Design*. CRC press, 2010.

[308] Rui Miranda Guedes, António Torres Marques, and Albert Cardon. Analytical and Experimental Evaluation of Nonlinear Viscoelastic-Viscoplastic Composite Laminates under Creep, Creep-Recovery, Relaxation and Ramp Loading, 1998.

[309] Kawai Kwok. Mechanics of viscoelastic thin-walled structures, 2013.

[310] Malcolm L. Williams, Robert F. Landel, and John D. Ferry. The Temperature Dependence of Relaxation Mechanisms in Amorphous Polymers and Other Glass-forming Liquids. *Journal of the American Chemical Society*, 77(14):3701–3707, 1955.

[311] John D. Ferry. *Viscoelastic properties of polymers*. John Wiley & Sons, 1980.

[312] Ian M Ward and Dennis W Hadley. *An introduction to the mechanical properties of solid polymers*, volume 42. Wiley, 2004.

[313] Yasushi Miyano and Masayuki Nakada. *Durability of Fiber-Reinforced Polymers*. Wiley Online Library, 2018.

[314] Arthur Tobolsky and Henry Eyring. Mechanical properties of polymeric materials. *The Journal of Chemical Physics*, 11(3):125–134, 1943.

[315] F. R.N. Nabarro. The time constant of logarithmic creep and relaxation. *Materials Science and Engineering A*, 309-310:227–228, 2001.

[316] F. R. Schwarzl. Numerical calculation of storage and loss modulus from stress relaxation data for linear viscoelastic materials. *Rheologica Acta*, 10(2):165–173, 1971.

[317] F. R. Schwarzl. Numerical calculation of storage and loss modulus from stress relaxation data for linear viscoelastic materials. *Rheologica Acta*, 10(2):165–173, 1971.

[318] P. Fernandes, R. Pinto, and N. Correia. Design and optimization of self-deployable damage tolerant composite structures: A review. *Composites Part B: Engineering*, 221:109029, 2021.

[319] P. Fernandes, B. Sousa, R. Marques, João Manuel R.S. Tavares, A. T. Marques, R. M.Natal Jorge, R. Pinto, and N. Correia. Influence of relaxation on the deployment behaviour of a CFRP composite elastic–hinge. *Composite Structures*, 259:113217, 2021.

[320] Fernandes, Pedro (2021), "Performance analysis of a damage-tolerant composite self-deployable elastic-hinge - ABAQUS input files", Mendeley Data, V1, doi: 10.17632/dz4yr438hb.1.

[321] Erik Andreassen, Anders Clausen, Mattias Schevenels, Boyan S. Lazarov, and Ole Sigmund. Efficient topology optimization in MATLAB using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43(1):1–16, 2011.

[322] Kai Liu and Andrés Tovar. An efficient 3D topology optimization code written in Matlab. *Structural and Multidisciplinary Optimization*, 50(6):1175–1196, 2014.

[323] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural Optimization*, 1(4):193–202, 1989.

[324] M. Zhou and G. I.N. Rozvany. The COC algorithm, Part II: Topological, geometrical and generalized shape optimization. *Computer Methods in Applied Mechanics and Engineering*, 89(1-3):309–336, 1991.

[325] G. I. N. Rozvany and J. Sobieszczanski-Sobieski. New optimality criteria methods: Forcing uniqueness of the adjoint strains by corner-rounding at constraint intersections. *Structural Optimization*, 4(3-4):244–246, 1992.

[326] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[327] Michael Yu Wang, Xiaoming Wang, and Dongming Guo. A level set method for structural topology optimization. *Computer Methods in Applied Mechanics and Engineering*, 192(1-2):227–246, 2003.

[328] Vivien J. Challis. A discrete level-set topology optimization code written in Matlab. *Structural and Multidisciplinary Optimization*, 41(3):453–464, 2010.

[329] V. Young, O. M. Querin, G. P. Steven, and Y. M. Xie. 3D and multiple load case bi-directional evolutionary structural optimization (BESO). *Structural Optimization*, 18(2-3):183–192, 1999.

[330] Xiaodong Huang and Yi Min Xie. A further review of ESO type methods for topology optimization. *Structural and Multidisciplinary Optimization*, 41(5):671–683, 2010.

[331] Zhi Hao Zuo and Yi Min Xie. A simple and compact Python code for complex 3D topology optimization. *Advances in Engineering Software*, 85:1–11, 2015.

[332] Pedro Fernandes. Python Code for Stress Constrained Topology Optimization in ABAQUS, 2021.

[333] Erik Holmberg, Bo Torstenfelt, and Anders Klarbring. Stress constrained topology optimization. *Structural and Multidisciplinary Optimization*, 48(1):33–47, 2013.

[334] Matteo Bruggi. On an alternative approach to stress constraints relaxation in topology optimization. *Structural and Multidisciplinary Optimization*, 36(2):125–141, 2008.

[335] Chau Le, Julian Norato, Tyler Bruns, Christopher Ha, and Daniel Tortorelli. Stress-based topology optimization for continua. *Structural and Multidisciplinary Optimization*, 41(4):605–620, 2010.

[336] Lirong Yang, Andrei V. Lavrinenko, Jørn M. Hvam, and Ole Sigmund. Design of one-dimensional optical pulse-shaping filters by time-domain topology optimization. *Applied Physics Letters*, 95(26), 2009.

[337] X. Huang and Y. M. Xie. Bi-directional evolutionary topology optimization of continuum structures with one or multiple materials. *Computational Mechanics*, 43(3):393–401, 2009.

[338] Martin P. Bendsøe. *Optimization of Structural Topology, Shape, and Material*. Springer, 1995.

[339] Serena Ferraro. Topology optimization and failure analysis of deployable thin shells with cutouts, 2020.

[340] O. Sigmund and J. Petersson. Numerical instabilities in topology optimization: A survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural Optimization*, 16(1):68–75, 1998.

[341] X. Huang and Y. M. Xie. Convergent and mesh-independent solutions for the bi-directional evolutionary structural optimization method. *Finite Elements in Analysis and Design*, 43(14):1039–1049, 2007.

[342] K. Svanberg. Algorithm for Optimum Structural Design Using Duality. In *Mathematical Programming Study*, number 20, pages 161–177. Springer, 1982.

[343] C. Fleury. Structural weight optimization by dual methods of convex programming. *International Journal for Numerical Methods in Engineering*, 14(12):1761–1783, 1979.

[344] Arjen Deetman. GCMMA-MMA-Python, 2020.

[345] Krister Svanberg. MMA and GCMMA Matlab code.

[346] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, and D. Cournapeau. SciPy 1.2.1: Fundamental Algorithms for Scientific Computing in Python. *Nature methods*, 17(3):261–272, 2020.

[347] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[348] Richard H. Byrd, Mary E. Hribar, and Jorge Nocedal. An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900, 1999.

[349] T. Buhl, C. B.W. Pedersen, and O. Sigmund. Stiffness design of geometrically nonlinear structures using topology optimization. *Structural and Multidisciplinary Optimization*, 19(2):93–104, 2000.

[350] P. Duysinx and O. Sigmund. New developments in handling stress constraints in optimal material distribution. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 1501–1509, 1998.

[351] David E. Goldberg and John H. Holland. Genetic Algorithms and Machine Learning. *Machine Learning*, 3(2):95–99, 1988.

[352] Singiresu S Rao. *Engineering optimization: theory and practice*. John Wiley & Sons, 2009.

[353] Richard L. Fox and C. D. Mote. Optimization Methods for Engineering Design. *Journal of Dynamic Systems, Measurement, and Control*, 94(2):172–173, 1972.

[354] Maurice Clerc. Particle Swarm Optimization. In *Particle Swarm Optimization*, volume 4, pages 1942–1948. Citeseer, 2010.

[355] Serkan Kiranyaz. Particle swarm optimization. *Adaptation, Learning, and Optimization*, 15(1):45–82, 2014.

[356] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[357] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Xiaodong Li. An analysis of the velocity updating rule of the particle swarm optimization algorithm. *Journal of Heuristics*, 20(4):417–452, 2014.

[358] Maurice Clerc. Particle Swarm Optimization. In *Particle Swarm Optimization*, pages 760–766. Springer, US, 2010.

[359] F. Chorlton. *Basic optimisation methods, by B. D. Bunday. Pp 136. £7·50. 1984. ISBN 0-7131-3506-9 (Arnold)*, volume 69. Edward Arnold London, 1985.

Appendix A

Genetic Algorithm

A.1 Overview of the classic implementation

In the classical implementation, each solution is represented by a vector whose coordinates indicate the coded value of the design variables. In the scope of genetic algorithms, the literature refers to the possible solutions as "individuals", the vector defining the solution (individual) is a "chromosome", and the coordinates or segments of the vector (chromosome) that define a single design variable are called "genes" [351–353]. The algorithm generates a group of individuals, referred to as “population”, whose chromosomes are randomly selected and, therefore, include possible solutions scattered through the design space.

The structure of a GA allows the individuals of a population to interact with each other and be subject to an external factor, creating an analogy to a population of a given species in its environment [351–353]. This external factor is a "fitness function" that determines the suitability of the solution described by each individual to the given problem. Depending on its performance, the individual will have a higher or lower probability of moving on to the next iteration. This sequence of operations referred to as “Evaluation”.

Individuals with higher performances have a better chance of surviving the “Selection” process, where the size of the population decreases due to the removal of possible solutions and moving on to the “Reproduction” stage. This phase aims to generate new possible solutions, which could prove more suitable to the given problem, based on the best results obtained. The sequence of operations that replicate the reproduction of two individuals is referred to as “crossover”, where the offspring is created by recombining the chromosomes of the progenitors. According to the classical GA, the chance of an individual being selected to generate offspring is proportional to its performance and equal to the ratio between the performance of the individual and the sum of performances of each individual in the population [352]. The obtained offspring may also be subject to a "mutation" process, causing part of the chromosome of the offspring to be different from the chromosomes of the parents. This phenomenon observed in nature is introduced into the GA as a tool that increases the diversity in a population by creating new solutions in the neighborhood of the current solution.

The sequence of "Reproduction", "Selection" and "Evaluation" phases is then repeated until a convergence criterion is met.

A.2 Modifications and implementation details

The GA implemented in this research considers some modifications compared to the classical version, aiming to increase the diversity of possible solutions in each population and the computational efficiency. These differences are detailed and discussed throughout this section.

The definition of the "genes" is what makes the GA a naturally discrete method. It is common to use a binary code to represent discrete values for each design variable. Increasing the number of discrete values considered leads to a more refined search but increases the computational cost and time to convergence of the algorithm. Unlike the traditional codification, the GA implemented in this research considered a decimal codification [1].

The second modification is the implementation of an “overpopulation factor”, which makes the number of individuals in the initial population larger than in the following populations, increasing the “gene” pool in the first iteration without a significant increase in the computational cost of the algorithm.

The third modification is the implementation of a genetic operator that forbids the existence of two equal solutions throughout the optimization process, from here onwards referred to as “diversity check”. The diversity check operator creates a record of every chromosome generated during the optimization process. Every time an individual is generated, the diversity check assesses if the individual has a chromosome that was already recorded. If not, the chromosome is added to the record, otherwise the individual will be subject to as many mutations as necessary until its chromosome represents a solution that had never been evaluated by the GA and then adds it to the record. To enhance this virtue, the fitness of each solution was recorded alongside the diversity record. This decision allows the algorithm to avoid the re-evaluation of individuals that were kept from one generation to the other, further reducing the computational time needed to run the algorithm.

The selection of individuals that move to the next generation and that are selected to reproduce is also different from the classic GA. Every generation, the top performing individuals in the population are automatically selected to move to the next generation, while the remaining individuals are given a small chance of survival. Then, pairs of individuals are randomly selected to reproduce through uniform crossover [352, 353]. This creates an elitist process, ensuring the survival of the best solution, and is balanced with a random choice that allows individuals to reproduce regardless of their performances, enhancing the explorative nature of the algorithm.

Unless stated otherwise in a particular chapter, the GA implemented considered the following internal parameters:

[1] In Chapter 3, the GA was used to perform a global search, which did not require an extensive discretization of the design space and further reduced the time to convergence of the algorithm.

- The minimum population size is equal to 30 individuals. The initial population considers an overpopulation factor of 5, leading to an initial total of 150 individuals.
- In each generation, the seven fittest solutions define the "elite group", which are always allowed to pass during the Selection process. The remaining individuals in the population have a survival rate equal to the one defined in the classical GA. However, individuals removed have a 5 % chance of being reintroduced in the following generation regardless of their performance to promote the diversity of the population. This may lead to a population larger than 30 individuals.
- The reproduction is performed according to a uniform crossover operator, meaning that each gene in the chromosome has an equal chance of being equal to one of the two progenitors [352, 353].
- After the crossover operation, each gene in the chromosome has a 7.5 % chance of suffering a mutation. A mutation corresponds to the assignment of a value randomly chosen between the minimum and maximum range of acceptable values for the corresponding design variable.
- It is assumed that the algorithm has converged after 30 generations with no improvement on the performance of the elite group.

Appendix B

Particle Swarm Optimization

B.1 Overview of the implementation

The Particle Swarm Optimization method mimics the behavior of social organisms [354]. In this algorithm, each particle denotes an intelligent individual of the swarm whose goal is to find a location rich in resources. The position of each particle represents a possible solution for the optimization problem. During this search, the particles are allowed to interact with each other, sharing information about the quality of the position they have visited. As a result, if one particle discovers a good solution, the rest of the swarm will be capable of converging towards it, evaluating other possible solutions along the way [355, 352, 354].

The information gathered is shared within the swarm, allowing its members to redefine their trajectories. The trajectory of each particle, defined by its velocity, depends on three parameters whose goal is to represent the social interaction with the swarm. The first is an inertial parameter, representing its current motion and its resistance when trying to stray from its path. The second represents the memory or cognitive behavior of the particle, promoting the movement towards the best position each individual particle has visited. The last one is a social term that indicates the influence of the swarm, favoring the movement towards the best position visited by at least one member of the swarm. As a result, the velocity of each particle is defined by the sum of three vectors [356, 355, 354, 357, 358], as shown in Figure B.1 (figure cited from [296]).

In the form of an equation, the velocity of a particle and its following position can be calculated as [357]:

$$v_{i+1} = v_i c_0 + c_1 r_1 (P_B^g - P_i) + c_2 r_2 (P_B^g - P_i) \tag{B.1}$$

$$P_{i+1} = P_i + dt v_{i+1} \tag{B.2}$$

Where:

- v_{i+1} and v_i are the velocity in the following and current increment, respectively;
- P_{i+1} and P_i are the position of the particle in the following and current increment, respectively;

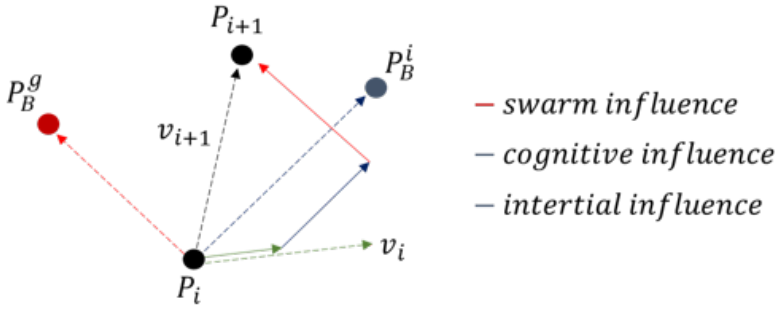

Figure B.1: Update of the position of a particle as a function of the inertial, cognitive and swarm influences (image cited from [296]).

- c_0, c_1 and c_2 are coefficients that represent the inertial, cognitive and swarm influences on the particle;
- P_i is the current position of the particle;
- P_B^i is the best position visited by the particle;
- P_B^g is the best position visited by at least one particle of the swarm;
- r_1 and r_2 are random values within the interval [0,1];
- dt is the time step increment between each iteration.

The coefficients c_0 through c_2 denote the relative importance of the inertial, cognitive and social terms [356, 355, 357, 358]. r_1 and r_2 are random numbers that avoid the particles to move directly towards a given location, exploiting new solutions around the global and individual best solution found and diversifying the particles for a more effective search [357]. Notice that for each particle, once v_{i+1} is determined, the problem can be converted into a single variable optimization problem in function of dt. Given this condition, the implemented version of the PSO method includes the optimization of the time step increment dt through the golden section search method [359]. Furthermore, as a result of Equation B.2, each particle can move through a continuous design space [1].

Unless stated otherwise in a particular chapter, the following internal parameters were used in accordance with [356, 355]:

- $c_0 = 0.7298$
- $c_1 = c_2 = 1.496181$
- Number of particles equal to 10.
- Convergence is assumed after 30 iterations with no improvement of the best solution found.

[1] Due to this characteristic, in Chapter 3 the PSO method was chosen to perform a local search near the best solution found during the global search. This allowed the GA to use a less refined discretization and reduce the computational cost of the optimization process.

Appendix C

Python code for stress-constrained topology optimization in ABAQUS®

```
# -*- coding: utf-8 -*-
"""
@author: pnfernandes
"""

#%% Imported modules.
from __future__ import division
from abaqus import getInputs
from abaqusConstants import *
from odbAccess import openOdb
import mesh
import numpy as np
from numpy import random
import math
import customKernel
import os
import operator
from numpy import diag as diags
from numpy.linalg import solve, inv
from operator import attrgetter
import displayGroupMdbToolset as dgm
import scipy
from scipy.optimize import LinearConstraint


#%% ABAQUS model preparation.
class ModelPreparation:
    """ Model Preparation class

    This class prepares the ABAQUS model for the topology optimization process.
    The operations can be divided in three major tasks:

    - the generation of materials, sections and element/node sets.
    - requesting specific outputs, extracting user-defined information
      (such as pre-existing sets) and, for stress dependent optimization with
      S4 elements, information on the nodes coordinates and normal vectors.
    - updating the material properties assigned to each element based on
      changes to the design variables.

    Attributes:
    -----------
    - mdb (Mdb): ABAQUS model database.
    - model_name (str): Name of the ABAQUS model.
    - nonlinearities (boolean): Indicates if the problem considers geometrical
      nonlinearities (True) or not (False).
    - part_name (str): Name of the ABAQUS part to be optimized.
    - material_name (str): Name of the ABAQUS material to be considered.
    - section_name (str): Name of the ABAQUS material section to be considered.
    - elmts (MeshElementArray): element_array from ABAQUS with the relevant
```

```
    elements in the part.
  - all_elmts (MeshElementArray): element_array from ABAQUS with all the
    elements in the part.
  - model (Mdb): model from the ABAQUS model database.
  - reference_material (Material): ABAQUS material.
  - reference_section (Section): ABAQUS material section.
  - part (Part): ABAQUS part.
  - opt_method (int): variable defining the optimization method to be used.
  - xe_min (float): minimum density allowed for the element. I.e. minimum
    value allowed for the design variables.
  - dp (int): number of decimals places to be considered. By definition,
    equal to the number of decimal places in xe_min.
  - p (float): SIMP penalty factor.

  Methods:
  ---------
  - get_model_information(): extracts pre-existing information from the
    ABAQUS model.
  - format_model(): decorator defining the
    sequence of operations required to format the ABAQUS model.
  - property_update(editable_xe): updates the properties assigned to each
    element based on their current design variable.

  Auxiliary methods:
  ------------------
  - property_extraction(): extracts the material properties found in the
    ABAQUS model.
  - generate_materials(): creates different ABAQUS materials for each
    possible value of the design variables.
  - calculte_property(rho): determines the properties that each design
    variable should have.
  - prop_val(prop, rho): interpolates the value of a property for a given
    design variable value.
  - generate_output_request(opt_method): requests the ABAQUS variable
    outputs necessary.
  - generate_sets(opt_method): creates the node/element sets
    required.
  - return_sets(): returns a list of pre-defined sets created by the user.
  - get_model_information(): extracts user-defined information from the
    ABAQUS model (element type, sets, boundary conditions, and normal
    vectors).
  - get_element_type(): returns the type of element used in the model.
  - get_active_loads(): identifies the pre-existing active loads.
  - get_active_boundary_conditions(): identifies the active BCs.
  - get_node_coordinates(): identifies the element node coordinates.
  - get_node_normal_vectors(): identifies the vectors normal to each node.
  - normal_vectors(v1,v2): determines the three normal vectors of a node.
  - calculate_normal_vectors(v1,v2): determines the vector normal to 2
    vectors (v1, v2).
  - parallel_vector_check(vector): checks if a vector is normal to [0,1,0].
  """
  def __init__(
          self, mdb, model_name, nonlinearities, part_name, material_name,
          section_name, elmts, all_elmts, xe_min, opt_method, dp, p
      ):

      self.mdb = mdb
      self.model_name = model_name
      self.nonlinearities = nonlinearities
      self.part_name = part_name
      self.material_name = material_name
      self.section_name = section_name
      self.elmts = elmts
      self.all_elmts = all_elmts
      self.model = mdb.models[self.model_name]
      self.reference_material = self.model.materials[self.material_name]
      self.reference_section = self.model.sections[self.section_name]
      self.part = self.model.parts[self.part_name]
      self.opt_method = opt_method
      self.xe_min = xe_min
      self.dp = dp
      self.p = p

  def format_model(self):
```

```
        """ Format model method

        Method that modifies the .CAE file from ABAQUS. It serves mainly as
        a decorator that organizes the different tasks that need to be executed
        in order to prepare the ABAQUS model for the topology optimization
        process, which can be summarized as follows:

        - Creates the ABAQUS materials for each possible design density.
        - Request the ABAQUS outputs necessary (ex: to determine the compliance
          sensitivity).
        - Create the node and element sets necessary (to assign properties
          efficiently and to create the adjoint problem, if used).
        """
        self.property_extraction()
        self.generate_materials()
        self.generate_output_request()
        self.generate_sets()

    def property_extraction(self):
        """Property Extraction method

        Function that reads the material properties defined in the CAE file by
        the user. The function outputs 5 boolean variables that classify the
        existence of each property. Furthermore, the function will create
        global variables with the float value of the properties defined in the
        CAE file, as well as the material thickness considered (if defined)
        making them accessible in other steps of the topology optimization
        process.

        Outputs:
        -------
        Although not specified with a return statement, this method will assign
        the following attributes to the ModelPreparation class:

        - density_properties (boolean): checks the existence of density
          properties defined by the user.
        - elastic_properties (boolean): checks the existence of elastic
          properties defined by the user.
        - failstrain_properties (boolean): checks the existence of fail
          strain parameters, defined by the user.
        - failstress_properties (boolean): checks the existence of fail
          stress parameters, defined by the user.
        - hashindamageinitiation (boolean): checks the existence of the
          parameters necessary to apply the Hashins' failure criteria,
          defined by the user.
        - thickness (float): thickness assigned to the material section. Set to
          1.0 if not specified.

        Depending on the material properties used in the numerical model, this
        method may create the following global variables:

        If there is a definition of the material density:
            - Density (float): material density.

        If the material is elastic and isotropic:
            - Youngs_modulus (float): Young's modulus.
            - Poisson (float): Poisson's coefficient.

        If the material is elastic and described through engineering constants:
            - E11 (float): Young's modulus, direction 11.
            - E22 (float): Young's modulus, direction 22.
            - E33 (float): Young's modulus, direction 33.
            - Nu12 (float): Poisson's coefficient, direction 12.
            - Nu13 (float): Poisson's coefficient, direction 13.
            - Nu23 (float): Poisson's coefficient, direction 23.
            - G12 (float): Shear modulus, direction 12.
            - G13 (float): Shear modulus, direction 13.
            - G23 (float): Shear modulus, direction 23.

        If the material considers fail strain parameters:
            - Strain_xt (float): longitudinal tensile fail strain.
            - Strain_xc (float): longitudinal compressive fail strain.
            - Strain_yt (float): transverse tensile fail strain.
            - Strain_yc (float): transverse compressive fail strain.
```

```
        - Strain_s (float): shear fail strain.

    If the material considers fail stress parameters:
        - Xt (float): Longitudinal tensile stress.
        - Xc (float): Longitudinal compressive stress.
        - Yt (float): Transverse tensile stress.
        - Yc (float): Transverse compressive stress.
        - S (float): Shear stress.
        - Cross_prod (float): Material cross product.
        - Material_stress_limit (float): Material stress limit.

    If the material considers Hashin's failure criteria parameters:
        - H_xt (float): Hashin's longitudinal tensile stress
        - H_xc (float): Hashin's longitudinal compressive stress
        - H_yt (float): Hashin's transverse tensile stress
        - H_yc (float): Hashin's transverse compressive stress
        - H_st (float): Hashin's shear tensile stress
        - H_sc (float): Hashin's shear compressive stress

    Notes:
    ------
    - The function will always output the material property 'Density'.
      If not defined, its value will be None, which is used later to double
      check the topology optimization conditions requested by the user.
    - The global variables containing the material properties are not
      named in accordance with PEP8 to prevent conflicts with ABAQUS
      internal variables (according to PEP8, constants should be named in
      all capital letters).
    """
    # Redefines the material property, for improved readability, and
    # creates boolean variables defining the existance of a given property.
    material=self.reference_material
    density_properties = False
    elastic_properties = False
    failstrain_properties = False
    failstress_properties = False
    hashindamageinitiation = False

    # Creates the thickness variable.
    global Thickness
    Thickness = self.model.sections[self.section_name].thickness
    if Thickness == None:
        Thickness = 1.0

    # Checks the existence of material density properties.
    global Density
    if hasattr(material, 'density'):
        density_properties = True
        Density = material.density.table[0][0]
    else:
        Density = None

    # Checks the existence of elastic material properties, defined as
    # either ISOTROPIC or through ENGINEERING_CONSTANTS.
    if hasattr(material, 'elastic'):
        elastic_properties = True
        if material.elastic.type == ISOTROPIC:
            global Youngs_modulus, Poisson

            Youngs_modulus, Poisson = material.elastic.table[0]

        elif material.elastic.type == ENGINEERING_CONSTANTS:
            global E11, E22, E33, Nu12, Nu13, Nu23, G12, G13, G23

            (
                E11,
                E22,
                E33,
                Nu12,
                Nu13,
                Nu23,
                G12,
                G13,
                G23,
```

```
            ) = material.elastic.table[0]

        else:
            print(
                "No material properties found in the form of 'ISOTROPIC'"
                "or 'ENGINEERING_CONSTANTS' for material{}."
            ).format(self.material_name)

        # Checks the existence of fail strain parameters.
        if hasattr(material.elastic, 'failStrain'):
            failstrain_properties = True
            global Strain_xt, Strain_xc, Strain_yt, Strain_yc, Strain_s

            (
                Strain_xt,
                Strain_xc,
                Strain_yt,
                Strain_yc,
                Strain_s
            ) = material.elastic.failStrain.table[0]

        # Checks the existence of fail stress parameters.
        if hasattr(material.elastic, 'failStress'):
            failstress_properties = True
            global Xt, Xc, Yt, Yc, S, Cross_prod, Material_stress_limit

            (
                Xt,
                Xc,
                Yt,
                Yc,
                S,
                Cross_prod,
                Material_stress_limit,
            ) = material.elastic.failStress.table[0]

    # Checks the existence of Hashin failure criteria parameters.
    if hasattr(material,'hashinDamageInitiation'):
        hashindamageinitiation = True
        global H_xt, H_xc, H_yt, H_yc, H_st, H_sc

        (
            H_xt,
            H_xc,
            H_yt,
            H_yc,
            H_st,
            H_sc,
        ) = material.hashinDamageInitiation.table[0]

    # Returns the boolean variables, that describe the existence of each
    # material property, back to the class as a self variable.
    self.density_properties = density_properties
    self.elastic_properties = elastic_properties
    self.failstrain_properties = failstrain_properties
    self.failstress_properties = failstress_properties
    self.hashindamageinitiation = hashindamageinitiation

def generate_materials(self):
    """Generate materials method

    For each possible material density, create an ABAQUS material, section,
    and element set. The elements are sorted into sets as a function of
    their density. Then, each set is assigned the material and section that
    corresponds to its density.

    Ex: the material with 'rho_1,0' will be assigned to a set with all
    elements that have a design density of 1.0. This procedure is more
    computationally efficient than creating one material and one section
    for each element, since the number of possible design densities tends
    to be smaller than the number of elements in the model.
    """
    # Increment defined as a function of the number of decimal places (dp)
    # considered in the optimization process.
```

```
        inc = 10.0 ** (-self.dp)

        # For densities between xe_min and 1.0 (rounded at dp).
        rho_range = np.arange(self.xe_min, 1.0 + inc, inc)
        for rho in np.round(rho_range, self.dp):

            # Create a material
            rho_name = 'Rho_' + str(rho).replace(".",",")
            self.model.Material(name = rho_name,
                                objectToCopy = self.reference_material)

            # Determine the values of its properties.
            self.calculate_property(rho)

            # Create a section and assign it to the corresponding material.
            self.model.Section(name = rho_name,
                               objectToCopy = self.reference_section)
            self.model.sections[rho_name].setValues(material = rho_name)

            # Create an empty element set and add elements with equal density.
            self.part.Set(elements = self.part.elements[0:0],
                          name = rho_name)
            self.part.SectionAssignment(self.part.sets[rho_name], rho_name)

    def calculate_property(self,rho):
        """Calculate property method

        Calculates the properties of the material as a function of the design
        density. The function will check the existence of several material
        properties. If the properties exist, calls the prop_val method
        to determine the property value according to the SIMP model.

        Inputs:
        -------
        - rho (float): design density of the element (between xe_min and 1.0).

        Notes: The Poisson coefficients and cross_prod parameters used for the
        fail stress analysis are not updated.
        """
        rho_name = 'Rho_' + str(rho).replace(".",",")
        material = self.model.materials[rho_name]

        # Material density (units of mass/volume)
        if self.density_properties == True:
            material.Density(table = ((self.prop_val(Density, rho), ), ))

        # Elastic properties (defined as ISOTROPIC or by ENGINEERING_CONSTANTS)
        if self.elastic_properties == True:

            if self.reference_material.elastic.type == ISOTROPIC:
                material.Elastic(table = ((self.prop_val(Youngs_modulus, rho),
                                           Poisson), ))

            elif self.reference_material.elastic.type == ENGINEERING_CONSTANTS:
                material.Elastic(type = ENGINEERING_CONSTANTS,
                                 table = ((self.prop_val(E11, rho),
                                           self.prop_val(E22, rho),
                                           self.prop_val(E33, rho),
                                           Nu12,
                                           Nu13,
                                           Nu23,
                                           self.prop_val(G12, rho),
                                           self.prop_val(G13, rho),
                                           self.prop_val(G23, rho)), )
                )
            else:
                print(
                    "Error when checking if the material elastic properties \n"
                    "are defined as ISOTROPIC or ENGINEERING_CONSTANTS in \n"
                    "function 'calculate_property'."
                )

        #Fail strain parameters
        if self.failstrain_properties == True:
```

```
            material.elastic.FailStrain(
                                    table = ((self.prop_val(Strain_xt,rho),
                                              self.prop_val(Strain_xc,rho),
                                              self.prop_val(Strain_yt,rho),
                                              self.prop_val(Strain_yc,rho),
                                              self.prop_val(Strain_s,rho)),))
        #Fail stress parameters
        if self.failstress_properties == True:
            material.elastic.FailStress(
                                    table = ((self.prop_val(Xt,rho),
                                              self.prop_val(Xc,rho),
                                              self.prop_val(Yt,rho),
                                              self.prop_val(Yc,rho),
                                              self.prop_val(S,rho),
                                              Cross_prod,
                                              self.prop_val(
                                              Material_stress_limit,rho)), ))
        #Hashin failure criteria parameters
        if self.hashindamageinitiation == True:
            material.HashinDamageInitiation(
                                        table = ((self.prop_val(H_xt,rho),
                                                  self.prop_val(H_xc,rho),
                                                  self.prop_val(H_yt,rho),
                                                  self.prop_val(H_yc,rho),
                                                  self.prop_val(H_st,rho),
                                                  self.prop_val(H_sc,rho)),))

    def prop_val(self, prop, rho):
        """Property value method

        Wrapper function (or decorator function) that outputs the estimated
        value of a material property based on the SIMP (Solid Isotropic
        Material Penalty) model.

        Inputs:
        -------
        - prop (float): value of the property to be interpolated,
          considering a full density element.
        - rho (float): design density of the element (between xe_min
          and 1.0).

        Notes:
        ------
        This function will try to round the material properties to the same
        number of decimal places as defined in the xe_min variable.

        However, if this leads to a property value of 0.0, the function will
        round the value at 9 orders of magnitude below the original property
        value. Note that this difference in orders of magnitude is the maximum
        difference allowed by ABAQUS for properties such as the Young's
        modulus, as larger differences cause numerical errors.

        The function will choose the largest of the two rounded values. This
        serves two purposes:
        - Allow the differenciation between distinct design variables and the
          resulting material properties.
        - Avoid using an excessive number of decimal places in the material
          definition, which may lead to an unnecessary computational cost.
        """
        # Property value predicted according to the design variable (rho), with
        # lower limite of 9 orders of magniute below the property value.
        max_dp = int( -(math.floor(math.log10(prop)) - 9))
        prop_value = np.around(prop * rho ** (self.p), max_dp)

        # Lower limit imposed by the minimum density defined (xe_min)
        lower_limit = np.round(prop * self.xe_min ** (self.p), -self.dp)

        # If the lower limite imposed by the minimum density is measurable,
        # returns that value. Otherwise, an exception is made and the property
        # will be assigned a value with more decimal places to avoid null
        # properties.
        output = (lower_limit if lower_limit > 0.0 else prop_value)

        return output
```

```
    def generate_output_request(self):
        """Generate output request method

        Define output requests for all steps except the initial (since the
        output is not available at the initial step). For stress dependent
        optimization, request the total strain components defined in ABAQUS by
        the variable 'E' or 'LE' for geometrically linear or non-linear
        problems, respectively.
        """
        # Select the energy output variable.
        if self.nonlinearities == False:
            variables = ('ELEDEN')
        elif self.nonlinearities == True:
            variables = ('ENER')
        else:
            raise Exception(
                "Unexpected value for attribute 'nonlinearities' of the\n"
                "class 'ModelPreparation'."
            )

        # Add strain variables for stress dependent problems and format the
        # variable list.
        if self.opt_method >= 4:
            variables = (variables,) + ('E', 'LE',)
        else:
            variables = (variables,)

        # Request outputs.
        for stp in self.model.steps.keys()[1:]:
            self.model.FieldOutputRequest('TopOpt_FOR_' + stp, stp,
                    variables=variables)
            self.model.HistoryOutputRequest('TopOpt_HOR_' + stp, stp,
                    variables=('ALLWK', ))

    def generate_sets(self):
        """ Generate sets method

        Creates the sets used in the topology optimization process.

        In the particular case of the stress dependent topology optimization,
        the function also generates the ABAQUS node sets required to apply the
        adjoint method.
        """
        # Create a set with all elements that are relevant for
        # the topology optimization.
        self.part.Set(elements = self.all_elmts, name = 'All_Elements')
        self.set_list = self.part.sets.keys()

        # For stress dependent optimization.
        if self.opt_method >= 4:
            nodes = self.part.nodes
            strain_elmts = self.part.elements
            self.part.Set(elements = strain_elmts, name = 'STRAIN_ELEMENTS')
            self.set_list.append('STRAIN_ELEMENTS')

            # Create node sets
            for i in range(0,len(nodes)):
                self.part.Set(nodes = nodes[i:i+1],
                              name = "adjoint_node-" + str(nodes[i].label)
                )

            # Create sets to display stress.
            elmt_sec = self.elmts[0:0]
            for stress_val in np.arange(0, 12):
                set_name = 'stress_val_' + str(stress_val).replace(".",",")
                self.part.Set(elements = elmt_sec, name = set_name)

    def property_update(self, editable_xe):
        """Property update function

        Updates the material property assigned to each element. This process is
        done by sorting the elements into sets as a function of their design
        variable. These sets are then used to assign properties generated in
```

```
        Abaqus by function format_Model to the corresponding elements.

        Inputs:
        -------
        - editable_xe (dict): dictionary with the densities (design variables)
          of each editable element in the model.
        """
        elmt_rho = {}
        for key, value in editable_xe.items():
            elmt_rho.setdefault(value, list()).append(key)

        part_elmts = self.part.elements

        #Prevents Abaqus from updating the color code every iteration.
        session.viewports['Viewport: 1'].disableColorCodeUpdates()

        # Prepare 'for' loop.
        minimum = self.xe_min
        maximum = 1.0 + 10.0 ** (-self.dp)
        inc = 10.0 ** (-self.dp)
        density_values = np.arange(minimum, maximum, inc)

        # Reorganize the elements into sets based on their design density.
        # Unused sets are kept, although they are empty, to prevent the need
        # of re-asigning a color code in future iterations where the set may
        # be needed.
        for rho in np.round(density_values, self.dp):
            if rho in elmt_rho.keys():

                # Initiate empty set and add elements with corresponding design
                # density.
                elmt_set = part_elmts[0:0]
                for label in elmt_rho[rho]:
                    elmt_set += part_elmts[label-1:label]
                self.part.Set(elements = elmt_set,
                              name = 'Rho_' + str(rho).replace(".", ",")
                )

            else:
                # If no elements were added, keep the set as empty.
                self.part.Set(elements = part_elmts[0:0],
                              name = 'Rho_' + str(rho).replace(".", ",")
                )

        # Removes the restriction previously placed.
        # Only executes 1 color code update loop.
        session.viewports['Viewport: 1'].enableColorCodeUpdates()
        session.viewports['Viewport: 1'].disableColorCodeUpdates()

    def get_model_information(self):
        """ Get model information method

        Method that extracts user-defined information from the ABAQUS model.
        Note that the material properties are handled separately by the
        'property_extraction' method.

        The information extracted is described by the following outputs.

        Outputs:
        --------
        - element_type (str): code defining the ABAQUS element type used.
        - set_list (list): list of the sets created by the user.

        For stress dependent topology optimization, the following information
        is also extracted (else, returns None for each case):
        - active_loads (list): list with the keys (names) of the loads that are
          active during the simulation (i.e.: non-supressed loads).
        - active_bc (dict): dictionary with the data of non-zero boundary
          conditions imposed in the model (such as non-zero displacements).
        - node_coords (dict): dictionary with the coordinates of each node.

        When using shell elements in a stress dependent problem, also
        extracts:
        - node_normal_vectors (dict): dictionary with three vectors (normal to
```

```
        each node) used to define the local coordinate system of each element
        and consider the influence of node rotation in the FEA.
        """
        # Get eleemnt type and set list.
        element_type = self.get_element_type()
        set_list = self.return_sets()

        # For stress dependent optimization, identify the boundary conditions
        # and node coordinates.
        if self.opt_method >= 4:
            active_loads = self.get_active_loads()
            active_bc = self.get_active_boundary_conditions()
            node_coords = self.get_node_coordinates()

            # For stress dependent optimization with S4 elements, get the node
            # normal vectors.
            if element_type == 'S4':
                node_normal_vectors = self.get_node_normal_vectors(node_coords)
            else:
                node_normal_vectors = None
        else:
            active_loads, active_bc, node_coords, node_normal_vectors = (
                None, None, None, None)

        return (element_type, set_list, active_loads, active_bc, node_coords,
                node_normal_vectors)

    def return_sets(self):
        """Return set method

        Returns a list of the use-defined ABAQUS sets, which excludes the ones
        generated by the code to store the nodes and elements.
        """
        return self.set_list

    def get_element_type(self):
        """ Get element type method

        Returns the type of the first element in the elmts variable.

        Output:
        -------
        - element_type (str): ABAQUS code defining the element type.
        """
        return str(self.elmts[0].type)

    def get_active_loads(self):
        """ Get active loads method

        Returns a list with the keys of the active loads applied in the ABAQUS
        model.

        Output:
        -------
        - active_loads (list): list with the keys (names) of the loads that are
          active during the simulation (i.e.: non-supressed loads).
        """
        active_loads = []
        loads = self.mdb.models[self.model_name].loads

        for load in loads.keys():
            if loads[load].suppressed == False:
                active_loads.append(load)

        return active_loads

    def get_active_boundary_conditions(self):
        """ Get active boundary conditions method

        Returns a dictionary with the information of the non-zero boundary
        conditions applied in the ABAQUS model.
        The function selects all bouncary conditions and then excludes
        null displacement or rotation conditions.
```

```
        Output:
        -------
        - active_bc (dict): dictionary with the data of non-zero boundary
          conditions imposed in the model (such as non-zero displacements).
        """
        active_bc = {}
        b_condition = self.mdb.models[self.model_name].boundaryConditions

        for key in b_condition.keys():
            if b_condition[key].suppressed == False:
                active_bc[key] = {}

        # Look for non-zero displacements/rotations in the active boundary
        # conditions.
        steps = self.mdb.models[self.model_name].steps
        for key in active_bc.keys():
            for step in steps.keys():
                bc = steps[step].boundaryConditionStates[key]

                # If a bouncary condition has a displacement (cond_1) and it is
                # non-zero (cond_2) or was non-zero in a previous step
                # (cond_3), then save the information of the boundary
                # condition.
                cond_1 = hasattr(bc,"u1")

                if cond_1 == True:

                    cond_2 = any(x not in [0] for x in (bc.u1,
                                                        bc.u2,
                                                        bc.u3,
                                                        bc.ur1,
                                                        bc.ur2,
                                                        bc.ur3)
                    )

                    cond_3 = SET in (bc.u1State,
                                     bc.u2State,
                                     bc.u3State,
                                     bc.ur1State,
                                     bc.ur2State,
                                     bc.ur3State)

                    if cond_2 or cond_3:
                        active_bc[key][step] = {}
                        value = [bc.u1, bc.u2, bc.u3, bc.ur1, bc.ur2, bc.ur3]
                        state = [bc.u1State, bc.u2State, bc.u3State,
                                 bc.ur1State, bc.ur2State, bc.ur3State]

                        active_bc[key][step]['value'] = value
                        active_bc[key][step]['state'] = state

            # Exclude null displacements/rotations found in the active boundary
            # conditions
            if active_bc[key] == {}:
                del active_bc[key]

        return active_bc

    def get_node_coordinates(self):
        """ Get node coordinates method

        Returns a dictionary storing an array with the coordinates of each
        node.

        Output:
        -------
        - node_coordinates (dict): dictionary with the coordinates of each
          node.
        """
        assembly = self.mdb.models[self.model_name].rootAssembly
        nodes = assembly.instances[self.part_name + "-1"].nodes
        node_coordinates = {}
        for node in nodes:
            coords = assembly.getCoordinates(node)
```

```
            node_coordinates[node.label] = np.array(coords)

        return node_coordinates

    def get_node_normal_vectors(self,node_coordinates):
        """ Get node normal vectors method

        Returns a dictionary with the three normal vectors of each node, in
        each element.
        These vectors are determined by the coordinates of the four nodes in
        the element, defining a vector that goes from one node towards the
        next. Therefore, each node will have 1 vector going towards it, and 1
        vector going away from it. These two vectors are used to define the
        normal direction through a cross product.

        Input:
        ------
        - node_coordinates (dict): dictionary with the coordinates of each
        node.

        Output:
        -------
        - node_normal_vectors (dict): dictionary with three vectors (normal to
          each node) used to define the local coordinate system of each element
          and consider the influence of node rotation in the FEA.
        """

        node_normal_vect = {}

        for elmt in self.all_elmts:

            # Identify nodes and create unit_vectors.
            node_normal_vect[elmt.label] = {}
            unit_vector = np.array([1,1,1,1])
            node_1, node_2, node_3, node_4 = elmt.connectivity + unit_vector

            # Determine in-plane vectors at each node.
            v12 = node_coordinates[node_2] - node_coordinates[node_1]
            v23 = node_coordinates[node_3] - node_coordinates[node_2]
            v34 = node_coordinates[node_4] - node_coordinates[node_3]
            v41 = node_coordinates[node_1] - node_coordinates[node_4]

            # Determine normal vectors.
            node_normal_vect[elmt.label][node_1] = self.normal_vectors(v41,v12)
            node_normal_vect[elmt.label][node_2] = self.normal_vectors(v12,v23)
            node_normal_vect[elmt.label][node_3] = self.normal_vectors(v23,v34)
            node_normal_vect[elmt.label][node_4] = self.normal_vectors(v34,v41)

        return node_normal_vect

    def normal_vectors(self,v1,v2):
        """ Node normal vectors method

        Determines the three normal vectors of a node through a cross product
        between two vectors, one going towards the node, and one going away
        from the node.

        Additionally, this method checks if the two vectors used are parallel
        to avoid numerical errors.

        Inputs:
        -------
        - v1 (array): vector going towards the node.
        - v2 (array): vector going away from the node.

        Output:
        -------
        - node_normal_vectors (dict): dictionary with three vectors (normal to
          each node) used to define the local coordinate system to be
          considered in one node (and account for the influence of node
          rotation in the FEA process).
        """
        # Determines the normal vector.
        vector = self.calculate_normal_vector(v1, v2)
```

```
        # Checks if it is parallel to [0,1,0].
        parallel_check = self.parallel_vector_check(vector)
        if parallel_check == False:
            node_v1 = np.cross(np.array([0,1,0]), vector)
        else:
            node_v1 = np.array([0,0,1])

        # Determines and normalizes in-plane node normal vectors.
        node_v1 = node_v1/np.linalg.norm(node_v1)
        node_v2 = np.around(np.cross(vector, node_v1), 5)

        node_normal_vectors = {"v1":node_v1, "v2":node_v2, "vn":vector}

        return node_normal_vectors

    def calculate_normal_vector(self,v1,v2):
        """ Calculate normal vector method

        Determines a unit vector that is normal to two input vectors, using the
        cross product of two arrays.

        Inputs:
        -------
        - v1 (array): first vector.
        - v2 (array): second vector.

        Output:
        -------
        - normal_vector (array): unit vector normal to v1 and v2.
        """
        cross_prod = np.cross(v1,v2)
        vector_norm = np.linalg.norm(cross_prod)
        ratio = cross_prod/vector_norm

        return np.around(ratio,5)

    def parallel_vector_check(self,vector):
        """ Parallel vector check method

        Returns a boolean variable stating if a vector is parallel with the
        axis [0,1,0] (True) or not (False).

        The axis [0,1,0] is used as reference for the local axys system used
        in shell element. This procedure is in accordance with the method
        described in the book Finite Element Procedures (2nd edition), written
        by Klaus-Jurgen Bathe, in section 5.4, page 439.

        Inputs:
        -------
        - vector (array): vector to be evaluated.

        Output:
        -------
        - check (bool): indicates if the vector is parallel to [0,1,0] (True)
          or not (False).
        """
        unit_vector = np.array([0,1,0])
        parallel_check = np.cross(unit_vector, vector)

        order_of_magnitude = int(math.floor(np.linalg.norm(parallel_check)) -5)
        order_of_magnitude = int(-order_of_magnitude)

        parallel_check = np.around(parallel_check, order_of_magnitude)

        return np.array_equal(np.array([0,0,0]), parallel_check)


#%% State and Adjoint model submission, and sensitvities.
class AbaqusFEA():
    """ ABAQUS Finite Element Analysis class

    This class is responsible for the execution of the finite element analysis
    in ABAQUS, as well as the extraction of the necessary outputs for the
```

```
    topology optimization process.

    Attributes:
    -----------
    - iteration (int): number of the current iteration in the topology
      optimization process.
    - mdb (Mdb): ABAQUS model database.
    - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.
    - ae (dict): dictionary with the sensitivity of the objective function to
      changes in each design variable.
    - p (float): SIMP penalty factor.
    - element_type (str): ABAQUS code defining the element type.
    - last_frame (int): variable defining if only the results of the last
      frame should be considered or not (only last frame = 1 / all frames = 0).
    - nDomains (int): number of job domains to be considered in the FEA.
    - nCPUs (int): number of CPUs to be used in the execution of the FEA.
    - opt_method (int): variable defining the optimization method to be used.
    - node_normal_vector (dict): dictionary with three vectors (normal to
      each node) used to define the local coordinate system of each element
    - nonlinearities (boolean): Indicates if the problem considers geometrical
      nonlinearities (True) or not (False).
    - instance_name (str): name of the ABAQUS part when referenced from the
      assembly module.
    - instance (OdbInstance): ABAQUS part when referenced from the assembly
      module.

    Methods:
    --------
    - run_simulation(iteration, xe): submits the FEA, waits for its completion,
      and organizes the data extraction from the ABAQUS odb file.

    Auxiliary methods:
    ------------------
    - init_dictionaries(opdb): creates the dictionaries used to store the data
      extracted from the ABAQUS odb file.
    - execute_FEA(): submits the FEA, waits for its completion and opens the
      odb file created.
    - compliance_sensitivity(strain_energy, xe): determines the compliance
      sensitivity based on the straine nergy and on the design variables of
      each element.
    - get_strain_energy(frame, strain_energy): determines the strain energy in
      the current frame and updates the data record if necessary.
    - get_compliance(step, compliance): extracts the maximum value of the
      compliance observed in the model at the current step. Updates the data
      record if necessary.
    - get_local_coord_system(opdb): determines the local coordinate system set
      by ABAQUS at each node of a shell element. Returns and empty dictionary
      for other elements.
    - get_strains(frame, strain, strain_mag): extracts the maximum strain, in
      each integration point at the current frame, and updates the data record
      if necessary.
    - get_rotations(opdb, frame, rotation, rotation_mag): extracts the maximum
      rotatation in each node of a shell element, in the current frame, and
      updates the data record if necessary.
    - get_displacements(opdb, frame, displacement, displacement_mag): extracts
      the maximum node displacement in the current frame and updates the data
      record if necessary.
    - converted_node_rotation(node_rotation): converts the node rotations from
      the global to the local coordinate system.
    """

    def __init__(
            self, iteration, mdb, model_name, part_name, ae, p, element_type,
            last_frame, nDomains, nCPUs, opt_method, node_normal_vector,
            nonlinearities
        ):

        self.iteration = iteration
        self.mdb = mdb
        self.model_name = model_name
        self.part_name = part_name
        self.ae = ae
        self.p = p
```

```
        self.element_type = element_type
        self.last_frame = last_frame
        self.nDomains = nDomains
        self.nCPUs = nCPUs
        self.opt_method = opt_method
        self.node_normal_vector = node_normal_vector
        self.nonlinearities = nonlinearities
        self.instance_name = self.part_name.upper()+'-1'
        self.instance = None

    def run_simulation(self, iteration, xe):
        """ Run simulation method

        This method performs the following actions:
        - Update the iteration number (class attribute);
        - Submit a job, wait for its completion, and open the odb file.
        - Initialize dictionaries to store the odb information in them.
        - Iterate through every step and frame of the odb file, extracting
          the necessary information.
        - Close the odb and delete the ABAQUS generated files.

        This method will always extract the compliance and compliance
        sensitivity.

        In stress dependent problems, also extracts the element strains at
        the integration points, and the node displacements.
        If the stress dependent problem was also solved with 'S4' type
        elements, the method will also extract the node rotations and the
        local coordinate assigned to each node by ABAQUS.

        Inputs:
        -------
        - iteration (int): number of the current iteration in the topology
          optimization process.
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.

        Outputs:
        --------
        - compliance (float): maximum value of the compliance observed during
          the FEA.
        - ae (dict): dictionary with the compliance sensitivity of each
          element.
        - strain (dict): dictionary of dictionaries, storing the maximum strain
          of each integration point (second key) in each element (first key).
        - displacement (dict): dictionary with the displacement of each
          node.
        - rotation (dict): dictionary with the rotation of each node.
        - local_coord_sys (dict): dictionary with the local coordinate system
          assigned to each element by ABAQUS.
        """

        # Update iteration counter, submit simulation and initialize
        # dictionaries.
        self.iteration = iteration
        opdb = self.execute_FEA()
        self.instance = opdb.rootAssembly.instances[self.instance_name]

        (
            strain_energy,
            strain,
            strain_mag,
            rotation,
            rotation_mag,
            displacement,
            displacement_mag,
        ) = self.init_dictionaries(opdb)

        compliance = 0

        # Determine the local coordinate system, if using shell elements in a
        # stress dependent problem. Else, returns an empty dictionary.
        local_coord_sys = self.get_local_coord_system(opdb)
```

```
        # Loop through each step and frame, extracting the information needed.
        for stp in opdb.steps.values():

            compliance = self.get_compliance(stp, compliance)

            frames = [stp.frames[-1]] if self.last_frame == 1 else stp.frames
            for frame in frames:
                strain_energy = self.get_strain_energy(frame, strain_energy)

                # For stress dependent problems.
                if self.opt_method >= 4:
                    strain, strain_mag = self.get_strains(
                        opdb, frame, strain, strain_mag
                    )

                    displacement, displacement_mag = self.get_displacements(
                        opdb, frame, displacement, displacement_mag
                    )

                    if self.element_type == 'S4':
                        rotation, rotation_mag = self.get_rotations(
                            opdb, frame, rotation, rotation_mag
                        )

        # Convert the node rotation to the element local coordinate system.
        if self.opt_method >= 4 and self.element_type == 'S4':
            rotation = self.convert_node_rotation(rotation)

        # Determine the compliance sensitivity to changes in the design
        # variables.
        ae = self.compliance_sensitivity(strain_energy, xe)
        self.ae = ae

        #Closes odb and removes files created by ABAQUS.
        opdb.close()
        remove_files(iteration, 'Design_Job')
        del self.mdb.jobs['Design_Job'+str(iteration)]

        return compliance, ae, strain, displacement, rotation, local_coord_sys

    def init_dictionaries(self, opdb):
        """ Initialize dictionaries method

        Creates the dictionaries, and necessary entries, used to store the data
        extracted from the ABAQUS odb file.

        Input:
        ------
        - opdb (Odb): ABAQUS output data base.

        Output:
        -------
        - dictionaries (tuple): dictionaries created to store the strain
          energy, strain, strain magnitude, rotation, rotation magnitude,
          displacement, and displacement magnitude.
        """
        strain_energy = {}
        strain, strain_mag = {}, {}
        rotation, rotation_mag = {}, {}
        displacement, displacement_mag = {}, {}

        # For stress dependent problems:
        if self.opt_method >= 4:

            elmts = self.instance.elements
            nodes = self.instance.nodes

            for elmt in elmts:
                strain[elmt.label] = {}
                strain_mag[elmt.label] = {}

            for node in nodes:
                rotation[node.label] = 0.0
                rotation_mag[node.label] = 0.0
```

```
                displacement[node.label] = 0.0
                displacement_mag[node.label] = 0.0

        dictionaries = (strain_energy, strain, strain_mag, rotation,
                        rotation_mag, displacement, displacement_mag
        )

        return dictionaries

    def execute_FEA(self):
        """ Execute finite element analysis method

        Submits the ABAQUS job, waits for its completion, and then opens and
        returns its output database file (odb).

        Output:
        -------
        - opdb (Odb): ABAQUS output data base.
        """
        job_name = 'Design_Job'+str(self.iteration)
        odb_name = job_name + '.odb'

        mdb.Job(name = job_name, model = self.model_name,
                numDomains = self.nDomains, numCpus = self.nCPUs).submit()
        mdb.jobs[job_name].waitForCompletion()

        opdb = openOdb(odb_name)

        return opdb

    def compliance_sensitivity(self, strain_energy, xe):
        """ Compliance sensitivity method

        Determines the compliance sensitivity based on the strain energy and on
        the design variable (design density) of each element.
        Stores this information as a class attribute.

        Inputs:
        -------
        - strain_energy (dict): dictionary with the strain energy value for
          each element.
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.

        Outputs:
        --------
        - ae (dict): dictionary with the sensitivity of the objective function
          to changes in each design variable.
        """
        ae = {}
        for key in xe:
            ae[key] = -self.p * strain_energy[key] / xe[key]

        self.ae = ae

        return self.ae

    def get_strain_energy(self, frame, strain_energy):
        """ Get strain energy method

        Method used to determine the maximum strain energy of each element in
        the current frame, considering the sum of both elastic and plastic
        components.
        The function compares the values observed in the current frame with
        previous records and, if necessary, updates the record.

        Inputs:
        -------
        - frame (OdbFrame): current frame of the ABAQUS odb.
        - strain_energy (dict): dictionary with the strain energy value for
          each element.

        Output:
        -------
```

```
        - strain_energy (dict): updated dictionary with the strain energy value
          for each element.
        """
        temp_dict = {}
        attributes = 'data', 'elementLabel', 'instance'

        if self.nonlinearities == False:
            # Elastic strain energy component.
            # Create a generator that selects the nodes that belong to the
            # editable instance. Note that item[2] is the instance name.
            sener = frame.fieldOutputs['ESEDEN']
            sener_info = map(attrgetter(*attributes), sener.values)
            relevant_sener_info = (
                item for item in sener_info if item[2] == self.instance
            )

            for elmt in relevant_sener_info:
                elmt_data = elmt[0]
                elmt_label = elmt[1]
                temp_dict[elmt_label] = elmt_data

        elif self.nonlinearities == True:
            # Elastic strain energy component.
            # Create a generator that selects the nodes that belong to the
            # editable instance. Note that item[2] is the instance name.
            sener = frame.fieldOutputs['SENER']
            sener_info = map(attrgetter(*attributes), sener.values)
            relevant_sener_info = (
                item for item in sener_info if item[2] == self.instance
            )

            for elmt in relevant_sener_info:
                elmt_data = elmt[0]
                elmt_label = elmt[1]
                temp_dict[elmt_label] = elmt_data

            # Adds plastic strain energy component.
            # Create a generator that selects the nodes that belong to the
            # editable instance. Note that item[2] is the instance name.
            pener = frame.fieldOutputs['PENER']
            pener_info = map(attrgetter(*attributes), pener.values)
            relevant_pener_info = (
                item for item in pener_info if item[2] == self.instance
            )

            for elmt in relevant_pener_info:
                elmt_data = elmt[0]
                elmt_label = elmt[1]
                temp_dict[elmt_label] += elmt_data

        else:
            raise Exception(
                "Unexpected value for attribute 'nonlinearities' of class \n"
                "AbaqusFEA.")

        # If strain_energy is not empty, selects the maximum value.
        # Otherwise, assigns the first value.
        if strain_energy:
            for key in strain_energy.keys():
                strain_energy[key] = max(strain_energy[key], temp_dict[key])
        else:
            strain_energy = temp_dict

        return strain_energy

    def get_compliance(self, step, compliance):
        """ Get compliance method

        Method used to extract the value of the maximum value of the compliance
        observed in the model, at the current step.
        The function compares the value observed in the current step with
        previous records and, if necessary, updates the record.

        Inputs:
```

```
        -------
        - step (OdbStep): current step of the ABAQUS odb.
        - compliance (float): maximum value of the compliance observed during
          the FEA.

        Output:
        -------
        - compliance (float): maximum value of the compliance observed during
          the FEA.
        """

        model_data = (step.historyRegions['Assembly ASSEMBLY']
                      .historyOutputs['ALLWK'].data)

        current_compliance = max([item[1] for item in model_data])

        compliance = max(compliance, current_compliance)

        return compliance

    def get_local_coord_system(self, opdb):
        """ Get local coordinate system method
        Method used to determine the local coordinate system set by ABAQUS at
        each element.
        If the element used is not of the type 'S4', an empty dictionary is
        returned.

        Input:
        ------
        - opdb (Odb): ABAQUS output data base.

        Output:
        -------
        - coord_sys (dict): dictionary with the local coordinate systems set by
          ABAQUS for each element.
        """
        coord_sys = {}

        if self.element_type == "S4":
            first_step = opdb.steps.keys()[0]
            attributes = 'elementLabel', 'localCoordSystem', 'instance'
            # Create a generator that selects the elements that belong to the
            # editable instance. Note that item[2] is the instance name.
            temp_coord = opdb.steps[first_step].frames[-1].fieldOutputs['S']
            stress_coord = map(attrgetter(*attributes), temp_coord.values)
            relevant_rotations = (
                item for item in stress_coord if item[2] == self.instance
            )
            # Rounds the coordinates of the vector to avoid float errors.
            for item in relevant_rotations:
                item_label = item[0]
                item_coord = item[1]
                coord_sys[item_label] = np.around(item_coord, 5)

        return coord_sys

    def get_strains(self, opdb, frame, strain, strain_mag):
        """ Get strains method
        Method used to extract the maximum strain in each integration point.
        The function compares the value observed in the current frame with
        previous records and, if necessary, updates the record.

        The strains are stored in the ABAQUS variables 'E' or 'LE' depending
        on the step being geometrically linear or non-linear, respectively.

        Inputs:
        -------
        - opdb (Odb): ABAQUS output data base.
        - frame (OdbFrame): current frame of the ABAQUS odb.
        - strain (dict): dictionary of dictionaries, storing the maximum strain
          of each integration point (second key) in each element (first key).
        - strain_mag (dict): dictionary of dictionaries, storing the magnitude
          of the maximum strain of each integration point (second key) in each
          element (first key).
```

```
        Output:
        -------
        - strain (dict): dictionary of dictionaries, storing the maximum strain
          of each integration point (second key) in each element (first key).
        - strain_mag (dict): dictionary of dictionaries, storing the magnitude
          of the maximum strain of each integration point (second key) in each
          element (first key).
        """
        # Indicate that the data should be extracted from the integration
        # points.
        instance_name = self.part_name.upper()+'-1'
        instance = opdb.rootAssembly.instances[instance_name]
        region = instance.elementSets['STRAIN_ELEMENTS']
        position = INTEGRATION_POINT

        # The strains are stored in the ABAQUS variables 'E' or 'LE' depending
        # on the step being geometrically linear or non-linear, respectively.
        if 'E' in frame.fieldOutputs:
            temp_strain = frame.fieldOutputs['E'].getSubset(
                region = region,
                position = position
            )
        elif 'LE' in frame.fieldOutputs:
            temp_strain = frame.fieldOutputs['LE'].getSubset(
                region = region,
                position = position
            )
        else:
            raise Exception("None of the strain variables 'E' or 'LE' were "
                            "detected by the FEA function when performing a "
                            "stress dependent optimization.")

        attributes = 'data','elementLabel','maxPrincipal','integrationPoint'
        strains = map(attrgetter(*attributes), temp_strain.values)

        for item in strains:
            item_data = item[0]
            item_label = item[1]
            item_maxPrincipal = item[2]
            item_intPoint = item[3]

            # Cond_1 == True indicates that no previous value has been stored.
            cond_1 = item_intPoint not in strain_mag[item_label].keys()

            # Cond_2 == True indicates that the current value is larger than
            # the previous record.
            if cond_1 == False:
                prev_val = abs(strain_mag[item_label][item_intPoint])
                cond_2 = abs(item_maxPrincipal) >= prev_val
            else:
                cond_2 = False

            # If its the first dictionary entry, or there is a larger value,
            # update the dictionary entry.
            if cond_1 or cond_2:
                strain_mag[item_label][item_intPoint] = item_maxPrincipal
                if self.element_type in ['C3D8']:
                    strain_vector = item_data
                elif self.element_type in ['CPS4', 'CPE4', 'S4']:
                    strain_vector= np.array(
                        [item_data[0], item_data[1], item_data[3]]
                    )
                else:
                    raise Exception("Unexpected strain vector at the "
                                    "integration points.")

                strain[item_label][item_intPoint] = strain_vector

        return strain, strain_mag

    def get_rotations(self, opdb, frame, rotation, rotation_mag):
        """ Get rotations method
```

```
        Method used to extract the maximum rotation in each node.
        The function compares the value observed in the current frame with
        previous records and, if necessary, updates the record.

        Inputs:
        -------
        - opdb (Odb): ABAQUS output data base.
        - frame (OdbFrame): current frame of the ABAQUS odb.
        - rotation (dict): dictionary storing the maximum rotation in each
          node.
        - rotation_mag (dict): dictionary of dictionaries, storing the
          magnitude of the maximum rotation in each node.

        Output:
        -------
        - rotation (dict): dictionary storing the maximum rotation in each
          node.
        - rotation_mag (dict): dictionary of dictionaries, storing the
          magnitude of the maximum rotation in each node.
        """
        # Create a generator that selects the nodes that belong to the
        # editable instance. Note that item[3] is the instance name.
        temp_rot = frame.fieldOutputs['UR']
        attributes = 'data', 'nodeLabel', 'magnitude', 'instance'
        rotations = map(attrgetter(*attributes), temp_rot.values)
        relevant_rotations = (
            item for item in rotations if item[3] == self.instance
        )

        for item in relevant_rotations:
            item_data = item[0]
            item_nodeLabel = item[1]
            item_magnitude = item[2]

            # Cond_1 == True indicates that the current value is larger than
            # the previous record.
            cond_1 = (item_magnitude >= abs(rotation_mag[item_nodeLabel]))

            # If its the first dictionary entry, or there is a larger value,
            # update the dictionary entry.
            if cond_1:
                rotation_mag[item_nodeLabel] = item_magnitude
                rotation[item_nodeLabel] = item_data

        return rotation, rotation_mag

    def get_displacements(self, opdb, frame, displacement, displacement_mag):
        """ Get displacements method

        Method used to extract the maximum displacement in each node.
        The function compares the value observed in the current frame with
        previous records and, if necessary, updates the record.

        Inputs:
        -------
        - opdb (Odb): ABAQUS output data base.
        - frame (OdbFrame): current frame of the ABAQUS odb.
        - displacement (dict): dictionary storing the maximum dispalcement in
          each node.
        - displacement_mag (dict): dictionary of dictionaries, storing the
          magnitude of the maximum displacement in each node.

        Output:
        -------
        - displacement (dict): dictionary storing the maximum dispalcement in
          each node.
        - displacement_mag (dict): dictionary of dictionaries, storing the
          magnitude of the maximum displacement in each node.
        """
        # Create a generator that selects the nodes that belong to the
        # editable instance. Note that item[3] is the instance name.
        temp_disp = frame.fieldOutputs['U']
        attributes = 'data', 'nodeLabel', 'magnitude', 'instance'
        displacements = map(attrgetter(*attributes), temp_disp.values)
```

```
        relevant_displacements = (
            item for item in displacements if item[3] == self.instance
        )

        for item in relevant_displacements:
            item_data = item[0]
            item_nodeLabel = item[1]
            item_magnitude = item[2]

            # Cond_1 == True indicates that the current value is larger than
            # the previous record.
            cond_1 = (item_magnitude >= abs(displacement_mag[item_nodeLabel]))

            # If its the first dictionary entry, or there is a larger value,
            # update the dictionary entry.
            if cond_1:
                displacement_mag[item_nodeLabel] = item_magnitude
                displacement[item_nodeLabel] = item_data.copy()
                displacement[item_nodeLabel].resize(3)

        return displacement, displacement_mag

    def convert_node_rotation(self, node_rotation):
        """ Convert node rotation method

        Converts the node rotations from the global to the local
        coordinate system.

        Input:
        ------
        - node_rotation (dict): dictionary with the rotations of each
          node, in each element.

        Output:
        -------
        - converted_node_rotation (dict): dictionary with the converted
          node rotations.
        """
        converted_node_rotation = {}

        for elmt in self.node_normal_vector.keys():
            converted_node_rotation[elmt] = {}

            for node in self.node_normal_vector[elmt].keys():

                line_1 = self.node_normal_vector[elmt][node]["v1"]
                line_2 = self.node_normal_vector[elmt][node]["v2"]
                line_3 = self.node_normal_vector[elmt][node]["vn"]

                transformation_matrix = np.array(
                    [line_1,
                     line_2,
                     line_3]
                )

                converted_node_rotation[elmt][node] = \
                    np.dot(transformation_matrix, node_rotation[node])

        return converted_node_rotation


def init_AdjointModel(
        mdb, model_name, part_name, material_name, section_name, nodes, elmts,
        p, planar, element_type, elmt_volume, node_normal_vector, opt_method,
        nDomains, nCPUs, last_frame
    ):
    """ Initialize Adjoint model function

    Creates and returns an AdjointModel, if it is necessary for the
    optimization process requested. Otherwise, returns None.

    Inputs:
    -------
    - mdb (Mdb): ABAQUS model database.
```

```
    - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.
    - material_name (str): Name of the ABAQUS material to be considered.
    - section_name (str): Name of the ABAQUS material section to be considered.
    - nodes (MeshNodeArray): mesh node array from ABAQUS with all nodes that
      belong to elements considered in the topology optimization process.
    - elmts (MeshElementArray): element_array from ABAQUS with the relevant
      elements in the model.
    - p (float): SIMP penalty factor.
    - planar (int): variable identifying the type of part considered (2D or
      3D).
    - element_type (str): ABAQUS code defining the element type.
    - elmt_volume (dict): dictionary with the element volume of each element.
    - node_normal_vector (dict): dictionary with three vectors (normal to
      each node) used to define the local coordinate system of each element.
    - opt_method (int): variable defining the optimization method to be used.
    - nDomains (int): number of job domains to be considered in the FEA.
    - nCPUs (int): number of CPUs to be used in the execution of the FEA.
    - last_frame (int): variable defining if only the results of the last
      frame should be considered or not (only last frame = 1 / all frames = 0).

    Output:
    -------
    - adj_model (class): adjoint model class.
    """

    if opt_method >= 4:
        adj_model = AdjointModel(
            mdb, model_name, part_name, material_name, section_name, nodes,
            elmts, p, planar, element_type, elmt_volume, node_normal_vector,
            nDomains, nCPUs, last_frame
        )
    else:
        adj_model= None

    return adj_model


class AdjointModel():
    """ Adjoint model class

    Analogous to the AbaqusFEA class, the Adjoint model class is responsible
    for the execution of the finite element analysis of the adjoint model in
    ABAQUS, as well as the extraction of the necessary outputs for the
    topology optimization process.

    Attributes:
    -----------
    - mdb (Mdb): ABAQUS model database.
    - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.
    - material_name (str): Name of the ABAQUS material to be considered.
    - section_name (str): Name of the ABAQUS material section to be considered.
    - nodes (MeshNodeArray): mesh node array from ABAQUS with all nodes that
      belong to elements considered in the topology optimization process.
    - elmts (MeshElementArray): element_array from ABAQUS with the relevant
      elements in the model.
    - p (float): SIMP penalty factor.
    - planar (int): variable identifying the type of part considered (2D or
      3D).
    - element_type (str): ABAQUS code defining the element type.
    - elmt_volume (dict): dictionary with the element volume of each element.
    - node_normal_vector (dict): dictionary with three vectors (normal to
      each node) used to define the local coordinate system of each element.
    - nDomains (int): number of job domains to be considered in the FEA.
    - nCPUs (int): number of CPUs to be used in the execution of the FEA.
    - last_frame (int): variable defining if only the results of the last
      frame should be considered or not (only last frame = 1 / all frames = 0).
    - iteration (int): number of the current iteration in the topology
      optimization process.
    - part (Part): ABAQUS part to be optimized.
    - all_elmts (MeshElementArray): element_array from ABAQUS with all the
      elements in the part.
    - material_type (Material_type): ABAQUS code defining the type of the
```

```
    material considered.
  - shell_thickness (float): Total thickness of the shell element.
  - inv_int_p (float): inverse of the number of integration points in the
    model.
  - elmt_points (range): range of evaluation points (nodes or integration
    points) in the element.
  - deformation_vector (dict): dictionary with the deformation vectors in
    each element node.
  - deformation_int (dict): dictionary with the deformation vectors in each
    element integration point.
  - global_node_force (dict): dictionary with the nodal forces to the applied
    in the adjoint model.
  - stress_vector (dict): dictionary with the stress vectors in each element
    node.
  - stress_vector_int (dict): dictionary with the stress vectors in the
    integration points of each element.
  - b_matrix (dict): dictionary with the B matrix determined in each node of
    each element.
  - b_matrix_int (dict): dictionary with the B matrix determined in each
    integration point of each element.
  - jacobian (dict): dictionary with the Jacobian matrix determined in each
    node of each element.
  - jacobian_int (dict): dictionary with the Jacobian matrix determined in
    each integration point of each element.
  - c_matrix (dict): dictionary with the D (stiffness) matrix determined for
    each element.
  - p_norm_spf (dict): dictionary with the component of the derivative of the
    p-norm function that contains the stress penalization factor.
  - p_norm_displacement (dict): dictionary with the component of the
    derivative of the p-norm function that contains the node displacements.

  Methods:
  --------
  - run_adjoint_simulation(node_displacement, xe, node_rotation,
    node_coordinates, local_coord_system, q, active_bc, active_loads,
    iteration): prepares, submits, and extracts data from the adjoint model.
  - determine_stress_and_deformation(node_displacement, xe, node_rotation,
    node_coordinates, local_coord_system): determines the stress and strain
    vectors (as well as Jacobian and B matrixes) in the element nodes and
    integration points.
  - determine_adjoint_load(q): determines the nodal adjoint load.
  - stress_sensitivity(xe, q, state_strain, adjoint_strain): determines the
    p-norm maximum Von-Mises stress sensitivity to changes in the design
    variables.

  Auxiliary methods:
  ------------------
  - init_dictionaries(opdb): initializes the dictionaries required to store
    the outputs from the adjoint model.
  - execute_adjoint_FEA(): submits the adjoint model.
  - get_adjoint_strain(opdb, frame, strain, strain_mag): extracts the strain
    values from the adjoint model.
  - non_zero_force_check(node_label): check if the an adjoint nodal load is
    not zero.
  - apply_adjoint_loads(active_bc, active_loads): applies the adjoint loads.
  - apply_nodal_load(node_label): applies the adjoint load to a given node.
  - remove_adjoint_loads(active_bc, active_loads): removes the adjoint loads
    from the ABAQUS model.
  - supress_dispalcement_BC(bc_list): supresses non-zero displacement
    boundary conditions from the state model.
  - resume_displacement_BC(bc_list): resumes non-zero displacement boundary
    conditions from the state model.
  - coordinate_vectors(elmt, node_coords): sorts the element node coordinates
    into three lists.
  - rotation_vectors(elmt, node_rotation): sorts the node rotations in to
    two lists.
  - elmt_node_displacement_vect(elmt, node_displacement,
    node_rotation = None): combines the node displacements and rotations into
    a single, ordered, vector.
  - node_normal_vectors(elmt): sorts the node normal vectors into three
    lists.
  - vect_transf_matrix(elmt, local_coord_system): creates a transformation
    matrix for vectors, converting them from the global to the local
    coordinate system.
```

```
    - matx_transf_matrix(elmt, local_coord_system): creates a transformation
      matrix for matrixes, converting them from the global to the local
      coordinate system.
    - surface_selection(elmt, node_displacement_vector, s, t, v, x_coord,
      y_coord, z_coord, v1_vector, v2_vector, vn_vector, a_rot, b_rot,
      elmt_formulation): selects the shell surface with the largest
      stress-strain state.
    - xe_all(label, xe): returns the design density of an element, even if it
      does not belong to the editable region.
    - determine_stress_vector(elmt, vector_trans_m, matrix_trans_m,
      deformation, xe): determines the stress vector in a given point, based on
      the strain and element design density.
    - multiply_VM_matrix(v1, v2): returns the product of two vectors by the
      Von-Mises stress matrix.
    - local_c_matrix(matrix_trans_m, elmt): converts the element C matrix to
      the local coordinate system.
    - determine_d_pnorm_displacement(xe, state_strain, adjoint_strain):
      determines the component of the stress sensitivity that is dependent
      on the node displacements.
    - determine_d_pnorm_spf(xe, q): determines the component of the stress
      sensitivity that is dependent on the stress penalization factor.
    """
    def __init__(
            self, mdb, model_name, part_name, material_name, section_name,
            nodes, elmts, p, planar, element_type, elmt_volume,
            node_normal_vector, nDomains, nCPUs, last_frame
        ):

        self.mdb = mdb
        self.model_name = model_name
        self.part_name = part_name
        self.material_name = material_name
        self.section_name = section_name
        self.nodes = nodes
        self.elmts = elmts
        self.p = p
        self.planar = planar
        self.element_type = element_type
        self.elmt_volume = elmt_volume
        self.node_normal_vector = node_normal_vector
        self.nDomains = nDomains
        self.nCPUs = nCPUs
        self.last_frame = last_frame
        self.iteration = None
        self.part = mdb.models[model_name].parts[part_name]
        self.all_elmts = self.part.elements
        self.material_type = (mdb.models[model_name]
                              .materials[material_name].elastic.type)

        shell_thickness = (mdb.models[model_name]
                           .sections[section_name].thickness)

        if shell_thickness == None:
            self.shell_thickness = 1.0
        else:
            self.shell_thickness = shell_thickness

        if element_type in ["CPS4", "CPE4", "S4"]:
            self.inv_int_p = 1.0 / (4.0 * len(elmts))
        elif element_type in ["C3D8"]:
            self.inv_int_p = 1.0 / (8.0 * len(elmts))
        else:
            raise Exception(
                "Unexpected 'element_type' attribute in 'AdjointModel' class."
            )

        self.elmt_points = range(0, len(self.elmts[0].connectivity))
        self.deformation_vector = {}
        self.deformation_int = {}
        self.global_node_force = {}
        self.stress_vector = {}
        self.stress_vector_int = {}
        self.b_matrix = {}
        self.b_matrix_int = {}
```

```
        self.jacobian = {}
        self.jacobian_int = {}
        for node in self.nodes:
            self.deformation_vector[node.label] = {}
            self.global_node_force[node.label] = 0
            self.stress_vector[node.label] = {}
            self.b_matrix[node.label] = {}
            self.jacobian[node.label] = {}

        self.c_matrix = {}
        c_matrix_temp = c_matrix_function(self.element_type,
                                          self.material_type, self.planar)

        for elmt in self.all_elmts:
            self.c_matrix[elmt.label] = c_matrix_temp
            self.deformation_int[elmt.label] = {}
            self.stress_vector_int[elmt.label] = {}
            self.b_matrix_int[elmt.label] = {}
            self.jacobian_int[elmt.label] = {}

    def run_adjoint_simulation(
            self, node_displacement, xe, node_rotation, node_coordinates,
            local_coord_system, q, active_bc, active_loads, iteration
        ):
        """ Run adjoint simulation method

        This method performs the following actions:
        - Determine the loads of the adjoint model, and apply them.
        - Submit a job, wait for its completion, and open the odb file.
        - Initialize dictionaries to store the odb strain information.
        - Iterate through every step and frame of the odb file, extracting
          the necessary information.
        - Revert the changes made when applying the adjoint loads.
        - Close the odb and delete the ABAQUS generated files.

        Inputs:
        -------
        - node_displacement (dict): dictionary with the displacement of each
          node.
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.
        - node_rotation (dict): dictionary with the node rotations of nodes in
          each element.
        - node_coordinates (dict): dictionary with the coordinates of each
          node.
        - local_coord_system (dict): dictionary with the local coordinate
          systems of each element.
        - q (float): P-norm factor used in the stress approximation function.
          Here referred as 'q' to avoid confusion with the SIMP penalty factor.
        - active_bc (dict): dictionary with the data of non-zero boundary
          conditions imposed in the model (such as non-zero displacements).
        - active_loads (list): list with the keys (names) of the loads that are
          active during the simulation (i.e.: non-supressed loads).
        - iteration (int): number of the current iteration in the topology
          optimization process.

        Outputs:
        --------
        - strain (dict): dictionary of dictionaries, storing the maximum strain
          of each integration point (second key) in each element (first key).
        """
        # Determine the adjoint loads, apply them, and submit the model.
        self.iteration = iteration
        self.determine_stress_and_deformation(node_displacement, xe,
                node_rotation, node_coordinates, local_coord_system)
        self.determine_adjoint_load(q)
        self.apply_adjoint_loads(active_bc, active_loads)
        opdb = self.execute_adjoint_FEA()

        # Initiate dictionaries and extract data from odb file.
        strain, strain_mag = self.init_dictionaries(opdb)
        for stp in opdb.steps.values():

            frames = [stp.frames[-1]] if self.last_frame == 1 else stp.frames
```

```
            for frame in frames:
                strain, strain_mag = self.get_adjoint_strains(
                    opdb, frame, strain, strain_mag
                )

        # Remove adjoint loads, close odb file, and delete temporary files.
        self.remove_adjoint_loads(active_bc, active_loads)
        opdb.close()
        remove_files(iteration, 'Adjoint_Job')
        del self.mdb.jobs['Adjoint_Job'+str(iteration)]

        return strain

    def execute_adjoint_FEA(self):
        """ Execute the adjoint finite element analysis method

        Submits the adjoint ABAQUS job, waits for its completion, and then
        opens and returns its output database file (odb).

        Output:
        -------
        - opdb (Odb): ABAQUS output data base.
        """
        job_name = 'Adjoint_Job'+str(self.iteration)
        odb_name = job_name + '.odb'

        # Create an ABAQUS job, submit it, wait for completion, and open odb.
        mdb.Job(name = job_name, model = self.model_name,
                numDomains = self.nDomains, numCpus = self.nCPUs).submit()
        mdb.jobs[job_name].waitForCompletion()
        opdb = openOdb(odb_name)

        return opdb

    def init_dictionaries(self, opdb):
        """ Initialize dictionaries method

        Creates the dictionaries, and necessary entries, used to store the
        strain and strain magnitude extracted from the ABAQUS odb file.

        Input:
        ------
        - opdb (Odb): ABAQUS output data base.

        Output:
        -------
        - dictionaries (tuple): dictionaries created to store the strain and
          strain magnitude.
        """
        strain, strain_mag = {}, {}

        instance_name = self.part_name.upper()+'-1'
        elmts = opdb.rootAssembly.instances[instance_name].elements

        for elmt in elmts:
            strain[elmt.label] = {}
            strain_mag[elmt.label] = {}

        dictionaries = (strain, strain_mag)

        return dictionaries

    def get_adjoint_strains(self, opdb, frame, strain, strain_mag):
        """ Get adjoint strains method
        Method used to extract the maximum strain in each integration point.
        The function compares the value observed in the current frame with
        previous records and, if necessary, updates the record.

        The strains are stored in the ABAQUS variables 'E' or 'LE' depending
        on the step being geometrically linear or non-linear, respectively.

        Inputs:
        -------
```

```
        - opdb (Odb): ABAQUS output data base.
        - frame (OdbFrame): current frame of the ABAQUS odb.
        - strain (dict): dictionary of dictionaries, storing the maximum strain
          of each integration point (second key) in each element (first key).
        - strain_mag (dict): dictionary of dictionaries, storing the magnitude
          of the maximum strain of each integration point (second key) in each
          element (first key).

        Output:
        -------
        - strain (dict): dictionary of dictionaries, storing the maximum strain
          of each integration point (second key) in each element (first key).
        - strain_mag (dict): dictionary of dictionaries, storing the magnitude
          of the maximum strain of each integration point (second key) in each
          element (first key).
        """

        # Indicate that the data should be extracted from the integration
        # points.
        instance_name = self.part_name.upper()+'-1'
        instance = opdb.rootAssembly.instances[instance_name]
        region = instance.elementSets['STRAIN_ELEMENTS']
        position = INTEGRATION_POINT

        # The strains are stored in the ABAQUS variables 'E' or 'LE' depending
        # on the step being geometrically linear or non-linear, respectively.
        if 'E' in frame.fieldOutputs:
            temp_strain = frame.fieldOutputs['E'].getSubset(region = region,
                                                            position = position)
        elif 'LE' in frame.fieldOutputs:
            temp_strain = frame.fieldOutputs['LE'].getSubset(region = region,
                                                             position = position)
        else:
            raise Exception(
                "None of the strain variables 'E' or 'LE' were detected by \n"
                "the FEA function when performing a stress dependent \n"
                "optimization.\n"
            )

        # Extract the relevant strain data.
        attributes = 'data','elementLabel','maxPrincipal','integrationPoint'
        strains = map(attrgetter(*attributes), temp_strain.values)
        for item in strains:
            item_data = item[0]
            item_label = item[1]
            item_maxPrincipal = item[2]
            item_intPoint = item[3]

            # Cond_1 == True indicates that no previous value has been stored.
            cond_1 = (item_intPoint
                      not in strain_mag[item_label].keys())

            # Cond_2 == True indicates that the current value is larger than
            # the previous record.
            if cond_1 == False:
                prev_val = abs(strain_mag[item_label][item_intPoint])
                cond_2 = abs(item_maxPrincipal) >= prev_val
            else:
                cond_2 = False

            # If its the first dictionary entry, or there is a larger value,
            # update the dictionary entry.
            if cond_1 or cond_2:
                strain_mag[item_label][item_intPoint] = item_maxPrincipal
                if self.element_type in ['C3D8']:
                    strain_vector = item_data
                elif self.element_type in ['CPS4', 'CPE4', 'S4']:
                    strain_vector= np.array(
                        [item_data[0], item_data[1], item_data[3]]
                    )
                else:
                    raise Exception(
                        "Unexpected strain vector at the integration points."
                    )
```

```
                strain[item_label][item_intPoint] = strain_vector

        return strain, strain_mag

    def determine_adjoint_load(self, q):
        """ Determine adoint load method

        Determines the value of the adjoint load that should be applied on
        each node of the ABAQUS model.
        The load values are then rounded up at 12 orders of magnitude below
        the maximum adjoint load observed. This is done to prevent the
        application of loads that result from float point
        approximations/errors, and improve the efficiency of the code.

        The output is stored in the class attribute 'global_node_force'.

        Input:
        - q (float): P-norm factor used in the stress approximation function.
          Here referred as 'q' to avoid confusion with the SIMP penalty factor.
        """
        if self.element_type in ['CPS4', 'CPE4']:
            inc = 2
        elif self.element_type in ['C3D8']:
            inc = 3
        elif self.element_type in ['S4']:
            raise Exception(
                "The code provided does not allow the stress dependent \n"
                "topology optimization with shell elements, yet.\n"
                "To do so, at least the following tasks need to be done: \n"
                " - Determine the adjoint load and convert it back to the \n"
                "   global coordinate system.\n"
                " - Apply the adjoint loads depending on the dimension of \n"
                "   problem (2D or 3D)."
            )
        else:
            raise Exception(
                "Unexpected element type found in the \n"
                "'determine_adjoint_load' method."
            )

        # Array with the Von-Mises stress vector at each integration point of
        # each element.
        vm_int_p = np.array([self.multiply_VM_matrix(int_p, int_p) ** 0.5
                             for elmt in self.stress_vector_int.values()
                             for int_p in elmt.values()])

        # Determine the first term of the P-norm derivative w.r.t. Von-Mises
        # stress vector. Only depends on the sum of stress values.
        d_pnorm_vm_1 = sum(self.inv_int_p * vm_int_p ** q) ** ((1 / q) - 1)

        # Determine the second term of the P-norm derivative w.r.t. Von-Mises
        # Stress vector, and the derivative of the Von-Mises stress w.r.t.
        # the amplified stress vector.
        for elmt in self.all_elmts:
            force_elmt = 0
            c_matrix = self.c_matrix[elmt.label]

            for i in self.elmt_points:
                force = 0
                sv = self.stress_vector[elmt.connectivity[i]+1][elmt.label]
                b_matrix = self.b_matrix[elmt.connectivity[i]+1][elmt.label]
                jacobian = self.jacobian[elmt.connectivity[i]+1][elmt.label]

                von_mises_squared = self.multiply_VM_matrix(sv, sv)
                if float(von_mises_squared) != 0:
                    db_matrix = np.dot(c_matrix, b_matrix)

                    d_pnorm_vm_2 = (
                        (von_mises_squared ** ((q - 1) / 2)) * self.inv_int_p
                    )
                    d_pnorm_vm = d_pnorm_vm_1 * d_pnorm_vm_2

                    d_vm_sigmaA = (von_mises_squared ** -0.5) \
```

```
                                * self.multiply_VM_matrix(sv, db_matrix) \
                                * self.shell_thickness*np.linalg.det(jacobian)

                    # Determines the nodal force and adds its contribution.
                    force = d_pnorm_vm * d_vm_sigmaA
                    force_elmt += force

            # Sorts the nodal force contributions.
            for i in self.elmt_points:
                if hasattr(force_elmt,'shape'):
                    self.global_node_force[elmt.connectivity[i]+1] \
                        += force_elmt[0][i * inc : i * inc + inc]

        # Round the node forces at 12 orders of magnitude below the maximum
        # force observed.
        # Determines the number of decimal places.
        max_load = max([abs(item)
                        for sublist in self.global_node_force.values()
                        if hasattr(sublist,'shape')
                        for item in sublist])

        dp = int( -(math.floor(math.log10(max_load)) - 12))
        node_range = range(0,len(self.nodes))

        # Rounds the vectors depending on the problem being 2D or 3D.
        if self.planar == 1:
            for i in node_range:
                if hasattr(self.global_node_force[self.nodes[i].label],
                           'shape'):

                    self.global_node_force[self.nodes[i].label][0] = np.around(
                        self.global_node_force[self.nodes[i].label][0], dp)

                    self.global_node_force[self.nodes[i].label][1] = np.around(
                        self.global_node_force[self.nodes[i].label][1], dp)

        elif self.planar == 0:
            for i in node_range:
                if hasattr(self.global_node_force[self.nodes[i].label],
                           'shape'):

                    self.global_node_force[self.nodes[i].label][0] = np.around(
                        self.global_node_force[self.nodes[i].label][0], dp)

                    self.global_node_force[self.nodes[i].label][1] = np.around(
                        self.global_node_force[self.nodes[i].label][1], dp)

                    self.global_node_force[self.nodes[i].label][2] = np.around(
                        self.global_node_force[self.nodes[i].label][2], dp)

        else:
            raise Exception(
                "Unexpected value for 'planar' variable in \n"
                "'determine_adjoint_load' method of class AdjointModel."
            )

        return self.global_node_force

    def remove_adjoint_loads(self, active_bc, active_loads):
        """ Remove adjoint loads method

        Removes the adjoint loads applied during the simulation of the adjoint
        model.

        Inputs:
        -------
        - active_bc (dict): dictionary with the data of non-zero boundary
          conditions imposed in the model (such as non-zero displacements).
        - active_loads (list): list with the keys (names) of the loads that are
          active during the simulation (i.e.: non-supressed loads).
        """
        #Disable loads of the adjoint model
        for i in self.active_nodes:
            self.mdb.models[self.model_name].loads["adjoint_load-"+str(i)] \
```

```
                .suppress()

        # Resume non-adjoint active loads.
        for item in active_loads:
            self.mdb.models[self.model_name].loads[item].resume()

        # Resume imposed non-zero displacements.
        for key in active_bc.keys():
            for step in active_bc[key].keys():
                (
                    u1,
                    u2,
                    u3,
                    ur1,
                    ur2,
                    ur3,
                ) = self.resume_displacement_BC(active_bc[key][step])

                self.mdb.models[self.model_name].boundaryConditions[key] \
                    .setValuesInStep(step, u1 = u1, u2 = u2, u3 = u3,
                                     ur1 = ur1, ur2 = ur2, ur3 = ur3)

    def apply_adjoint_loads(self, active_bc, active_loads):
        """ Apply adjoint loads method

        This method edits the boundary conditions and loads applied in the
        ABAQUS model, performing the following tasks:
        - Applies the nodal loads determined by the 'determine_adjoint_load'
          method, if they are non-zero.
        - Supresses non-adjoint active loads.
        - Supresses non-zero displacements.

        The loads suppressed during are identified in list, which is stored
        in the class attribute "active_nodes". This list is then used to
        reverse the changes made by this method once the adjoint model
        concludes its analysis.

        Inputs:
        -------
        - active_bc (dict): dictionary with the data of non-zero boundary
          conditions imposed in the model (such as non-zero displacements).
        """
        self.active_nodes = []
        node_range = range(0,len(self.nodes))

        # Create a nodal force on the nodes and record their label in order to
        # disable the adjoint loads at the end of the process.
        for i in node_range:
            non_zero_force = self.non_zero_force_check(self.nodes[i].label)
            if non_zero_force == True:
                self.active_nodes.append(self.nodes[i].label)
                self.apply_nodal_load(self.nodes[i].label)

        # Suppress non-adjoint active loads.
        for item in active_loads:
            self.mdb.models[self.model_name].loads[item].suppress()

        # Supress imposed non-zero displacements.
        for key in active_bc.keys():
            for step in active_bc[key].keys():
                (
                    u1,
                    u2,
                    u3,
                    ur1,
                    ur2,
                    ur3,
                ) = self.suppress_displacement_BC(active_bc[key][step])

                self.mdb.models[self.model_name].boundaryConditions[key] \
                    .setValuesInStep(step, u1 = u1, u2 = u2, u3 = u3,
                                     ur1 = ur1, ur2 = ur2, ur3 = ur3)

    def non_zero_force_check(self, node_label):
```

```
        """ Non zero force check method

        Determines if the adjoint load assigned to a node is null or not.

        The method verifies if a load has been assigned to the node, and then
        checks if at least one coordinate of the load vector is different than
        zero.

        Input:
        ------
        - node_label (int): label of the node whose force is being evaluated.

        Output:
        -------
        - check (bool): boolean variable determining if the force applied to
          the node is not null (True) or not (False).
        """
        check = None

        # If the node was assigned a load:
        if hasattr(self.global_node_force[node_label], 'shape'):

            # Check if it has at least one non-zero component (2D vector).
            if self.planar == 1:
                cond_1 = (self.global_node_force[node_label][0] != 0.0)
                cond_2 = (self.global_node_force[node_label][1] != 0.0)

                check = (cond_1 or cond_2)

            # Check if it has at least one non-zero component (3D vector)
            elif self.planar == 0:
                cond_1 = (self.global_node_force[node_label][0] != 0.0)
                cond_2 = (self.global_node_force[node_label][1] != 0.0)
                cond_3 = (self.global_node_force[node_label][2] != 0.0)

                check = (cond_1 or cond_2 or cond_3)

            else:
                raise Exception(
                    "Unexpected value for 'planar' variable in "
                    " 'non_zero_force_check' method of class AdjointModel."
                )
        else:
            check = False

        return check

    def apply_nodal_load(self, node_label):
        """ Apply nodal load method

        Applies a nodal load, determined by the 'determine_adjoint_load'
        method, at a given load. The load is created at the first step of the
        ABAQUS model.

        Input:
        ------
        - node_label (int): label of the node whose force is being evaluated.
        """
        # Selects the node.
        node_region = self.mdb.models[self.model_name].rootAssembly \
                      .instances[self.part_name+'-1'] \
                      .sets["adjoint_node-"+str(node_label)]

        # Identifies the first step of the ABAQUS simulation.
        first_step = self.mdb.models[self.model_name].steps.keys()[1]

        # Applies the nodal load (2D vector).
        if self.planar == 1:
            self.mdb.models[self.model_name].ConcentratedForce(
                name = "adjoint_load-"+str(node_label),
                createStepName = first_step,
                region = node_region,
                cf1 = float(self.global_node_force[node_label][0]),
                cf2 = float(self.global_node_force[node_label][1]),
```

```
                distributionType = UNIFORM,
                field = '',
                localCsys = None
            )

        # Applies the nodal load (3D vector).
        elif self.planar == 0:
            self.mdb.models[self.model_name].ConcentratedForce(
                name = "adjoint_load-"+str(node_label),
                createStepName = first_step,
                region = node_region,
                cf1 = float(self.global_node_force[node_label][0]),
                cf2 = float(self.global_node_force[node_label][1]),
                cf3 = float(self.global_node_force[node_label][2]),
                distributionType = UNIFORM,
                field = '',
                localCsys = None
            )

        else:
            raise Exception(
                "Unexpected value for 'planar' variable in 'apply_nodal_load'\n"
                "method of class AdjointModel."
            )

    def suppress_displacement_BC(self, bc_list):
        """ Suppress displacement boundary conditions method

        This method checks if a boundary condition is active and if it applies
        a non-zero displacement. If both conditions are confiremd, the method
        will suppress the boundary condition.

        This method outputs 6 variables, indicating if any possible degree of
        freedom of the boundary condition (3 displacements and 3 rotations)
        was suppressed.

        Input:
        ------
        - bc_list (dict): dictionary with the value and state variables of an
          ABAQUS boundary condition.

        Output:
        -------
        - u1, u2, u3, ur1, ur2, ur3 (symbolicConstants.SymbolicConstant):
          ABAQUS variables defining if the degrees of freedom of the boundary
          condition were suppressed (FREED) or not (UNCHANGED).
        """
        value = bc_list['value']
        state = bc_list['state']
        output_var = []

        # Changes non-zero "SET" boundary conditions to "FREED".
        for i in range(0,6):
            if value[i] != 0 and state[i] == SET:
                output_var.append(FREED)
            else:
                output_var.append(UNCHANGED)

        u1, u2, u3, ur1, ur2, ur3 = output_var

        return u1, u2, u3, ur1, ur2, ur3

    def resume_displacement_BC(self, bc_list):
        """ Resume displacement boundary conditions method

        This method reverts the changes made by the 'suppress_displacement_BC'
        method, assigning the original 'state' and 'value' of the boundary
        condition

        Input:
        ------
        - bc_list (dict): dictionary with the value and state variables of an
          ABAQUS boundary condition.
```

```
        Output:
        -------
        - u1, u2, u3, ur1, ur2, ur3 (symbolicConstants.SymbolicConstant):
          ABAQUS variables defining, either, the original displacement imposed
          at each degree of freedom of the boundary condition, or that the
          value should be equal to the one defined in the previous step
          (UNCHANGED).
        """
        value = bc_list['value']
        state = bc_list['state']
        output_var = []

        # Changes non-zero "SET" boundary conditions to their original value.
        for i in range(0,6):
            if value[i] != 0 and state[i] == SET:
                output_var.append(value[i])
            else:
                output_var.append(UNCHANGED)

        u1, u2, u3, ur1, ur2, ur3 = output_var

        return u1, u2, u3, ur1, ur2, ur3

    def determine_stress_and_deformation(
            self, node_displacement, xe, node_rotation, node_coordinates,
            local_coord_system
        ):
        """ Determine stress and deformation method

        Determines the stress and deformation vectors at the nodes and
        integration points of each element.

        The output is stored in the class attributes: deformation_vector,
        deformation_int, stress_vector, and stress_vector_int.

        During the process, the strain-displacement matrix (B matrix) is also
        determined at each node and integration point. This information is also
        stored in the class attributes: b_matrix, and b_matrix_int.

        Inputs:
        -------
        - node_displacement (dict): dictionary with the displacement of each
          node.
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.
        - node_rotation (dict): dictionary with the node rotations of nodes in
          each element.
        - node_coordinates (dict): dictionary with the coordinates of each
          node.
        - local_coord_system (dict): dictionary with the local coordinate
          systems of each element.
        """

        # Determins the local coordinates of the element nodes and integration
        # points.
        elmt_formulation = ElementFormulation(self.element_type)
        s, t, v = elmt_formulation.local_node_coordinates()
        s_int, t_int, v_int = elmt_formulation.local_int_point_coordinates()

        # If the element type is not S4, sets unused variables to None.
        if self.element_type != 'S4':
            a_rot, b_rot = None, None
            v1_vector, v2_vector, vn_vector = None, None, None
            vect_transf_m, mat_transf_m = None, None

        # For each element:
        for elmt in self.all_elmts:

            # Determins the node global coordinates.
            xyz_coord = self.coordinate_vectors(elmt, node_coordinates)
            x_coord, y_coord, z_coord = xyz_coord

            # Creates the node displacement vector.
            node_disp_vector = self.elmt_node_displacement_vect(
```

```
            elmt, node_displacement, node_rotation
        )

        # For S4 elements:
        # Determines the node rotations, normal vectors, transformation
        # matrixes, selects the most stressed surface, and sets the
        # C matrix to the local coordinate system.
        if self.element_type == 'S4':
            a_rot, b_rot = self.rotation_vectors(elmt, node_rotation)

            v1_vector, v2_vector, vn_vector = \
                self.node_normal_vectors(elmt)

            vect_transf_m = self.vect_transf_matrix(
                elmt, local_coord_system
            )

            mat_transf_m = self.matx_transf_matrix(
                v1_vector[0], v2_vector[0], vn_vector[0]
            )

            v = self.surface_selection(
                elmt,
                node_disp_vector,
                s, t, v,
                x_coord, y_coord, z_coord,
                v1_vector, v2_vector, vn_vector,
                a_rot, b_rot,
                elmt_formulation
            )

            self.c_matrix[elmt.label] = self.local_c_matrix(
                mat_transf_m, elmt
            )

        # For each node:
        for i in self.elmt_points:

            # Determines the B and Jacobian matrixes in the node and
            # integration point.
            b_matrix, jacobian = elmt_formulation.b_matrix_and_jac(
                s[i], t[i], v[i],
                x_coord, y_coord, z_coord,
                v1_vector, v2_vector, vn_vector,
                a_rot, b_rot,
                self.shell_thickness
            )
            b_matrix_int, jacobian_int = elmt_formulation.b_matrix_and_jac(
                s_int[i], t_int[i], v_int[i],
                x_coord, y_coord, z_coord,
                v1_vector, v2_vector, vn_vector,
                a_rot, b_rot,
                self.shell_thickness
            )

            # Determines the stress and strain vectors in the node and
            # integration point.
            deformation = np.dot(b_matrix, node_disp_vector)
            deformation_int = np.dot(b_matrix_int, node_disp_vector)

            stress_vect = self.determine_stress_vector(
                elmt, vect_transf_m, mat_transf_m, deformation, xe
            )
            stress_vect_int = self.determine_stress_vector(
                elmt, vect_transf_m, mat_transf_m, deformation_int, xe
            )

            # Sorts the data into the class attributes.
            self.deformation_vector[elmt.connectivity[i]+1][elmt.label] = \
                deformation
            self.deformation_int[elmt.label][i+1] = deformation_int

            self.stress_vector[elmt.connectivity[i]+1][elmt.label] = \
                stress_vect
```

```
                self.stress_vector_int[elmt.label][i+1] = stress_vect_int

                self.jacobian[elmt.connectivity[i]+1][elmt.label] = jacobian
                self.jacobian_int[elmt.label][i+1] = jacobian_int

                self.b_matrix[elmt.connectivity[i]+1][elmt.label] = b_matrix
                self.b_matrix_int[elmt.label][i+1] = b_matrix_int

    def determine_stress_vector(
            self, elmt, vector_trans_m, matrix_trans_m, deformation, xe
        ):
        """ Determine stress vector method

        Determines the stress vector based on the deformation observed in a
        given point (node or integration point) and on the stiffness of the
        element.

        If the point belongs to a shell element, the stress vector is converted
        to the default coordinate system assigned by ABAQUS.

        Inputs:
        -------
        - elmt (MeshElementArray): shell element where the stress will be
          determined.
        - vector_trans_m, matrix_trans_m (numpy.array): transformation
          matrixes.
        - deformation (array): vector with the deformations observed at the
          node or integration_point of the element.
        - xe (dict): dictionary with the design densities of all elements in
          the topology optimization process.

        Output:
        -------
        - stress_vector (array): stress vector in the default coordinate system
          assigned by ABAQUS.
        """

        # Fetches the C matrix and determins the amplified stress vector.
        elmt_c_matrix = self.c_matrix[elmt.label]
        sqrt_rho = math.sqrt(self.xe_all(elmt.label, xe))
        sv = np.dot(elmt_c_matrix, deformation) * sqrt_rho

        # For S4 elements, rotates the vector to the local coordinate system.
        if self.element_type == 'S4':
            s_matrix = np.array([[sv[0][0],sv[3][0],sv[4][0]],
                                 [sv[3][0],sv[1][0],sv[5][0]],
                                 [sv[4][0],sv[5][0],sv[2][0]]])

            s_matrix = np.dot(vector_trans_m.T,
                              np.dot(s_matrix, vector_trans_m))

            stress_vector = np.array([[s_matrix[0][0]],
                                      [s_matrix[1][1]],
                                      [s_matrix[2][2]],
                                      [s_matrix[0][1]],
                                      [s_matrix[0][2]],
                                      [s_matrix[1][2]]])
        else:
            stress_vector = sv

        return stress_vector

    def elmt_node_displacement_vect(
            self, elmt, node_displacement, node_rotation = None
        ):
        """ Elemental node displacement vector method

        Creates a vertical array with the displacement of the nodes in a given
        element. The nodes are organized according to ABAQUS labelling
        sequence. If the element is a shell element (S4), the code will append
        the node rotations, as they also constitute 2 possible degrees of
        freedom for the shell nodes. In this case, the rotation along the third
        axis is discarded, as it was set to zero during the transformation to
        the local coordinate system.
```

```
        Inputs:
        -------
        - elmt (MeshElementArray): element of the nodes to be organized.
        - node_displacement (dict): dictionary with the displacement of each
          node.
        - node_rotation (dict): dictionary with the node rotations of nodes in
          each element.

        Outputs:
        --------
        - node_disp_vector (array): vertical vector with the node
          displacements, and node rotation if it is a shell element.
        """

        node_disp_vector = None

        # Selects the relevant node displacement coordinates (for 2D or 3D
        # problems) and node rotations (for shell elements).
        # Then, stacks them into a single vector.
        for node in elmt.connectivity:
            displacements = node_displacement[node+1]

            if self.element_type in ["CPS4", "CPE4"]:
                disp_vector = np.array([[item] for item in displacements[0:2]])
            else:
                disp_vector = np.array([[item] for item in displacements])

            if hasattr(node_disp_vector, "shape"):
                node_disp_vector = np.vstack((node_disp_vector, disp_vector))
            else:
                node_disp_vector = disp_vector

            if self.element_type == 'S4':
                rotations = node_rotation[elmt.label][node+1]
                rot_vector = np.array([[item] for item in rotations[0:2]])
                node_disp_vector = np.vstack((node_disp_vector, rot_vector))

        return node_disp_vector

    def coordinate_vectors(self, elmt, node_coords):
        """ Coordinate vectors method
        Organizes the node coordinates in three lists, following the node
        labelling sequence set by ABAQUS.

        If a third dimension does not exist, the 'z_coord' dictionary is
        returned empty.

        Inputs:
        -------
        - elmt (MeshElementArray): element of the nodes to be organized.
        - node_coords (dict): dictionary with the node coordinates of each
          element.
        - number_nodes (int): number of nodes in the element.

        Outputs:
        --------
        - x_coord, y_coord, z_coord (lists): lists with the node coordinates,
          following the node labelling sequence set by ABAQUS.
        """
        x_coord, y_coord, z_coord = [], [], []

        elmt_points = self.elmt_points

        x_coord = [node_coords[elmt.connectivity[i]+1][0] for i in elmt_points]
        y_coord = [node_coords[elmt.connectivity[i]+1][1] for i in elmt_points]

        if self.element_type in ['S4', 'C3D8']:
            z_coord = [node_coords[elmt.connectivity[i]+1][2]
                       for i in elmt_points]

        return x_coord, y_coord, z_coord

    def rotation_vectors(self, elmt, node_rotation):
```

```
        """ Rotation vectors method
        Organizes the node rotations in two lists, following the node
        labelling sequence set by ABAQUS.

        This method is similar to the 'elmt_node_displacement_vect' method,
        differing on the fact that it returns the node rotations separately.

        Inputs:
        -------
        - elmt (MeshElementArray): element of the nodes to be organized.
        - node_rotation (dict): dictionary with the node rotations of nodes in
          each element.
        - number_nodes (int): number of nodes in the element.

        Outputs:
        --------
        - a_rot, b_rot (lists): lists with the node rotations of a given
          element, following the node labelling sequence set by ABAQUS.
        """
        elmt_points = self.elmt_points
        rotations = node_rotation[elmt.label]

        a_rot = [rotations[elmt.connectivity[i]+1][0] for i in elmt_points]
        b_rot = [rotations[elmt.connectivity[i]+1][1] for i in elmt_points]

        return a_rot, b_rot

    def node_normal_vectors(self, elmt):
        """ Node normal vectors method

        Returns the node normal vectors as three different list variables.

        Inputs:
        -------
        - elmt (MeshElementArray): element of the nodes to be organized.

        Outputs:
        --------
        - v1_vector, v2_vector, vn_vector (lists): lists with the node normal
          directions.
        """
        v1_vector = []
        v2_vector = []
        vn_vector = []
        normal_vector = self.node_normal_vector[elmt.label]

        for node in elmt.connectivity:
                v1_vector.append(normal_vector[node + 1]["v1"])
                v2_vector.append(normal_vector[node + 1]["v2"])
                vn_vector.append(normal_vector[node + 1]["vn"])

        return v1_vector, v2_vector, vn_vector

    def vect_transf_matrix(self, elmt, local_coord_system):
        """ Vector transformation matrix method

        Returns a transformation matrix suitable to convert a vector from the
        global to the element local coordinate system.

        Inputs:
        -------
        - elmt (MeshElementArray): element to be considered.
        - local_coord_system (dict): dictionary with the local coordinate
          systems of each element.

        Output:
        -------
        - transformation_matrix (numpy.array): array with the transformation
          matrix.
        """
        local_coord_vectors = local_coord_system[elmt.label]

        transformation_matrix = np.array([local_coord_vectors[0],
                                          local_coord_vectors[1],
```

```
                                        local_coord_vectors[2]])

        transformation_matrix = np.linalg.inv(transformation_matrix)

        return transformation_matrix

    def matx_transf_matrix(self, v1, v2, vn):
        """ Matrix transformation matrix method

        Returns a transformation matrix suitable to convert a matrix from the
        global to the element local coordinate system.

        Inputs:
        -------
        - v1, v2, vn (numpy.array): vectors defining the element local
          coordinate system.

        Output:
        -------
        - transformation_matrix (numpy.array): array with the transformation
          matrix.
        """
        ex = np.array([1.0, 0, 0])
        ey = np.array([0, 1.0, 0])
        ez = np.array([0, 0, 1.0])
        l1 = np.dot(ex, v1) / (np.linalg.norm(ex) * np.linalg.norm(v1))
        l2 = np.dot(ex, v2) / (np.linalg.norm(ex) * np.linalg.norm(v1))
        l3 = np.dot(ex, vn) / (np.linalg.norm(ex) * np.linalg.norm(v1))
        m1 = np.dot(ey, v1) / (np.linalg.norm(ey) * np.linalg.norm(v2))
        m2 = np.dot(ey, v2) / (np.linalg.norm(ey) * np.linalg.norm(v2))
        m3 = np.dot(ey, vn) / (np.linalg.norm(ey) * np.linalg.norm(v2))
        n1 = np.dot(ez, v1) / (np.linalg.norm(ez) * np.linalg.norm(vn))
        n2 = np.dot(ez, v2) / (np.linalg.norm(ez) * np.linalg.norm(vn))
        n3 = np.dot(ez, vn) / (np.linalg.norm(ez) * np.linalg.norm(vn))

        line_1 = [l1 ** 2, m1 ** 2, n1 ** 2, l1 * m1, n1 * l1, m1 * n1]
        line_2 = [l2 ** 2, m2 ** 2, n2 ** 2, l2 * m2, n2 * l2, m2 * n2]
        line_3 = [l3 ** 2, m3 ** 2, n3 ** 2, l3 * m3, n3 * l3, m3 * n3]
        line_4 = [2 * l1 * l2, 2 * m1 * m2, 2 * n1 * n2, l1 * m2 + l2 * m1,
                  n1 * l2 + n2 * l1, m1 * n2 + m2 * n1]
        line_5 = [2 * l3 * l1, 2 * m3 * m1, 2 * n3 * n1, l3 * m1 + l1 * m3,
                  n3 * l1 + n1 * l3, m3 * n1 + m1 * n3]
        line_6 = [2 * l2 * l3, 2 * m2 * m3, 2 * n2 * n3, l2 * m3 + l3 * m2,
                  n2 * l3 + n3 * l2, m2 * n3 + m3 * n2]

        transformation_matrix = np.array([line_1,
                                          line_2,
                                          line_3,
                                          line_4,
                                          line_5,
                                          line_6])

        return transformation_matrix

    def surface_selection(
            self, elmt, node_displacement_vector, s, t, v, x_coord, y_coord,
            z_coord, v1_vector, v2_vector, vn_vector, a_rot, b_rot,
            elmt_formulation
        ):

        """ Surface selection method

        Determines if the largest deformation, and consequently largest stress,
        occurs in the upper or lower surface of a shell element.

        The process requires the determination of the b_matrix, and finally the
        deformation on both sides of the shell element. Based on the largest
        deformation, the method outputs the local coordinate value of the upper
        or lower surface (1.0 or -1.0, respectively).

        Inputs:
        -------
        - elmt (MeshElementArray): shell element to be evaluated.
        - node_displacement_vector (array): vertical vector with the node
```

```
        displacements, and node rotation if it is a shell element.
      - s, t, v (dicts): dictionaries with the local coordinates of each
        node.
      - x_coord, y_coord, z_coord (lists): lists with the node coordinates,
        following the node labelling sequence set by ABAQUS.
      - v1_vector, v2_vector, vn_vector ():  Vectors indicating the in-plane
        directions of the node local coordinate system (as illustrated in the
        book Finite Element Procedures, 2nd edition, written by Klaus-Jurgen
        Bathe, in section 5.4, page 437, figure 5.33).
      - a_rot, b_rot (lists): lists with the node rotations of a given
        element, following the node labelling sequence set by ABAQUS.
      - elmt_formulation (ElementFormulation class): class with information
        regarding the formulation of the element being used in the ABAQUS
        model.

      Output:
      -------
      - v (float): local coordinate value of the upper or lower shell surface
        (1.0 or -1.0, respectively).
      """
      upper_def, lower_def = 0, 0

      # Determines the deformation on the upper surface.
      for key in v.keys():
          v[key] = 1.0

      for i in self.elmt_points:

          b_matrix, _ = elmt_formulation.b_matrix_and_jac(
              s[i], t[i], v[i], x_coord, y_coord, z_coord, v1_vector,
              v2_vector, vn_vector, a_rot, b_rot, self.shell_thickness
          )
          deformation = np.dot(b_matrix, node_displacement_vector)
          upper_def += np.linalg.norm(deformation)

      # Determines the deformation on the lower surface.
      for key in v.keys():
          v[key] = -1.0

      for i in self.elmt_points:

          b_matrix, _ = elmt_formulation.b_matrix_and_jac(
              s[i], t[i], v[i], x_coord, y_coord, z_coord, v1_vector,
              v2_vector, vn_vector, a_rot, b_rot, self.shell_thickness
          )
          deformation = np.dot(b_matrix, node_displacement_vector)
          lower_def += np.linalg.norm(deformation)

      # Selects the surface with the largest deformation.
      if upper_def >= lower_def:
          for key in v.keys():
              v[key] = 1.0
      else:
          for key in v.keys():
              v[key] = -1.0

      return v

  def xe_all(self, label, xe):
      """ Xe all method

      Returns the design density of a given element. If the element is not
      part of the editable_region, returns 1.0.

      Inputs:
      -------
      - label (int): label of the element to be evaluated.
      - xe (dict): dictionary with the design densities of all elements.

      Output:
      -------
      - rho (float): design density of the element. Set to 1.0 if the element
        does not belong to the editable region.
      """
```

```
        if label in xe.keys():
            return xe[label]
        else:
            return 1.0

    def multiply_VM_matrix(self, v1, v2):
        """ Multiply by Von-Mises matrix method

        This method multiplies two vectors by the Von-Mises matrix, used to
        determine the Von-Mises stress vector.

        Note: If v1 is equal to v2, the output of this function is equal to
        the square of the Von-Mises stress.

        Inputs:
        -------
        - v1, v2 (array): vectors to be multiplied by the Von-Mises matrix, on
          the left-hand and right-hand side of the matrix, respectively.

        Output:
        -------
        - vm_vector (array): product of the multiplication by the Von-Mises
          matrix.
        """

        dim = int(max(v1.shape))

        # Selects the Von-Mises matrix based on the vector size.
        if dim == 3:
            matrix = np.array([[1,-0.5,0],
                               [-0.5, 1,0],
                               [0, 0,3]])
        elif dim == 6:
            matrix = np.array([[1,-0.5,-0.5,0,0,0],
                               [-0.5, 1,-0.5,0,0,0],
                               [-0.5, -0.5, 1,0,0,0],
                               [0, 0, 0,3,0,0],
                               [0, 0, 0,0,3,0],
                               [0, 0, 0,0,0,3]])
        else:
            raise Exception(
                "Unexpected dimension for the stress vector in the \n"
                "'multiply_von_mises_matrix' method."
            )

        # Returns the product.
        return np.dot(v1.T, np.dot(matrix, v2))

    def local_c_matrix(self, matrix_trans_m, elmt):
        """ Local C matrix method

        Converts the element stiffness matrix to the element local coordinate
        system.

        Inputs:
        -------
        - matrix_trans_m (array): transformation matrix.
        - elmt (MeshElementArray): element where the transformation should
          be performed.
        """
        label = elmt.label
        local_c_matrix = np.dot(matrix_trans_m.T,
                                np.dot(self.c_matrix[label], matrix_trans_m)
        )

        return local_c_matrix

    def stress_sensitivity(self, xe, q, state_strain, adjoint_strain):
        """ Stress sensitivity method

        Determines the sensitivity of the P-norm maximum stress approximation
        to changes in the design density of each element, in accordance with
        the research article [1]. A brief and a more detailed explanation
        can be found below.
```

```
        This method determines and outputs the stress sensitivity. The two
        main intermediate terms ('d_pnorm_spf' and 'd_pnorm_displacement')
        are determined by the 'determine_d_pnorm_spf' and
        'determine_d_pnorm_displacement' methods, and stored as an attribute
        of this class for their mathematical relevance.

        Two attributes are generated by this method to store the element stress
        sensitivity, one for the discrete value and another for the continuous
        value. The continuous value is independent of the mesh used in the FEA,
        making it better as a reference for validations or comparisons with
        numerical/analitical derivatives. However, the discrete value is
        dependent on the mesh used, making it more suitable to be passed to the
        optimizers since, in the general case, the optimizers should not have
        information on the mesh or element size.

        Inputs:
        -------
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.
        - q (float): value of the exponent used in the p-norm approximation.
        - state_strain (dict): dictionary with the strains at the integration
          points of the elements that belong to the state model (original
          model).
        - adjoint_strain (dict): dictionary with the strains at the integration
          points of the elements that belong to the adjoint model.

        Outputs:
        --------
        - elmt_stress_sensitivity_discrete (dict): dictionary with the
          sensitivity of the P-norm maximum stress approximation to changes in
          the design densities.

        BRIEF MATHEMATICAL EXPLANATION:
        -------------------------------
        This is done through an analytical derivative, which can be reduced
        to the sum of two terms:

            d_pnorm_rho = d_pnorm_spf + d_pnorm_displacement

        Where:

        - 'd_pnorm_rho' is the derivative of the P-norm maximum stress
          approximation with respect to (w.r.t.) the design density.
        - 'd_pnorm_spf' is a term of the derivative that depends on the
          derivation of the stress penalization factor w.r.t. the design
          variables.
        - 'd_pnorm_displacement' is a term of the derivative that depends on
          the derivation of the displacement w.r.t. the design variables.
          Obtained from the adjoint model.

        DETAILED MATHEMATICAL EXPLANATION:
        ----------------------------------
        This is done through an analytical derivative, obtained by the chain
        rule, which considers three major terms:

            d_pnorm_rho = d_pnorm_vm * d_vm_sigmaA * d_sigmaA_rho

        Where:

        - 'd_pnorm_rho' is the derivative of the P-norm maximum stress
          approximation with respect to (w.r.t.) the design density.
        - 'd_pnorm_vm' is the derivative of the P-norm maximum stress
          approximation w.r.t. the Von-Mises stress.
        - 'd_vm_sigmaA' is the derivative of the Von-Mises stress w.r.t. the
          amplified stress state (stress multiplied by the stress penalization
          factor).
        - 'd_sigmaA_rho' is the derivative of the amplified stress state w.r.t.
          the design densities.

        The reader is reminded that 'sigmaA' is defined as:
            sigmaA = stress_amp_factor * C_matrix * b_matrix * displacement

        where the 'stress_amp_factor' is equal to the square root of the design
```

```
        density, as proposed in [1]. Therefore, since both the
        'stress_amp_factor' and the 'displacement' are functions of the design
        density, 'd_sigmaA_rho' has two terms, here defined as follows:

            d_sigmaA_rho = d_sigmaA_spf + d_sigmaA_displacement

        Where:

            d_sigmaA_spf = d_stress_amp_factor_rho * C_matrix * b_matrix \
                         * displacement
            d_sigmaA_displacement = stress_amp_factor * C_matrix * b_matrix \
                                  * d_displacement_rho

        (Note that the character '\' refers to the code line-break command,
        not the division operator '/').

        'd_sigmaA_spf' is easely determined analiticaly (derivative of a
        square root), while 'd_sigmaA_displacement' is be determined
        through an adjoint model.

        Due to the need of using the adjoint model, and to improve the
        computational efficiency, the Stress sensitivity method will determined
        'd_pnorm_rho' as the sum of two terms:

            d_pnorm_rho = d_pnorm_spf + d_pnorm_displacement

        Where:

            d_pnorm_spf = d_pnorm_vm * d_vm_sigmaA * d_sigmaA_spf

            d_pnorm_displacement = d_pnorm_vm * d_vm_sigmaA \
                                 * d_sigmaA_displacement

        Since the adjoint model already considered the term
        'd_pnorm_vm * d_vm_sigmaA' in 'd_pnorm_displacement', it can be finally
        rewritten as:
            d_pnorm_displacement = stress_amp_factor * adj_deformation \
                                 * d_stiffness_rho * deformation

        Where the 'adj_deformation' is deformation from the adjoint model,
        whose loads already considered the influence of
        'd_pnorm_vm * d_vm_sigmaA' for the sake of computational efficiency.
        Note that the product 'b_matrix * displacement' is replaced by the
        'deformation' of the regular model.


        REFERENCES:
        -----------
        [1] - Holmberg, Erik, Bo Torstenfelt, and Anders Klarbring.
        "Stress constrained topology optimization." Structural and
        Multidisciplinary Optimization 48.1 (2013): 33-47.
        """
        self.elmt_stress_sensitivity_continuous = {}
        self.elmt_stress_sensitivity_discrete = {}

        # Determines the two components of the derivative.
        self.determine_d_pnorm_spf(xe, q)
        self.determine_d_pnorm_displacement(xe, state_strain, adjoint_strain)

        # Determine the element stress sensitivity in continuous form.
        for elmt in self.all_elmts:
            self.elmt_stress_sensitivity_continuous[elmt.label] = 0.0

            # Note that d_pnorm_displacement should be a negative term,
            # resulting from the derivation process.
            self.elmt_stress_sensitivity_continuous[elmt.label] += (
                self.d_pnorm_spf[elmt.label] \
                + self.d_pnorm_displacement[elmt.label]
            )

        # Determine the element stress sensitivity in discrete form.
        for elmt in self.all_elmts:
            int_p = 1
            label = elmt.label
```

```
            det_jac = np.linalg.det(self.jacobian_int[label][int_p])
            self.elmt_stress_sensitivity_discrete[elmt.label] = \
                self.elmt_stress_sensitivity_continuous[elmt.label] / det_jac

        return self.elmt_stress_sensitivity_discrete

    def determine_d_pnorm_displacement(self, xe, state_strain, adjoint_strain):
        """ Determine d_pnorm_displacement method

        Determines the component of P-norm stress derivative w.r.t. the design
        densities which contains the element nodal displacement.

        The output is stored as a class attribute, for its mathematical
        relevance.

        Inputs:
        -------
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.
        - q (float): value of the exponent used in the p-norm approximation.
        - state_strain (dict): dictionary with the strains at the integration
          points of the elements that belong to the state model (original
          model).
        - adjoint_strain (dict): dictionary with the strains at the integration
          points of the elements that belong to the adjoint model.
        """
        self.d_pnorm_displacement = {}
        p = self.p
        for elmt in self.all_elmts:

            self.d_pnorm_displacement[elmt.label] = 0.0
            elmt_vol = self.elmt_volume[elmt.label]
            c_matrix = self.c_matrix[elmt.label]

            for i in self.elmt_points:

                jacobian_int = self.jacobian_int[elmt.label][i+1]
                det_jac = np.linalg.det(jacobian_int)
                rho = self.xe_all(elmt.label, xe)

                d_cMatrix_rho = p * c_matrix * rho ** (p - 1)
                state_strain_int_p = state_strain[elmt.label][i + 1]
                adj_strain_int_p = adjoint_strain[elmt.label][i + 1]

                # Note that the negative sign comes from the derivation
                # process (not explicit in the code).
                dMatrix_ss = np.dot(d_cMatrix_rho, state_strain_int_p)
                strain_products = -np.dot(adj_strain_int_p, dMatrix_ss)

                self.d_pnorm_displacement[elmt.label] += (
                    strain_products * det_jac * self.shell_thickness / elmt_vol
                )

    def determine_d_pnorm_spf(self, xe, q):
        """ Determine d_pnorm_spf method

        Determines the component of P-norm stress derivative w.r.t. the design
        densities which contains the stress penalization factor.

        The output is stored as a class attribute, for its mathematical
        relevance.

        Inputs:
        -------
        - xe (dict): dictionary with the design variables of all elements in
          the topology optimization process.
        - q (float): value of the exponent used in the p-norm approximation.
        """
        self.d_pnorm_spf = {}

        vm_int_p = np.array([self.multiply_VM_matrix(int_p, int_p) ** 0.5
                             for elmt in self.stress_vector_int.values()
                             for int_p in elmt.values()])
```

```
        d_pnorm_vm_1 = sum(self.inv_int_p * vm_int_p ** q) ** ((1 / q) - 1)

        for elmt in self.all_elmts:
            self.d_pnorm_spf[elmt.label] = 0.0
            elmt_vol = self.elmt_volume[elmt.label]
            c_matrix = self.c_matrix[elmt.label]
            for i in self.elmt_points:

                sv = self.stress_vector_int[elmt.label][i+1]
                jacobian_int = self.jacobian_int[elmt.label][i+1]
                deformation_int = self.deformation_int[elmt.label][i+1]

                det_jac = np.linalg.det(jacobian_int)
                von_mises_squared = self.multiply_VM_matrix(sv, sv)

                if float(von_mises_squared) != 0:
                    # 'stress_vector_int' and 'real_stress_int' may differ due
                    # to the stress penalization factor, which is included in
                    # the former but not in the latter. (They are qual for
                    # rho=0 or 1).
                    real_stress_int = np.dot(c_matrix, deformation_int)

                    d_pnorm_vm_2 = (
                        (von_mises_squared ** ((q - 1) / 2)) * self.inv_int_p
                    )
                    d_pnorm_vm = d_pnorm_vm_1 * d_pnorm_vm_2

                    d_vm_sigmaA = (von_mises_squared ** -0.5) \
                                * self.multiply_VM_matrix(sv, real_stress_int)\
                                * self.shell_thickness * det_jac

                    d_sigmaA_spf = 0.5 * self.xe_all(elmt.label, xe) ** (-0.5)

                    # Contribution of each integration point divided over the
                    # element volume.
                    self.d_pnorm_spf[elmt.label] += (
                        d_pnorm_vm * d_vm_sigmaA * d_sigmaA_spf / elmt_vol
                    )


def material_constraint_sensitivity(
        mdb, material_constraint, mesh_uniformity, opt_method, model_name,
        part_name, density = None
    ):
    """ Material Constraint Sensitivity function
    Determines the sensitivity of the mass or volume constraints to changes
    in the density of each element.
    The output is a dictionary with the values determined for each element.

    Unless the mesh is non-uniform (where the size of the elements can differ),
    the sensitivity will be set to 1.0 for all elements. This can be understood
    as all elements contributing equally to the mass or volume constraint
    imposed. This simplification is used to reduce the computational cost of
    the function.

    If the mesh is non-uniform, this function will execute multiple queries to
    the ABAQUS model, which can significantly increase the computational cost.

    Note: the commands specifyThickness=True,thickness=1.0 cause ABAQUS to
    consider a thickness of 1.0 only if the element does not have a thickness
    assigned to it. Therefore, in 2D cases with unknown thicness, the volume
    obtained is numerically equal to the area of the element.

    Inputs:
  -------
    - mdb (Mdb): model database from ABAQUS.
    - material_constraint (int): variable defining if the material constraint
      has been applied to the volume or mass of the region to be optimized.
    - mesh_uniformity (int): variable defining of the mesh is uniform (all
      elements have the same size) or not.
    - opt_method (int): variable defining the optimization method to be used.
    - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.
    - density (float): value of the material density (units of mass/volume).
```

```
    Outputs:
    --------
    - mat_const_sensitivity (dict): dictionary with the material constraint
      sensitivity of each element.
    - elmt_volume (dict): dictionary with the element volume of each element.
      """
      mat_const_sensitivity = {}
      elmt_volume = {}
      part = mdb.models[model_name].parts[part_name]
      all_elmts = part.elements

      # Confirm that the density material property has been defined if using a
      # mass-based material constraint in a model with non-uniform mesh.
      # Notice that if the mesh is uniform, we can set the sensitivities equal to
      # 1.0 (as they contribute equally to the total mass), and avoid the need
      # for a defined material density property.
      if material_constraint==0 and density==None and mesh_uniformity==0:
          raise Exception(
              "Missing material density property - it is necessary to define \n"
              "the density of the material used in order to apply a mass-based\n"
              "material constraint in a model with a non-uniform mesh. \n"
          )

      # Determine the sensitivities of the mass constraint to changes the in
      # mass or volume of each element.
      #
      # For volume constraint, the sensitivity is equal to the element volume.
      if material_constraint == 1:

          # If the mesh is uniform and all elements have the same size, we
          # can set the sensitivity equal to 1.0, reducing the number of
          # queries submitted in Abaqus and greatly reducing the processing
          # time.
          if mesh_uniformity == 1:
              for elmt in all_elmts:
                  mat_const_sensitivity[elmt.label] = 1.0

          # If the mesh is not uniform (ex: using adaptive meshes), the code
          # will query the volume of each individual element.
          # Please note that this loop may be computationally expensive due
          # to the potentially large number of queries submitted.
          # However, if the mesh is uniform, it is acceptable to set this
          # sensitivity equal and constant to all elements (usually, set to 1.0).
          elif mesh_uniformity == 0:
              for elmt in all_elmts:
                  region = mesh.MeshElementArray((elmt,))
                  vol = part.getMassProperties(regions = region,
                                               specifyThickness = True,
                                               thickness=1.0)['volume']
                  mat_const_sensitivity[elmt.label] = vol

          else:
              raise Exception(
                  "Unexpected value for the mesh_uniformity variable in the \n"
                  "in the material_constraint_sensitivity function"
              )

      # For mass constraint the sensitivity is equal to the mass of a fully solid
      # element (design variable or design density equal to 1.0).
      elif material_constraint == 0:

          # If the mesh is uniform and all elements have the same size
          # (and consequently, same mass), we can set the sensitivity equal
          # to 1.0, reducing the number of queries submitted in Abaqus and
          # greatly reducing the processing time.
          if mesh_uniformity == 1:
              for elmt in all_elmts:
                  mat_const_sensitivity[elmt.label] = 1.0

          # If the mesh is not uniform (ex: using adaptive meshes) causing
          # the elements to have different masses, the code will query the
          # volume of each individual element.
          # Please note that this loop may be computationally expensive due to
```

```
        # the potentially large number of queries submitted.
        # However, if the mesh is uniform, it is acceptable to set this
        # sensitivity equal and constant to all elements (usually, set to 1.0).
        # Notice that the code multiplies the density for the volume instead
        # of using the command getMassProperties()['mass'].
        # This is because user may set the initial element design density to
        # be different than 1.0, which would lead to an incorrect
        # sensitivity output.
        elif mesh_uniformity == 0:
           for elmt in all_elmts:
               region = mesh.MeshElementArray((elmt,))
               vol = part.getMassProperties(regions = region,
                                            specifyThickness = True,
                                            thickness = 1.0)['volume']

               mat_const_sensitivity[elmt.label] = density*vol
        else:
            raise Exception(
                "Unexpected value for the mesh_uniformity variable in the \n"
                "material_constraint_sensitivity function."
            )

    else:
        raise Exception(
            "Unexpected value in the material_constraint variable in the \n"
            "variable in the material_constraint_sensitivity function. \n"
        )

    # When solving stress dependent problems, it is necessary to determine
    # the volume of each element.
    # This information is used in the integration of the constraint values
    # through each element.
    if opt_method >= 4:

        # In volume constrained problems with non-uniform mesh, this
        # information has already been obtained in the previous loop.
        # The code will only copy the variable.
        if material_constraint == 1 and mesh_uniformity == 0:
            elmt_volume = mat_const_sensitivity.copy()

        # If the mesh is uniform, all elements have the same volume.
        # The code will query the volume of the first element, and assign it to
        # all other elements in the dictionary.
        elif mesh_uniformity == 1:
            sample_elmt = all_elmts[0]
            region = mesh.MeshElementArray((sample_elmt,))
            sample_volume = part.getMassProperties(regions = region,
                                                   specifyThickness = True,
                                                   thickness = 1.0)['volume']
            for elmt in all_elmts:
                elmt_volume[elmt.label] = sample_volume

        # If the element volume has not been extracted previously and the mesh
        # is non-uniform, the code will query each element individually.
        elif material_constraint == 0 and mesh_uniformity == 0:
            for elmt in all_elmts:
                region = mesh.MeshElementArray((elmt,))
                vol = part.getMassProperties(regions = region,
                                             specifyThickness = True,
                                             thickness = 1.0)['volume']
                elmt_volume[elmt.label] = vol
        else:
            raise Exception(
                "Unexpected combination of parameters found in the \n"
                "material_constraint_sensitivity when preparing the element\n"
                "volume for constrained topology optimization."
            )

    return mat_const_sensitivity, elmt_volume


#%% Material and stress constraint evaluation.
class MaterialConstraint():
    """ Material constraint class
```

```
    This class is responsible for updating the value of the material
    constraint at each iteration.

    The value of the material constraint may be constant or variable
    during the topology optimization process.

    When variable, the material constraint will be decreased in each
    iteration at a percentual ratio defined by the 'evol_ratio' variable.
    The result is a gradual decrease of the material constraint, until its
    desired value is reached. To set the material constraint as variable,
    the value of 'evol_ratio' should be lower than the value of the
    'target_material'.

    Choosing an 'evol_ratio' to 1.0 (or to any value larger than the
    intended material constraint) will set the material constraint to a
    constant value, equal to the target_material variable.

    Attributes:
    -----------
    - target_material (float): maximum value of the material constraint.
    - evol_ratio (float): ratio at which the material constraint should be
      imposed/reduced.
    - mat_const_sensitivities (dict): dictionary with the material constraint
      sensitivity to changes in the design variables.

    Method:
    -------
    - update_constraint(current_material, target_material_history,
      editable_xe): updates the current value of the material constraint and
      updates the data records.
    """
    def __init__(self, target_material, evol_ratio, mat_const_sensitivities):
        self.target_material = target_material
        self.evol_ratio = evol_ratio
        self.mat_const_sensitivities = mat_const_sensitivities

    def update_constraint(
            self, current_material, target_material_history, editable_xe
        ):
        """ Update constraint method

        Updates the value of the material constraint for the next iteration.

        The material constraint is updated according to the following formula,
        as long as it is larger than the target_material constraint value:

            Constraint = max(Target_material,
                             Current_material * (1 - evol_ratio)
            )

        Then, updates the material constraint records.
        Notice that the function uses the material constraint sensitivities,
        which are equal to the volume or mass of each element. This allows
        the code to account for the existance of non-uniform meshes.

        Inputs:
        -------
        - current_material (list): list with the current value of the material
          constraint.
        - target_material_history (list): list with the values of the material
          constraint that the code tried to acchieve.
        - editable_xe (dict): dictionary with the values of the design
          densities.

        Outputs:
        --------
        - current_material (list): list with the current value of the material
          constraint.
        - target_material_history (list): list with the values of the material
          constraint that the code tried to acchieve.
        """
        # Determines the current material fraction.
        max_mat = 0
```

```
        current_mat = 0
        for elmt_label, density in editable_xe.items():
            max_mat += self.mat_const_sensitivities[elmt_label]
            current_mat += self.mat_const_sensitivities[elmt_label] * density

        current_material.append(current_mat / max_mat)

        # Determines the target material fraction according to the evol_ratio.
        # Then selects the largest target material value and appends it to the
        # record.
        intermediate_mat_val = current_material[-1] * (1 - self.evol_ratio)
        next_constraint_value = max(self.target_material, intermediate_mat_val)
        target_material_history.append(next_constraint_value)

        return current_material, target_material_history


def p_norm_approximation(stress_vector_int, inv_int_p, q, mult_VM_matrix):
    """ P-norm maximum Von-Mises stress approximation function

    Determines the value of the maximum stress approximation. This function
    assumes that the maximum stress is determined by the P-norm approximation
    function, as:

        sigmaPN = (inv_int_p * sum(vm_stress ** q)) ** (1 / q)

    Where 'inv_int_p' is the inverse of the number of stress evaluation points,
    in this case the inverse of the number integration points.

    Inputs:
    -------
    - stress_vector_int (dict): dictionary with stress vector of each
      integration point in each element.
    - inv_int_p (float): inverse of the number of integration points.
    - q (float): value of the exponent used in the p-norm approximation.
    - mult_VM_matrix (function): function that multiplies two vectors by the
      Von-Mises matrix.

    Output:
    -------
    - stress_constraint (numpy array): value of the stress constraint, defined
      as a fraction.
    """
    vm_stress_q = []

    # Determine and store the Von-Mises stress in each integration point,
    # raised to the P-norm exponential factor.
    for elmt in stress_vector_int.values():
        for int_p in elmt.values():
            vm_stress_q.append((mult_VM_matrix(int_p, int_p) ** 0.5) ** q)

    # Calculate P-norm approximation and stress constraint.
    sigmaPN = np.sum(inv_int_p * np.array(vm_stress_q)) ** (1.0 / q)

    return sigmaPN


def stress_constraint_evaluation(sigmaPN, s_max):
    """ Stress constraint evaluation function

    Determines the value of the stress constraint, given the current maximum
    stress and the maximum allowable stress.

    Inputs:
    -------
    - sigmaPN (float): p-norm approximation of the maximum Von-Mises stress.
    - s_max (float): maximum stress allowed in the topology optimized design.

    Output:
    -------
    - stress_constraint (numpy array): value of the stress constraint, defined
      as a fraction.
    """
    stress_constraint = float((sigmaPN / s_max) - 1.0)
```

```
    return np.array(stress_constraint, ndmin=2)


#%% Data filtering.
def init_filter(rmax, elmts, all_elmts, nodes, mdb, model_name, part_name):
    """ Initiate filter function

    This wrapper function initializes and prepares a DataFilter object, which
    tis used o apply a blurring filter to the results obtained during the
    topology optimization process.

    If the user did not request the use of a blurring filter (setting the
    search radius 'rmax' to 0), the function outputs a None variable.

    Inputs:
    -------
    - rmax (float): search radius that defines the maximum distance between the
      center of the target element and the edge of its neighbouring region.
    - elmts (MeshElementArray): element_array from ABAQUS with the relevant
      elements in the model.
    - all_elmts (MeshElementArray): element array from ABAQUS with all elements
      considered in the topology optimization process.
    - nodes (MeshNodeArray): mesh node array from ABAQUS with all nodes that
      belong to elements considered in the topology optimization process.
    - mdb (Mdb): model database from ABAQUS.
  - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.

    Output:
    -------
    - opt_filter (class): DataFilter instance with the filter preparation
      already concluded.
    """
    if rmax > 0:
        opt_filter = DataFilter(rmax, elmts, all_elmts, nodes,
                                mdb, model_name, part_name)

        opt_filter.filter_preparation()
    else:
        opt_filter = None

    return opt_filter


class DataFilter:
    """ Data Filter class

    Class responsible for creating a filter map, defining the influence between
    the different elements, and applying it.

    Attributes:
    -----------
    - rmax (float): search radius that defines the maximum distance between the
      center of the target element and the edge of its neighbouring region.
    - elmts (MeshElementArray): element_array from ABAQUS with the relevant
      elements in the model.
    - all_elmts (MeshElementArray): element array from ABAQUS with all elements
      considered in the topology optimization process.
    - nodes (MeshNodeArray): mesh node array from ABAQUS with all nodes that
      belong to elements considered in the topology optimization process.
    - mdb (Mdb): model database from ABAQUS.
      - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.

    Methods:
    --------
    - filter_preparation(): creates a filter map, defining how the elements
      interact and influence each others.
    - filter_function(var_dictionary, elmt_keys): applies the blurring filter
      to the variable/property of a given list of elements selected.
    - filter_densities(editable_xe, xe, xe_min, dp): applies the
      'filter_function' method, considering the differences between
      'editable_xe' and 'xe', as well as the minimum density condition imposed
```

```
    by 'xe_min'.
    """
    def __init__(
            self, rmax, elmts, all_elmts, nodes, mdb, model_name, part_name
        ):

        self.rmax = rmax
        self.elmts = elmts
        self.all_elmts = all_elmts
        self.nodes = nodes
        self.mdb = mdb
        self.model_name = model_name
        self.part_name = part_name


    def filter_preparation(self):
        """Filter Preparation method

        This function outputs a dictionary that stores two lists for each
        element. The first list contains the labels of the elements that are
        within a radius 'rmax' of the center of the target element. The
        elements that are fully contained by this radius define the
        'neighborhood' of the target element. The second list contains a
        measurement of how close each element is to the target element,
        defined by the value of rmax minus the actual distance between
        elements.

        As a result, the dictionary output consists of a map that defines how
        each element is affected by the neighbouring elements, when using
        sensitivity filters. This information is also stored as an attribute
        of the DataFilter class.

        Note: The neighborhood of a given element only considers elements that
        are fully within the 'rmax' radius. Elements only partially intersected
        by the search sphere are not considered.

        Outputs:
        --------
        - filter_map (dict): dictionary containing, for each element, the
          labels of the elements in their neighbourhood and their pondered
          contribution to the filtered result.
        """
        center_coordinates, filter_map = {}, {}

        # Calculate the coordinates of the center of each element
        for elmt in self.all_elmts:

            #labels of the nodes connected to each element.
            node_labels = elmt.connectivity
            center_coordinates[elmt.label] = np.zeros((3))

            #calculates an average of the coordinates of the nodes connected to
            #the element, leading to the coordinate of the center of the
            #element.
            for label in node_labels:
                center_coordinates[elmt.label] += \
                    np.divide(self.nodes[label].coordinates,
                              len(node_labels))

        for el in self.elmts:
            filter_map[el.label] = [[],[]]
            center = (center_coordinates[el.label][0],
                      center_coordinates[el.label][1],
                      center_coordinates[el.label][2])
            radius = (self.rmax)

            # Selects the elements that are FULLY WITHIN a sphere of radius
            # rmax, centered in the middle of the element 'el'.
            neighborhood = self.all_elmts.getByBoundingSphere(center = center,
                                                              radius = radius)

            # If no element was totally within the search radius, include
            # the central element as the only member of the neighborhood.
            # Data recorded as a list to allow iteration of its contents.
```

```
            if len(neighborhood) == 0:
                neighborhood = [self.all_elmts.getFromLabel(el.label)]

            #The following three lines were intentionally left commented.
            #They create a set for each neighbourhood, which is useful for
            #debugging purposes and for understanding the functioning of the
            #filter.
            #
            #self.mdb.models[self.model_name].parts[self.part_name].Set(
            #                   elements=neighborhood,
            #                   name = "Neighborhood element " + str(el.label))

            # Determines the influence of each element in the neighborhood.
            for em in neighborhood:
                displacement_vector = np.subtract(center_coordinates[el.label],
                                                  center_coordinates[em.label])
                distance = np.sqrt(np.sum(np.power(displacement_vector,2)))

                # Records the labels of the elements within the neighborhood.
                filter_map[el.label][0].append(em.label)

                # Records 'how close' (as in the opposite of the distance)
                # the neighbours are to the central element.
                filter_map[el.label][1].append(self.rmax - distance)

            # Determines the influence of each neighbour to the central element.
            sum_proximity = np.sum(filter_map[el.label][1])
            elmt_influence = np.divide(filter_map[el.label][1], sum_proximity)
            filter_map[el.label][1] = elmt_influence

            self.filter_map = filter_map

    def filter_function(self, var_dictionary, elmt_keys):
        """Filter function method

        Applies a filter to each element. The filter applied considers a
        pondered average of the variable being filtered as a function of how
        close the neighboring elements are to the target element.

        Outputs a dictionary with the filtered variable for each element.

        Inputs:
        -------
        - var_dictionary (dict): dictionary with one entry for each element,
          storing the value of the variable to be filtered.
        - elmt_keys (list): list with the keys of the elements to be filtered.

        Output:
        -------
        - var_dictionary (dict): dictionary with one entry for each element,
          storing the value of the filtered variable.
        """
        unfiltered_data = var_dictionary.copy()
        for el in elmt_keys:
            var_dictionary[el] = 0.0

            # Calculates a pondered average of a variable (ex:sensitivity)
            # considering the contribution of each element in the neighborhood.
            # The contribution of each element is determined in the
            # function filter_preparation.
            for i in range(len(self.filter_map[el][0])):
                original_value = unfiltered_data[self.filter_map[el][0][i]]
                element_contribution = self.filter_map[el][1][i]
                var_dictionary[el] += original_value * element_contribution

        return var_dictionary

    def filter_densities(self, editable_xe, xe, xe_min, dp):
        """ Filter density method

        Decorator method.  Applies the sensitivity filter to both dictionaries
        'xe' and 'editable_xe', considering their differences. The non-editable
        elements included in 'xe' are not altered.
```

```
        Inputs:
        -------
        - editable_xe: dictionary with the densities (design variables) of each
          editable element in the model.
        - xe: dictionary with the densities (design variables) of each
          relevant element in the model.
        - xe_min: minimum density allowed for the element. I.e. minimum value
          allowed for the design variables.
        - dp: number of decimals places to be considered in the interpolation.
          By definition, equal to the number of decimal places in xe_min.
        """
        xe = self.filter_function(xe, editable_xe.keys())

        for key in editable_xe.keys():
            temp_value = max(xe_min, round(xe[key], dp))
            editable_xe[key] = temp_value
            xe[key] = temp_value

        return editable_xe, xe


#%% Optimization algorithms
def oc_discrete(
        editable_xe, xe, ae, p, target_material, mat_constr_sensitivities,
        xe_min
    ):
    """ Optimality Criteria function - discrete version

    Uses the optimality criteria to update the design variables. This
    implementation of the OC only considers the minimization of the objective
    function with a mass or volume constraint.
    This implementation considers both the increase and reduction of the
    elements density (bi-directional evolution).

    Inputs:
    -------
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.
    - move_limit (float): maximum change in the design variables during each
      iteration.
    - ae (dict): dictionary with the sensitivity of the objective function to
      changes in each design variable.
    - p (float): SIMP penalty factor.
    - target_material (float): ratio between the target volume or mass and the
      volume or mass of a full density design.
    - mat_constr_sensitivities (dict): dictionary with the material constraint
      sensitivities (mass or volume) of each element.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.

  Outputs:
  --------
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.

    Notes: setting the input variable 'p' to a large value will cause the
    topology optimization to behave in a discrete manner, considering only
    elements with either maximum or minimum density.
    """
    # Sets minimum and maximum values for the Lagrange Multiplier.
    ae_values = -np.array(ae.values())
    lo, hi = min(ae_values), max(ae_values)

    # Sorts data into arrays for easier processing.
    elmt_material = np.array([])
    total_material = 0.0
    for key in editable_xe.keys():
        elmt_material = np.append(elmt_material, mat_constr_sensitivities[key])
        total_material += mat_constr_sensitivities[key]
```

```
    # Applies bi-particion algorithm to determine the solid and void elements.
    while abs((hi - lo) / hi) > 1.0e-5:
        th = (lo + hi) / 2.0
        for key in editable_xe.keys():
            if -ae[key] > th:
                editable_xe[key], xe[key] = 1.0, 1.0
            else:
                editable_xe[key], xe[key] = xe_min, xe_min

        densities = np.array(editable_xe.values())

        # Confirms if the material constraint is respected and adjusts
        # accordingly.
        if sum(densities * elmt_material) / total_material > target_material:
            lo = th
        else:
            hi = th

    return editable_xe, xe


def oc_continuous(
        editable_xe, xe, move_limit, ae, p, target_material,
        mat_constr_sensitivities, xe_min, dp
    ):
    """ Optimality Criteria function - continuous version

    Uses the optimality criteria to update the design variables. This
    implementation of the OC only considers the minimization of the objective
    function with a mass or volume constraint.
    This implementation considers both the increase and reduction of the
    elements density (bi-directional evolution).

    Inputs:
    -------
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.
    - move_limit (float): maximum change in the design variables during each
      iteration.
    - ae (dict): dictionary with the sensitivity of the objective function to
      changes in each design variable.
    - p (float): SIMP penalty factor.
    - target_material (float): ratio between the target volume or mass and the
      volume or mass of a full density design.
    - mat_constr_sensitivities (dict): dictionary with the material constraint
      sensitivities (mass or volume) of each element.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.

    Outputs:
    --------
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.

    Notes: setting the input variable 'p' to a large value will cause the
    topology optimization to behave in a discrete manner, considering only
    elements with either maximum or minimum density.
    """
    # Sets minimum and maximum values for the Lagrange Multiplier.
    ae_values = -np.array(ae.values())
    lo, hi = min(ae_values), max(ae_values)

    densities = np.array([])
    sensitivities = np.array([])
    elmt_material = np.array([])
    labels = []
    total_material = 0.0

    # Reorganizes data into numpy arrays for an easier processing and
    # determines the total_material of the model.
```

```
    for key in editable_xe.keys():
        densities = np.append(densities, float(editable_xe[key]))
        sensitivities = np.append(sensitivities, float(ae[key]))
        labels.append(key)
        elmt_material = np.append(elmt_material, mat_constr_sensitivities[key])
        total_material += mat_constr_sensitivities[key]

    # Applies bi-particion algorithm to determine the solid and void elements.
    while abs((hi - lo) / hi) > 1.0e-4:
        th = (lo + hi) / 2.0
        temp_densities = densities * (-sensitivities / th) ** 0.5
        for i in range(0, len(temp_densities)):

            if temp_densities[i] <= max(xe_min, densities[i] - move_limit):
                temp_densities[i] = max(xe_min, densities[i] - move_limit)

            elif temp_densities[i] >= min(1.0, densities[i] + move_limit):
                temp_densities[i] = min(1.0, densities[i] + move_limit)

            else:
                pass

        # Confirms if the material constraint is respected and adjusts
        # accordingly.
        current_material = sum((temp_densities)*elmt_material)
        current_material_fraction = current_material / total_material
        if current_material_fraction > target_material:
            lo = th
        else:
            hi = th

    # Rounds the output considering 'xe_min'.
    for i in range(0, len(labels)):
        editable_xe[labels[i]] = max(xe_min, round(temp_densities[i], dp))
        xe[labels[i]] = max(xe_min, round(temp_densities[i], dp))

    return editable_xe, xe


def mma(
        editable_xe, xe, move_limit, obj_der, p, xe_min, target_material,
        material_gradient, opt_method, dp, objh, iteration, x1, x2, low, upp,
        p_norm_history = None, stress_const_gradient = None,
        stress_constraint = None, s_max = None
    ):
    """ Wrapper function for the Method of Moving Assymptotes

    Wrapper function (or decorator) that reformats the variables used in the
    topology optimization process, converting them into a format the is
    compatible with the MMA function implemented by Kristen Svanberg.

    The value of the objective function in the first iteration, as well as
    the value of the constraints in the first iteration, are used as a
    normalization factor in order to avoid numerical errors. In the
    particular case of the material gradient, the maximum allowed material
    used as a normalization factor.

    Inputs:
    -------
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.
    - move_limit (float): maximum change in the design variables during each
      iteration.
    - obj_der (dict): dictionary with the sensitivity of the objective function
      to changes in each design variable.
    - p (float): SIMP penalty factor.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.
    - target_material (float): ratio between the target volume or mass and the
      volume or mass of a full density design.
    - material_gradient (dict): dictionary with the material constraint
      sensitivities (mass or volume) of each element.
```

```
    - objh (list): record with values of the objective function.
    - iteration (int): number of the current iteration.
    - x1 (dict): equivalent of xe for the last iteration.
    - x2 (dict): equivalent of xe for the second to last iteration.
    - low (array): array with the minimum search value considered for each
      element. Obtained as an output of the mmasub function.
    - upp (array): array with the maximum search value considered for each
      element. Obtained as an output of the mmasub function.
    - p_norm_history (list): record with the values of the p-norm
      approximation.
    - stress_const_gradient (dict): sensitivity of the stress constraint to
      changes in the design variables.
    - stress_constraint (float): value of the stress constraint.
    - s_max (float): maximum allowable stress.

  Ouputs:
  -------
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.
    - low (array): array with the minimum search value considered for each
      element. Obtained as an output of the mmasub function.
    - upp (array): array with the maximum search value considered for each
      element. Obtained as an output of the mmasub function.
    - lam (array): vector with the Lagrange multipliers.
    - fval (array): vector with the constraint values.
    - ymma, zmma (array): arrays with the values of the variables y_i and z in
      the current MMA subproblem.

    Notes:
    ------
    - The functions 'mmasub' and 'subsolv' were developed by Arjen Deetman
      and shared under the terms of a GNU General Public License. The summary
      of the license description can be found in these comment section of
      both functions. For more information, please follow the source link:
      https://github.com/arjendeetman/GCMMA-MMA-Python
    """
    num_elements = len(editable_xe)

    # Initializes variables to store the inputs for the mmasub and mmasolve
    # functions (either arrays or scalar variables).
    labels = []
    f0val = objh[-1] #/ objh[0]
    xval = np.array([])
    df0dx = np.array([])
    xold1 = np.array([])
    xold2 = np.array([])
    elmt_material_list = np.array([])
    material_sensitivity = np.array([])
    xmin = np.ones((num_elements,1)) * xe_min
    xmax = np.ones((num_elements,1))

    # Converting from dictionaries to arrays. The labels are stored in a list
    # since dictionaries may be unsorted depending on the Python version.
    max_material = sum(material_gradient.values()) * target_material
    for key in editable_xe.keys():
        # Label, objective function value and its derivative.
        labels.append(key)
        xval = np.append(xval, editable_xe[key])
        df0dx = np.append(df0dx, obj_der[key]) #/ objh[0])

        # Material constraint gradient.
        norm_mat_grad = material_gradient[key] / (max_material)
        material_sensitivity = np.append(material_sensitivity, norm_mat_grad)
        elmt_material = material_gradient[key] * editable_xe[key]
        elmt_material_list = np.append(elmt_material_list, elmt_material)

    # Previous design variables.
    if iteration > 1:
        for key in editable_xe.keys():
            xold1 = np.append(xold1, x1[key])
            xold2 = np.append(xold2, x2[key])
```

```
    # Reshaping arrays.
    xval.shape = (num_elements, 1)
    df0dx.shape = (num_elements, 1)
    material_sensitivity.shape = (1, num_elements)

    # Determine current constraint values and reformat sensitivities.
    # For stress or compliance minimization:
    if opt_method in [2, 6]:
        num_constraints = 1

        mat_const_value = (sum(elmt_material_list) / max_material) - 1.0
        fval = np.array([[mat_const_value]])

        dfdx = material_sensitivity

    # For stress constrained compliance minimization:
    elif opt_method in [4]:
        num_constraints = 2

        stress_sens = np.array([])

        for key in editable_xe.keys():
            # Stress sensitivity.
            norm_stress_sens = stress_const_gradient[key] / s_max
            stress_sens = np.append(stress_sens, norm_stress_sens)

        # Stack the constraint values into a single array.
        mat_const_value = (sum(elmt_material_list) / max_material) - 1.0
        fval = np.concatenate(
                (np.array([[mat_const_value]]), stress_constraint), axis = 1
        ).reshape(2, 1)

        dfdx = np.vstack((material_sensitivity[0], stress_sens))

    else:
        raise Exception(
            "Unexpected value for 'opt_method' found in function 'MMA'."
        )

    # Determines the min and max values of the design variables, which can be
    # narrowed down by the MMA algorithm.
    if iteration > 1:
        xold1.shape = (num_elements, 1)
        xold2.shape = (num_elements, 1)
        for i in range(0,len(xmin)):
            low[i][0] = max(xmin[i][0], low[i][0])
            upp[i][0] = min(xmax[i][0], upp[i][0])

    # Defines the optimization problem for 'mmasub'.
    a0 = 1.0
    a = np.zeros((num_constraints,1))
    c = np.ones((num_constraints,1))*10**6
    d = np.zeros((num_constraints,1))
    move = move_limit

    xmma, ymma, zmma, lam, xsi, eta, mu, zet, s, low, upp = mmasub(
        num_constraints, num_elements, iteration, xval, xmin, xmax, xold1,
        xold2, f0val, df0dx, fval, dfdx, low, upp, a0, a, c, d, move
    )

    # Rounds the output considering 'xe_min'.
    for i in range(0,len(labels)):
        editable_xe[labels[i]] = max(xe_min, round(xmma[i][0], dp))
        xe[labels[i]] = max(xe_min, round(xmma[i][0], dp))

    return editable_xe, xe, low, upp, lam, fval, ymma, zmma


def mmasub(
        m, n, iteration, xval, xmin, xmax, xold1, xold2, f0val, df0dx, fval,
        dfdx, low, upp, a0, a, c, d, move
    ):
    """
    COPYRIGHT AND LICENSE: This function was extracted from the
```

```
GCMMA-MMA-Python code developed by Arjen Deetman.

Source: https://github.com/arjendeetman/GCMMA-MMA-Python
last visited on the 21st of October of 2020

################# Copyright (c) 2020 Arjen Deetman ########################
GCMMA-MMA-Python
Python code of the Method of Moving Asymptotes (Svanberg, 1987). Based on
the GCMMA-MMA-code written for MATLAB by Krister Svanberg. The original
work was taken from http://www.smoptit.se/ under the GNU General Public
License. If you download and use the code, Krister Svanberg would
appreciate if you could send him an e-mail and tell who you are and what
your plan is (e-mail adress can be found on his website). The user should
reference to the academic work of Krister Svanberg when work will be
published.

GCMMA-MMA-Python is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the Free
Software Foundation; either version 3 of the License, or (at your option)
any later version.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
more details.

You should have received a copy of the GNU General Public License (file
LICENSE) along with this file. If not, see http://www.gnu.org/licenses/.


############################## References ###################################
Svanberg, K. (1987). The Method of Moving Asymptotes - A new method for
structural optimization. International Journal for Numerical Methods in
Engineering 24, 359-373. doi:10.1002/nme.1620240207

Svanberg, K. (n.d.). MMA and GCMMA - two methods for nonlinear optimization.
Retrieved August 3, 2017 from https://people.kth.se/~krille/mmagcmma.pdf


########################## Function description ############################
This function mmasub performs one MMA-iteration, aimed at solving the nonlinear
                                          programming problem:

Minimize    f_0(x) + a_0*z + sum( c_i*y_i + 0.5*d_i*(y_i)^2 )
subject to  f_i(x) - a_i*z - y_i <= 0,  i = 1,...,m
            xmin_j <= x_j <= xmax_j,    j = 1,...,n
            z >= 0,   y_i >= 0,         i = 1,...,m
INPUT:
    m     = The number of general constraints. = 1
    n     = The number of variables x_j. = len(ELMTS)
    iteration  = Current iteration number ( =1 the first time mmasub is called)
                                                .
    xval  = Column vector with the current values of the variables x_j.
    xmin  = Column vector with the lower bounds for the variables x_j.
    xmax  = Column vector with the upper bounds for the variables x_j.
    xold1 = xval, one iteration ago (provided that iteration>1).
    xold2 = xval, two iterations ago (provided that iteration>2).
    f0val = The value of the objective function f_0 at xval.
    df0dx = Column vector with the derivatives of the objective function
            f_0 with respect to the variables x_j, calculated at xval.
    fval  = Column vector with the values of the constraint functions f_i,
                                                calculated at xval.
    dfdx  = (m x n)-matrix with the derivatives of the constraint functions
            f_i with respect to the variables x_j, calculated at xval.
            dfdx(i,j) = the derivative of f_i with respect to x_j.
    low   = Column vector with the lower asymptotes from the previous
            iteration (provided that iteration>1).
    upp   = Column vector with the upper asymptotes from the previous
            iteration (provided that iteration>1).
    a0    = The constants a_0 in the term a_0*z. = 1.0, which leads z to tend
                                                to 0.0
    a     = Column vector with the constants a_i in the terms a_i*z. # a0 = 1
                                                and ai = 0 for all i > 0
```

```
        c     = Column vector with the constants c_i in the terms c_i*y_i. = [large
                                                  number], makes y tend to 0 in an
                                                  optimal solution
        d     = Column vector with the constants d_i in the terms 0.5*d_i*(y_i)^2.
                                                  = [1.0]
        move  = Value of the allowable movement of each design variable.
    OUTPUT:
        xmma  = Column vector with the optimal values of the variables x_j
                in the current MMA subproblem.
        ymma  = Column vector with the optimal values of the variables y_i
                in the current MMA subproblem.
        zmma  = Scalar with the optimal value of the variable z
                in the current MMA subproblem.
        lam   = Lagrange multipliers for the m general MMA constraints.
        xsi   = Lagrange multipliers for the n constraints alfa_j - x_j <= 0.
        eta   = Lagrange multipliers for the n constraints x_j - beta_j <= 0.
        mu    = Lagrange multipliers for the m constraints -y_i <= 0.
        zet   = Lagrange multiplier for the single constraint -z <= 0.
        s     = Slack variables for the m general MMA constraints.
        low   = Column vector with the lower asymptotes, calculated and used
                in the current MMA subproblem.
        upp   = Column vector with the upper asymptotes, calculated and used
                in the current MMA subproblem.
    """

    epsimin = 0.0000001
    raa0 = 0.00001
    albefa = 0.1
    asyinit = 0.5
    asyincr = 1.2
    asydecr = 0.7
    eeen = np.ones((n, 1))
    eeem = np.ones((m, 1))
    zeron = np.zeros((n, 1))
    # Calculation of the asymptotes low and upp
    if iteration <= 2:
        low = xval-asyinit*(xmax-xmin)
        upp = xval+asyinit*(xmax-xmin)
    else:
        zzz = (xval-xold1)*(xold1-xold2)
        factor = eeen.copy()
        factor[np.where(zzz>0)] = asyincr
        factor[np.where(zzz<0)] = asydecr
        low = xval-factor*(xold1-low)
        upp = xval+factor*(upp-xold1)
        lowmin = xval-10*(xmax-xmin)
        lowmax = xval-0.01*(xmax-xmin)
        uppmin = xval+0.01*(xmax-xmin)
        uppmax = xval+10*(xmax-xmin)
        low = np.maximum(low,lowmin)
        low = np.minimum(low,lowmax)
        upp = np.minimum(upp,uppmax)
        upp = np.maximum(upp,uppmin)
    # Calculation of the bounds alfa and beta
    zzz1 = low+albefa*(xval-low)
    zzz2 = xval-move*(xmax-xmin)
    zzz = np.maximum(zzz1,zzz2)
    alfa = np.maximum(zzz,xmin)
    zzz1 = upp-albefa*(upp-xval)
    zzz2 = xval+move*(xmax-xmin)
    zzz = np.minimum(zzz1,zzz2)
    beta = np.minimum(zzz,xmax)
    # Calculations of p0, q0, P, Q and b
    xmami = xmax-xmin
    xmamieps = 0.00001*eeen
    xmami = np.maximum(xmami,xmamieps)
    xmamiinv = eeen/xmami
    ux1 = upp-xval
    ux2 = ux1*ux1
    xl1 = xval-low
    xl2 = xl1*xl1
    uxinv = eeen/ux1
    xlinv = eeen/xl1
    p0 = zeron.copy()
```

```
    q0 = zeron.copy()
    p0 = np.maximum(df0dx,0)
    q0 = np.maximum(-df0dx,0)
    pq0 = 0.001*(p0+q0)+raa0*xmamiinv
    p0 = p0+pq0
    q0 = q0+pq0
    p0 = p0*ux2
    q0 = q0*xl2
    P = np.zeros((m,n)) ## @@ make sparse with scipy?
    Q = np.zeros((m,n)) ## @@ make sparse with scipy?
    P = np.maximum(dfdx,0)
    Q = np.maximum(-dfdx,0)
    PQ = 0.001*(P+Q)+raa0*np.dot(eeem,xmamiinv.T)
    P = P+PQ
    Q = Q+PQ
    P = (diags(ux2.flatten(),0).dot(P.T)).T
    Q = (diags(xl2.flatten(),0).dot(Q.T)).T
    b = (np.dot(P,uxinv)+np.dot(Q,xlinv)-fval)
    # Solving the subproblem by a primal-dual Newton method
    xmma,ymma,zmma,lam,xsi,eta,mu,zet,s = subsolv(m,n,epsimin,low,upp,alfa,beta,p0,
                                                  q0,P,Q,a0,a,b,c,d)
    # Return values
    return xmma,ymma,zmma,lam,xsi,eta,mu,zet,s,low,upp


def subsolv(m, n, epsimin, low, upp, alfa, beta, p0, q0, P, Q, a0, a, b, c, d):
    """
    COPYRIGHT AND LICENSE: This function was extracted from the
    GCMMA-MMA-Python code developed by Arjen Deetman.

    Source: https://github.com/arjendeetman/GCMMA-MMA-Python
    last visited on the 21st of October of 2020

    ################## Copyright (c) 2020 Arjen Deetman ########################
    GCMMA-MMA-Python
    Python code of the Method of Moving Asymptotes (Svanberg, 1987). Based on
    the GCMMA-MMA-code written for MATLAB by Krister Svanberg. The original
    work was taken from http://www.smoptit.se/ under the GNU General Public
    License. If you download and use the code, Krister Svanberg would
    appreciate if you could send him an e-mail and tell who you are and what
    your plan is (e-mail adress can be found on his website). The user should
    reference to the academic work of Krister Svanberg when work will be
    published.

    GCMMA-MMA-Python is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the Free
    Software Foundation; either version 3 of the License, or (at your option)
    any later version.

    This program is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
    more details.

    You should have received a copy of the GNU General Public License (file
    LICENSE) along with this file. If not, see http://www.gnu.org/licenses/.


    ############################## References ##################################
    Svanberg, K. (1987). The Method of Moving Asymptotes - A new method for
    structural optimization. International Journal for Numerical Methods in
    Engineering 24, 359-373. doi:10.1002/nme.1620240207

    Svanberg, K. (n.d.). MMA and GCMMA - two methods for nonlinear optimization.
    Retrieved August 3, 2017 from https://people.kth.se/~krille/mmagcmma.pdf


    ########################## Function description ############################
    This function subsolv solves the MMA subproblem:

    minimize SUM[p0j/(uppj-xj) + q0j/(xj-lowj)] + a0*z + SUM[ci*yi + 0.5*di*(yi)^2
                                                  ],

    subject to SUM[pij/(uppj-xj) + qij/(xj-lowj)] - ai*z - yi <= bi,
```

```
        alfaj <=  xj <=  betaj,  yi >= 0,  z >= 0.

    Input:  m, n, low, upp, alfa, beta, p0, q0, P, Q, a0, a, b, c, d.
    Output: xmma,ymma,zmma, slack variables and Lagrange multiplers.
    """

    een = np.ones((n,1))
    eem = np.ones((m,1))
    epsi = 1
    epsvecn = epsi*een
    epsvecm = epsi*eem
    x = 0.5*(alfa+beta)
    y = eem.copy()
    z = np.array([[1.0]])
    lam = eem.copy()
    xsi = een/(x-alfa)
    xsi = np.maximum(xsi,een)
    eta = een/(beta-x)
    eta = np.maximum(eta,een)
    mu = np.maximum(eem,0.5*c)
    zet = np.array([[1.0]])
    s = eem.copy()
    itera = 0
    # Start while epsi>epsimin
    while epsi > epsimin:
        epsvecn = epsi*een
        epsvecm = epsi*eem
        ux1 = upp-x
        xl1 = x-low
        ux2 = ux1*ux1
        xl2 = xl1*xl1
        uxinv1 = een/ux1
        xlinv1 = een/xl1
        plam = p0+np.dot(P.T,lam)
        qlam = q0+np.dot(Q.T,lam)
        gvec = np.dot(P,uxinv1)+np.dot(Q,xlinv1)
        dpsidx = plam/ux2-qlam/xl2
        rex = dpsidx-xsi+eta
        rey = c+d*y-mu-lam
        rez = a0-zet-np.dot(a.T,lam)
        relam = gvec-a*z-y+s-b
        rexsi = xsi*(x-alfa)-epsvecn
        reeta = eta*(beta-x)-epsvecn
        remu = mu*y-epsvecm
        rezet = zet*z-epsi
        res = lam*s-epsvecm
        residu1 = np.concatenate((rex, rey, rez), axis = 0)
        residu2 = np.concatenate((relam, rexsi, reeta, remu, rezet, res), axis = 0)
        residu = np.concatenate((residu1, residu2), axis = 0)
        residunorm = np.sqrt((np.dot(residu.T,residu)).item())
        residumax = np.max(np.abs(residu))
        ittt = 0
        # Start while (residumax>0.9*epsi) and (ittt<200)
        while (residumax > 0.9*epsi) and (ittt < 200):
            ittt = ittt+1
            itera = itera+1
            ux1 = upp-x
            xl1 = x-low
            ux2 = ux1*ux1
            xl2 = xl1*xl1
            ux3 = ux1*ux2
            xl3 = xl1*xl2
            uxinv1 = een/ux1
            xlinv1 = een/xl1
            uxinv2 = een/ux2
            xlinv2 = een/xl2
            plam = p0+np.dot(P.T,lam)
            qlam = q0+np.dot(Q.T,lam)
            gvec = np.dot(P,uxinv1)+np.dot(Q,xlinv1)
            GG = (diags(uxinv2.flatten(),0).dot(P.T)).T-(diags(xlinv2.flatten(),0).
                                                         dot(Q.T)).T
            dpsidx = plam/ux2-qlam/xl2
            delx = dpsidx-epsvecn/(x-alfa)+epsvecn/(beta-x)
            dely = c+d*y-lam-epsvecm/y
```

```
delz = a0-np.dot(a.T,lam)-epsi/z
dellam = gvec-a*z-y-b+epsvecm/lam
diagx = plam/ux3+qlam/xl3
diagx = 2*diagx+xsi/(x-alfa)+eta/(beta-x)
diagxinv = een/diagx
diagy = d+mu/y
diagyinv = eem/diagy
diaglam = s/lam
diaglamyi = diaglam+diagyinv
# Start if m<n
if m < n:
    blam = dellam+dely/diagy-np.dot(GG,(delx/diagx))
    bb = np.concatenate((blam,delz),axis = 0)
    Alam = np.asarray(diags(diaglamyi.flatten(),0) \
        +(diags(diagxinv.flatten(),0).dot(GG.T).T).dot(GG.T))
    AAr1 = np.concatenate((Alam,a),axis = 1)
    AAr2 = np.concatenate((a,-zet/z),axis = 0).T
    AA = np.concatenate((AAr1,AAr2),axis = 0)
    solut = solve(AA,bb)
    dlam = solut[0:m]
    dz = solut[m:m+1]
    dx = -delx/diagx-np.dot(GG.T,dlam)/diagx
else:
    diaglamyiinv = eem/diaglamyi
    dellamyi = dellam+dely/diagy
    Axx = np.asarray(diags(diagx.flatten(),0) \
        +(diags(diaglamyiinv.flatten(),0).dot(GG).T).dot(GG))
    azz = zet/z+np.dot(a.T,(a/diaglamyi))
    axz = np.dot(-GG.T,(a/diaglamyi))
    bx = delx+np.dot(GG.T,(dellamyi/diaglamyi))
    bz = delz-np.dot(a.T,(dellamyi/diaglamyi))
    AAr1 = np.concatenate((Axx,axz),axis = 1)
    AAr2 = np.concatenate((axz.T,azz),axis = 1)
    AA = np.concatenate((AAr1,AAr2),axis = 0)
    bb = np.concatenate((-bx,-bz),axis = 0)
    solut = solve(AA,bb)
    dx = solut[0:n]
    dz = solut[n:n+1]
    dlam = np.dot(GG,dx)/diaglamyi-dz*(a/diaglamyi)+dellamyi/diaglamyi
    # End if m<n
dy = -dely/diagy+dlam/diagy
dxsi = -xsi+epsvecn/(x-alfa)-(xsi*dx)/(x-alfa)
deta = -eta+epsvecn/(beta-x)+(eta*dx)/(beta-x)
dmu = -mu+epsvecm/y-(mu*dy)/y
dzet = -zet+epsi/z-zet*dz/z
ds = -s+epsvecm/lam-(s*dlam)/lam
xx = np.concatenate((y,z,lam,xsi,eta,mu,zet,s),axis = 0)
dxx = np.concatenate((dy,dz,dlam,dxsi,deta,dmu,dzet,ds),axis = 0)
#
stepxx = -1.01*dxx/xx
stmxx = np.max(stepxx)
stepalfa = -1.01*dx/(x-alfa)
stmalfa = np.max(stepalfa)
stepbeta = 1.01*dx/(beta-x)
stmbeta = np.max(stepbeta)
stmalbe = max(stmalfa,stmbeta)
stmalbexx = max(stmalbe,stmxx)
stminv = max(stmalbexx,1.0)
steg = 1.0/stminv
#
xold = x.copy()
yold = y.copy()
zold = z.copy()
lamold = lam.copy()
xsiold = xsi.copy()
etaold = eta.copy()
muold = mu.copy()
zetold = zet.copy()
sold = s.copy()
#
itto = 0
resinew = 2*residunorm
# Start: while (resinew>residunorm) and (itto<50)
while (resinew > residunorm) and (itto < 50):
```

```
                itto = itto+1
                x = xold+steg*dx
                y = yold+steg*dy
                z = zold+steg*dz
                lam = lamold+steg*dlam
                xsi = xsiold+steg*dxsi
                eta = etaold+steg*deta
                mu = muold+steg*dmu
                zet = zetold+steg*dzet
                s = sold+steg*ds
                ux1 = upp-x
                xl1 = x-low
                ux2 = ux1*ux1
                xl2 = xl1*xl1
                uxinv1 = een/ux1
                xlinv1 = een/xl1
                plam = p0+np.dot(P.T,lam)
                qlam = q0+np.dot(Q.T,lam)
                gvec = np.dot(P,uxinv1)+np.dot(Q,xlinv1)
                dpsidx = plam/ux2-qlam/xl2
                rex = dpsidx-xsi+eta
                rey = c+d*y-mu-lam
                rez = a0-zet-np.dot(a.T,lam)
                relam = gvec-np.dot(a,z)-y+s-b
                rexsi = xsi*(x-alfa)-epsvecn
                reeta = eta*(beta-x)-epsvecn
                remu = mu*y-epsvecm
                rezet = np.dot(zet,z)-epsi
                res = lam*s-epsvecm
                residu1 = np.concatenate((rex,rey,rez),axis = 0)
                residu2 = np.concatenate((relam,rexsi,reeta,remu,rezet,res), axis =
                                                                    0)
                residu = np.concatenate((residu1,residu2),axis = 0)
                resinew = np.sqrt(np.dot(residu.T,residu))
                steg = steg/2
                # End: while (resinew>residunorm) and (itto<50)
            residunorm = resinew.copy()
            residumax = max(abs(residu))
            steg = 2*steg
            # End: while (residumax>0.9*epsi) and (ittt<200)
        epsi = 0.1*epsi
        # End: while epsi>epsimin

    xmma = x.copy()
    ymma = y.copy()
    zmma = z.copy()
    lamma = lam
    xsimma = xsi
    etamma = eta
    mumma = mu
    zetmma = zet
    smma = s
    # Return values
    return xmma,ymma,zmma,lamma,xsimma,etamma,mumma,zetmma,smma


def init_scipy_optimizer(
        algorithm, opt_method, editable_xe, xe, xe_min, dp, rmax,
        filter_densities, filter_sensitivities, mat_const_sensitivities,
        target_material_history, model_preparation, data_filter, abaqus_fea,
        adjoint_model, qi, s_max, active_bc, active_loads, iteration,
        set_display, node_coordinates, objh, p_norm_stress_history
    ):
    """ Initialize SciPy optimizer function

    Creates a 'SciPyOptimizer' class if required for the optimizatio selected.
    Otherwise, returns None.

    Inputs:
    -------
    - algorithm (str): name of the SciPy optimization algorithm to be used.
    - opt_method (int): variable defining the optimization method to be used.
    - editable_xe (dict): dictionary with the values of the design densities.
    - xe (dict): dictionary with the densities (design variables) of each
```

```
      relevant element in the model.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.
    - dp (int): number of decimals places to be considered. By definition,
    equal to the number of decimal places in xe_min.
    - rmax (float): search radius that defines the maximum distance between the
      center of the target element and the edge of its neighbouring region.
    - filter_densities (boolean): indicates if the blurring filter should be
      applied to the design densities determined during the optimization
      process.
    - filter_sensitivities (boolean): indicates if the blurring filter should
      be applied to the sensitivities determined during the optimization
      process.
    - mat_const_sensitivities (dict): dictionary with the material constraint
      sensitivity to changes in the design variables.
    - target_material_history (list): list with the values of the material
      constraint that the code tried to acchieve.
    - model_preparation (class): ModelPreparation class.
    - data_filter (class): DataFilter class.
    - abaqus_fea (class): AbaqusFea class.
    - adjoint_model (class): AdjointModel class.
    - qi (float): current value of the exponential of the P-norm stress
      approximation function. Although usually named "P" in the literature,
      the letter "Q" was adopted to avoid confusion with the SIMP penalty
      factor, which is also usually named "P" in the literature.
    - s_max (float): maximum value of the stress constraint imposed.
    - active_bc (dict): dictionary with the data of non-zero boundary
      conditions imposed in the model (such as non-zero displacements).
    - active_loads (list): list with the keys (names) of the loads that are
      active during the simulation (i.e.: non-supressed loads).
    - iteration (int): number of the current iteration in the topology
      optimization process.
    - SetDisplay (class): SetDisplay class.
    - node_coordinates (dict): dictionary with the coordinates of each node.
    - objh (list): list used to record the values of the objective function.
    - p_norm_stress_history (list): list used to record the values of the
      P-norm maximum stress approximation.
    """
    if opt_method in [3, 5, 7]:
        optimizer = SciPyOptimizer(
            algorithm, opt_method, editable_xe, xe, xe_min, dp, rmax,
            filter_densities, filter_sensitivities, mat_const_sensitivities,
            target_material_history, model_preparation, data_filter,
            abaqus_fea, adjoint_model, qi, s_max, active_bc, active_loads,
            iteration, set_display, node_coordinates, objh,
            p_norm_stress_history
        )
    else:
        optimizer = None

    return optimizer


class SciPyOptimizer():
    """ SciPy Optimizer class

    Class responsible for managing the optimization process when using the
    algorithms available in the SciPy module.

    Implementation note:
    --------------------
    The SciPy module has two particular characteristics:
    - the functions that define the optimization problem (objective,
      constraint, and derivative functions) must only take 1 argument as input,
      which should be the design variables.
    - the algorithm selected decides when, and how many times, a given function
      is called.

    Due to these two characteristics, the methods included in this class are
    an alternative version (less efficient) of the functions used by the OC
    and MMA optimization algorithms.

    Attributes:
    -----------
```

```
    - algorithm (str): name of the SciPy optimization algorithm to be used.
    - opt_method (int): variable defining the optimization method to be used.
    - editable_xe (dict): dictionary with the values of the design densities.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.
    - dp (int): number of decimals places to be considered. By definition,
    equal to the number of decimal places in xe_min.
    - rmax (float): search radius that defines the maximum distance between the
      center of the target element and the edge of its neighbouring region.
    - filter_densities (boolean): indicates if the blurring filter should be
      applied to the design densities determined during the optimization
      process.
    - filter_sensitivities (boolean): indicates if the blurring filter should
      be applied to the sensitivities determined during the optimization
      process.
    - mat_const_sensitivities (dict): dictionary with the material constraint
      sensitivity to changes in the design variables.
    - target_material_history (list): list with the values of the material
      constraint that the code tried to acchieve.
    - model_preparation (class): ModelPreparation class.
    - data_filter (class): DataFilter class.
    - abaqus_fea (class): AbaqusFea class.
    - adjoint_model (class): AdjointModel class.
    - qi (float): current value of the exponential of the P-norm stress
      approximation function. Although usually named "P" in the literature,
      the letter "Q" was adopted to avoid confusion with the SIMP penalty
      factor, which is also usually named "P" in the literature.
    - s_max (float): maximum value of the stress constraint imposed.
    - active_bc (dict): dictionary with the data of non-zero boundary
      conditions imposed in the model (such as non-zero displacements).
    - active_loads (list): list with the keys (names) of the loads that are
      active during the simulation (i.e.: non-supressed loads).
    - iteration (int): number of the current iteration in the topology
      optimization process.
    - set_display (class): SetDisplay class.
    - node_coordinates (dict): dictionary with the coordinates of each node.
    - objh: list used to record the values of the objective function.
    - p_norm_stress_history (list): list used to record the values of the
      P-norm maximum stress approximation.
    - current_material (list): list used to record the values of the
      material ratio.

    Methods:
    --------
    - call_solver(editable_xe, xe): prepares and manages the optimization
      process.
    - material_constraint(x): returns the value of the material constraint.
    - material_constraint_der(x): returns the derivative of the material
      constraint.
    - stress_constraint(x): returns the value of the stress constraint.
    - stress_constraint_der(x): returns the derivative of the stress
      constraint.
    - compliance(x): returns the value of the compliance.
    - compliance_der(x): returns the derivative of the compliance.
    - stress(x): returns the value of the maximum Von-Mises stress.
    - stress_der(x): returns the value of the derivative of the maximum
      Von-Mises stress.

    Auxiliary methods:
    ------------------
    - update_attributes(editable_xe, xe, target_material_history, qi,
      iteration): updates the class attributes.
    - return_record(): returns the variables recording the values of the
      objective function, maximum stress, and current iteration.
    """
    def __init__(
            self, algorithm, opt_method, editable_xe, xe, xe_min, dp, rmax,
            filter_densities, filter_sensitivities, mat_const_sensitivities,
            target_material_history, model_preparation, data_filter,
            abaqus_fea, adjoint_model, qi, s_max, active_bc, active_loads,
            iteration, set_display, node_coordinates, objh,
            p_norm_stress_history
```

```
        ):

        self.algorithm = algorithm
        self.opt_method = opt_method
        self.editable_xe = editable_xe
        self.xe = xe
        self.editable_keys = editable_xe.keys()
        self.xe_min = xe_min
        self.dp =  dp
        self.rmax = rmax
        self.filter_densities = filter_densities
        self.filter_sensitivities = filter_sensitivities
        self.mat_const_sensitivities = mat_const_sensitivities
        self.target_material_history = target_material_history
        self.model_preparation = model_preparation
        self.data_filter = data_filter
        self.abaqus_fea = abaqus_fea
        self.adjoint_model = adjoint_model
        self.qi = qi
        self.s_max = s_max
        self.active_bc = active_bc
        self.active_loads = active_loads
        self.iteration = iteration
        self.set_display = set_display
        self.node_coordinates = node_coordinates
        self.objh = objh
        self.p_norm_stress_history = p_norm_stress_history
        self.current_material = []

    def update_attributes(
            self, editable_xe, xe, target_material_history, current_material,
            qi, iteration
        ):
        """ Update attributes method

        Updates the attributes of the 'SciPyOptimizer' class. The scipy
        optimization algorithms require that the functions used as inputs
        only have one input variable, which should be the design variables.
        However, due to the connection with ABAQUS and the need to run
        FEA, it is necessary to provide more inputs other than the design
        variables. This method serves as a means of allowing the the optimizer
        to receive additional information without explicitly including it as
        function inputs.

        Inputs:
        -------
        - editable_xe (dict): dictionary with the densities (design variables)
          of each editable element in the model.
        - xe (dict): dictionary with the densities (design variables) of each
          relevant element in the model.
        - target_material_history (list): list with the values of the material
          constraint that the code tried to acchieve.
        - qi (float): current value of the exponential of the P-norm stress
          approximation function. Although usually named "P" in the literature,
          the letter "Q" was adopted to avoid confusion with the SIMP penalty
          factor, which is also usually named "P" in the literature.
        - iteration (int): number of the current iteration in the topology
          optimization process.
        """
        self.editable_xe = editable_xe
        self.xe = xe
        self.target_material_history = target_material_history
        self.current_material = current_material
        self.qi = qi
        self.iteration = iteration

    def call_solver(self, editable_xe, xe):
        """ Call solver method

        Calls the SciPy optimization algorithm 'SLSQP' or 'trust-constr'.
        This method prepares the objective function, its derivative,
        constraints, and constraint derivatives necessary for the topology
        optimization problem selected.
```

```
        Inputs:
        -------
        - editable_xe (dict): dictionary with the densities (design variables)
          of each editable element in the model.
        - xe (dict): dictionary with the densities (design variables) of each
          relevant element in the model.

        Outputs:
        --------
        - editable_xe (dict): dictionary with the densities (design variables)
          of each editable element in the model.
        - xe (dict): dictionary with the densities (design variables) of each
          relevant element in the model.
        """

        # Selects the constraints, objective function, and required
        # derivatives.
        #
        # For SLSQP:
        if self.algorithm == 'SLSQP':
            if self.opt_method == 3:

                con1 = {'type': 'ineq',
                        'fun': self.material_constraint,
                        'jac': self.material_constraint_der
                }
                constraints = [con1]
                obj_fun = self.compliance
                jacobian = self.compliance_der

            elif self.opt_method == 5:

                con1 = {'type': 'ineq',
                        'fun': self.material_constraint,
                        'jac': self.material_constraint_der
                }
                con2 = {'type': 'ineq',
                        'fun': self.stress_constraint,
                        'jac': self.stress_constraint_der
                }

                constraints = [con1, con2]
                obj_fun = self.compliance
                jacobian = self.compliance_der

            elif self.opt_method == 7:

                con1 = {'type': 'ineq',
                        'fun': self.material_constraint,
                        'jac': self.material_constraint_der
                }
                constraints = [con1]
                obj_fun = self.stress
                jacobian = self.stress_der

            else:
                raise Exception(
                    "Unexpected value for attribute 'opt_method' of class \n"
                    "'SciPyOptimizer' when attribute algorithm == 'SLSQP'."
                    "For this algorithm, 'opt_method' should be 3, 5, or 7."
                )

        # For Trust-constr:
        elif self.algorithm == 'trust-constr':

            total_material = sum(
                [value for key, value in self.mat_const_sensitivities.items()
                 if key in self.editable_keys]
            )

            vol_cnstr_vec = [
                value / total_material
                for key, value in self.mat_const_sensitivities.items()
                if key in self.editable_keys
```

```
        ]

        if self.opt_method == 3:

            linear_constraint = LinearConstraint(
                vol_cnstr_vec,
                -np.inf,
                self.target_material_history[-1],
                keep_feasible = True
            )
            constraints = [linear_constraint]
            obj_fun = self.compliance
            jacobian = self.compliance_der

        elif self.opt_method == 5:

            elmt_stress_constr_sens = self.stress_constraint_der(
                editable_xe.values()
            )

            stress_cnstr_vec = [value for value in elmt_stress_constr_sens]

            linear_constraint = LinearConstraint(
                [vol_cnstr_vec, stress_cnstr_vec],
                [-np.inf, -np.inf],
                [self.target_material_history[-1], 1.0],
                keep_feasible = True
            )
            constraints = [linear_constraint]
            obj_fun = self.compliance
            jacobian = self.compliance_der

        elif self.opt_method == 7:

            linear_constraint = LinearConstraint(
                vol_cnstr_vec,
                -np.inf,
                self.target_material_history[-1],
                keep_feasible = True
            )
            constraints = [linear_constraint]
            obj_fun = self.stress
            jacobian = self.stress_der

        else:
            raise Exception(
                "Unexpected value for attribute 'opt_method' of class \n"
                "'SciPyOptimizer' when attribute algorithm == 'SLSQP'."
                "For this algorithm, 'opt_method' should be 3, 5, or 7."
            )

    else:
        raise Exception(
            "Unexpected value for attribute 'method' of class \n"
            "'SciPyOptimizer'."
        )


    # Defines the allowable range for the design densities [0,1].
    upper_bound = np.ones(len(editable_xe.values()))
    lower_bound = upper_bound * self.xe_min
    bound_class = scipy.optimize.Bounds(
        lower_bound,
        upper_bound,
        keep_feasible = True
    )

    # Calls the SciPy solver.
    solution = scipy.optimize.minimize(
        obj_fun,
        self.editable_xe.values(),
        method = self.algorithm,
        bounds = bound_class,
        jac = jacobian,
```

```
            constraints = constraints
        )

        # Reassign the solution to 'editable_xe' and 'xe', considering the
        # minimum density.
        i = 0
        for key in editable_xe.keys():
            if solution.x[i] <= 0.0:
                editable_xe[key] = self.xe_min
                xe[key] = self.xe_min
            elif solution.x[i] > 1.0:
                editable_xe[key] = 1.0
                xe[key] = 1.0
            else:
                editable_xe[key] = round(solution.x[i], self.dp)
                xe[key] = round(solution.x[i], self.dp)
            i += 1
            xe[key] = editable_xe[key]

        return editable_xe, xe

    def material_constraint(self, x):
        """ Material constraint method

        Returns the current value of the material constraint in the form:

            current constraint = target_material - material_ratio

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - material_constraint (float): value of the material constraint.
        """

        # Determines the current material value.
        gradient = [grad for key, grad in self.mat_const_sensitivities.items()
                    if key in self.editable_keys]
        current_material = sum([rho * grad for rho, grad in zip(x, gradient)])

        # Determines the maximum material value.
        max_material = sum(
            [self.mat_const_sensitivities[i] for i in self.editable_keys]
        )

        # Determines the material fraction and corresponding constraint value.
        material_ratio = current_material / max_material
        material_constraint = float(
            self.target_material_history[-1] - material_ratio
        )

        self.current_material.append(material_ratio)

        return material_constraint

    def material_constraint_der(self, x):
        """ Material constraint derivative

        Returns an array with the constraint derivative for each element,
        normalized material as a function of the maximum material value.

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - material_constr_der (array): array with the normalized material
          constraint derivative.
        """

        # Determines the material constraint derivative vector.
```

```
        # It is defined as negative due to the problem definition used by SciPy
        # for this particular solver.
        material_der = np.array(
            [-self.mat_const_sensitivities[i] for i in self.editable_keys]
        )

        # Determines the maximum material value.
        max_material = sum(
            [self.mat_const_sensitivities[i] for i in self.editable_keys]
        )

        # Normalizes the constraint sensitivity.
        material_constr_der = material_der / max_material

        return material_constr_der

    def stress_constraint(self, x):
        """ Stress constraint method

        Determines the value of the stress constraint for a given list of
        design variables.

        Because SciPy may call this method at any given moment, it is necessary
        to repeat the whole process that leads up to the value of the stress
        constraint, including:
        - updating the material properties.
        - running the FEA.
        - determining the stress vectors on each integration point.
        - determining the stress constraint.

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - stress_constraint (float): value of the stress constraint.
        """

        # Rounds the input design variables into values acceptable by ABAQUS.
        temp_ed_xe = {}
        temp_xe = self.xe.copy()
        i = 0
        for key in self.editable_keys:
            if x[i] > 1.0:
                temp_ed_xe[key] = 1.0
                temp_xe[key] = 1.0
            elif x[i] <= self.xe_min:
                temp_ed_xe[key] = self.xe_min
                temp_xe[key] = self.xe_min
            else:
                temp_ed_xe[key] = max(self.xe_min, round(x[i], self.dp))
                temp_xe[key] = max(self.xe_min, round(x[i], self.dp))
            i+=1

        # Filter input design densities, if requested.
        if self.rmax > 0 and self.filter_densities == True:
            temp_ed_xe, temp_xe = self.data_filter.filter_densities(
                temp_ed_xe,
                temp_xe,
                self.xe_min,
                self.dp
            )

        # Update the material properties.
        self.model_preparation.property_update(temp_ed_xe)

        # Execute the FEA and extract relevant variables.
        (
            _,
            _,
            state_strain,
            node_displacement,
            node_rotation,
```

```
            local_coord_sys
        ) = self.abaqus_fea.run_simulation(self.iteration, temp_xe)

        # Determine the stresses at the integration points.
        self.adjoint_model.determine_stress_and_deformation(
            node_displacement,
            temp_xe,
            node_rotation,
            self.node_coordinates,
            local_coord_sys
        )

        # Determine the p-norm approximation of the maximum Von-Mises
        # stress.
        p_norm_stress = p_norm_approximation(
            self.adjoint_model.stress_vector_int,
            self.adjoint_model.inv_int_p,
            self.qi,
            self.adjoint_model.multiply_VM_matrix,
        )

        self.p_norm_stress_history.append(p_norm_stress)

        # Determine the stress constraint.
        stress_constraint = stress_constraint_evaluation(
            p_norm_stress,
            self.s_max
        )
        stress_constraint = float(stress_constraint[0][0])

        return stress_constraint

    def stress_constraint_der(self, x):
        """ Stress constraint derivative method

        Determines the value of the stress constraint derivativefor a given
        list of design variables.

        Because SciPy may call this method at any given moment, it is necessary
        to repeat the whole process that leads up to the value of the stress
        constraint, including:
        - updating the material properties.
        - running the FEA.
        - run the adjoint model.
        - determining the stress constraint derivative.

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - stress_constr_der_array (array): values of the stress constraint
          derivative for each element.
        """
        # Rounds the input design variables into values acceptable by ABAQUS.
        temp_ed_xe = {}
        temp_xe = self.xe.copy()
        i = 0
        for key in self.editable_keys:
            if x[i] > 1.0:
                temp_ed_xe[key] = 1.0
                temp_xe[key] = 1.0
            elif x[i] <= self.xe_min:
                temp_ed_xe[key] = self.xe_min
                temp_xe[key] = self.xe_min
            else:
                temp_ed_xe[key] = max(self.xe_min, round(x[i], self.dp))
                temp_xe[key] = max(self.xe_min, round(x[i], self.dp))
            i+=1

        # Filter input design densities, if requested.
        if self.rmax > 0 and self.filter_densities == True:
            temp_ed_xe, temp_xe = self.data_filter.filter_densities(
```

```
                temp_ed_xe,
                temp_xe,
                self.xe_min,
                self.dp
            )

        # Update the material properties.
        self.model_preparation.property_update(temp_ed_xe)

        # Execute the FEA and extract relevant variables.
        (
            _,
            _,
            state_strain,
            node_displacement,
            node_rotation,
            local_coord_sys
        ) = self.abaqus_fea.run_simulation(self.iteration, temp_xe)

        # Run adjoint model and extract the adjoint strains.
        adjoint_strain = self.adjoint_model.run_adjoint_simulation(
            node_displacement,
            temp_xe,
            node_rotation,
            self.node_coordinates,
            local_coord_sys,
            self.qi,
            self.active_bc,
            self.active_loads,
            self.iteration,
        )

        # Determine the stress sensitivity.
        elmt_stress_sensitivity = self.adjoint_model.stress_sensitivity(
            temp_xe,
            self.qi,
            state_strain,
            adjoint_strain
        )

        # Filter sensitivity, if requested.
        if self.rmax > 0 and self.filter_sensitivities == True:
            elmt_stress_sensitivity = self.data_filter.filter_function(
                elmt_stress_sensitivity,
                self.editable_keys
            )

        # Reformat sensitivity into an array.
        stress_constr_der_array = np.array(
            [elmt_stress_sensitivity[key] for key in self.editable_keys]
        ).reshape(len(temp_ed_xe))

        stress_constr_der_array = (
            stress_constr_der_array / self.s_max#self.stress_normalization
        )

        return stress_constr_der_array

    def compliance(self, x):
        """ Compliance function

        Determines the compliance of the model for a given set of design
        variables.

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - obj_norm (float): normalized compliance.
        """
        # Rounds the input design variables into values acceptable by ABAQUS.
        temp_ed_xe = {}
```

```
        temp_xe = self.xe.copy()
        i = 0
        for key in self.editable_keys:
            if x[i] > 1.0:
                temp_ed_xe[key] = 1.0
                temp_xe[key] = 1.0
            elif x[i] <= self.xe_min:
                temp_ed_xe[key] = self.xe_min
                temp_xe[key] = self.xe_min
            else:
                temp_ed_xe[key] = max(self.xe_min, round(x[i], self.dp))
                temp_xe[key] = max(self.xe_min, round(x[i], self.dp))
            i+=1

        # Filter input design densities, if requested.
        if self.rmax > 0 and self.filter_densities == True:
            temp_ed_xe, temp_xe = self.data_filter.filter_densities(
                temp_ed_xe,
                temp_xe,
                self.xe_min,
                self.dp
            )

        # Update the material properties and display.
        self.model_preparation.property_update(temp_ed_xe)
        self.set_display.update_display(
            self.qi,
            self.iteration,
            self.adjoint_model,
            self.xe
        )

        # When using the 'trust-constr' algorithm, evaluate the material
        # constraint in order to record its value. Note that this algorithm
        # requires the constraint information to be input in a vector form,
        # which does not allow a more elegant form of recording the data.
        if self.algorithm == 'trust-constr':
            _ = self.material_constraint(temp_ed_xe.values())

        # Execute the FEA and extract relevant variables.
        obj, _, _, _, _, _ = self.abaqus_fea.run_simulation(self.iteration,
                                                            temp_xe)

        self.objh.append(obj)
        save_data(self.qi, self.iteration, temp_ed_xe, temp_xe)
        self.iteration += 1

        norm_obj = float(obj)# / self.objh[-1])

        return norm_obj

    def compliance_der(self, x):
        """ Compliance function

        Determines the compliance sensitivity for a given set of design
        variables.

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - compliance_der_vector (array): array with the normalized compliance
          for each element.
        """
        # Rounds the input design variables into values acceptable by ABAQUS.
        temp_ed_xe = {}
        temp_xe = self.xe.copy()
        i = 0
        for key in self.editable_keys:
            if x[i] > 1.0:
                temp_ed_xe[key] = 1.0
                temp_xe[key] = 1.0
```

```
            elif x[i] <= self.xe_min:
                temp_ed_xe[key] = self.xe_min
                temp_xe[key] = self.xe_min
            else:
                temp_ed_xe[key] = max(self.xe_min, round(x[i], self.dp))
                temp_xe[key] = max(self.xe_min, round(x[i], self.dp))
            i+=1

        # Filter input design densities, if requested.
        if self.rmax > 0 and self.filter_densities == True:
            temp_ed_xe, temp_xe = self.data_filter.filter_densities(
                temp_ed_xe,
                temp_xe,
                self.xe_min,
                self.dp
            )

        # Update the material properties.
        self.model_preparation.property_update(temp_ed_xe)

        # Execute the FEA and extract relevant variables.
        _, ae, _, _, _, _ = self.abaqus_fea.run_simulation(self.iteration,
                                                           temp_xe)

        if self.rmax > 0 and self.filter_sensitivities == True:
            ae = self.data_filter.filter_function(
                ae,
                self.editable_keys
            )

        compliance_der_vector = np.array(
            [ae[key] for key in self.editable_keys]
        ) #/ self.objh[-1]

        return compliance_der_vector

    def stress(self, x):
        """ Stress method

        Determines the value of the p-norm approximation of the maximum
        Von-Mises stress for a given list of design variables.

        Because SciPy may call this method at any given moment, it is necessary
        to repeat the whole process that leads up to the value of the stress
        constraint, including:
        - updating the material properties.
        - running the FEA.
        - determining the stress vectors on each integration point.
        - determining the stress constraint.

        Inputs:
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - max_stress (float): value of the maximum stress approximation.
        """
        # Rounds the input design variables into values acceptable by ABAQUS.
        temp_ed_xe = {}
        temp_xe = self.xe.copy()
        i = 0
        for key in self.editable_keys:
            if x[i] > 1.0:
                temp_ed_xe[key] = 1.0
                temp_xe[key] = 1.0
            elif x[i] <= self.xe_min:
                temp_ed_xe[key] = self.xe_min
                temp_xe[key] = self.xe_min
            else:
                temp_ed_xe[key] = max(self.xe_min, round(x[i], self.dp))
                temp_xe[key] = max(self.xe_min, round(x[i], self.dp))
            i+=1
```

```
        # Filter input design densities, if requested.
        if self.rmax > 0 and self.filter_densities == True:
            temp_ed_xe, temp_xe = self.data_filter.filter_densities(
                temp_ed_xe,
                temp_xe,
                self.xe_min,
                self.dp
            )

        # Update the material properties and display
        self.model_preparation.property_update(temp_ed_xe)
        self.set_display.update_display(
            self.qi,
            self.iteration,
            self.adjoint_model,
            self.xe
        )

        # When using the 'trust-constr' algorithm, evaluate the material
        # constraint in order to record its value. Note that this algorithm
        # requires the constraint information to be input in a vector form,
        # which does not allow a more elegant form of recording the data.
        if self.algorithm == 'trust-constr':
            _ = self.material_constraint(temp_ed_xe.values())

        # Execute the FEA and extract relevant variables.
        (
            _,
            _,
            state_strain,
            node_displacement,
            node_rotation,
            local_coord_sys
        ) = self.abaqus_fea.run_simulation(self.iteration, temp_xe)

        # Determine the stresses at the integration points.
        self.adjoint_model.determine_stress_and_deformation(
            node_displacement,
            temp_xe,
            node_rotation,
            self.node_coordinates,
            local_coord_sys
        )

        # Determine the p-norm approximation of the maximum Von-Mises stress.
        p_norm_stress = p_norm_approximation(
            self.adjoint_model.stress_vector_int,
            self.adjoint_model.inv_int_p,
            self.qi,
            self.adjoint_model.multiply_VM_matrix,
        )

        self.p_norm_stress_history.append(p_norm_stress)
        self.objh = self.p_norm_stress_history
        save_data(self.qi, self.iteration, temp_ed_xe, temp_xe)
        self.iteration += 1

        return p_norm_stress

    def stress_der(self, x):
        """ Stress derivative method

        Determines the value of the stress derivative for a given list of
        design variables.

        Because SciPy may call this method at any given moment, it is necessary
        to repeat the whole process that leads up to the value of the stress
        constraint, including:
        - updating the material properties.
        - running the FEA.
        - run the adjoint model.
        - determining the stress constraint derivative.

        Inputs:
```

```
        -------
        - x (list): list of the design variables.

        Outputs:
        --------
        - stress_der_array (array): values of the stress derivative for each
          element.
        """
        # Rounds the input design variables into values acceptable by ABAQUS.
        temp_ed_xe = {}
        temp_xe = self.xe.copy()
        i = 0
        for key in self.editable_keys:
            if x[i] > 1.0:
                temp_ed_xe[key] = 1.0
                temp_xe[key] = 1.0
            elif x[i] <= self.xe_min:
                temp_ed_xe[key] = self.xe_min
                temp_xe[key] = self.xe_min
            else:
                temp_ed_xe[key] = max(self.xe_min, round(x[i], self.dp))
                temp_xe[key] = max(self.xe_min, round(x[i], self.dp))
            i+=1

        # Filter input design densities, if requested.
        if self.rmax > 0 and self.filter_densities == True:
            temp_ed_xe, temp_xe = self.data_filter.filter_densities(
                temp_ed_xe,
                temp_xe,
                self.xe_min,
                self.dp
            )

        # Update the material properties.
        self.model_preparation.property_update(temp_ed_xe)

        # Execute the FEA and extract relevant variables.
        (
            _,
            _,
            state_strain,
            node_displacement,
            node_rotation,
            local_coord_sys
        ) = self.abaqus_fea.run_simulation(self.iteration, temp_xe)

        # Run adjoint model and extract the adjoint strains.
        adjoint_strain = self.adjoint_model.run_adjoint_simulation(
            node_displacement,
            temp_xe,
            node_rotation,
            self.node_coordinates,
            local_coord_sys,
            self.qi,
            self.active_bc,
            self.active_loads,
            self.iteration,
        )

        # Determine the stress sensitivity.
        elmt_stress_sensitivity = self.adjoint_model.stress_sensitivity(
            temp_xe,
            self.qi,
            state_strain,
            adjoint_strain
        )

        # Filter sensitivity, if requested.
        if self.rmax > 0 and self.filter_sensitivities == True:
            elmt_stress_sensitivity = self.data_filter.filter_function(
                elmt_stress_sensitivity,
                self.editable_keys
            )
```

```
        # Reformat sensitivity into an array.
        stress_constr_der_array = np.array(
            [elmt_stress_sensitivity[key] for key in self.editable_keys]
        ).reshape(len(temp_ed_xe))

        #stress_constr_der_array = (
        #    stress_constr_der_array / self.p_norm_stress_history[-1]
        #)

        return stress_constr_der_array

    def return_record(self):
        """ Return record method

        Returns the records of the objetive function, P-norm stress
        approximation, and iteration number.

        Outputs:
        --------
        - objh (list): record with values of the objective function.
        - p_norm_stress_history (list): list used to record the values of the
          P-norm maximum stress approximation.
        - iteration (int): number of the current iteration.
        """
        data_record = (
            self.objh,
            self.p_norm_stress_history,
            self.current_material,
            self.iteration
        )

        return data_record


#%% Display definition
class SetDisplay():
    """ Set display class

    The present class is responsible for modifying the ABAQUS color codes such
    that it is possible to represent the design variables considered in each
    iteration of the topology optimization process, or the resulting stresses
    installed in each iteration.

    In 2D problems, this representation is acchieved through the use of a
    grey-scale color code, where white represents a design density of 0 and
    black a design density of 1.

    In 3D problems, the same principle applies. However, to allow a less
    obstructed view of some regions, it is possible to hide elements with
    a design density below a given value.

    Attributes:
    -----------
    - mdb (Mdb): ABAQUS model database.
    - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.
    - set_list (list): List of the user-defined (pre-existing) sets.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.
    - dp (int): number of decimals places to be considered in the
      interpolation. By definition, equal to the number of decimal places
      in xe_min.
    - opt_method (int): variable defining the optimization method to be used.
    - plot_density, plot_stress, plot_stress_p, plot_stress_a,
      plot_stress_a_p (boolean): variables indicating which plot should be
      displayed.
    - preferred_plot (int): number of the preferred plot.
    - max_stress_legend (float): defines the maximum stress value of the
      scale used as a legend in the stress plots.

    Methods:
    --------
    - prepare_density_display(): assigns grey-scale colors to the sets that
      contain the elements as a function of the possible design variable values
```

```
    (i.e. sets the color code for the element sets named
    'Rho_'+(density_val)).
  - prepare_stress_display(): assigns a blue-red color scale to 12 element
    sets that sort the elements as a function of their stress state.
    (i.e. sets the color code for the element sets named 'stress_val_0' to
    'stress_val_11').
  - update_display(qi, iteration, AdjointModel, xe): updates the display,
    considering possible changes in the plot options.
  - save_print(name, q, iteration): saves a printscreen of the current
    plot.
  - hide_elements(rho_threshold): hides all elements with a design density
    lower than 'rho_threshold'. This function can help visualise 3D problems.

  Auxiliary methods:
  ------------------
  - rgb_to_hex(rho): determines an RGB code as a function of the design
    variable of an element. The RGB code is then converted into an
    hexadecimal code.
  - plot_elmt_stress(elmt_stress): sorts the elements into sets as a function
    of their stress state.
  - average_element_stress(requested_plot, AdjointModel, xe, qi): determines
    the average stress installed in an element based on the data determined
    for the integration points of each element. Depending on the plot
    requested. The result may be edited to consider, or remove, the influence
    of the stress penalization factor (square root of the design density)
    and/or the exponent of the P-norm stress approximation function.
  """
  def __init__(
          self, mdb, model_name, part_name, set_list, xe_min, dp,
          opt_method, plot_density, plot_stress, plot_stress_p,
          plot_stress_a, plot_stress_a_p, preferred_plot, max_stress_legend
      ):

      self.mdb = mdb
      self.model_name = model_name
      self.part_name = part_name
      self.set_list = set_list
      self.xe_min = xe_min
      self.dp = dp
      self.opt_method = opt_method
      self.part = mdb.models[model_name].parts[part_name]
      self.plot_density = plot_density
      self.plot_stress = plot_stress
      self.plot_stress_p = plot_stress_p
      self.plot_stress_a = plot_stress_a
      self.plot_stress_a_p = plot_stress_a_p
      self.preferred_plot = preferred_plot
      self.max_stress_legend = max_stress_legend

  def prepare_density_display(self):
      """ Prepare density display method

      Method that prepares the ABAQUS interface to display the densities
      of each element.

      The procedure consists on defining a color map scheme (cmap in ABAQUS)
      in which:
      - only the sets 'Rho_'+density_value are visible.
      - each 'Rho_'+density_value set has a different color as a function of
        the density value.
      - neighbouring regions are colored in blue.
      - non-editable regions are colored in red.

      During the definition of the color scheme, the color code updates are
      disabled. The reason is that ABAQUS, by default, will loop through all
      items in the color scheme every time a single item is updated. To avoid
      an exponential number of loops and to significantly reduce the
      computational cost, the updates are disabled during this process and
      only reactivated briefly at the end. The result is a single loop, after
      which the color code updates are disabled again untill necessary.
      """

      nodes = self.part.nodes
      override = {}
```

```
        # Display ABAQUS sets, with the mesh visible, and frozen elements
        # painted red.
        session.viewports['Viewport: 1'].setValues(
                                                displayedObject = self.part)
        session.viewports['Viewport: 1'].partDisplay.setValues(mesh = ON)
        session.viewports['Viewport: 1'].enableMultipleColors()
        session.viewports['Viewport: 1'].setColor(initialColor = '#BD0011')
        cmap = session.viewports['Viewport: 1'].colorMappings['Set']

        # Hide user-defined sets.
        for part_set in self.set_list:
            override[part_set] = (False,)

        # Define the color of each possible design variable.
        # 0 density in white, 1 density in black.
        density_range = np.arange(self.xe_min, 1.0, 10.0 ** (-self.dp))
        for rho in np.round(density_range, self.dp):
            hex_color = self.rgb_to_hex(rho)
            color_info = (True, hex_color, 'Default', hex_color)
            override['Rho_' + str(rho).replace(".",",")] = color_info

        # If rho=1.0, assign the same color as rho=0.99 due to RGB
        # conversion issues.
        hex_color = self.rgb_to_hex(0.99)
        color_info = (True, hex_color, 'Default', hex_color)
        override['Rho_1,0'] = color_info

        # For stress dependent problems, hide the sets created for the
        # adjoint problem.
        if self.opt_method >= 4:
            for i in range(0, len(nodes)):
                override["adjoint_node-"+str(nodes[i].label)] = (False,)
            for i in range(0, 12):
                override['stress_val_' + str(i)] = (False,)

        # neighbouring region is colored in blue.
        color_info = (True, '#177BBD', 'Default', '#177BBD')
        override['neighbouring_region'] = color_info
        cmap.updateOverrides(overrides = override)

        # Update the color scheme once and disable updates untill necessary.
        # Note: updating the color scheme only when necessary will severely
        # increase the code efficiency.
        session.viewports['Viewport: 1'].setColor(colorMapping=cmap)
        session.viewports['Viewport: 1'].enableColorCodeUpdates()
        session.viewports['Viewport: 1'].disableMultipleColors()

    def rgb_to_hex(self, rho):
        """ RGB to Hexadecimal method

        Converts the value of the design variable into an RGB code in
        Hexadecimal, allowing the plot of the density of the element in a
        grey-scale pattern.

        Note that the value of the density is rounded to 2 decimal places, as
        the color codes available in ABAQUS do not allow a more detailed
        discretization.

        Input:
        ------
        - rho (float): float with value of the design variable.

        Output:
        -------
        - hex_code (str): hexadecimal code representing the color to be
          assigned to an element set.
        """
        rho = round(rho, 2)
        rgb = (255*(1-rho), 255*(1-rho), 255*(1-rho))
        a = '%02x%02x%02x' % rgb

        return "#"+a
```

```
    def hide_elements(self, rho_threshold):
        """ Hide elements method

        The standard display will create a grey-scale plot of the element
        densities. However, in 3D problems, this may difficult the corrent
        visualisation of the density distribution. The 'hide elements' function
        will hide all elements with a design density value lower than the
        input value 'rho_threshold'.

        Input:
        ------
        - rho_threshold (float): minimimum value of the design density to be
          displayed in the viewport.
        """
        leaf = dgm.Leaf(leafType = DEFAULT_MODEL)

        session.viewports['Viewport: 1'].partDisplay.displayGroup \
            .replace(leaf = leaf)

        rho = self.xe_min
        inc = 10.0 ** (-self.dp)
        sets_to_hide = []

        while rho < rho_threshold:
            elmt_set = 'Rho_' + str(rho).replace(".",",")
            sets_to_hide.append(elmt_set)
            rho += inc

        leaf = dgm.LeafFromSets(sets = sets_to_hide)
        session.viewports['Viewport: 1'].partDisplay.displayGroup\
            .remove(leaf = leaf)


    def prepare_stress_display(self):
        """ Prepare stress display method

        Method that prepares the ABAQUS interface to display the stress state
        of each element. The color-code applied is the same as the standard
        option included in the ABAQUS 'Visualization' module, with dark-blue
        indicating the lowest stress region, and red the highest stress region.

        The procedure consists on defining a color map scheme (cmap in ABAQUS)
        in which:
        - only the sets 'stress_val_'+[0, 11] are visible.
        - each set has a different color as a function of the stress value.
        - non-editable regions are colored in white.

        During the definition of the color scheme, the color code updates are
        disabled. The reason is that ABAQUS, by default, will loop through all
        items in the color scheme every time a single item is updated. To avoid
        an exponential number of loops and to significantly reduce the
        computational cost, the updates are disabled during this process and
        only reactivated briefly at the end. The result is a single loop, after
        which the color code updates are disabled again untill necessary.
        """

        override = {}
        session.viewports['Viewport: 1'].setValues(displayedObject = self.part)
        session.viewports['Viewport: 1'].partDisplay.setValues(mesh = ON)
        session.viewports['Viewport: 1'].enableMultipleColors()
        # Set initial color to white.
        session.viewports['Viewport: 1'].setColor(initialColor='#FFFFFF')
        cmap=session.viewports['Viewport: 1'].colorMappings['Set']

        # Disable density display.
        inc = 10.0 ** (-self.dp)
        rho_range = np.arange(self.xe_min, 1.0 + inc, inc)
        for rho in np.round(rho_range, self.dp):
            override['Rho_' + str(rho).replace(".",",")] = (False,)

        # ABAQUS standard color-code used in the Visualization module.
        cmap.updateOverrides(overrides={
            'stress_val_11': (True, '#FF0000', 'Default', '#FF0000'),
            'stress_val_10': (True, '#FF7B04', 'Default', '#FF7B04'),
```

```
            'stress_val_9': (True, '#FFB800', 'Default', '#FFB800'),
            'stress_val_8': (True, '#F9FF0F', 'Default', '#F9FF0F'),
            'stress_val_7': (True, '#B2FF00', 'Default', '#B2FF00'),
            'stress_val_6': (True, '#1CE900', 'Default', '#1CE900'),
            'stress_val_5': (True, '#00F94A', 'Default', '#00F94A'),
            'stress_val_4': (True, '#00FF72', 'Default', '#00FF72'),
            'stress_val_3': (True, '#00FFF2', 'Default', '#00FFF2'),
            'stress_val_2': (True, '#00C2D0', 'Default', '#00C2D0'),
            'stress_val_1': (True, '#0A81FF', 'Default', '#0A81FF'),
            'stress_val_0': (True, '#0010FF', 'Default', '#0010FF')})

        # Apply color-code once.
        cmap.updateOverrides(overrides=override)
        session.viewports['Viewport: 1'].setColor(colorMapping = cmap)
        session.viewports['Viewport: 1'].enableColorCodeUpdates()
        session.viewports['Viewport: 1'].disableMultipleColors()

    def plot_elmt_stress(self, elmt_stress):
        """ Plot element stress method

        Sorts the elements into 12 sets ('stress_val_0' to 'stress_val_11')
        as a function of the average element stress installed.
        If a maximum value for the stress scale was not provided, the code
        will consider the maximum equal to the largest stress observed.
        Otherwise, all elements with average stress above the maximum specified
        will be placed in the same set ('stress_val_11').

        Inputs:
        -------
        - elmt_stress (dict): dictionary with the average stress observed in
          each element.
        """

        session.viewports['Viewport: 1'].disableColorCodeUpdates()
        all_elmts = self.part.elements

        if self.max_stress_legend == None:
            max_scale = np.max(elmt_stress.values())
        else:
            max_scale = self.max_stress_legend

        # Loops through all elements, selecting them based on the current
        # upper and lower bound stress values.
        for stress_val in np.arange(0, 12):
            elmt_sec = self.part.elements[0:0]

            lower_bound = max_scale * (stress_val / 11.0)
            upper_bound = max_scale * ((stress_val + 1) / 11.0)

            if stress_val == 0:
                keys = [key for key,value in elmt_stress.items()
                        if value < upper_bound]

            elif stress_val == 11:
                keys = [key for key,value in elmt_stress.items()
                        if lower_bound <= value]

            else:
                keys = [key for key,value in elmt_stress.items()
                        if lower_bound <= value < upper_bound]

            for key in keys:
                elmt_sec += all_elmts[key-1:key]

            set_name = 'stress_val_' + str(stress_val).replace(".", ",")
            self.part.Set(elements = elmt_sec, name = set_name)

        # Update color-code once.
        session.viewports['Viewport: 1'].enableColorCodeUpdates()
        session.viewports['Viewport: 1'].disableMultipleColors()

    def average_element_stress(self, requested_plot, adjoint_model, xe, qi):
        """ Average element stress method
```

```
    Determines the average stress in an element. Depending on the plot
    requested, this may require the inclusion/removal of the stress
    penalization factor (square-root of the design density) and of the
    P-norm stress approximation factor.
    The average data is based on the Von-Mises stresses observed on the
    integration points of each element.

    Inputs:
    -------
    - requested_plot (int): code identifying the plot requested.
    - adjoint_model (class): AdjointModel class, containing the stresses
      and jacobian matrixes observed on each integration point, the volume
      of each element, element thickness, and the 'multiply_VM_matrix' and
      'xe_all' methods.
    - xe (dict): dictionary with the densities (design variables) of each
      relevant element in the model.
    - qi (float): current value of the exponential of the P-norm stress
      approximation function. Although usually named "P" in the literature,
      the letter "Q" was adopted to avoid confusion with the SIMP penalty
      factor, which is also usually named "P" in the literature.

    Output:
    -------
    - elmt_stress (dict): dictionary with the average stress observed in
      each element.
    """
    elmt_stress = {}
    elmt_data = adjoint_model.stress_vector_int
    xe_all = adjoint_model.xe_all
    jacobian_int = adjoint_model.jacobian_int
    thickness = adjoint_model.shell_thickness
    elmt_volume = adjoint_model.elmt_volume
    vm_f = adjoint_model.multiply_VM_matrix


    # For stress dependent plots, the code will determine the average
    # of the stress observed in the integration points of each element.
    # Otherwise, returns None.

    if requested_plot == 1: # Plot density.
        elmt_stress = None

    elif requested_plot == 2: # Plot stress.
        for elmt_label, stress_vectors in elmt_data.items():
            elmt_stress[elmt_label] = 0.0
            vol = elmt_volume[elmt_label]
            stress_amp = math.sqrt(xe_all(elmt_label, xe))

            for int_point, vector in stress_vectors.items():
                det_j = np.linalg.det(jacobian_int[elmt_label][int_point])
                relative_weight = det_j * thickness / vol
                stress_int_p = vector / stress_amp
                stress_int_p = vm_f(stress_int_p, stress_int_p) ** 0.5
                elmt_stress[elmt_label] += stress_int_p * relative_weight

    elif requested_plot == 3: # Plot stress ** qi.
        for elmt_label, stress_vectors in elmt_data.items():
            elmt_stress[elmt_label] = 0.0
            vol = elmt_volume[elmt_label]
            stress_amp = math.sqrt(xe_all(elmt_label, xe))

            for int_point, vector in stress_vectors.items():
                det_j = np.linalg.det(jacobian_int[elmt_label][int_point])
                relative_weight = det_j * thickness / vol
                stress_int_p = (vector / stress_amp)
                stress_int_p = vm_f(stress_int_p, stress_int_p) ** 0.5
                stress_int_p = stress_int_p ** qi
                elmt_stress[elmt_label] += stress_int_p * relative_weight

    elif requested_plot == 4: # Plot amplified stress.
        for elmt_label, stress_vectors in elmt_data.items():
            elmt_stress[elmt_label] = 0.0
            vol = elmt_volume[elmt_label]
            stress_amp = math.sqrt(xe_all(elmt_label, xe))
```

```
                for int_point, vector in stress_vectors.items():
                    det_j = np.linalg.det(jacobian_int[elmt_label][int_point])
                    relative_weight = det_j * thickness / vol
                    vector = vm_f(vector, vector) ** 0.5
                    stress_int_p = vector
                    elmt_stress[elmt_label] += stress_int_p * relative_weight

        elif requested_plot == 5: # Plot amplified stress ** qi.
            for elmt_label, stress_vectors in elmt_data.items():
                elmt_stress[elmt_label] = 0.0
                vol = elmt_volume[elmt_label]
                stress_amp = math.sqrt(xe_all(elmt_label, xe))

                for int_point, vector in stress_vectors.items():
                    det_j = np.linalg.det(jacobian_int[elmt_label][int_point])
                    relative_weight = det_j * thickness / vol
                    vector = vm_f(vector, vector) ** 0.5
                    stress_int_p = vector ** qi
                    elmt_stress[elmt_label] += stress_int_p * relative_weight

        else:
            raise Exception(
                "Unexpected plot request found in 'average_element_stress' \n"
                "method of class 'SetDisplay'.")

        return elmt_stress

    def update_display(self, qi, iteration, adjoint_model, xe):
        """ Update display method

        Loops through the display requests, updates the display, and saves
        a print screen of each plot. The preferred display is printed last,
        since it leads to a larger display period between iterations.

        Inputs:
        -------
        - q (float): value of the exponent used in the p-norm approximation.
        - iteration (int): number of the current iteration.
        - adjoint_model (class): AdjointModel class with the information of
          the stresses determined at the integration points of each element.
          Only used when requesting stress plots. Otherwise, set to None.
        - xe (dict): dictionary with the densities (design variables) of each
          relevant element in the model.
        """

        requested_plots = []
        plot_name = {
            1: 'density',
            2: 'stress',
            3: 'stress_p',
            4: 'stress_a',
            5: 'stress_a_p'
        }

        if self.plot_density == True:
            requested_plots.append(1)
        if self.plot_stress == True:
            requested_plots.append(2)
        if self.plot_stress_p == True:
            requested_plots.append(3)
        if self.plot_stress_a == True:
            requested_plots.append(4)
        if self.plot_stress_a_p == True:
            requested_plots.append(5)

        non_preferred_plots = set(requested_plots) - set([self.preferred_plot])
        requested_plots = list(non_preferred_plots)
        requested_plots.append(self.preferred_plot)

        for request in requested_plots:
            if request == 1:
                self.prepare_density_display()
                self.save_print(plot_name[request], qi, iteration)
```

```
            elif request in [2,3,4,5]:
                self.prepare_stress_display()
                elmt_stress = self.average_element_stress(request,
                                                          adjoint_model,
                                                          xe, qi)
                self.plot_elmt_stress(elmt_stress)
                self.save_print(plot_name[request], qi, iteration)

            else:
                raise Exception(
                    "Unexpected plot request found in method \n"
                    "'update_display' of class 'SetDisplay'."
                )

    def save_print(self, name, q, iteration):
        """ Save Print method

        Saves a .png file with a printscreen of the current plot.
        The name of the file is set equal to 'NAME_Q(Q_value)_I(I_value).png'.

        Inputs:
        -------
        - name (str): name of the file.
        - q (float): value of the exponent used in the p-norm approximation.
        - iteration (int): number of the current iteration.
        """
        path = os.getcwd()
        file_name = path + '\\' + str(name) + '_Q' + str(q) + '_I' + \
            str(iteration) + '.png'
        canvas = session.viewports['Viewport: 1']
        session.printToFile(fileName = file_name, format=PNG,
                            canvasObjects=(canvas, ))


def plot_result(mdb, set_display):
    """ Plot result function

    Creates 3 ABAQUS viewports, each displaying: the final solution, the
    graphic of the objective function, and the graphic of the material
    constraint.

    The elements of the final solution that have a design density lower than
    0.5 are hidden from the display.

    Inputs:
    -------
    - mdb (Mdb): model database from ABAQUS.
    - set_display (class): SetDisplay class.
    """
    n_coords = len(mdb.customData.History['obj'])

    # Display final design.
    vp1 = session.viewports['Viewport: 1']
    p = mdb.models['Model-1'].parts['Part-1']
    vp1.setValues(displayedObject = p)
    set_display.hide_elements(0.5)

    # Plot objective function history.
    vp2 = session.Viewport('Objective history',
                           origin = (89.0, 14.0),
                           width = 89.0,
                           height = 106.0
    )
    obj_plot = session.XYPlot('Objective function')
    graph2 = obj_plot.charts.values()[0]
    obj_data = [(k, mdb.customData.History['obj'][k]) for k in range(n_coords)]
    xy_obj = session.XYData('Objective function', obj_data)
    graph2.setValues(curvesToPlot = [session.Curve(xy_obj)])
    graph2.axes1[0].axisData.setValues(title = 'Iteration')
    graph2.axes2[0].axisData.setValues(title = 'Objective function')
    vp2.setValues(displayedObject = obj_plot)

    # Plot material fraction history.
```

```
    #
    # The number of coordinates is updated, as the optimizers may run a
    # different number of objective and constraint function evaluations.
    n_coords = len(mdb.customData.History['mat'])
    vp3 = session.Viewport('Material history',
                           origin = (0.0, 14.0),
                           width = 89.0,
                           height = 106.0
    )
    mat_plot = session.XYPlot('Material fraction')
    graph1 = mat_plot.charts.values()[0]
    mat_data = [(k, mdb.customData.History['mat'][k]) for k in range(n_coords)]
    xy_mat = session.XYData('Material fraction', mat_data)
    graph1.setValues(curvesToPlot = [session.Curve(xy_mat)])
    graph1.axes1[0].axisData.setValues(title = 'Iteration')
    graph1.axes2[0].axisData.setValues(title = 'Material fraction')
    vp3.setValues(displayedObject = mat_plot)

    # Reposition the first viewport. This is only possible after the other
    # viewports have been created.
    vp1.setValues(
        origin = (178.0, 14.0),
        width = 89.0,
        height = 106.0
    )


#%% Data recording.
def save_data(q, iteration, temp_ed_xe = None, temp_xe = None):
    """ Save Data function

    Creates a .txt file with the values of all relevant variables of the
    current topology optimization iteration.
    The name of the file is set to 'save_file_Q(Q_value)_I(I_value).txt'.

    Inputs:
  -------
    - q (float): value of the exponent used in the p-norm approximation.
    - iteration (int): number of the current iteration.
    - temp_ed_xe (dict): design densities of the editable elements. Only
      introduced by the SciPy optimizers, as they do not output the results
      at the end of every iteration.
    - temp_xe (dict) design densities of all elements. Only introduced by the
      SciPy optimizers, as they do not output the results at the end of every
      iteration.
    """

    # When using the SciPy algorithms, the design densities need to be
    # input directly, as these functions only output the result in the final
    # iteration.
    # The other methods work differently and update the global variables in
    # every iteration. Hence the global assignment.
    if temp_ed_xe != None and temp_xe != None:
        editable_xe = temp_ed_xe
        xe = temp_xe
    else:
        editable_xe = Editable_xe
        xe = Xe

    # Prevent the Low and Upp arrays from being written in a compact form,
    # (ex: low = np.array([1.0,...,1.0])), thus allowing a direct input into
    # the command line.
    if hasattr(Low,'all') == True:
        low = [item[0] for item in Low]
        upp = [item[0] for item in Upp]
        low_text = 'Low = np.array('+str(low)+').reshape('+str(len(Low))+',1)'
        upp_text = 'Upp = np.array('+str(upp)+').reshape('+str(len(Upp))+',1)'

    else:
        low_text = 'Low = '+str(Low)
        upp_text = 'Upp = '+str(Upp)

    # Rewritte the following variables to allow a direct input into the
    # console.
```

```
    Lam_history_str = ""
    for item in Lam_history:
        Lam_history_str +='np.array('+str(item)+'), '

    Fval_history_str = ""
    for item in Fval_history:
        Fval_history_str +='np.array('+str(item)+'), '

    algorithm = "'"+str(ALGORITHM)+"'" if ALGORITHM != None else ALGORITHM

    # Create file and its content. Then write the content, save and close the
    # file.
    tempo = open('save_file_Q'+str(q)+'_I'+str(iteration)+'.txt', 'w+')
    head= "#---Iteration Variables---#"\
    +'\n'+'Xe = '+str(xe)\
    +'\n'+'Editable_xe = '+str(editable_xe)\
    +'\n'+'Xold1 = '+str(Xold1)\
    +'\n'+'Xold2 = '+str(Xold2)\
    +'\n'+'Ae = '+str(Ae)\
    +'\n'+'OAe = '+str(OAe)\
    +'\n'+'OAe2 = '+str(OAe2)\
    +'\n'+'Target_material_history = '+str(Target_material_history)\
    +'\n'+'Current_Material = '+str(Current_Material)\
    +'\n'+'Objh = '+str(Objh)\
    +'\n'+'Fval_history = ['+Fval_history_str+']'\
    +'\n'+'P_norm_history = '+str(P_norm_history)\
    +'\n'+'Lam_history = ['+Lam_history_str+']'\
    +'\n'+low_text\
    +'\n'+upp_text\
    +'\n'+'Change = '+str(Change)\
    +'\n'+'Iter = '+str(Iter)\
    +'\n\n' + "#---User Inputs---#" \
    +'\n'+'CAE_NAME = "'+str(CAE_NAME)+'"'\
    +'\n'+'MODEL_NAME = "'+str(MODEL_NAME)+'"'\
    +'\n'+'PART_NAME = "'+str(PART_NAME)+'"'\
    +'\n'+'MATERIAL_NAME = "'+str(MATERIAL_NAME)+'"'\
    +'\n'+'SECTION_NAME = "'+str(SECTION_NAME)+'"'\
    +'\n'+'MESH_UNIFORMITY = '+str(MESH_UNIFORMITY)\
    +'\n'+'N_DOMAINS = '+str(N_DOMAINS)\
    +'\n'+'N_CPUS = '+str(N_CPUS)\
    +'\n'+'LAST_FRAME = '+str(LAST_FRAME)\
    +'\n'+'MATERIAL_CONSTRAINT = '+str(MATERIAL_CONSTRAINT)\
    +'\n'+'OPT_METHOD = '+str(OPT_METHOD)\
    +'\n'+'NONLINEARITIES = '+str(NONLINEARITIES)\
    +'\n'+'TARGET_MATERIAL = '+str(TARGET_MATERIAL)\
    +'\n'+'EVOL_RATIO = '+str(EVOL_RATIO)\
    +'\n'+'XE_MIN = '+str(XE_MIN)\
    +'\n'+'DP = '+str(DP)\
    +'\n'+'RMAX = '+str(RMAX)\
    +'\n'+'FILTER_SENSITIVITIES = '+str(FILTER_SENSITIVITIES)\
    +'\n'+'FILTER_DENSITIES = '+str(FILTER_DENSITIES)\
    +'\n'+'P = '+str(P)\
    +'\n'+'INITIAL_DENSITY = '+str(INITIAL_DENSITY)\
    +'\n'+'MOVE_LIMIT = '+str(MOVE_LIMIT)\
    +'\n'+'CONSIDER_FROZEN_REGION = '+str(CONSIDER_FROZEN_REGION)\
    +'\n'+'CONSIDER_NEIGHBOURING_REGION = ' +str(CONSIDER_NEIGHBOURING_REGION)\
    +'\n'+'S_MAX = '+str(S_MAX)\
    +'\n'+'Qi = '+str(Qi)\
    +'\n'+'QF = '+str(QF)\
    +'\n'+'P_norm_stress = '+str(P_norm_stress)\
    +'\n'+'Stress_sensitivity = '+str(Stress_sensitivity)\
    +'\n'+'PLOT_DENSITY = '+str(PLOT_DENSITY)\
    +'\n'+'PLOT_STRESS = '+str(PLOT_STRESS)\
    +'\n'+'PLOT_STRESS_P = '+str(PLOT_STRESS_P)\
    +'\n'+'PLOT_STRESS_A = '+str(PLOT_STRESS_A)\
    +'\n'+'PLOT_STRESS_A_P = '+str(PLOT_STRESS_A_P)\
    +'\n'+'PREFERRED_PLOT = '+str(PREFERRED_PLOT)\
    +'\n'+'MAX_STRESS_LEGEND = '+str(MAX_STRESS_LEGEND)\
    +'\n'+'ALGORITHM = '+str(algorithm)\
    +'\n\n'+"#---RESTART Inputs---#"\
    +'\n'+"RESTART = True"\
    +'\n'+"Mdb = openMdb(CAE_NAME)"
    tempo.write(head)
    tempo.close()
```

```
def save_mdb(mdb, current_material, objh, cae_name):
    """ Save Mdb function

    Saves the ABAQUS Mdb in a new CAE file, containing two additional custom
    data inputs: one for the material history, and another for the objective
    function history.

    Inputs:
    -------
    - mdb (Mdb): model database from ABAQUS.
    - current_material (list): list with the current value of the material
      constraint.
    - objh (list): record with values of the objective function.
    - cae_name (str): string with the name of the ABAQUS CAE file.
    """
    mdb.customData.History = {'mat':current_material, 'obj':objh}
    mdb.saveAs("TopOpt-"+cae_name)


#%% Element formulation and C matrix (stiffness matrix)
class ElementFormulation():
    """ Element formulation class

    This class contains finite element information that is dependent on the
    element type used during the simulation, such as the B matrixes
    (strain-displacement matrix), the Jacobian matrix, and the shape function
    or their derivatives.
    The information contained in this class is used in stress dependent
    topology optimization problems, in order to determine the derivative of
    the maximum stress as a function of changes in the design variables.

    Attribute:
    ----------
    - element_type (str): ABAQUS code defining the element type.

    Methods:
    --------
    - b_matrix_and_jac(s, t, v, x_coord, y_coord, z_coord, v1_vector,
      v2_vector, vn, a_rot, b_rot, shell_thickness): determines the B and
      Jacobian matrixes.
    - b_matrix(s, t, v, jacobian, v1_vector = None, v2_vector = None,
      shell_thickness = None): determines the B matrix of an element.
    - jacobian_matrix(s, t, v, x_coord, y_coord, z_coord, vn = None,
      a_rot = None, b_rot = None, shell_thickness = None): determines the
      Jacobian matrix of an element.

    Auxiliary methods:
    ------------------
    - b_matrix_2DQ4(s, t, jacobian): determines the B matrix of a 2DQ4 element.
    - b_matrix_S4(s, t, v, jacobian, v1_vector, v2_vector, shell_thickness):
      determines the B matrix of an S4 element.
    - b_matrix_C3D8(s, t, v, jacobian): determines the B matrix of a C3D8
      element.
    - jacobian_2DQ4(s, t, x_coord, y_coord): determines the Jacobian matrix of
      a 2DQ4 element.
    - jacobian_S4(s, t, v, x_coord, y_coord, z_coord, vn, a_rot, b_rot,
      shell_thickness): determines the Jacobian matrix of an S4 element.
    - jacobian_C3D8(s, t, v, x_coord, y_coord, z_coord): determines the
      Jacobian matrix of a C3D8 element.
    - shape_eq_2DQ4(i, s, t): shape equation of a 2DQ4 element.
    - dN_ds_2DQ4(i, s, t), dN_dt_2DQ4(i, s, t): derivative of the shape
      quations of a 2DQ4 element.
    - dN_ds_C3D8(i, s, t, v), dN_dt_C3D8(i, s, t, v), dN_dv_C3D8(i, s, t, v):
      dertivatives of the shape equation of a C3D8 element.
    - local_node_coordinates(): creates three dictionaries with the local
      coordinates of the element nodes.
    - local_int_point_coordinates(): creates three dictionaries with the local
      coordinates of the integration points of an element.
    """
    def __init__(self, element_type):
        self.element_type = element_type
```

```
    def b_matrix_and_jac(
            self, s, t, v, x_coord, y_coord, z_coord, v1_vector, v2_vector,
            vn, a_rot, b_rot, shell_thickness
        ):
        """ B Matrix and Jacobian method
        Returns the B and Jacobian matrixes of an element, in a given local
        poimt.

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - x_coord, y_coord, z_coord (lists): lists with the node coordinates,
          following the node labelling sequence set by ABAQUS.
        - v1_vector, v2_vector, vn (arrays): Only used for S4 elements.
          Vectors indicating the in-plane directions of the node local
          coordinate system (as illustrated in the book Finite Element
          Procedures, 2nd edition, written by Klaus-Jurgen Bathe, in section
          5.4, page 437, figure 5.33).
        - a_rot, b_rot (lists): lists with the node rotations of a given
          element, following the node labelling sequence set by ABAQUS.
        - shell_thickness (float): Only used for S4 elements. Total thickness
          of the shell element.

        Outputs:
        --------
        - b_matrix (array): strain-displacement matrix.
        - jacobian (array): jacobian matrix.
        """
        jacobian = self.jacobian_matrix(s, t, v, x_coord, y_coord, z_coord,
                                        vn, a_rot, b_rot, shell_thickness)

        b_matrix = self.b_matrix(s, t, v, jacobian, v1_vector, v2_vector,
                                 shell_thickness)

        return b_matrix, jacobian

    def b_matrix(
            self, s, t, v, jacobian, v1_vector = None, v2_vector = None,
            shell_thickness = None
        ):
        """ B Matrix method
        Outputs the B Matrix, establishing the relation between the
        displacement and deformation of an element. This method checks the
        type of element being used and returns a matrix with the suitable
        form.

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - jacobian (array): jacobian matrix of the element.
        - v1_vector, v2_vector (arrays): Only used for S4 elements.
          Vectors indicating the in-plane directions of the node local
          coordinate system (as illustrated in the book Finite Element
          Procedures, 2nd edition, written by Klaus-Jurgen Bathe, in section
          5.4, page 437, figure 5.33).
        - shell_thickness (float): Only used for S4 elements. Total thickness
          of the shell element.

        Output:
        -------
        - b_matrix (array): displacement-strain matrix of an element.
        """
        if self.element_type in ["CPS4", "CPE4"]:
            b_matrix = self.b_matrix_2DQ4(s, t, jacobian)

        elif self.element_type == "S4":
            b_matrix = self.b_matrix_S4(s, t, v, jacobian, v1_vector,
                                        v2_vector, shell_thickness)

        elif self.element_type == "C3D8":
            b_matrix = self.b_matrix_C3D8(s, t, v, jacobian)
```

```
        else:
            raise Exception(
                'Unsuported element type encountered in the "b_matrix" \n'
                'method.'
            )

        return b_matrix

    def b_matrix_2DQ4(self, s, t, jacobian):
        """ B Matrix 2DQ4 method
        Creates the B matrix, establishing the relation between the element
        displacement and deformation, for an ABAQUS 2D element with 4 nodes
        (2DQ4).

        Inputs:
        -------
        - s, t (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - jacobian (array): jacobian matrix of the element.

        Output:
        -------
        - b_matrix (array): displacement-strain matrix of a 2DQ4 element.
        """
        b_matrix = 0
        inv_jacobian = inv(jacobian)

        # Determines the matrix that represents the contribution of each node
        # displacement to the strain. Then stacks this matrix to form the
        # element B matrix.
        for i in range(0,4):

            der_shape_functions = np.array([[self.dN_ds_2DQ4(i, s, t)],
                                            [self.dN_dt_2DQ4(i, s, t)]])

            dN_vector = np.dot(inv_jacobian, der_shape_functions)

            b_i = np.array([[dN_vector[0][0], 0],
                            [0, dN_vector[1][0]],
                            [dN_vector[1][0], dN_vector[0][0]]])

            if hasattr(b_matrix,"shape"):
                b_matrix = np.hstack((b_matrix, b_i))
            else:
                b_matrix = b_i

        return b_matrix

    def b_matrix_S4(
            self, s, t, v, jacobian, v1_vector, v2_vector, shell_thickness
        ):
        """ B Matrix S4 method
        Creates the B matrix, establishing the relation between the element
        displacement and deformation, for an ABAQUS shell element with 4 nodes
        (S4).

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - jacobian (array): jacobian matrix of the element.
        - v1_vector, v2_vector (arrays): vectors indicating the in-plane
          directions of the node local coordinate system (as illustrated in
          the book Finite Element Procedures, 2nd edition, written
          by Klaus-Jurgen Bathe, in section 5.4, page 437, figure 5.33).
        - shell_thickness (float): total thickness of the shell element.

        Output:
        -------
        - b_matrix (array): displacement-strain matrix of an S4 element.
        """
        b_matrix = 0
        inv_jacobian = inv(jacobian)
```

```
        # Determines the matrix that represents the contribution of each node
        # displacement to the strain. Then stacks this matrix to form the
        # element B matrix.
        for i in range(0,4):

            g1k = -0.5 * shell_thickness * v2_vector[i]
            g2k = 0.5 * shell_thickness * v1_vector[i]

            du_11 = self.dN_ds_2DQ4(i, s, t)
            du_12 = self.dN_ds_2DQ4(i, s, t) * g1k[0] * v
            du_13 = self.dN_ds_2DQ4(i, s, t) * g2k[0] * v
            du_21 = self.dN_dt_2DQ4(i, s, t)
            du_22 = self.dN_dt_2DQ4(i, s, t) * g1k[0] * v
            du_23 = self.dN_dt_2DQ4(i, s, t) * g2k[0] * v
            du_31 = 0.0
            du_32 = self.shape_eq_2DQ4(i, s, t) * g1k[0]
            du_33 = self.shape_eq_2DQ4(i, s, t) * g2k[0]

            du_dstv = np.array([np.array([du_11, du_12, du_13]),
                                np.array([du_21, du_22, du_23]),
                                np.array([du_31, du_32, du_33])])

            dv_11 = self.dN_ds_2DQ4(i, s, t)
            dv_12 = self.dN_ds_2DQ4(i, s, t) * g1k[1] * v
            dv_13 = self.dN_ds_2DQ4(i, s, t) * g2k[1] * v
            dv_21 = self.dN_dt_2DQ4(i, s, t)
            dv_22 = self.dN_dt_2DQ4(i, s, t) * g1k[1] * v
            dv_23 = self.dN_dt_2DQ4(i, s, t) * g2k[1] * v
            dv_31 = 0.0
            dv_32 = self.shape_eq_2DQ4(i, s, t) * g1k[1]
            dv_33 = self.shape_eq_2DQ4(i, s, t) * g2k[1]

            dv_dstv = np.array([np.array([dv_11, dv_12, dv_13]),
                                np.array([dv_21, dv_22, dv_23]),
                                np.array([dv_31, dv_32, dv_33])])

            dw_11 = self.dN_ds_2DQ4(i, s, t)
            dw_12 = self.dN_ds_2DQ4(i, s, t) * g1k[2] * v
            dw_13 = self.dN_ds_2DQ4(i, s, t) * g2k[2] * v
            dw_21 = self.dN_dt_2DQ4(i, s, t)
            dw_22 = self.dN_dt_2DQ4(i, s, t) * g1k[2] * v
            dw_23 = self.dN_dt_2DQ4(i, s, t) * g2k[2] * v
            dw_31 = 0.0
            dw_32 = self.shape_eq_2DQ4(i, s, t) * g1k[2]
            dw_33 = self.shape_eq_2DQ4(i, s, t) * g2k[2]

            dw_dstv = np.array([np.array([dw_11, dw_12, dw_13]),
                                np.array([dw_21, dw_22, dw_23]),
                                np.array([dw_31, dw_32, dw_33])])


            du_dxyz = np.dot(inv_jacobian, du_dstv)
            dv_dxyz = np.dot(inv_jacobian, dv_dstv)
            dw_dxyz = np.dot(inv_jacobian, dw_dstv)

            b_line_1 = [du_dxyz[0][0], 0, 0, du_dxyz[0][1], du_dxyz[0][2]]
            b_line_2 = [0, dv_dxyz[1][0], 0, dv_dxyz[1][1], dv_dxyz[1][2]]
            b_line_3 = [0, 0, dw_dxyz[2][0], dw_dxyz[2][1], dw_dxyz[2][2]]
            b_line_4 = [du_dxyz[1][0], dv_dxyz[0][0], 0,
                du_dxyz[1][1] + dv_dxyz[0][1], du_dxyz[1][2] + dv_dxyz[0][2]]
            b_line_5 = [du_dxyz[2][0], 0, dw_dxyz[0][0],
                du_dxyz[2][1] + dw_dxyz[0][1], du_dxyz[2][2] + dw_dxyz[0][2]]
            b_line_6 = [0, dv_dxyz[2][0], dw_dxyz[1][0],
                dv_dxyz[2][1] + dw_dxyz[1][1], dv_dxyz[2][2] + dw_dxyz[1][2]]

            b_i = np.array([b_line_1,
                            b_line_2,
                            b_line_3,
                            b_line_4,
                            b_line_5,
                            b_line_6])

            if hasattr(b_matrix,"shape"):
                b_matrix = np.hstack((b_matrix, b_i))
```

```
            else:
                b_matrix = b_i

        return b_matrix

    def b_matrix_C3D8(self, s, t, v, jacobian):
        """ B Matrix C3D8 method
        Creates the B matrix, establishing the relation between the element
        displacement and deformation, for an ABAQUS 3D element with 8 nodes
        (C3D8).

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - jacobian (array): jacobian matrix of the element.

        Output:
        -------
        - b_matrix (array): displacement-strain matrix of a C3D8 element.
        """
        b_matrix = 0
        inv_jacobian = inv(jacobian)

        # Determines the matrix that represents the contribution of each node
        # displacement to the strain. Then stacks this matrix to form the
        # element B matrix.
        for i in range(0,8):
            der_shape_functions = np.array([[self.dN_ds_C3D8(i, s, t, v)],
                                            [self.dN_dt_C3D8(i, s, t, v)],
                                            [self.dN_dv_C3D8(i, s, t, v)]])

            dN_vector = np.dot(inv_jacobian, der_shape_functions)

            b_i = np.array([[dN_vector[0][0], 0, 0],
                            [0, dN_vector[1][0], 0],
                            [0, 0, dN_vector[2][0]],
                            [dN_vector[1][0], dN_vector[0][0], 0],
                            [dN_vector[2][0], 0, dN_vector[0][0]],
                            [0, dN_vector[2][0], dN_vector[1][0]]])

            if hasattr(b_matrix,"shape"):
                b_matrix = np.hstack((b_matrix,b_i))
            else:
                b_matrix = b_i

        return b_matrix

    def jacobian_matrix(
            self, s, t, v, x_coord, y_coord, z_coord, vn = None, a_rot = None,
            b_rot = None, shell_thickness = None
        ):
        """ Jacobian matrix method

        Determines the Jacobian matrix of an element, in a given local point,
        as a function of the element type.

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - x_coord, y_coord, z_coord (lists): lists with the node coordinates,
          following the node labelling sequence set by ABAQUS.
        - vn (array): Only used for S4 elements. Vector indicating the normal
          direction of the shell surface (as illustrated in the book Finite
          Element Procedures, 2nd edition, written by Klaus-Jurgen Bathe,
          in section 5.4, page 437, figure 5.33).
        - a_rot, b_rot (lists): lists with the node rotations of a given
          element, following the node labelling sequence set by ABAQUS.
        - shell_thickness (float): Only used for S4 elements. Total thickness
          of the shell element.

        Output:
        -------
```

```
        - jacobian (array): Jacobian matrix.
    """
        if self.element_type in ["CPS4", "CPE4"]:
            jacobian = self.jacobian_2DQ4(s, t, x_coord, y_coord)

        elif self.element_type == "S4":
            jacobian = self.jacobian_S4(s, t, v, x_coord, y_coord, z_coord, vn,
                                        a_rot, b_rot, shell_thickness)

        elif self.element_type == "C3D8":
            jacobian = self.jacobian_C3D8(s, t, v, x_coord, y_coord, z_coord)

        else:
            raise Exception('Unsuported element type encountered in the '
                            '"jacobian_matrix" method.')

        return jacobian

    def jacobian_2DQ4(self, s, t, x_coord, y_coord):
        """ Jacobian 2DQ4 method

        Determines the Jacobian matrix of a 2DQ4 element, in a given local
        point, as a function of the element type.

        Inputs:
        -------
        - s, t (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - x_coord, y_coord (lists): lists with the node coordinates,
          following the node labelling sequence set by ABAQUS.

        Output:
        -------
        - jacobian (array): Jacobian matrix.
        """
        j11 = sum([self.dN_ds_2DQ4(i, s, t) * x_coord[i] for i in range(0,4)])
        j12 = sum([self.dN_ds_2DQ4(i, s, t) * y_coord[i] for i in range(0,4)])
        j21 = sum([self.dN_dt_2DQ4(i, s, t) * x_coord[i] for i in range(0,4)])
        j22 = sum([self.dN_dt_2DQ4(i, s, t) * y_coord[i] for i in range(0,4)])

        jacobian = np.array([[j11, j12],
                             [j21, j22]])

        return jacobian

    def jacobian_S4(
            self, s, t, v, x_coord, y_coord, z_coord, vn, a_rot, b_rot,
            shell_thickness
        ):
        """ Jacobian S4 method

        Determines the Jacobian matrix of an S4 element, in a given local
        point, as a function of the element type.

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - x_coord, y_coord, z_coord (lists): lists with the node coordinates,
          following the node labelling sequence set by ABAQUS.
        - vn (array): Only used for S4 elements. Vector indicating the normal
          direction of the shell surface (as illustrated in the book Finite
          Element Procedures, 2nd edition, written by Klaus-Jurgen Bathe,
          in section 5.4, page 437, figure 5.33).
        - a_rot, b_rot (lists): lists with the node rotations of a given
          element, following the node labelling sequence set by ABAQUS.
        - shell_thickness (float): Only used for S4 elements. Total thickness
          of the shell element.

        Output:
        -------
        - jacobian (array): Jacobian matrix.
        """
        j11 = sum(
```

```
            [self.dN_ds_2DQ4(i, s, t) * x_coord[i] + self.dN_ds_2DQ4(i, s, t)
            * v * 0.5 * shell_thickness * (vn[i][0])
            for i in range(0,4)])

        j12 = sum(
            [self.dN_ds_2DQ4(i, s, t) * y_coord[i] + self.dN_ds_2DQ4(i, s, t)
            * v * 0.5 * shell_thickness * (vn[i][1])
            for i in range(0,4)])

        j13 = sum(
            [self.dN_ds_2DQ4(i, s, t) * z_coord[i] + self.dN_ds_2DQ4(i, s, t)
            * v * 0.5 * shell_thickness * (vn[i][2])
            for i in range(0,4)])

        j21 = sum(
            [self.dN_dt_2DQ4(i, s, t) * x_coord[i] + self.dN_dt_2DQ4(i, s, t)
            * v * 0.5 * shell_thickness * (vn[i][0])
            for i in range(0,4)])

        j22 = sum(
            [self.dN_dt_2DQ4(i, s, t) * y_coord[i] + self.dN_dt_2DQ4(i, s, t)
            * v * 0.5 * shell_thickness * (vn[i][1])
            for i in range(0,4)])

        j23 = sum(
            [self.dN_dt_2DQ4(i, s, t) * z_coord[i] + self.dN_dt_2DQ4(i, s, t)
            * v * 0.5 * shell_thickness * (vn[i][2])
            for i in range(0,4)])

        j31 = sum(
            [0.5 * shell_thickness * self.shape_eq_2DQ4(i, s, t) * (vn[i][0])
            for i in range(0,4)])

        j32 = sum(
            [0.5 * shell_thickness * self.shape_eq_2DQ4(i, s, t) * (vn[i][1])
            for i in range(0,4)])

        j33 = sum(
            [0.5 * shell_thickness * self.shape_eq_2DQ4(i, s, t) * (vn[i][2])
            for i in range(0,4)])

        jacobian = np.array([[j11, j12, j13],
                             [j21, j22, j23],
                             [j31, j32, j33]])

        return jacobian

    def jacobian_C3D8(self, s, t, v, x_coord, y_coord, z_coord):
        """ Jacobian C3D8 method

        Determines the Jacobian matrix of a C3D8 element, in a given local
        point, as a function of the element type.

        Inputs:
        -------
        - s, t, v (floats): coordinates indicating where the B matrix should be
          determined, in the element local coordinate system.
        - x_coord, y_coord, z_coord (lists): lists with the node coordinates,
          following the node labelling sequence set by ABAQUS.

        Output:
        -------
        - jacobian (array): Jacobian matrix.
        """
        n_nodes = range(0,8)
        j11 = sum([self.dN_ds_C3D8(i, s, t, v) * x_coord[i] for i in n_nodes])
        j12 = sum([self.dN_ds_C3D8(i, s, t, v) * y_coord[i] for i in n_nodes])
        j13 = sum([self.dN_ds_C3D8(i, s, t, v) * z_coord[i] for i in n_nodes])
        j21 = sum([self.dN_dt_C3D8(i, s, t, v) * x_coord[i] for i in n_nodes])
        j22 = sum([self.dN_dt_C3D8(i, s, t, v) * y_coord[i] for i in n_nodes])
        j23 = sum([self.dN_dt_C3D8(i, s, t, v) * z_coord[i] for i in n_nodes])
        j31 = sum([self.dN_dv_C3D8(i, s, t, v) * x_coord[i] for i in n_nodes])
        j32 = sum([self.dN_dv_C3D8(i, s, t, v) * y_coord[i] for i in n_nodes])
        j33 = sum([self.dN_dv_C3D8(i, s, t, v) * z_coord[i] for i in n_nodes])
```

```
        jacobian = np.array([[j11, j12, j13],
                             [j21, j22, j23],
                             [j31, j32, j33]])

        return jacobian

    def shape_eq_2DQ4(self, i, s, t):
        """ 2DQ4 Shape equation method

        Determines the value of the shape function of a 2DQ4 in a given local
        point.

        Inputs:
        -------
        - i (int): node number.
        - s, t (dicts): dictionaries with the local coordinates of each
          node.

        Output:
        -------
        - shape_value (float): value of the shape function.
        """
        if i == 0:
            shape_value = 0.25 * (1 - s) * (1 - t)
        elif i == 1:
            shape_value = 0.25 * (1 + s) * (1 - t)
        elif i == 2:
            shape_value = 0.25 * (1 + s) * (1 + t)
        elif i == 3:
            shape_value = 0.25 * (1 - s) * (1 + t)
        else:
            raise Exception(
                "Unexpected value 'i' in method 'shape_eq_2DQ4'. \n"
                "Variable i should be equal to 0, 1, 2 or 3."
            )

        return shape_value

    def dN_ds_2DQ4(self, i, s, t):
        """ 2DQ4 Shape function derivative method (w.r.t. s)

        Outputs the derivative of the shape function (for 2D or shell elements)
        with respect to the local axis (variable) s.

        - Inputs:
        ---------
        - i (int): number of the node whose shape function derivative is being
          determined.
        - s, t (floats): local coordinates of where the derivative should be
          determined.

        - Output:
        ---------
        - dN_ds (float): derivative of the shape function w.r.t. the s local
          axis (variable).
        """
        if i == 0:
            dN_ds_2DQ4 = (t - 1)
        elif i == 1:
            dN_ds_2DQ4 = (1 - t)
        elif i == 2:
            dN_ds_2DQ4 = (1 + t)
        elif i == 3:
            dN_ds_2DQ4 = (-1 - t)
        else:
            raise Exception("Unexpected shape function index 'i' in method "
                            "dN_ds_2DQ4.")

        return 0.25 * dN_ds_2DQ4

    def dN_dt_2DQ4(self, i, s, t):
        """ 2DQ4 Shape function derivative method (w.r.t. t)
```

```
        Outputs the derivative of the shape function (for 2D or shell elements)
        with respect to the local axis (variable) t.

        - Inputs:
        ---------
        - i (int): number of the node whose shape function derivative is being
          determined.
        - s, t (floats): local coordinates of where the derivative should be
          determined.

        - Output:
        ---------
        - dN_dt (float): derivative of the shape function w.r.t. the t local
          axis (variable).
        """
        if i == 0:
            dN_dt_2DQ4 = (s - 1)
        elif i == 1:
            dN_dt_2DQ4 = (-s - 1)
        elif i == 2:
            dN_dt_2DQ4 = (1 + s)
        elif i == 3:
            dN_dt_2DQ4 = (1 - s)
        else:
            raise Exception("Unexpected shape function index 'i' in method "
                            "dN_dt_2DQ4.")

        return 0.25*dN_dt_2DQ4

    def dN_ds_C3D8(self, i, s, t, v):
        """ C3D8 Shape function derivative method (w.r.t. s)

        Outputs the derivative of the shape function (for 3D elements) with
        respect to the local axis (variable) s.

        - Inputs:
        ---------
        - i (int): number of the node whose shape function derivative is being
          determined.
        - s, t, v (floats): local coordinates of where the derivative should be
          determined.

        - Output:
        ---------
        - dN_ds (float): derivative of the shape function w.r.t. the s local
          axis (variable).
        """
        if i == 0:
            dN_ds_C3D8 = -0.125 * (-t + 1) * (-v + 1)
        elif i == 1:
            dN_ds_C3D8 = 0.125 * (-t + 1) * (-v + 1)
        elif i == 2:
            dN_ds_C3D8 = 0.125 * (t + 1) * (-v + 1)
        elif i == 3:
            dN_ds_C3D8 = -0.125 * (t + 1) * (-v + 1)
        elif i == 4:
            dN_ds_C3D8 = -0.125 * (-t + 1) * (v + 1)
        elif i == 5:
            dN_ds_C3D8 = 0.125 * (-t + 1) * (v + 1)
        elif i == 6:
            dN_ds_C3D8 = 0.125 * (t + 1) * (v + 1)
        elif i == 7:
            dN_ds_C3D8 = -0.125 * (t + 1) * (v + 1)
        else:
            raise Exception("Unexpected shape function index 'i' in method "
                            "dN_ds_C3D8.")

        return dN_ds_C3D8

    def dN_dt_C3D8(self, i, s, t, v):
        """ C3D8 Shape function derivative method (w.r.t. t)

        Outputs the derivative of the shape function (for 3D elements) with
        respect to the local axis (variable) t.
```

```
        - Inputs:
        ---------
        - i (int): number of the node whose shape function derivative is being
          determined.
        - s, t, v (floats): local coordinates of where the derivative should be
          determined.

        - Output:
        ---------
        - dN_dt (float): derivative of the shape function w.r.t. the t local
          axis (variable).
        """
        if i == 0:
            dN_dt_C3D8 = -(-0.125 * s + 0.125)*(-v + 1)
        elif i == 1:
            dN_dt_C3D8 = -(0.125 * s + 0.125)*(-v + 1)
        elif i == 2:
            dN_dt_C3D8 = (0.125 * s + 0.125) * (-v + 1)
        elif i == 3:
            dN_dt_C3D8 = (-0.125 * s + 0.125) * (-v + 1)
        elif i == 4:
            dN_dt_C3D8 = -(-0.125 * s + 0.125) * (v + 1)
        elif i == 5:
            dN_dt_C3D8 = -(0.125 * s + 0.125) * (v + 1)
        elif i == 6:
            dN_dt_C3D8 = (0.125 * s + 0.125) * (v + 1)
        elif i == 7:
            dN_dt_C3D8 = (-0.125 * s + 0.125) * (v + 1)
        else:
            raise Exception("Unexpected shape function index 'i' in method "
                            "dN_dt_C3D8.")

        return dN_dt_C3D8

    def dN_dv_C3D8(self, i, s, t, v):
        """ C3D8 Shape function derivative method (w.r.t. v)

        Outputs the derivative of the shape function (for 3D elements) with
        respect to the local axis (variable) v.

        - Inputs:
        ---------
        - i (int): number of the node whose shape function derivative is being
          determined.
        - s, t, v (floats): local coordinates of where the derivative should be
          determined.

        - Output:
        ---------
        - dN_ds (float): derivative of the shape function w.r.t. the v local
          axis (variable).
        """
        if i == 0:
            dN_dv_C3D8 = -(-0.125 * s + 0.125) * (-t + 1)
        elif i == 1:
            dN_dv_C3D8 = -(0.125 * s + 0.125) * (-t + 1)
        elif i == 2:
            dN_dv_C3D8 = -(0.125 * s + 0.125) * (t + 1)
        elif i == 3:
            dN_dv_C3D8 = -(-0.125 * s + 0.125) * (t + 1)
        elif i == 4:
            dN_dv_C3D8 = (-0.125 * s + 0.125) * (-t + 1)
        elif i == 5:
            dN_dv_C3D8 = (0.125 * s + 0.125) * (-t + 1)
        elif i == 6:
            dN_dv_C3D8 = (0.125 * s + 0.125) * (t + 1)
        elif i == 7:
            dN_dv_C3D8 = (-0.125 * s + 0.125) * (t + 1)
        else:
            raise Exception("Unexpected shape function index 'i' in method "
                            "dN_dv_C3D8.")

        return dN_dv_C3D8
```

```
    def local_node_coordinates(self):
        """ Local node coordinates method

        Outputs three dictionaries with the coordinates of the element nodes
        as seen in the local element coordinate system.

        If a third dimension does not exist, the 'v' dictionary is returned
        empty.

        Output:
        -------
        - s, t, v (dicts): dictionaries with the local coordinates of each
          node.
        """
        s, t, v = {}, {}, {}

        # coordinates of the nodes for each element type.
        if self.element_type in ["CPS4", "CPE4"]:
            s[0], t[0], v[0] = -1, -1, None
            s[1], t[1], v[1] = 1, -1, None
            s[2], t[2], v[2] = 1, 1, None
            s[3], t[3], v[3] = -1, 1, None

        elif self.element_type == "S4":
            s[0], t[0], v[0] = -1, -1, 0
            s[1], t[1], v[1] = 1, -1, 0
            s[2], t[2], v[2] = 1, 1, 0
            s[3], t[3], v[3] = -1, 1, 0

        elif self.element_type == "C3D8":
            s[0], t[0], v[0] = -1, -1, -1
            s[1], t[1], v[1] = 1, -1, -1
            s[2], t[2], v[2] = 1, 1, -1
            s[3], t[3], v[3] = -1, 1, -1
            s[4], t[4], v[4] = -1, -1, 1
            s[5], t[5], v[5] = 1, -1, 1
            s[6], t[6], v[6] = 1, 1, 1
            s[7], t[7], v[7] = -1, 1, 1

        else:
            raise Exception('Unsuported element type encountered in the '
                            '"local_node_coordinates" method.')

        return s, t, v

    def local_int_point_coordinates(self):
        """ Local integration point coordinates method

        Outputs three dictionaries with the coordinates of the integration
        points of an element as seen in the local element coordinate system.

        If a third dimension does not exist, the 'v_int' dictionary is returned
        empty.

        Output:
        -------
        - s_int, t_int, v_int (dicts): dictionaries with the local coordinates
          of each integration point.
        """
        a = 3.0**(-0.5)
        s_int, t_int, v_int = {}, {}, {}

        # Coordinates of the integration points for each element type.
        if self.element_type in ["CPS4", "CPE4"]:
            s_int[0], t_int[0], v_int[0] = -a, -a, None
            s_int[1], t_int[1], v_int[1] = a, -a, None
            s_int[2], t_int[2], v_int[2] = -a, a, None
            s_int[3], t_int[3], v_int[3] = a, a, None

        elif self.element_type == "S4":
            s_int[0], t_int[0], v_int[0] = -a, -a, 0
            s_int[1], t_int[1], v_int[1] = a, -a, 0
            s_int[2], t_int[2], v_int[2] = -a, a, 0
```

```
            s_int[3], t_int[3], v_int[3] = a, a, 0

        elif self.element_type == "C3D8":
            s_int[0], t_int[0], v_int[0] = -a, -a, -a
            s_int[1], t_int[1], v_int[1] = a, -a, -a
            s_int[2], t_int[2], v_int[2] = -a, a, -a
            s_int[3], t_int[3], v_int[3] = a, a, -a
            s_int[4], t_int[4], v_int[4] = -a, -a, a
            s_int[5], t_int[5], v_int[5] = a, -a, a
            s_int[6], t_int[6], v_int[6] = -a, a, a
            s_int[7], t_int[7], v_int[7] = a, a, a

        else:
            raise Exception(
                'Unsuported element type encountered in the '
                '"local_int_point_coordinates" method.'
            )

        return s_int, t_int, v_int


def c_matrix_function(element_type, material_type, planar):
    """ Stiffness Matrix (D) function

    Determines the stiffness matrix of an element as a function of the
    element type and material type. The function does not consider the
    influence of the SIMP interpolation.

    Inputs:
    -------
    - element_type (str): ABAQUS code defining the element type.
    - material_type (Material_type): ABAQUS code defining the type of the
      material considered.
    - planar (int): variable identifying the type of part considered (2D or
      3D).

    Output:
    -------
    - c_matrix (array): stiffness matrix of the element.
    """
    # If the material properties are given for an Isotropic material:
    if material_type == ISOTROPIC:

        # Plane stress case.
        if element_type == "CPS4":
            e1 = Youngs_modulus
            c11 = e1 / (1 - Poisson ** 2)
            c12 = c11 * Poisson
            c13 = c23 = c12
            c33 = c22 = c11
            num = e1 * (1 - 2 * Poisson) * 0.5
            denom = ((1 - Poisson * 2) * (1 + Poisson))
            c44 = c55 = c66 = num / denom

        elif element_type == "CPE4":
            e1 = Youngs_modulus
            delta = e1 / ((1 + Poisson) * (1 - 2 * Poisson))
            c11 = c22 = c33 = delta * (1 - Poisson)
            c12 = c13 = c23 = delta * Poisson
            c44 = c55 = c66 = delta * (1 - 2 * Poisson) * 0.5

        # Shell element case.
        elif element_type == "S4":
            e1 = Youngs_modulus
            c11 = e1 / (1 - Poisson ** 2)
            c12 = c11 * Poisson
            c13 = c23 = 0.0
            c22 = c11
            c33 = 0.0
            num = e1 * (1 - 2 * Poisson) * 0.5
            denom = ((1 - Poisson * 2) * (1 + Poisson))
            c44 = c55 = c66 = num / denom

        # 3D element case.
```

```
        elif element_type == "C3D8":
            e1 = Youngs_modulus
            c11 = e1 * (1 - Poisson) / ((1 - Poisson* 2 ) * (1 + Poisson))
            c12 = e1 * (Poisson) / ((1 - Poisson * 2) * (1 + Poisson))
            c13 = c23 = c12
            c33 = c22 = c11
            c44 = c55 = c66 = (((1 - Poisson) / 2 * e1) / (1 - Poisson ** 2))

        # For other cases, assume the 3D Hook's Law.
        else:
            delta = Youngs_modulus / ((1 + Poisson) * (1 - 2 * Poisson))
            c11 = c22 = c33 = delta * (1 - Poisson)
            c12 = c13 = c23 = delta * Poisson
            c44 = c55 = c66 = delta * ((1 - 2 * Poisson) / 2)

    # If the material properties are defined by engineering constants:
    elif material_type == ENGINEERING_CONSTANTS:
        e1 = E11
        e2 = E22
        e3 = E33
        Nu21 = e2 * Nu12 / e1
        Nu32 = e3 * Nu23 / e2
        Nu31 = e3 * Nu13 / e1
        num = (1 - Nu12 * Nu21 - Nu23 * Nu32 - Nu31 * Nu13
               - 2 * Nu21 * Nu32 * Nu13)
        denom = (e1 * e2 * e3)
        delta = num / denom
        c11 = (1 - Nu23 * Nu32) / (e2 * e3 * delta)
        c12 = (Nu12 + Nu32 * Nu13) / (e1 * e3 * delta)
        c13 = (Nu13 + Nu12 * Nu23) / (e1 * e2 * delta)
        c22 = (1 - Nu13 * Nu31)/(e1 * e3 * delta)
        c23 = (Nu23 + Nu21 * Nu13) / (e1 * e3 * delta)
        c33 = (1 - Nu12 * Nu21) / (e1 * e2 * delta)
        c44 = G12 * 0.5
        c55 = G23 * 0.5
        c66 = G13 * 0.5

    else:
        raise Exception(
      "The 'stiffness_matrix' function found no material properties \n"
      "in the form of 'ISOTROPIC' or 'ENGINEERING_CONSTANTS' for \n"
      "material {}.".format(MATERIAL_NAME)
    )

    # Build the C matrix for 2D or 3D problems.
    if planar == 1:
        c_matrix = np.array([[c11, c12,     0],
                             [c12, c22,     0],
                             [  0,   0,   c44]])

    elif planar == 0:
        c_matrix = np.array([[c11, c12, c13,   0,   0,   0],
                             [c12, c22, c23,   0,   0,   0],
                             [c13, c23, c33,   0,   0,   0],
                             [  0,   0,   0, c44,   0,   0],
                             [  0,   0,   0,   0, c55,   0],
                             [  0,   0,   0,   0,   0, c66]])

    else:
        raise Exception(
            "Unexpected combination of 'element_type' and 'planar' variables\n"
            "in the 'c_matrix_function'."
        )

    return c_matrix


#%% Parameter input request, domain definition, and variable generation.
class ParameterInput():
    """ Parameter input class

    This class creates a simple graphic interface asking the user to input
    information about the numerical model, the topology optimization problem
    and the parameters to be used.
```

```
    While collecting the information, this class will also check if the user
    has input correct information (i.e. if the model exists or if the values
    are within their expectable domain).

    Due to the large number of inputs, the variables are generated through
    the Global command instead of the Python return function, as a means to
    maintain the code organized and the main section cleaner.

    Methods:
    --------
    - model_information(): creates a pop-up requesting general information the
      ABAQUS file and the numerical model to be used in the topology
      optimization process.
    - problem_statement(): creates a pop-up requesting the user to identify
      the type of topology optimization process to be considered and the
      internal parameters to be used.
    - return_inputs(model_inputs, user_inputs): based on the information input,
      a large number of global variables are created and 'returned' through the
      global command.
    """
    def __init__(self):
        pass

    def model_information(self):
        """ Model information method

        Creates a pop-up requesting the user to input the name of the ABAQUS
        CAE file, the model name, material name, indicate if the mesh is
        uniform, the number of job domains and CPUs to consider, and if the
        code should only consider the output obtained for the last frame of
        each step.

        Output:
        -------
        - model_inputs (list): list containing the variables with the
          information introduced by the user.
        """
        #Input the name of the CAE file, model, part, and material considered,
        #as well as the mesh uniformity and the number of job domains and CPUs.
        pars = (('CAE file:','L-bracket.cae'),
                ('Model name:','Model-1'),
                ('Part name:','Part-1'),
                ('Material name:','Material-1'),
                ('Section name:','Section-1'),
                ('Is the mesh uniform? (Y=1/N=0)','1'),
                ('Number of domains of the Job:', '4'),
                ('Number of CPUs used in the FEA:','4'),
                ('Check only the outputs from the last frame? (Y=1/N=0)','1'))

        exception_message = (
            "Invalid input(s) in the 'Model and Job' tab. \n"
            "Please consider the following requirements: \n"
            "- Select an ABAQUS CAE file that exists in the current \n"
            "  working directory."
            "- The number of domains must be equal to or a multiple of \n "
            "  the number of processors (CPUs). \n"
            "- Both the number of domains and the number of CPUs must \n "
            "  be larger than 0. \n"
            "- The input for the mesh uniformity should be either 0 or 1."
            "\n"
            "- Please use either 1 or 0 (Y=1/N=0) to indicate if the \n "
            "  program should only check results from the last frame of \n"
            "  the odb file or from all available frames of the odb."
        )

        temp_output = getInputs(pars,
                                dialogTitle = 'Model and Job information')

        try:
            cae_name, model_name, part_name, material_name, section_name = [
               str(k) if k not in [None, ''] else pars[k][1]
               for k in temp_output[0:5]]
```

```
            mesh_uniformity, n_domains, n_cpus, last_frame = [
                        int(float(k)) if k not in [None, ''] else 0
                        for k in temp_output[5:]]
        except:
            raise Exception(exception_message)

        #Confirm the usage, or not, of the file extension '.cae' in the CAE
        #name input and act accordingly.
        if cae_name[-4:] == ".cae" or cae_name[-4:] == ".CAE":
            cae_name = cae_name[:-4] + ".cae"
        elif cae_name[-4:] != ".cae" or cae_name[-4:] != ".CAE":
            cae_name = cae_name + ".cae"
        else:
            raise Exception(
                    "Unexpected error in the cae_name verification loop. \n")

        #Open the CAE file.
        mdb = openMdb(cae_name)

        #Confirm the existence of the Model.
        if model_name == '': model_name = 'Model-1'
        if model_name not in mdb.models.keys():
            raise Exception("Model named {} not found in the {} file. \n"\
                            .format(model_name,cae_name))

        #Confirm the existence of the Part.
        if part_name == '': part_name = 'Part-1'
        if part_name not in mdb.models[model_name].parts.keys():
            raise Exception("Part named {} not found in Model {}. \n"\
                            .format(part_name,model_name))

        #Confirm the existence of the Material.
        if material_name == '': material_name = 'Material-1'
        if material_name not in mdb.models[model_name].materials.keys():
            raise Exception("Material named {} not found in the materials "
                            "of model {}. \n".format(material_name,model_name))

        #Confirm the existence of the Section.
        if section_name == '': section_name = 'Section-1'
        if section_name not in mdb.models[model_name].sections.keys():
            raise Exception("Section named {} not found in model {}. \n"\
                            .format(section_name,model_name))

        #Confirm that the inputs have acceptable values.
        if (n_domains <= 0
            or n_cpus <= 0
            or n_domains % n_cpus != 0
            or mesh_uniformity not in [0,1]
            or last_frame not in [0,1]
        ):
            raise Exception(exception_message)

        model_inputs = [mdb, cae_name, model_name, part_name, material_name,
                        section_name, mesh_uniformity, n_domains, n_cpus,
                        last_frame]

        return model_inputs

    def problem_statement(self):
        """ Problem statement method

        Creates a pop-up requesting the user to input information regarding the
        topology optimization problem to be solved. This information includes
        the selection of an optimization solver, the constraints to be
        considered, and the necessary internal parameters.

        Output:
        -------
        - user_inputs (list): list containing the variables with the
          information introduced by the user.
        """

        pars = (
            ('Problem statement and solver selected:','1'),
```

```
    ('Consider constrained Mass or Volume? \n '
    '(Mass = 0 / Volume = 1):', '1'),
    ('Consider geometric non-linearities? (Yes=1 / No=0):', '0')
)

Label = (
    'Please introduce the number corresponding to the problem \n'
    'type and optimization solver that you would like to use. \n \n'
    'Compliance minimization solved with: \n'
    '      0 - OC for discrete design variables. \n'
    '      1 - OC for continuous design variables. \n'
    '      2 - MMA for continuous design variables. \n'
    '      3 - SciPy solver for continuous design variables. \n \n'
    'Stress constrained compliance minimization solved with: \n'
    '      4 - MMA for continuous design variables. \n'
    '      5 - SciPy solver for continuous design variables. \n \n'
    'Maximum stress minimization solved with: \n'
    '      6 - MMA for continuous design variables. \n'
    '      7 - SciPy solver for continuous design variables. \n \n'
    '(*) Notes: \n'
    '      a) OC - Optimality Criteria. \n'
    '      b) MMA - Method of Moving Asymptotes. \n'
    '      c) Continuous design variables assume a Solid Isotropic \n'
    '      Material with Penalization (SIMP). \n'
    '      d) The SciPy solver may require the user to edit the code \n'
    '      to access all internal parameters and options.'
    '\n'
)

exception_message = (
    "Invalid input in the 'Parameters tab'. \n"
    "Please indicate the optimization method with an integer number \n"
    "from 0 up to 8, and all answer the remaining questions in this \n"
    "tab with either 0 or 1. \n"
)

try:
    (
        opt_method,
        material_constraint,
        nonlinearities,
    ) = [int(float(k)) if k not in [None, ''] else 0
         for k in getInputs(pars,
                            dialogTitle = 'Problem statement',
                            label = Label)]
except:
    raise Exception(exception_message)

# Confirm problem statement and optimization method requirements.
if (
        material_constraint not in [0,1]
        or nonlinearities not in [0,1]
        or opt_method not in range(0,8)
    ):
    raise Exception(exception_message)

nonlinearities = True if nonlinearities == 1 else False

# Request additional information for the problem statement defined, if
# valid.

# For discrete compliance minimization problems.
if opt_method in [0]:

    pars = (('Material constraint ratio (Target Volume or Mass over '
             'a fully solid design, between 0 and 1)','0.5'),
            ('Material constraint evolution ratio:', '0.05'),
            ('Min. Element density:', '0.01'),
            ('Filter radius:','7.5'),
            ('Filter sensitivities? (Y=1/N=0)', '1'),
            ('Filter design densities? (Y=1/N=0)', '0'),
            ('SIMP penalty factor:','3.0'),
            ('Initial density of the elements (set 0 for a '
             'random distribution)','1.0'),
```

```
            ('Consider frozen region? (Y=1/N=0)', '0'),
            ('Consider neighbouring region? (Y=1/N=0)', '0'))

exception_message = (
    "Invalid input in the 'Topology Optimization parameters'. \n"
    "Please consider the following requirements: \n"
    "-The material constraint ratio, minimum element density, \n"
    " material constraint evolution ratio, and initial density, \n"
    " inputs should be a value between 0 and 1. \n"
    "-The initial density should only be equal to 0 when \n"
    " requesting a random density distribution for the first \n"
    " iteration. \n"
    "-The SIMP penalty factor should be larger than 0 . \n"
    "-The filter radius should be larger or equal to 0.0. \n"
    "-Indicate if you want to filter the sensitivities and/or \n"
    " design densities by answering with either 1 or 0 to each \n"
    " question (Yes=1 / No=0).\n"
    "-The 'consider_frozen_region' and \n"
    " 'consider_neighbouring_region' variables, defining the \n"
    " consideration of frozen and neighbouring regions, should \n"
    " be equal to 0 or 1. \n"
)

try:
    (
        target_material,
        evol_ratio,
        xe_min,
        rmax,
        filter_sensitivities,
        filter_densities,
        p,
        initial_density,
        consider_frozen_region,
        consider_neighbouring_region,
    ) = [float(k) if k not in [None, ''] else 0
         for k in getInputs(pars,
                            dialogTitle = "Problem statement")]
except:
    raise Exception(exception_message)

consider_frozen_region = int(round(consider_frozen_region,0))
consider_neighbouring_region = int(round(
                                    consider_neighbouring_region,0))
filter_sensitivities = int(round(filter_sensitivities,0))
filter_densities = int(round(filter_densities,0))

# Set the unused variables.
move_limit = None
s_max, p_norm_stress = None, None
qi, qf = 1.0, 1.0
stress_sensitivity = {}
algorithm = None

# Confirm that the topology optimization parameters input have
# acceptable values.
if (
    target_material <= 0.0
    or target_material > 1.0
    or evol_ratio > 1.0
    or evol_ratio <= 0.0
    or xe_min <= 0.0
    or xe_min > 1.0
    or rmax < 0
    or p < 0.0
    or initial_density < 0.0
    or initial_density > 1.0
    or consider_frozen_region not in [0,1]
    or consider_neighbouring_region not in [0,1]
    or filter_sensitivities not in [0,1]
    or filter_densities not in [0,1]
):
    raise Exception(exception_message)
```

```
        print ("Casual reminder: \n"
               "-The problem defined by these inputs may be solved \n"
               " as a continuous problem with the Optimality Criteria, \n"
               " Method of Moving Asymptotes, or through the SciPy \n"
               " optimizers implemented, and referenced, in the code \n"
               " provided. \n")

    # For continuous compliance minimization problems.
    elif opt_method in [1, 2]:

        pars = (('Material constraint ratio (Target Volume or Mass over '
                 'a fully solid design, between 0 and 1)','0.5'),
                ('Material constraint evolution ratio:', '0.05'),
                ('Min. Element density:', '0.01'),
                ('Filter radius:','7.5'),
                ('Filter sensitivities? (Y=1/N=0)', '1'),
                ('Filter design densities? (Y=1/N=0)', '1'),
                ('SIMP penalty factor:','3.0'),
                ('Initial density of the elements (set 0 for a '
                 'random distribution)','1.0'),
                ('Move limit of the design variables:', '0.2'),
                ('Consider frozen region? (Y=1/N=0)', '0'),
                ('Consider neighbouring region? (Y=1/N=0)', '0'))

        exception_message = (
            "Invalid input in the 'Topology Optimization parameters'. \n"
            "Please consider the following requirements: \n"
            "-The material constraint ratio, minimum element density, \n"
            " material constraint evolution ratio, initial density, \n"
            " and move limit inputs should be a value between 0 and 1. "
            "\n"
            "-The initial density should only be equal to 0 when \n"
            " requesting a random density distribution for the first \n"
            " iteration. \n"
            "-The SIMP penalty factor should be larger than 0 . \n"
            "-The filter radius should be larger or equal to 0.0. \n"
            "-Indicate if you want to filter the sensitivities and/or \n"
            " design densities by answering with either 1 or 0 to each \n"
            " question (Yes=1 / No=0).\n"
            "-The 'consider_frozen_region' and \n"
            " 'consider_neighbouring_region' variables, defining the \n"
            " consideration of frozen and neighbouring regions, should \n"
            " be equal to 0 or 1. \n"
        )

        try:
            (
                target_material,
                evol_ratio,
                xe_min,
                rmax,
                filter_sensitivities,
                filter_densities,
                p,
                initial_density,
                move_limit,
                consider_frozen_region,
                consider_neighbouring_region,
            ) = [float(k) if k not in [None, ''] else 0
                 for k in getInputs(pars,
                                    dialogTitle = "Problem statement")]
        except:
            raise Exception(exception_message)

        consider_frozen_region = int(round(consider_frozen_region,0))
        consider_neighbouring_region = int(round(
                                           consider_neighbouring_region,0))
        filter_sensitivities = int(round(filter_sensitivities,0))
        filter_densities = int(round(filter_densities,0))

        # Set the unused variables.
        s_max, p_norm_stress = None, None
        qi, qf = 1.0, 1.0
        stress_sensitivity = {}
```

```
        algorithm = None

        # Confirm that the topology optimization parameters input have
        # acceptable values.
        if (
            target_material <= 0.0
            or target_material > 1.0
            or evol_ratio > 1.0
            or evol_ratio <= 0.0
            or xe_min <= 0.0
            or xe_min > 1.0
            or rmax < 0
            or p < 0.0
            or initial_density < 0.0
            or initial_density > 1.0
            or move_limit <= 0.0
            or move_limit > 1.0
            or consider_frozen_region not in [0,1]
            or consider_neighbouring_region not in [0,1]
            or filter_sensitivities not in [0,1]
            or filter_densities not in [0,1]
        ):
            raise Exception(exception_message)

        print ("Casual reminder: \n"
               "-The problem defined by these inputs may be solved \n"
               " with the Optimality Criteria, Method of Moving \n"
               " Asymptotes, or through the SciPy optimizers implemented,"
               "\n and referenced, in the code provided. \n")

    elif opt_method in [3]:

        pars = (('Material constraint ratio (Target Volume or Mass over '
                 'a fully solid design, between 0 and 1)','0.5'),
                ('Material constraint evolution ratio:', '0.05'),
                ('Min. Element density:', '0.01'),
                ('Filter radius:','7.5'),
                ('Filter sensitivities? (Y=1/N=0)', '1'),
                ('Filter design densities? (Y=1/N=0)', '1'),
                ('SIMP penalty factor:','3.0'),
                ('Initial density of the elements (set 0 for a '
                 'random distribution)','1.0'),
                ('Consider frozen region? (Y=1/N=0)', '0'),
                ('Consider neighbouring region? (Y=1/N=0)', '0'),
                ('Solve with "SLSQP" or "trust-constr"? '
                 '(SLSQP=1/trust-constr=0)', '1'))

        exception_message = (
            "Invalid input in the 'Topology Optimization parameters'."
            "\n Please consider the following requirements: \n"
            "-The material constraint ratio, minimum element density, \n"
            " material constraint evolution ratio, and initial density \n"
            " inputs should be a value between 0 and 1. \n"
            "-The initial density should only be equal to 0 when \n"
            " requesting a random density distribution for the first \n"
            " iteration. \n"
            "-The SIMP penalty factor should be larger than 0 . \n"
            "-The filter radius should be larger or equal to 0.0. \n"
            "-Indicate if you want to filter the sensitivities and/or \n"
            " design densities by answering with either 1 or 0 to each \n"
            " question (Yes=1 / No=0).\n"
            "-The 'consider_frozen_region' and \n"
            " 'consider_neighbouring_region' variables, defining the \n"
            " consideration of frozen and neighbouring regions, should \n"
            " be equal to 0 or 1. \n"
            "-Select the optimization solver with either 1 or 0 \n"
            " (SLSQP=1/trust-constr=0)."
        )

        try:
            (
                target_material,
                evol_ratio,
                xe_min,
```

```
                rmax,
                filter_sensitivities,
                filter_densities,
                p,
                initial_density,
                consider_frozen_region,
                consider_neighbouring_region,
                algorithm
            ) = [float(k) if k not in [None, ''] else 0
                 for k in getInputs(pars,
                                    dialogTitle = "Problem statement")]
        except:
            raise Exception(exception_message)

        consider_frozen_region = int(round(consider_frozen_region,0))
        consider_neighbouring_region = int(round(
                                            consider_neighbouring_region,0))
        filter_sensitivities = int(round(filter_sensitivities,0))
        filter_densities = int(round(filter_densities,0))
        algorithm = int(round(algorithm, 0))

        # Set the unused variables.
        move_limit = None
        s_max, p_norm_stress = None, None
        qi, qf = 1.0, 1.0
        stress_sensitivity = {}

        #Confirm that the topology optimization parameters input have
        #acceptable values.
        if (
            target_material <= 0.0
            or target_material > 1.0
            or evol_ratio > 1.0
            or evol_ratio <= 0.0
            or xe_min <= 0.0
            or xe_min > 1.0
            or rmax < 0
            or p < 0.0
            or initial_density < 0.0
            or initial_density > 1.0
            or consider_frozen_region not in [0,1]
            or consider_neighbouring_region not in [0,1]
            or filter_sensitivities not in [0,1]
            or filter_densities not in [0,1]
        ):
            raise Exception(exception_message)

        algorithm = 'SLSQP' if algorithm == 1 else 'trust-constr'

        print ("Casual reminder: \n"
               "-The problem defined by these inputs may be solved \n"
               " with the Optimality Criteria, Method of Moving \n"
               " Asymptotes, or through the SciPy optimizers implemented,"
               "\n and referenced, in the code provided. \n")


    # For stress constrained compliance minimization problems:
    elif opt_method in [4]:
        pars = (('Material constraint ratio (Target Volume or Mass over a'
                 'fully solid design, between 0 and 1)','0.5'),
                ('Material constraint evolution ratio:', '1.0'),
                ('Max. Stress value', '350.0'),
                ('Min. Element density:', '0.01'),
                ('Filter radius:','5.0'),
                ('Filter sensitivities? (Y=1/N=0)', '1'),
                ('Filter design densities? (Y=1/N=0)', '1'),
                ('SIMP penalty factor:','3.0'),
                ('Initial density of the elements (set 0 for a random '
                 'distribution)','1.0'),
                ('Move limit of the design variables:', '0.2'),
                ('Initial P_norm factor:','8.0'),
                ('Maximum P_norm factor:','8.0'),
                ('Consider frozen region? (Y=1/N=0)', '0'),
                ('Consider neighbouring region? (Y=1/N=0)', '0'))
```

```
exception_message = (
    "Invalid input in the 'Topology Optimization parameters'."
    "\n Please consider the following requirements: \n"
    "-The material constraint ratio, minimum element density, \n"
    " material constraint evolution ratio, initial density, \n"
    " and move limit inputs should be a value between 0 and 1. "
    "\n"
    "-The initial density should only be equal to 0 when \n"
    " requesting a random density distribution for the first \n"
    " iteration. \n"
    "-The stress constraint value and the SIMP penalty factor \n"
    " should be larger than 0 . \n"
    "-The filter radius should be larger or equal to 0.0. \n"
    "-Indicate if you want to filter the sensitivities and/or \n"
    " design densities by answering with either 1 or 0 to each \n"
    " question (Yes=1 / No=0).\n"
    "-The initial and maximum P_norm factors should both be \n"
    " larger than 0, with the final actor being larger than \n"
    " the initial. \n"
    "-The 'consider_frozen_region' and \n"
    " 'consider_neighbouring_region' variables, defining the \n"
    " consideration of frozen and neighbouring regions, should \n"
    " be equal to 0 or 1. \n \n"
    "Additionally, the user is reminded that: \n"
    "-The stress constraint variable should be 'S' unless the \n"
    " user has modified the code provided. This input is not \n"
    " verified by the code provided. \n"
    "-Likewise, the additional constraint variable(s) \n"
    " requested and corresponding maximum value are not \n"
    " verified by the code provided. Therefore, the user \n"
    " should verify the input introduced. \n"
)

#In this case, the information is processed differently to avoid
#additional pop-ups.
temp_outputs = getInputs(
            pars, dialogTitle = 'Topology Optimization parameters')

try:
    (
        target_material,
        evol_ratio,
        s_max,
        xe_min,
        rmax,
        filter_sensitivities,
        filter_densities,
        p,
        initial_density,
        move_limit,
        qi,
        qf,
        consider_frozen_region,
        consider_neighbouring_region
    ) = [float(k) if k not in [None, ''] else 0
         for k in temp_outputs]
except:
    raise Exception(exception_message)

consider_frozen_region = int(round(consider_frozen_region,0))
consider_neighbouring_region = int(round(
                                    consider_neighbouring_region,0))
filter_sensitivities = int(round(filter_sensitivities,0))
filter_densities = int(round(filter_densities,0))

qi, qf = float(round(qi,0)), float(round(qf,0))
p_norm_stress, stress_sensitivity = None, None
algorithm = None

#Confirm that the topology optimization parameters input have
#acceptable values.
if (
    target_material <= 0.0
```

```
            or target_material > 1.0
            or evol_ratio > 1.0
            or evol_ratio <= 0.0
            or s_max <= 0
            or xe_min <= 0.0
            or xe_min > 1.0
            or rmax < 0
            or p < 0.0
            or initial_density < 0.0
            or initial_density > 1.0
            or move_limit <= 0.0
            or move_limit > 1.0
            or qi <= 0.0
            or qf <= 0.0
            or qf < qi
            or consider_frozen_region not in [0,1]
            or consider_neighbouring_region not in [0,1]
            or filter_sensitivities not in [0,1]
            or filter_densities not in [0,1]
        ):
            raise Exception(exception_message)

        print("Casual reminder: \n"
              "-The problem defined by these inputs may be solved \n"
              " with the Method of Moving Asymptotes, or through the \n"
              " SciPy optimizers implemented, and referenced, in the \n"
              " code provided. \n")

    elif opt_method in [5]:
        pars = (('Material constraint ratio (Target Volume or Mass over a'
                 'fully solid design, between 0 and 1)','0.5'),
                ('Material constraint evolution ratio:', '1.0'),
                ('Max. Stress value', '350.0'),
                ('Min. Element density:', '0.01'),
                ('Filter radius:','5.0'),
                ('Filter sensitivities? (Y=1/N=0)', '1'),
                ('Filter design densities? (Y=1/N=0)', '1'),
                ('SIMP penalty factor:','3.0'),
                ('Initial density of the elements (set 0 for a random '
                 'distribution)','1.0'),
                ('Initial P_norm factor:','8.0'),
                ('Maximum P_norm factor:','8.0'),
                ('Consider frozen region? (Y=1/N=0)', '0'),
                ('Consider neighbouring region? (Y=1/N=0)', '0'),
                ('Solve with "SLSQP" or "trust-constr"? '
                 '(SLSQP=1/trust-constr=0)', '1'))

        exception_message = (
            "Invalid input in the 'Topology Optimization parameters'."
            "\n Please consider the following requirements: \n"
            "-The material constraint ratio, minimum element density, "
            " material constraint evolution ratio, and initial density "
            " inputs should be a value between 0 and 1. \n"
            "-The initial density should only be equal to 0 when "
            " requesting a random density distribution for the first "
            " iteration. \n"
            "-The stress constraint value and the SIMP penalty factor "
            " should be larger than 0 . \n"
            "-The filter radius should be larger or equal to 0.0. \n"
            "-Indicate if you want to filter the sensitivities and/or \n"
            " design densities by answering with either 1 or 0 to each \n"
            " question (Yes=1 / No=0).\n"
            "-The initial and maximum P_norm factors should both be "
            " larger than 0, with the final actor being larger than "
            " the initial. \n"
            "-The 'consider_frozen_region' and "
            " 'consider_neighbouring_region' variables, defining the "
            " consideration of frozen and neighbouring regions, should "
            " be equal to 0 or 1. "
            "-Select the optimization solver with either 1 or 0 \n"
            " (SLSQP=1/trust-constr=0).\n \n"
        )

        #In this case, the information is processed differently to avoid
```

```
        #additional pop-ups.
        temp_outputs = getInputs(
                    pars, dialogTitle = 'Topology Optimization parameters')

        try:
            (
                target_material,
                evol_ratio,
                s_max,
                xe_min,
                rmax,
                filter_sensitivities,
                filter_densities,
                p,
                initial_density,
                qi,
                qf,
                consider_frozen_region,
                consider_neighbouring_region,
                algorithm
            ) = [float(k) if k not in [None, ''] else 0
                 for k in temp_outputs]
        except:
            raise Exception(exception_message)

        consider_frozen_region = int(round(consider_frozen_region,0))
        consider_neighbouring_region = int(round(
                                        consider_neighbouring_region,0))
        filter_sensitivities = int(round(filter_sensitivities,0))
        filter_densities = int(round(filter_densities,0))
        algorithm = int(round(algorithm, 0))

        qi, qf = float(round(qi,0)), float(round(qf,0))
        move_limit = None
        p_norm_stress, stress_sensitivity = None, None

        #Confirm that the topology optimization parameters input have
        #acceptable values.
        if (
            target_material <= 0.0
            or target_material > 1.0
            or evol_ratio > 1.0
            or evol_ratio <= 0.0
            or s_max <= 0
            or xe_min <= 0.0
            or xe_min > 1.0
            or rmax < 0
            or p < 0.0
            or initial_density < 0.0
            or initial_density > 1.0
            or qi <= 0.0
            or qf <= 0.0
            or qf < qi
            or consider_frozen_region not in [0,1]
            or consider_neighbouring_region not in [0,1]
            or filter_sensitivities not in [0,1]
            or filter_densities not in [0,1]
            or algorithm not in [0,1]
        ):
            raise Exception(exception_message)

        algorithm = 'SLSQP' if algorithm == 1 else 'trust-constr'

        print("Casual reminder: \n"
              "-The problem defined by these inputs may be solved \n"
              " with the Method of Moving Asymptotes, or through the \n"
              " SciPy optimizers implemented, and referenced, in the \n"
              " code provided. \n")

    # For stress minimization problems:
    elif opt_method in [6]:
        pars = (('Material constraint ratio (Target Volume or Mass over a'
                 'fully solid design, between 0 and 1)','0.5'),
                ('Material constraint evolution ratio:', '1.0'),
```

```
            ('Min. Element density:', '0.01'),
            ('Filter radius:','5.0'),
            ('Filter sensitivities? (Y=1/N=0)', '1'),
            ('Filter design densities? (Y=1/N=0)', '1'),
            ('SIMP penalty factor:','3.0'),
            ('Initial density of the elements (set 0 for a random '
             'distribution)','1.0'),
            ('Move limit of the design variables:', '0.2'),
            ('Initial P_norm factor:','8.0'),
            ('Maximum P_norm factor:','8.0'),
            ('Consider frozen region? (Y=1/N=0)', '0'),
            ('Consider neighbouring region? (Y=1/N=0)', '0'))

exception_message = (
    "Invalid input in the 'Topology Optimization parameters'."
    "\n Please consider the following requirements: \n"
    "-The material constraint ratio, minimum element density, "
    " material constraint evolution ratio, initial density, "
    " and move limit inputs should be a value between 0 and 1. "
    "\n"
    "-The initial density should only be equal to 0 when "
    " requesting a random density distribution for the first "
    " iteration. \n"
    "-The SIMP penalty factor should be larger than 0. \n"
    "-The filter radius should be larger or equal to 0.0. \n"
    "-Indicate if you want to filter the sensitivities and/or \n"
    " design densities by answering with either 1 or 0 to each \n"
    " question (Yes=1 / No=0).\n"
    "-The initial and maximum P_norm factors should both be "
    " larger than 0, with the final actor being larger than "
    " the initial. \n"
    "-The 'consider_frozen_region' and "
    " 'consider_neighbouring_region' variables, defining the "
    " consideration of frozen and neighbouring regions, should "
    " be equal to 0 or 1. \n \n"
    "Additionally, the user is reminded that: \n"
    "-The stress constraint variable should be 'S' unless the "
    " user has modified the code provided. This input is not "
    " verified by the code provided. \n"
    "-Likewise, the additional constraint variable(s) "
    " requested and corresponding maximum value are not "
    " verified by the code provided. Therefore, the user "
    " should verify the input introduced. \n"
)

#In this case, the information is processed differently to avoid
#additional pop-ups.
temp_outputs = getInputs(
            pars, dialogTitle = 'Topology Optimization parameters')

try:
    (
        target_material,
        evol_ratio,
        xe_min,
        rmax,
        filter_sensitivities,
        filter_densities,
        p,
        initial_density,
        move_limit,
        qi,
        qf,
        consider_frozen_region,
        consider_neighbouring_region
    ) = [float(k) if k not in [None, ''] else 0
         for k in temp_outputs]
except:
    raise Exception(exception_message)

consider_frozen_region = int(round(consider_frozen_region,0))
consider_neighbouring_region = int(round(
                                    consider_neighbouring_region,0))
```

```
        filter_sensitivities = int(round(filter_sensitivities,0))
        filter_densities = int(round(filter_densities,0))
        qi, qf = float(round(qi,0)), float(round(qf,0))
        p_norm_stress, stress_sensitivity = None, None
        algorithm = None
        s_max = 1.0
        #Confirm that the topology optimization parameters input have
        #acceptable values.
        if (
            target_material <= 0.0
            or target_material > 1.0
            or evol_ratio > 1.0
            or evol_ratio <= 0.0
            or xe_min <= 0.0
            or xe_min > 1.0
            or rmax < 0
            or p < 0.0
            or initial_density < 0.0
            or initial_density > 1.0
            or move_limit <= 0.0
            or move_limit > 1.0
            or qi <= 0.0
            or qf <= 0.0
            or qf < qi
            or consider_frozen_region not in [0,1]
            or consider_neighbouring_region not in [0,1]
            or filter_sensitivities not in [0,1]
            or filter_densities not in [0,1]
        ):
            raise Exception(exception_message)

        print("Casual reminder: \n"
              "-The problem defined by these inputs may be solved \n"
              " with the Method of Moving Asymptotes, or through the \n"
              " SciPy optimizers implemented, and referenced, in the \n"
              " code provided. \n")

    elif opt_method in [7]:
        pars = (('Material constraint ratio (Target Volume or Mass over a'
                 'fully solid design, between 0 and 1)','0.5'),
                ('Material constraint evolution ratio:', '1.0'),
                ('Min. Element density:', '0.01'),
                ('Filter radius:','5.0'),
                ('Filter sensitivities? (Y=1/N=0)', '1'),
                ('Filter design densities? (Y=1/N=0)', '1'),
                ('SIMP penalty factor:','3.0'),
                ('Initial density of the elements (set 0 for a random '
                 'distribution)','1.0'),
                ('Initial P_norm factor:','8.0'),
                ('Maximum P_norm factor:','8.0'),
                ('Consider frozen region? (Y=1/N=0)', '0'),
                ('Consider neighbouring region? (Y=1/N=0)', '0'),
                ('Solve with "SLSQP" or "trust-constr"? '
                 '(SLSQP=1/trust-constr=0)', '1'))

        exception_message = (
            "Invalid input in the 'Topology Optimization parameters'."
            "\n Please consider the following requirements: \n"
            "-The material constraint ratio, minimum element density, "
            " material constraint evolution ratio, and initial density "
            " inputs should be a value between 0 and 1. \n"
            "-The initial density should only be equal to 0 when "
            " requesting a random density distribution for the first "
            " iteration. \n"
            "-The SIMP penalty factor should be larger than 0. \n"
            "-The filter radius should be larger or equal to 0.0. \n"
            "-Indicate if you want to filter the sensitivities and/or \n"
            " design densities by answering with either 1 or 0 to each \n"
            " question (Yes=1 / No=0).\n"
            "-The initial and maximum P_norm factors should both be "
            " larger than 0, with the final actor being larger than "
            " the initial. \n"
            "-The 'consider_frozen_region' and "
            " 'consider_neighbouring_region' variables, defining the "
```

```
    " consideration of frozen and neighbouring regions, should "
    " be equal to 0 or 1. "
    "-Select the optimization solver with either 1 or 0 \n"
    " (SLSQP=1/trust-constr=0). \n \n"
)

#In this case, the information is processed differently to avoid
#additional pop-ups.
temp_outputs = getInputs(
            pars, dialogTitle = 'Topology Optimization parameters')

try:
    (
        target_material,
        evol_ratio,
        xe_min,
        rmax,
        filter_sensitivities,
        filter_densities,
        p,
        initial_density,
        qi,
        qf,
        consider_frozen_region,
        consider_neighbouring_region,
        algorithm
    ) = [float(k) if k not in [None, ''] else 0
         for k in temp_outputs]
except:
    raise Exception(exception_message)

consider_frozen_region = int(round(consider_frozen_region,0))
consider_neighbouring_region = int(round(
                                consider_neighbouring_region,0))
filter_sensitivities = int(round(filter_sensitivities,0))
filter_densities = int(round(filter_densities,0))
algorithm = int(round(algorithm, 0))

qi, qf = float(round(qi,0)), float(round(qf,0))
move_limit = None
p_norm_stress, stress_sensitivity = None, None
s_max = 1.0
#Confirm that the topology optimization parameters input have
#acceptable values.
if (
    target_material <= 0.0
    or target_material > 1.0
    or evol_ratio > 1.0
    or evol_ratio <= 0.0
    or xe_min <= 0.0
    or xe_min > 1.0
    or rmax < 0
    or p < 0.0
    or initial_density < 0.0
    or initial_density > 1.0
    or qi <= 0.0
    or qf <= 0.0
    or qf < qi
    or consider_frozen_region not in [0,1]
    or consider_neighbouring_region not in [0,1]
    or filter_sensitivities not in [0,1]
    or filter_densities not in [0,1]
    or algorithm not in [0,1]
):
    raise Exception(exception_message)

algorithm = 'SLSQP' if algorithm == 1 else 'trust-constr'

print("Casual reminder: \n"
      "-The problem defined by these inputs may be solved \n"
      " with the Method of Moving Asymptotes, or through the \n"
      " SciPy optimizers implemented, and referenced, in the \n"
      " code provided. \n")
```

```
        else:
            raise Exception("Unexpected error in the selection of the "
                            "Optimization Method in the 'Problem statement' "
                            "tab. \n")

        filter_sensitivities = True if filter_sensitivities == 1 else False
        filter_densities = True if filter_densities == 1 else False

        if opt_method >= 4:
            pars = (
                ('1 - Plot element design density?','1'),
                ('2 - Plot element stress?', '0'),
                ('3 - Plot element stress raised to the P-norm exponent?',
                '0'),
                ('4 - Plot element amplified stress?', '0'),
                ('5 - Plot element amplified stress raised to the P-norm '
                'exponent?', '0'),
                ('Number of the preferred plot?', '1'),
                ('Maximum value of the scale in the stress plot (optional):',
                 '')
             )

            Label = (
                'You have defined a stress dependent topology optimization \n'
                'problem. Please select which information you would like to \n'
                'plot, by answering "1" or "0" in each box (Yes=1/No=0). \n \n'
                'If requesting a stress plot, please indicate if you would \n'
                'like to set a maximum value for the stress legend. \n'
                '(*) Notes: \n'
                '     a) You can select multiple options. \n'
                '     b) Screenshot(s) of the selected option(s) will be '
                'saved. \n'
                '     c) The preferred option will be displayed in between \n'
                'each iteration. \n'
                '     d) Secondary options will only be displayed for the \n'
                'time required to plot and save the screenshot.\n'
                '     e) All elements with a stress larger than the \n'
                'maximum legend scale value will be painted in the same \n'
                'color. An empty box will set the maximum stress observed \n'
                'as the upper limit of the legend.\n\n'
            )

            exception_message = (
                'Invalid input in "Plot options". \n'
                'Please consider the following requirements: \n'
                '- The answer to questions 1 through 5 should be either \n'
                '  0 or 1. \n'
                '- The preferred plot should be identified by its \n'
                '  question number (1 through 5).\n'
                '- The preferred plot must be one of the requested plots.'
                '\n\n'
            )

            temp_outputs = getInputs(pars, dialogTitle = 'Plot options',
                                     label = Label)

            try:
                (
                    plot_density,
                    plot_stress,
                    plot_stress_p,
                    plot_stress_a,
                    plot_stress_a_p,
                    preferred_plot
                ) = [int(float(k)) if k not in [None, ''] else 0
                     for k in temp_outputs[:-1]]

                temp_value = temp_outputs[-1]
                max_stress_legend = (float(temp_value)
                                     if temp_value not in [None, ''] else None)
            except:
                raise Exception(exception_message)

            if (
```

```
                plot_density not in [0,1]
                or plot_stress not in [0,1]
                or plot_stress_p not in [0,1]
                or plot_stress_a not in [0,1]
                or plot_stress_a_p not in [0,1]
                or preferred_plot not in range(1,6)
                or ( preferred_plot == 1 and plot_density not in [1]
                    or preferred_plot == 2 and plot_stress not in [1]
                    or preferred_plot == 3 and plot_stress_p not in [1]
                    or preferred_plot == 4 and plot_stress_a not in [1]
                    or preferred_plot == 5 and plot_stress_a_p not in [1]
                )
            ):
                raise Exception(exception_message)

            plot_density = True if plot_density == 1 else False
            plot_stress = True if plot_stress == 1 else False
            plot_stress_p = True if plot_stress_p == 1 else False
            plot_stress_a = True if plot_stress_a == 1 else False
            plot_stress_a_p = True if plot_stress_a_p == 1 else False

        else:
            plot_density = True
            plot_stress = None
            plot_stress_p = None
            plot_stress_a = None
            plot_stress_a_p = None
            preferred_plot = 1
            max_stress_legend = None

        user_inputs = [material_constraint, opt_method, nonlinearities,
                       target_material, evol_ratio, xe_min, rmax,
                       filter_sensitivities, filter_densities, p,
                       initial_density, move_limit, consider_frozen_region,
                       consider_neighbouring_region, s_max, qi, qf,
                       p_norm_stress, stress_sensitivity, plot_density,
                       plot_stress, plot_stress_p, plot_stress_a,
                       plot_stress_a_p, preferred_plot, max_stress_legend,
                       algorithm]

        return user_inputs

    def return_inputs(self, model_inputs, user_inputs):
        """ Return input method
        Returns several variables defining the information input by the user
        in the pop-up boxes.

        Due to the large number of inputs, the variables are generated through
        the Global command instead of the Python return function, as a means to
        maintain the code organized and the main section cleaner.

        Inputs:
        -------
        - model_inputs (list): list of model inputs obtained from the
          'model_information' method of the class 'ParameterInput'.
        - user_inputs (list): list of the user inputs obtained from the
          'problem_statement' method of the class 'ParameterInput'.

        (Global) Outputs:
        -----------------
        - Mdb (Mdb): model database from ABAQUS.
        - CAE_NAME (str): string with the name of the ABAQUS CAE file.
        - MODEL_NAME (str): string with the name of the ABAQUS model.
        - PART_NAME (str): string with the name of the ABAQUS part to be
          optimized.
        - MATERIAL_NAME (str): string with the name of the ABAQUS material to
          be considered.
        - SECTION_NAME (str): string with the name of the ABAQUS material
          section to be considered.

        - MESH_UNIFORMITY (int): variable defining if the mesh is uniform or
          not (Yes=1/No=0).
        - N_DOMAINS (int): number of job domains to be considered in the FEA.
        - N_CPUS (int): number of CPUs to be used in the execution of the FEA.
```

```
        - LAST_FRAME (int): variable defining if only the results of the last
          frame should be considered or not (only last frame = 1 /
          all frames = 0).

        - MATERIAL_CONSTRAINT (int): variable defining if the material
          constraint is imposed on the volume or mass of the model
          (Mass=0/Vol=1).
        - OPT_METHOD (int): variable defining the optimization method to be
          used (Optimality criteria = 0 / Method of Moving Asymptotes = 1).
        - NONLINEARITIES (boolean): Indicates if the problem considers
          geometrical nonlinearities (True) or not (False).

        - TARGET_MATERIAL (float): ratio between the target volume or mass and
          the volume or mass of a full density design.
        - EVOL_RATIO (float): ratio at which the material constraint is imposed
          during each iteration. Ex: if set to 0.05, the material constraint
          starts at 1.0 (no constraint imposed) and is decreased by 0.05 each
          iteration until the TARGET_MATERIAL is reached. If set to 1.0, the
          constraint is always constant and equal to the TARGET_MATERIAL
          value.
        - XE_MIN (float): minimum density allowed for the element. I.e.
          minimum value allowed for the design variables.
        - RMAX (float): maximum radius of the filter, starting at the center of
          each element. Note that the filter only includes elements FULLY
          WITHIN the radius RMAX around the center of the element.
        - FILTER_SENSITIVITIES (boolean): indicates if the blurring filter
          should be applied to the sensitivities determined during the
          optimization process.
        - FILTER_DENSITIES (boolean): indicates if the blurring filter
          should be applied to the design densities determined during the
          optimization process.
        - P (float): SIMP penalty factor.
        - DP (int): number of decimal places to be considered in the material
          interpolation. By definition, equal to the number of decimal places
          in XE_MIN.

        - INITIAL_DENSITY (float): value of the initial design density to be
          assigned to each element in the topology optimization problem.
          If set to 0, the program will assign a random density value
          (between 0 and 1) to each element. If set to 0.0, will generate an
          initial case with a random density for each element. Otherwise,
          all elements will start with the design density value specified.
        - MOVE_LIMIT (float): maximum allowable change for the design
          variables. Not applicable to the SciPy optimizers.
        - CONSIDER_FROZEN_REGION (int): variable defining if the filter should
          consider the influence of the elements in the frozen region
          (Yes=1/No=0).
        - CONSIDER_NEIGHBOURING_REGION (int): variable defining if the filter
          should consider the influence of the elements in the neighbouring
          region (Yes=1/No=0).

        - S_MAX (float): maximum value of the stress constraint imposed. Set to
          None for stress unconstrained problems.
        - Qi (float): initial (or minimum) value of the exponential of the
          P-norm stress approximation function. Although usually named "P" in
          the literature, the letter "Q" was adopted to avoid confusion with
          the SIMP penalty factor, which is also usually named "P" in the
          literature.
        - QF (float): final (or maximum) value of the exponential of the P-norm
          stress approximation function. Although usually named "P" in the
          literature, the letter "Q" was adopted to avoid confusion with the
          SIMP penalty factor, which is also usually named "P" in the
          literature.

        - PLOT_DENSITY, PLOT_STRESS, PLOT_STRESS_P, PLOT_STRESS_A,
          PLOT_STRESS_A_P (boolean): variables defining the the user requested
          the plot of the density, stress, or amplified stress distribution
          (raised to the P-norm exponent or not) in the model during the
          optimization process.
        - PREFERRED_PLOT (int): defines which plot should be printed for the
          largest period of time. Only applicable when requesting multiple
          plots.
        - MAX_STRESS_LEGEND (float): defines the maximum stress value of the
          scale used as a legend in the stress plots.
```

```
        - ALGORITHM (str): name of the SciPy optimization algorithm to be used.
          Only used when using the SciPy optimization module.

        - RESTART (boolean): indicates if the user is trying to restart an
          optimization process (True) or not (False).
        """
        global Mdb, CAE_NAME, MODEL_NAME, PART_NAME, MATERIAL_NAME, \
                SECTION_NAME
        Mdb = model_inputs[0]
        CAE_NAME = model_inputs[1]
        MODEL_NAME = model_inputs[2]
        PART_NAME = model_inputs[3]
        MATERIAL_NAME = model_inputs[4]
        SECTION_NAME = model_inputs[5]

        global MESH_UNIFORMITY, N_DOMAINS, N_CPUS, LAST_FRAME
        MESH_UNIFORMITY = model_inputs[6]
        N_DOMAINS = model_inputs[7]
        N_CPUS = model_inputs[8]
        LAST_FRAME = model_inputs[9]

        global MATERIAL_CONSTRAINT, OPT_METHOD, NONLINEARITIES
        MATERIAL_CONSTRAINT = user_inputs[0]
        OPT_METHOD = user_inputs[1]
        NONLINEARITIES = user_inputs[2]

        global TARGET_MATERIAL, EVOL_RATIO, XE_MIN, RMAX, \
                FILTER_SENSITIVITIES, FILTER_DENSITIES, P, DP
        TARGET_MATERIAL = user_inputs[3]
        EVOL_RATIO = user_inputs[4]
        XE_MIN = user_inputs[5]
        RMAX = user_inputs[6]
        FILTER_SENSITIVITIES = user_inputs[7]
        FILTER_DENSITIES = user_inputs[8]
        P = user_inputs[9]
        DP = str(XE_MIN)[::-1].find('.')

        global INITIAL_DENSITY, MOVE_LIMIT, CONSIDER_FROZEN_REGION,\
                CONSIDER_NEIGHBOURING_REGION
        INITIAL_DENSITY = user_inputs[10]
        MOVE_LIMIT = user_inputs[11]
        CONSIDER_FROZEN_REGION = user_inputs[12]
        CONSIDER_NEIGHBOURING_REGION = user_inputs[13]

        global S_MAX, Qi, QF
        S_MAX = user_inputs[14]
        Qi = user_inputs[15]
        QF = user_inputs[16]

        global P_norm_stress, Stress_sensitivity
        P_norm_stress = user_inputs[17]
        Stress_sensitivity = user_inputs[18]

        global PLOT_DENSITY, PLOT_STRESS, PLOT_STRESS_P, PLOT_STRESS_A,\
                PLOT_STRESS_A_P, PREFERRED_PLOT, MAX_STRESS_LEGEND
        PLOT_DENSITY = user_inputs[19]
        PLOT_STRESS = user_inputs[20]
        PLOT_STRESS_P = user_inputs[21]
        PLOT_STRESS_A = user_inputs[22]
        PLOT_STRESS_A_P = user_inputs[23]
        PREFERRED_PLOT = user_inputs[24]
        MAX_STRESS_LEGEND = user_inputs[25]

        global ALGORITHM
        ALGORITHM = user_inputs[26]

        global RESTART
        RESTART = False


class EditableDomain:
    """ Editable domain class
```

```
    The present class is responsible for identifying the domain of the model
    to be optimized (elements and nodes).

    Attributes:
    -----------
    - mdb (Mdb): ABAQUS model database.
    - model_name (str): Name of the ABAQUS model.
    - part_name (str): Name of the ABAQUS part to be optimized.
    - part (Part): ABAQUS part to be optimized.
    - consider_frozen_region (int): variable defining if the filter should
      consider the influence of the elements in the frozen region (Yes=1/No=0).
    - consider_neighbouring_region (int): variable defining if the filter
      should consider the influence of the elements in the neighbouring
      region (Yes=1/No=0).

    Method:
    -------
    - identify_domain(): identifies the nodes and elements that belong to the
      editable domain of the problem, considering the interaction with frozen
      and neighbouring regions.
    """
    def __init__(
            self, mdb, model_name, part_name, consider_frozen_region,
            consider_neighbouring_region
        ):
        self.mdb = mdb
        self.model_name = model_name
        self.part_name = part_name
        self.part = self.mdb.models[self.model_name].parts[self.part_name]
        self.consider_frozen_region = consider_frozen_region
        self.consider_neighbouring_region = consider_neighbouring_region

    def identify_domain(self):
        """ Identify domain method

        Checks the type of part considered in the numerical model (2D or 3D).

        Identifies the nodes and elements to be considered, accounting for the
        possible existance of frozen of neighbouring regions.

        Outputs:
        --------
        - elmts (MeshElementArray): array with the elements included in the
          editable region of the topology optimization problem.
        - nodes (MeshNodeArray): array with the nodes of the ABAQUS part
          considered in the topology optimization problem.
        - all_elmts (MeshElementArray): array with all elements that belong
          to the part considered in the topology optimization problem.
        - planar (int): variable identifying the type of part considered (2D or
          3D).
        """
        if self.part.space == THREE_D:
            planar = 0
        elif self.part.space == TWO_D_PLANAR:
            planar = 1
        elif self.part.space == AXISYMMETRIC:
            raise Exception("The code implementation the provided does not "
                            "support Axisymmetric problems.")
        else:
            raise Exception("Unexpected value for self.part.space")

        # Exclude frozen areas of the model if they exist.
        # otherwise, do nothing and consider all elements of the self.part.
        if 'editable_region' in self.part.sets.keys():
            elmts = self.part.sets['editable_region'].elements
            all_elmts = self.part.elements
            nodes = self.part.nodes
            print "Frozen areas excluded. \n"
            if len(elmts) == 0:
                raise Exception(
                    "All elements are frozen. There are no elements \n"
                    "available for the topology optimization. \n"
                )
```

```
        else:
            print (
                "No frozen areas detected. \n"
               "Casual reminder: \n"
               "-The name of the set 'editable_region' is case sensitive.\n "
               "-The algorithm will not detect this set if its name is \n"
               " not spelled exactly as indicated above. \n"
            )
            elmts, nodes = self.part.elements, self.part.nodes
            all_elmts = self.part.elements

        # If the frozen regions are not considered the algorithm checks if
        # it should consider the neighbouring region of the editable elements.
        # If the consideration of a neighbouring region was requested but the
        # region was not found, the algorithm prints an error message.
        if self.consider_frozen_region == 0:
            if self.consider_neighbouring_region == 0:
                all_elmts = elmts
            else:
                if 'neighbouring_region' not in self.part.sets:
                    raise Exception(
                        "The information input in the 'Topology optimization' "
                        "tab suggest the intention of considering a "
                        "'neighbouring_region', which was not found in Part "
                        "{} of Model {}. \n"
                        "Please, either create the set selecting the "
                        "'neighouring_region' or change the information input "
                        "in the 'Topology Optimization' tab when asked "
                        "'Consider neighbouring region?'. \n"
                        "Furthermore, the user is reminded that the name of "
                        "the set 'neighbouring_region' is case sensitive. \n"
                        "The algorithm will not detect this set if its name "
                        "is not spelled exactly as indicated above. \n"
                    ).format(part_name, model_name)

                if 'neighbouring_region' in self.part.sets:
                    all_elmts = (
                        elmts + self.part.sets['neighbouring_region'].elements
                    )

        return elmts, nodes, all_elmts, planar


class VariableGenerator:
    """ Variable generator class

    Due to the large number of inputs, the variables are generated through
    the Global command instead of the Python return function, as a means to
    maintain the code organized and the main section cleaner.

    Attributes:
    -----------
    - initial_density (float): initial design density to be assigned to the
      elements. If set to 0, creates a random density distribution.
    - all_elmts (MeshElementArray): element_array from ABAQUS with all the
      elements in the part.
    - elmts (MeshElementArray): element_array from ABAQUS with the relevant
      elements in the part.
    - xe_min (float): minimum density allowed for the element. I.e. minimum
      value allowed for the design variables.
    - dp (int): number of decimals places to be considered in the
      interpolation. By definition, equal to the number of decimal places
      in xe_min.
    - opt_method (int): variable defining the optimization method to be used.
    - restart (boolean): indicates if the user is trying to restart an
      optimization process (True) or not (False).

    Methods:
    --------
    - create_variables(): wrapper function organizing the creation of
      variables.
    - create_lists(): creates the lists used to store the topology optimization
      data.
    - create_dictionaries(): creates dictionaries used to store the element
```

```
    and node-level data used in the topology optimization process.
  - create_floats(): creates the floats and none variables used in the
    topology optimization process.
  """
  def __init__(
          self, initial_density, all_elmts, elmts, xe_min, dp, opt_method,
          restart
      ):
      self.initial_density = initial_density
      self.all_elmts = all_elmts
      self.elmts = elmts
      self.xe_min = xe_min
      self.dp = dp
      self.opt_method = opt_method
      self.restart = restart

  def create_variables(self):
      """ Create variables method

      Wrapper function organizing the creation of variables.

      If the user is restarting the optimization process, this process is
      skipped to avoid overwritting the information from the previous run.
      """
      if self.restart == False:
          self.create_lists()
          self.create_dictionaries()
          self.create_floats()

      elif self.restart == True:
          pass

      else:
          raise Exception(
              "Unexpected value for attribute 'restart' of class \n"
              "'VariableGenerator'."
          )

  def create_lists(self):
      """ Create lists method
      Returns several lists used to create a record of the relevant variables
      used during the topology optimization process.

      (Global) List Outputs:
      ----------------------
      - Objh: list used to record the values of the objective function.
      - Target_material_history: list used to record the value of the
        material constraint that the algorithm tried to reach in each
        iteration. Note that due to the existance of the EVOL_RATIO
        parameter, it is expectable that the values recorded in this list
        are not always equal to the TARGET_MATERIAL.
      - Current_Material: list with the values of the material constraint
        in either mass or volume ratios.

      For stress dependent problems, the following lists are also
      created:
      - P_norm_history: list used to record the values of the P-norm
        maximum stress approximation.
      - Lam_history: list used to record the Lagrangee multipliers.
      - Fval_history: list used to record the values of the constraints,
        as determined by the MMA function.
      """

      global Objh, Target_material_history, Current_Material
      Objh = []
      Target_material_history = []
      Current_Material = []

      # When solving a stress dependent problems, the following lists are
      # required:
      global P_norm_history, Lam_history, Fval_history
      P_norm_history = []
      Lam_history = []
      Fval_history = []
```

```
    def create_dictionaries(self):
        """ Create dictionaries method

        Returns several dictionaries used to store the values of the design
        variables and compliance sensitivities in the current and up to 2
        previous iterations.

        The dictionaries with the design variables are initiated with the
        initial density (INITIAL_DENSITY) requested by the user. If the
        variable Initial_density is set to 0, it will generate an initial case
        with a random density for each element. Otherwise, all elements will
        start with the design density value specified.

        (Global) Dictionary Outputs:
        ----------------------------
        - Xe: dictionary with the densities (design variables) of each
          relevant element in the model.
        - Editable_xe: dictionary with the densities (design variables) of
          each editable element in the model.
        - Xold1: dictionary with the data of Xe for the previous iteration.
        - Xold2: dictionary with the data of Xe for the second to last
          iteration.
        - Ae: dictionary with the sensitivity of the objective function to
          changes in each design variable.
        - OAe: dictionary with the data of Ae for the last iteration.
        - OAe2: dictionary with the data of Ae for the second to last
          iteration.
        - Xold_temp: auxiliary dictionary used when updating the
          dictionaries with the design variables.
        - Ae_temp: auxiliary dictionary used when updating the
          dictionaries with the sensitivities.
        """
        global Xe, Editable_xe, Xold1, Xold2, Ae, OAe, OAe2, Xold_temp, Ae_temp
        Xe, Editable_xe = {}, {}
        Xold1, Xold2, = {}, {}
        Ae, OAe, OAe2, = {}, {}, {}

        for el in self.all_elmts:
            Ae[el.label] = 0.0
            OAe[el.label] = 0.0
            OAe2[el.label] = 0.0
            Xe[el.label] = 1.0


        if self.initial_density == 0:
            for el in self.elmts:
                x = round(random.uniform(self.xe_min, 1.0), self.dp)
                Editable_xe[el.label], Xe[el.label] = x, x
        else:
            for el in self.elmts:
                Xe[el.label] = self.initial_density
                Editable_xe[el.label] = self.initial_density

        Xold_temp = Editable_xe.copy()
        Ae_temp = Ae.copy()

    def create_floats(self):
        """ Create floats method

        Creates several float variables, as well as None variables which are
        only used in stress dependent problems.

        (Global) Outputs:
        - Iter (int): number of the current iteration.
        - Change (float): variable with the relative difference between the
          objective function of the last 10 iterations. Used to evaluate
          convergence.
        - low (array): array with the minimum design value considered for each
          element, according to the convergence of the MMA.
          Although obtained as an output of the mmasub functionm it is
          initialized as None.
        - Upp (array): array with the maximum design value considered for each
          element, according to the convergence of the MMA.
```

```
          Although obtained as an output of the mmasub functionm it is
          initialized as None.
        """

        global Iter, Change, Low, Upp
        Iter = -1
        Change = 1
        Low, Upp = None, None


#%% Miscelaneous or auxiliary functions.
def average_ae(iteration, ae, oae, oae2):
    """ Average objective derivative function

    Averages the sensitivities of the objective function with the results from
    up to 2 previous iterations to improve convergence and reduce the
    influence of large changes in the design variables.

    Inputs:
    -------
    - iteration (int): number of the current iteration in the topology
      optimization process.
    - ae (dict): dictionary with the sensitivity of the objective function to
      changes in each design variable.
    - oae (dict): dictionary with the values of 'ae' in the previous iteration.
    - oae2 (dict): dictionary with the values of 'ae' two iterations ago.

    Output:
    -------
    - ae (dict): dictionary with the sensitivity of the objective function to
      changes in each design variable, after the averaging process.
    """

    if iteration == 1:
        ae = dict([(k,(ae[k] + oae[k]) /2.0) for k in ae.keys()])

    if iteration > 1:
        ae = dict([(k ,(ae[k] + oae[k] + oae2[k]) / 3.0) for k in ae.keys()])

    return ae


def update_past_info(ae, editable_xe, oae, xold1, oae2, xold2, iteration):
    """ Update past information function

    Updates the variables that store previous values of the design variables,
    the sensitivity of the objective function, and the iteration counter.

    Inputs:
    -------
    - ae (dict): dictionary with the sensitivity of the objective function to
      changes in each design variable.
    - editable_xe (dict): dictionary with the densities (design variables) of
      each editable element in the model.
    - oae, oae2 (dict): equivalent to 'ae' for the last and second to last
      iterations.
    - xold1, xold2 (dict): equivalent to 'editable_xe' for the last and second
      to last iterations.
    - iteration (int): number of the current iteration.

    Outputs:
    --------
    - oae, oae2 (dict): equivalent to 'ae' for the last and second to last
      iterations.
    - xold1, xold2 (dict): equivalent to 'editable_xe' for the last and second
      to last iterations.
    - iteration (int): number of the current iteration.
    """
    ae_temp = ae.copy()
    xold_temp = editable_xe.copy()
    if iteration >= 1:
        oae2 = oae.copy()
        xold2 = xold1.copy()
```

```
    oae = ae_temp.copy()
    xold1 = xold_temp.copy()

    return oae, xold1, oae2, xold2


def evaluate_change(objh, p_norm_history, iteration, opt_method):
    """ Evaluate change function

    Evalutes the change in the objective function for the last 10 iterations.
    If the number of iterations is lower than 10, returns the initial value
    (set to 1.0 by default).

    If the optimization selected is based on the SciPy module, the function
    assumes convergence automatically, since the functions in this module
    have their own convergence criteria implemented.

    Inputs:
    -------
    - objh (list): record with values of the objective function.
  - p_norm_history (list): record with the values of the p-norm
    stress approximation.
    - iteration (int): number of the current iteration.
    - opt_method (int): variable defining the optimization method to be used.

    Output:
    -------
    - change (float): ratio of the change in the objective function.
    """
    if opt_method in [3,5,7]:
        change = 0

    elif opt_method in [0,1,2,4]:
        if iteration > 10:
            num = (sum(objh[iteration-4: iteration+1])
                   -sum(objh[iteration-9: iteration-4])
            )
            denom = sum(objh[iteration-9: iteration-4])
            change=math.fabs(num / denom)
        else:
            change = 1.0

    elif opt_method in [6]:
        if iteration > 10:
            num = (sum(p_norm_history[iteration-4: iteration+1])
                   -sum(p_norm_history[iteration-9: iteration-4])
            )
            denom = sum(p_norm_history[iteration-9: iteration-4])
            change=math.fabs(num / denom)
        else:
            change = 1.0

    else:
        raise Exception(
            "Unexpected value for 'opt_method' in function 'evaluate_change'."
        )

    return change


def remove_files(i, name, del_odb = True):
    """Remove Files function

    Removes all ABAQUS generated files related to a given iteration (i) of
    the optimization algorithm, except the Output Database file (.odb).

    Inputs:
    -------
    - i (int): number of the current iteration.
    - name (str): name of the ABAQUS job.
    - del_odb (bool): indicates if the odb file should be removed after each
      iteration.
    """
```

```
    file_list = ['.com','.inp','.dat','.msg','.sim','.prt','.sta','.log',
                 '.mdl','.pac','.res','.sel','.stt','.abq','.ipm','.lck']

    if del_odb == True:
        file_list.append('.odb')

    abaqus_rpy = ['abaqus.rpy','abaqus.rpy.1','abaqus.rpy.2']

    #Tries to remove the files listed, if the files exist.
    for abaqus_file in file_list:
        try:
            os.remove(name+str(i)+abaqus_file)
        except:
            pass
    for rpy_file in abaqus_rpy:
        try:
            os.remove(rpy_file)
        except:
            pass


#%% Main program.
if __name__ == '__main__':
    # Create a ParameterInput object to recieve the user inputs, model
    # information, problem statement, and return the necessary global
    # variables accordingly.
    # Due to the large number of variables created, the output is fully
    # described in the class description (code lines XXXXXXX).
    # If the user is restarting an optimization process, this step is skipped,
    # as the necessary information was recorded in the data save file.
    if 'RESTART' in globals() and RESTART == True:
        pass
    else:
        Get_Inputs = ParameterInput()
        MODEL_INPUTS = Get_Inputs.model_information()
        USER_INPUTS = Get_Inputs.problem_statement()
        Get_Inputs.return_inputs(MODEL_INPUTS, USER_INPUTS)

    # Identify the region to be optimized, relevant elements, nodes, and type
    # of geometry.
    Editable_domain = EditableDomain(
        Mdb, MODEL_NAME, PART_NAME, CONSIDER_FROZEN_REGION,
        CONSIDER_NEIGHBOURING_REGION
    )
    ELMTS, NODES, ALL_ELMTS, PLANAR = Editable_domain.identify_domain()

    # Create the necessary global variables to store optimization data.
    # Due to the large number of variables created, the output is fully
    # described in the class description (code lines XXXXXXX).
    Var_generator = VariableGenerator(
        INITIAL_DENSITY, ALL_ELMTS, ELMTS, XE_MIN, DP, OPT_METHOD, RESTART
    )
    Var_generator.create_variables()

    # Formats the ABAQUS model:
    # - Creates the materials and sections for the possible design variables;
    # - Extracts the user-defined information (existing sets);
    # - Assigns the materials created to the ABAQUS model elements.
    Model_preparation = ModelPreparation(
        Mdb, MODEL_NAME, NONLINEARITIES, PART_NAME, MATERIAL_NAME,
        SECTION_NAME, ELMTS, ALL_ELMTS, XE_MIN, OPT_METHOD, DP, P
    )
    Model_preparation.format_model()
    (
        ELEMENT_TYPE,
        SET_LIST,
        ACTIVE_LOADS,
        ACTIVE_BC,
        NODE_COORDINATES,
        NODE_NORMAL_VECTOR,
    ) = Model_preparation.get_model_information()
    Model_preparation.property_update(Editable_xe)

    # Create a blurring filter. If not requested, returns None.
```

```
Filter = init_filter(
    RMAX, ELMTS, ALL_ELMTS, NODES, Mdb, MODEL_NAME, PART_NAME
)

# Determine the material (mass or volume) constraint sensitivity and its
# value.
MAT_CONST_SENSITIVITIES, ELMT_VOLUME = material_constraint_sensitivity(
    Mdb,MATERIAL_CONSTRAINT,MESH_UNIFORMITY,OPT_METHOD, MODEL_NAME,
    PART_NAME, Density
)
Material_const = MaterialConstraint(
    TARGET_MATERIAL, EVOL_RATIO, MAT_CONST_SENSITIVITIES
)

# Format color mapping set by Abaqus in order to display the element
# densities and their changes.
Set_display = SetDisplay(
    Mdb, MODEL_NAME, PART_NAME, SET_LIST, XE_MIN, DP, OPT_METHOD,
    PLOT_DENSITY, PLOT_STRESS, PLOT_STRESS_P, PLOT_STRESS_A,
    PLOT_STRESS_A_P, PREFERRED_PLOT, MAX_STRESS_LEGEND
)
Set_display.prepare_density_display()

# Creates the classes that submit the State and Adjoint models in ABAQUS.
Abaqus_FEA = AbaqusFEA(
    Iter, Mdb, MODEL_NAME, PART_NAME, Ae, P, ELEMENT_TYPE, LAST_FRAME,
    N_DOMAINS, N_CPUS, OPT_METHOD, NODE_NORMAL_VECTOR, NONLINEARITIES
)
Adjoint_Model = init_AdjointModel(
    Mdb, MODEL_NAME, PART_NAME, MATERIAL_NAME, SECTION_NAME, NODES, ELMTS,
    P, PLANAR, ELEMENT_TYPE, ELMT_VOLUME, NODE_NORMAL_VECTOR, OPT_METHOD,
    N_DOMAINS, N_CPUS, LAST_FRAME
)

# Creates a class that manages the use of the optimization functions
# available in the SciPy module. If not requested, returns None.
Scipy_optimizer = init_scipy_optimizer(
    ALGORITHM, OPT_METHOD, Editable_xe, Xe, XE_MIN, DP, RMAX,
    FILTER_DENSITIES, FILTER_SENSITIVITIES, MAT_CONST_SENSITIVITIES,
    Target_material_history, Model_preparation, Filter, Abaqus_FEA,
    Adjoint_Model, Qi, S_MAX, ACTIVE_BC, ACTIVE_LOADS, Iter, Set_display,
    NODE_COORDINATES, Objh, P_norm_history
)

while Qi <= QF:

    Min_iter = 0
    while Change > 0.001 or Min_iter < 10:
        Min_iter += 1
        Iter += 1

        # Update the value of the material constraint and record the
        # value.
        Current_Material, Target_material_history = (
            Material_const.update_constraint(Current_Material,
                                             Target_material_history,
                                             Editable_xe)
        )

        # Execute the FEA and extract relevant variables.
        # When using SciPy, this step is skipped, as the solver will call
        # this function on its own.
        if OPT_METHOD not in [3, 5, 7]:
            (
                Obj,
                Ae,
                State_strain,
                Node_displacement,
                Node_rotation,
                Local_coord_sys,
            ) = Abaqus_FEA.run_simulation(Iter, Xe)

            # Store the value of the objective function.
            Objh.append(Obj)
```

```
        # Filter the sensitivities of the objective function.
        if RMAX > 0 and FILTER_SENSITIVITIES == True:
            Ae = Filter.filter_function(Ae, Editable_xe.keys())

    # Selection of the optimization solver:
    # 0 - Compliance minimization with discrete Optimality Criteria.
    if OPT_METHOD == 0:
        # Average the sensitivities of the objective function with the
        # results from up to 2 previous iterations to improve
        # convergence and reduce the influence of large changes in
        # the design variables.
        Ae = average_ae(Iter, Ae, OAe, OAe2)

        # Use the selected algorithm to update the design variables.
        Editable_xe, Xe = oc_discrete(
            Editable_xe, Xe, Ae, P, Target_material_history[-1],
            MAT_CONST_SENSITIVITIES, XE_MIN
        )

    # 1 - Compliance minimization with continuous Optimality Criteria.
    elif OPT_METHOD == 1:
        #Average the sensitivities of the objective function with the
        #results from up to 2 previous iterations to improve
        #convergence and reduce the influence of large changes in
        #the design variables.
        Ae = average_ae(Iter, Ae, OAe, OAe2)

        #Use the selected algorithm to update the design variables.
        Editable_xe, Xe = oc_continuous(
            Editable_xe, Xe, MOVE_LIMIT, Ae, P,
            Target_material_history[-1], MAT_CONST_SENSITIVITIES,
            XE_MIN, DP
        )

    # 2 - Compliance minimization with MMA.
    elif OPT_METHOD == 2:
        #Use the selected algorithm to update the design variables.
        Editable_xe, Xe, Low, Upp, Lam, Fval, Ymma, Zmma = mma(
            Editable_xe, Xe, MOVE_LIMIT, Ae, P, XE_MIN,
            Target_material_history[-1], MAT_CONST_SENSITIVITIES,
            OPT_METHOD, DP, Objh, Iter, Xold1, Xold2, Low, Upp
        )

    # 3, 5, 7 - Compliance minimization, stress constrained compliance
    # minimization, or stress minimization with SciPy.
    elif OPT_METHOD in [3, 5, 7]:
        Scipy_optimizer.update_attributes(
            Editable_xe, Xe, Target_material_history, Current_Material,
            Qi, Iter
        )

        Editable_xe, Xe = Scipy_optimizer.call_solver(
            Editable_xe, Xe
        )

        Objh, P_norm_history, Current_Material, Iter = (
            Scipy_optimizer.return_record()
        )

        # Imposes the convergence criteria of the SciPy optimizer.
        Change = 0.0001
        Min_iter = 10

    # 4, 6 - Stress dependent optimization with MMA.
    elif OPT_METHOD in [4, 6]:

        # Run adjoint model and extract the adjoint strains.
        Adjoint_strain = Adjoint_Model.run_adjoint_simulation(
            Node_displacement, Xe, Node_rotation, NODE_COORDINATES,
            Local_coord_sys, Qi, ACTIVE_BC, ACTIVE_LOADS, Iter
        )

        # Determine the stress sensitivity.
```

```
        Elmt_stress_sensitivity = Adjoint_Model.stress_sensitivity(
            Xe, Qi, State_strain, Adjoint_strain
        )

        if RMAX > 0 and FILTER_SENSITIVITIES == True:
            Elmt_stress_sensitivity = Filter.filter_function(
                Elmt_stress_sensitivity, Editable_xe.keys()
            )

        # Determine the p-norm approximation of the maximum Von-Mises
        # stress.
        P_norm_stress = p_norm_approximation(
            Adjoint_Model.stress_vector_int,
            Adjoint_Model.inv_int_p,
            Qi,
            Adjoint_Model.multiply_VM_matrix,
        )

        # Store the value of the p-norm stress approximation.
        P_norm_history.append(float(P_norm_stress))

        # Stress constrained compliance minimization with MMA.
        if OPT_METHOD == 4:
            # Determine the value of the stress constraint.
            Stress_constraint = stress_constraint_evaluation(
                P_norm_stress,
                S_MAX
            )

            # Use the selected algorithm to update the design variables.
            Editable_xe, Xe, Low, Upp, Lam, Fval, Ymma, Zmma = mma(
                Editable_xe, Xe, MOVE_LIMIT, Ae, P, XE_MIN,
                Target_material_history[-1], MAT_CONST_SENSITIVITIES,
                OPT_METHOD, DP, Objh, Iter, Xold1, Xold2, Low, Upp,
                P_norm_history, Elmt_stress_sensitivity,
                Stress_constraint, S_MAX
            )

        # Stress minimization with MMA.
        elif OPT_METHOD == 6:
            # Use the selected algorithm to update the design variables.
            Editable_xe, Xe, Low, Upp, Lam, Fval, Ymma, Zmma = mma(
                Editable_xe, Xe, MOVE_LIMIT, Elmt_stress_sensitivity,
                P, XE_MIN, Target_material_history[-1],
                MAT_CONST_SENSITIVITIES, OPT_METHOD, DP,
                P_norm_history, Iter, Xold1, Xold2, Low, Upp
            )

        else:
            raise Exception(
                "Unexpected value for 'OPT_METHOD' in the main \n"
                "optimization loop. Value should be either 4 or 6.")

        # Store data obtained from the optimization algorithm.
        Lam_history.append([[float(item)] for item in Lam])
        Fval_history.append([[float(item)] for item in Fval])

    else:
        raise Exception(
            "Unexpected value for the OPT_METHOD variable in the \n"
            "main optimization loop."
        )

    # Filter the design densities, if requested.
    if RMAX > 0 and FILTER_DENSITIES == True:
        Editable_xe, Xe = Filter.filter_densities(
            Editable_xe, Xe, XE_MIN, DP)

    #Make a record of the values obtained in the iteration.
    OAe, Xold1, OAe2, Xold2 = update_past_info(
        Ae, Editable_xe, OAe, Xold1, OAe2, Xold2, Iter
    )

    # Update the material properties according to the new design
```

```
            # variables.
            Model_preparation.property_update(Editable_xe)

            # Plot the new element densities and save a print screen.
            Set_display.update_display(Qi, Iter, Adjoint_Model, Xe)

            # Check convergence after the first 10 iterations.
            Change = evaluate_change(Objh, P_norm_history, Iter, OPT_METHOD)

            # Save a file with the data used in the current iteration.
            save_data(Qi, Iter)

        Qi+=1.0
        Change = 1.0


    # Save and plot results
    save_mdb(Mdb, Current_Material, Objh, CAE_NAME)
    plot_result(Mdb, Set_display)
```