TÉCNICO LISBOA

Topology Optimization of Flat Structures with Adaptive Finite Elements

Diogo Miguel Fael Paraíso

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. José Arnaldo Pereira Leite Miranda Guedes
Prof. Hélder Carriço Rodrigues

Examination Committee

Chairperson: Prof. Filipe Szolnoky Ramos Pinto Cunha
Supervisor: Prof. José Arnaldo Pereira Leite Miranda Guedes
Member of the Committee: Prof. Aurélio Lima Araújo

November 2021

In memory of my grandfather Manuel.

Em memória do meu avô Manuel.

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Firstly, I would like to thank my supervisor, Professor José Miranda Guedes, for allowing me to work with him, and especially, for his patience throughout this last year.

Secondly, to my parents, Cristina and Fernando, for their love, trust, and support, throughout this particularly hard time that was University.

Lastly, to my friends, Guilherme Pinto and Guilherme Neves, with me since the beginning, and Rita, the best person university has given me. A special thank you to you, you have made it all better.

Resumo

A otimização topológica, como um dos campos da otimização estrutural, tem vindo a ganhar ampla utilização como ferramenta de projeto em problemas de todos os campos da engenharia e da indústria, no entanto, a crescente complexidade dos problemas que estão a ser resolvidos está a provocar um aumento do custo computacional necessário para resolvê-los. Os elementos finitos adaptativos ou refinamento de malha adaptativo apresenta-se como uma solução natural para este problema, reduzindo o número de elementos finitos e, portanto, o custo computacional.

Nesta dissertação, a otimização topológica de estruturas planas é abordada com recurso a elementos finitos adaptativos. Para o efeito, um programa em MATLAB que utiliza elementos quadriláteros de 4 nós foi desenvolvido para resolver o problema de otimização de energia mínima (*minimum compliance*), com recurso ao modelo de Material Isotrópico Sólido com Penalização e otimizando com recurso ao método do Critério de Otimalidade.

Para melhor compreensão do algoritmo foi realizado um estudo paramétrico, onde o impacto do critério de refinamento de malha adaptativo e dos parâmetros de otimização foram testados. As soluções ótimas obtidas com recurso ao algoritmo desenvolvido foram depois analisadas e comparadas com as soluções obtidas com recurso a uma malha uniforme comparável. Os resultados demonstram (1) que o algoritmo de refinamento de malha adaptativo produziu um bom comportamento da malha e o refinamento foi limitado às regiões necessárias, (2) produziu resultados adequados com consideravelmente menos elementos finitos, e (3) técnicas de filtragem adicionais precisam de ser estudadas para utilizar a total capacidade do algoritmo.

Palavras-chave: Otimização topológica, Elementos finitos adaptativos, Refinamento de malha adaptativo, Material isotrópico sólido com penalização, Malha não-uniforme.

Abstract

Topology optimization, as a field of structural optimization, has gained extensive use as a tool in design problems across all fields of engineering and industry alike, however, the growing complexity of the problems being solved is making it more computationally expensive to solve them. Adaptive finite elements or adaptive mesh refinement presents itself as a natural solution to this problem, by reducing the number of finite elements and thus, the computational cost.

In this dissertation, the topology optimization of flat structures is approached using an adaptive finite element algorithm. For this purpose, a MATLAB program using quadrilateral 4-node elements was developed to perform a minimum compliance optimization, using the Solid Isotropic Material with Penalization and solving it using the Optimality Criteria method.

To better understand the algorithm, a parametric study was conducted where the impact of the adaptive mesh refinement criteria and the topology optimization parameters was tested. The optimum designs using the developed adaptive mesh refinement program were also analyzed and compared against the design obtained on a comparable uniform mesh. The results demonstrate (1) that the adaptive mesh refinement algorithm produced a good mesh behavior and refinement was limited to the necessary regions, (2) it produced adequate results with considerably fewer finite elements, and (3) further filtering techniques need to be studied in order to better utilize the algorithm.

Keywords: Topology optimization, Adaptive finite elements, Adaptive mesh refinement, Solid isotropic material with penalization, Non-uniform mesh.

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

η	Damping coefficient
ν	Poisson's ratio
Ω	Design domain
ρ	Density

Roman symbols

c	Compliance
E	Young's modulus
F	Load vector
f	Volume fraction
H	Weight factor
K	Stiffness matrix
$\Lambda \setminus \lambda$	Lagrange multiplier
$\mathcal{L}$	Lagrangian function
m	Move limit
p	Penalization power
U	Displacement vector
V	Volume
x	Design variable

Subscripts

0	Initial
ad	Admissible

e Element

i, j, k, l Computational indexes

min Minimum

Superscripts

e Element

+ Upper limit

- Lower limit

mat Material

T Transpose.

Glossary

1D One Dimensional.

2D Two Dimensional.

3D Three Dimensional.

AMR Adaptive Mesh Refinement.

CAD Computer Aided Design.

CPU Central Processing Unit.

FE Finite Element.

FEM Finite Element Method.

MATLAB Programming language and numeric computing environment.

OC Optimality Criteria.

QUAD4 Four-node Quadrilateral.

QUAD8 Eight-node Quadrilateral.

RAM Random Access Memory.

SIMP Solid Isotropic Material with Penalization.

TO Topology Optimization.

Chapter 1

Introduction

Topology optimization at its core, as one of the basic categories of structural optimization, has always been present in the mind of engineers and thinkers alike. The need to find the optimum design of a structure in order to solve a problem is intemporal. Finding that optimal design, before the advancements in computational aided design (CAD), was always a major challenge and based on previous knowledge and experimentation.

The main goal of this dissertation is developing an algorithm that produces optimum designs with less computational resources, while being easy to use and modify, and relying on one of the programming languages most used in education and research, MATLAB. This is to be done through the implementation of a technique known as adaptive finite elements, which adaptively selects elements to be refined before a new optimization cycle and thus reducing the need of a large number of finite elements when computing the problem.

Firstly, the motivation for this dissertation is given in section 1.1, next an overview of topology optimization in section 1.2. Followed by the objectives in section 1.3 and finally a quick outline of this dissertation in section 1.4.

1.1 Motivation

As the field of topology optimization grows and matures, its use is becoming more common, with numerous applications in both academia and industry alike. The advancements and availability of CAD software made it the natural choice for solving problems such as the one of minimizing weight while maintaining (or increasing) stiffness in structures like the body of a car.

In Aeronautics, for example, the use of topology optimization, often combined with manufacturing restrictions, is increasingly used as a mean to save material, weight and fuel. In recent years, it has been used to achieve an optimal design for an aircraft components, like the seat and seat legs [1]; in the design of morphing wing structures [2, 3] and for the design of the landing gear and engine mount [4].

As the complexity of the problems being solved increases, there is also a growing need of more computational capacity for solving them. An algorithm that produces results comparable to traditional

methods, but requiring less computational resources is of particular interest.

1.2 Topic Overview

Topology optimization (TO) as a subject has been researched for many years, dating back to 1988 when was first introduced by Bendsøe and Kikuchi [5]. Numerous methods have been developed since then with the use of the homogenization method [5], Solid Isotropic Material with Penalty (SIMP) [6] and more recent ones, such as the moving morphable components method [7].

The use of the SIMP approach, used to solve the 0-1 material problem, was able to overcome the enormous computational requirements needed to solve the continuum problem and was quickly popularized. Not without its problems, such as the presence of intermediate densities and microstuctures, SIMP continues to be one of the most used and crucial methods of solving optimization problems. This is especially true with the use of additive manufacturing and composite materials [8, 9].

1.3 Objectives

The main purpose of this dissertation is to develop an algorithm for solving the optimization problem while using adaptive finite elements (also called, adaptive mesh refinement, AMR).

To achieve this, two MATLAB codes for topology optimization [10, 11] were considered and used as a template to build the proposed algorithm. The program should be easy to use and modify, require less computational resources when compared to using a uniform mesh, but still produce adequate results. To better understand the influence of the user defined parameters a parametric study will be conducted. Validation of the results against an optimization using a large uniform mesh will also be done.

Following this, a careful study of the results is done in order to understand its limitations and future work is proposed to improve the implementation of the algorithm.

1.4 Thesis Outline

This dissertation is composed of 5 chapters, Introduction, Background, Implementation, Results and Conclusions. The first chapter is this present introduction of the theme and dissertation. The second focuses on the required technical background and literature review, where the topics of topology optimization and adaptive mesh refinement are addressed. The third chapter is composed of a detailed explanation of the developed algorithm, where a brief introduction to the finite element method is made followed by the algorithm implementation. In chapter four the results of the algorithm are presented, starting with a brief description of the problem, followed by a parametric study and validation against different designs. Finally, in chapter 5, a careful conclusion on the results obtained as well as suggestions for future work are presented. In appendix, some results and the MATLAB code can be found.

Chapter 2

Background

This chapter introduces the results of the literature review done as a premise for this dissertation. It provides general knowledge on theory, methods, difficulties and applications of Topology Optimization as one of the basic categories of Structural Optimization, as first introduced by Bendsøe and Kikuchi [5] in 1988 . A brief assessment of Adaptive Mesh Refinement strategies is also provided.

This chapter provides a theoretical summary of Topology Optimization in section 2.1, followed by a brief overview of Adaptive Mesh Refinement in section 2.2.

2.1 Topology Optimization

As previously stated, Topology Optimization is one of the basic categories of Structural Optimization (SO), with the others being Sizing Optimization and Shape Optimization (see Figure 2.1). As such, the

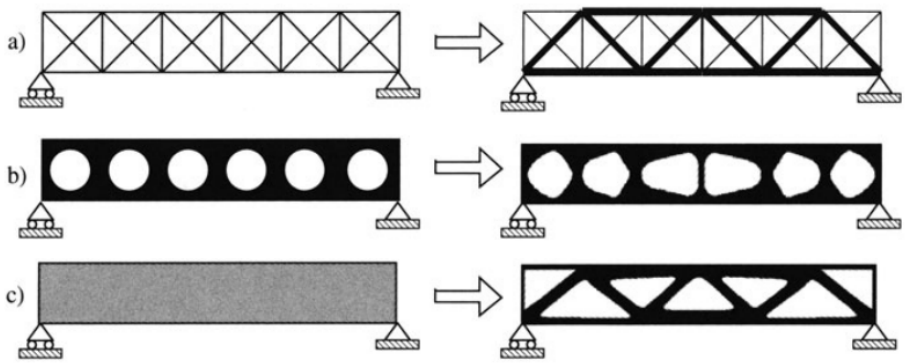


Figure 2.1: Three categories of structural optimization. a) sizing optimization, b) shape optimization and c) topology optimization. The initial problems are shown at the left hand side and the optimal solutions are shown at the right. Reprinted from Bendsøe and Sigmund [12, p.2].

general form of the problem is the same and can be simply described as finding the 'best' (optimized) solution to a design problem. This design is mathematically evaluated using an objective function, $f(x, y)$, which measures the performance of the design and represents the aim of the optimization (maximizing or minimizing f). The objective function is dependent on the design variable (x) -

representing geometry - and the state variable (y) - representing displacement or force - and the general problem takes the form [13]:

$$\text{(SO)} \begin{cases} \text{minimize } f(x,y) \text{ with respect to } x \text{ and } y \\ \text{subject to} \begin{cases} \text{behavioral constraints on } y \\ \text{design constraints on } x \\ \text{equilibrium constraint} \end{cases} \end{cases}$$

The TO problem can be described as a *material distribution problem*, whose solution defines the distribution of material within a defined design domain (see Figure 2.1,c), and minimizes/maximizes the objective function. For the purpose of this literature review, the focus will be on one of the most simple design problems, minimum compliance design, in which the minimum strain energy (maximum global stiffness) is the objective function.

2.1.1 Problem Formulation

As previously stated, the problem at hand will focus on solving the TO problem formulated as a minimum compliance design. Starting by defining the problem as finding the optimal stiffness tensor $E_{ijkl}(x)$ over the domain, Ω, we can introduce the energy bilinear form (internal work) as[5, 12]:

$$a(u,v) = \int_\Omega E_{ijkl}(x)\varepsilon_{ij}(\boldsymbol{u})\varepsilon_{kl}(\boldsymbol{v})d\Omega$$

with linearized strains $\varepsilon_{ij}(u) = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$ and load linear form (external work)

$$l(u) = \int_\Omega f_i u_i d\Omega + \int_{\Gamma_T} t_i u_i ds,$$

where f denotes body forces and t boundary tractions, and u and v are equilibrium and displacement vectors, respectively. We are now able to write the general minimum compliance problem for continuum structures as

$$\begin{aligned} \min_{u\in U,E} \quad & l(u) \\ \text{subject to}: \quad & a_E(u,v) = l(v), \text{ for all } v \in V,\, E \in E_{\text{ad}} \end{aligned} \tag{2.1}$$

where a denotes the energy at the equilibrium u and displacement v, l the load and E_{ad} denotes the admissible stiffness tensors for the design. V represents the space of displacements that are compatible with the Kinematic boundary conditions. Since equation 2.1 needs to be solved for a discretized domain, using finite elements, one needs to rewrite it, taking the form [12],

$$\begin{aligned} \min_{u,E} \quad & \boldsymbol{u}^T \boldsymbol{K} \boldsymbol{u} \\ \text{subject to}: \quad & \boldsymbol{K}(E_e)\boldsymbol{u} = \boldsymbol{f}, \\ & E_e \in E_{\text{ad}} \end{aligned} \tag{2.2}$$

where $\boldsymbol{u}$ are the displacements, $\boldsymbol{f}$ the load vectors and $\boldsymbol{K}$ the stiffness matrix, which depends on E_e, the stiffness tensor for element e.

For our *material distribution problem* we are interested in determining the optimal placement of isotropic material (or remain void) in our discretized design domain, represented by a finite element mesh. This corresponds to viewing the geometry as a collection of black and white 'pixels' that represent a rough description of the optimal continuum structure. This implies that one of the previously stated constraints, that weighs on the admissible stiffness tensors, can be written as

$$E_{ijkl} = 1_{\Omega mat} E^0_{ijkl} \ , \ 1_{\Omega mat} = \begin{cases} 1 \text{ if} x \in \Omega^{mat}, \\ 0 \text{ if} x \in \Omega \setminus \Omega^{mat}, \end{cases} \tag{2.3}$$

where the inequality

$$\int_\Omega 1_{\Omega mat} d\Omega = Vol(\Omega^{mat}) \leq V$$

represents a limit on the volume(V) of material available and Ω^{mat} the optimal subset of material points that respect the admissible stiffness tensors. This means that we have formulated a discrete solution to our problem and only black and white material points (0-1 values) are possible. The solution to this integer problem leads to non-linear binary/integer programming problems which is computationally heavy, since it results in many design variables/functions, limiting its application on large scale problems [9, 12, 14, 15]. Nonetheless, numerous research papers, from various authors, have been published on the TO integer problem (often using a *linear mixed programming* approach) and its application on specific problems [16–20].

A popular way to address the problem is to make use of interpolation models for the material properties so that it depends on a continuous function, this is achieved by modifying the stiffness matrix so that it depends on a continuous function which can be interpreted as the density of the material. This heuristic method of solving the discretized problem introduces a penalization method for intermediate material points (values between 0-1), making them less favorable in terms of stiffness contribution per material used, such that the design tends to a "0-1" solution. One of the first methods was introduced by Bendsøe and Kikuchi [5] using an homogenization method, followed closely by Bendsøe [21] where a method using penalized intermediate densities was introduced, later popularized by Bendsøe and Sigmund [6] with the name Solid Isotropic Material with Penalization; a number of other methods, with and without penalized intermediate densities, have also been studied, namely, Rational Approximation of Material Properties (RAMP) [22] and evolutionary algorithms [23], respectively.

Considering the SIMP method, equation 2.3 can be written as:

$$\begin{gathered} E_{ijkl} = \rho(x)^p E^0_{ijkl} \ , \ p > 1, \\ \int_\Omega \rho(x) d\Omega \leq V; \ \ 0 \ \leq \rho_{min} \leq \rho(x) \leq 1, \ \ x \in \Omega \end{gathered} \tag{2.4}$$

where the density $\rho(x)$ is the design function and E^0_{ijkl} the material properties of an isotropic material.

We can refer to ρ as density since the volume is evaluated as $\int_\Omega \rho(x)d\Omega$, and as previously stated, the function interpolates the material properties between $E_{ijkl}(\rho = \rho_{min}) = E_{min}$ and $E_{ijkl}(\rho = 1) = E^0_{ijkl}$, that directly modifies the stiffness matrix (see equation 2.2). The use of a minimum density (ρ_{min}), slightly larger than zero, is essential to avoid possible singularities when solving the equilibrium problem. For the implementation of the AMR algorithm, and since it provides easier implementation of different filters, such as a density filter, the algorithm will use a modified approach [24], later explained in section 3.2.3. The penalization of intermediate densities for different values of p can be seen in figure 2.2. A small reference to the importance of choosing the penalization power needs to be made without going into too much details. In order to consider the SIMP model as a material model, i.e., with physical significance to the densities provided, the power p must satisfy the following [12]:

$$p \geq max\left\{\frac{2}{1-\nu}, \frac{4}{1+\nu}\right\} \quad \text{in 2D} \tag{2.5}$$

where ν is the Poisson ratio of the base material.

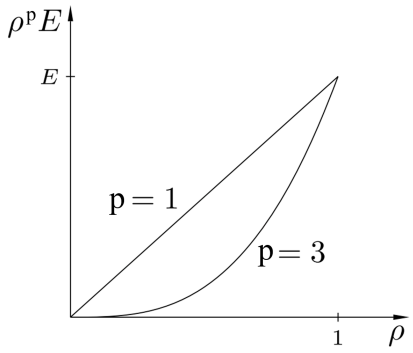


Figure 2.2: Graphical representation of the penalization of intermediate densities for the SIMP model. Adapted from Christensen and Klarbring [13, p.189].

We can now write the entire discretized problem based on the SIMP interpolation where the objective is to minimize compliance[11]

$$\begin{aligned} \min_x : \quad & c(\boldsymbol{x}) = \boldsymbol{U}^T\boldsymbol{K}\boldsymbol{U} = \sum_{e=1}^{N}(x_e)^p \boldsymbol{u}_e^T \boldsymbol{k}_0 \boldsymbol{u}_e \\ \text{subject to} : \quad & \frac{V(\boldsymbol{x})}{V_0} = f \\ & \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F} \\ & \boldsymbol{0} < \boldsymbol{x}_{min} \leq \boldsymbol{x} \leq \boldsymbol{1} \end{aligned} \tag{2.6}$$

where $\boldsymbol{U}$ and $\boldsymbol{F}$ are the global displacement and load vectors, respectively, $\boldsymbol{K}$ is the global stiffness matrix, $\boldsymbol{u}_e$ and $\boldsymbol{k}_e$ are the element displacement vectors and stiffness matrix, respectively, $\boldsymbol{x}$ is the design variables vector (to be interpreted as relative density), $\boldsymbol{x}_{min}$ the minimum relative density, N is the number of elements used in the finite element (FE) mesh, p is the penalization power, $V(\boldsymbol{x})$ and V_0 are the material volume and design domain volume, respectively, and f is the volume fraction (user defined).

For the purpose of this review no further details will be given on any other interpolation method.

2.1.2 Solution methods

Now that we have formulated the entire problem (see equation 2.6) we need to discuss how to solve it through computational methods, with the provision that the problem implies a design variable for each discretized element, thus requiring efficient computational methods to solve.

It is well known that most topology optimization problems are non-convex, which brings the added difficulty of solving them, since usually convergence is attained for local minima, which is difficult for large-scale problems. To solve this issue one can make use of approximate explicit (and convex/concave) subproblems that approximate the original problem and solve these instead.

There are several methods to solve the problem, such as the Optimality Criteria (OC) method, Sequential Linear Programming (SLP), Sequential Quadratic Programming (SQP), the Method of Moving Asymptotes (MMA) and others; with the OC method being the classical and often chosen method, since it can provide an extremely efficient way to solve some optimization problems [12, 13, 25]. For the purpose of this literature review, and since it was the method employed by Sigmund [11], the Optimality Criteria method is the sole focus for this chapter, with brief mentions to other methods.

Optimality Criteria Method

The Optimality Criteria method relies on finding and selecting designs that fulfill the necessary conditions of optimality, and iteratively, select better and better designs in an attempt to find a global minima (or maxima). These conditions of optimality come from the Lagrange function, so for the problem at hand, see equation 2.6, we can write it as [12, 26–28]

$$\mathcal{L} = c(x) - \{KU - F\} - \Lambda[V(x) - V_0 \cdot f)] \tag{2.7}$$

where Λ is the Lagrange multiplier and the side constraints have been concealed, since we know that λ^- and λ^+ are greater or equal to zero, and for the switching conditions $(0 - x_{min})$ and $(x - 1)$, respectively, are zero, meaning that for intermediate densities $(x_{min} < x < 1)$ are also zero. The optimality criteria update scheme can then be formulated as [10–12]:

$$x_e^{new} = \begin{cases} max(0, x_e - m) & \text{if } x_e B_e^{\eta} \leq max(0, x_e - m) \\ min(1, x_e + m) & \text{if } x_e B_e^{\eta} \geq min(1, x_e - m) \\ x_e B_e^{\eta} & otherwise \end{cases} \tag{2.8}$$

where m is a positive move limit, η is a numerical damping coefficient and B_e is obtained from the optimality condition as:

$$B_e = \frac{-\frac{\partial c}{\partial x_e}}{\Lambda \frac{\partial V}{\partial x_e}} \tag{2.9}$$

One should note that the value of x_e^{new} is dependant on the present value of the Lagrange multiplier Λ, meaning it must be chosen so that the volume constraint is satisfied. Since the volume of the updated schemes is a continuous and always decreasing function of Λ we can use a bisection method to determine its value. The value of the move limit m and numerical damping coefficient η must be chosen by experiment and can be adjusted in order to obtain a more rapid and stable convergence of the iteration scheme [12]. To complete the update scheme we must first obtain the derivative of the objective function in respect to the design variable x_e, also called sensitivity of the objective function, which can be written as

$$\frac{\partial c}{\partial x_e} = -px_e^{p-1}(E_0 - E_{min})\boldsymbol{u}_e^T \boldsymbol{k}_0 \boldsymbol{u}_e \tag{2.10}$$

The derivative $\frac{\partial V}{\partial x_e}$ is dependent on the size of the element, for unitary elements this assumes the value one, for non-uniform elements, as is the case here, $\frac{\partial V}{\partial x_e} = a_e$, which is the area of the element.

Although a straightforward method for some problems, like the one at hand, it can be difficult to implement in others, and different methods should be considered when dealing with non-structural constraints, non-self-adjoint problems or when constraints are not physically intuitive.[12]

Computational implementation

With the basic outline of the problem already understood it is possible to implement the algorithm computationally and solve the proposed problem. In a very summarized way, the algorithm must follow these steps (adapted from Bendsøe and Sigmund [12, p.12-13]):

- Pre-processing:
  - Choose a suitable reference domain and choose which part or parts of said domain should be designed or should be left solid/void.
  - Construct a finite element mesh for the structure.
  - Construct the finite element spaces for the independent fields of displacements and the design variables.
- Optimization:
  - Make the initial design (e.g. homogeneous distribution of material).

    Start of iterative part:
  - For the distribution of density at hand compute, using the finite element method, the displacements and strains compatible for the set equilibrium problem (use of the SIMP interpolation method).
  - Compute the compliance of the design (this is how we measure the success of the design and is used to stop the iterations).
  - Compute the update of the density variable based on the optimization scheme (OC method in this case).

  - Repeat the iteration loop.

- Post-processing
  - Interpret the optimal distribution of material as defining a shape.

This is a very generalized implementation that can be used on any type of FE mesh and reference design domain, one could also replace the use of the SIMP interpolation method for another equivalent material interpolation method and/or change the optimization scheme used.

2.1.3 Difficulties

To complete the implementation of the computational calculation of our optimization problem we must discuss complications and possible solutions to these. The two most important issues that arise when using the standard density approach to topology optimization is the appearance of checkerboards patterns and mesh-dependency. These refer to a checkerboard like design of alternating solid and void elements that appear in the final design and to the difference in final design when solving the same optimization problem with different discretizations, respectively.

The use of a filtering technique is then necessary to avoid these and ensure the existence of solutions [12].

Checkerboard Problem

The appearance of a checkerboard like pattern (see figure 2.3) in the design is linked to the artificially high stiffness that the discretization of the continuum causes, specifically due to bad numerical modelling that overestimates the stiffness of checkerboards. It has been shown that in an uniform grid, of square Q4 elements, the stiffness of a checkerboard pattern is comparable to the stiffness of a $\rho = 1/2$ variable thickness sheet, for any applied loads [12]. It is then expected, that for the minimum compliance problem, a checkerboard version as an optimal design.

To avoid this numerical instability a number of prevention techniques can be used, such as, employment of higher order finite elements, use of the patch technique and filtering methods. The use of other restrictions, usually employed for mesh-dependency, can also reduce the appearance of the checkerboard pattern, since, in general, results in 'stronger' constraints, solving the problem by 'compacting' the set of feasible designs. [24, 29].

Figure 2.3: Example of the checkerboard pattern. Reprinted from Sotola et al. [30, p.6].

Mesh-dependency

The problem of mesh-dependency is well known in topology optimization and arrises from the fact that for the problem previously formulated, more holes, without changing the structural volume, will generally increase the efficiency of a given structure, i.e., the use of a different and finer mesh, would result in the appearance of more holes leading to a more efficient structure (see figure 2.4). This is counter intuitive, as one expects that the use of a finer mesh (mesh-refinement) would produce better results, with the same optimal design but with a more detailed description of boundaries, not in a different structure. This mesh-dependency, at more extreme cases, when refining the mesh, can lead to the formation of a fine-scale internal structure lay-out, similar to microstructures [12, 29].

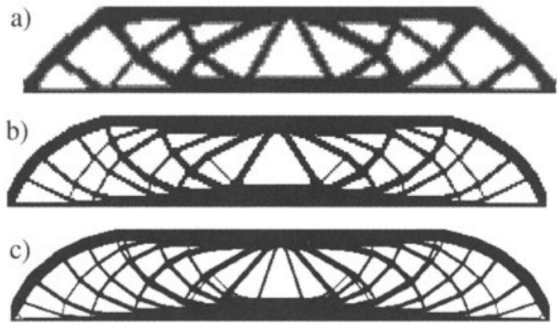


Figure 2.4: Example of the mesh dependence of the optimal topology for a MBB-beam. Solution for a discretization with a) 2700, b) 4800 and c)17200 elements. Reprinted from Bendsøe and Sigmund [12, p.30].

To solve this problem we can deploy the use of design constraints or filtering techniques [31]. The previous being the one chosen by Sigmund [11] (sensitivity filter) and by Andreassen et al. [10] (sensitivity and density filter) on their MATLAB implementation.

The use of a sensitivity filter has proven to ensure mesh independence in a highly efficient way, with the added benefit of easy implementation. This is a purely heuristic filter that produces similar results to local gradient constraints based filters. The filter works by modifying the sensitivity as follows [12]:

$$\widehat{\frac{\partial c}{\partial x_e}} = \frac{1}{x_e \sum_{i \in N_e} H_{ei}} \sum_{i \in N_e} H_{ei} x_i \frac{\partial c}{\partial x_i} \tag{2.11}$$

where N_e is the set of elements i for which the center-to-center distance $\Delta(e,i)$ to element e is smaller than the filter radius r_{min} (see figure 2.5) and H_{ei} is a weight factor defined by $H_{ei} = max(0, r_{min} - \Delta(e,i))$. The modified sensitivities are then used in the optimization scheme (OC method).

The use of a density filter can also solve the problems mentioned above by directly limiting the variation of the densities that appear in the set of admissible stiffness tensor E_{ad} by only admitting filtered densities. Essentially, the filter transforms the original densities x_e into:

$$\tilde{x}_e = \frac{1}{\sum_{i \in N_e} H_{ei}} \sum_{i \in N_e} H_{ei} x_i \tag{2.12}$$

This new way to refer to the densities, $\tilde{x}_e$, emphasizes the difference between the original densities x_e, which should now be referred to as design variables, and filtered densities $\tilde{x}_e$, which refer to physical densities. Equation 2.10 remains valid but is now in respect to the physical densities $\tilde{x}_e$. The sensitivity in respect to the design variables x_j is given by

$$\frac{\partial \psi}{\partial x_j} = \sum_{e \in N_j} \frac{\partial \psi}{\partial \tilde{x}_e} \frac{\partial \tilde{x}_e}{\partial x_j} = \sum_{e \in N_j} \frac{1}{\sum_{i \in N_e} H_{ei}} \frac{\partial \psi}{\partial \tilde{x}_e} \tag{2.13}$$

where ψ is the objective function c or the volume V.

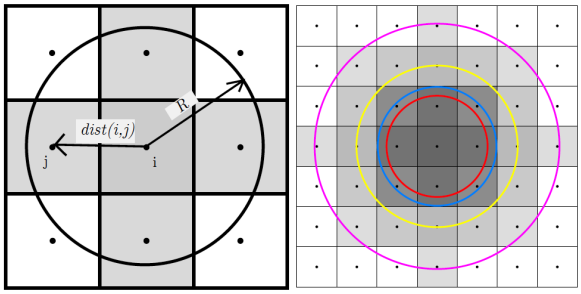


Figure 2.5: Visual representation of the filter radius (R) and the affected elements ($R = 1.2$ element size (ES) in red to $R = 3.0ES$ in purple). Reprinted from Sotola et al. [30, p.9].

Now that every component of a TO computational implementation have been discussed, including the necessary filtering techniques, its useful to schematically present the algorithm of a TO program, see figure 2.6.

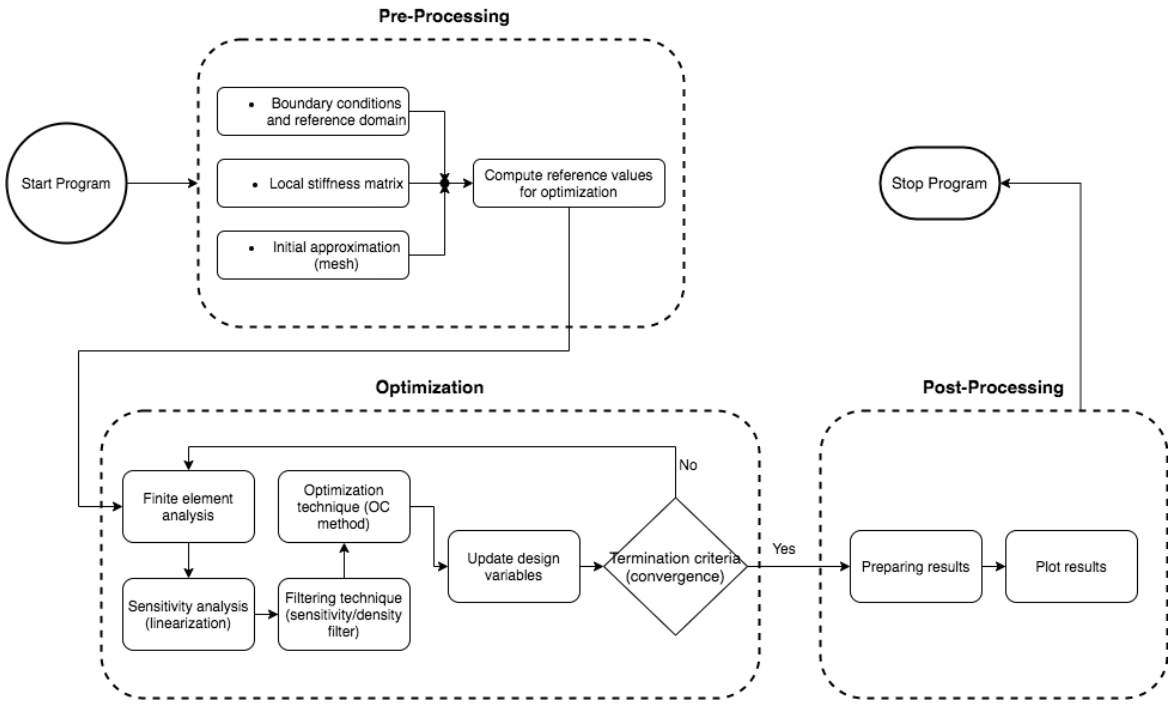


Figure 2.6: Flowchart of a topology optimization program. Adapted from Bendsøe and Sigmund [12], Sotola et al. [30].

2.1.4 Applications

Now that the topology optimization problem, as a whole, is understood, it is relevant to briefly realize the importance of the field of topology optimization. With the ever increasing complexity of structures and problems of other fields, an efficient way to reach an optimum design is of extreme usefulness.

Topology optimization can be used to create lighter and/or stiffer structures, can be used in dynamic problems, such as the optimization of vibrating structures [32, 33] (see example in figure 2.7), or even in domains such as fluid design, where, for example, novel approaches use topology optimization to minimize flow power loss (hydrodynamic drag minimization problem) [34].

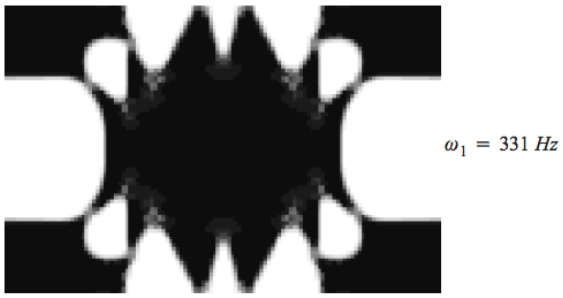


Figure 2.7: Optimized topology for maximization of the fundamental eigenfrequency of a simply supported Mindlin plate with 10% non-structural mass at the center and volume fraction of 60%. Reprinted from Bendsøe and Sigmund [12, p.75].

2.2 Adaptive Mesh Refinement

Topology Optimization, as previously mentioned, can be very computationally heavy, since problems are commonly solved using an uniform mesh with a large number of finite elements so as to achieve high accuracy designs. As the TO problem is being solved regions of solid or void material form that don't require such a fine mesh, however, these regions are unknown *a priori*. So it is only natural that adaptive mesh refinement be researched as a solution to making solving the TO problem more economical. Thus, the purpose of AMR when applied to TO can be described as a way to obtain accurate results, comparable to those obtained when using a uniform mesh, but through the use of considerably less elements, by refining the mesh when and where necessary. This is not without risk, as the problem of mesh-dependency can be exacerbated by the refinement (and subsequent refinement) of elements.

A number of different approaches have been published on how to integrate AMR in TO to improve computational cost and accuracy. The earliest of these approaches was proposed by Maute and Ramm [35], where it follows a strategy of using the previous optimization to compute the elements to refine followed by optimization of the new mesh. Other techniques that follow the same rule (or a similar variation), some with analysis error estimator and mesh quality indicators, have also been proposed by Stainko [36], Costa Jr and Alves [37], Bruggi and Verani [38], Wang et al. [39], Lambe and Czekanski [40] and others. Another approach is to use AMR after each design change in the optimization cycle as proposed by Wang et al. [41], Zhang et al. [42], Salazar de Troya and Tortorelli [43].

The flowchart previously presented (see figure 2.6), now with AMR represented can be seen in figure 2.8.

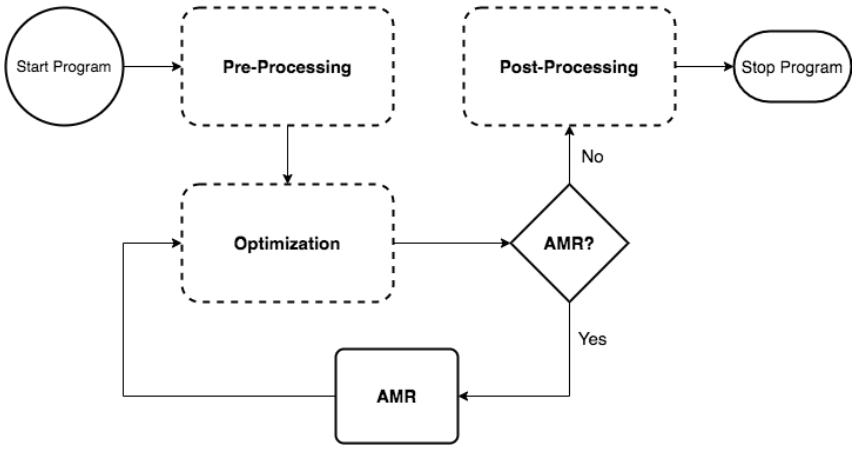


Figure 2.8: Flowchart of a topology optimization program with AMR, following the optimize → refine technique.

2.2.1 Theory, difficulties and refinement criterion

In order to implement a basic adaptive mesh refinement some modifications need to be made to the topology optimization program. Without going into computational implementation details, a brief summary of the theory involved is necessary to better understand the algorithm.

The original MATLAB program by Sigmund [11] uses bilinear quadrilateral elements (Q4 elements) for the necessary discretization, and this is maintained for simplicity. The refinement of the mesh is done by dividing the existing element into four equal elements of the same type, see figure 2.9 a). By refining an existing element we are creating new nodes and some of those will be classified as 'hanging nodes', this designation is used to identify nodes that have a single-level mesh-incompatibility (again, see figure 2.9(b)).

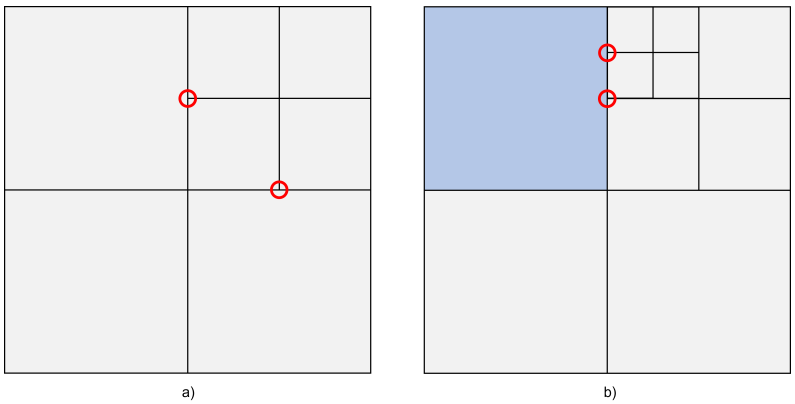


Figure 2.9: Illustration of the mesh refinement and mesh incompatibilities that arise from it. The refinement represented at a) satisfies single-level mesh-incompatibility, while refinement b) does not. Circles in red indicate 'hanging nodes'. The blue tint means the element is marked for refinement.

In order to fully compute the mesh the following constraint needs to be imposed to these nodes

$$u_i = (\frac{u_j + u_k}{2}) \tag{2.14}$$

where the subscript i denotes the 'hanging node' and j and k the nodes at the vertices of the same edge. This constrains the displacement of the 'hanging node' to the displacement of its adjacent nodes. In order to auto impose this for element i, the element stiffness matrix needs to be modified with the following:

$$\begin{aligned}
\boldsymbol{u}_e^T \boldsymbol{k}_e \boldsymbol{u}_e = \\
= \begin{Bmatrix} u_l \\ u_m \\ u_i = (\frac{u_j+u_k}{2}) \\ u_j \end{Bmatrix}^T \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{12} & k_{22} & k_{23} & k_{24} \\ k_{13} & k_{23} & k_{33} & k_{34} \\ k_{14} & k_{24} & k_{34} & k_{44} \end{bmatrix} \begin{Bmatrix} u_l \\ u_m \\ u_i = (\frac{u_j+u_k}{2}) \\ u_j \end{Bmatrix} = \\
= \begin{Bmatrix} u_l \\ u_m \\ u_k \\ u_j \end{Bmatrix}^T \begin{bmatrix} k_{11} & k_{12} & \frac{k_{13}}{2} & (k_{14} + \frac{k_{13}}{2}) \\ k_{12} & k_{22} & \frac{k_{23}}{2} & (k_{24} + \frac{k_{23}}{2}) \\ \frac{k_{13}}{2} & \frac{k_{23}}{2} & \frac{k_{33}}{4} & (\frac{k_{34}}{2} + \frac{k_{33}}{4}) \\ (k_{14} + \frac{k_{13}}{2}) & (k_{24} + \frac{k_{23}}{2}) & (\frac{k_{34}}{2} + \frac{k_{33}}{4}) & (\frac{k_{33}}{4} + k_{44} + k_{34}) \end{bmatrix} \begin{Bmatrix} u_l \\ u_m \\ u_k \\ u_j \end{Bmatrix}
\end{aligned} \tag{2.15}$$

This method for solving the mesh-incompatibility only allows for single-level incompatibilities, for the level-two incompatibilities shown at figure 2.9 b) (the upper red circle shows a node that would be constrained by an already 'hanging node') the explained method does not work. Since an implementation for a level-two incompatibility is non-practical, a simple way to solve this is to force the refinement of the element adjacent to it (again, see figure 2.9 b), element in blue) so as to eliminate the level-two incompatibility.

It is also important to note some changes to the problem formulation previously presented, see equation 2.6. Since the discretized domain now consists of different sized elements, as previously mentioned, the derivative of V in order of the design variable is $\frac{\partial V}{\partial x_e} = a_e$.

The next step in an AMR is to define the refinement criterion. Several methods have been used, namely, selecting the elements that define the design boundary [42] or selecting the elements that define the design [41]. In a material problem this can be approximated by selecting the intermediate densities, i.e., $\rho_{t^-} \leq \rho_e \leq \rho_{t^+}$ where ρ_e is the density of element e and ρ_{t^-}/ρ_{t^+} represent the lower and upper threshold; or by selecting the solid densities, i.e., $\rho_{t^-} \leq \rho_e \leq 1$. This is the way the developed algorithm selects the elements to be refined.

Chapter 3

Implementation

The main purpose of this dissertation is to implement a MATLAB algorithm for topology optimization with adaptive mesh refinement. For this, the MATLAB code written and published by Sigmund [11] in "A 99 line topology optimization code written in Matlab" and Andreassen et al. [10] in "Efficient topology optimization in MATLAB using 88 lines of code" was used as a foundation for the implementation of the adaptive mesh refinement algorithm, with the necessary modifications for a non-uniform finite element mesh. The code produced aims at a simple and easily understandable algorithm that is capable of a more efficient topology optimization by using AMR. The use of MATLAB allows for easy readability and modification without the need for substantial coding knowledge.

Here, the detailed implementation of a finite element method (FEM) using MATLAB for the purpose of topology optimization with adaptive finite elements is explored.

Firstly, a brief mention of the finite element method is made is section 3.1, followed by the detailed implementation of the algorithm in section 3.2.

3.1 Finite element method

The finite element method is a widely used computational method for solving partial differential equations that describe a discretized problem, either be it from structural mechanics or thermodynamics. To solve the problem, the FEM discretizes the problem into small parts, called finite elements. These can be of three basic groups, line elements, planar elements or solid elements, depending on whether the problem is 1D, 2D or 3D, respectively, with each group having different types of elements for different applications. For a 2D application, like the one at hand, two different types of elements can be used, triangular or rectangular elements (see figure 3.1). With each having its advantages and disadvantages, the most common choice is the QUAD4 element (or sometimes just called Q4), which provides good results without the need for the extra computational cost associated with QUAD8 elements. The element used in the development of the MATLAB algorithm is the QUAD4 element.

Like all finite element method implementations two coordinate systems are needed, a local and a

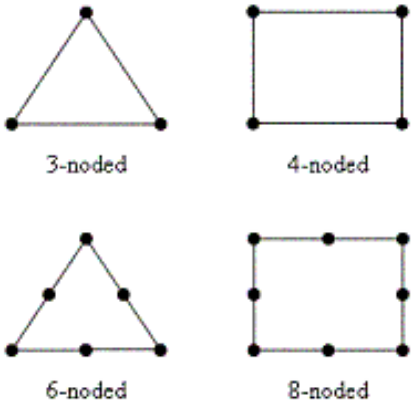
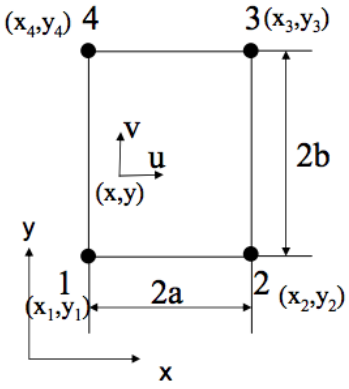
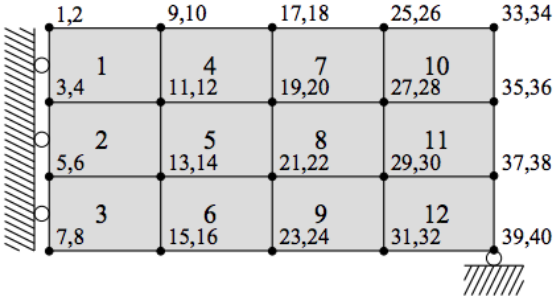


Figure 3.1: Two common element types in 2D finite element analysis. TRI3 elements on the top left, TRI6 on the bottom left, QUAD4 on the top right and QUAD8 on the bottom right.

global coordinate system. The local coordinate system is used for the numerical computation of the element stiffness matrix, whereas the global system is used to define the finite element domain being discretized (see figure 3.2)



(a) Local coordinate system for a QUAD4 element



(b) Global coordinate system for a domain with 12 elements

Figure 3.2: Coordinate systems for the finite element analysis. Reprinted from [44] (left) and from [10] (right).

For a domain discretization using only square QUAD4 elements, the assembly of the global stiffness matrix can be made knowing that the element stiffness matrix for non-hanging nodes is:

$$K^e = \frac{E}{1-\nu^2} * \begin{bmatrix} \frac{3-\nu}{6} & \frac{1+\nu}{8} & -\frac{3+\nu}{8} & -\frac{1+3\nu}{12} & -\frac{3+\nu}{12} & -\frac{1+\nu}{8} & \frac{\nu}{6} & \frac{1-3\nu}{8} \\ & \frac{3-\nu}{6} & \frac{1-3\nu}{8} & \frac{\nu}{6} & -\frac{1+\nu}{8} & -\frac{3+\nu}{12} & -\frac{1+3\nu}{8} & -\frac{3+\nu}{12} \\ & & \frac{3-\nu}{6} & -\frac{1+\nu}{6} & \frac{\nu}{6} & -\frac{1+3\nu}{8} & -\frac{3+\nu}{12} & \frac{1+\nu}{8} \\ & & & \frac{3-\nu}{6} & \frac{1-3\nu}{8} & -\frac{3+\nu}{12} & \frac{1+\nu}{8} & -\frac{3+\nu}{12} \\ & & & & \frac{3-\nu}{6} & \frac{1+\nu}{8} & -\frac{3+\nu}{12} & -\frac{1+3\nu}{8} \\ & & Sym. & & & \frac{3-\nu}{6} & \frac{1-3\nu}{8} & \frac{\nu}{6} \\ & & & & & & \frac{3-\nu}{6} & -\frac{1+\nu}{8} \\ & & & & & & & \frac{3-\nu}{6} \end{bmatrix} \tag{3.1}$$

Then, using the stiffness matrix above and the one shown in equation 2.15, the discretized problem can be solved for the equilibrium described as $[K]\{u\} = \{F\}$, for a given displacement (U) and load (F) vectors.

3.2 Algorithm implementation

This section discusses, in detail, the implementation of the algorithm, starting by giving a brief introduction to the programming language, MATLAB. For easier understanding please refer to appendix B.1 for the entire code.

MATLAB, which is short for "Matrix Laboratory", didn't start as a programming language, but as a simple FORTRAN program to allow simple access to a matrix calculator. The program eventually evolved to the language we know today, with matrix mathematics at his core and developed with engineers and scientists in mind. MATLAB is now a powerful computational tool, widely used in education at universities, in industry and academia for research. This characteristics, combined with its ease of use, made MATLAB a natural programming language to develop the algorithm, further exacerbated by its use for the development of the original topology optimization code, in which this algorithm is based.

The first step was identifying the several sections of code used in the original programs and its functions, in a simple description the sections can be described as a "Finite Element Analysis" and "Optimization Loop", following the scheme previously showed in figure 2.6. For both sections some parts could be transcribed, usually with minor changes, to the new code, with most MATLAB functions remaining the same or being replaced with functions that produce a similar outcome, with the most common change being the implementation of loops, since its no longer possible to use vectors in all situations. Since the use of vectorized loops was one of the main efficiency gains achieved by the authors in "Efficient topology optimization in MATLAB using 88 lines of code" [10, p.2], the present code is expected to be computationally heavier, especially when constructing and refining the mesh. To achieve the desired outcome of adaptive mesh refinement a new section needed to be implemented, a "Refinement Section", this section is intertwined with the "Finite Element Analysis" section and is run as a whole in the second part of the program.

The MATLAB code, being an adaptation, is based on the same optimization strategy that was used on the original codes, described by the authors as a "standard topology optimization code" [10]. This means the algorithm uses the SIMP interpolation method along with the OC method for optimization with a sensitivity or density filter. Thus, the main focus of the code implementation will be on the necessary adaptations and refinement section.

The program can be called using the prompt:

```
fem_ite(nelx,nely,rmin,ite_max,filter_type)
```

where $nelx$ and $nely$ are the number of elements in the horizontal and vertical directions, respectively, $rmin$ is the filter initial size, ite_max is the number of optimization cyles that the program will compute

and output, and $filter\_type$ selects the type of filter (0 for sensitivity filter, 1 for density filter). Other variables, such as volume fraction, which is also the initial (guess) density of each element, and penalization power, which were prompts in the original code, are now defined in the pre-processing section along with other important variables, like the name of the directory where the outputs will be saved, material properties, the refinement criteria bounds (crt_low and crt_high are ρ_{t^-} and ρ_{t^+}, respectively), the frequency of filter radius change and the boundary conditions, all of these can be edited by the user.

The code can be divided into four parts, the initial starting stage (pre-processing), the finite element analysis/refinement, the optimization loop, and the final stage (post-processing). Since the computational code is one of the primary outcomes of this dissertation an extensive and detailed explanation of each part implementation will be given below so that anyone can easily modify it to best fit its needs. A pseudo-code of the implemented algorithm can be found bellow (see table 3.1).

Table 3.1: Pseudo-code : Simplified AMR algorithm

Algorithm 1 Simplified AMR algorithm

procedure $fem_{ite}(nelx, nely, rmin, ite\_max, filter\_type)$ ▷ program prompt
  Pre-processing stage ▷ Starting variables and material properties
  while Infinite loop **do**
    Refinement/FE Stage
    if refine == 'yes' **then**
      procedure REFINEMENT ALGORITHM
        Numbering of new nodes
        Update node connectivity matrix
        Registry of refinement in 'ledger'
        Update coordinate system
        Update possible node constraints (for single-level mesh incompatibility)
        Update notes at design boundary (if applicable)
        Update DoF connectivity matrix
        Update area matrix
        Detecting and solving two-level mesh incompatibility
        Assign old element density to new elements
        Constraint 'hanging nodes' ▷ use of node constraints matrix
        Compute $[KE]$ for constrained elements and then $[K]$
      end procedure
    elseInitial Iteration
      Uniform mesh solution
    end if
    procedure OPTIMIZATION ALGORITHM
      TO algorithm with small modifications to for non-uniform mesh
    end procedure
    Post-processing stage ▷ Plots and Outputs
    if ite_{max} == 'Inputted maximum number of iterations' **then**
      break
    end if
    AMR Criterion
  end while
end procedure

3.2.1 Pre-processing – Variables and Material Properties (lines 2-64)

As previously mentioned, the pre-processing stage houses the initialization of important variables, such as the reference design domain, the initial finite element mesh and material properties, and this algorithm is no exception. A summary of the pre-processing stage can be found in figure 3.3.

The program starts by defining starting variables, such as the penalization power ($penal$), volume fraction ($volfrac$), frequency of filter radius change (div_r), the lower and upper bounds of the refinement criteria (crt_low and crt_high, respectively) and the name used for the directory where the outputs will be stored, which is immediately created. The material properties are also immediately defined, $E0$ is the Young's modulus, $Emin$ is the artificial Young's modulus for the void regions, nu is the Poisson's ratio and the element stiffness matrix for a square element, KE, is calculated using equation 3.1.

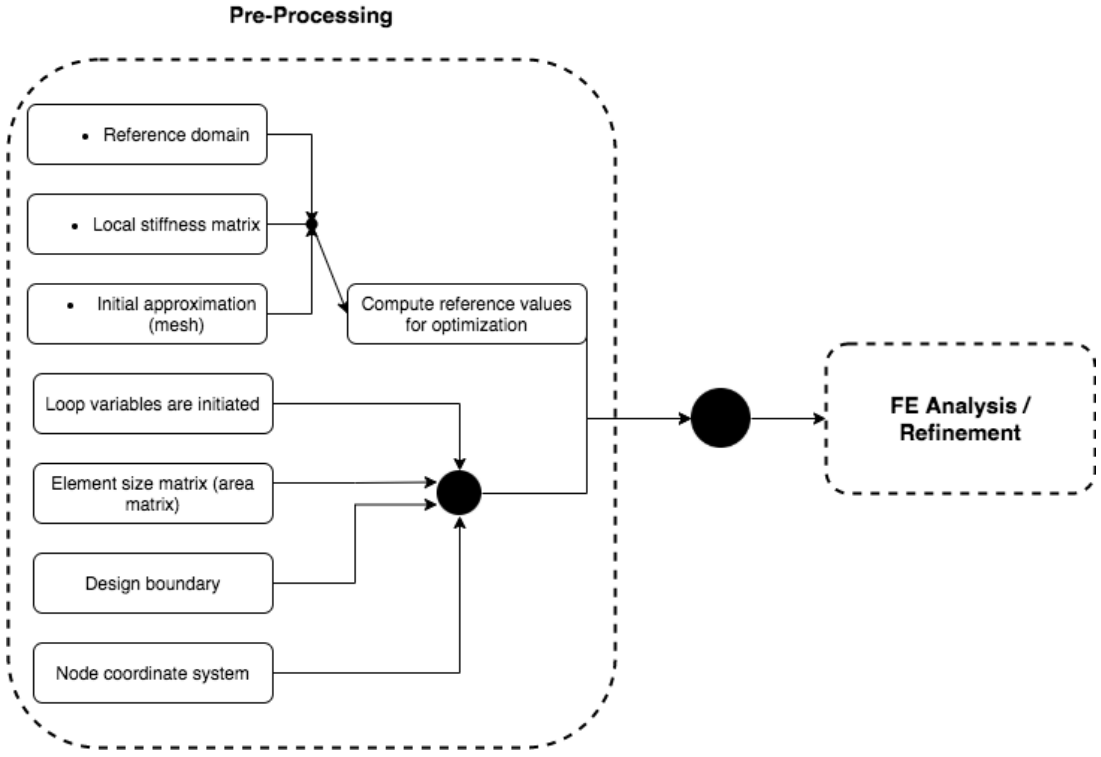


Figure 3.3: Flowchart for the "Pre-processing" stage, altered to reflect the implemented algorithm.

The program then creates an initial degree of freedoms connectivity matrix, based on the number of elements inputted by the user, that will be used in the following sections, as well as a connectivity matrix based on node numbers with element identification.

This is followed by the creation of several other variables, non existing in the original code, such as an element area matrix (created simultaneously with the node connectivity matrix) , a node coordinate matrix and an initial design domain boundary matrix, that identifies nodes at the boundary. This boundary matrix is created knowing that the design boundary is quadrangular in shape and needs modification for other design domains, the matrix is a binary identification system, where 1 represents a node at the boundary. Since the algorithm is being implemented from scratch there is a need to implement such variables, considering there is no straightforward way (for a non-uniform mesh) to do

some verifications along the way without these variables; the coordinate system is also used in the last stage for plotting. It is worth mentioning that the program doesn't define the boundary conditions at this stage, instead, they are defined at the end of the refinement cycle, where the program computes the results of the equilibrium equation for the initial mesh.

3.2.2 Finite Element Analysis / Refinement Section (lines 66-478)

After the starting stage of the program the code is completely written within an infinite *while* loop that stops with a *break* command when the number of iterations reaches the number inputted by the user, this is because the refinement process is written to work in a loop, i.e., after the initial mesh an optimization cycle follows, the refinement criteria then outputs the elements to be refined and a refinement cycle is started again, this is then followed by an optimization cycle and so on, until the break command is issued. Please see algorithm 1 for a summary of the steps needed.

Speaking in particular of the refinement section, the first step is identifying if it's the initial iteration, if this is the case, and since the initial mesh is always uniform and generated using user inputs, the same as if we were using the original programs, the program uses the same code to generate it and is then used as a starting point and no refinement takes place. This means that the *If* condition in which the refinement section is run first selects if its a starting iteration or a refinement iteration, if it's the first, the program renames and creates variables so that they can be used in the optimization loop without the need to identify, again, if it's a starting iteration; if its the second, meaning it's a refinement iteration, we enter the refinement code, where the bulk of the programming work was done.

Since the initial iteration code is only eight lines long, and easily comprehensible after explaining the refinement iteration, we will focus only on the last. The program starts by initiating a *while* loop (line 74) that will refine the elements present in the vector $refinar$, this loop will be active until all elements of the vector have been refined; an updated node and degree of freedom (DoF) connectivity matrix are the primarily output of this loop (ending in line 337). In more detail, the program searches for the element to refine in the existing node connectivity matrix and reads and stores the four nodes that identify it, after this, and for compatibility issues, the program checks if this is the first element refinement or a subsequent one, the only difference is the way the new nodes will be numbered, since the nodes created from the first refined element will always be new, so there is no need to check for node repetition. This node repetition check is important since the program doesn't renumber the nodes for each new mesh, but simply adds new nodes to the already existing one, meaning repetitions can occur in adjacent elements when they are both refined. As seen on figure 3.4, when elements are refined up to five new nodes can be created, four in the middle of the edges that make the original element and one middle node; the middle node will always be new and there is no need to check for repetitions, but the other four nodes can either be new or already exist if the adjacent element has already been refined, this is the case if the element marked in blue tint is refined, then the node circled in black would be a repeating node. An example of said repetition check can be seen in lines 81-95, where the code checks for a "left" node repetition (the left node is the node that will appear in the middle of the left edge of

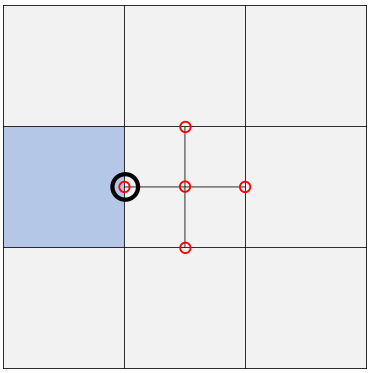


Figure 3.4: Illustration of the mesh refinement and the newly created nodes, indicated by the red circles. The circle in black indicates an already existing node in case of refinement of the element in blue.

the original element), assessing whether the node already exists or if there is a need to create a new node(lines 82-91). This repetition check is done using the node connectivity matrix and going line by line to check for the existence of a node between the nodes that make the side in question, in this example, the left side, if it doesn't exist the program creates it and updates the maximum node number, used for the next verifications (lines 92-95).

It is during this loop that variables that will be used later on are created and updated, such as a coordinate matrix ($coordinates$), a registry of all the original elements and subsequent refinements ($registry$), a matrix where the area of each element is stored ($area$), the nodes at each side of the one created ($node\_restrictions$ – they will be possible constraints to the node, if it's a 'hanging node', remember equation 2.14) and a boundary matrix that identifies all the elements at the boundary of the design domain ($fronteira$). As previously stated the node and DoF connectivity matrices are also updated with the new elements, and the originating element is deleted. It can be useful to understand how the coordinate system was implemented, the program computes each node's coordinates by using the coordinates of the original (refined) element and knowing that the new nodes will always be at half the distance between the original ones.

It is also important to note a small piece of code placed near the end of the loop (lines 272-323), this is a final verification, done after the last element has been refined, designed in order to solve the mesh incompatibility mentioned in section 2.2.1, this issue arrises from the way the program constrains floating nodes that are created when an element is refined. In order to solve the problem of level two incompatibilities, which occurs when an element is four times smaller than that adjacent to it, the program checks for a possible incompatibility and refines the adjacent elements, which removes the mentioned incompatibility by downgrading it to a single-level incompatibility (remember figure 2.9). This is done in a simple but crude way, using the area matrix the program searches for smaller elements in comparison to those four times bigger, after identification of the elements, and since the program doesn't know the location of each element *a priori*, except that of the original elements, we use the registry of element refinements and place each small and large elements at the location of its original elements, and then refine all the elements inside of the large original element. This is done because of the computational intensive calculation that is to calculate the distance between elements, so this is an easy way to,

without knowing what elements are next to each other, be able to solve the mesh-incompatibility. To further illustrate the algorithm that solves this problem a basic scheme was made, see figure 3.5, here illustrating the initial problem (on the left) and the solution (on the right); the initial size elements are in gray, the blue elements have been refined once and the orange elements have just been refined. The mesh on the left illustrates the stage of the mesh at which the two-level mesh-incompatibility is detected, this problem was created by refinement of the element in orange, to solve this we need to refine the elements marked with a blue circle. But since the program doesn't know the location of each newly created elements it solves this by refining all the elements that have been originated by the elements in green.

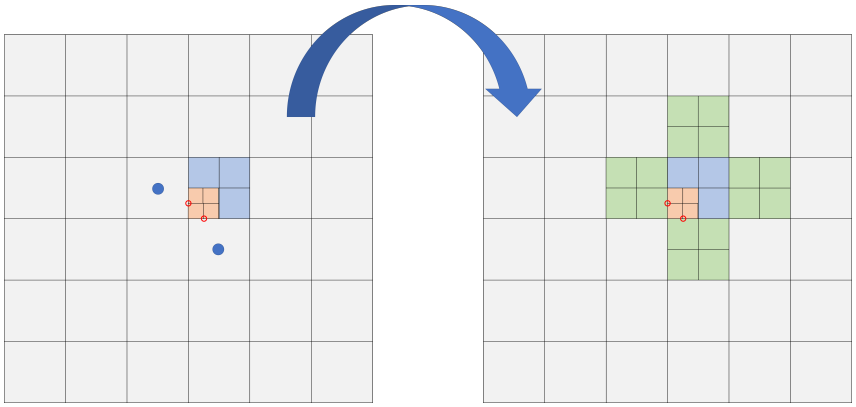


Figure 3.5: Illustration of how the program solves mesh-incompatibilities. On the left is the problematic mesh, on the right is the solved mesh.

Since we use the initial element location, this technique produces the expected outcome of resolving the incompatibilities but with the disadvantage of losing fine control over each large element, which means that some elements will be refined that didn't require it (the green elements on the right that weren't marked by the blue circle), in other words, the finer the initial mesh the better the outcome of this technique.

At the very end of this loop (lines 324-339), the program assigns the element density computed at the previous iteration to the subsequent elements originated from the refinement; this piece of code is only active after the first optimization, since it's the source of the density vector used here.

After the loop as been finished the program identifies all the possible 'hanging nodes' and then removes those at the design domain boundary (since they aren't true 'hanging nodes'), leaving us with all the real 'hanging nodes' and those that need to be constrained. The constraints are applied (lines 359-389) using a temporary copy of the node connectivity matrix, for easier coding, meaning the program then needs to transform it back into a DoF connectivity matrix (lines 391-402) before using it to compute the global stiffness matrix, K. Using the previous information, and because the elements where constraints have been applied don't use the "standard" element stiffness matrix, KE, the program computes (lines 403-434) the required element stiffness matrix for each constrained element, KE_e, and then uses it to generate a required variable to compute K (lines 435 - 444). With all the necessary variables generated, the program can now compute the global stiffness matrix and the

subsequent displacement matrix (lines 456-478), using the user defined boundary conditions (lines 461-464).

The refinement section of the code ends here, in line 478, with all the necessary variables ready for the optimization cycle.

3.2.3 Optimization Section (lines 479-584)

The optimization loop is almost a complete transcription of the original code, with the necessary adjustments and modifications needed, namely the fact that there are individual stiffness matrices for each element since not all are equal, as was the case in the original code, and that the filter needed a way to calculate the distance between elements. In this optimization section of the code it is also included all the necessary outputs used in the post-processing for plots and variable outputs that will be stored in the user defined directory, such as a registry of compliance over time.

The program starts by checking whether it's an iteration where the radius of the filter will need to be reduced or not (lines 482-484), this is important due to the nature of an adaptive mesh refinement, where different sized elements will appear, leading to the need of a smaller radius for the filter as the size of the elements decreases. In line 486, the user defined optimization convergence criterion can be changed according to the user needs. The actual optimization loop (lines 504-583) starts after a few loop variables and other needed variables are defined, for example, the name of the plots and other outputs names. As previously stated, the optimization loop is considered a 'standard topology optimization' loop, and is very similar to the original code, with the first necessary adjustment on line 503, where a 'for' cycle needed to be implemented in order to separate the computation of the compliance and its sensitivity, because of the different element stiffness matrices; the program goes row by row on the DoF connectivity matrices and for each row checks whether or not is a constrained element, in which case it needs its own, previously computed, element stiffness matrix. Its important to note the use of the modified SIMP approach, as previously mentioned, for easier implementation of the density filter [10], with the only modification being

$$E_e(x_E) = E_{min} + x_e^p(E_0 - E_{min}), \quad x_e \in [0, 1] \tag{3.2}$$

with the sensitivity of the objective function as

$$\frac{\partial c}{\partial x_e} = -px_e^{p-1}(E_0 - E_{min})\boldsymbol{u}_e^T \boldsymbol{k}_0 \boldsymbol{u}_e \tag{3.3}$$

The next necessary adjustment can be seen in the filtering function (called in lines 528-532); for the filter to operate it needs to know where each element is, this is where the coordinate system previously computed is needed, the program uses the coordinates of each node to compute the centroid of each element (lines 667-723) and once all the centroids have been computed it uses that information to compute a matrix with the distance between all elements (line 696); this is where one of the limitations of the program is, being a RAM - Random Access Memory - intensive process, that scales with the

number of elements in the mesh). With the distance between elements stored in a variable, variable that will only be computed once throughout the optimization cycle, the filter can now operate normally (lines 698-723) for the remaining of the cycle. It's important to note the use of the MATLAB function *squareform* that allows for a more readable matrix, adding to the easy adaptation of the filtering code.

The optimality criteria method remains practically the same (lines 534-550), with a move limit of $m = 0.2$ and a damping coefficient $\eta = 1/2$, with minor changes in lines 537 and 543, where the different size elements needed to be taken into account, hence the appearance of the variable "area" that wasn't previously there. After each optimization iteration, and until a convergence is reached, the code has some new variables and verifications, for example, the program stores the compliance history and to ensure that we don't end up in a very big cycle, if convergence is not reached within a user defined amount of iterations (max_opt_ite), or if the the variable $change$ is approximately the same for a specified number of iterations(max_change_acc), "convergence" is forced and the program jumps to its next step.

Finally, this is where the optimization loop ends and the user defined convergence criteria will be checked; the same criteria as the original code is used, meaning that it reaches convergence when, between two consecutive designs, the change in design variable is less than 1%.

3.2.4 Final Stage - Plots and Outputs (lines 589-618)

The program now enters its final stage; after each optimization cycle the program saves a number of useful variables, such as the compliance, displacement and densities; it also plots, with and without the mesh and stores the plots to the specified directory. The node matrix and coordinate matrix are also stored so that it's possible to posteriorly plot the densities. It's also of note that a new way to plot (lines 630-665) the densities was needed, since the mesh is no longer uniform the original way to plot the densities no longer works. The function used to plot the mesh is a simple code that extracts the coordinates of each node and stores it in a "X" and "Y" variable, those can than be used to plot the elements and fill them with the corresponding densities.

Finally this is where the codes ends with two possible outcomes, if the user specified number of optimization iterations has been reached (line 608) the code *breaks*, leaving the infinite loop and ending the program, if that's not the case the code continues, the elements to be refined are defined using the AMR criterion and the code starts again from the Refinement Section. This last possibility is coded on line 617, which as previously stated selects the new elements to be refined. This is done by selecting the elements that are within specified density bounds. The user can modify the refinement criterion by changing the way the program selects the elements and storing them in the vector, $refinar$. This contributes to the requirement of a straightforward and easy to modify code.

Chapter 4

Results

In this chapter the results of several topology optimization problems are presented along with a parametric analysis and performance comparison with the original code [11], as well as the optimized code [10].

This chapter will follow the sequence in which the studies were conducted. Firstly, it starts by presenting a brief description of the problem used for an extensive parametric study, followed by the parametric study itself in section 4.1, and secondly, a validation of the algorithm is made by comparing against several known problems, followed by a brief performance analysis, in section 4.2.

4.1 Parametric Study

In order to try to select the best parameters in which to run the algorithm, a parametric study was conducted. For this, the parameters used by the original authors were used as a starting point, when appropriate. For the entire study the MBB-beam problem, mentioned bellow, was considered. Unless stated otherwise, the volume fraction (equal to 50%), penalization power (the usual value of $p = 3$ [11]), filter radius ($r_{min} = 1.5$) and the filter radius change frequency ($div_r = 2$) will remain constant. The convergence criterion remains the same as the original codes, at less than 1% for consecutive change in design variable. To ensure results in a timely manner the optimization iterations were kept at a maximum of 1000 per cycle, unless stated otherwise. The number of iterations was kept at 5 to ensure a maximum refinement of four times per element.

Problem Description

Like the original code presented by Sigmund, the TO of a MBB-Beam was chosen (with symmetry boundary conditions) as a main comparison problem. This is not only for simplicity sake, but as well because it's a very well documented problem. As seen in figure 4.1, it is a simple problem with a rectangular design domain, this makes the programming of the design much simpler, since the boundary is quadrangular and we can use this to (in a simple way) code the 'boundary matrix' previously mentioned.

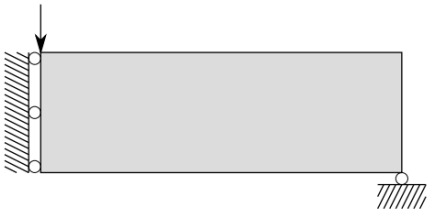


Figure 4.1: Illustration of the symmetric MBB-beam design domain, boundary conditions, and external load for the topology optimization problem. Reprinted from Andreassen et al. [10, p.2].

The parametric study is important to better understand the behaviour of the algorithm to changes in the several AMR and optimization parameters and subsequently select the 'best' parameters, that will be used in later studies. The mesh considered consists of 32×24 elements, this was selected in order to have a sufficiently large initial mesh, without compromising the number of AMR iterations, since a larger initial mesh would prove too heavy for the available computational resources. Other parameters, such as the stiffness of the material ($E_0 = 1$), the stiffness of void regions ($E_{min} = 1e^{-9}$) and the Poisson's ratio ($\nu = 0.3$), remain constant throughout.

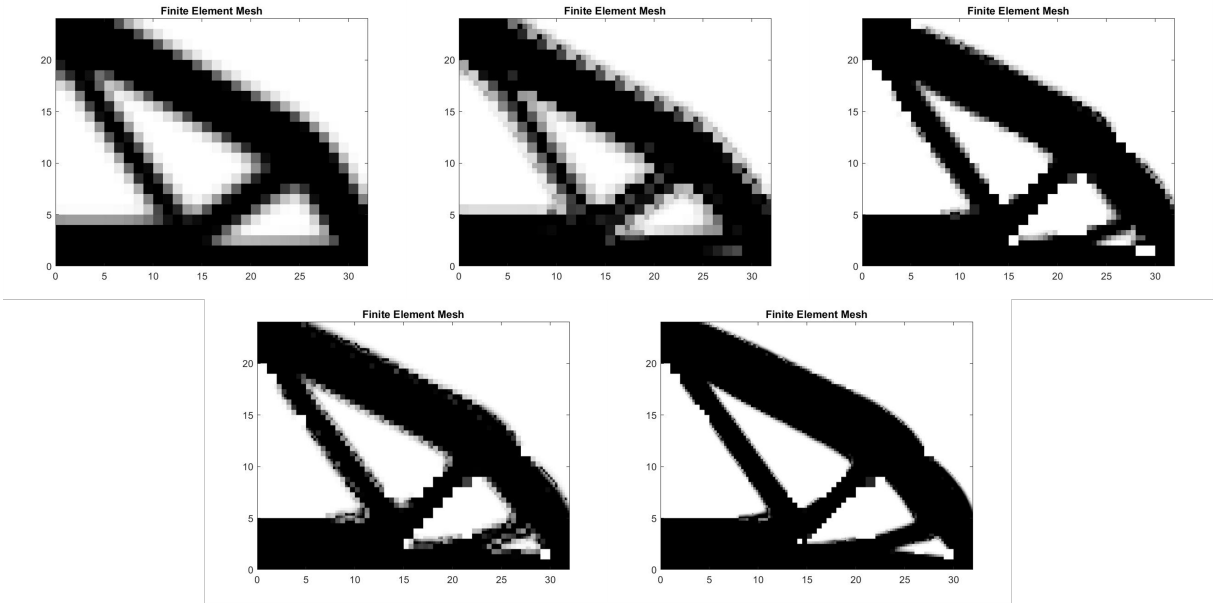
4.1.1 Refinement Criteria

First, establishing the "baseline" refinement criteria in which all other problems were run is needed. As previously mentioned (see section 2.2.1) we will approach this in two ways, refinement of the intermediate density elements and refinement of 'solid' elements. In order to only change the refinement criteria the volume fraction, penalization power, filter radius and the filter radius change frequency will remain constant as previously mentioned. For simplicity the filtering technique is also kept constant and is a sensitivity filter.

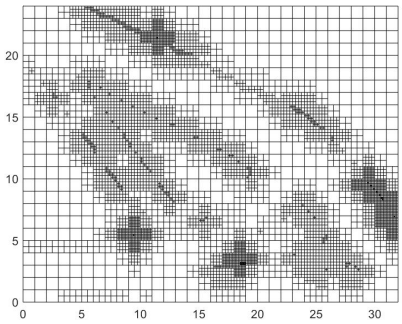
Refinement of intermediate densities

For the parametric study a number of different density limits were selected, for the lower limit the study focused on four densities, $\{0.2, 0.3, 0.4, 0.5\}$, and for the upper limit three densities, $\{0.7, 0.8, 0.9\}$. So as to not burden the text with repeated results the first iteration will only be shown once, since it represents the solution on a uniform mesh. It was also decided to limit the number of iterations to five. The evolution of the AMR algorithm by iteration will only be shown once, for the first density range, and only the last solution will be presented for the remaining.

Firstly, figure 4.2 shows the result for the first interval, $0.2 - 0.7$, where the evolution of the design can be seen, along with the resulting mesh. The subsequent figures, 4.3 to 4.7 will show a short version for the additional intervals (some results for some intervals were omitted and can be found in appendix A.1).

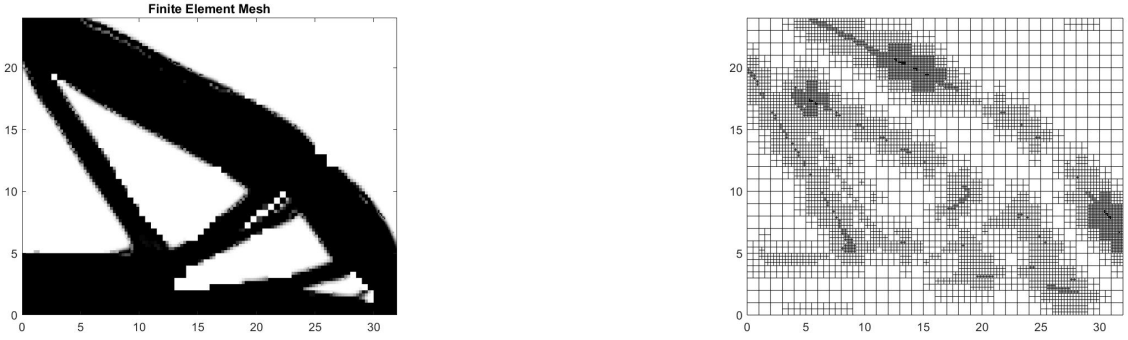


(a) Resulting topology optimized design



(b) Resulting mesh

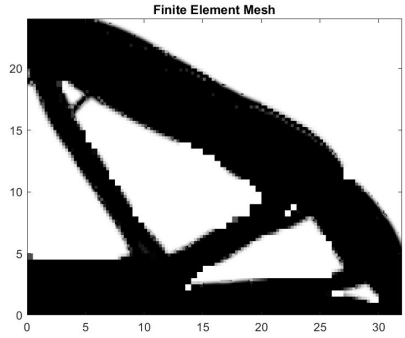
Figure 4.2: Evolution of the topology optimization algorithm design with five iterations and final resulting mesh, for a refinement criterion of $0.2 - 0.7$.



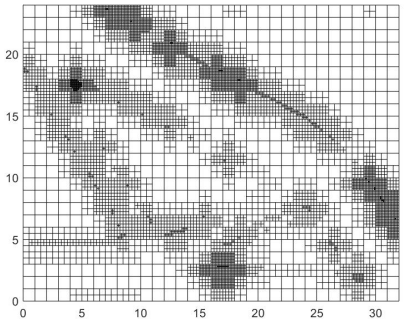
(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.3: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.2 - 0.8$.

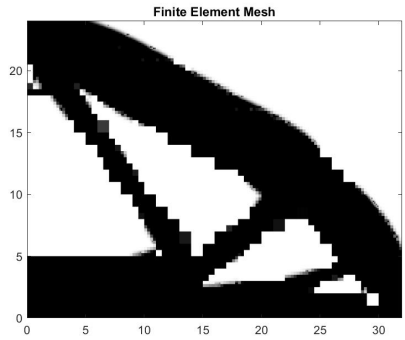


(a) Resulting topology optimized design

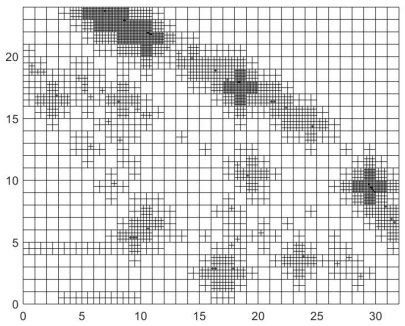


(b) Resulting mesh

Figure 4.4: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.3 - 0.8$.

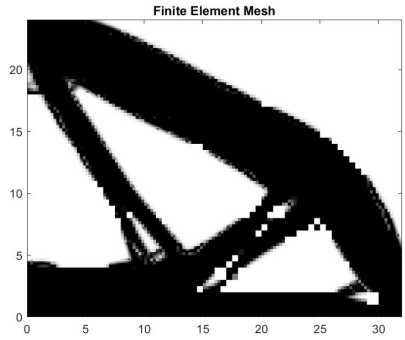


(a) Resulting topology optimized design

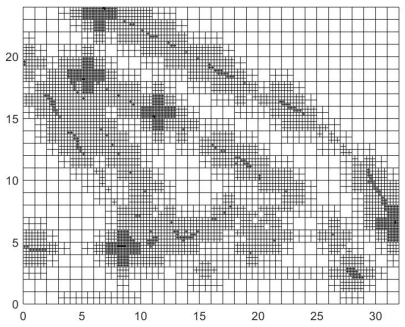


(b) Resulting mesh

Figure 4.5: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.4 - 0.7$.



(a) Resulting topology optimized design



(b) Resulting mesh

Figure 4.6: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.4 - 0.9$.

(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.7: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.5 - 0.7$.

After analyzing not only the final design, but also the final mesh, it's possible to exclude the narrowest intervals, those starting at 0.5, this is because of poor design boundary, as one can see in figure 4.7 a), also visible on the same figure is the lack of definition on the resulting mesh. Not quite to the same extent, the intervals starting with 0.4 suffer from the same problem, with rough 'edges' around the boundary especially seen at interval $0.4 - 0.7$ (see 4.5), with better definition are the remaining intervals, $0.4 - 0.8$ and $0.4 - 0.9$. These last two also suffer from the appearance of holes in the structure, which might be, again, due to the poor selection of all the boundary (see figure 4.6 b)). The better results seem to appear when selecting the interval $0.2 - 0.8$, with the highest 'range' resulting in better definition of the boundary, as can be seen in figure 4.3. The results edge those from the intervals starting with 0.3, which with satisfactory results don't select the boundary as well (see figure 4.4 b)). It is also of interest to see the behavior of the compliance over time for the interval chosen, see figure 4.8. Even though the compliance over time is lower with the use of this AMR criterion the same can be said for the others, making it a less than an ideal way to judge the 'performance' of the criteria.

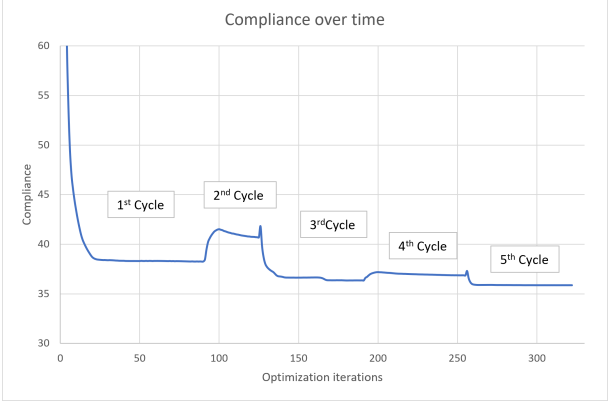
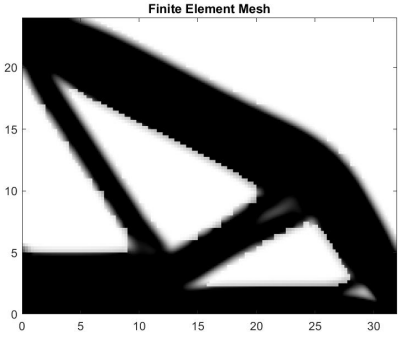


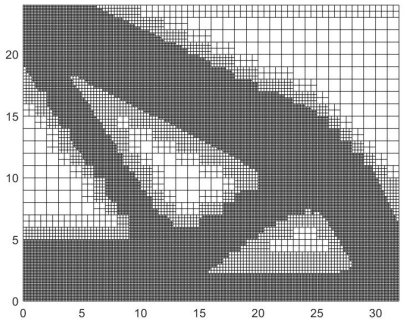
Figure 4.8: Evolution of the compliance over time for the AMR criterion of $0.2 - 0.8$.

Refinement of 'solid' densities

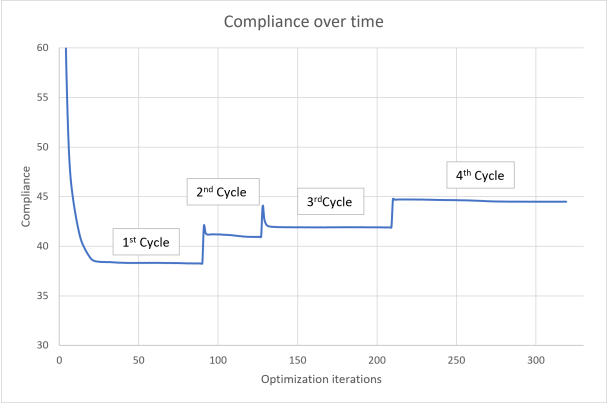
As previously stated another strategy is to refine the densities which are approximately solid, this is done by choosing every density bigger than 0.8 (see code, line 614). This essentially produced a very predictable mesh that reliably represents the resulting design (see figure 4.9), proving to be a good way to select the 'solid' densities. At first glance the resulting design is very well behaved around the boundary and reduces the number of elements when compared to a uniform mesh, but with each 'AMR $\rightarrow$optimization' cycle the compliance increases (see figure 4.9 c)), indicating a less stiff design. This is not necessarily a bad behaviour, since a cleanlier boundary description will increase the number of elements leading to a more 'realistic' design with possibly lower compliance. It's also important to note that this method is more computing intensive since it leads to considerable more elements and more RAM usage, in fact, four cycles was the limit of cycles before using all the available RAM (as previously mentioned, the way the distance between centroids is computed is very RAM intensive). With this analysis concluded, henceforward, all problems will be solved with an AMR criterion of $0.2 - 0.8$.



(a) Resulting topology optimized design



(b) Resulting mesh

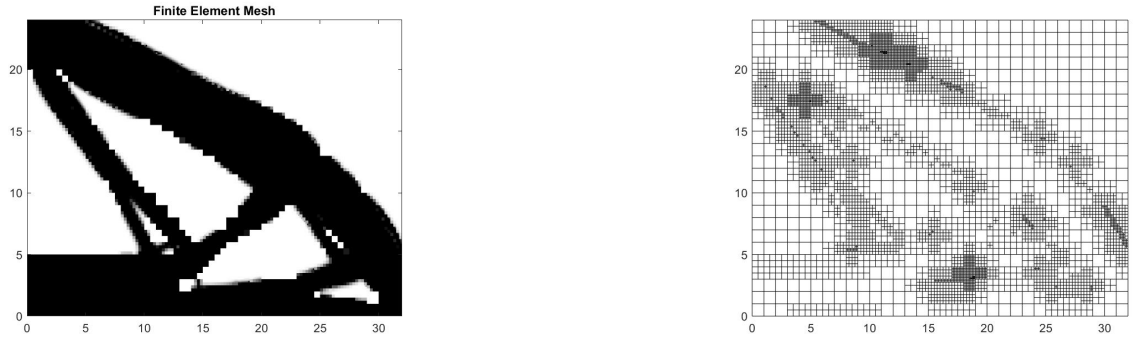


(c) Compliance over time

Figure 4.9: Topology optimized design with AMR and resulting mesh, for a refinement criterion of 'solid' densities, along with the compliance over time.

4.1.2 Penalization power

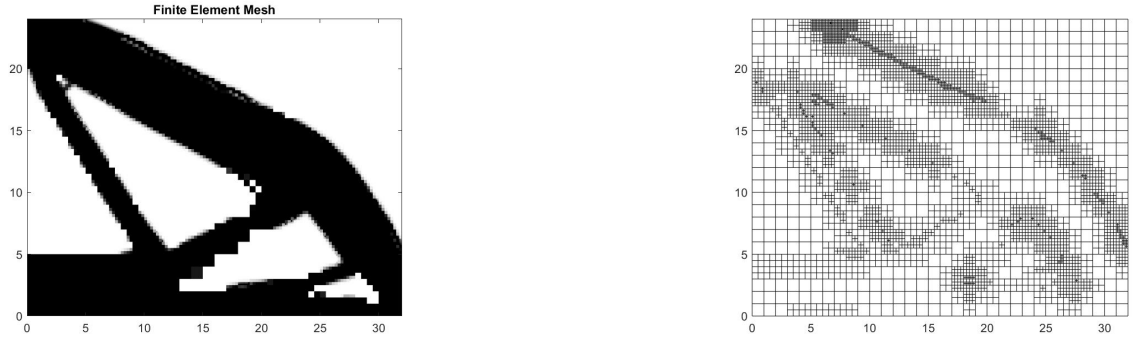
We start by remembering equation 2.5, which states that in 2D the penalization power must be bigger than a prescribed interval, for $\nu = 0.3$ that means it must be bigger than three. With this in mind four values were tested, $3, 4, 5$ and 6. Once again, a visual analysis of the obtained final design and the final mesh is done to draw conclusions about the best penalization power. It is worth mentioning that one of the problems, with $p = 6$, surpassed the iteration limit but it was allowed to keep going, until convergence, without limitation. There is no need to plot, once more, the results for a penalization power of 3, as this is the same as the one shown in figure 4.3, for the rest of the p they are presented bellow.



(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.10: Topology optimized design with AMR and resulting mesh, for a penalization power of 4.



(a) Resulting topology optimized design

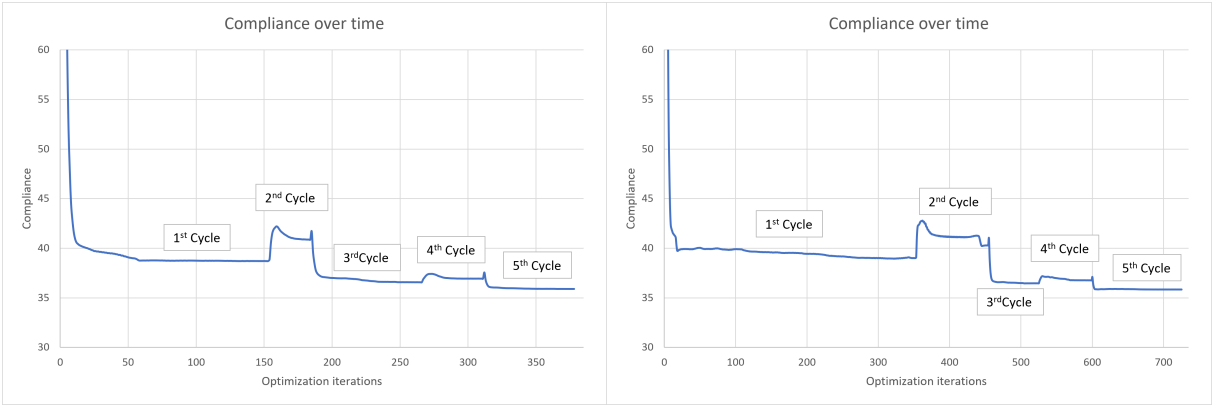
(b) Resulting mesh

Figure 4.11: Topology optimized design with AMR and resulting mesh, for a penalization power of 5.

(a) Resulting topology optimized design

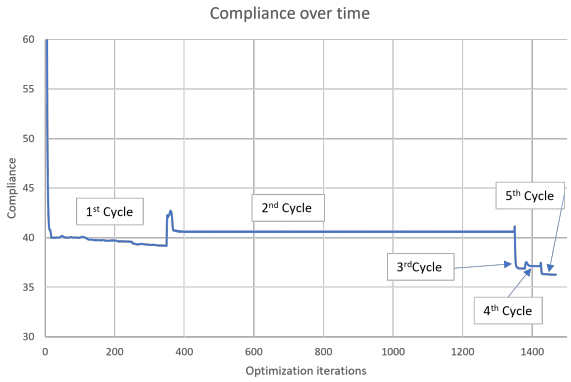
(b) Resulting mesh

Figure 4.12: Topology optimized design with AMR and resulting mesh, for a penalization power of 6.



(a) $p = 4$

(b) $p = 5$



(c) $p = 6$

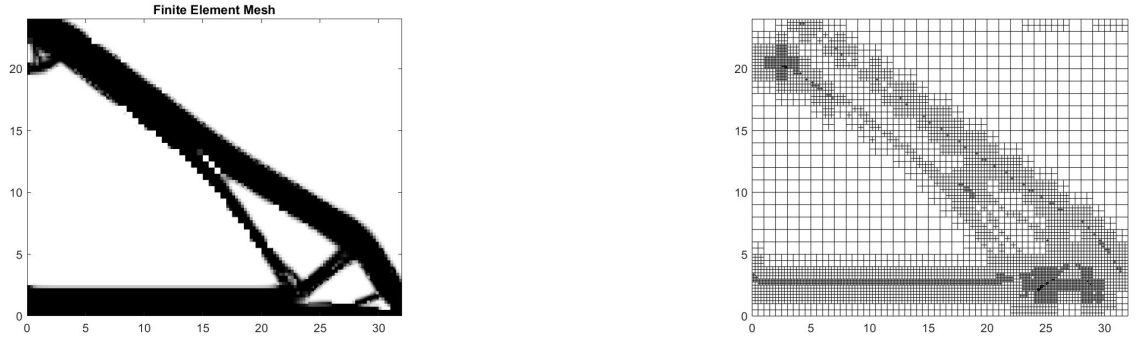
Figure 4.13: Compliance over time for three different penalization power, $p = 4$, $p = 5$ and $p = 6$

The obtained results show a clear difference between designs, especially those with a high value of penalization power ($p = 5$ and $p = 6$), the design boundary starts to appear rugged and with numerous edges, see figures 4.11 and 4.12, this is also valid to some extent on $p = 4$, see figure 4.10, where some edges appear along the boundary. The aforementioned is the consequence of locking in the 'solid' densities earlier in the process, thus resulting in 'under-refinement' of the material in some parts and resulting in the edges shown. It's important to remember the algorithm is selecting the intermediate

densities for refinement, so early locking of 'solid' densities makes it less reliable at following the boundary of the design, as seen, for example, in figure 4.10 b) where locations resembling crosses appear on the mesh surrounded by unrefined mesh. The analysis of the compliance over time also proves useful to better understand the behaviour of the algorithm for high p; the plot for $p = 3$ was previously shown in figure 4.8, where for the others can be seen in figure 4.13. Analysis of the plots show a significant increase of the iterations performed for higher p, even though the final compliance is around the same. This behaviour was expected as the same happens on the original codes, but when using the AMR algorithm the effect is increased with the added disadvantages already described. For this reason, and since the value typically used is $p = 3$, this is the value selected to be used, subsequently.

4.1.3 Volume fraction

A quick study on the influence of the volume fraction was also conducted, and so three values were selected, $f = 25\%$, $f = 50\%$ and $f = 75\%$. Since we've already established some parameters, we already have the results for a volume fraction of 50%, see figure 4.3, for the remainder are as follows.



(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.14: Topology optimized design with AMR and resulting mesh, for a volume fraction of 25%.

(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.15: Topology optimized design with AMR and resulting mesh, for a volume fraction of 75%.

As we can see from figure 4.15, the resulting design for a volume fraction of 75% is filled with holes and edges on the 'inside' boundary and the mesh doesn't follow the boundary. For the volume fraction

of 25% the design boundary is much more detailed and the resulting mesh clearly follows along the boundary. This considerable difference in behaviour appears to be the result of poor refinement combined with the limitations of the filtering technique, since in the bigger volume fraction the resulting mesh creates a space where large amounts of elements appear and seem to create the effects seen in mesh-dependency. The relatively large volume fraction makes it so that only a small number of intermediate densities exist and are all located in the 'interior' of the design, this is visible in figure 4.16, where it shows the initial optimized design on a uniform mesh, thus creating a situation where the algorithm behaves poorly.

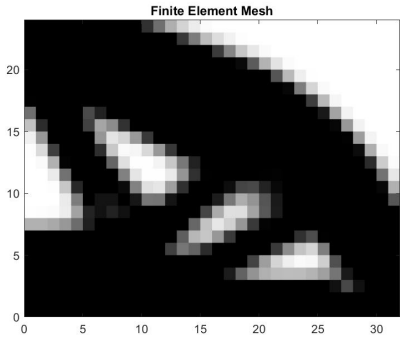


Figure 4.16: Topology optimized design with uniform mesh for a volume fraction of 75%.

To confirm the cause, the algorithm was allowed to run a 6^{th} iteration, expecting it to further refine the inner region and somewhat reduce the problem (the filtering radius remain the same). When analyzing the results, see on figure 4.17, we see that the further refinement did improve the 'edges' previously seen and a cleaner border region is formed. Mesh-dependency is still present and the conclusion remains the same, the algorithm behaves poorly for this high volume fraction. For a volume fraction of 50% the algorithm behaves well, as previously seen, and gives good results with a good resulting mesh. Thus, we can say that the AMR algorithm behaves well for lower volume fractions. For this reason, and since a volume fraction of 50% was used in the original codes, this was the selected value.

(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.17: 6^{th} iteration of a the topology optimized design with AMR and resulting mesh, for the volume fraction of 75%.

4.1.4 Filtering technique and radius

As previously mentioned a filtering technique is required to solve the complications that arise from the discretized computational calculation of the optimization problem. The previous designs were all computed using sensitivity filtering but the algorithm is capable of deploying density filtering, thus, an analysis on the different methods was required. It is also important to mention that a simultaneous analysis of the filter radius is also needed.

For basic knowledge on how these filtering techniques work please remember section 2.1.3.

Sensitivity filtering

To better understand the role of the sensitivity filter, as well as the radius of the filter, a number of experiments were carried out. These involved changing the radius and implementing a solution where the filtering radius decreases with the iterations.

Firstly, a constant filter radius was analyzed, for this a filter radius of $r = 1.5$ was selected, this was considered the minimum filter radius, since a lower radius would result in not including the diagonal elements (see figure 2.5, for unitary elements the diagonal distance between centroids is $\sqrt{2} \approx 1.41$). As seen in figure 4.18, the use of a constant filter radius results in the appearance of hole like structures in the design with a very poor definition of boundaries. This is a less than ideal behaviour which probably occurs from the fact the filter radius becomes too large for some areas of the mesh.

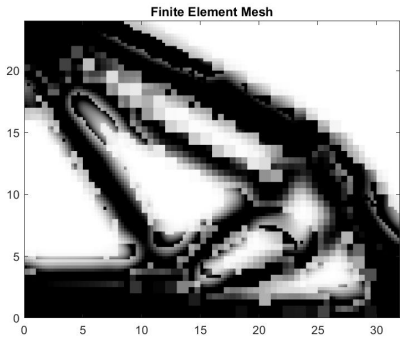
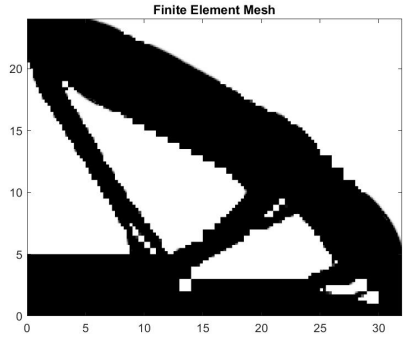


Figure 4.18: Design for sensitivity filter with constant filter radius of $r = 1.5$.

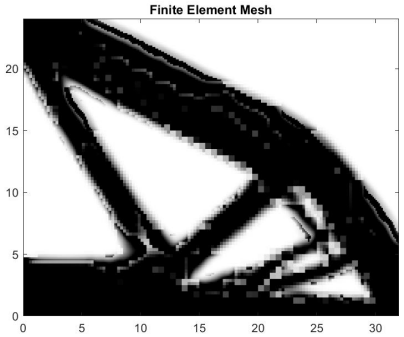
In an attempt to solve this, the idea of reducing the filter radius as the mesh was refined was implemented. This meant that the algorithm would divide the radius by a factor of two every div_r iteration. For this, three values, $div_r = \{1, 2, 3\}$, were selected with a starting radius of $r_0 = 1.5$. The results for $div_r = 2$ have already been presented and can bee seen in figure 4.3, for $div_r = 1$ and $div_r = 3$, see figure 4.19.

The design for $div_r = 3$ resembles the one for a constant filter radius, even though with better details, presenting the same hole like structures inside the design, meaning the radius continues to be too large for the size of the elements. For the design with $div_r = 1$, we quickly obtain a more 'black' solution with much less intermediate densities, this is a positive result and confirms the need for a smaller filter radius as the refinement progresses. When comparing to the solution already obtained for $div_r = 2$, at first

glance the design is identical, we need to look at the resulting meshes to identify important differences. That can be seen in figure 4.20, where on figure b) we can see a further refinement of the boundary, which is the preferred behaviour.

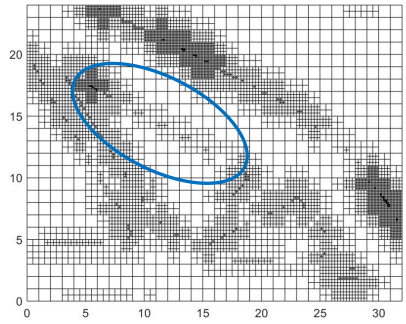


(a) $div_r = 1$ – final radius of 0.1875

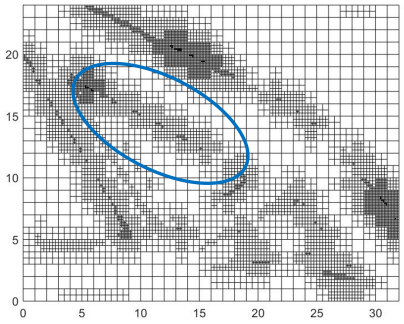


(b) $div_r = 3$ – final radius of 0.75

Figure 4.19: Design for $div_r = 1$ and $div_r = 3$. Note: for figure (a) the design represents the 5^{th} iteration, while (b) represents the 6^{th} iteration, this was done in order to plot the last design before the radius would decrease again.



(a) $div_r = 1$ – final radius of 0.1875



(b) $div_r = 2$ – final radius of 0.375

Figure 4.20: Mesh detail for $div_r = 1$ and $div_r = 2$.

With this in mind, testing with two more starting radius, which effectively means the final design had a radius of $r = 0.5$ and $r = 0.625$, produced subpar results and means that the better radius change frequency, for this problem, is $div\_r = 2$ with a starting radius of $r_0 = 1.5$, producing a final radius of $r = 0.375$.

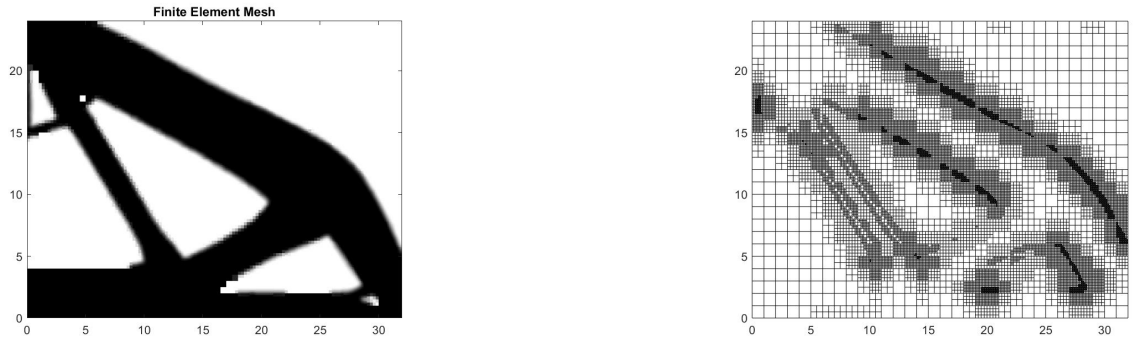
However, this is an indication that the filter radius for the last design iteration must be carefully chosen in order to produce good results, for this particular problem the (expected) smallest element should be $\frac{1}{256}$ of the starting element, and these smaller elements require efficient filtering as well.

Density filtering

With the results for the sensitivity filtering in mind, the next step is to test the algorithm with a density filter. It is worth noting that this filter is computationally heavier than sensitivity filtering and is expected to produce a 'blurred' region as the ratio of $radius/element\ size$ increases [30].

In order to test the behaviour of the filter more briefly, the refinement criteria used was limited to $0.2 - 0.8$ and approximately 'solid' densities with $div\_r = 2$ and starting radius of $r_0 = 1.5$; a constant radius was also tested with refinement criterion of $0.2 - 0.8$ and the same starting radius. For the refinement criterion of $0.2 - 0.8$ the results can be seen on figure 4.21, figure 4.22 shows the results for approximately 'solid' densities and figure 4.23 shows the results for a constant radius.

The results show an overall good behaviour of the density filter for both refinement criteria, and shows that the use of a constant radius is infeasible for both filtering techniques. The algorithm produces a good description of the design for intermediate densities, as seen on figure 4.21, with a good detailed mesh around the boundary. For the refinement of 'solid' densities (see figure 4.22) the result is comparable to that previously obtained with sensitivity filtering (see figure 4.9), but with less detail around the boundary. This might be because of the relatively high final filtering radius, since the algorithm was limited to four iterations due to the high number of elements. Once again, the use of a constant filter (see figure 4.23) produces a less than ideal result, with a 'blurred' design, characteristic of a large filter radius when compared to element size; it is also relevant to mention that the problem did not converge.



(a) Resulting topology optimized design

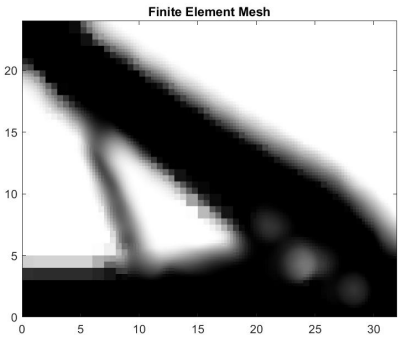
(b) Resulting mesh

Figure 4.21: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.2 - 0.8$. Final filter radius of $r = 0.375$.

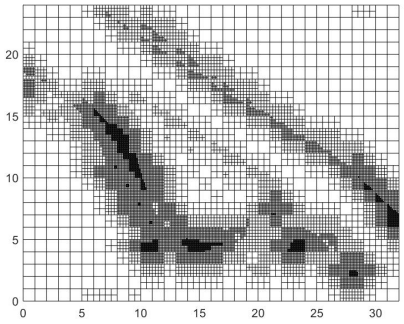
(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.22: Topology optimized design with AMR and resulting mesh, for a refinement criterion of 'solid' densities. Final filter radius of $r = 0.75$.



(a) Resulting topology optimized design

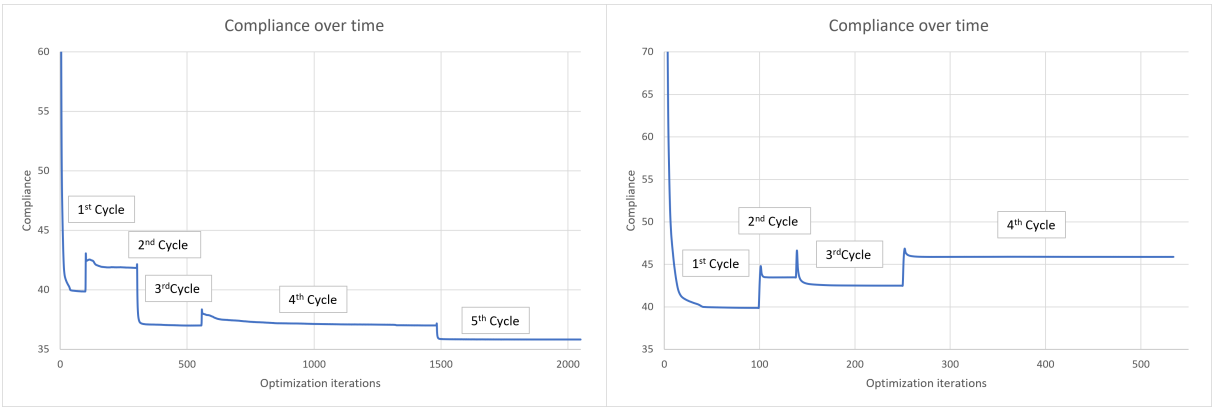


(b) Resulting mesh

Figure 4.23: Topology optimized design with AMR and resulting mesh, for a constant filter radius of $r = 1.5$.

When analyzing the compliance over time, for the two successful designs, see figure 4.24, the same behaviour as the one previously observed for sensitivity filtering is presented, meaning the compliance decreases for the 'intermediate' density refinement and increases for the 'solid' densities refinement.

Ultimately, and since the use of a density filter produced similar results with added computation time, the use of the sensitivity filter is preferred.



(a) Refinement criterion of $0.2 - 0.8$

(b) Refinement criterion of 'solid' densities

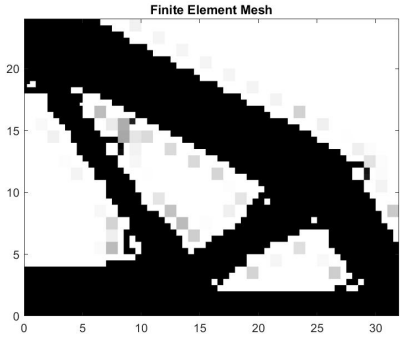
Figure 4.24: Compliance over time with the use of density filtering for a refinement criteria of $0.2 - 0.8$ (a) and 'solid' densities (b).

Modifying the filtering radius

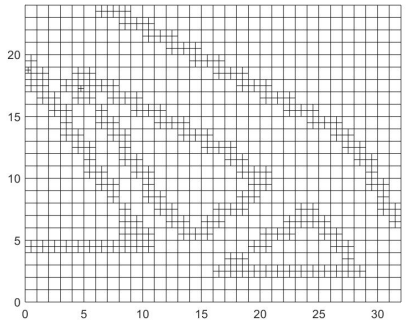
At this point it's obvious that the filtering radius plays a big role in obtaining good results. As a natural evolution, a method for finer control of the radius was tried. For this, a simple line of code was used to adapt the radius of the filter to the size of the element being filtered, i.e., the filter radius directly depends on each element size (ES). This modification can bee seen, in comment, in line 702 of the code.

For a simple analysis, with both filtering techniques, two sizes were selected, $r = 1.5ES$ and $r = 2ES$. The results can be seen bellow and show a failed attempt at implementing a new 'rule' for the filter radius. Present in all final designs for $r = 1.5ES$ (see figures 4.25 and 4.26), is a layer of

intermediate densities around the material, the result for the density filter also shows evidence of checkerboard pattern, indicating the filter was ineffective. Since there is this layer around the material, one could suppose the bigger radius of $r = 2ES$ would 'bridge the gap' and produce a region of low (but still gray) densities. This was not the case and the results for both filtering types (see figures 4.27 and 4.28) still show the same layer and now checkerboard pattern is present in both.

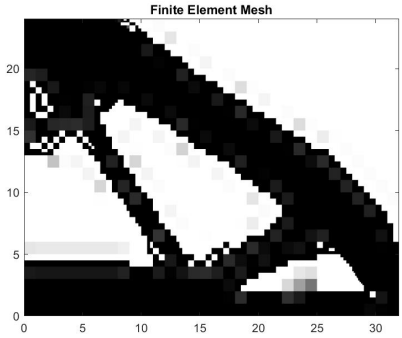


(a) Resulting topology optimized design

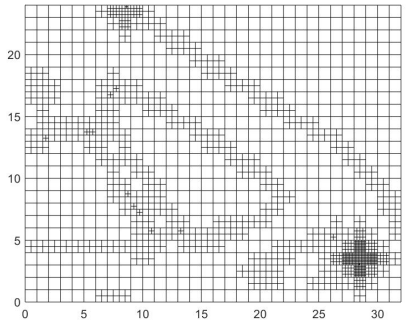


(b) Resulting mesh

Figure 4.25: Topology optimized design with AMR and resulting mesh, for a sensitivity filter with radius of $r = 1.5ES$

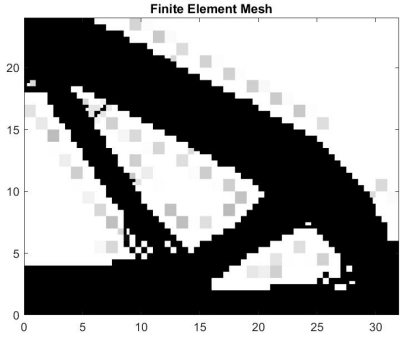


(a) Resulting topology optimized design

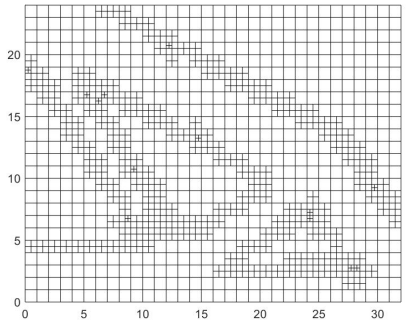


(b) Resulting mesh

Figure 4.26: Topology optimized design with AMR and resulting mesh, for a density filter with radius of $r = 1.5ES$



(a) Resulting topology optimized design



(b) Resulting mesh

Figure 4.27: Topology optimized design with AMR and resulting mesh, for a sensitivity filter with radius of $r = 2ES$

(a) Resulting topology optimized design

(b) Resulting mesh

Figure 4.28: Topology optimized design with AMR and resulting mesh, for a density filter with radius of $r = 2ES$

Another alternative was implemented to try and improve the filter, the program was allowed to run (with a sensitivity filter) for five iterations with refinement and then the filtering radius was decreased without any refinement, this was done to see if continuing to reduce the radius would produce any effect. The evolution of the last iterations is shown in figure 4.29. As one can see, after turning off the refinement, lowering the filtering radius produced some effects, smoothing out the border but having little impact in the design, especially as the radius continues to decrease, where it has no effect. This produced somewhat of a good result, with compliance decreasing from $C = 35.8774$ at the 5$^{\text{th}}$ iteration to $C = 35.4125$ at the 6$^{\text{th}}$, but nothing too significant, nonetheless, one might use it to achieve a better description of the border after the AMR algorithm has finished. Even though lowering the filter radius after the final iteration is not negligible, the main parameter to use as a primary mechanism to better control the filter, is the filter division frequency of div_r.

4.2 Validation

Now that a parametric study was conducted to establish the best parameters to use in the algorithm, we are able to compare the solutions of the developed algorithm with those produced by one of the original codes, the '88 lines of code' [10], using a uniform mesh (the more optimized code was selected due to the use of a relatively large mesh).

The way of comparing the algorithms is by using a uniform mesh of comparable element size. This means that we first compute the design using the developed AMR algorithm and retrieve the size of the smallest element, after having the size we can extrapolate the number of elements along the x and y sides to be equivalent to that element size. This is done since the original code uses unitary elements and we can't directly tell it the element size.

The validation of the algorithm consisted of analysing the solutions for four well known problems, the MBB-beam, the cantilever beam, the stocky cantilever beam and the 'wheel', after all results were analyzed a small paragraph is made about the difficulties in comparing the designs, resulting from the choice of the filter radius. A brief performance analysis of the algorithm was also conducted.

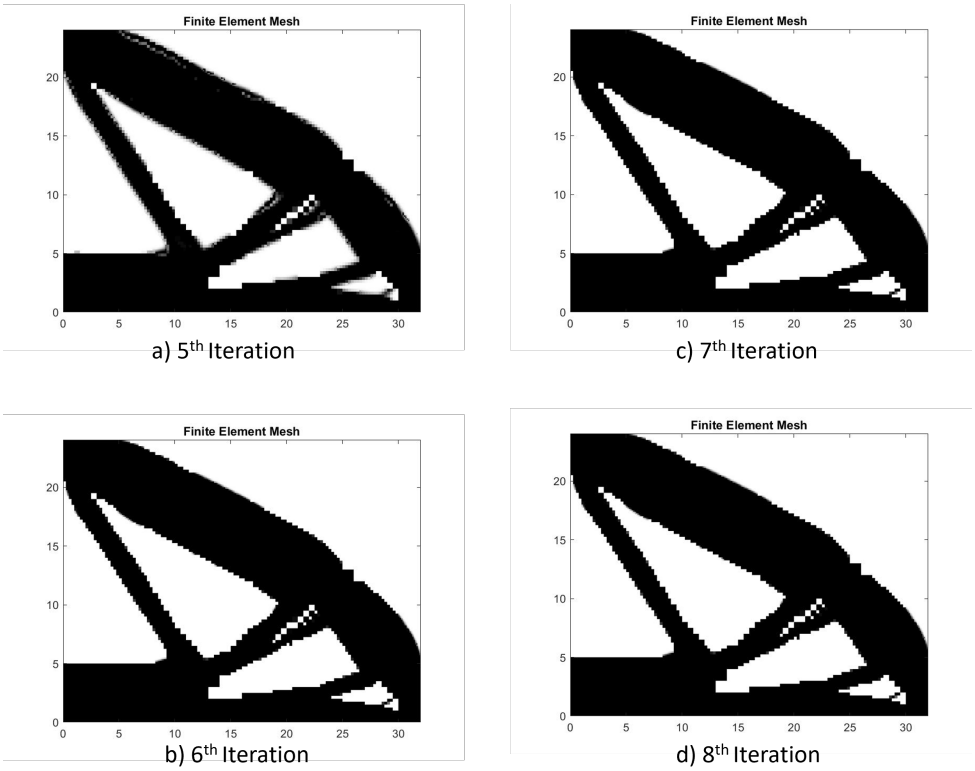


Figure 4.29: Evolution of the topology optimized design for a decreasing filter radius without refinement. The last iterations are showed with filter radius of a) $r = 0.375$, b) $r = 0.1875$, c) $r = 0.09375$ and d) $r = 0.046875$.

4.2.1 MBB-beam problem

The MBB-beam problem as already been described in the beginning of section 4.1 (remember figure 4.1 for design domain) and is the default state for the MATLAB code. Nevertheless, the boundary conditions are as follows:

```
F(2,1)= -1;
```

and

```
fixeddofs = union([1:2:2*(nely+1)],[2*(nelx+1)*(nely+1)]);
```

The problem was solved using the following parameters, penalization power of $p = 3$, volume fraction of $f = 0.5$, sensitivity filter with radius of $r = 1.5$ and an original mesh of 32×24. Except for the size of the initial mesh, these were the parameters used by Sigmund [11]. The AMR method chosen was the refinement of intermediate densities ($crt\_low = 0.2$ and $crt\_high = 0.8$) and for a better result the program was allowed to decrease the radius of the filter once, after refinement ended.

Firstly, we are going to establish the baseline solution for the MBB-beam problem using the original code. Knowing that the smallest element using the AMR algorithm after five iterations is ES_f we are able to extrapolate the size of the uniform mesh using the following equation

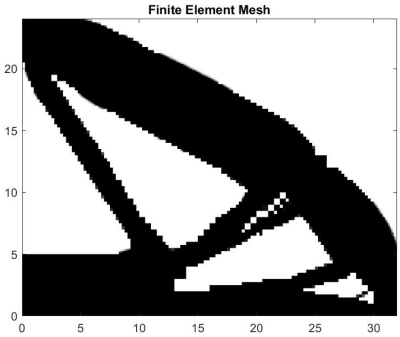
$$N = \sqrt{\frac{1}{ES_f}} \tag{4.1}$$

where N represents the factor by which we need to multiply the amount of original elements along x ($nelx$) and y ($nely$).

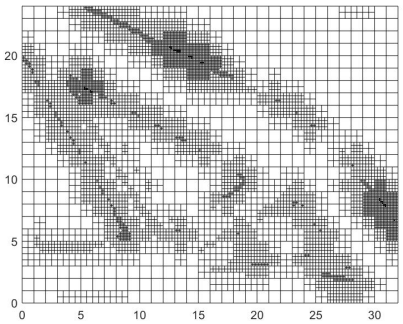
We are now able to compute N and obtain the size of the comparable uniform mesh, so for the original mesh of 32×24 and $ES_f = \frac{1}{256}$ we get $N = 16$ and thus the uniform mesh is 512×384.

Another problem was quickly identified, what filter radius to choose for the uniform mesh. In order to be comparable to the radius used in the last iteration of the AMR algorithm, the filter would need to encompass an equivalent number of elements, but since the AMR algorithm mesh is non-uniform this means that the number of elements affected by the filter is constantly changing. The solution thought to be the fairest is to use the maximum number of elements affected by the filter in the last iteration of the AMR algotirhm, and use a filter in the uniform mesh that affects the same (or approximate) number of elements. For the MBB-beam problem the final filter radius is $r = 0.1875$, which equates to a filter radius in the uniform mesh of $r = 3$.

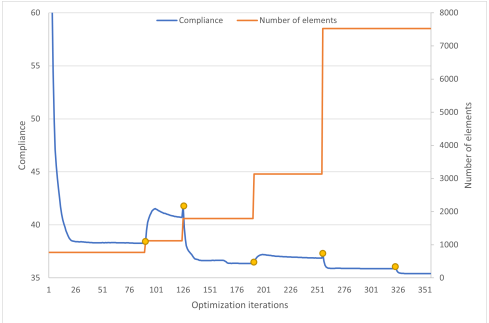
Now that all parameters have been established, and for easier comparison, the results for the MBB-beam, even though they've been previously shown, are reproduced again, along with the uniform mesh optimized design, in figure 4.30.



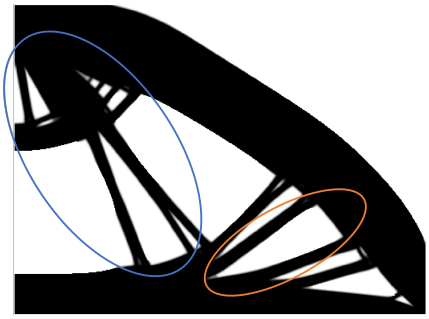
(a) Topology optimized design using AMR ($C = 35.4125$)



(b) Resulting mesh for AMR



(c) Compliance and number of elements over time (final number of elements, $N_{ele} = 7521$), the yellow dots represents the start of a new cycle



(d) Topology optimized design using uniform mesh ($C = 44.9256$), with details

Figure 4.30: Topology optimized design of a MBB-beam with AMR and resulting mesh, for a refinement criterion of intermediate' densities, along with the compliance and number of iterations over time. The optimized design for a comparable uniform mesh is also shown.

Firstly, and for the uniform mesh, these results demonstrate a certain level of mesh-dependency, since the design is not just a better description of the design found at coarser meshes (see figure 4.2(a),

where the design for the starting conditions on a coarse uniform mesh are shown), but also a different design with 'wire-like' structures. Since, in theory, this mesh should be comparable with the AMR mesh, this could be evidence that the filtering radius on the AMR mesh is also insufficient for some regions. The results between algorithms show similarities, on the lower right corner of the designs there is the formation of a 'beam' that is present in both, the middle 'beam' also splits in two in both designs and the bottom decreases its height in both. The main difference is on the top left corner and along the small 'beam' that originates in it (see detail in blue in figure 4.30 (d)), on the AMR algorithm the 'beam' remains single and the corner doesn't bulge out. This might be because of the phenomena previously described that an AMR algorithm that follows the optimize $\rightarrow$refine strategy can 'lock' in local minima from the first iterations. The space in the detail in orange, on the same figure, is also present in the design from the AMR algorithm. When looking at the behaviour of the mesh (see figure 4.30(b)), as was previously seen, the AMR algorithm produced a good mesh following the boundary of the material. Analyzing now the compliance over time, see that it decreased (see figure 4.30(c)) with the iterations, sometimes increasing as the cycles progress, this is because of refinement creating elements that are initially 'populated' with intermediate densities, being especially present from the 1st to the 2nd iteration. The number of elements over time, as it should, increased over time, except for the last cycle, as this was exclusive for decreasing the filter radius. Over time the number of elements increased from the initial 768 to the final 7521, but significantly less when compared to the uniform mesh's 196608. Overall, the results from the AMR algorithm were good with compliance decreasing from $C = 44.9256$ to $C = 35.4125$, for the uniform mesh and AMR algorithm, respectively.

4.2.2 Cantilever beam problem

The cantilever beam problem can be described with the design domain shown in figure 4.31. To implement it, the boundary conditions and fixed degrees of freedom variables need to be updated to the following:

```
F(2*(nelx+1)*(nely+1),1)= -1;
```

and

```
fixeddofs = [1:2*(nelx+1)];
```

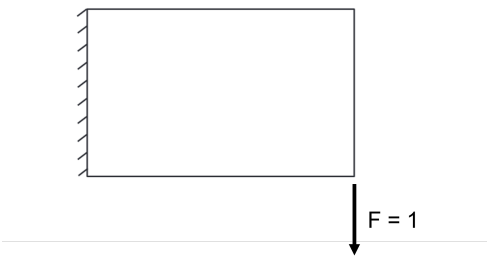


Figure 4.31: Design domain for the cantilever beam problem.

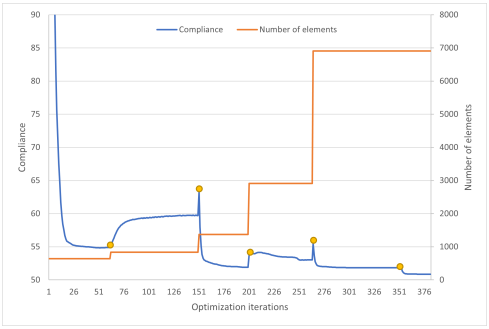
To solve this problem the following parameters were used, penalization power of $p = 3$, volume

fraction of $f = 0.4$, sensitivity filter with radius of $r = 1.2$ and an original mesh of 32×20. These were the parameters used by Sigmund [11] and was decided to maintain them. The AMR method chosen was the refinement of intermediate densities ($crt\_low = 0.2$ and $crt\_high = 0.8$). Just like before we make use of equation 4.1 to compute the comparable uniform mesh, so knowing that the original mesh is 32×20 and $ES_f = \frac{1}{256}$ we get $N = 16$ and thus the uniform mesh is 512×320. To select the radius we repeat the process described for the previous problem (see section 4.2.1), for a final filter radius of $r = 0.15$ the uniform mesh filter radius is $r = 2.4$. The results are shown in figure 4.32. Just like before, the results show some mesh-dependency, which can, by now, be classified as a shortcoming of the AMR algorithm as is implemented. Starting by mentioning the final mesh of the AMR algorithm (see figure 4.32(b)), we see a good behavior of the refinement criterion, as was expected, with large areas of unrefined mesh. Focusing now on the differences between designs (see figure 4.32 (a) and 4.33 (b)), this time some differences appear in the shape of the designs, with the uniform mesh design being more angular in shape, but still very similar to what we see in the AMR design. There are some other differences, as highlighted by the details in the plot. The blue one, less significant, shows a larger hole than the one present in the AMR design, at the right of the hole the lines split and causes another opening that is not present in the AMR design, this is no longer the case for the opening on the left, that is present in both. The opening at the left of the blue highlight and beneath it are also present, but with a smaller size. The highlight in orange shows a bigger difference, as the AMR design shows it but significantly smaller. Once again, the final compliance achieved by the AMR algorithm ($C = 50.8624$) is smaller than the one achieved in the uniform mesh ($C = 53.3906$), and the compliance over time behaved as expected for this design (see figure 4.33 (a)). Overall, the designs are comparable and the AMR algorithm produced a good result, that resembles the design from the uniform mesh, using significantly less elements (6907 and 163840, respectively).

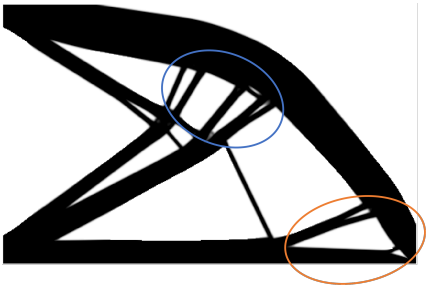
(a) Topology optimized design using AMR ($C = 50.8624$)

(b) Resulting mesh for AMR

Figure 4.32: Topology optimized design of a cantilever beam with AMR and resulting mesh, for a refinement criterion of intermediate' densities, along with the compliance and number of iterations over time. The optimized design for a comparable uniform mesh is also shown.



(a) Compliance and number of elements over time (final number of elements, $N_{ele} = 6907$), the yellow dots signify the start of a new cycle



(b) Topology optimized design using uniform mesh ($C = 53.3906$), with details

Figure 4.33: Topology optimized design of a cantilever beam with for a comparable uniform mesh along with compliance over time and number of elements for the AMR algorithm.

4.2.3 Stocky cantilever beam problem

The Stocky cantilever beam problem can be described with the design domain shown in figure 4.34. To implement it, the boundary conditions and fixed degrees of freedom variables need to be updated to the following:

```
F(((2*(nelx+1)*(nely+1))-(nely+1))+1,1)= -1;
```

and

```
fixeddofs = [1:2*(nelx+1)];
```

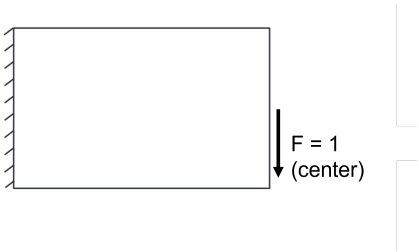
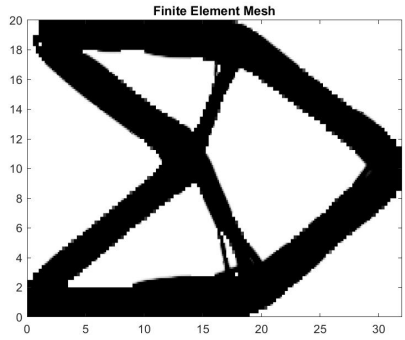


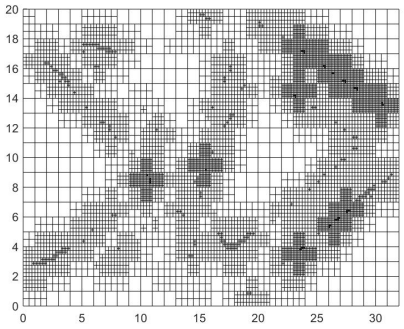
Figure 4.34: Design domain for the stocky cantilever beam problem.

For the problem above the following parameters used were, penalization power of $p = 3$, volume fraction of $f = 0.4$, sensitivity filter with radius of $r = 1.2$ and an original mesh of 32×20. Once again, and since this is a particular cantilever beam, the parameters were selected to mimic the ones used by Sigmund [11]. The AMR method chosen was the refinement of intermediate densities ($crt\_low = 0.2$ and $crt\_high = 0.8$). As before, using equation 4.1 and for an $ES_f = \frac{1}{256}$ we get $N = 16$, so the uniform mesh is 512×320, and knowing that in the last iteration filter radius is $r = 0.15$, the radius for the uniform mesh is $r = 2.4$ (see section 4.2.1 for a detailed explanation). The results for the stocky cantilever beam are shown in figure 4.35. The results diverge from what was previously seen, just like before the uniform mesh result shows signs of mesh-dependency, but this is no longer the case with the AMR algorithm, as the final design is very similar to the design of the first iteration (except for two small holes that appear), on a coarse mesh, just better defined (see figure 4.36). Considering what was previously seen this

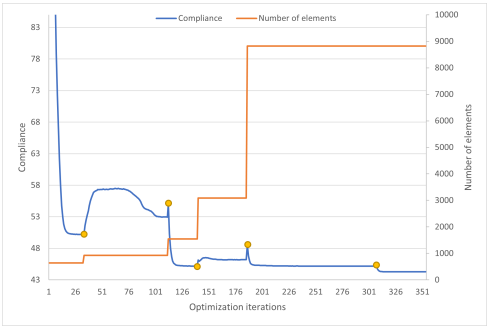
behaviour was unexpected, and can be, perhaps, explained by a stronger performance of the filtering technique for this particular radius and design domain. Even so, the AMR algorithm produced a good mesh (see figure 4.35(b)). As already mentioned, the designs differ and this can be seen in figure 4.35(a) and 4.35(b), the 'beams' in the inner part of the design are split on the uniform mesh and this is not the case on the AMR design, with the common part being the outer shape and location of the larger holes. The AMR design produced is of lower compliance, with $C = 44.2898$ when compared to the uniform mesh's compliance of $C = 46.4406$, and this is, once again, a positive for the AMR design, this was achieved with less elements (8818 and 163840, respectively).



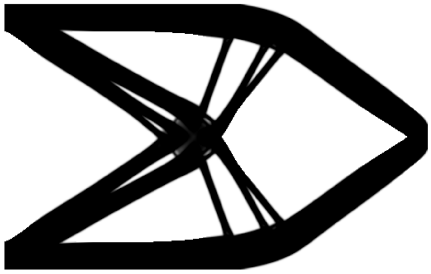
(a) Topology optimized design using AMR ($C = 44.2898$)



(b) Resulting mesh for AMR



(c) Compliance and number of elements over time (final number of elements, $N_{ele} = 8818$), the yellow dots signify the start of a new cycle



(d) Topology optimized design using uniform mesh ($C = 46.4406$), with details

Figure 4.35: Topology optimized design of a stocky cantilever beam with AMR and resulting mesh, for a refinement criterion of intermediate' densities, along with the compliance and number of iterations over time. The optimized design for a comparable uniform mesh is also shown.

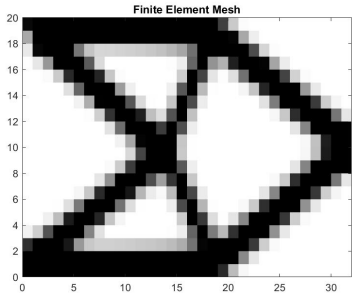


Figure 4.36: Topology optimized design of a stocky cantilever beam in a coarse mesh, corresponding to the first design iteration of the AMR algorithm.

4.2.4 'Wheel' problem

The 'wheel' problem can be described with the design domain shown in figure 4.37. To implement it, the boundary conditions and fixed degrees of freedom variables need to be updated to the following:

```
F((2*(nely+1))+(nelx/2)*(2*(nely+1)))= -1;
```

and

```
fixeddofs = union([2*(nely+1)-1, 2*(nely+1)],[2*(nelx+1)*(nely+1)])
```

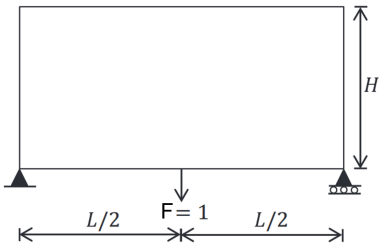
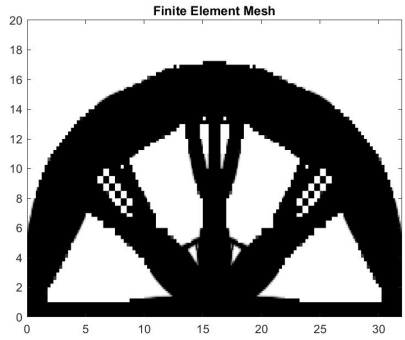


Figure 4.37: Design domain for the 'wheel' problem. Adapted from [45].

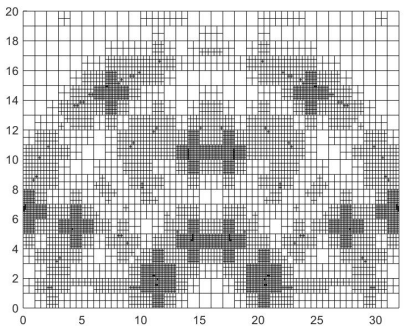
To solve this problem the following parameters used were, penalization $p = 3$, volume fraction of $f = 0.5$, sensitivity filter with radius of $r = 1.2$ and an original mesh of 32×20. The AMR method chosen was the refinement of intermediate densities ($crt\_low = 0.2$ and $crt\_high = 0.8$). Once more, using using equation 4.1 and for $ES_f = \frac{1}{256}$ we get $N = 16$, so the uniform mesh is 512×320, and for a final filter radius of $r = 0.15$ the uniform mesh filter radius is $r = 2.4$ (see section 4.2.1 for a detailed explanation).

The results can be seen in figure 4.38. We see mesh-dependency of the design on a uniform mesh (see 4.38(d)), where as the design obtain using the AMR algorithm (see figure 4.38(a)) is much closer to the expected design, still, the 'wheel spokes' split near the end, which is a sign that the filter did an insufficient work. The resulting mesh of the AMR algorithm shows, again, a good refinement of the intermediate densities, this time with less unrefined areas, this is because of the initial optimized design on a coarse mesh (see figure 4.39), that has a significant number of elements with intermediate

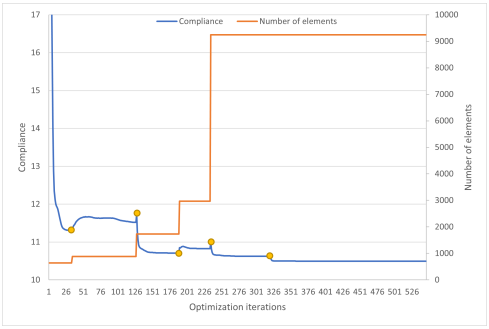
densities. The differences in designs are more noticeable due to the mesh dependency of the uniform mesh design, but the AMR design still captures the 'wheel spokes' that are splitting along the middle. With that said, the AMR design is better, retaining the shape seen in a coarser mesh (see figure 4.39) and improving the definition of the boundary. The AMR design final compliance is also lower than that on the uniform mesh, with $C = 10.4921$ and $C = 14.9795$, respectively.



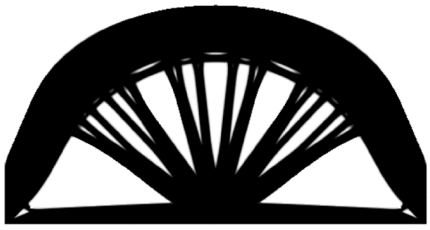
(a) Topology optimized design using AMR ($C = 10.4921$)



(b) Resulting mesh for AMR



(c) Compliance and number of elements over time (final number of elements, $N_{ele} = 9244$), the yellow dots signify the start of a new cycle



(d) Topology optimized design using uniform mesh ($C = 14.9795$), with details

Figure 4.38: Topology optimized design of a 'wheel' with AMR and resulting mesh, for a refinement criterion of intermediate' densities, along with the compliance and number of iterations over time. The optimized design for a comparable uniform mesh is also shown.

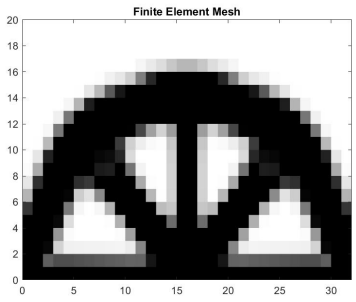


Figure 4.39: Topology optimized design of a 'wheel' in a coarse mesh, corresponding to the first design iteration of the AMR algorithm.

4.2.5 Performance analysis

To analyze the performance of the AMR algorithm the problems solved in section 4.2 were used. The performance was measured using a built-in function of MATLAB (`cputime`) that quantifies the total CPU time used by MATLAB in seconds. Using this function three times were measured, the time per optimization iteration and the time required to run the entire program for each test case, on both algorithms, and the third, the time required to refine 1000 elements on the AMR algorithm. It's important to note that this function doesn't return the actual time that the program takes to run, instead, it returns the time spent by MATLAB utilizing the CPU. This is a more fair comparison, since it focuses on time computing and doesn't favor parallel processing. It's also worth noting that the code used for comparison is the '88 lines code'[10]. Lastly, it is also of relevance to mention that the code ran on MATLAB R2020b on an AMD 3900X Processor with 128GB of RAM.

The results are shown in table 4.1. The result for the optimization iteration is averaged between all iterations (just for the last cycle, in the case of the AMR algorithm). The time comparisons show that the optimization cycles are similar or shorter in the uniform mesh program, with the ones that are similar coinciding with the two designs with less total elements (MBB-beam and Cantilever beam), even though not the desired outcome, this is because of the highly optimized code used for comparison. The developed code takes use of for loops for the computing the stiffness matrix and filtering, which adds extra computing time to the optimization iteration. When comparing to the original '99 line code'[11], that took a `cputime = 25 min` per optimization iteration (for the MBB-beam until canceling), one can see the potential gains to be had using the AMR algorithm. More interesting, is the fact that the whole program, except for the 'wheel' problem, is always faster than that of the uniform mesh. This could be because of two things, first, the AMR algorithm required less iterations overall to solve the same problem, leading to less computing time, and second, due to the nature of the function that measures time. Since parallel computing, using multiple cores, leads to less wall-clock time but is still counted as computing time, the highly efficient code is still requiring expensive computing time, which means that discounting the inefficiencies due to the programming nature of the developed code, the AMR algorithm requires less computational power to solve the same problem.

Table 4.1: Table of performance comparison between the AMR algorithm and the '88 lines code' [10]. Note: for the AMR algorithm the optimization iteration shows two times, corresponding to the time without and with the last filtering stage without refinement.

	AMR Algorithm		'88 lines code'	
	Optimization iteration (s)	Whole program (min)	Optimization iteration (s)	Whole program (min)
MBB-beam	7.6 - 9.48	22.45	9.78	71.59
Cantilever beam	8.86 - 9.64	31.35	8.03	65.37
Stocky cantilever beam	11.26 - 15.60	42.99	7.8	53.34
'Wheel'	16.44 - 17.13	95.61	7.98	50.07

Lastly, a brief mention to the performance of the refinement portion of the code. The time required per element varied based on the problem being solved, as seen on table 4.2, and overall the time seems low at an average of ≈ 0.0324 seconds per element refined, yet, when considering the size of a large mesh, this is a considerable amount of time. It's important to note, though, that the initial mesh is computed on an optimized code, meaning that a larger number of initial elements would bring no added computational time when compared to the '88 lines code', and only subsequent meshes would be computed using the refinement cycle.

Table 4.2: Table of performance for the refinement cycle of the developed algorithm. Note: the time was measured in the last refinement iteration and is in seconds per element refined.

	Refinement cycle
MBB-beam	0.0595
Cantilever beam	0.0252
Stocky cantilever beam	0.0225
'Wheel'	0.0223

Chapter 5

Conclusions

This chapter has the purpose of presenting the conclusions that can be drawn from this dissertation. It serves as a reflection on the work done as well as identified shortcomings and potential future work.

5.1 Achievements

The main purpose of this dissertation was to build an easy to use and modify, MATLAB code for topology optimization using adaptive finite elements, which has been successful.

Nevertheless, a number of shortcomings have been identified that need to be addressed in order to use the algorithm at its full potential.

Based on the results previously presented the main limitation seems to be choosing the most adequate radius for the filtering technique. This problem was not as evident when selecting the refinement of 'solid' densities, but when using sensitivity filtering combined with the refinement of 'intermediate' densities the choice of radius was most important. Achieving a 'perfect' radius, that would prevent checkerboard pattern and mesh-dependency, at every iteration, was deemed unsuccessful. This is especially hard due to the nature of a non-uniform mesh. Even so, the results, when compared to a comparable uniform mesh, were adequate and achieved lower compliance. The refinement criteria also proved to be effective at selecting the solid material or the design border.

The developed algorithm, although less optimized, showed that for most problems it requires less computational resources. Especially when compared to the original '99 line code' from Sigmund [11], which is a better approximation, due to the use of for loops. This showcases the best scenario for the use of AMR in topology optimization. In insight, the use of a separate FEM library to compute the mesh and subsequent refinements, would have improved the efficiency of the algorithm.

It is also important to acknowledge that the designs obtained, even though crude in some ways, always produced a good description of an optimum design.

5.2 Future Work

To better implement the algorithm and allow it to be used in a larger scale several improvements need to be made.

Already mentioned, is the need to achieve a better filtering of the design, either through the use of a better method for selecting the filter radius, or through the use of a different filtering technique. This was identified as the main shortcoming that needs to be addressed in order to successfully use the algorithm in all situations.

Improving the computational efficiency of the algorithm is also needed. The widespread use of for loops comes at the expense of compute time and the code should be optimized in a latter iteration.

To further improve the algorithm, especially for large scale problems, the implementation of derefinement would allow the algorithm to save further compute time by reducing the elements in void regions that were once material regions.

Lastly, the use of a MATLAB function to calculate distance between all points results in high memory usage, a different method, to allow for larger problems, should be investigated.

Bibliography

[1] S. W. Roper, H. Lee, M. Huh, and I. Y. Kim. Simultaneous isotropic and anisotropic multi-material topology optimization for conceptual-level design of aerospace components. *Structural and Multidisciplinary Optimization*, 64(1):441–456, 2021. doi:10.1007/s00158-021-02893-4.

[2] F. Sousa, F. Lau, and A. Suleman. Topology optimization of a wing structure. *Engineering Optimization 2014*, page 507–512, 2014. doi:10.1201/b17488-91.

[3] P. D. Jensen, F. Wang, I. Dimino, and O. Sigmund. Topology optimization of large-scale 3d morphing wing structures. *Actuators*, 10(9):217, 2021. doi:10.3390/act10090217.

[4] D. J. Munk, D. J. Auld, G. P. Steven, and G. A. Vio. On the benefits of applying topology optimization to structural design of aircraft components. *Structural and Multidisciplinary Optimization*, 60(3): 1245–1266, 2019. doi:10.1007/s00158-019-02250-6.

[5] M. P. Bendsøe and N. Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, 71(2): 197–224, 1988. doi:10.1016/0045-7825(88)90086-2.

[6] M. P. Bendsøe and O. Sigmund. Material interpolation schemes in topology optimization. *Archive of Applied Mechanics (Ingenieur Archiv)*, 69(9-10):635–654, 1999. doi:10.1007/s004190050248.

[7] X. Guo, W. Zhang, and W. Zhong. Doing topology optimization explicitly and geometrically — a new moving morphable components based framework. *Journal of Applied Mechanics*, 81(8), 2014. doi:10.1115/1.4027609.

[8] J. Liu and Y. Ma. A survey of manufacturing oriented topology optimization methods. *Advances in Engineering Software*, 100:161–175, 2016. doi:10.1016/j.advengsoft.2016.07.017.

[9] Y. Liang and G. Cheng. Topology optimization via sequential integer programming and canonical relaxation algorithm. *Computer Methods in Applied Mechanics and Engineering*, 348:64–96, 2019. doi:10.1016/j.cma.2018.10.050.

[10] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, and O. Sigmund. Efficient topology optimization in matlab using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43(1): 1–16, 2011. doi:10.1007/s00158-010-0594-7.

[11] O. Sigmund. A 99 line topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 21(2):120–127, 2001. doi:10.1007/s001580050176.

[12] M. P. Bendsøe and O. Sigmund. *Topology Optimization: Theory, methods and applications*. Springer, 2003. ISBN:978-3-540-42992-1.

[13] P. W. Christensen and A. Klarbring. *An introduction to structural optimization*. Springer, 2009. ISBN:978-1-4020-8665-6.

[14] C. Hvejsel and E. Lund. Material interpolation schemes for unified topology and multi-material optimization. *Structural and Multidisciplinary Optimization*, 43:811–825, 2011. doi:10.1007/s00158-011-0625-z.

[15] C. E. K. Pin and Z. S. H. *An introduction to optimization*. Wiley, 2001. ISBN:0-471-39126-3.

[16] M. Stolpe and K. Svanberg. Modelling topology optimization problems as linear mixed 0-1 programs. *International Journal for Numerical Methods in Engineering*, 57(5):723–739, 2003. doi:10.1002/nme.700.

[17] E. Muñoz and M. Stolpe. Generalized benders' decomposition for topology optimization problems. *Journal of Global Optimization*, 51(1):149–183, 2010. doi:10.1007/s10898-010-9627-4.

[18] G. L. Nemhauser, M. W. Savelsbergh, and G. C. Sigismondi. Minto, a mixed integer optimizer. *Operations Research Letters*, 15(1):47–58, 1994. doi:10.1016/0167-6377(94)90013-2.

[19] C. Cordier, H. Marchand, R. Laundy, and L. A. Wolsey. Bc-opt: A branch-and-cut code for mixed integer programs. *Mathematical Programming*, 86(2):335–353, 1999. doi:10.1007/s101070050092.

[20] M. Stolpe and M. P. Bendsøe. Global optima for the zhou–rozvany problem. *Structural and Multidisciplinary Optimization*, 43(2):151–164, 2010. doi:10.1007/s00158-010-0574-y.

[21] M. P. Bendsøe. Optimal shape design as a material distribution problem. *Structural Optimization*, 1 (4):193–202, 1989. doi:10.1007/bf01650949.

[22] M. Stolpe and K. Svanberg. An alternative interpolation scheme for minimum compliance topology optimization. *Structural and Multidisciplinary Optimization*, 22(2):116–124, 2001. doi:10.1007/s001580100129.

[23] Y. Xie and G. Steven. A simple evolutionary procedure for structural optimization. *Computers & Structures*, 49(5):885–896, 1993. doi:10.1016/0045-7949(93)90035-c.

[24] O. Sigmund. Morphology-based black and white filters for topology optimization. *Structural and Multidisciplinary Optimization*, 33(4-5):401–424, 2007. doi:10.1007/s00158-006-0087-x.

[25] G. Rozvany. Aims, scope, methods, history and unified terminology of computer-aided topology optimization in structural mechanics. *Structural and Multidisciplinary Optimization*, 21(2):90–108, 2001. doi:10.1007/s001580050174.

[26] G. Chiandussi. On the solution of a minimum compliance topology optimisation problem by optimality criteria without a priori volume constraint specification. *Computational Mechanics*, 38 (1):77–99, 2005. doi:10.1007/s00466-005-0722-1.

[27] B. Hassani and E. Hinton. Structural topology optimization using optimality criteria methods. *Homogenization and Structural Topology Optimization*, page 71–101, 1999. doi:10.1007/978-1-4471-0891-7_4.

[28] N. H. Kim, T. Dong, D. Weinberg, and J. Dalidd. Generalized optimality criteria method for topology optimization. *Applied Sciences*, 11(7):3175, 2021. doi:10.3390/app11073175.

[29] 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. doi:10.1007/bf01214002.

[30] M. Sotola, P. Marsalek, D. Rybansky, M. Fusek, and D. Gabriel. Sensitivity analysis of key formulations of topology optimization on an example of cantilever bending beam. *Symmetry*, 13 (4):712, 2021. doi:10.3390/sym13040712.

[31] B. Bourdin. Filters in topology optimization. *International Journal for Numerical Methods in Engineering*, 50(9):2143–2158, 2001. doi:10.1002/nme.116.

[32] Y. Maeda, S. Nishiwaki, K. Izui, M. Yoshimura, K. Matsui, and K. Terada. Structural topology optimization of vibrating structures with specified eigenfrequencies and eigenmode shapes. *International Journal for Numerical Methods in Engineering*, 67(5):597–628, 2006. doi:10.1002/nme.1626.

[33] N. Pedersen. Maximization of eigenvalues using topology optimization. *Structural and Multidisciplinary Optimization*, 20(1):2–11, 2000. doi:10.1007/s001580050130.

[34] A. Ghasemi and A. Elham. A novel topology optimization approach for flow power loss minimization across fin arrays. *Energies*, 13(8):1987, 2020. doi:10.3390/en13081987.

[35] K. Maute and E. Ramm. Adaptive topology optimization. *Structural Optimization*, 10(2):100–112, 1995. doi:10.1007/bf01743537.

[36] R. Stainko. An adaptive multilevel approach to the minimal compliance problem in topology optimization. *Communications in Numerical Methods in Engineering*, 22(2):109–118, 2005. doi:10.1002/cnm.800.

[37] J. C. Costa Jr and M. K. Alves. Layout optimization withh-adaptivity of structures. *International Journal for Numerical Methods in Engineering*, 58(1):83–102, 2003. doi:10.1002/nme.759.

[38] M. Bruggi and M. Verani. A fully adaptive topology optimization algorithm with goal-oriented error control. *Computers and Structures*, 89(15-16):1481–1493, 2011. doi:10.1016/j.compstruc.2011.05.003.

[39] Y. Wang, Z. Kang, and Q. He. Adaptive topology optimization with independent error control for separated displacement and density fields. *Computers and Structures*, 135:50–61, 2014. doi:10.1016/j.compstruc.2014.01.008.

[40] A. B. Lambe and A. Czekanski. Topology optimization using a continuous density field and adaptive mesh refinement. *International Journal for Numerical Methods in Engineering*, 113(3):357–373, 2017. doi:10.1002/nme.5617.

[41] S. Wang, E. de Sturler, and G. H. Paulino. Dynamic adaptive mesh refinement for topology optimization. *ArXiv*, abs/1009.4975, 2010.

[42] S. Zhang, A. L. Gain, and J. A. Norato. Adaptive mesh refinement for topology optimization with discrete geometric components. *Computer Methods in Applied Mechanics and Engineering*, 364: 112930, 2020. doi:10.1016/j.cma.2020.112930.

[43] M. A. Salazar de Troya and D. A. Tortorelli. Adaptive mesh refinement in stress-constrained topology optimization. *Structural and Multidisciplinary Optimization*, 58(6):2369–2386, 2018. doi:10.1007/s00158-018-2084-2.

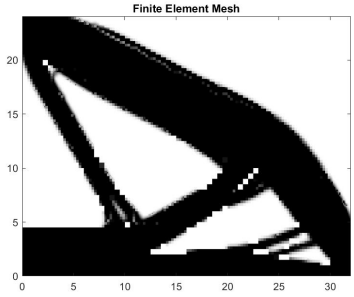
[44] Mane 4240 & civl 4240: Introduction to finite elements – four-noded rectangular element. `https://homepages.rpi.edu/~des/4NodeQuad.pdf`. Accessed: 2021-09-30.

[45] L. Li and K. Khandelwal. Volume preserving projection filters and continuation methods in topology optimization. *Engineering Structures*, 85:144–161, 2015. doi:10.1016/j.engstruct.2014.10.052.

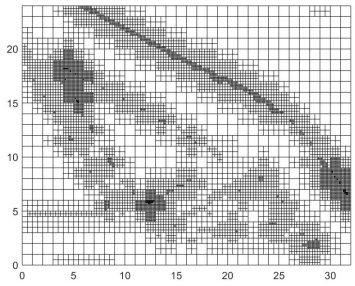
Appendix A

Parametric study

A.1 Results for different refinement criteria

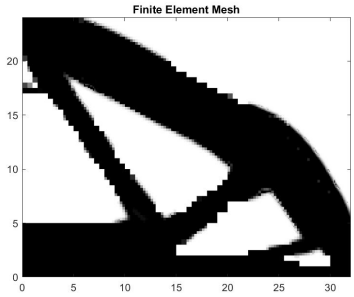


(a) Resulting topology optimized design

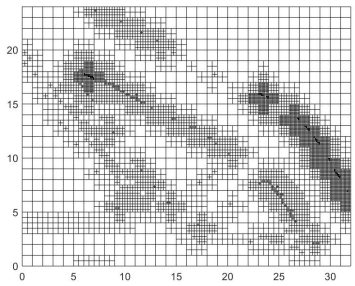


(b) Resulting mesh

Figure A.1: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.2 - 0.9$.

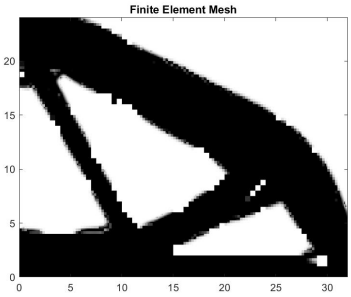


(a) Resulting topology optimized design

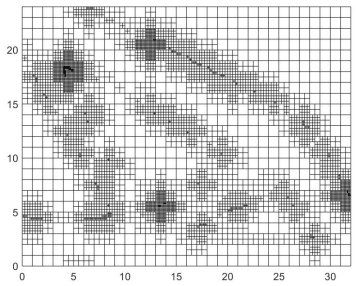


(b) Resulting mesh

Figure A.2: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.3 - 0.7$.

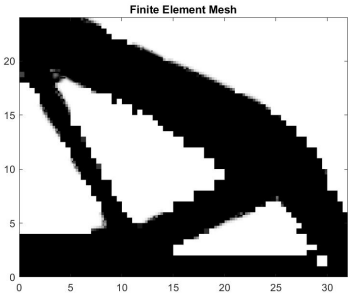


(a) Resulting topology optimized design

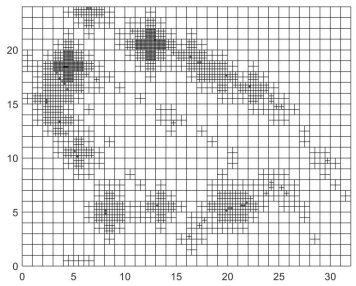


(b) Resulting mesh

Figure A.3: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.4 - 0.8$.

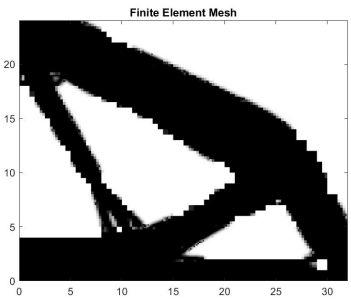


(a) Resulting topology optimized design

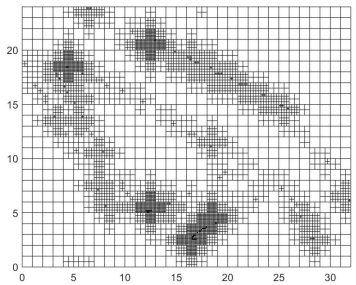


(b) Resulting mesh

Figure A.4: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.5 - 0.8$.



(a) Resulting topology optimized design



(b) Resulting mesh

Figure A.5: Topology optimized design with AMR and resulting mesh, for a refinement criterion of $0.5 - 0.9$.

Appendix B

MATLAB CODE

B.1 Topology Optimization with AMR algorithm

```
function [U,xPhys,area_min] = fem_ite(nelx,nely,rmin,ite_max,filter_type)
%%Starting variables
tStart = cputime;
penal=3;
volfrac=0.5;
div_r = 2; %frequency of filter radius change
crt_low = 0.2;
crt_high = 0.8;
r0 = rmin;
folder = ['Nelx_' num2str(nelx) '_Nely_' num2str(nely) '_r0_' num2str(r0) '_'↙
num2str(crt_low) '_' num2str(crt_high) '_' num2str(penal) '\'];
if exist(folder,'dir')
    folder = [folder(1:end-1) '_new\'];
    mkdir(folder);
elseif ~exist(folder,'dir')
    mkdir(folder);
end
compliance = [];
refinar = [];
% Material properties and element sitffness matrix
E0 = 1.;
Emin = 1e-9;
nu = 0.3;
a = 1;% size of side a of the element
b = 1;% size of side b of the element
E = E0;
KE = [(1 / a * b * E / (1 - nu ^ 2)) / 0.3e1 + (a / b * E / (2 + 2 * nu)) / 0.3e1↙
(nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (1 - nu↙
^ 2)) / 0.3e1 + (a / b * E / (2 + 2 * nu)) / 0.6e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 - (E↙
/ (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (1 - nu ^ 2)) / 0.6e1 - (a / b * E / (2 + 2 *↙
nu)) / 0.6e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 (1 / a * b *↙
E / (1 - nu ^ 2)) / 0.6e1 - (a / b * E / (2 + 2 * nu)) / 0.3e1 -(nu * E / (1 - nu ^ 2))↙
/ 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1; (nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 *↙
nu)) / 0.4e1 (1 / a * b * E / (2 + 2 * nu)) / 0.3e1 + (a / b * E / (1 - nu ^ 2)) / 0.3↙
e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (2 +↙
2 * nu)) / 0.3e1 + (a / b * E / (1 - nu ^ 2)) / 0.6e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1↙
- (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (2 + 2 * nu)) / 0.6e1 - (a / b * E / (1↙
- nu ^ 2)) / 0.6e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 (1 / a↙
* b * E / (2 + 2 * nu)) / 0.6e1 - (a / b * E / (1 - nu ^ 2)) / 0.3e1; -(1 / a * b * E /↙
(1 - nu ^ 2)) / 0.3e1 + (a / b * E / (2 + 2 * nu)) / 0.6e1 -(nu * E / (1 - nu ^ 2)) /↙
0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 (1 / a * b * E / (1 - nu ^ 2)) / 0.3e1 + (a / b * E↙
/ (2 + 2 * nu)) / 0.3e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1↙
(1 / a * b * E / (1 - nu ^ 2)) / 0.6e1 - (a / b * E / (2 + 2 * nu)) / 0.3e1 (nu * E /↙
(1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (1 - nu ^ 2)) /↙
0.6e1 - (a / b * E / (2 + 2 * nu)) / 0.6e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 +↙
2 * nu)) / 0.4e1; (nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 -(1 / a↙
* b * E / (2 + 2 * nu)) / 0.3e1 + (a / b * E / (1 - nu ^ 2)) / 0.6e1 -(nu * E / (1 - nu↙
^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 (1 / a * b * E / (2 + 2 * nu)) / 0.3e1 + (a↙
/ b * E / (1 - nu ^ 2)) / 0.3e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) /↙
0.4e1 (1 / a * b * E / (2 + 2 * nu)) / 0.6e1 - (a / b * E / (1 - nu ^ 2)) / 0.3e1 (nu *↙
E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (2 + 2 * nu))↙
/ 0.6e1 - (a / b * E / (1 - nu ^ 2)) / 0.6e1; -(1 / a * b * E / (1 - nu ^ 2)) / 0.6e1 -↙
(a / b * E / (2 + 2 * nu)) / 0.6e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 *↙
nu)) / 0.4e1 (1 / a * b * E / (1 - nu ^ 2)) / 0.6e1 - (a / b * E / (2 + 2 * nu)) / 0.3↙
e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 (1 / a * b * E / (1 -↙
```

```
nu ^ 2)) / 0.3e1 + (a / b * E / (2 + 2 * nu)) / 0.3e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 +↙
(E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (1 - nu ^ 2)) / 0.3e1 + (a / b * E / (2 +↙
2 * nu)) / 0.6e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1; -(nu * E↙
/ (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (2 + 2 * nu)) /↙
0.6e1 - (a / b * E / (1 - nu ^ 2)) / 0.6e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 +↙
2 * nu)) / 0.4e1 (1 / a * b * E / (2 + 2 * nu)) / 0.6e1 - (a / b * E / (1 - nu ^ 2)) /↙
0.3e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 (1 / a * b * E / (2↙
+ 2 * nu)) / 0.3e1 + (a / b * E / (1 - nu ^ 2)) / 0.3e1 -(nu * E / (1 - nu ^ 2)) / 0.4↙
e1 + (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (2 + 2 * nu)) / 0.3e1 + (a / b * E /↙
(1 - nu ^ 2)) / 0.6e1; (1 / a * b * E / (1 - nu ^ 2)) / 0.6e1 - (a / b * E / (2 + 2 *↙
nu)) / 0.3e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b *↙
E / (1 - nu ^ 2)) / 0.6e1 - (a / b * E / (2 + 2 * nu)) / 0.6e1 (nu * E / (1 - nu ^ 2))↙
/ 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (1 - nu ^ 2)) / 0.3e1 + (a / b *↙
E / (2 + 2 * nu)) / 0.6e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1↙
(1 / a * b * E / (1 - nu ^ 2)) / 0.3e1 + (a / b * E / (2 + 2 * nu)) / 0.3e1 -(nu * E /↙
(1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1; -(nu * E / (1 - nu ^ 2)) / 0.4e1 +↙
(E / (2 + 2 * nu)) / 0.4e1 (1 / a * b * E / (2 + 2 * nu)) / 0.6e1 - (a / b * E / (1 -↙
nu ^ 2)) / 0.3e1 (nu * E / (1 - nu ^ 2)) / 0.4e1 + (E / (2 + 2 * nu)) / 0.4e1 -(1 / a *↙
b * E / (2 + 2 * nu)) / 0.6e1 - (a / b * E / (1 - nu ^ 2)) / 0.6e1 (nu * E / (1 - nu ^↙
2)) / 0.4e1 - (E / (2 + 2 * nu)) / 0.4e1 -(1 / a * b * E / (2 + 2 * nu)) / 0.3e1 + (a /↙
b * E / (1 - nu ^ 2)) / 0.6e1 -(nu * E / (1 - nu ^ 2)) / 0.4e1 - (E / (2 + 2 * nu)) /↙
0.4e1 (1 / a * b * E / (2 + 2 * nu)) / 0.3e1 + (a / b * E / (1 - nu ^ 2)) / 0.3e1;];
 27 %Creates original dof connectivity mtrix
 28 nodenrs = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx);
 29 edofVec = reshape(2*nodenrs(1:end-1,1:end-1)+1,nelx*nely,1);
 30 edofMat = repmat(edofVec,1,8)+repmat([0 1 2*nely+[2 3 0 1] -2 -1],nelx*nely,1);
 31 %creates a matrix with the format [#element node numbers]
 32 node_matrix = zeros(nelx*nely,5);
 33 area = zeros(nelx*nely,2);
 34 i=1;
 35 for k = 1:nelx
 36     for h = 1:nely
 37         node_matrix(i,:) = [i reshape(nodenrs(h:h+1,k:k+1).',1,[])];
 38         area(i,:) = [i 1];
 39         i=i+1;
 40     end
 41 end
 42 %creates initial coordinate matrix
 43 l = linspace(0,nelx,nelx+1) ;
 44 b = linspace(0,nely,nely+1) ;
 45 [X,Y] = meshgrid(l,b) ;
 46 Y = flip(Y);
 47 coordinates=[];
 48 for i=1:(nelx+1)*(nely+1)
 49 coordinates = [coordinates; [i X(i) Y(i)]];
 50 end
 51 %creates initial design domain matrix (based on rectangular shape), nodes
 52 %at boundary are identified with number "1"
 53 fronteira = zeros((nelx+1)*(nely+1),2);
 54 fronteira(:,1)=1:1:length(fronteira);
 55 fronteira(1:nely+1,2)=1;
 56 fronteira(1:nely+1:length(fronteira),2)=1;
 57 fronteira(nely+1:nely+1:length(fronteira),2)=1;
 58 fronteira(length(fronteira)-(nely+1):length(fronteira),2)=1;
 59 node_restrictions = [];
```

```
edofMat_original = edofMat;
edofMat = [[1:size(edofMat,1)]' edofMat];
registry = [];
run = 1;
rmin_i = 0;
while true
    %% Refinement Section
    %Start of the Refinement Cycle
    if run == ite_max
        break
    end
    if isempty(refinar) == 0
        p=1;
        %edofMat
        while (p < length(refinar)+1)
            [~,m]=ismember(refinar(p),node_matrix(:,1)); %searches for the element to refine in the "ledger"
            node_eletoref = node_matrix(m,2:end); %retrieves element to refine
            %If statement that searches for the repetitions of node numbers
            if length(node_matrix) > nelx*nely
                buffer = node_matrix((nelx*nely)+1:end,2:end);
                max_node = max(buffer(:));
                done_left = []; %Search for the left node
                for u = 1:length(buffer)-1
                    buffer_i = buffer(u:u+1,:);
                    if (buffer_i(3) == node_eletoref(1) && buffer_i(8) == node_eletoref(3) && buffer_i(4) == buffer_i(7))
                        new_left = buffer_i(4);
                        done_left = true;
                    elseif (buffer_i(1) == node_eletoref(1) && buffer_i(6) == node_eletoref(3) &&  buffer_i(2) == buffer_i(5))
                        new_left = buffer_i(2);
                        done_left = true;
                    end
                end
                if isempty(done_left) == 1
                    new_left = max_node+1;
                    max_node = new_left;
                end
                done_top = []; %Search for the top node
                for u = 1:length(buffer)-2
                    buffer_i = buffer(u:u+2,:);
                    if (buffer_i(1) == node_eletoref(1) && buffer_i(6) == node_eletoref(2) && buffer_i(3) == buffer_i(4))
                        new_top = buffer_i(3);
                        done_top = true;
                    elseif (buffer_i(7) == node_eletoref(1) && buffer_i(12) == node_eletoref(2) && buffer_i(9) == buffer_i(10))
                        new_top = buffer_i(9);
                        done_top = true;
                    end
                end
                if isempty(done_top) == 1
                    new_top = max_node+1;
                    max_node = new_top;
```

```
110                 end
111                 new_middle = max_node+1; %The middle node is always new
112                 max_node = new_middle;
113                 done_bottom= []; %Search for the bottom node
114                 for u = 1:length(buffer)-2
115                     buffer_i = buffer(u:u+2,:);
116                     if (buffer_i(1) == node_eletoref(3) && buffer_i(6) ==↙
node_eletoref(4) && buffer_i(3) == buffer_i(4))
117                         new_bottom = buffer_i(3);
118                         done_bottom = true;
119                     elseif (buffer_i(7) == node_eletoref(3) && buffer_i(12) ==↙
node_eletoref(4) && buffer_i(9) == buffer_i(10))
120                         new_bottom = buffer_i(9);
121                         done_bottom = true;
122                     end
123                 end
124                 if isempty(done_bottom) == 1
125                     new_bottom = max_node+1;
126                     max_node = new_bottom;
127                 end
128                 done_right = []; %Search for the right node
129                 for u = 1:length(buffer)-1
130                     buffer_i = buffer(u:u+1,:);
131                     if (buffer_i(3) == node_eletoref(2) && buffer_i(8) ==↙
node_eletoref(4) && buffer_i(4) == buffer_i(7))
132                         new_right = buffer_i(4);
133                         done_right = true;
134                     elseif (buffer_i(1) == node_eletoref(2) && buffer_i(6) ==↙
node_eletoref(4) && buffer_i(2) == buffer_i(5))
135                         new_right = buffer_i(2);
136                         done_right = true;
137                     end
138                 end
139                 if isempty(done_right) == 1
140                     new_right = max_node+1;
141                 end
142                 %Call registry function and add new nodes to the node
143                 %connectivity matrix
144                 registry = reg_func(node_matrix,refinar,registry,p);
145                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 node_matrix↙
(refinar(p),2) new_top new_left new_middle];
146                 registry = reg_func(node_matrix,refinar,registry,p);
147                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 new_left↙
new_middle node_matrix(refinar(p),4) new_bottom];
148                 registry = reg_func(node_matrix,refinar,registry,p);
149                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 new_top↙
node_matrix(refinar(p),3) new_middle new_right];
150                 registry = reg_func(node_matrix,refinar,registry,p);
151                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 new_middle↙
new_right new_bottom node_matrix(refinar(p),5)];
152                 %Add new nodes and its coordinates to the coordinate
153                 %matrix
154                 buff_coord = coordinates(node_matrix(refinar(p),2),:);
155                 dist_left = coordinates(node_matrix(refinar(p),2),3)-coordinates↙
(node_matrix(refinar(p),4),3);
```

```
156                 dist_up = coordinates(node_matrix(refinar(p),3),2)-coordinates↙
(node_matrix(refinar(p),2),2);
157                 dist_right = coordinates(node_matrix(refinar(p),3),3)-coordinates↙
(node_matrix(refinar(p),5),3);
158                 if isempty(find(ismember(coordinates(:,1),new_left)))
159                     coordinates = [coordinates; new_left buff_coord(2) buff_coord↙
(3)-(dist_left/2)];
160                 end
161                 if isempty(find(ismember(coordinates(:,1),new_top)))
162                     coordinates = [coordinates; new_top buff_coord(2)+(dist_up/2)↙
buff_coord(3)];
163                 end
164                 coordinates = [coordinates; new_middle buff_coord(2)+(dist_up/2)↙
buff_coord(3)-(dist_left/2)];
165                 if isempty(find(ismember(coordinates(:,1),new_bottom)))
166                     coordinates = [coordinates; new_bottom buff_coord(2)+↙
(dist_up/2) buff_coord(3)-dist_left];
167                 end
168                 if isempty(find(ismember(coordinates(:,1),new_right)))
169                     coordinates = [coordinates; new_right buff_coord(2)+dist_up↙
buff_coord(3)-(dist_right/2)];
170                 end
171                 %Add new node to the possible constraints matrix with the
172                 %following structure
173                 %node_restrictions = [#node left/bottom constraint right/top↙
constraint]
174                 node_restrictions = [node_restrictions; new_left node_matrix↙
(refinar(p),4) node_matrix(refinar(p),2)];
175                 node_restrictions = [node_restrictions; new_top node_matrix(refinar↙
(p),2) node_matrix(refinar(p),3)];
176                 node_restrictions = [node_restrictions; new_bottom node_matrix↙
(refinar(p),4) node_matrix(refinar(p),5)];
177                 node_restrictions = [node_restrictions; new_right node_matrix↙
(refinar(p),5) node_matrix(refinar(p),3)];
178                 %Update boundary matrix
179                 fronteira_index = find(ismember(node_matrix(end-3:end,2:end),↙
[fronteira(fronteira(:,2)==1)']));
180                 front_buffer = node_matrix(end-3:end,2:end);
181                 if length(fronteira_index) == 3
182                     if isequal(fronteira_index,[1 7 10]')
183                         fronteira = [fronteira;front_buffer(fronteira_index(1)+1)↙
1];
184                         fronteira = [fronteira;front_buffer(fronteira_index(1)+size↙
(front_buffer,1)) 1];
185                     elseif isequal(fronteira_index,[7 10 16]')
186                         fronteira = [fronteira;front_buffer(fronteira_index(3)-size↙
(front_buffer,1)) 1];
187                         fronteira = [fronteira;front_buffer(fronteira_index(3)-1)↙
1];
188                     elseif isequal(fronteira_index,[1 10 16]')
189                         fronteira = [fronteira;front_buffer(fronteira_index(2)-1)↙
1];
190                         fronteira = [fronteira;front_buffer(fronteira_index(2)+size↙
(front_buffer,1)) 1];
191                     elseif isequal(fronteira_index,[1 7 16]')
```

```
192                         fronteira = [fronteira;front_buffer(fronteira_index(2)-size↙
(front_buffer,1)) 1];
193                         fronteira = [fronteira;front_buffer(fronteira_index(2)+1)↙
1];
194                     end
195                 elseif length(fronteira_index) == 2
196                     if front_buffer(fronteira_index(1)+1) == front_buffer↙
(fronteira_index(2)-1)
197                         fronteira = [fronteira;front_buffer(fronteira_index(1)+1)↙
1];
198                     end
199                     if front_buffer(fronteira_index(1)+size(front_buffer,1)) ==↙
front_buffer(fronteira_index(2)-size(front_buffer,1))
200                         fronteira = [fronteira;front_buffer(fronteira_index(1)+size↙
(front_buffer,1)) 1];
201                     end
202                 end
203             else
204                 %It does the same as previously done but, for compatibility
205                 %issues, only for the first element refinement
206                 max_node = max(node_matrix(:));
207                 registry = reg_func(node_matrix,refinar,registry,p);
208                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 node_matrix↙
(refinar(p),2) max_node+2 max_node+1 max_node+3];
209                 registry = reg_func(node_matrix,refinar,registry,p);
210                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 max_node+1↙
max_node+3 node_matrix(refinar(p),4) max_node+4];
211                 registry = reg_func(node_matrix,refinar,registry,p);
212                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 max_node+2↙
node_matrix(refinar(p),3) max_node+3 max_node+5];
213                 registry = reg_func(node_matrix,refinar,registry,p);
214                 node_matrix = [node_matrix; max(node_matrix(:,1))+1 max_node+3↙
max_node+5 max_node+4 node_matrix(refinar(p),5)];
215                 buff_coord = coordinates(node_matrix(refinar(p),2),:);
216                 dist_left = coordinates(node_matrix(refinar(p),2),3)-coordinates↙
(node_matrix(refinar(p),4),3);
217                 dist_up = coordinates(node_matrix(refinar(p),3),2)-coordinates↙
(node_matrix(refinar(p),2),2);
218                 dist_right = coordinates(node_matrix(refinar(p),3),3)-coordinates↙
(node_matrix(refinar(p),5),3);
219                 coordinates = [coordinates; max_node+1 buff_coord(2) buff_coord(3)-↙
(dist_left/2)];
220                 coordinates = [coordinates; max_node+2 buff_coord(2)+(dist_up/2)↙
buff_coord(3)];
221                 coordinates = [coordinates; max_node+3 buff_coord(2)+(dist_up/2)↙
buff_coord(3)-(dist_left/2)];
222                 coordinates = [coordinates; max_node+4 buff_coord(2)+(dist_up/2)↙
buff_coord(3)-dist_left];
223                 coordinates = [coordinates; max_node+5 buff_coord(2)+dist_up↙
buff_coord(3)-(dist_right/2)];
224                 node_restrictions = [node_restrictions; max_node+1 node_matrix↙
(refinar(p),4) node_matrix(refinar(p),2)];
225                 node_restrictions = [node_restrictions; max_node+2 node_matrix↙
(refinar(p),2) node_matrix(refinar(p),3)];
226                 node_restrictions = [node_restrictions; max_node+4 node_matrix↙
```

```
(refinar(p),4) node_matrix(refinar(p),5)];
227                 node_restrictions = [node_restrictions; max_node+5 node_matrix↙
(refinar(p),5) node_matrix(refinar(p),3)];
228                 fronteira_index = find(ismember(node_matrix(end-3:end,2:end),↙
[fronteira(fronteira(:,2)==1)']));
229                 front_buffer = node_matrix(end-3:end,2:end);
230                 if length(fronteira_index) == 3
231                     if isequal(fronteira_index,[1 7 10]')
232                         fronteira = [fronteira;front_buffer(fronteira_index(1)+1)↙
1];
233                         fronteira = [fronteira;front_buffer(fronteira_index(1)+size↙
(front_buffer,1)) 1];
234                     elseif isequal(fronteira_index,[7 10 16]')
235                         fronteira = [fronteira;front_buffer(fronteira_index(3)-size↙
(front_buffer,1)) 1];
236                         fronteira = [fronteira;front_buffer(fronteira_index(3)-1)↙
1];
237                     elseif isequal(fronteira_index,[1 10 16]')
238                         fronteira = [fronteira;front_buffer(fronteira_index(2)-1)↙
1];
239                         fronteira = [fronteira;front_buffer(fronteira_index(2)+size↙
(front_buffer,1)) 1];
240                     elseif isequal(fronteira_index,[1 7 16]')
241                         fronteira = [fronteira;front_buffer(fronteira_index(2)-size↙
(front_buffer,1)) 1];
242                         fronteira = [fronteira;front_buffer(fronteira_index(2)+1)↙
1];
243                     end
244                 elseif length(fronteira_index) == 2
245                     if front_buffer(fronteira_index(1)+1) == front_buffer↙
(fronteira_index(2)-1)
246                         fronteira = [fronteira;front_buffer(fronteira_index(1)+1)↙
1];
247                     end
248                     if front_buffer(fronteira_index(1)+size(front_buffer,1)) ==↙
front_buffer(fronteira_index(2)-size(front_buffer,1))
249                         fronteira = [fronteira;front_buffer(fronteira_index(1)+size↙
(front_buffer,1)) 1];
250                     end
251                 end
252             end
253             %Updates the dof connectivity matrix for the new nodes
254             edofMat(size(edofMat,1)+1,:) = [max(edofMat(:,1))+1 node_matrix(end-↙
3,4)*2-1 node_matrix(end-3,4)*2 node_matrix(end-3,5)*2-1 node_matrix(end-3,5)*2↙
node_matrix(end-3,3)*2-1 node_matrix(end-3,3)*2 node_eletoref(1)*2-1 node_eletoref(1)↙
*2];
255             edofMat(size(edofMat,1)+1,:) = [max(edofMat(:,1))+1 node_eletoref(3)*2-↙
1 node_eletoref(3)*2 node_matrix(end-2,5)*2-1 node_matrix(end-2,5)*2 node_matrix(end-↙
2,3)*2-1 node_matrix(end-2,3)*2 node_matrix(end-2,2)*2-1 node_matrix(end-2,2)*2];
256             edofMat(size(edofMat,1)+1,:) = [max(edofMat(:,1))+1 node_matrix(end-↙
1,4)*2-1 node_matrix(end-1,4)*2 node_matrix(end-1,5)*2-1 node_matrix(end-1,5)*2↙
node_eletoref(2)*2-1 node_eletoref(2)*2 node_matrix(end-1,2)*2-1 node_matrix(end-1,2)↙
*2];
257             edofMat(size(edofMat,1)+1,:) = [max(edofMat(:,1))+1 node_matrix(end,4)↙
*2-1 node_matrix(end,4)*2 node_eletoref(4)*2-1 node_eletoref(4)*2 node_matrix(end,3)*2-↙
```

```
1 node_matrix(end,3)*2 node_matrix(end,2)*2-1 node_matrix(end,2)*2];
258             %Removes the row, of the dof connectivity matrix, corresponding
259             %to the element that was just refined
260             [~,n]=ismember(refinar(p),edofMat(:,1));
261             edofMat(n,:) = [];
262             %updates the area matrix for the new elements and removes the
263             %row corresponding to the refined element
264             new_area = node_matrix(end-3:end,1);
265             for i=1:1:4
266                 area = [area; [new_area(i) area(area(:,1)==refinar(p),2)/4]];
267             end
268             area(area(:,1)==refinar(p),:) = [];
269             %If statement used to check for incompatibilities after the
270             %last element is refined
271             refinar_add = [];
272             if p == length(refinar)
273                 [area1,area2,area3] = unique(area(:,2));
274                 for area_i = 1:length(area1)
275                     if isempty(area1(area1(area_i)/16 == area1)) == 0
276                         ele_high = area(find(area(:,2)==area1(area_i)),1);
277                         ele_low = area(area(:,2)==area1(area1(area_i)/16 == area1),↙
1);
278                         fun = @(registry) registry(1);
279                         firstele = cellfun(fun,registry);
280                         ele_low_pos = find(ismember(firstele,ele_low));
281                         ele_0 = [];
282                         for i_pos=1:length(ele_low_pos)
283                             ele_0 = [ele_0; registry{ele_low_pos(i_pos),1}(length↙
(registry{ele_low_pos(i_pos),1}))];
284                         end
285                         ele_0 = unique(ele_0);
286                         ele_0_h = [];
287                         ele_high_pos = find(ismember(firstele,ele_high));
288                         for i_posh=1:length(ele_high_pos)
289                             ele_0_h = [ele_0_h; registry{ele_high_pos(i_posh),1}↙
(length(registry{ele_high_pos(i_posh),1}))];
290                         end
291                         ele_0_h_u = unique(ele_0_h);
292                         for i_high = 1:length(ele_0_h)
293                             if ismember(ele_0_h(i_high),1:nely:((nely*nelx)-(nely-↙
1))) %upper design boundary
294                                 ele_around_high = [ele_0_h(i_high)+1 ele_0_h↙
(i_high)-nely ele_0_h(i_high)+nely];
295                             elseif ismember(ele_0_h(i_high),nely:nely:(nelx*nely))↙
%lower design boundary
296                                 ele_around_high = [ele_0_h(i_high)-1 ele_0_h↙
(i_high)-nely ele_0_h(i_high)+nely];
297                             elseif ismember(ele_0_h(i_high),1:1:nely) %left design↙
boundary
298                                 ele_around_high = [ele_0_h(i_high)-1 ele_0_h↙
(i_high)+1 ele_0_h(i_high)+nely];
299                             elseif ismember(ele_0_h(i_high),(((nely*nelx)-(nely-↙
1)):1:(nely*nelx))) %right design boundary
300                                 ele_around_high = [ele_0_h(i_high)-1 ele_0_h↙
(i_high)+1 ele_0_h(i_high)-nely];
```

```
301                             else
302                                 ele_around_high = [ele_0_h(i_high)-1 ele_0_h↙
(i_high)+1 ele_0_h(i_high)-nely ele_0_h(i_high)+nely]; %for "middle" elements
303                             end
304                             ele_around_high = ele_around_high(ele_around_high>0);
305                             ele_around_high = unique(ele_around_high);
306                             ele_around_high = [ele_around_high, ele_0_h(i_high)];
307                             ele_around_low = ele_around_high(ismember↙
(ele_around_high,ele_0));
308                             if isempty(ele_around_low) == 0
309                                 refinar_add = [refinar_add, firstele(ele_high_pos↙
(i_high))];
310                             end
311                         end
312                         ele_0_h_original = find(ismember(ele_high,[ele_0-1 ele_0+1↙
ele_0-nely ele_0+nely]));
313                         ele_0_h_original = unique(ele_0_h_original);
314                         for i_original = 1:length(ele_0_h_original)
315                             if ismember(ele_high(ele_0_h_original(i_original)),1:1:↙
nelx*nely)
316                                 refinar_add = [refinar_add, ele_high↙
(ele_0_h_original(i_original))];
317                             end
318                         end
319                     end
320                 end
321                 refinar_add_u = unique(refinar_add);
322                 refinar = [refinar, refinar_add_u];
323             end
324             %Ledger used to assign the density of the original element to
325             %the newly created, corresponding, elements
326             [~,ipm]=ismember(refinar(p),xPhys_ledger(:,1));
327             if isempty(ipm) == 0
328                 xPhys_ledger = [xPhys_ledger; max(xPhys_ledger(:,1))+1 xPhys_ledger↙
(ipm,2)];
329                 xPhys_ledger = [xPhys_ledger; max(xPhys_ledger(:,1))+1 xPhys_ledger↙
(ipm,2)];
330                 xPhys_ledger = [xPhys_ledger; max(xPhys_ledger(:,1))+1 xPhys_ledger↙
(ipm,2)];
331                 xPhys_ledger = [xPhys_ledger; max(xPhys_ledger(:,1))+1 xPhys_ledger↙
(ipm,2)];
332                 xPhys_ledger(ipm,:)=[];
333             else
334                 xPhys_ledger = [xPhys_ledger; node_matrix(end-3:1:end,1) [volfrac;↙
volfrac;volfrac;volfrac]];
335             end
336             p=p+1;
337         end
338         xPhys = xPhys_ledger(:,2)';
339         x=xPhys;
340         %Updates boundary matrix to remove non boundary nodes
341         i_front = fronteira(:,2)==0;
342         fronteira(i_front,:)=[];
343         %Identification of free floating dofs
344         [~,~,ix] = unique(reshape(edofMat(:,2:end),size(edofMat,1)*(size(edofMat,2)↙
```

```
-1),1)');
345         C = accumarray(ix,1);
346         i = 1;
347         i_hanging = [];
348         for i = 1:length(C)
349             if C(i) > 1 & C(i) < 4
350                 i_hanging = [i_hanging i];
351             else
352                 i=i+1;
353             end
354         end
355         n_hanging = i_hanging(rem(i_hanging,2)==0)/2; %Transforms dofs in nodes
356         %Removes the nodes at the boundary, leaving only the "true" free
357         %floating nodes
358         n_hanging(find(ismember(n_hanging,fronteira(:,1)')))=[];
359         %Section corresponding to the aplication of the constraints to the
360         %free floating nodes
361         buffer_edofMat=edofMat(:,2:end);
362         node_dofMat=[];
363         for i=1:length(buffer_edofMat)
364             node_dofMat = [node_dofMat; buffer_edofMat(i,2)/2 buffer_edofMat(i,4)/2↙
buffer_edofMat(i,6)/2 buffer_edofMat(i,8)/2];
365         end
366         node_dofMat = [edofMat(:,1) node_dofMat];
367         edofMat_truque = node_dofMat(:,2:end);
368         for i=1:length(n_hanging)
369             i_n_restrictions = find(ismember(node_restrictions(:,1),n_hanging(i)));
370             [xxx,yyy] = find(ismember(edofMat_truque,n_hanging(i)));
371             local = [xxx yyy];
372             [rest_1_x rest_1_y] = find(ismember(edofMat_truque(local(:,1),:),↙
node_restrictions(i_n_restrictions,2))); %Searches fot the first constraint in the↙
constraint matrix
373             rest_1 = [rest_1_x rest_1_y];
374             [rest_2_x rest_2_y] = find(ismember(edofMat_truque(local(:,1),:),↙
node_restrictions(i_n_restrictions,3))); %Searches fot the second constraint in the↙
constraint matrix
375             rest_2 = [rest_2_x rest_2_y];
376             if  rest_1(1,1) == 1
377                 node_restrictions(i_n_restrictions,:);
378                 edofMat_truque(local(:,1),:);
379                 edofMat_truque(local(1,1),local(1,2)) = node_restrictions↙
(i_n_restrictions,3);
380                 edofMat_truque(local(2,1),local(2,2)) = node_restrictions↙
(i_n_restrictions,2);
381                 edofMat_truque(local(:,1),:);
382             elseif rest_1(1,1) == 2
383                 node_restrictions(i_n_restrictions,:);
384                 edofMat_truque(local(:,1),:);
385                 edofMat_truque(local(2,1),local(2,2)) = node_restrictions↙
(i_n_restrictions,3);
386                 edofMat_truque(local(1,1),local(1,2)) = node_restrictions↙
(i_n_restrictions,2);
387                 edofMat_truque(local(:,1),:);
388             end
389         end
```

```
390         %Restores the dof connectivity matrix with the applied constraints
391         final_dofMat = [];
392         h=1;
393         for k=1:size(edofMat_truque,1)
394             for i = 1:size(edofMat_truque,2)
395                 dofMat(h) = edofMat_truque(k,i)*2-1;
396                 h=h+1;
397                 dofMat(h) = edofMat_truque(k,i)*2;
398                 h=h+1;
399             end
400             final_dofMat = [final_dofMat; dofMat];
401             h=1;
402         end
403         %Cycle to compute the striffness matrix for the elements where
404         %constraints were applied
405         [finx,~]=find(ismember(node_dofMat(:,2:end),n_hanging));
406         ele_KE = unique(finx); %elements with true free floating nodes (where↙
constraints have been applied)
407         KE_e = cell(1,((nelx*nely)-length(refinar)+length(ele_KE)));
408         buffer_node_dofMat = node_dofMat(ele_KE,2:end);
409         for i=1:size(buffer_node_dofMat,1)
410             i_n = find(ismember(buffer_node_dofMat(i,:),n_hanging(:)));
411             hanging_node = [];
412             h=1;
413             for l = 1:length(i_n)
414                 hanging_node(h) = i_n(l)*2-1;
415                 h=h+1;
416                 hanging_node(h) = i_n(l)*2;
417                 h=h+1;
418             end
419             buff_if = buffer_node_dofMat(i,:);
420             find_if = find(ismember(node_restrictions(:,1),buff_if(i_n)));
421             if length(i_n) == 1
422                 sum_n_i = ismember(node_restrictions(find_if(1),2:3),buff_if);
423                 buff_sum_n_i = node_restrictions(find_if(1),2:3);
424                 sum_n_n = buff_sum_n_i(sum_n_i);
425                 sum_n = find(ismember(buff_if,sum_n_n));
426                 sum_node = [sum_n*2-1 sum_n*2];
427             elseif length(i_n) == 2
428                 sum_n_i = node_restrictions(find_if(1),ismember(node_restrictions↙
(find_if(1),:),node_restrictions(find_if(2),:)));
429                 sum_n = find(ismember(buff_if,sum_n_i));
430                 sum_node = [sum_n*2-1 sum_n*2];
431             end
432             %Calls the necessary function
433             KE_e{ele_KE(i)} = KE_i(KE,hanging_node,sum_node);
434         end
435         sK=zeros(64,size(final_dofMat,1));
436         for i = 1:size(final_dofMat,1)
437             if ismember(i,ele_KE)
438                 sK(:,i)= KE_e{i}(:)*(Emin+xPhys(i).^penal*(E0-Emin));
439             else
440                 sK(:,i)= KE(:)*(Emin+xPhys(i).^penal*(E0-Emin));
441             end
442         end
```

```
443         sK = reshape(sK,64*size(final_dofMat,1),1);
444     else
445         %In case there is no refinement the If statement jumps here
446         final_dofMat = edofMat_original;
447         x = kron(volfrac,ones(size(final_dofMat,1),1))';
448         xPhys = x;
449         n_hanging = [];
450         i_front = fronteira(:,2)==0;
451         fronteira(i_front,:)=[];
452         ele_KE = [size(edofMat_original,1)*4];
453         sK = reshape(KE(:)*(Emin+xPhys(:)'.^penal*(E0-Emin)),64*size↙
(edofMat_original,1),1);
454         xPhys_ledger = [node_matrix(:,1),xPhys'];
455     end
456     iK = reshape(kron(final_dofMat,ones(8,1))',64*size(final_dofMat,1),1);
457     jK = reshape(kron(final_dofMat,ones(1,8))',64*size(final_dofMat,1),1);
458     K = sparse(iK,jK,sK);
459     K = (K+K')/2;
460     %Boundary Conditions and new dofs from the new elements
461     F = sparse(2*max(max(node_matrix(:,2:end))),1);
462     F(2,1) = -1;  %mbb
463     U = zeros(2*max(max(node_matrix(:,2:end))),1);
464     fixeddofs = union([1:2:2*(nely+1)],[2*(nelx+1)*(nely+1)]);  %mbb
465     more_dofs = setdiff(node_matrix(nelx*nely+1:end,2:end),n_hanging);
466     extra_dofs = setdiff(more_dofs,node_matrix(1:nelx*nely,2:end));
467     alldofs = [1:2*(nely+1)*(nelx+1)];
468     dof_buffer = [];
469     h=1;
470     for i = 1:length(extra_dofs)
471         dof_buffer(h) = extra_dofs(i)*2-1;
472         h=h+1;
473         dof_buffer(h) = extra_dofs(i)*2;
474         h=h+1;
475     end
476     alldofs = [alldofs dof_buffer];
477     freedofs = setdiff(alldofs,fixeddofs);
478     U(freedofs) = K(freedofs,freedofs)\F(freedofs);
479     %% Optimization
480     optimization = true;
481     while optimization
482     if ismember(rmin_i,[0:div_r:50]) && rmin_i > 1
483         rmin = rmin/2; %Devides the filter radius every "div_r" iteration
484     end
485     loop = 0;
486     change = 1;
487     change_acc = [];
488     max_opt_ite = 500;
489     max_change_acc = 100;
490     change_loop = 0.01; %Optimization loop convergence criteria
491     dist_centroid = [];
492     centroid = [];
493     if isempty(refinar) == 0
494         name = ['Com refinamento' ' e com Raio = ' num2str(rmin)];
495         fig_name = ['IteN_' num2str(run) '_Nelx_' num2str(nelx) '_Nely_' num2str↙
(nely) '_Raio_' num2str(rmin) '_change_' num2str(change_loop)];
```

```
496     else
497         name = ['Sem refinamento com Raio = ' num2str(rmin)];
498         fig_name = ['IteN_' num2str(run) '_Nelx_' num2str(nelx) '_Nely_' num2str↙
(nely) '_Raio_' num2str(rmin) '_change_' num2str(change_loop)];
499     end
500     file_name = ['OptCycle_Nelx_' num2str(nelx) '_Nely_' num2str(nely) '_r0_'↙
num2str(r0) '.txt'];
501     fileiD = fopen([folder file_name],'a');
502     fprintf('\n\n ------ Novo ciclo de otimização ------ Refinement Ite.: %.f \n',↙
run);
503     fprintf(fileiD,'\n\n ------ Novo ciclo de otimização ------ Refinement Ite.: %.↙
f \n',run);
504     while change > change_loop
505         loop = loop +1;
506         c=0.;
507         dc=[];
508         for i = 1:size(final_dofMat,1)
509             if ismember(i,ele_KE)
510                 Ue = U(final_dofMat(i,:));
511                 c = c + xPhys(i)^penal*Ue'*KE_e{i}*Ue;
512                 dc(i,1) = -penal*(E0-Emin)*xPhys(i)^(penal-1)*Ue'*KE_e{i}*Ue;
513             else
514                 Ue = U(final_dofMat(i,:));
515                 c = c + xPhys(i)^penal*Ue'*KE*Ue;
516                 dc(i,1) = -penal*(E0-Emin)*xPhys(i)^(penal-1)*Ue'*KE*Ue;
517             end
518         end
519         if isempty(refinar) == 1
520             node_dofMat(:,1) = node_matrix(:,1);
521             node_dofMat(:,2) = node_matrix(:,4);
522             node_dofMat(:,3) = node_matrix(:,5);
523             node_dofMat(:,4) = node_matrix(:,3);
524             node_dofMat(:,5) = node_matrix(:,2);
525         end
526         dv = area(:,2);
527         %Calls the filtering function
528         if filter_type == 1
529             [dc,dist_centroid,centroid,dv,H,Hs] = filter(node_dofMat,node_matrix,↙
xPhys,coordinates,rmin,dc,refinar,dist_centroid,centroid,dv,filter_type,area);
530         elseif filter_type == 2
531             [dc,dist_centroid,centroid,dv,H,Hs] = filter(node_dofMat,node_matrix,x,↙
coordinates,rmin,dc,refinar,dist_centroid,centroid,dv,filter_type,area);
532         end
533         %Optimality Criteria Update
534         l1 = 0; l2 = 1e9; move = 0.2;
535         while (l2-l1)/(l1+l2) > 1e-3
536             lmid = 0.5*(l2+l1);
537             xnew = max(0.001,max(x'-move,min(1.,min(x'+move,x'.*sqrt((-dc./dv)↙
/lmid)))));
538             if filter_type == 1
539                 xPhys = xnew';
540             elseif filter_type == 2
541                 xPhys = (sum(H(:,:).*xnew(:))'./Hs)';
542             end
543             if sum(xPhys'.*area(:,2)) - volfrac*nelx*nely > 0
```

```
544                 l1 = lmid;
545             else
546                 l2 = lmid;
547             end
548         end
549         change = max(max(abs(xnew-x')));
550         x = xnew';
551         %Print results
552         fprintf(' It.:%5i Obj.:%11.4f Vol.:%7.4f ch.:%7.4f\n',loop,c, ...
553             mean(xPhys(:)),change);
554         fprintf(fileiD,' It.:%5i Obj.:%11.4f Vol.:%7.4f ch.:%7.4f\n',loop,c, ...
555             mean(xPhys(:)),change);
556         compliance = [compliance; c];
557         %Different steps to ensure that its not an infinite loop
558         change_acc = [change_acc, change];
559         if length(change_acc) > max_change_acc && isempty(setdiff(change_acc(end-↙
max_change_acc:end),change_acc(end)))
560             change = change_loop - 0.005;
561             name = [name '_NoConvergence'];
562         end
563         if length(change_acc) > max_opt_ite
564             change = change_loop - 0.005;
565             name = [name '_NoConvergence'];
566         end
567         if isempty(refinar) == 0
568             sK=zeros(64,size(final_dofMat,1));
569             for i = 1:size(final_dofMat,1)
570                 if ismember(i,ele_KE)
571                     sK(:,i)= KE_e{i}(:)*(Emin+xPhys(i).^penal*(E0-Emin));
572                 else
573                     sK(:,i)= KE(:)*(Emin+xPhys(i).^penal*(E0-Emin));
574                 end
575             end
576             sK = reshape(sK,64*size(final_dofMat,1),1);
577         else
578             sK = reshape(KE(:)*(Emin+xPhys(:)'.^penal*(E0-Emin)),64*size(edofMat,↙
1),1);
579         end
580         K = sparse(iK,jK,sK);
581         K = (K+K')/2;
582         U(freedofs) = K(freedofs,freedofs)\F(freedofs);
583     end
584     fclose(fileiD);
585     xPhys_ledger(:,2) = xPhys';
586     dist_centroid = [];
587     centroid = [];
588     %% Final Stage - Plots and File Outputs
589     plot_mesh(coordinates(:,2:3),node_dofMat(:,2:end),xPhys,name,1,nelx,nely);
590     saveas(gcf,[folder 'Malha_' fig_name '.fig'],'fig')
591     saveas(gcf,[folder 'Malha_' fig_name '.jpeg'],'jpeg')
592     plot_mesh(coordinates(:,2:3),node_dofMat(:,2:end),xPhys,name,0,nelx,nely);
593     saveas(gcf,[folder fig_name '.fig'],'fig')
594     saveas(gcf,[folder fig_name '.jpeg'],'jpeg')
595     plot_mesh(coordinates(:,2:3),node_dofMat(:,2:end),xPhys,name,3,nelx,nely);
596     saveas(gcf,[folder 'NoFill_' fig_name '.fig'],'fig')
```

```
597     saveas(gcf,[folder 'NoFill_' fig_name '.jpeg'],'jpeg')
598     comp_name = ['Compliance_Nelx_' num2str(nelx) '_Nely_' num2str(nely) '_r0_'↙
num2str(r0) '.mat'];
599     save([folder comp_name],'compliance')
600     U_name = ['U_Ite_' num2str(run) '_Nelx_' num2str(nelx) '_Nely_' num2str(nely)↙
'_r0_' num2str(r0) '.mat'];
601     save([folder U_name],'U')
602     xPhys_name = ['xPhys_Ite_' num2str(run) '_Nelx_' num2str(nelx) '_Nely_' num2str↙
(nely) '_r0_' num2str(r0) '.mat'];
603     save([folder xPhys_name],'xPhys')
604     node_dofMat_name = ['node_dofMat_Ite_' num2str(run) '_Nelx_' num2str(nelx)↙
'_Nely_' num2str(nely) '_r0_' num2str(r0) '.mat'];
605     save([folder node_dofMat_name],'node_dofMat')
606     coord_name = ['coord_Ite_' num2str(run) '_Nelx_' num2str(nelx) '_Nely_' num2str↙
(nely) '_r0_' num2str(r0) '.mat'];
607     save([folder coord_name],'coordinates')
608     if run == ite_max
609         area_min = min(min(area));
610         tEnd = cputime - tStart;
611         tEnd_min = tEnd/60;
612         disp(tEnd_min)
613         break
614     end
615     rmin_i = rmin_i+1;
616     %Refinement Criteria (input at the beginning of the code)
617     refinar = node_dofMat(find(xPhys>crt_low & xPhys<crt_high),1)';
618     %refinar = node_dofMat(find(abs(xPhys-1)<=0.2),1)';
619     if run < 5
620         optimization = false;
621     end
622     if isempty(refinar)
623         break
624     end
625     run = run+1;
626     end
627 end
628 end
629
630 function plot_mesh(coordinates,nodes,xPhys,name,line,nelx,nely)
631 %%Plot Mesh Function
632 %The line input variable is used to specify if the plot is with or without
633 %lines. line == 0 means it plots without mesh lines
634 nel = length(nodes); % number of elements
635 nnel = size(nodes,2); % number of nodes per element
636 % Initialization of the required matrices
637 X = zeros(nnel,nel) ;
638 Y = zeros(nnel,nel) ;
639 for iel=1:nel
640     nd = nodes(iel,:) ;
641     X(:,iel)=coordinates(nd,1);     % extract x value of the node
642     Y(:,iel)=coordinates(nd,2);     % extract y value of the node
643 end
644 % Plotting the FEM mesh
645 if line == 1
646     fig1 = figure('Name',['Malha_' name]);
```

```
647     figure(fig1)
648     colormap(gray)
649     fill(X,Y,1-xPhys,'LineStyle','-')
650     title('Finite Element Mesh') ;
651     axis ([0 nelx 0 nely])
652 elseif line == 0
653     fig1 = figure('Name',name);
654     figure(fig1)
655     colormap(gray)
656     fill(X,Y,1-xPhys,'LineStyle','none')
657     title('Finite Element Mesh') ;
658     axis ([0 nelx 0 nely])
659 elseif line == 3
660     fig1=figure('Name',['NoFill_' name]);
661     figure(fig1)
662     patch(X,Y,'white');
663     axis ([0 nelx 0 nely])
664 end
665 end
666
667 function [dc,dist_centroid,centroid,dv,H,Hs] = filter(node_dofMat,node_matrix,↙
xPhys,coordinates,rmin,dc,refinar,dist_centroid,centroid,dv,filter_type,area)
668 %%Filter Function
669 if rmin > 0
670     if isempty(refinar) == 0
671         %Computes de Centroid of each element
672         if length(dist_centroid) ~= length(node_dofMat)
673             centroid = zeros(size(node_dofMat,1),3);
674             for i = 1:size(node_dofMat,1)
675                 distance_cent = pdist([coordinates(node_dofMat(i,4),2) coordinates↙
(node_dofMat(i,4),3); ...
676                     coordinates(node_dofMat(i,5),2) coordinates(node_dofMat(i,5),↙
3)]);
677                 distance_cent = distance_cent/2;
678                 centroid(i,:) = [node_dofMat(i,1) coordinates(node_dofMat(i,5),2)↙
+distance_cent coordinates(node_dofMat(i,5),3)-distance_cent];
679             end
680         end
681     else
682         node_dofMat = node_matrix;
683         if length(dist_centroid) ~= length(node_dofMat)
684             centroid = zeros(size(node_dofMat,1),3);
685             for i = 1:size(node_dofMat,1)
686                 distance_cent = pdist([coordinates(node_matrix(i,3),2) ...
687                     coordinates(node_matrix(i,3),3);coordinates(node_matrix(i,2),2)↙
coordinates(node_matrix(i,2),3)]);
688                 distance_cent = distance_cent/2;
689                 centroid(i,:) = [node_matrix(i,1) coordinates(node_matrix(i,2),2)↙
+distance_cent coordinates(node_matrix(i,2),3)-distance_cent];
690             end
691         end
692     end
693     if length(dist_centroid) ~= length(node_dofMat)
694         %Computes de distance between all centroids and presents it in a
695         %more "readable" way
```

```
696         dist_centroid = squareform(pdist(centroid(:,2:3)));
697     end
698     for i = 1:size(node_dofMat,1)
699         %Adjustable filter radius (if user wants to activate it)
700         %
701         %
702         %rmin = 1.5*area(i,2);
703         h_operator_m(:,i) = rmin-dist_centroid(:,i);
704         h_operator_sum_m(:,i) = max(0,h_operator_m(:,i));
705         h_i_m{i} = find(h_operator_sum_m(:,i)>0);
706         sum_h_matrix(i,1) = sum(h_operator_sum_m(:,i),1);
707     end
708     for i = 1:size(node_dofMat,1) %Filtering cycle
709         h_operator_sum = h_operator_sum_m(:,i);
710         h_i =  h_i_m{i};
711         sum_h = sum_h_matrix(i,1);
712         if filter_type == 1
713             dcn(i,1) = sum(h_operator_sum(h_i).*xPhys(h_i)'.*dc(h_i))/(max(1e-3,↙
xPhys(i))*sum_h);
714         elseif filter_type == 2
715             dcn(i,1) = sum(h_operator_sum(h_i).*(dc(h_i)./sum_h_matrix(h_i)));
716             dv(i,1) = sum(h_operator_sum(h_i).*(dv(h_i)./sum_h_matrix(h_i)));
717         end
718     end
719     dc = dcn;
720     H =  h_operator_sum_m;
721     Hs = sum_h_matrix;
722 end
723 end
724
725 function [KE_i] = KE_i(KE,hanging_node,sum_dof)
726 %% Function that outputs the element stiffness matrix for constrained elements
727 % the entries are strcutured as follows:
728 % hanging_node=[1,2,5,6]; #of free floating nodes
729 % sum_node=[3,4]; #of nodes where changes will occur
730 KE_i = KE;
731 KE_i(:,hanging_node(:)) = (KE_i(:,hanging_node(:))/2);
732 KE_i(:,sum_dof(1)) = KE_i(:,sum_dof(1)) + sum(KE_i(:,hanging_node(1:2:end)),2);
733 KE_i(:,sum_dof(2)) = KE_i(:,sum_dof(2)) + sum(KE_i(:,hanging_node(2:2:end)),2);
734 KE_i(hanging_node(:),:) = (KE_i(hanging_node(:),:)/2);
735 KE_i(sum_dof(1),:) = KE_i(sum_dof(1),:) + sum(KE_i(hanging_node(1:2:end),:),1);
736 KE_i(sum_dof(2),:) = KE_i(sum_dof(2),:) + sum(KE_i(hanging_node(2:2:end),:),1);
737 end
738
739 function [registry] = reg_func(node_matrix,refinar,registry,p)
740 %% Fuction used to create a registry of the original element and all subsequent↙
refined elements
741 index_registry = [];
742 for ir = 1:size(registry,1)
743     ir_m = find(ismember(registry{ir,1},refinar(p)));
744     if isempty(ir_m) == 0
745         index_registry = [index_registry; ir ir_m];
746     end
747 end
748 if isempty(index_registry)
```

```
749     registry{size(registry,1)+1,1} = [max(node_matrix(:,1))+1 refinar(p)];
750 elseif index_registry(2) ~= 1
751     registry{size(registry,1)+1,1} = [max(node_matrix(:,1))+1 registry↙
{index_registry(1),1}(2:end)];
752 else
753     registry{index_registry(1),1} = [max(node_matrix(:,1))+1, registry↙
{index_registry(1),1}];
754 end
755 end
756
757
```